

Ver. 3.0.13
2010-04-01

Linux Kernel Internal & Embedded Linux



Version 3.0.12
2002년 11월 12일

낭종호 교수님께 드립니다.

Copyright Notice

본 문서는 Linux에 관심이 있고 자신의 시간과 열정을 나누려는 사람들을 위해서 배포합니다. 따라서, 본 문서는 어떠한 목적의 상업적인 이용을 금하며, 문서를 재배포 및 수정을 원하는 분들은 반드시 저자에게 동의를 구할 것을 부탁드립니다.

저자 : 권수호 (suho.kwon@gmail.com)

1. INTRODUCTION	21
1.1. 목적	21
1.2. 범위	21
1.3. KERNEL이란?	21
1.4. KERNEL의 구성.....	23
1.5. 리눅스 커널의 COMPILE방법	25
1.6. LINUX KERNEL IMAGE만들기의 상세	26
1.7. KERNEL의 특징.....	27
2. PROCESS MANAGEMENT.....	29
2.1. TASK_STRUCT 구조체	29
2.2. PROCESS들의 관계	34
2.3. PROCESS의 상태	36
2.4. PROCESS의 CONTEXT	37
2.5. PROCESS의 MODE	37
2.6. PROCESS의 SCHEDULING.....	38
2.7. LINUX PROCESS SCHEDULING ALGORITHM.....	39
2.8. 기타 SCHEDULING과 관련된 함수들	46
2.9. TIMER INTERRUPT의 처리	52
2.10. THREAD IN LINUX.....	54
2.10.1. Kernel Thread	54
2.10.2. Lightweight Process.....	55
2.10.3. User Thread	56
2.11. SYSTEM CALL.....	58
2.12. INTERRUPT의 처리	59
2.13. 프로세스의 생성	61
2.14. 프로세스의 실행.....	66
2.15. 프로세스의 종료	69
2.16. SELECT AND POLL	70
2.16.1. Poll의 구현	70
2.16.2. Select의 구현	77
2.17. ELF 파일의 형식(FORMAT)	86
2.17.1. ELF 헤더의 정의	88
2.17.2. ELF 포맷의 검출	90
2.17.3. Section	92
2.17.4. 특별한 의미를 가지는 섹션들	97
2.18. PROCESS간 통신(INTERPROCESS COMMUNICATION)	102
2.18.1. Signal	103
2.18.2. Pipe	114
2.18.3. FIFO(First In First Out)	120
2.18.4. Semaphore	124
2.18.5. Message Queue	137
2.18.6. Shared Memory	148
2.19. PROCESS의 MEMORY IMAGE LAYOUT.....	155
2.20. 커널에서 제공하는 기능들	156
2.20.1. Bottom Half	156
2.20.2. Task Queue	158
2.20.3. Timer.....	160
2.20.4. Wait Queue	161
2.21. TASKLET, BOTTOM HALF, SOFTWARE IRQ AND TASK QUEUE	163

2.21.1. Tasklet의 데이터 구조.....	164
2.21.2. Kernel Software IRQ Daemon.....	166
2.21.3. Tasklet의 초기화.....	168
2.21.4. Bottom Half 설정	172
2.21.5. Tasklet의 수행.....	174
2.21.6. Task Queue	176
2.22. LINUX O(1) SCHEDULER.....	181
2.22.1. 기존의 Linux 스케줄링 메커니즘	182
2.22.2. 커널 버전 2.5의 Scheduler의 특징.....	182
2.22.3. Preemptive Kernel의 분석	183
2.22.4. schedule() 함수의 분석	188
3. FILE SYSTEM.....	207
3.1. VFS(VIRTUAL FILE SYSTEM).....	207
3.2. FILE SYSTEM OBJECTS	207
3.2.1. File Object	208
3.2.2. Dentry Object.....	210
3.2.3. Inode Object.....	211
3.2.4. Super Block Object.....	214
3.3. FILE SYSTEM INITIALIZATION.....	216
3.4. EXT2 FILE SYSTEM.....	216
3.4.1. EXT2의 Super 블록	217
3.4.2. EXT2의 Descriptor 테이블 블록	221
3.4.3. EXT2의 Bitmap 블록.....	221
3.4.4. EXT2의 Inode 테이블.....	221
4. MEMORY MANAGEMENT	224
4.1. SEGMENTATION	224
4.2. PRIVILEGE LEVEL.....	231
4.3. PAGING.....	233
4.4. RESERVED MEMORY AREA	238
4.5. 커널에서 사용하는 페이지 테이블.....	239
4.6. 페이지 디렉토리 및 페이지 테이블에 대한 초기화	240
4.7. 페이지에 대한 초기화	246
4.8. 페이지의 할당과 해제	255
4.9. BUDDY 알고리즘	265
4.10. SLAB ALLOCATOR 알고리즘.....	267
4.11. 프로세스의 주소 공간 관리	279
4.12. 페이지 오류(FAULT)의 처리	298
5. NETWORK.....	305
5.1. NETWORK.....	305
5.2. BSD SOCKET LAYER.....	309
5.3. SOCKET의 생성	311
5.4. SOCKET을 이용한 데이터 보내기	318
5.5. INET SOCKET LAYER.....	322
5.6. TCP AND UDP LAYER.....	327
5.7. IP LAYER.....	336
5.8. TCP/UDP LAYER에서 받기	351
5.9. INET SOCKET LAYER에서 받기	355
5.10. BSD SOCKET LAYER에서 받기	356
5.11. ARP LAYER.....	364
5.12. NETWORK DEVICE DRIVER LAYER.....	365
6. LINUX DEVICE DRIVER BASIC	366

6.1.	KERNEL STRUCTURE에 대한 이해.....	367
6.2.	기본 하드웨어의 구조	368
6.2.1.	Bus의 이해	368
6.2.1.1.	ISA Bus	369
6.2.1.2.	Micro Channel Architecture(MCA) Bus.....	370
6.2.1.3.	EISA Bus.....	370
6.2.1.4.	PCI Bus.....	371
6.2.1.5.	PCI Bridge	373
6.2.1.6.	PCI에서 사용하게될 기본 함수들.....	374
6.2.2.	I/O method	378
6.3.	PCI(PERIPHERAL COMPONENT INTERFACE) BUS.....	378
6.4.	DMA(DIRECT MEMORY ACCESS)	383
6.5.	KERNEL MODULE PROGRAMMING.....	384
6.6.	INTERRUPT HANDLER의 설치.....	393
6.7.	IOCTL(I/O CONTROL)	394
6.8.	어떻게 COMPILE할 것인가?	397
6.9.	DEBUGGING METHOD.....	397
6.10.	CHARACTER DEVICE DRIVER	398
6.10.1.	문자 디바이스 드라이버의 예.....	404
6.11.	BLOCK DEVICE DRIVER.....	408
6.11.1.	blk.h 파일	416
6.11.2.	블록 모드 디바이스 드라이버에서 사용하는 전역 변수들	418
6.11.3.	블록 디바이스 드라이버의 예.....	419
6.12.	NETWORK DEVICE DRIVER.....	427
6.12.1.	Ethernet에 대한 이해	428
6.12.2.	Protocol이란?.....	429
6.12.3.	Socket이란? 그리고, socket buffer란?	430
6.12.4.	전체적인 구조	432
6.12.5.	Network의 Setting	433
6.12.6.	네트워크 디바이스 구조체.....	433
6.12.7.	간단한 Network Device Driver의 구현 예	437
6.12.8.	특정 운영 체제에서의 구현	444
7.	REAL-TIME CLOCK(RTC)의 분석	445
7.1.	RTC 하드웨어의 분석	445
7.2.	RTC의 초기화와 해제	449
7.3.	RTC의 OPEN과 RELEASE.....	459
7.4.	RTC의 READ.....	461
7.5.	RTC의 POLL	462
7.6.	RTC의 FASYNC.....	462
7.7.	RTC의 INTERRUPT HANDLER.....	465
7.8.	RTC의 I/O CONTROL	466
7.9.	RTC의 PROC FILE SYSTEM 인터페이스	476
7.10.	RTC를 이용하는 예제 프로그램	478
8.	POWER MANAGEMENT IN LINUX	484
8.1.	POWER MANAGEMENT를 위한 자료구조	484
8.2.	POWER MANAGEMENT의 등록	486
8.3.	POWER MANAGEMENT의 해제.....	487
8.4.	POWER MANAGEMENT의 찾기	488
8.5.	POWER MANAGEMENT의 명령 내리기	489
8.6.	POWER MANAGEMENT IN PCI	491
9.	ADVANCED POWER MANAGEMENT(APM).....	499

9.1.	APM BIOS.....	499
9.2.	BOOTING에서의 APM관련 부분에 대한 분석.....	502
9.3.	APM MODULE의 적재와 해제	508
9.4.	APM BIOS FUNCTION의 종류	521
9.5.	APM SETUP STRING의 해석	523
9.6.	APM BIOS의 파일 연산 벡터(FILE OPERATION VECTOR).....	524
9.6.1.	Open 함수의 분석	525
9.6.2.	Release 함수의 분석	526
9.6.3.	Read 함수의 분석	531
9.6.4.	I/O Control 함수의 분석	533
9.6.5.	Poll 함수의 분석	536
9.7.	SYSTEM IDLE 상태에서의 수행	536
9.8.	APM KERNEL DAEMON	541
10.	USB(UNIVERSAL SERIAL BUS) DEVICE DRIVER.....	550
10.1.	USB HOST CONTROLLER	550
10.2.	USB DEVICE들과 TRANSFER CHARACTERISTICS	552
10.2.1.	USB Hubs	553
10.2.2.	Data Flow.....	553
10.3.	DESCRIPTOR	553
10.4.	WINDOWS DEVICE DESCRIPTORS FOR USB	558
10.5.	DEVICE CLASS	560
10.6.	USB DEVICE DRIVER	561
10.7.	LINUX USB SUBSYSTEM	561
10.8.	USB DEVICE DRIVER의 FRAME WORK	562
10.9.	FRAMEWORK FUNCTION	564
10.10.	CONFIGURING USB DEVICES	564
10.11.	USB DATA TRANSFER.....	567
10.12.	URB(USB REQUEST BLOCK)	571
10.13.	URB와 관련된 함수 및 매크로들	576
10.14.	USB ERROR CODE & STATUS CODE.....	578
10.15.	예제를 통한 USB DEVICE DRIVER의 분석	579
10.15.1.	USB device driver의 자료구조	579
10.15.2.	USB device driver의 초기화 및 제거하기	580
10.15.3.	USB device driver의 해제.....	581
10.15.4.	USB device의 검출과 연결 끊기	582
10.15.5.	Buffer의 할당과 해제	584
10.15.6.	PLUSUSB에 대한 network driver입출력 함수.....	588
10.15.7.	PLUSUSB에 대한 USB 입출력 함수	592
11.	IEEE 1394 FOR LINUX.....	596
11.1.	상위 레벨 드라이버란?.....	597
11.2.	상위 레벨 드라이버의 초기화 설정	598
11.3.	IP OVER 1394의 구현	599
11.3.1.	Driver의 초기화와 삭제	600
11.3.2.	Network 디바이스 드라이버의 초기화	603
11.3.3.	Highlevel Driver Operations.....	606
11.3.3.1.	Add Host	607
11.3.3.2.	Remove Host	607
11.3.3.3.	Reset Host	608
11.3.4.	Highlevel Driver Address Space Operations	609
11.3.5.	Network Interface Operations	609
11.3.5.1.	Open	609
11.3.5.2.	Release	610
11.3.5.3.	Hard Header	610

11.3.5.4.	Rebuild Header.....	611
11.3.5.5.	Receive.....	611
11.3.5.6.	Transmit	614
11.3.5.7.	Config.....	619
11.3.5.8.	Ioctl	620
11.3.5.9.	Statistics	620
11.4.	IEEE 1394 FOR LINUX에서 남은 일	620
12.	PCMCIA FOR LINUX.....	621
12.1.	PCMCIA란?	621
12.2.	CARD SERVICES(CS)	622
12.3.	SOCKET SERVICE(SS).....	622
12.4.	CLIENT DEVICE DRIVERS(CDD)	622
12.5.	LINUX에서의 PCMCIA구현	624
12.5.1.	Socket interface.....	625
12.5.2.	Socket controller	626
12.6.	CARD SERVICES에 대한 인터페이스.....	626
12.7.	CARD SERVICES의 EVENT HANDLING	632
12.8.	DRIVER SERVICE에 대한 인터페이스	635
12.8.1.	Client Driver를 Driver Service에 등록하기	639
12.8.2.	Client Driver를 Driver Service에서 제거하기	640
12.8.3.	CardBus Client의 인터페이스	643
12.8.3.1.	CardBus Driver의 등록	644
12.8.3.2.	CardBus Driver의 해제	645
12.8.3.3.	CardBus Enabler의 Attach() 함수	646
12.8.3.4.	CardBus Enabler의 Detach() 함수	647
12.8.3.5.	CardBus Enabler의 Config() 함수	649
12.8.3.6.	CardBus Enabler의 Release() 함수	651
12.8.3.7.	CardBus Enabler의 Event() 함수	652
12.9.	I/O MAPPING AND MEMORY MAPPING	654
12.10.	CIS(CARD INFORMATION STRUCTURE)	656
12.10.1.	CIS의 구조	656
12.10.2.	What is Tuple?	657
12.10.3.	Configuration Table Entry	658
12.10.4.	Multiple Function in PC Card	659
12.11.	CARD CONFIGURATION REGISTER	660
12.11.1.	Configuration Option Register(COR).....	660
12.11.2.	Card Configuration and Status Register(CCSR).....	661
12.11.3.	Pin Replacement Register(PPR).....	663
12.11.4.	Socket and Copy Register(SCR).....	664
12.11.5.	Extended Status Register(ESR)	664
12.11.6.	I/O Base and Limit Register	665
13.	PCMCIA CLIENT DEVICE DRIVER.....	666
13.1.	DUMMY_CS 모듈의 적재와 해제	666
13.2.	DUMMY_CS의 ATTACH와 DETACH	667
13.3.	DUMMY_CS의 RELEASE	674
13.3.1.	RequestIO & ReleaseIO	675
13.3.2.	RequestIRQ & ReleaseIRQ	677
13.3.3.	RequestConfiguration & ReleaseConfiguration	677
13.3.4.	RequestWindow & ReleaseWindow	677
13.4.	DUMMY_CS의 EVENT처리	679
13.4.1.	CS_EVENT_CARD_REMOVAL & CS_EVENT_CARD_INSERTION event의 처리	680
13.4.2.	CS_EVENT_PM_SUSPEND & CS_EVENT_PM_RESUME event의 처리	680
13.4.3.	CS_EVENT_RESET_PHYSICAL & CS_EVENT_CARD_RESET event의 처리	681
13.5.	DUMMY_CS의 ERROR REPORTING	681

13.6.	DUMMY_CS의 CONFIGURATION.....	682
14.	CARDBUS CLIENT DRIVER의 분석	695
15.	INPUT DEVICE DRIVER IN LINUX.....	696
15.1.	INPUT DEVICE DRIVER	696
15.1.1.	Input Device의 등록.....	696
15.1.2.	Input Device의 해제.....	700
15.1.3.	Input Device Handle의 등록	701
15.1.4.	Input Device Handle의 해제	702
15.1.5.	Input Device의 Handler의 등록.....	702
15.1.6.	Input Device의 Handler의 해제	703
15.1.7.	Input Device의 Event 처리	704
15.1.8.	Input Device Driver의 초기화 및 해제.....	706
15.2.	LINUX USB MOUSE DRIVER.....	708
15.2.1.	USB Mouse Driver의 등록과 해제.....	709
15.2.2.	USB Mouse Driver의 Probing.....	711
15.2.3.	USB Mouse Driver의 Disconnect.....	714
15.2.4.	USB Mouse Driver의 Open과 Close.....	714
15.2.5.	USB Mouse Interrupt Handler	715
15.3.	LINUX USB KEYBOARD DRIVER	715
15.3.1.	USB Keyboard Driver의 등록과 해제	715
15.3.2.	USB Keyboard Driver의 Probe	716
15.3.3.	USB Keyboard Driver의 Disconnect.....	720
15.3.4.	USB Keyboard Driver의 Open과 Close	720
15.3.5.	USB Keyboard Driver의 Event 처리	721
15.3.6.	USB Keyboard Interrupt Handler	722
15.3.7.	USB Keyboard Driver의 LED 처리	723
16.	BOOTING	724
16.1.	BOOTING이란?.....	724
16.2.	BOOTSECT.S의 분석	724
16.3.	SETUP.S의 분석	730
16.4.	HEAD.S의 분석	742
16.5.	KERNEL의 HEAD.S의 분석	744
16.6.	LILO(LINUX LOADER)	753
16.6.1.	Disk의 구성	753
16.6.2.	LILO에서 사용하는 파일	754
16.6.3.	MAP File	755
16.6.4.	파라미터의 설정	759
16.6.5.	외부 인터페이스	760
16.6.6.	bootsect.S의 분석	761
16.6.7.	first.S의 분석	761
16.6.8.	second.S의 분석	764
17.	EMBEDDED LINUX IMPLEMENTATION	765
17.1.	WHAT IS EMBEDDED SYSTEM?.....	765
17.2.	CROSS COMPILER	765
17.2.1.	Binutil Compile.....	765
17.2.2.	Kernel Installation.....	766
17.2.3.	GCC Compile	767
17.2.4.	Glibc Compile.....	769
17.2.5.	GCC re-compile	770
17.2.6.	Reference Web Site	771

17.3.	SA1100의 GPIO(GENERAL PURPOSE INPUT/OUTPUT).....	771
17.3.1.	GPIO Pin-Level Register(GPLR)	772
17.3.2.	GPIO Pin Direction Register(GPDR)	773
17.3.3.	GPIO Pin Output Set Register(GPSR)과 Pin Output Clear Register(GPCR).....	773
17.3.4.	GPIO Rising-Edge Detect Register(GRER)과 Falling-Edge Detect Register(GFER)	774
17.3.5.	GPIO Edge Detect Status Register(GEDR).....	775
17.3.6.	GPIO Alternate Function Register(GAFR).....	775
17.3.7.	GPIO Register Location.....	777
17.4.	SA1100의 INTERRUPT CONTROLLER.....	778
17.4.1.	Interrupt Controller Pending Register(ICPR)	779
17.4.2.	Interrupt Controller IRQ Pending Register(ICIP) and FIQ Pending Register(ICFP)	781
17.4.3.	Interrupt Controller Mask Register(ICMR)	781
17.4.4.	Interurpt Controller Level Register(ICLR)	782
17.4.5.	Interrupt Controller Control Register(ICCR).....	782
17.4.6.	Interurpt Controller Register Location.....	783
17.5.	SA1100의 MEMORY MAP	784
17.6.	BOOT LOADER(BLOB).....	786
17.6.1.	start.S의 분석	788
17.6.2.	main.c의 분석	796
17.6.3.	BLOB의 command 분석	815
17.6.3.1.	Boot Command	816
17.6.3.2.	Clock Command.....	816
17.6.3.3.	Download Command.....	819
17.6.3.4.	Flash Command.....	825
17.6.3.5.	Help Command & Status Command	829
17.6.3.6.	Reblob Command.....	831
17.6.3.7.	Reboot Command.....	832
17.6.3.8.	Reload Command	833
17.6.3.9.	Reset Command	833
17.6.3.10.	Speed Command	833
17.7.	SA1100의 BOOTING.....	835
17.7.1.	head.S의 분석	838
17.7.2.	head-armv.S의 분석	854
17.7.3.	start_kernel() 함수의 분석	869
17.7.3.1.	setup_arch() 함수의 분석	870
17.7.3.2.	paging_init() 함수의 분석	881
17.7.3.3.	trap_init() 함수의 분석	894
17.7.3.4.	init_IRQ() 함수의 분석	897
17.7.3.5.	do_IRQ() 함수의 분석	906
17.7.3.6.	Interrupt Service의 구현	909
17.7.3.7.	sched_init() 함수의 분석	914
17.7.3.8.	time_init() 함수의 분석	915
17.7.3.9.	softirq_init() 함수의 분석	918
17.7.3.10.	console_init() 함수의 분석	920
17.7.3.11.	printf() 함수의 분석	929
17.7.3.12.	init_modules() 함수의 분석	931
17.7.3.13.	kmem_cache_init() 함수의 분석	933
17.7.3.14.	mem_init() 함수의 분석	936
17.7.3.15.	kmem_cache_sizes_init() 함수의 분석	938
17.7.3.16.	proc_root_init() 함수의 분석	939
17.7.3.17.	fork_init() 함수의 분석	942
17.7.3.18.	proc_caches_init() 함수의 분석	943
17.7.3.19.	vfs_caches_init() 함수의 분석	944
17.7.3.20.	buffer_init() 함수의 분석	946
17.7.3.21.	page_cache_init() 함수의 분석	948
17.7.3.22.	kiobuf_setup() 함수의 분석	949
17.7.3.23.	signals_init() 함수의 분석	950

17.7.3.24.	bdev_init() 함수의 분석	950
17.7.3.25.	inode_init() 함수의 분석	952
17.7.3.26.	ipc_init() 함수의 분석	953
17.7.3.27.	dquota_init_hash() 함수의 분석	956
17.7.3.28.	check_bugs() 함수의 분석	959
17.7.3.29.	smp_init() 함수의 분석	962
17.7.3.30.	kernel_thread() 함수의 분석	963
17.7.3.31.	cpu_idle() 함수의 분석	965
17.7.4.	init() 함수의 분석	968
17.7.4.1.	do_basic_setup() 함수의 분석	969
17.7.4.2.	sock_init() 함수의 분석	971
17.7.4.3.	do_linuxrc() 함수의 분석	973
17.7.4.4.	do_initcalls() 함수의 분석	974
17.7.4.5.	free_initmem() 함수의 분석	974
17.8.	NETWORK DEVICE DRIVER.....	976
17.8.1.	Register	976
17.8.2.	Module	987
17.8.3.	Probe	989
17.8.4.	Reset	999
17.8.5.	Open	1000
17.8.6.	Close	1010
17.8.7.	Send	1011
17.8.8.	Interrupt	1012
17.8.9.	DMA	1014
17.8.10.	Receive	1019
17.8.11.	Statistics	1020
17.8.12.	Multicasting	1021
17.8.13.	MAC address	1022
17.8.14.	Timeout	1022
18.	IR (INFRA-RED) DEVICE DRIVER.....	1024
18.1.	NSCIR 모듈의 분석	1024
18.1.1.	Protocol의 등록/해지 및 변경	1032
18.1.2.	File Operation Vector	1035
18.1.3.	Interrupt Handler의 설정과 해제	1037
18.1.4.	Protocol Bottom Half Handler	1039
18.2.	MOUSE 모듈의 분석	1041
18.2.1.	Infrared Bus Mouse 모듈의 분석	1041
18.2.2.	Linux Kernel Bus Mouse Interface의 분석	1044
18.2.2.1.	Bus Mouse의 등록과 해제	1044
18.2.2.2.	Bus Mouse의 Event 알리기	1046
18.2.2.3.	Bus Mouse의 파일 연산 벡터	1048
19.	LINUX I2C BUS SYSTEM.....	1055
19.1.	I2C BUS FOR LINUX의 분석	1055
19.1.1.	i2c-core.c의 분석	1059
20.	REAL-TIME LINUX(RTLINUX)	1092
20.1.	EMBEDDED SYSTEM VS. REALTIME SYSTEM	1092
20.2.	WHAT IS RTLINUX?	1092
20.2.1.	Open RTLinux와 RTLinux Pro	1095
20.2.2.	MiniRTL	1096
20.3.	RTLinux의 설치	1096
20.4.	RTLinux의 THREAD 구현	1101
20.4.1.	PThread의 생성	1102
20.4.2.	PThread의 종료	1104

20.4.3.	PThread의 취소	1106
20.4.4.	PThread의 Signal.....	1106
20.5.	RTLINUX의 TIMER 구현	1107
20.6.	RTLINUX의 SCHEDULER 구현.....	1111
20.7.	RTLINUX의 FIFO(FIRST IN FIRST OUT).....	1115
20.7.1.	FIFO 모듈의 초기화와 해제.....	1116
20.7.2.	Open/Release 함수	1119
20.7.3.	기타 FIFO와 관련된 함수들.....	1122
20.7.4.	Read/Write 함수	1122
20.7.5.	IOCTL 함수	1125
20.8.	RTLINUX의 PERFORMANCE분석	1127
20.8.1.	Interrupt Latency.....	1127
20.8.2.	Context Switching Time	1128
20.8.3.	Preemption Time.....	1129
20.9.	RTLINUX의 응용	1130
20.10.	결론(CONCLUSION).....	1130
21.	앞으로 남은 일	1133
22.	APPENDIX.....	1134
22.1.	LINUX NETWORK DEVICE DRIVER 개발 예	1134
22.1.1.	Data Structure의 사용법	1136
22.1.2.	Transmit Queue Initialization	1136
22.1.3.	Buffer List Initialization	1137
22.1.4.	Free Descriptor Area Initialization.	1137
22.1.5.	Device private structure	1138
22.1.6.	Module적재	1138
22.1.7.	Module 해제	1139
22.1.8.	Device의 초기화	1139
22.1.9.	Device structure의 등록	1141
22.1.10.	Device open	1143
22.1.11.	Packet의 전송	1144
22.1.12.	Interrupt의 처리	1145
22.1.13.	Reception interrupt처리	1146
22.1.14.	Transmission Interrupt처리	1147
22.1.15.	Software Reset	1148
22.1.16.	Multicast Address의 Setting	1149
22.1.17.	Device I/O control	1150
22.1.18.	Device Driver의 Close.....	1151
22.1.19.	Network의 Setting	1151
22.2.	INIT.H 파일의 분석	1152
22.2.1.	초기화(initialization) 코드의 수행	1152
22.2.2.	커널 Parameter에 대한 해석	1158
22.2.3.	Assembly에서 사용되는 section 정의	1159
22.2.4.	모듈(Module)이 정의된 경우의 init.h 파일의 해석	1160
22.2.5.	그 외의 정의들	1161
23.	참고 문헌.....	1163

그림 목차

그림 1. 사용자 관점에서의 KERNEL	21
그림 2. LINUX 커널의 기본구조	22
그림 3. 커널 구성 요소간의 연관 관계	23
그림 4. LINUX KERNEL의 SOURCE TREE구성	24
그림 5. LINUX에서의 KERNEL COMPILE	25
그림 6. INTEGRATED (MONOLITHIC) ARCHITECTURE VS. MICRO KERNEL ARCHITECTURE	27
그림 7. PROCESS RELATION FOR LINUX	34
그림 8. PROCESS SESSION, GROUP와 TTY	35
그림 9. PROCESS TRANSITION DIAGRAM FOR LINUX	36
그림 10. MULTI-LEVEL FEEDBACK QUEUE	39
그림 11. SCHEDULE() 함수의 FLOWCHART	45
그림 12. GOODNESS() 함수의 FLOWCHART	46
그림 13. LIGHT WEIGHT PROCESS	56
그림 14. USER THREAD의 구성	57
그림 15. POLL_TABLE_STRUCT(OR POLL_TABLE) 구조체의 연관 관계	77
그림 16. ELF 파일의 포맷 - 관점에 따른 형식	87
그림 17. 데이터 ENCODING	91
그림 18. 세마포어의 커널내 구조	131
그림 19. 메시지 큐의 전체 구조	139
그림 20. 공유 메모리의 전체 구조	151
그림 21. PROCESS의 MEMORY내의 IMAGE	155
그림 22. TASK QUEUE의 구조	159
그림 23. TIMER TABLE의 구조	160
그림 24. TIMER LIST의 구조	161
그림 25. WAIT QUEUE의 구조	162
그림 26. TOP HALF와 BOTTOM HALF	164
그림 27. TASKLET, BOTTOM HALF, SOFTWARE IRQ의 구조	176
그림 28. LINUX에서의 SCHEDULER 호출	182
그림 29. FILE SYSTEM을 구성하는 요소들	208
그림 30. EXT2 파일 시스템의 구조	217
그림 31. EXT2 파일 시스템에서의 SUPER BLOCK OBJECT의 구조와 디스크 이미지의 관계	220
그림 32. I_BLOCK필드가 가리키는 데이터 블록	223
그림 33. INTEL X86 시스템에서의 주소 변환	225
그림 34. 세그먼트 디스크립터	226
그림 35. 세그먼트 SELECTOR를 이용한 선형 주소의 생성	229
그림 36. 특권 레벨의 링	232
그림 37. INTEL CPU의 페이지	234
그림 38. 리눅스에서의 페이지	235
그림 39. 페이지 디렉토리와 테이블의 구조	236
그림 40. 커널과 예약 영역을 위한 페이지 프레임들	238
그림 41. 페이지 할당을 위한 커널 데이터 구조의 사용 예	249
그림 42. BUDDY ALGORITHM의 예	267
그림 43. KMEM_CACHE_T(=KMEM_CACHE_S)와 SLAB_T(SLAB_S)의 관계	270
그림 44. 프로세스의 주소 공간	287
그림 45. NETWORK ARCHITECTURE IN LINUX KERNEL	305
그림 46. IP ADDRESS CLASS	306
그림 47. 리눅스의 네트워크 구조	308
그림 48. 프로세스와 관련된 BSD/INET 소켓의 생성	318

그림 49. BSD SOCKET 및 INET SOCK, SK_BUFF의 상관 관계.....	326
그림 50. LINUX의 NETWORK ARCHITECTURE.....	364
그림 51.LINUX KERNEL의 구조	368
그림 52. I/O ADDRESS SPACE	369
그림 53. ISA Bus상의 ADDRESS	370
그림 54. EISA BUS ARCHITECTURE	371
그림 55.PCI BUS ARCHITECTURE.	372
그림 56. PCI의 전체 구조.....	381
그림 57. DMA의 구조	383
그림 58. 모듈의 전체 구조	393
그림 59.문자 디바이스 드라이버와 블록 디바이스 드라이버의 구성.....	409
그림 60.블록 디바이스에 대한 접근.....	411
그림 61.NETWORK DEVICE DRIVER의 기본 구조.....	428
그림 62. ETHERNET FRAME STRUCTURE	429
그림 63.PROTOCOL STACK및 INTERFACE	430
그림 64.SOCKET BUFFER STRUCTURE	432
그림 65. RTC HARDWARE MODULE의 구성	446
그림 66. RTC의 ADDRESS MAP	446
그림 67. RTC의 REGISTER A, B, C, D의 포맷(FORMAT).....	447
그림 68. LINUX의 POWER MANAGEMENT 구성	486
그림 69. APM의 구성 요소	500
그림 70. SYSTEM POWER STATE의 천이(TRANSITION).....	501
그림 71. LINUX의 APM ARCHITECTURE.....	502
그림 72. SEGMENT DESCRIPTOR의 필드 정의.....	503
그림 73. USB의 BUS TOPOLOGY	550
그림 74. USB DEVICE DRIVER STACK	551
그림 75. WINDOWS USB DRIVER ARCHITECTURE.....	552
그림 76. USB DESCRIPTOR HIERARCHY	554
그림 77. USB CORE API LAYERS	561
그림 78. USB의 DATA TRANSFER구조.....	567
그림 79. USB TRANSACTION의 구조.....	568
그림 80. ENDPOINT의 STATE TRANSITION	568
그림 81. CONTROL TRANSFER PHASE	569
그림 82. BULK/INTERRUPT TRANSFER PHASE	570
그림 83. ISOCHRONOUS TRANSFER PHASE	571
그림 84. HOST와 DEVICE간의 PIPE를 이용한 통신.....	573
그림 85. USB SETUP PACKET의 구조	577
그림 86. SOCKET BUFFER구조체.....	594
그림 87. IEEE1394 DEVICE DRIVER의 구성	596
그림 88. IP OVER 1394의 ARCHITECTURE.....	600
그림 89. IP OVER 1394의 ARP PACKET의 구조	616
그림 90. PC CARD SOFTWARE의 ARCHITECTURE	623
그림 91. LINUX에서의 PCMCIA 구현 ARCHITECTURE	625
그림 92. DEV_LINK_T 구조체와 DEV_NODE_T 구조체의 관계	637
그림 93. PC CARD의 메모리 윈도우	655
그림 94. TUPLE을 가진 CIS의 예.....	657
그림 95. MULTIFUNCTIONAL PC CARD의 CIS 구조.....	659
그림 96. INPUT DEVICE의 ARCHITECTURE	700
그림 97. BOOTSECT.S의 RAM IMAGE(1).....	725
그림 98. BOOTSECT.S RAM IMAGE(2)	726
그림 99. BOOTSECT.S의 수행 이후의 MEMORY IMAGE	729

그림 100. INTEL의 CONTROL REGISTER들	746
그림 101. INTEL CPU의 EFLAGS 레지스터	750
그림 102. FLOPPY DISK의 구성	753
그림 103. MS-DOS의 BOOT SECTOR의 구조	754
그림 104. LILO에서 사용하는 파일들	755
그림 105. MAP 파일의 구조	756
그림 106. MAP FILE의 IMAGE 디스크립터의 구조	757
그림 107. 다른 운영체제를 위한 MAP FILE의 DESCRIPTOR	757
그림 108. LILO이 메모리 구성	759
그림 109. 커널 파라미터 라인의 생성	760
그림 110. GPLR	772
그림 111. GPDR	773
그림 112. GPSR OR GPCR	773
그림 113. GRER OR GFER	774
그림 114. GEDR	775
그림 115. GAFR	776
그림 116. ICIP OR ICFP	781
그림 117. ICMR	781
그림 118. ICLR	782
그림 119. ICCR	782
그림 120. INTERRUPT CONTROLLER의 블록 DIAGRAM	783
그림 121. SA1100의 메모리 맵	784
그림 122. BLOB의 실행 순서	787
그림 123. RELOAD() 함수 수행이후의 메모리 상태	806
그림 124. BLOB의 COMMAND LOOP	816
그림 125. UUENCODE와 UUDECODE	824
그림 126. 압축된 커널 이미지	835
그림 127. KERNEL IMAGE의 LOADING과 DECOMPRESSION	838
그림 128. 커널의 재배치	854
그림 129. ARM COPROCESSOR 15 – REGISTER 0 : PROCESSOR ID REGISTER	858
그림 130. COPROCESSOR 15의 REGISTER 2의 FORMAT	868
그림 131. COPROCESSOR 15의 CONTROL REGISTER 1의 BIT 필드 정의	869
그림 132. TERMIOS 구조체의 사용 방식	923
그림 133. 커널의 배치도	937
그림 134. Rx에서의 DMA BUFFER 사용 예(RX가 RING BUFFER의 두 부분으로 나뉘어진 경우)	1018
그림 135. NSC SET-TOP BOX의 INFRA-RED 디바이스 드라이버의 구성	1024
그림 136. LINUX의 BUS MOUSE DRIVER의 구성	1054
그림 137. I2C BUS의 구성	1055
그림 138. I2C BUS의 자료구조 간의 관계	1059
그림 139. RTLinux의 ARCHITECTURE	1094
그림 140. RTLinux FIFO의 구조(ARCHITECTURE)	1115
그림 141. RTLinux의 적용 사례	1130
그림 142. FRAME 및 BUFFER DESCRIPTOR에 대한 이해	1136

표 1. TASK_STRUCT의 FLAG필드의 값.....	30
표 2. TASK_STRUCT의 PTRACE필드의 값.....	30
표 3. PERSONALITY값.....	31
표 4. PROCESS의 상태	36
표 5. IRQ LINE STATUS.....	60
표 6. CLONE_FLAG의 값	63
표 7. LINUX_BINPRM 구조체의 필드 정의	67
표 8. POLL의 설정 FLAG에 대한 정의	72
표 9. ELF헤더의 필드 정의	89
표 10. E_IDENT[]의 필드	90
표 11. 특수 목적의 섹션 인덱스	93
표 12. 섹션 헤더의 필드 정의	94
표 13. 섹션 타입	96
표 14. 섹션 헤더 테이블 엔트리 0에 대한 설정예	96
표 15. SH_FLAGS필드의 BIT정의	97
표 16. 섹션 타입에 따른 SH_LINK와 SH_INFO의 해석	97
표 17. 특별한 의미를 가지는 섹션들.....	98
표 18. 특별한 의미를 가지는 섹션들.....	99
표 19. 시그널과 관련된 시스템 콜 인터페이스	105
표 20. SIGACTION구조체의 FLAG설정 값	113
표 21. FILE 구조체의 필드 값	209
표 22. DIRECTORY ENTRY 구조체의 필드 값	211
표 23. INODE 구조체의 필드 값	213
표 24. SUPER BLOCK 구조체의 필드 값	215
표 25. 디스크상의 SUPER BLOCK의 구조.....	218
표 26. 메모리 상의 SUPER BLOCK의 구조	219
표 27. 메모리상의 EXT2_INODE_INFO구조체의 정의.....	221
표 28. 디스크 상의 DESCRIPTOR 구조체의 정의	221
표 29. 디스크 상의 INODE TABLE의 구조.....	222
표 30. 페이지의 종류.....	235
표 31. 페이지 구조체의 필드 정의	246
표 32. 페이지 할당을 위한 MASK값	256
표 33. KMEM_CACHE_S(KMEM_CACHE_T)의 정의.....	269
표 34. SLAB_S(SLAB_T)의 정의.....	269
표 35. KMEM_CACHE_CREATE()함수의 FLAG 파라미터 정의	275
표 36. MM_STRUCT의 필드 정의	281
표 37. VM_AREA_STRUCT의 필드 정의	285
표 38. VM_AREA_STRUCT의 FLAGS에 들어가는 값	286
표 39. 소켓에 대한 시스템 콜	309
표 40.SOCK 구조체의 필드정의	325
표 41.SK_BUFF구조체의 필드들	326
표 42.EISA Bus IRQ ASSIGNMENTS	371
표 43. PCI DEVICE의 CONFIGURATION ADDRESS SPACE	373
표 44. 모듈 구조체의 FLAG값	388
표 45. 버퍼 헤드의 필드값	413
표 46. 요청 기술자의 필드값	415
표 47. NET_DEVICE구조체의 정의.....	435

표 48. NET_DEVICE구조체의 FEATURE필드의 값	436
표 49. NET_DEVICE_STATS구조체의 정의	436
표 50. NET_DEVICE구조체의 FLAG값	440
표 51. RTC 레지스터 A, B, C, D의 필드 정의	448
표 52. RTC REGISTER A의 RATE BIT 설정에 따른 PERIODIC INTERRUPT의 발생 비율(RATE)와 SQ OUTPUT의 주파수	449
표 53. CX 레지스터의 INT 0x15 BIOS FUNCTION 호출 결과	504
표 54. INT 15 BIOS FUNCTION 0x05303의 호출 결과를 가지는 레지스터들에 대한 정의	505
표 55. APM BIOS FUNCTION의 요약(SUMMARY)	523
표 56. USB DEVICE의 DEVICE DESCRIPTOR의 FIELD들에 대한 설명	557
표 57. USB DEVICE의 CONFIGURATION DESCRIPTOR	557
표 58. USB DEVICE의 INTERFACE DESCRIPTOR	557
표 59. USB DEVICE의 ENDPOINT DESCRIPTOR	558
표 60. USB DEVICE의 STRING DESCRIPTOR	558
표 61. 작업 큐	588
표 62. 소켓버퍼 구조체의 중요 필드들에 대한 설명	594
표 63. IEEE 1394구현에 관련된 파일들	597
표 64. CARD SERVICE의 주요 SERVICE의 정의	631
표 65. CLIENT EVENT에 대한 설명	634
표 66. CONFIGURATION REGISTER의 포맷	639
표 67. SOCKET_INFO_T 구조체의 STATE 필드 값 정의	642
표 68. 기본적인 TUPLE의 포맷	657
표 69. CONFIGURATION TABLE ENTRY TUPLE을 구성하는 CIS TUPLE의 정의	659
표 70. CONFIGURATION REGISTER의 정의	660
표 71. CONFIGURATION OPTION REGISTER의 필드 정의	661
표 72. CARD CONFIGURATION AND STATUS REGISTER의 필드 정의	662
표 73. PIN REPLACEMENT REGISTER의 필드 정의	663
표 74. SOCKET AND COPY REGISTER의 필드 정의	664
표 75. EXTENDED STATUS REGISTER의 필드 정의	665
표 76. I/O LIMIT REGISTER가 가질 수 있는 값과 ADDRESS RANGE의 상관관계	665
표 77. IRQ_REQ_T의 ATTRIBUTE FIELD 값에 대한 정의	669
표 78. ATTRIBUTES 필드 값에 대한 정의	671
표 79. CONFIG_REG_T 구조체의 CONFIGBASE의 값에 대한 정의	672
표 80. CLIENT_REG_T 구조체의 ATTRIBUTES값 정의	673
표 81. REQUESTIO와 RELEASEIO에서의 ATTRIBUTES1과 ATTRIBUTES2가 가지는 값의 정의	677
표 82. WIN_REQ_T의 ATTRIBUTES 값의 정의	678
표 83. DUMMY_CS의 CARD SERVICE EVENT에 대한 정의	679
표 84. TUPLE_T 구조체의 ATTRIBUTES FIELD 값의 정의	683
표 85. CONFIG_INFO_T 구조체의 CARDVALUES 필드가 가지는 값의 정의	685
표 86. CISTPL_CFTABLE_ENTRY_T 구조체의 FLAGS 필드 값의 정의	687
표 87. CARD CONFIGURATION AND STATUS REGISTER의 값 정의	688
표 88. CISTPL_POWER_T 구조체의 PRESENT필드의 값 정의	689
표 89. CISTPL_IRQ_T 구조체의 IRQINFO1 필드 값의 정의	689
표 90. CISTPL_IO_T 구조체의 FLAGS 필드 값의 정의	691
표 91. INPUT DRIVER의 모듈들	696
표 92. INPUT_DEV 구조체의 필드 정의	697
표 93. CR4 레지스터의 필드	747
표 94. EFLAGS 레지스터의 필드	750
표 95. CR0 레지스터의 필드 정의	751
표 96. 파라미터 옵셋의 정의	760
표 97. GPLR 레지스터의 BIT 정의	772

표 98. GPDR 레지스터의 BIT정의	773
표 99. GPSR 레지스터의 BIT정의	774
표 100. GPCR 레지스터의 BIT 정의.....	774
표 101. GRER 레지스터의 BIT정의	774
표 102. GFER 레지스터의 BIT정의.....	775
표 103. GEDR 레지스터의 BIT정의	775
표 104. GAFR 레지스터의 BIT정의	776
표 105. GPIO ALTERNATE FUNCTION의 정의	776
표 106. GPIO 레지스터의 위치 정의	777
표 107. ICPR 레지스터의 BIT정의	779
표 108. ICMR 레지스터의 BIT 정의	782
표 109. ICLR 레지스터의 BIT 정의	782
표 110. ICCR 레지스터의 BIT 정의	783
표 111. INTERRUPT CONTROLLER REGISTER의 위치 정의	783
표 112. SA1100의 메모리 설정 레지스터	790
표 113. FLASH에 대한 명령 및 STATUS 값에 대한 정의	829
표 114. SA1100의 COPROCESSOR REGISTER 0의 필드 정의	858
표 115. DOMAIN ACCESS의 값	868
표 116. LOCAL MODE의 설정 요약	924
표 117. MODULE구조체의 정의	932
표 118. 자원의 한계에 대한 상수 값들의 정의	943
표 119. BLOCK_DEVICE구조체의 정의	951
표 120. 파일의 타입을 나타내는 상수(CONSTANT) 값들	956
표 121. 접근 허가를 나타내는 상수(CONSTANT) 값들	956
표 122. PACKETPAGE 메모리 맵(1)- 기본 설정부분	977
표 123. PACKETPAGE 메모리 맵(2)- EEPROM INTERFACE부분	977
표 124. PACKETPAGE 메모리 맵(3)- CONFIGURATION과 CONTROL REGISTER부분	977
표 125. PACKETPAGE 메모리 맵(4)- STATUS와 EVENT REGISTER부분	978
표 126. PACKETPAGE 메모리 맵(5)- Rx와 Tx를 위한 REGISTER와 ADDRESS관련 REGISTER부분	978
표 127. PP_RxCFG의 RECEIVE CONFIGURATION과 INTERRUPT MASK BIT 정의	979
표 128. PP_RxCTL의 RECEIVE CONTROL BIT의 정의	979
표 129. PP_TxCFG TRANSMIT CONFIGURATION INTERRUPT MASK BIT DEFINITION	980
표 130. PP_TxCMD TRANSMISSION COMMAND STATUS REGISTER BIT DEFINITION	980
표 131. PP_BUFCFG BUFFER CONFIGURATION INTERRUPT MASK BIT DEFINITION	981
표 132. PP_LINECTL LINE CONTROL BIT DEFINITION	981
표 133. PP_SELFCTL SOFTWARE SELF CONTROL BIT DEFINITION	982
표 134. PP_BUSCTL ISA BUS CONTROL BIT DEFINITION	982
표 135. PP_TESTCTL TEST CONTROL BIT DEFINITION	983
표 136. PP_RxEVENT RECEIVE EVENT BIT DEFINITION	983
표 137. PP_TxEVENT TRANSMIT EVENT BIT DEFINITION	984
표 138. PP_BUFEVENT BUFFER EVENT BIT DEFINITION	985
표 139. PP_LINEST ETHERNET LINE STATUS BIT DEFINITION	985
표 140. PP_SELFST CHIP SOFTWARE STATUS BIT DEFINITION	986
표 141. PP_BUSST ISA BUS STATUS BIT DEFINITION	986
표 142. PP_AUTONEGCTL AUTO NEGOTIATION CONTROL BIT DEFINITION	986
표 143. PP_AUTONEGST AUTO NEGOTIATION STATUS BIT DEFINITION	986
표 144. SELECTING MEDIA FOR CS89x0 ETHERNET CONTROLLER CHIP	994
표 145. INTERRUPT LATENCY의 측정	1128
표 146. CONTEXT SWITCHING TIME측정	1128

저자의 말

난 아마추어다. 난 아마추어이기에 기꺼이 이런 글을 쓸 수 있다. Linux는 그런 아마추어가 만든 것이며, 이젠 서서히 상업 시장으로의 발길을 재촉하고 있다. 어려운 용어들이 난무하고 디지털이 어쩌구 한다. Linux는 어쩌면 이러한 어려운 용어들 속에서 쉽게 개발자와 친근해 질 수 있는 친구로서, 네트워크화된 사회에서 나올 수 밖에 없는 필연적인 결과물(혹은 중간 결과물) 일지도 모른다.

예전에 군대를 갔다가 복학했을 때가 생각난다. 숙제를 FTP를 이용해서 제출하라는 말에, 멍한 눈으로 교수를 응시했다가, 옆에 앉은 후배를 다그쳐서 그게 원지를 알게 되었을 때이다. 결국 난 숙제를 그냥 디스켓에 담아서 냈다. 인터넷은 그렇게 내게 별로 관대하지 못했다. 복학생이 뭘 알기에... 조금 더 후에 친구 녀석이 Linux라는 것을 디스켓으로 70여장에 가깝게 주었을 때, 이게 어떤 것인가를 고민하다가 X-Window까지 올리는데 무려 3개월이 걸렸다. 친구 녀석은 내에게서 3개월간의 밤을 빼앗아 갔던 것이다. 요즘은 1시간도 안될 꺼리를... 그 좋은 CD-ROM을 두고서 말이다.

결국 난 지금에서야 인터넷의 도움으로 쇼핑도 하고, 책과 자료들도 찾고 ADSL로 집에서 상영되지 않은 영화도 열심히 다운 받아서 본다. 바로 이 편리함 속에서 그 방대한 양의 지식들이 불과 몇 초 사이에 지구를 몇 바퀴나 돌고 있는 것이다. 아직 인터넷의 “인”자가 참을 “인”자로 생각되는 곳에서도 많은 사람들이 Linux의 코드를 열심히 분석하고 있고, 더 나은 것을 만들려고 노력하고 있다. 난 그 중의 한 사람이며 그들 개발자들보다 더 많이 알지도 못한다. 다만 그들의 지적 노력의 산물을 내 나름대로 분석하고 내 나름대로 정의해서 보고 있을 뿐이다.

이 글을 읽는 사람이라면, 적어도 책의 첫 몇 페이지는 넘긴 사람일 것이다. 그런 사람들을 위해서 책의 내용이 어떤 것을 말하는지를 알리고자 다음에 몇자 적어본다. 참고하기 바라지만 절대적인 기준은 못된다.

이런 사람이라면 이 글을 읽어볼 만 하다고 생각한다.

- 밤이 깊어도 잠이 오지 않아 이리저리 뒤판이는 사람.
- 인터넷에 널린 문서들과 서점의 외국어 책 가판 대에 진열된 많은 Linux관련 도서를 다 읽기는 싫고, 영어실력도 못 따라가는 사람. 그렇다고, 영어 공부도 할 수 없는 사람. 하지만, 기본적인 영어는 아는 사람이며, 적어도 영한 사전을 책상 위에 하나쯤은 두고 사는 사람.
- 적어도 Linux를 가지고 밥을 먹고자 하는 사람.(아마 나 자신일지도 모른다. 하지만 배가 고픈 것도 사실이다. ^^;)
- 인터넷 서점에서 Linux를 키워드로 책을 찾았지만, next라는 표시와 함께 한도 끝도 없이 많은 책들로 어떤 책을 사야 할지를 고민하는 사람. 그리고, 그 많은 책들을 다 볼 시간도 없는 사람.

하지만.. 정작 책장을 넘기면 이런 사람들이 읽어야 할 것이다.

- Linux는 설치하고 ls와 cd를 칠 줄은 알지만, 더 많은 것을 알기를 원하는 사람.
- 실제적인 개발에 조금이라도 도움을 원하는 사람.(정말 조금 밖에는 도움이 안될지 모르지만)
- 디바이스 드라이버 입문 책은 다 보았지만, 정작 구현하려고 관련된 소스 코드를 찾았지만 뭐가 원지 구분이 안 되는 사람.

하지만, 이런 사람에게는 다른 책을 권하고 싶다.

- Linux를 이용하지만, 커널이나 기타 구조에 대해서는 관심이 없는 사람.
- Linux 명령어를 배우려고 공부하는 사람.
- C 언어는 알지만 Hello world이외의 프로그램을 해본 경험이 없거나, Assembly어는 노가다를 좋아하는 사람만 한다고 생각하는 사람.
- 운영체제를 자세히 알거나, 운영체제의 이론을 처음으로 배우려고 하는 사람.

- 기타등등의 앞에서 나열한 사람들에 속하지 않는 사람들.

어쨌든 어떤 사람들에게는 유용한 자료가 될 것이지만, 다른 사람들에게는 치명적인 자료가 될지도 모른다. 헛된 시간을 낭비하지 않기를 바란다. 이제 이 책은 내 책상 위의 컴퓨터 속에 한 폴더를 벗어나 세상으로 나가려고 한다. 내 목소리를 흉내내지만 내가 아니며, 나의 글 쓰는 스타일을 틀어줘고 있지만 나의 밥 먹는 스타일은 아니다. 잘못된 지식을 전파하는 것은 나쁜 일이기에 설불리 꺼내 놓기가 두려운 것도 사실이다. 그럼에도 불구하고 이 글을 출판하는 것은... 역시 나는 아마추어이기 때문이다. 용서 하기 바란다. 더 좋은 글을 쓸 수 있도록 노력하길 빌어주었으면 한다.

2002년 월드컵의 해에 드립니다.

권 수 호.

Acknowledgements

이 글은 혼자만의 노력이 아니다. 전세계에 있는 리눅스 개발자들의 노력을 모은 것에 불과하며, 잠시동안 이렇게 문서를 쓸 수 있도록 허락해준 분들의 뜻이다. 난 단순히 열심히 자판을 두드려서 문장으로 만든 노력만을 더했을 뿐이다. 삼성전자의 CTO전략실 RTOS팀에서 일하는 여러 리눅스 개발자들이 이 다큐먼트를 쓰도록 자극해 주었으며(물리적인 힘과 정신적인), 난 틈틈이 시간이 날 때마다 열심히 리눅스 커널 코드를 뜯어보는 일로 눈을 피곤하게 만들었을 뿐.

문서의 뒷부분에 참조한 여러 자료들을 열거해 두었는데, 그들의 노력은 참으로 절실했다. 그렇지 않다면, 어떻게 그 많은 부분들을 분석해 낼 수 있었겠는가? 그들 모두가 자신들의 노력을 별다른 대가 없이 제공 인터넷으로 혹은 책의 형태로(돈을 조금 드린 부분이다.) 했으므로, 이 글이 가능한 것이다.

내가 아는 어떤 신부님이 아마도 나에게 있어서, 리눅스를 보게 만든 가장 큰 계기를 던져주었으므로 그 분께 가장 큰 감사를 드리고 싶다. 또한, 논문을 지도해 주시며 항상 관심을 가지고 지켜봐 주셨던 낭종호 교수님과, 예전에 연구실에서 같이 먹고 공부하며 열심히 오락에 미쳐주었던 흥승욱 선배와 말없이 원가에 항상 열중하던 김창섭 선배 비롯해서, 허성관과 조만준에게 감사의 뜻을 전하고 싶다.

같은 팀에서 일하며, 라면과 늘어나는 뱃살을 걱정해준 유모 수석님과 박모 책임연구원에게도 감사하고 싶다. 그들이 아니라면, 밤하늘에 대고 열심히 숨을 고를 일도 없었으리라(요즘 살 뺀다고 열심히 저녁에 운동중이다.). 또한, 항상 일과 강의로 바쁜 여러 연구원들이 그들이 했던 일들에 대해서 이런 저런 형태로 알려주었기에 나름대로 혼자서 쓸데없는 일을 하고 있다는 생각이 들지 않도록 해 주었다.

마지막으로 윤 모씨의 둘째로 태어나, 2000년의 마지막 달에 나의 아내가 되어준 만정에게 항상 감사하며 이 글을 맺는다.

2002년 권수호

1. Introduction

1.1. 목적

본 문서의 목적은 리눅스(Linux) 커널(kernel)의 완전한 분석을 목적으로 하며, 참조하는 예제는 모두 커널 버전(version) 2.4이상으로 사용할 것이다. 본 문서는 이해를 목적으로 작성되었기 때문에 문서의 사용자는 리눅스에 대한 열정과 노력을 많이 투자할 수 있는 사람들이라 믿는다. 하지만, 기본적인 것들을 알고자 한다면 각 장(chapter)의 이름과 그것이 무엇을 말하는지 정도만 보아도 상관없으리라 생각한다. 본 문서는 사용자의 입장에서 얻을 수 있는 많은 문서들과 책, 그리고 사용자의 입장에서 코드를 직접 들여 다 보면서 작성되었기에, 더 많은 것을 알기를 원하는 사람들은 직접 그런 것들을 찾아서 보기 바란다.

1.2. 범위

본 문서에서 다루고자 하는 것은 리눅스 커널이므로 이것에 한정된 설명을 한다. 만약 본 문서의 독자가 다른 것에 관심이 있다면 인터넷에서 그와 연관된 글을 찾아보아야 할 것이다. 어렵다고 생각되면 과감히 생략하는 것도 필요하다.

1.3. Kernel이란?

커널(kernel)이란 운영체제의 핵심을 이루는 요소로서 컴퓨터내의 자원을 사용자 프로그램이 사용할 수 있도록 관리하는 프로그램이다. 커널에서 제공하는 기능은 크게 프로세스의 관리, 파일 시스템, 메모리 관리, 네트워크가 있으며, 사용자 프로그램은 이러한 기능들을 정해진 규칙에 따라서 커널에 요구하게 되며, 커널은 이러한 요구들을 만족시켜 주어야 한다. [그림1]은 현재의 커널구성을 간략히 보여준다. 즉, 시스템의 제일 핵심에 커널이 자리를 잡고 있음을 알 수 있다. 사용자 어플리케이션(user application)은 커널에 대한 요청을 시스템 콜의 형태로 하게 되며, 커널은 이 시스템 콜을 처리해서 사용자 어플리케이션에 처리의 결과를 알려주어야 한다.

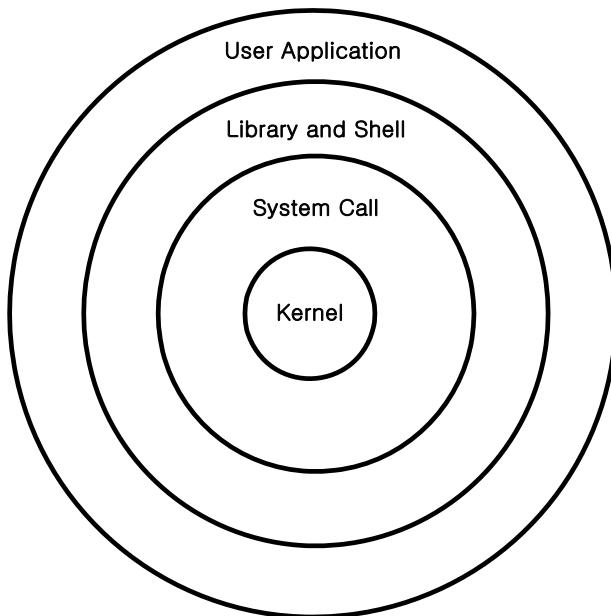


그림 1. 사용자 관점에서의 Kernel

사용자 어플리케이션은 반드시 메모리(memory)상에 올라와야지 실행할 수 있으며, 또한 미리 저장된 형태가 있어야 한다. 이러한 목적으로 메모리관리(memory management)와 파일 시스템(file system)이 필요하게 되며, 실제로 메모리상에 올라온 사용자 어플리케이션의 관리를 위해서 프로세스 관리(process

management)가 필요하다. 네트워크(Network)는 반드시 운영체제를 위해서 필요한 요소는 아니지만, 현재의 컴퓨터 시스템에서 네트워크가 안 된다면, 많은 운영체제들과의 경합에서 밀릴 수 밖에 없다. 또한 네트워크로 연결된 다른 컴퓨터들간의 통신을 가능하게 하기 위해서 널리 인정된 프로토콜(protocol)을 지원하는 것은 당연하다고 볼 수 있겠다. 커널의 구조를 좀더 자세히 들여다보면 [그림2]는 리눅스의 커널에 대한 기본구조를 간략화 해서 보여주는 것이다.

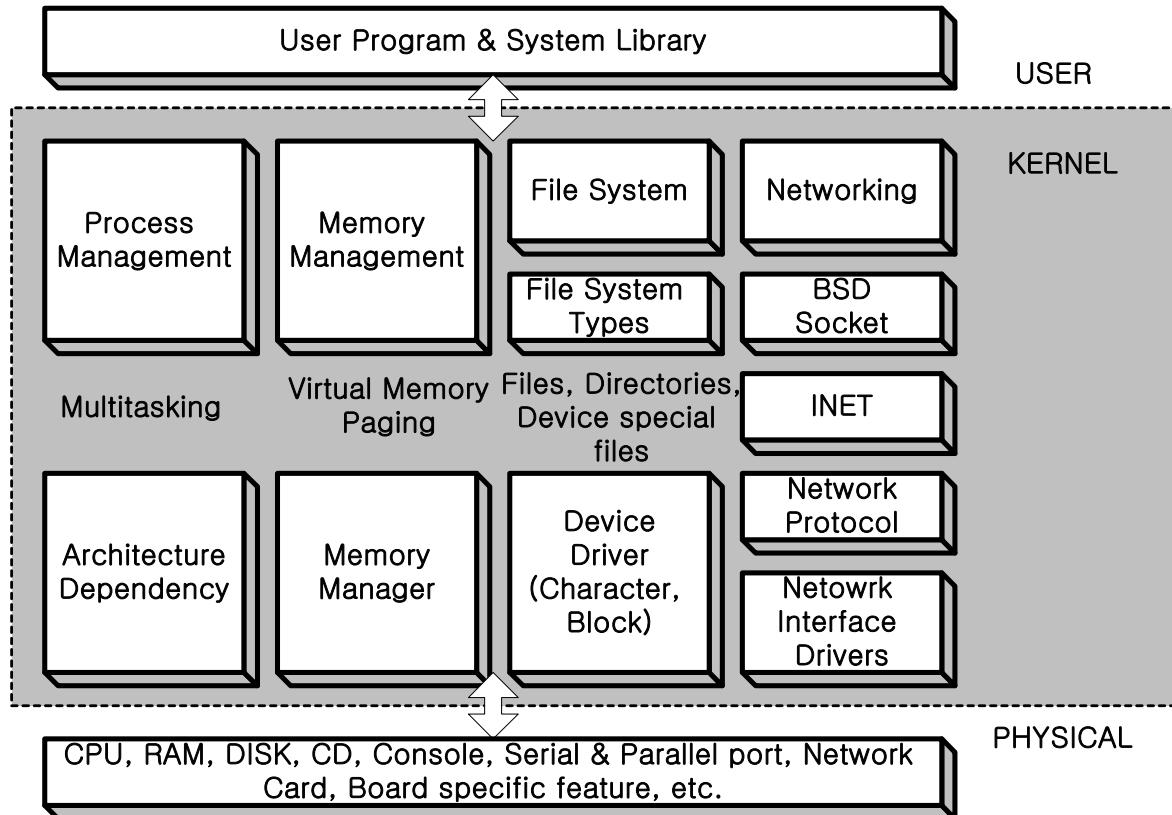


그림 2. Linux 커널의 기본구조

앞으로 우리가 보게 될 내용을 요약해서 보면, 프로세스의 관리는 리눅스에서 실행되는 프로세스에 대한 관리를 맡고있으며, 메모리 관리는 메모리의 할당과 회수를, 파일 시스템은 물리적으로 자료를 저장 및 다시 가져오는 기능을, 그리고, 마지막으로 네트워크에서는 컴퓨터들과의 통신에 대한 것을 다루고 있다. 이러한 각 부분은 상호 독립적으로 구현되어있으며, 전체적으로 둑여있다. 이러한 구조를 가장 잘 볼 수 있는 것이 프로세스의 구조를 명시하는 `task_struct`구조체일 것이다. `task_struct`구조체는 프로세스에 대한 정보를 담는 데이터 구조로 프로세스의 실행에 관련된 모든 정보를 알 수 있다. 이하에서는 이러한 구조를 머리 속에 넣고 하나하나의 구성요소(component)에 대해서 실제적으로 코드(code)를 보면서 설명을 진행할 것이다.

위의 [그림2]는 기능적인 분류에 대한 설명이며 이것을 좀더 연관된 구조로서 이해하기 위해선 각각이 하는 역할에 대해서 이해가 필요하다. [그림3]은 상호 연관된 구조로서 각각의 커널 구성요소(kernel component)들에 대한 것을 보여준다.

사용자 프로그램은 시스템 라이브러리(system library)의 도움을 받거나 아니면 직접적으로 소프트웨어 인터럽트(software interrupt)를 이용해서 트랩(trap)을 걸어서 커널에 접근하게 되며, 이러한 모든 접근은 시스템 콜 인터페이스(system call interface)를 통한다. 일단 커널로 제어(control)가 전송된 상황에서는 커널 모드(kernel mode)로 실행되게 되며, 환경전환(context switch)이 일어나기 전에는 사용자 프로세스 환경(user process context)으로 진행된다. 파일 시스템(File system) 혹은 프로세스 관리(process management) 하부구조(subsystem)에서 다시 커널 내부의 코드를 수행하게 되고, 이는 다시 하위의 하드웨어 제어 유닛(hardware control unit)으로 제어가 넘어간다. 여기서 하드웨어에 대한 접근(access)이 일어나게 되며,

복귀(return) 시에는 다시 이상에서의 과정을 역으로 밟아서 사용자 프로그램으로 제어를 넘기게 된다. 물론 이것은 극히 간단한 사용자 프로그램과 커널, 그리고 하드웨어에 대한 접근 과정을 보인 것이다. 실제적으로는 좀더 복잡한 과정이 뒤 따르지만, 도입이니 만큼 여기까지만 하도록 하자. [그림3]은 이상의 과정을 좀더 운영체제의 관점에서 좀더 체계적으로 보여준다.

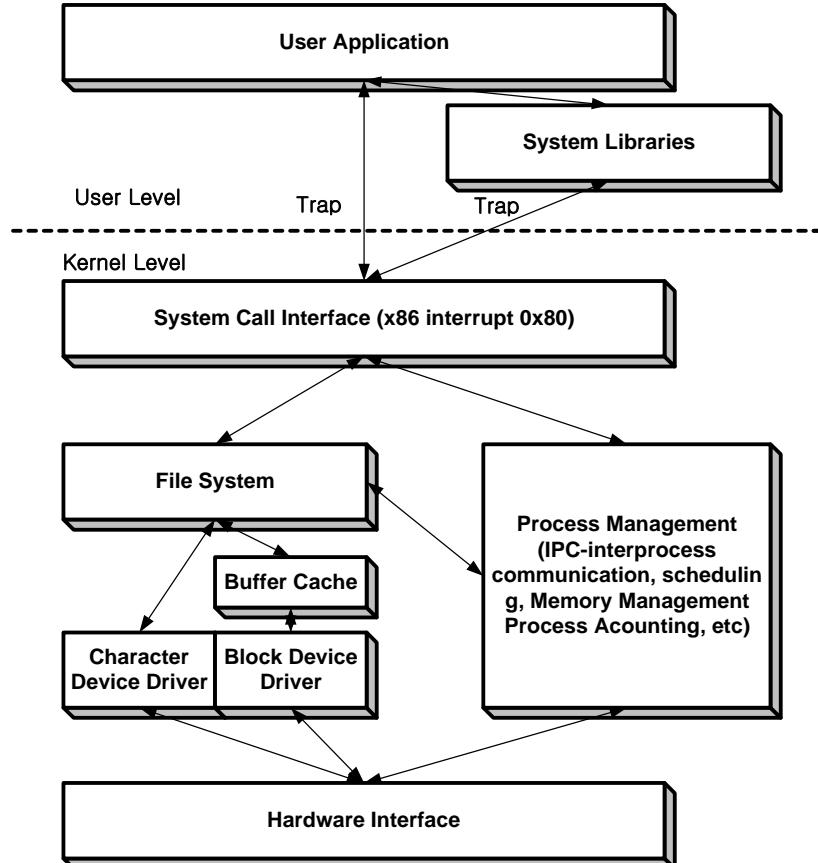


그림 3. 커널 구성 요소간의 연관 관계

위와 같은 과정에서 가장 중요한 점은 리눅스는 모든 프로세스가 어느 정도는 자원을 점유할 수 있도록 보장해 주어야 한다는 것이다. 즉, 특정의 사용자 어플리케이션에 전적으로 자원을 다 할당하지 않고, 공평하게 자원을 사용할 수 있도록 해준다는 것이다. 위의 [그림3]에서는 네트워크에 대한 것이 보이지 않는데, 이는 네트워크를 프로세스간 통신(inter-process communication)의 일부로서 보기 때문이다.

이상에서 우리는 리눅스 커널의 기본적인 구성을 그림과 함께 보았다. 여기서 유의할 점은 커널은 사용자 어플리케이션이 원활하게 수행되도록 보장하기 위한 기능을 제공하기 위해서 항상 커널의 자료구조를 내부에 유지한다는 점이다. 커널에서의 연산은 비용이 크기 때문에 커널 프로그램 코드는 최적화되어 있어야 하며, 또한 크기도 작을수록 좋을 것이다. 따라서, 자원을 관리하기 위한 자료구조는 최적화로 구현되어야 할 것이다. 리눅스는 현재 커널에서 많은 최적화를 위한 팁(tip)들이 있으며, 이는 커널의 이해를 어렵게 하는 요소로 작용할 수도 있지만, 좋은 성능을 보장해 준다는 점에서 충분히 가치가 있다고 볼 수 있을 것이다.

1.4. Kernel의 구성

커널은 대부분이 C code로 작성되어 있으며 프로세스의 구조에 의존적인(processor architecture dependent) 한 부분들과 속도를 요하는 부분만 기계어 코드(assembler code)로 작성되어 있다. 따라서 커널의 구조에 의존적인 (architecture dependent) 부분을 고침으로 해서 새로운 프로세스로의 포팅(porting)이 가능하다.

자유로이 쓸 수 있는 커널의 코드를 사용자가 구한다면, 아마 다음의 두 가지 정도를 들 수 있을 것이다. 즉, FreeBSD와 리눅스 정도가 될 것이다. 두 운영체제 모두 인텔(Intel)의 x86에서 동작하면 어느 시장을 가지고 있다. 리눅스의 경우는 `/usr/src/linux`에 커널의 소스코드(kernel source code)가 들어가 있다. [그림4]는 간단한 리눅스 커널의 소스 트리(source tree)구조를 보여준다.

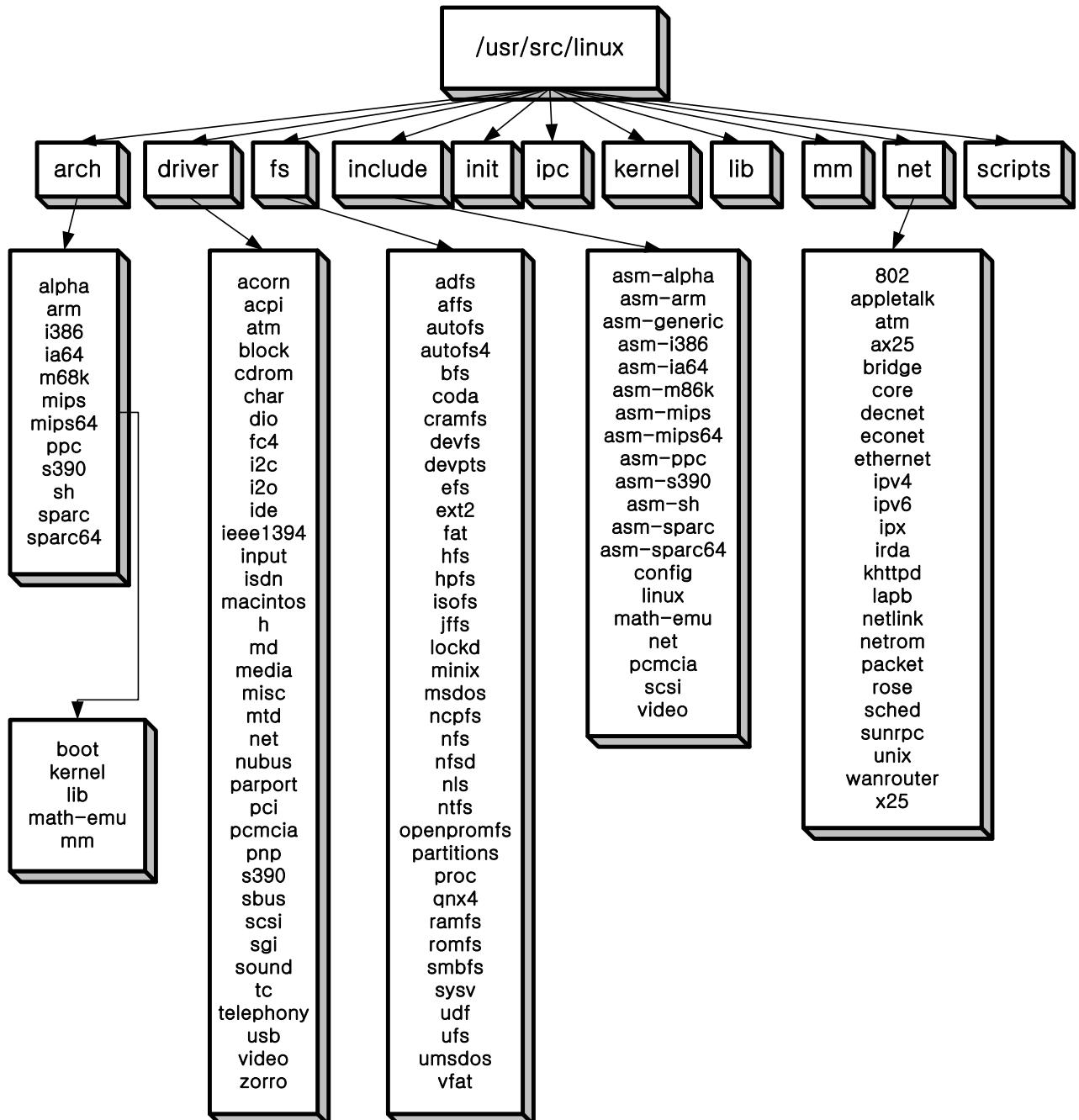


그림 4. Linux Kernel의 source tree 구성

전제적으로 말하자면, 커널의 source 코드를 보고 공부를 하고자 한다면 FreeBSD를 보는 것이 좋을 것이다. 오래된 만큼 그 깊이도 심오하기 때문이다. 하지만, 새로운 기술과 주제를 따라가고자 한다면 리눅스쪽을 선택하는 것이 더 좋을 것이다. 현재 리눅스는 [그림4]에서와 같이 많은 CPU에 포팅(porting)되어 있으며, 새로운 주변기기 및 알고리즘들을 적용해 나가고 있기 때문이다. 또한 개발자들이 전 세계적으로 소스를 공유하고 개발에 동참함으로써 개발 속도도 빠르다. 한가지

약점이라면 너무 빠르게 업그레이드 되는 측면인데, 어떤 제품에 적용할 운영체제를 선택하기가 까다로울 수 있다. 둘 다 공짜로 다운 받아서 사용할 수 있기에 이용에는 부담이 없다.

1.5. 리눅스 커널의 compile방법

리눅스를 처음 접하는 사람이 가장 힘든 것 중의 하나가 dir과 cd밖에 모른다는 점과, “왜 내 컴퓨터에 있는 장치를 리눅스에서는 사용 못하고 윈도우에서만 가능할까?”라는 점일 것이다. 이는 잘못된 생각이다. 윈도우에서 가능하다면 리눅스에서도 가능하다. 적어도 어느 정도의 시간이 걸리기는 하겠지만, 직접 혹은 간접적으로 도움을 받아서 가능해 질 것이다.

가장 중요한 것은 리눅스에서는 어떤 장치를 사용하기 위해 커널을 새로이 만드는(build) 과정이 필요하다는 것이다. 윈도우 같은 경우는 필요한 장치에 대한 드라이버(driver)를 설치하라는 과정을 통해서 쓸 수 있게 되지만, 리눅스에서는 필요한 장치를 커널에 심거나 혹은 모듈(module)로서 로딩/loading)해 주어야 한다는 점이 차이가 난다. 간단한 리눅스 커널의 컴파일 방법은 [그림5]와 같다.

```
#cd /usr/src/linux
#make mrproper
#make config ( or make menuconfig, make xconfig, make oldconfig )
#make dep
#make zImage ( or bzImage )
#make modules( 만약, make config중에 module로서 compile할 것이 있다고 선택했을 경우 )
#make modules_install ( 역시 make modules를 해서 원가를 module로서 compile했다고 했을 경우 )
#make install
```

그림 5. Linux에서의 Kernel Compile.

*make mrproper*는 현재의 시스템의 프로세스 구조(processor architecture)에 맞도록 make 환경을 맞추는 것이며, *make config*은 자신의 시스템의 상황에 맞는 적절한 선택을 해주는 것이다. *make dep*는 앞에서 구성(configuration)한 것의 소스 의존성(source dependency)을 맞추는 것이며, *make zImage*는 kernel의 image를 만들고(build), *make modules*는 모듈로서 선택된 것들에 대한 컴파일을, *make modules_install*은 이와 같이 컴파일된 module을 /lib/modules 아래에 현재 커널의 버전 디렉토리(version directory)를 만들어서 복사하는 역할을 한다. 그리고 마지막으로 *make install*은 커널을 /boot 디렉토리 하에 넣어주고, lilo를 새로이 갱신하는 일을 한다. Lilo란 이후에서 다시 만나게 될 주제 중의 하나인데, 리눅스 loader의 약자이다. 커널을 적절히 선택적으로 부팅/booting)할 수 있도록 해주는 프로그램이다.

먼저 커널을 만들기(build) 위해서 준비해 주어야 할 것들을 알아보자. 첫번째로 해주어야 할 일이 자신의 하드웨어(hardware)에 대한 이해이다. 물론 완전한 하드웨어에 대한 이해를 말하는 것은 아니고, 자신의 시스템에 달려있는 기기들이 사용하는 칩셋(chipset) 및 기기의 연결 방식이 될 것이다. 확신하지 않는 것은 선택하지 않는다는 것이 지론이다. 또한 몇몇 기기들은 DMA 및 I/O 포트(port), IRQ(interrupt) 번호와 같은 것을 알아야 하는 경우도 있다. 충분히 이해가 되지 않았다면, 일단 부딪혀서 해결해 보는 것도 좋을 것이다. *make config*시에 도움말이 이러한 역할을 하도록 준비되어 있으며, 친절한 도움말이 주어질 것이다.¹

자, 이젠 커널을 구성하여 컴파일하는 방법까지 알아보았다. 그럼 커널은 어디서 구할 수 있을 것인가? 리눅스 커널은 자주 업그레이드(upgrade)되어 새로운 기능이 보완되던지 아니면, 기존에 있던 버그(bug)가 고쳐(fix)진다. 따라서, 자주 자주 업그레이드에 대한 것을 알아볼 필요가 있다. 리눅스 커널의 공식적인 배포는 <http://www.kernel.org>에서 하고 있으므로 이곳에 가서 kernel을 다운 받아서 쓰도록 하자. 물론 각각의 필요한 모듈에 대한 최신정보는 그 모듈의 개발 사이트(site)에서 알아봐서 패치(patch)해야 할 것이다.

¹ 가끔 없는 경우도 있으니, 너무 바라지는 말도록 하자.

1.6. Linux Kernel Image만들기의 상세

이번 장에서는 실제적으로 리눅스 커널의 만들기(build)과정을 상세히 보도록 하겠다. 먼저 빌드(build)하는 절차는 앞에서 살펴본 바와 같이 프로세스 구조(processor architecture)에 의존적인 부분들이 많으며 절차는 앞의 과정을 그대로 따르면 될 것이다. 여기서 보는 과정은 앞에서 입력한 명령이 어떻게 내부적으로 진행되는지를 보는 것이다.

커널 이미지(Image)를 만드는 과정은 아래와 같다.(물론 이것은 우리가 앞에서 본 명령어를 입력(type)할 때 연속적으로 일어나는 내부 과정에 대한 것이다.)

1. C와 기계어 소스 파일(assembly source file)들은 컴파일되어서 ELF relocatable² object format(.o)가 되며, 이들 중의 일부는 *ar* binary file utility를 이용해서 (.a)라는 형식의 라이브러리(library)로 만들어진다.
2. *ld utility*를 사용해서 1의 과정에서 생성된 .o file 및 .a file들을 vmlinux로 만들어준다. vmlinux는 정적으로 link되었으며, non-stripped ELF 32-bit LSB 80386 실행화일 형식이 된다.
3. *nm utility*를 이용해서 *system.map* 파일을 생성하고, 관련이 없는 심벌(symbol)들을 제거한다.
4. 프로세스 구조에 의존적인 부분에 대한 컴파일을 준비하기 위해서 ~/arch/i386/boot로 디렉토리를 바꾼다. 그리고 나서, bootsect.S를 target의 image에 따라서, 즉, bzImage(big image)인지 아니면 zImage인지 보고 전처리(preprocessing)해서 bbootsect.s 혹은 bootsect.s를 만든다.
5. 4에서 만들어진 bbootsect.s 혹은 bootsect.s를 기계어 컴파일러(assembly compiler)를 이용해서 “raw binary” 파일 형태인 bbootsect 혹은 bootsect를 만든다.
6. setup.S와 video.S를 전처리 후 5와 같이 bsetup.s나 setup.s를 만들어 “raw binary” 형태인 bsetup 혹은 setup를 만들어준다.
7. 만들어진 커널을 압축(compression)하기 위해서 ~/arch/i386/boot/compressed 디렉토리로 들어가서, 이전 과정에서 만들어진 vmlinux의 필요 없는 부분인 .note와 .comment ELF section을 잘라내 임시적인 커널 이미지(\$tmppiggy)를 생성한다.
8. 7에서 만들어진 임시적인 이미지를 압축한다. 이것은 *gzip* utility를 이용해서 가장 좋은(best compress but slow) 압축 효과를 가진 이미지로 gzip -9 < \$tmppiggy > \$tmppiggy.gz과 같이 만들어준다.
9. 다시 *ld utility*의 -r option을 이용해서 \$tmppiggy.gz를 relocatable 이미지인 piggy.o로 만들어준다.
10. 압축에 관계된 head.S와 misc.c를 컴파일해서 ELF object file 형식의 head.o와 misc.o를 만든다.
11. 9와 10에서 만들어진 piggy.o와 head.o, misc.o를 link(*ld utility*를 사용)해서 bvmlinux(big image일 경우) 혹은 vmlinux를 생성한다. 여기서 주의할 점은 big image일 경우에는 *ld*에 -Ttext 0x100000을, 그렇지 않을 경우에는 -Ttext 0x1000을 주어서 link한다는 점이다. 이렇게 해주면 주어진 주소 값을 text segment의 시작 주소(starting address)로 사용하게 된다.
12. 다시 한번 11에서 생성된 이미지를 .note 및 .comment ELF section을 제거한 “raw binary” 이미지 파일인 bvmlinux.out을 혹은 vmlinux.out을 만들어낸다.
13. 마지막 과정으로 이전에 만들어진 모든 것들(bbootsect, bsetup, bvmlinux.out, 혹은 bootsect, setup, vmlinux.out)을 bzImage(혹은 zImage)로 묶어 주는 단계로 ~/arch/i386/boot 디렉토리의 tool/build를 이용해서 연결 시켜준다. 그리고, bootsector에 정의된 setup_sects와 root_dev 변수를 쓰게 된다. tool/build는 disk의 image를 만들어주는 구실을 하게 된다.

위와 같은 과정에서 bootsect.S는 boot sector 역할을 하게 됨으로 반드시 512 bytes의 크기를 가지게 되며, setup의 크기는 대략 4 sector(1 sector = 512 bytes)를 차지하도록 만든다.

이상에서 우리는 리눅스 커널이 어떤 과정을 밟아 가면서 생성되는지를 살펴보았다. 나중에 커널의 부팅에 대해서 관련된 코드들을 살펴보게 될 때, 좀더 자세히 보도록 하자.

² Object file 내에서 함수나 기타 변수를 access하는 것을 재배치 가능하도록 만든다.(즉, 고정된 주소로 만들지 않는다는 것을 말한다.)

1.7. Kernel의 특징

커널에는 크게 두 가지 종류가 있으며, monolithic 커널과 micro 커널이 그것이다. Monolithic 커널이란 커널이 사용자에게 서비스(service)를 제공하기 위한 기능을 전부 하나의 구조에 묶고 있는 것을 말하며, Micro 커널이란 운영체제의 핵심적인 기능만을 커널에 넣고, 나머지를 서비스 프로세스(service process)의 형태로 사용자 어플리케이션처럼 동작시키는 구조를 가지는 것이다. [그림6]은 두 가지 커널의 구조를 간단히 보여준다.

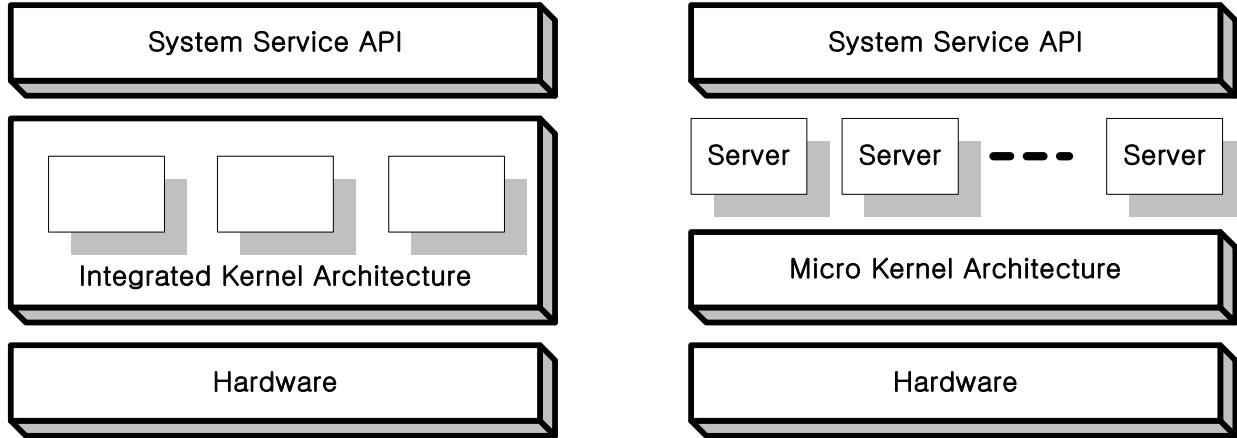


그림 6. Integrated (Monolithic) Architecture VS. Micro Kernel Architecture

Micro 커널의 예로는 Mach가 있으며, 리눅스 커널은 Monolithic한 구조를 가지고 있다. 이들의 장점 및 단점은 Monolithic 커널에서는 속도는 좋으나 업그레이드가 어렵다는 것이며, Micro 커널은 속도면에서는 한번 더 거치는 것이 있기에 상대적으로 느리지만 upgrade가 용이하다는 점이다. 하지만, 리눅스는 Monolithic한 구조를 가지지만 커널 모듈이란 것을 지원함으로써 그 확장성이나 모듈성을 증가 시켰다. 즉, 새로운 기능의 추가가 용이하다는 점인데, 특정의 기능을 하나의 모듈로 구현해서 커널에 넣을 수 있다는 것이다. 이는 디바이스 드라이버(device driver)를 개발하는데 있어서 편의를 제공하는데, 커널의 새로운 빌드(build)없이 테스트를 할 수 있다는 점이 Monolithic 커널에서는 보이지 않는 점이다. 또한 커널의 크기를 필요 없는 부분을 사용자 프로세스의 형태로 간직해서 필요한 때마다 가져와서 사용하게 됨으로 커널 자체의 크기도 줄일 수 있어서 좋다. 2.4 버전의 커널에서는 커널 데몬(daemon)형태를 바꾸어, 커널 내부에서 모듈에 대한 처리를 하도록 하고 있다. 나중에 가서 커널이 Module을 지원하는 기능에 대해서 좀더 자세히 볼 것이다.

리눅스는 또한 여러 가지의 파일 시스템(filesystem)에 대해서 지원한다. [그림4]의 커널 소스트리에서 알 수 있듯이 지원하는 파일시스템은 윈도우에서 쓰는 파일 시스템부터 Solaris에서 쓰는 것까지 다양하다. 이러한 파일 시스템을 일관된 인터페이스(interface)를 통해서 지원한다는 점이 중요하다. 나중에 이 부분에 대해서 설명할 때 자세히 들여다보도록 하자.

리눅스는 멀티테스킹(multitasking)을 위해서 만들어 졌으며, 멀티유저(multiuser)로 사용되는 서버(server)급의 운영체제이기에, 많은 사용자들이 동시에 접속해서 개개의 사용자가 다양한 응용프로그램을 동시에 여러 가지를 실행 시킬 수 있다. 또한 커널 쓰레드(thread)를 사용하고 있으며, 사용자 어플리케이션 수준(user application level)에서의 pthread를 라이브러리 형태로 지원한다. 프로세스들간에는 메모리를 공유(share)할 수 있고, 프로세스의 메모리는 요구 페이징(demand-paging) 원칙하에 커널로부터 할당 받을 수 있다. 물론 이런 기능들을 다 제공하기 위해서 커널이 자료구조를 유지하는데 드는 비용(cost)이 있다.

이외에도 많은 특징이랄 수 있는 것들이 있겠으나, 차츰 가면서 설명하도록 하겠다. 지금까지 나온 용어만 가지고도 머리가 아파질 수 있기 때문에 여기서 리눅스 커널의 특징에 대한 것은 마무리하도록 한다.

Linux Kernel Internal & Embedded Linux

앞으로는 커널의 각 부분들에 대해서 차근차근 이야기해 나가도록 하겠다.

2. Process Management

프로세스란 실행중인 프로그램이다. 즉, 프로그램의 메모리 내 image에 해당한다. 모든 프로그램은 실행되기 위해선 메모리로 loading되어야 하며, 운영체제는 이러한 프로세스를 관리하기 위한 정보를 내부에 유지한다.

리눅스에서는 *task_struct*라는 구조체를 이용해서 이러한 프로세스를 관리하며, 이 구조체에 프로세스를 위한 모든 정보를 포함하고 있다. 따라서 이번 장에서는 *task_struct* 구조체를 중심으로 프로세스 및 thread에 대해서 알아보기로 한다.³

2.1. task_struct 구조체

task_struct 구조체는 리눅스에서 프로세스가 실행되기 위한 모든 정보를 다 담고 있어야 한다. 프로세스는 생성과 실행, 대기 등을 거치면서 메모리에 있기도 하고, 스왑(swap) 영역에 있기도 하며, 또한 다른 프로세스들로 인해서 멈추기도 한다. 프로세스는 디스크 상에 있는 파일도 접근하게 되며, 자신의 고유한 ID값을 가지고 있기도 하다. 즉, 이러한 모든 것을 *task_struct*는 나타낼 수 있어야 한다는 점이다. 그럼, 이젠 차례로 그 *task_struct*의 필드(field)들을 살펴보기로 하자. 여기서 논의하지 않은 필드들에 대해서는 이하에서 다른 설명과 같이 볼 수 있을 것이다.

```
struct task_struct {
    /*
     * offsets of these are hardcoded elsewhere - touch with care
     */
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    unsigned long flags; /* per process flags, defined below */
    int sigpending;
    mm_segment_t addr_limit;

    volatile long need_resched;
    unsigned long ptrace;

    int lock_depth; /* Lock depth */
    ...
}
```

가장 먼저 나오는 것이 프로세스의 상태를 나타내는 *state*이다. 이것은 프로세스의 상태에 대해서 논의하게 될 때 볼 것이다. *flag*은 현재 시스템의 상태를 나타내는 값으로 아래와 같은 값을 가질 수 있다.

이름	값	설명
PF_ALIGNWARN	0x00000001	Alignment 경고 메시지를 출력하라.
PF_STARTING	0x00000002	현재 프로세스가 생성중이다.
PF_EXITING	0x00000004	현재 프로세스가 멈추고 있다.
PF_FORKNOEXEC	0x00000040	fork()되었지만 아직 exec()를 호출하지는 않았다.
PF_SUPPERPRIV	0x00000100	super user 권한으로 사용되고 있다.
PF_DUMPCORE	0x00000200	core dump되었다.
PF_SIGNALED	0x00000400	signal에 의해서 kill되었다.
PF_MEMALLOC	0x00000800	메모리를 할당하고 있다.
PF_VFORK	0x00001000	mm_release()시에 부모 프로세스를 깨워라.
PF_USEDFPU	0x00100000	Floating point unit을 프로세스가 현재 quantum ⁴ 동안에 사용했다.

³ 리눅스에서의 프로세스는 테스크(task)에 해당한다. 쓰레드(Thread) 역시 하나의 프로세스처럼 관리가 되며, 쓰레드를 생성한 프로세스와 정보를 공유한다. 따라서, 프로세스와 테스크를 같은 개념으로 이해해도 상관없으리라 생각한다.

표 1. task_struct의 flag필드의 값.

또한 프로세스에 대해서 현재 전달되기를 원하는 시그널(signal)들을 가리키는 값과, 사용하는 주소공간에 대한 한계를 나타내는 필드들이 온다. 여기서 잠시 커널과 사용자 프로세스간에 사용할 수 있는 주소공간을 말하자면, 사용자 프로세스는 마치 현재의 물리적인 메모리(physical memory)가 4Gigabytes인 것처럼 인식하기에 마지막의 1Gigabyte는 커널이 사용한다고 가정하고 그 나머지 3Gigabyte가 자신에게 할당되었다고 생각한다.

만약 현재의 프로세스가 커널 모드에서 진행 중, 스케줄링 요구를 하게 된다면 이를 나타내는 need_resched 필드를 설정하게 된다. 이 경우 나중에 커널 모드에서 사용자 모드로의 전환시 새로운 프로세스가 선택되어 실행 상태로 들어가게 된다.

ptrace필드는 디버깅(debugging)할 목적으로 사용되며, 아래와 같은 값을 가질 수 있다.

이름	값	설명
PT_PTRACED	0x00000001	현재 프로세스가 trace되고 있다.
PT_TRACESYS	0x00000002	현재 프로세스가 trace되고 있으며, 다음 번 시스템 콜까지 진행하라.
PT_DTRACE	0x00000004	지연된 trace를 하고 있다.
PT_TRACESYSGOOD	0x00000008	현재 프로세스가 trace되고 있으며, 시스템 콜이 예러가 없을 때까지 진행하라.

표 2. task_struct의 ptrace필드의 값.

lock_depth 필드는 현재 프로세스가 어디에서 lock이 걸렸는지를 나타내는 필드이다.

```
...
struct mm_struct *mm;
int has_cpu, processor;
unsigned long cpus_allowed;
struct list_head run_list;
unsigned long sleep_time;

struct task_struct *next_task, *prev_task;
struct mm_struct *active_mm;
...
```

프로세스가 사용하는 메모리 주소공간에 대한 기술(description)과 어느 CPU에서 실행되고 있는지, 그리고, 현재 실행 가능한 프로세스들의 리스트들에 대한 포인터를 가진다. 그리고, 현재 프로세스가 수면(sleep)상태에 있다면 얼마나 그 상태에 있었는지에 대한 정보를 나타내는 필드와 프로세스의 연결 list를 만드는 필드, 현재 활동 상태인 메모리 주소공간에 대한 기술자를 가진다.

```
...
struct linux_binfmt *binfmt;
int exit_code, exit_signal;
int pdeath_signal; /* The signal sent when the parent dies */
/* ??? */
unsigned long personality;
int dumpable:1;
int did_exec:1;
...
/* PID hash table linkage. */
struct task_struct *pidhash_next;
struct task_struct **pidhash_pprev;
```

⁴ 프로세스에게 한번에 실행될 수 있는 최대 시간 값을 말한다.

...

binfmt는 리눅스에서 지원하는 이진 파일의 포맷(format)을 명시하는 것으로 현재의 프로세스가 어떤 이진 파일 포맷을 실행하고 있는지를 나타낸다. 이에 따라서 실행을 위해서 필요한 모듈(module)이 달라지게 된다. 즉, 파일의 내용 중 일부를 읽어서, 이것을 실행할 수 있는 모듈을 적재한 다음, 이 모듈을 이용해서 실행하도록 하기 때문이다.

exit_code과 exit_signal은 프로세스의 종료 시 돌려줄 값과 보내야 할 시그널(signal)을 가진다. Personality 필드는 현재의 프로세스가 실행되기 위해서 에뮬레이션(emulation)할 시스템을 나타내며, `~/include/linux/personality.h`에서 정의된 값을 가진다. 아래와 같다.

```
#define STICKY_TIMEOUTS          0x4000000
#define WHOLE_SECONDS            0x2000000
#define ADDR_LIMIT_32BIT         0x0800000

#define PER_MASK                 (0x00ff)
#define PER_LINUX                (0x0000)
#define PER_LINUX_32BIT          (0x0000 | ADDR_LIMIT_32BIT)
#define PER_SVR4                 (0x0001 | STICKY_TIMEOUTS)
#define PER_SVR3                 (0x0002 | STICKY_TIMEOUTS)
#define PER_SCOSVR3              (0x0003 | STICKY_TIMEOUTS | WHOLE_SECONDS)
#define PER_WYSEV386              (0x0004 | STICKY_TIMEOUTS)
#define PER_ISCR4                (0x0005 | STICKY_TIMEOUTS)
#define PER_BSD                  (0x0006)
#define PER_SUNOS                (PER_BSD | STICKY_TIMEOUTS)
#define PER_XENIX                (0x0007 | STICKY_TIMEOUTS)
#define PER_LINUX32               (0x0008)
#define PER_IRIX32                (0x0009 | STICKY_TIMEOUTS) /* IRIX5 32-bit */
#define PER_IRIXN32               (0x000a | STICKY_TIMEOUTS) /* IRIX6 new 32-bit */
#define PER_IRIX64                (0x000b | STICKY_TIMEOUTS) /* IRIX6 64-bit */
#define PER_RISCOS               (0x000c)
#define PER_SOLARIS              (0x000d | STICKY_TIMEOUTS)
```

표 3. personality값.

이것은 시스템 콜 `personality()`를 이용해서, 현재 프로세스의 `exec_domain`값을 바꿔주는 것으로 가능하다. `dumpable`은 현재 프로세스를 덤프(dump)할 수 있는지를 나타내며, `did_exec`는 `exec()` 시스템 콜을 호출했는지를 설정할 수 있다. PID를 가지고 찾고자 하는 프로세스를 쉽게 알 수 있는 hash 테이블에 대한 포인터는 `pidhash_next`와 `pidhash_prev`로 나타낸다. 프로세스는 생성시 PID를 부여 받게 되며, 이를 이용해서 프로세스의 검색을 용이하게 하는 해쉬(hash) 테이블에 놓이게 된다.

```
wait_queue_head_t wait_chldexit;
struct semaphore *vfork_sem;
unsigned long rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
struct tms times;
unsigned long start_time;
long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
```

`wait_chldexit`는 자식 프로세스가 종료할 때까지 기다리는 `wait queue`를 연결하기 위해서 사용한다. `vfork_sem`은 `vfork()`를 호출 했을 때 사용되는 세마포어 값이다. `rt_priority`값은 프로세스에 주어진 정적인 우선 순위를 나타내며, `it_real_value`, `it_prof_value`, `it_virt_value` 및 `it_real_incr`, `it_prof_incr`, `it_virt_incr` 등은 타임머(timer)의 사용과 관련된 값으로, 앞의 3값은 타임머가 작동하기 전에 얼마나 시간이 경과 해야 할지를 나타내는 tick값이며, 뒤의 3개는 타임머를 초기화 하는데 사용될 값이다. 또한 `real_timer`는 실시간(real-time) 인터벌(interval) 타임머의 구현에 사용된다.

`times`는 프로세스의 회계(accounting)에 사용되며, `start_time`은 프로세스가 실행된 시점을, `per_cpu_otime`과 `per_cpu_stime`은 각각 CPU마다 실행된 사용자 타임과 시스템 타임을 가진다.

```
unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
int swappable:1;
```

`min_flt`, `maj_flt`,은 메모리 트랩(trap)이 발생한 횟수를 나타내는 것으로 각각이 페이지(page)의 로드(load)가 없이 처리된 횟수와 디스크로부터 하나의 페이지를 로드하는 동안에 처리된 수를 나타낸다. `cmin_flt`과 `cmaj_flt`은 자식 프로세스에 대한 경우를 나타낸다. `nswap`는 프로세스의 페이지가 스웨프 공간에 들어있는 수를, `cnswap`은 자식 프로세스의 페이지가 스웨프 공간에 들어가 있는 수를 각각 나타낸다. `swappable`은 프로세스가 스웨프될 수 있는 가를 가리키는 값이다.

```
uid_t uid,euid,suid,fsuid;
gid_t gid,egid,sgid,fsgid;
int ngroups;
gid_t groups[NGROUPS];
kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
int keep_capabilities:1;
struct user_struct *user;
```

프로세스의 인증(credential)을 나타내는 값으로 `uid`, `euid`, `suid`, `fsuid`, `gid_egid`, `sgid`, `fsgid`가 있다. 이 값을 프로세스의 증명서 역할을 하는 것으로 접근 제어에 사용된다. 각각은 다음과 같다.

- `uid` : 사용자 ID값.(User ID)
- `euid` : 유효 사용자 값.(Effective User ID)
- `suid` : 보관된 사용자 ID값.(Saved User ID)
- `fsuid` : 파일 시스템 사용자 ID값.(File System User ID)
- `gid` : 그룹 ID값.(Group ID)
- `egid` : 유효 그룹 ID값.(Effective Group ID)
- `sgid` : 보관된 그룹 ID값.(Saved Group ID)
- `fsgid` : 파일 시스템 그룹 ID값.(File System Group ID)

먼저 하나의 프로세스를 생성하게 되면, 사용자의 ID와 사용자가 속한 그룹의 ID가 프로세스에 등록된다. 만약 사용자가 `super user`의 권한으로 어떤 실행을 해야 할 경우에는 실행하고자 하는 프로그램의 소유자 ID값을 가지도록 하는 것이다. 이와 같은 일을 해주도록 하는 시스템 콜이 `setuid()`이다. 이 시스템 콜은 현재 실행되는 프로세스의 `euid`값을 바꾸어준다. 그룹일 경우도 마찬가지이다. `fsuid`과 `fsgid`는 NFS를 사용하게 될 때 적용되는 것으로 `setfsuid()`나 `setfsgid()`로 바꾸어 줄 수 있다. `ngroup`은 프로세스가 속한 그룹이 몇 개인가를, 그리고, `groups`는 그 그룹을 나타낸다. `cap_effective`와 `cap_inheritable`, `cap_permitted`는 현재 프로세스가 가지는 capability를 나타내는 필드이며, `keep_capability`는 현재 프로세스가 어떤 capability를 가지는지를 표시한다. `user` 필드는 사용자와 관련된 정보에서 얼마나 많은 프로세스를 사용자가 가지고 있는지, 얼마나 많은 열린 파일들을 가지는지 등을 나타낸다.

```
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
```

`rlim`은 프로세스의 자원 사용현황을 보여주며, `used_math`은 프로세스가 수치보조 프로세스를 사용했는지의 여부를 알려준다. `comm`은 현재 실행중인 프로세스의 명을 지칭한다. 주로 파일이름이 된다.

```
int link_count;
struct tty_struct *tty; /* NULL if no tty */
unsigned int locks; /* How many file locks are being held */
```

link_count는 실행하기 위해서 파일이름의 검색에 사용했던 심볼릭(symbolic) 링크(link)의 수를, tty는 프로세스에 할당된 터미널(terminal)을 나타내는 데이터 구조체에 대한 포인터를, locks는 파일에 락(lock)을 얼마나 걸었는가를 나타낸다.

```
struct sem_undo *semundo;
struct sem_queue *semsleeping;
struct thread_struct thread;
struct fs_struct *fs;
struct files_struct *files;
```

semundo와 semsleeping은 프로세스가 세마포어 자원을 사용하려 할 때 사용된다. 주로 세마포어는 프로세스간의 동기화에 사용되며, 만약 하나의 프로세스가 자원을 점유하길 원한다면 세마포어를 가져야 한다. 그렇지 않다면 세마포어를 가진 프로세스가 그 자원의 사용을 끝낼 때까지 기다려서 다시 시도해야 한다. semsleeping은 현재 수면(sleep)상태에 있는 프로세스들의 세마포어 대기 큐(wait queue)를 나타낸다. semundo는 세마포어의 리스트(list)를 나타낸다. 자세한 설명은 나중에 IPC(Interprocess Communication)에서 다시 보게 될 것이다.

thread는 프로세서(processor)의 레지스터(register)값을 보관하며, fs는 파일에 대한 접근시 확인할 정보를 가진다. files는 프로세스에 의해서 열려진 파일 기술자(file descriptor)들에 대한 포인터이다.

```
spinlock_t sigmask_lock; /* Protects signal and blocked */
struct signal_struct *sig;
sigset_t blocked;
struct sigpending pending;
unsigned long sas_ss_sp;
size_t sas_ss_size;
int (*notifier)(void *priv);
void *notifier_data;
sigset_t *notifier_mask;
```

프로세스간의 통신 방법중의 하나가 시그널(signal)이다. 시그널은 주로 프로세스에게 어떤 사건(event)가 발생했음을 알려주는 한가지 방법으로 프로세스들은 만약 이러한 시그널을 받았을 경우 특정의 일을 수행한다던가 무시, 혹은 시스템에 그 처리를 맡길 수 있다.

sigmask_lock은 시그널과 관련된 연산에 적용되는 락(lock)이며, sig는 시그널 구조체를 가르키는 포인터로, 시그널 구조체는 시그널에 할당된 action의 기술자(descriptor)를 가진다. blocked는 매스킹(masking)된 시그널을 나타내며, pending은 처리를 기다리고 있는 시그널에 대한 리스트(list)를 나타낸다. 블록킹(blocking)된 시그널은 프로세스로 전달되지 않으며, 처리를 기다리고 있는 시그널은 나중에 프로세스가 실행되는 시점에서 처리를 요한다.

sas_ss_sp는 시그널과 관련된 스택(stack)을 가리키며, sas_ss_size는 스택의 크기를 가진다. notifier는 기다리고 있던 시그널이 도착했음을 알려주는 함수며, notifier_data를 넘겨받아서 처리하게 되고, 관심을 가지는 시그널은 notifier_mask로 알 수 있다.

```
u32 parent_exec_id;
u32 self_exec_id;
spinlock_t alloc_lock;
};
```

parent_exec_id와 self_exec_id는 각각 부모의 exec_id와 자신의 exec_id를 가진다. fork()시에는 부모의 self_exec_id를 생성되는 프로세스의 parent_exec_id로 받게 되며, exit()시에 이곳에 저장된 값을 확인해서 어떤 시그널들을 전달해줄지를 결정하게 된다.

alloc_lock은 메모리 및 파일, 혹은 파일 시스템이나 터미널에 대한 할당이나 제거 시에 사용되는 락(lock)이다.

이상에서 우리는 task_struct에 대한 대략적인 설명을 보았다. 여기서 설명하지 않은 필드들은 앞으로 관련된 부분에서 설명해 나갈 것이며, 또한 리눅스의 다른 부분들을 설명할 때 또 보게 될 것이다.

이처럼 하나의 프로세스는 많은 정보들을 각각이 가지고 있어야 하며, 커널은 이러한 정보를 바탕으로 자원들을 프로세스간에서 관리할 수 있게 되는 것이다.

2.2. Process들의 관계

리눅스에서의 모든 프로세스(단, init 프로세스를 제외)는 `fork()`에 의해서 생성되며, 부모(parent) 프로세스를 가진다. 즉, 새로이 생성된 프로세스는 자식(child) 프로세스가 되며, 부모를 가리키는 pointer를 `task_struct`에 가진다. 또한, 만약 부모가 새로운 자식 프로세스를 생성하게 되면, 새로이 생성된 자식 프로세스를 가리키는 포인터가 자식 프로세스들 간에도 생성된다. [그림7]은 이러한 리눅스 프로세스들간의 관계를 보여준다.

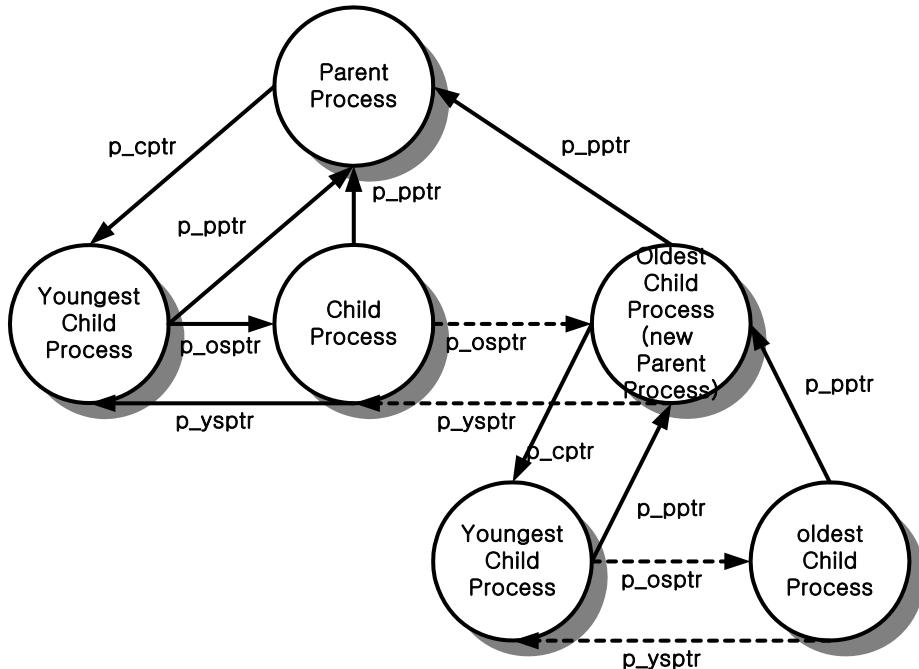


그림 7. Process Relation for Linux

자식 프로세스들간에는 생성된 순서에 따라서 `p_osptr`과 `p_ysptr`이 자신보다 오래된 것과 자신보다 늦게 생성된 자식 프로세스를 가리킨다. 또한 모든 자식 프로세스들은 `p_pptr`로 자신의 부모(parent) 프로세스를 가리키도록 한다.

또한 모든 프로세스들은 `init_task`라는 전역 변수에 의해서 2중으로 연결된 구조를 가지게 되며, 이 연결 list를 이용해서 각 프로세스들에 대한 접근이 가능하다. 때때로 이렇게 연결된 전체 list를 다 훑어 보는 것이 필요하게 되며, 예로는 priority를 새로이 계산하거나 현재 system 내의 모든 프로세스를 나타낼 때 사용되어질 수 있다.

Init 프로세스는 system이 booting 할 때 인위적으로 만들어지며, system이 기동중인 상황에서는 종료되지 않는 프로세스이다. init 프로세스는 kernel thread로서 생성되어지며, 프로세스 ID로 0을 부여 받는다.

```
kernel_thread(init, NULL, 0);
```

이 프로세스의 source는 `/init/main.c`에 있으며, 기본적인 초기화 작업을 수행한 후, `/dev/console`을 `open`한 다음, `disk`에서 `init` 프로그램을 읽어와서 실행하는 역할을 한다. 이후에는 단순히 CPU를 소모하는 역할만을 하게 된다.

```
cpu_idle();
```

이와 같은 것이 필요한 이유는 실행 가능한 프로세스가 존재하지 않을 때를 대비한 것이라고 보면 될 것이다. 즉, 아무 일도 하지 않으면서 어떤 event가 생기기를 기다리던지 아니면, 어떤 프로세스가

실행가능하게 되길 기다리는 것이다. 만약 새로운 프로세스가 실행 가능하게 되면, 그 프로세스로 제어가 넘어가서 실행될 것이다.

프로세스들은 여러 개가 끝여서 프로세스 그룹을 이룬다. 프로세스는 고유한 ID로 pid를 운영체제로부터 할당 받게 되며, 속한 그룹의 값으로는 pgrp를 가진다. 또한 프로세스는 session에 속하게 되며, session에는 리더(leader)가 되는 프로세스가 존재하게 된다.

```
pid_t pid;
pid_t pgrp;
pid_t tty_old_pgrp;
pid_t session;
pid_t tgid;
int leader;
```

코드 1. task_struct 구조체의 일부.

프로세스 그룹(process group)이란 여러 프로세스를 끝여서 만든 것으로 커널에서 그룹에 속한 프로세스들에게 어떤 작용(action)을 가하기 위한 것이다. 일반적으로 프로세스들은 pgrp값을 부모로부터 상속(inherit)하며, 각각의 그룹에는 로그인(login) 터미널(terminal)⁵을 하나씩 가지고 있고 /dev/tty file에 관련 짓는다. 나아가 여러 개의 그룹을 끝여서 세션(session)을 만들며, 모든 세션은 리더를 가진다. 리더가 되는 프로세스는 자신의 pid로 pgid과 session값을 초기화하게 된다.. [그림8]을 참고하기 바란다.

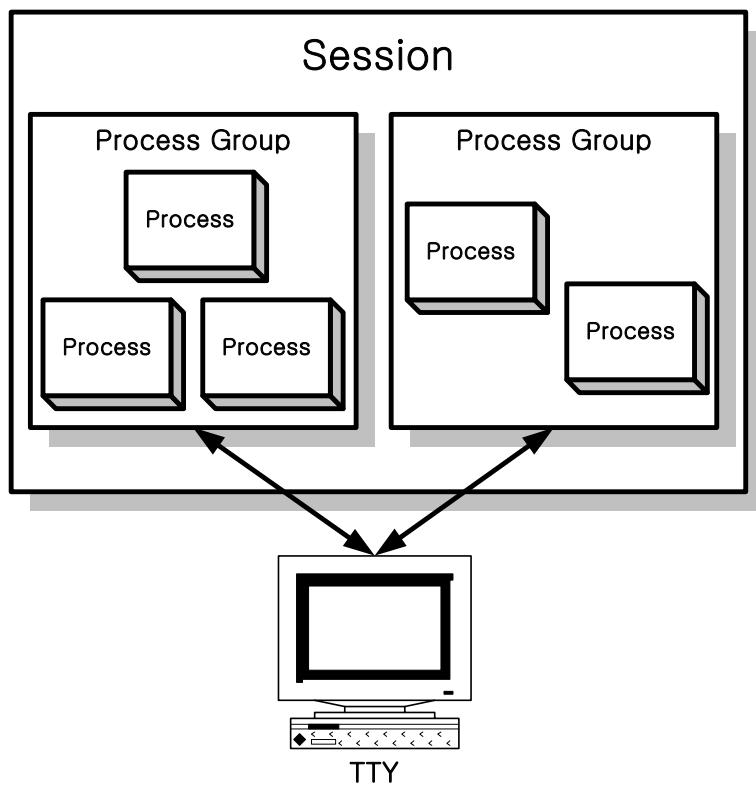


그림 8. Process Session, Group와 TTY

⁵ 이것을 controlling terminal이라고 한다.

tty_old_pgrp는 세션의 리더가 자신과 연관된 터미널을 제거할 때, 터미널의 세션과 관련된 프로세스 그룹의 리더들이 자신이 어떤 터미널에 연관되어 있었느냐를 기억하기 위해서 사용한다. tgid는 쓰레드(thread)의 그룹 ID값을 나타낸다.

2.3. Process의 상태

프로세스는 생성되는 순간에서부터 여러 상태를 거치다가 종료하게 된다. 이러한 프로세스의 상태 정보는 프로세스를 관리하는 커널에서는 중요한 정보가 되며, scheduler는 이러한 정보를 바탕으로 실행할 프로세스를 선택하던가, 아니면 어떠한 event를 알려줄 것인가를 결정할 수 있게 된다. 리눅스에서는 프로세스의 상태를 다음과 같이 정의하고 있다.

```
#define TASK_RUNNING
#define TASK_INTERRUPTIBLE
#define TASK_UNINTERRUPTIBLE
#define TASK_ZOMBIE
#define TASK_STOPPED
#define TASK_EXCLUSIVE
```

표 4. Process의 상태

일반적인 process의 상태 전이는 [그림9]와 같다.

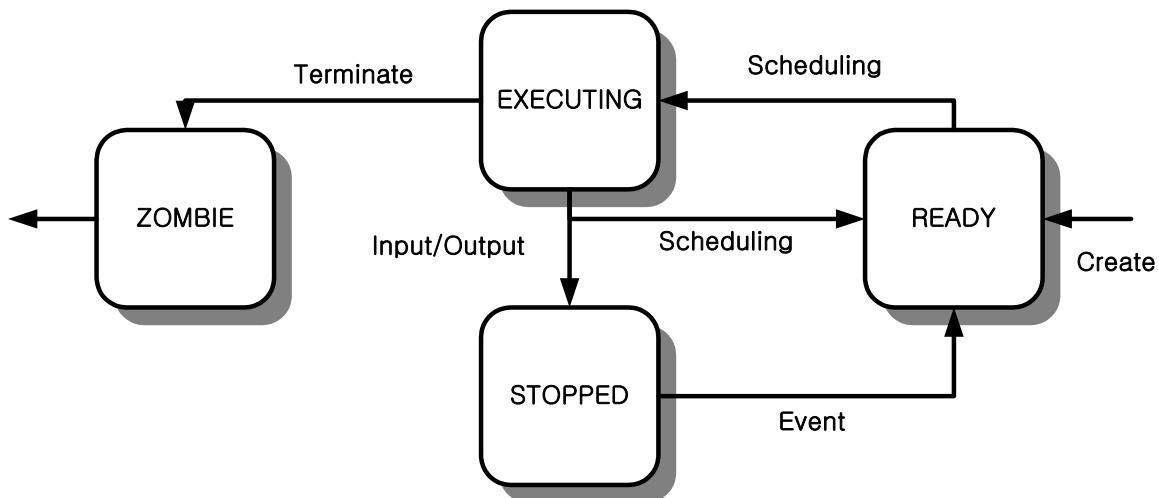


그림 9. Process Transition Diagram For Linux

Process는 생성되면 ready 상태로 가게 되며, 이곳에서 scheduling되기를 기다리게 된다. 그리고, scheduling이 되어서 실행되면 executing상태로 가게 되며, input이나 output을 요구하게 되면, stop 상태가 될 수 있다. 그리고, 다시 요구한 것을 마치게 되면 ready 상태에 놓이게 되며, 이곳에서 다시 scheduling을 기다리게 된다. 실행을 다 마친 process는 zombie상태로 가며, 이곳에서 자신을 생성한 process가 wait system call을 호출할 때까지 기다리게 된다. Process를 생성한 process가 wait system call을 호출하게 되면, 생성된 process는 자신의 exit status정보와 다른 정보들을 생성한 process로 돌려주고 자신의 모든 resource들을 다시 반환하고 실행을 마치게 된다.

Zombie process란 process가 종료할 때 부모(parent process)에 의해서 지워지기 전의 상태에 있는 process이다. 이 상태에서 가지는 process의 자원은 일반적인 Unix에서는 proc structure밖에 없다. 즉, 이미 memory를 해제한 상태이며, 부모에 알려줄 정보만을 가지고 있는 상태이다. 가지는 정보는 exit status정보와 자원 사용 정보가 있겠으며, 부모 process는 wait를 호출해서 이것을 가져오게 된다. 만약 자식(child) process가 종료하기 전에 부모 process가 먼저 종료하게 되면, 자식 process를 init process가 넘겨받게 되며, init가 wait를 호출해서 자식 process가 가진 proc구조체를 해제하게 된다.

Interruptible과 uninterruptible에 대해서는 나중에 task queue와 관련된 것에 대해서 논의 할 때 살펴볼 것이다. 여기서는 간단히 task가 현재 signal에 대해서 interrupt될 수 있느냐 없느냐를 구분하는 것이라고만 알아 두도록 하자.

2.4. Process의 Context

Context란 무엇인가? 보통의 우리말 표현을 보면 문맥이라는 식으로 번역을 한다. 하지만, 이 말은 적절한 표현 같지가 않기에 환경이라는 말로 바꿔서 보도록 하겠다. 일반적으로 context란 하나의 process가 실행되는 snapshot이라고 보는 편이 좋다.

이러한 process의 context에 들어갈 수 있는 내용으로는 다음과 같은 것이 있겠다.

- 사용자 주소 공간 : text와 data, stack 및 공유메모리(shared memory) 등이 있다.
- 제어 정보 : process에 대한 정보를 kernel에서 관리하기 위해서 있는 것으로, 보통의 unix의 경우에는 u area와 proc(리눅스에서는 task_struct)을 들 수 있겠다.
- credentials : 이것은 process와 연관된 보안을 유지하기 위해서 필요한 정보로 user ID나 group ID 등이 있다.
- 환경변수 : 프로그램이 수행되기 위해서 필요한 것들을 주로 command라인 상에서 변수에 값을 입력하는 방식으로 전달된다. 이러한 값들은 부모 process로부터 넘겨받아서 사용하는 경우가 대부분이다.
- Hardware context : hardware register들에 대한 내용으로 general purpose register들과 system에서 사용하는 특별한 register들이다. 대표적으로 PC(Program Counter, 혹은 Instruction Pointer : IP), stack pointer나 PSW(Processor status word, 혹은 x86의 EFLAGS), MMU(Memory Management Unit) register, FPU(Floating Point Unit) register들이 있겠다.

이들 중에서 새로운 process로 환경전환(context switch) 시에 필요한 것들로는 주로 hardware context이며, 새로운 process로의 전환이전에 PCB(Process Control Block)이라는 구조에 저장될 필요가 있다. 나중에 다시 scheduling이 되어서 실행되어야 할 경우 이곳에 있는 값들이 각각의 register로 새로이 읽혀 들어오게 된다. Process의 환경전환은 자주 있을 가능성이 있으므로 저장되는 양은 적을수록 좋고, 빠르게 일어나야 할 것이다. 새로운 process로의 실행 변경은 register들을 읽어서 복구한 후에 PC(Program Counter)로 long jump를 하면 될 것이다. 대부분의 경우 이러한 환경 전환 code들은 assembly로 되어 있으며, 이는 이해보다는 속도측면을 고려한 것이다.

또한 여기서 중요한 것은 interrupt가 발생했을 경우에 어떤 context하에서 실행 되는 가이다. Interrupt는 asynchronous하게 발생 되기 때문인데, interrupt하에서는 현재의 process가 요청한 결과에 대한 것인지, 아니면 어떤 process가 요청을 기다리고 있는지를 판단할 수가 없다. 따라서, system programmer의 입장에서는 interrupt시에도 자신의 process context를 가지고 interrupt를 처리하고 있다고 봐선 안 될 것이다. 리눅스의 경우 device driver를 작성할 때, interrupt handler를 설치하는 경우가 많은데, 이 경우에 유의하기 바란다.

2.5. Process의 Mode

Process는 크게 두 가지의 모드로 동작한다. 그것이 바로 User Mode와 Kernel Mode이다. User mode에서는 자신만의 address space만을 사용할 수 있으며, 다른 process의 주소 공간에 대한 access는 금지된다. 또한 특정의 instruction(Privileged instruction)은 수행할 수 없으며, kernel mode로의 전환을 위해서는 system call을 사용하여야 한다. 이러한 system call은 trap을 발생시키게 되며, kernel mode에서는 엄격히 정해진 규칙에 따라서 다른 process의 address로의 접근 및 privileged instruction을 사용할 수 있게 된다.

MS-DOS와 같은 경우에는 이러한 제한이 없지만, multi-user, multi-tasking을 할 수 있다는 말을 쓰는 모든 system은 전부 이러한 mode를 가지고 있다. CPU의 경우에도 privileged instruction을 사용하기 위해서 모드의 변경을 지원하는데, Intel의 경우에는 ring 0에서 ring 3까지 4개의 level을 가지고 있다. 리눅스 및 Windows NT의 경우 2개의 ring을 사용해서 이러한 mode를 나타낸다. ring0와 ring3이 이러한 목적으로 사용되는 실행 level이다.

User mode에서 사용되는 instruction들은 security를 보장해주지 못한다. 따라서, 이러한 mode에서 사용되는 프로그램들은 특정의 system call을 통해서만 system에 접근하게 되고, kernel에서 요청을 받아서 처리하게

됨으로 kernel의 bug는 아주 치명적일 수 있다. 적어도 kernel은 이러한 오류가 없도록 작성되었다는 것을 보장해야 할 것이다.

2.6. Process의 Scheduling

Process scheduling이란 CPU와 main memory를 process에 할당해 주는 것이다. 즉, 실행 가능한 상태에 있는 프로세스를 선택해서 실행시켜주는 것이다. 시스템에는 많은 프로세스들이 존재할 수 있으며, 이들을 적절히 선택해서 실행시켜주는 것은 중요한 것이며, 이러한 일을 하기 위해서 운영체제는 process와 연관된 우선순위(priority)를 부여한다. 스케줄링을 적용해서 실행할 프로세스를 찾는 역할을 하는 것이 scheduler이며, 가장 높은 우선순위를 가지는 프로세스가 선택되어 실행된다.

이러한 scheduling에 영향을 미치는 것들을 scheduling parameter라고 하며, 적용하는 scheduling 방법을 scheduling policy라고 한다. 크게 scheduling에 연관된 process들의 우선순위에는 real-time process scheduling class라는 것과, time-sharing process scheduling class가 있으며, real-time process가 time-sharing process보다 항상 높은 우선순위를 가지고 있다. 예를 들어 FreeBSD의 경우 scheduling priority의 0에서 31까지는 real-time process가, 그리고 32에서 63까지는 time-sharing process가 우선순위 값으로 가진다. 여기서는 물론 작은 값을 가지는 것이 높은 우선 순위를 가진다고 보기 때문이다.

실시간(real-time)이라는 것은 그러면 무엇인가? 실시간이란 어떤 연산이 반드시 어떤 주어진 시간 안에 끝이 나야 한다는 의미를 가지고 있다. 여기에는 soft real-time과 hard real-time으로 또 다시 분류될 수 있는 여지가 있으며, 대부분의 real-time class process들은 상용 시스템에서 완벽히 hard real-time을 지원해주고 있지는 못하다. 물론 real-time kernel로 나온 pSOS system같은 경우는 real-time성을 모두 지원한다고 볼 수 있겠다. 완전한 hard real-time보다는 soft real-time을 지원한다는 말은 반드시 어떤 주어진 시간 안에는 마치지 못하더라도 그 연산의 결과가 받아들일 만 하다고 여긴다는 말이다. 가령 video를 play할 경우 초당 30 frame을 decoding해야 하지만, 26 frame정도만 decoding해도 출력되어 보여지는 결과에는 크게 영향을 못 막는다는 것을 예로 들 수 있겠다. hard real-time과 같은 것은 주로 산업용의 장비에 들어가는 system으로 반드시 연산의 결과가 주어진 시간 안에 나와야 한다는 것이다.

리눅스에서는 실시간 프로세스를 위한 process scheduling class를 두고 있으며, 또한 normal process(time-sharing process)를 위한 class도 있다. 다음과 같이 구분된다.

- **SCHED_FIFO** : 우선순위의 변화가 불가능한 실시간 process들.
- **SCHED_RR** : 우선순위의 변화가 가능한 실시간 process들
- **SCHED_OTHER** : 일반적인 time-sharing process들

각각에 대해서 조금 더 자세히 설명하자면, SCHED_FIFO는 가장 높은 우선순위를 가지며, pre-empt될 수 없다. 즉, 다른 process들에 의해서 선점되지 않는다는 말이다. 하지만 만약 더 높은 우선 순위의 SCHED_FIFO process가 실행 대기 상태에 있거나, process가 event를 기다리고 있을 경우, 그리고, process가 자발적으로 `sched_yield()`를 호출한 경우에는 scheduler는 다른 process를 선택해서 실행하게 된다. SCHED_RR인 process들은 SCHED_FIFO와 유사하지만, time-slot이라는 것을 두고 그 time-slot 동안에 실행된다는 점에서 차이가 난다. 만약 time-slot이 다 끝나면, SCHED_FIFO이나 SCHED_RR인 process들 중에서 현재 process보다 우선순위가 같거나 더 높은 process를 scheduler가 선택해서 실행시켜준다. SCHED_OTHER는 ready상태인 real-time process들이 없는 경우에만 실행되며, 동적인 우선순위(dynamic priority)를 가진다. 이러한 동적인 우선순위는 `nice()`나 `setpriority()`를 통해서, 그리고, 운영체제가 계산한 값에 따라서 결정되며, 수행시간이 지남에 따라 우선순위가 감소하게 된다. 즉, 실행시간이 오래되면 오래될수록 우선 순위가 낮아진다는 말이다. 이렇게 하므로 해서 전체 process들이 기아(starvation)상태⁶를

⁶ 기아 상태(starvation)이란 프로세스가 더 높은 우선 순위의 프로세스들 때문에 계속적으로 실행이 되지 못하는 상태로 이러한 프로세스들이 많으면 많을수록 system의 성능이 떨어지는 것처럼 보여진다. 즉, 사용자의 입장에서 보면 자신이 실행한 프로그램이 연산의 결과를 알려주지 못한다고 느껴질 것이다. 이러한 상황을 배제하는 것이 또한 scheduling algorithm의 중요한 특성이다.

겪지 않도록 예방책을 만들어 두고 있다. 리눅스에서는 위와 같은 것을 표현하기 위해서 task_struct내에 다음과 같은 field들이 존재한다.

```
unsigned long policy; /* SCHED_FIFO, SCHED_RR, SCHED_OTHER */
unsigned long rt_priority;
```

그럼 이제부터는 리눅스에서 사용하는 scheduling algorithm을 살펴보도록 하자.

2.7. Linux Process Scheduling Algorithm

리눅스는 기본적으로 process의 실행을 clock cycle과 연관 시켜서 생각하고 있다. 즉, 특정한 값을 프로세스마다 할당한 다음, 이를 click tick interrupt에 따라서 감소시켜가면서 실행을 하도록 한다. 따라서, 오랫동안 수행된 process는 낮은 값을 가지게 될 것이며, 실행대기 중인 제일 높은 값을 가지는 process를 선택하는 것이 scheduler가 하는 일이다.

여기서 일반적인 Unix에 적용되는 multilevel feed-back queue에 대해서 잠시 알아보고 넘어가도록 하자. FreeBSD의 경우에 이와 같은 queue를 사용해서 process들을 scheduling하는데, 이는 실행할 process를 새로이 선택해야 할 경우 가장 높은 우선순위의 queue에서 그 프로세스를 선택하는 것이며, 실행이 되고 난 후, 다시 실행되기 위해서는 우선 순위가 재 계산되어 그 queue로 들어가게 되는 방식이다. [그림10]은 이것을 보여준다.

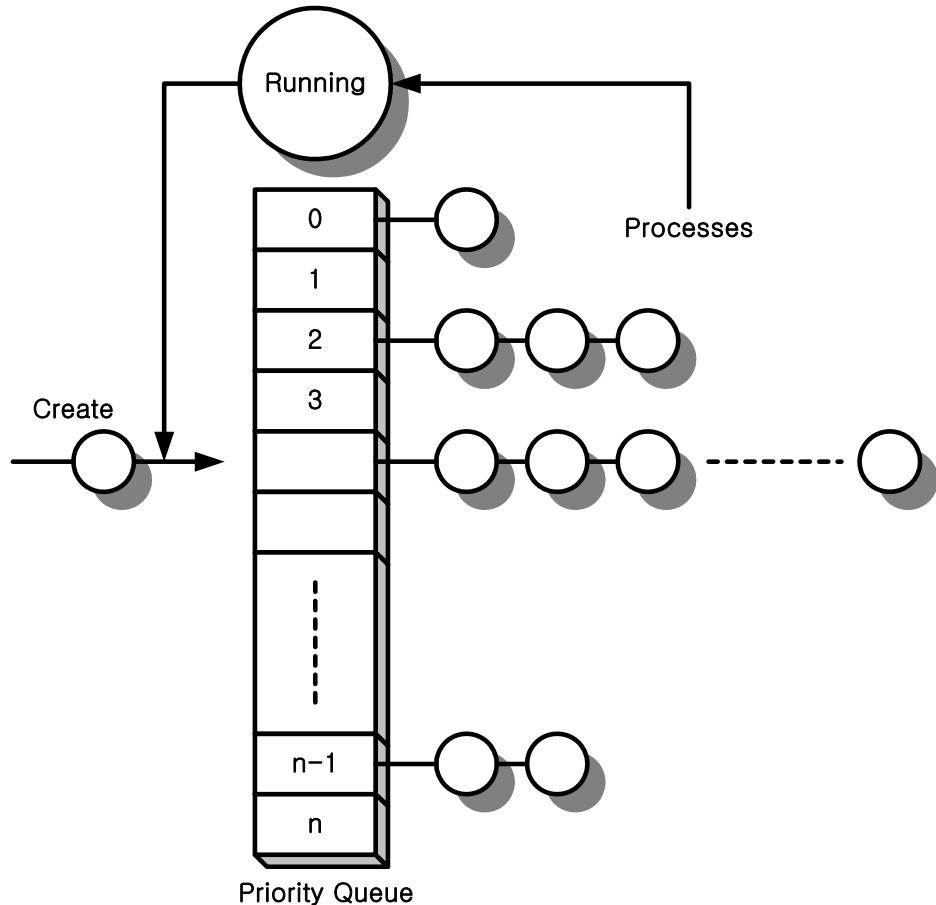


그림 10. Multi-level Feedback Queue

이러한 multi-level feedback queue를 사용하는 경우에는 시간의 경과에 따라서, process들은 자신이 CPU를 사용한 만큼씩 priority를 감소시켜 나가면서 priority queue에서 자리를 이동하게 된다. 따라서, CPU의 사용을 많이 하는 process일수록 priority가 낮아지게 됨으로, 다른 process가 실행 될 수 있는 기회를 만들어준다. FreeBSD에서는 priority에 따라서 낮은 priority값을 가지는 process가 더 높은 priority를 가지고 있다고 보며, 빠른 선택과 context switching을 위해서 위와 같은 부분을 assembly 언어로 처리하고

있다. 즉, priority queue에서 상위에서부터 실행 가능한 process가 있는지를 확인한 후에, queue의 앞부분에서 실행할 process를 가져와서 실행시켜준다. 실행이 끝난 process는 다시 priority를 계산한 후에, priority에 맞게 queue의 마지막으로 삽입된다.

그럼 이제는 리눅스에서 priority를 결정하는 방법에 대해서 보도록 하자. 리눅스는 전체 process를 선형적으로 찾아나가면서 priority를 계산하고, 그 중에서 실행 가능한 가장 높은 priority를 가지는 process를 선택해서 실행 시켜준다. 각각의 process는 priority와 연관된 field로 counter라는 field를 가지고 있으며, 이 값의 매 scheduling 요구마다 새롭게 계산된다. 실제적인 code를 보면서 설명하도록 하겠다. Code는 `~/kernel/sched.c`에서 따온 것이다.

```
asmlinkage void schedule(void)
{
    struct schedule_data * sched_data;
    struct task_struct *prev, *next, *p;
    struct list_head *tmp;
    int this_cpu, c;

    if (!current->active_mm) BUG();
    if (tq_scheduler)
        goto handle_tq_scheduler;
    ...

handle_tq_scheduler:
    /*
     * do not run the task queue with disabled interrupts,
     * cli() wouldn't work on SMP
     */
    sti();
    run_task_queue(&tq_scheduler);
    goto tq_scheduler_back;
```

코드 2. sched.c (1)

먼저 task queue에 등록된 모든 routine들을 처리한다. 이는 bottom half로 주어진 것으로 빠른 처리를 요하지는 않아서 interrupt handle에서 나누어진 부분들이다. scheduler에서는 process를 scheduling하기 전에 이렇게 등록된 routine들을 처리 함으로서 기다리고 있을지도 모를 process들을 실행 가능 상태로 만든다.

```
tq_scheduler_back:

    prev = current;
    this_cpu = prev->processor;

    if (in_interrupt())
        goto scheduling_in_interrupt;

    release_kernel_lock(prev, this_cpu);

    /* Do "administrative" work here while we don't hold any locks */
    if (softirq_active(this_cpu) & softirq_mask(this_cpu))
        goto handle_softirq;
    ...

handle_softirq:
    do_softirq();
    goto handle_softirq_back;
    ...

scheduling_in_interrupt:
    printk("Scheduling in interrupt\n");
    BUG();
```

```
return;
```

코드 3. sched.c (2)

Task queue에 등록된 routine들을 처리했다면, 이제는 interrupt에 대한 처리를 한다. 이 경우 interrupt중에 scheduling이 된 경우로 error로 처리한다. 그리고 나서, software interrupt가 현재 scheduling하고 있는 CPU에 대해서 있는가를 확인한 다음 software interrupt를 처리한다.

```
handle_softirq_back:
```

```
/*
 * 'sched_data' is protected by the fact that we can run
 * only one process per CPU.
 */
sched_data = & aligned_data[this_cpu].schedule_data;

spin_lock_irq(&runqueue_lock);

/* move an exhausted RR process to be last.. */
if (prev->policy == SCHED_RR)
    goto move_rr_last;
...
move_rr_last:
if (!prev->counter) {
    prev->counter = NICE_TO_TICKS(prev->nice);
    move_last_runqueue(prev);
}
goto move_rr_back;
```

코드 4. sched.c (3)

Software IRQ의 처리를 마치면 이제는 run queue상에 있는 process들에 대한 scheduling을 하게 된다. 가장 먼저 현재 process의 scheduling policy가 SCHED_RR(Round-robin)인 경우에 nice값을 counter로 변화시킨 후 run queue의 마지막으로 보낸다.

```
move_rr_back:
```

```
switch (prev->state & ~TASK_EXCLUSIVE) {
    case TASK_INTERRUPTIBLE:
        if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
        }
    default:
        del_from_runqueue(prev);
    case TASK_RUNNING:
}
```

코드 5. sched.c (4)

그리고 나서, process의 state에 따라서 만약 pending인 signal이 있다면, process의 state를 TASK_RUNNING으로 만들어주고, TASK_RUNNG인 상태라면 아무 일도 하지않으며, default로는 run queue에서 제거하는 일을 한다. 또한 process가 이미 scheduling처리가 되고 있으므로 re-rescheduling될 필요가 없다고 만들어준다.

```
repeat_schedule:
```

```

/*
 * Default process to select..
 */
next = idle_task(this_cpu);
c = -1000;
if (prev->state == TASK_RUNNING)
    goto still_running;
...
still_running:
c = goodness(prev, this_cpu, prev->active_mm);
next = prev;
goto still_running_back;

```

코드 6. sched.c (5)

현재의 CPU에서 idle task로 돌고 있는 process를 선택한 다음, process들의 우선 순위를 결정하는 계산에 들어간다. 이와 같은 계산은 *goodness()*에서 일어난다. 먼저 process가 가장 최근에 수행된 CPU와 현재의 CPU가 같을 경우, 그 process의 weight를 높여준다. 그리고, mm에 대한 advantage를 준 다음, nice값에 대한 조정을 한다⁷. 실시간 process들에 대해서는 1000을 더한 값을 advantage로 준다.

```

static inline int goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
{
    int weight;

    weight = -1;
    if (p->policy & SCHED_YIELD)
        goto out;

    /*
     * Non-RT process - normal case first.
     */
    if (p->policy == SCHED_OTHER) {
        weight = p->counter;
        if (!weight)
            goto out;

#define CONFIG_SMP
        /* Give a largish advantage to the same processor... */
        /* (this is equivalent to penalizing other processors) */
        if (p->processor == this_cpu)
            weight += PROC_CHANGE_PENALTY;
#endif

        /* .. and a slight advantage to the current MM */
        if (p->mm == this_mm || !p->mm)
            weight += 1;
        weight += 20 - p->nice;
        goto out;
    }
    weight = 1000 + p->rt_priority;
out:
    return weight;
}

```

코드 7. sched.c (5) – goodness() function

⁷ nice값은 process가 자신에게 직접 줄 수 있는 priority를 변동시키는 값이다.

이 계산에서 주어지는 weight값은 process가 다음에 선택되어 실행될 수 있느냐를 따지는 것으로, 큰 값을 가질수록 다음에 선택되어 실행될 가능성이 많다고 보면 된다.

```
still_running_back:
    list_for_each(tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p, this_cpu)) {
            int weight = goodness(p, this_cpu, prev->active_mm);
            if (weight > c)
                c = weight, next = p;
        }
    }

    /* Do we need to re-calculate counters? */
    if (!c)
        goto recalculate;
...
recalculate:
{
    struct task_struct *p;
    spin_unlock_irq(&runqueue_lock);
    read_lock(&tasklist_lock);
    for_each_task(p)
        p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
    read_unlock(&tasklist_lock);
    spin_lock_irq(&runqueue_lock);
}
goto repeat_schedule;
```

코드 8. sched.c (6)

각각의 process들에 대해서 우선 순위가 높은 process를 찾는 것이다. 전체 process를 뒤지면서, goodness()가 return하는 weight값이 가장 큰 것을 찾도록 한다. 만약 이렇게 해서 찾은 weight값이 0인 경우에는 각 process의 counter field를 갱신한 다음 새로운 계산에 들어간다.

```
sched_data->curr = next;
#endif CONFIG_SMP
next->has_cpu = 1;
next->processor = this_cpu;
#endif
spin_unlock_irq(&runqueue_lock);

if (prev == next)
    goto same_process;
...
same_process:
    reacquire_kernel_lock(current);
    if (current->need_resched)
        goto tq_scheduler_back;
#endif CONFIG_SMP
    sched_data->last_schedule = get_cycles();
#endif /* CONFIG_SMP */
```

코드 9. sched.c (7)

이전 가장 큰 계산된 goodness() function의 weight값을 지니는 process를 찾았다. 따라서, 다음 번에 수행할 process를 찾은 것이다. 따라서, 이 process로 진행하기 위한 정보들을 save하자. 만약 이전에 수행하던

process가 또다시 실행될 기회를 얻었다면, context switching할 필요가 없게 된다. 하지만 만약 같은 process가 선택되었다고 해도, process가 rescheduling을 필요로 한다면, 다른 process를 선택하도록 해준다. 그리고 나서, SMP machine인 경우에는 마지막에 scheduling이 일어난 시점을 저장해둔다.

```

kstat.context_swch++;
prepare_to_switch();
{
    struct mm_struct *mm = next->mm;
    struct mm_struct *oldmm = prev->active_mm;
    if (!mm) {
        if (next->active_mm) BUG();
        next->active_mm = oldmm;
        atomic_inc(&oldmm->mm_count);
        enter_lazy_tlb(oldmm, next, this_cpu);
    } else {
        if (next->active_mm != mm) BUG();
        switch_mm(oldmm, mm, next, this_cpu);
    }

    if (!prev->mm) {
        prev->active_mm = NULL;
        mmdrop(oldmm);
    }
}
switch_to(prev, next, prev);
__schedule_tail(prev);
...
return;
}

```

코드 10. sched.c (8)

자, 이제는 context switching을 하는 일만 남았다. 먼저 mm(memory management)을 switch하고, 그리고 나서 register와 stack에 대한 switching을 한다. 그리고, 이전의 process를 queue의 끝으로 보낸다.

끝으로 지금까지 설명한 Linux의 process scheduling에서 가장 핵심에 속하는 schedule() 함수의 flowchart를 보도록 하자.

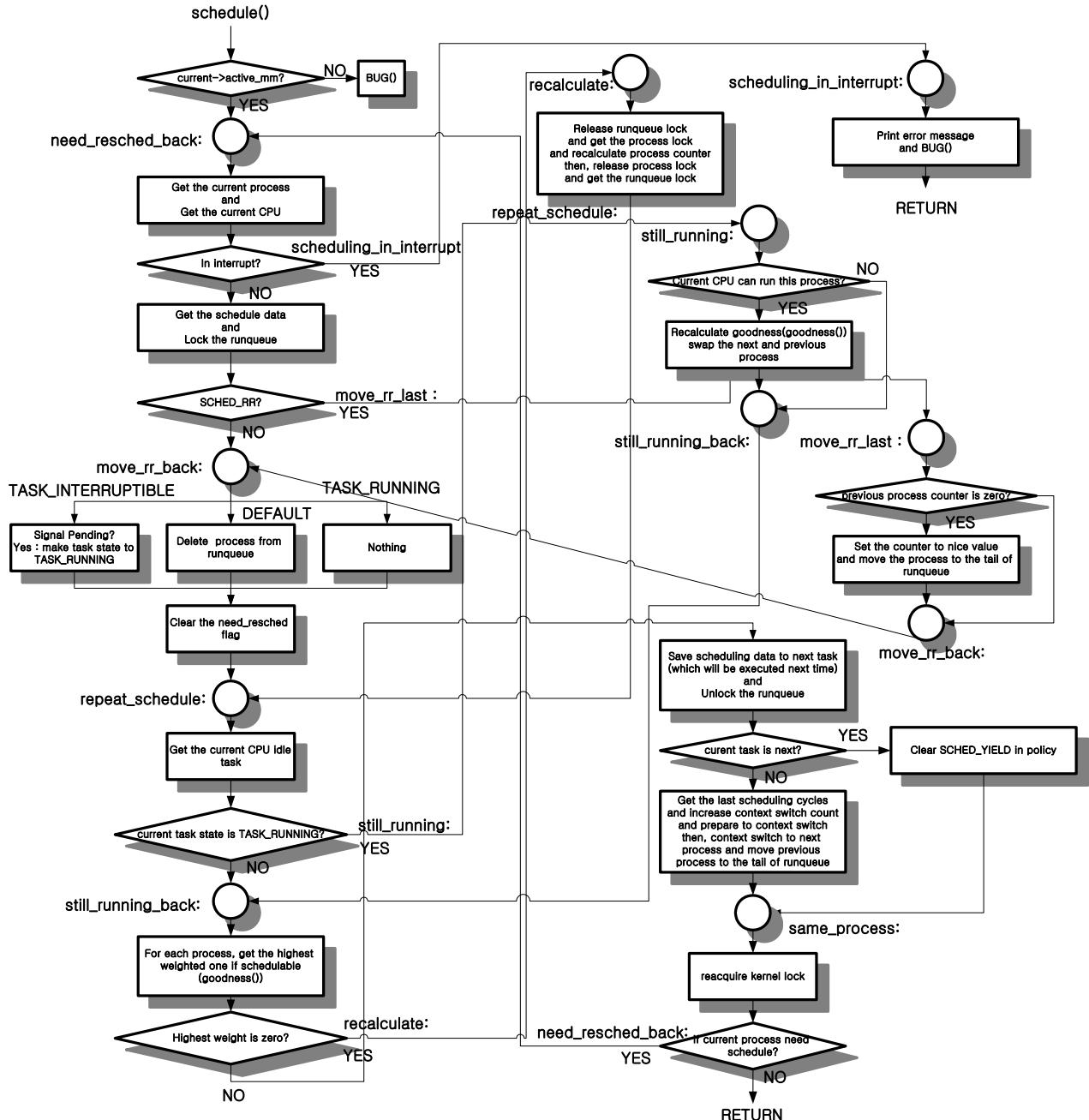


그림 11. schedule() 함수의 flowchart

[그림 11]은 앞에서 설명한 schedule() 함수의 전체 flowchart를 분석한 것이다. 그림에서 보듯이 schedule() 함수는 전체 프로세스 중에서 우선 순위(여기서는 weight 값이 될 것이다.)가 가장 높은 프로세스를 찾아서 이 프로세스로 context switching을 한다. 또한 이 함수에서 그러한 우선 순위 값에 영향을 주소는 요소로는 goodness() 값과 nice값이 있으며, 이 값을 적절히 활용해서 최상의 우선 순위를 가지는 프로세스를 찾는 일을 하고 있음을 알 수 있다.

또한, goodness() 함수의 전체 flowchart는 다시 아래와 같이 볼 수 있다. [그림 12]에서 알 수 있듯이 goodness() 함수에서는 real-time 프로세스와 시분할(time-sharing : SCHED_OTHER) 프로세스에 대해서 각각 weight값을 달리 계산한다는 것을 볼 수 있을 것이다.

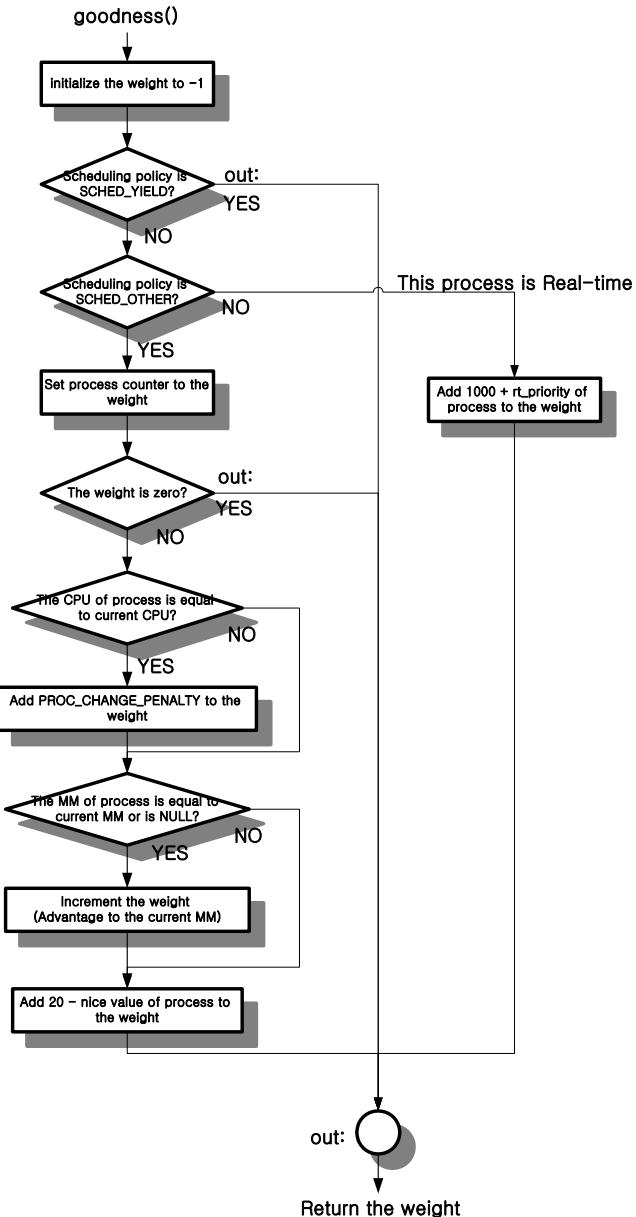


그림 12. goodness() 함수의 flowchart

즉, SCHED_OTHER인 scheduling policy를 가지는 프로세스에 대해서는 counter + 20 - nice + advantage를 weight값으로 주게 되지만, SCHED_FIFO나 SCHED_RR인 scheduling policy를 가지는 프로세스에 대해서는 weight값으로 1000 + rt_priority로 주어서 상대적으로 weight 값이 높도록 만들어 주고 있다. 물론 SCHED_YIELD로 되어있는 경우의 프로세스에 대해서는 weight 값이 -1로 설정되어 scheduling되지 않도록 만들어 주고 있다.

2.8. 기타 scheduling과 관련된 함수들

```

signed long schedule_timeout(signed long timeout)
{
    struct timer_list timer;
    unsigned long expire;

    switch (timeout)
  
```

```

{
    case MAX_SCHEDULE_TIMEOUT:
        schedule();
        goto out;
    default:
        if (timeout < 0)
        {
            printk(KERN_ERR "schedule_timeout: wrong timeout "
                   "value %lx from %p\n", timeout,
                   __builtin_return_address(0));
            current->state = TASK_RUNNING;
            goto out;
        }
    }
    expire = timeout + jiffies;
    init_timer(&timer);
    timer.expires = expire;
    timer.data = (unsigned long) current;
    timer.function = process_timeout;

    add_timer(&timer);
    schedule();
    del_timer_sync(&timer);
    timeout = expire - jiffies;
out:
    return timeout < 0 ? 0 : timeout;
}

```

코드 11. schedule_timeout() 함수의 정의

schedule_timeout()은 커널에서 timer를 설정해서 이 timer가 timeout이 되었을 경우에 특정한 일을 수행하기 위해서 사용한다. 넘겨받는 인자는 단순히 timeout값을 나타내는 signed long 변수이다. 만약 timeout값이 MAX_SCHEDULE_TIMEOUT이라면, schedule() 함수를 호출해서 다른 프로세스를 실행하도록 만든다. 만약 현재의 프로세스가 TASK_RUNNING 상태가 된다면, 이하의 goto out부분을 실행할 것이며, timeout값이 0보다 작은가를 비교해서 0 혹은 timeout값을 돌려준다.

timeout값이 MAX_SCHEDULE_TIMEOUT이 아니라면, default부분을 실행할 것이다. timeout값이 0보다 작다면, 이것은 오류가 되므로 현재 프로세스의 상태를 TASK_RUNNING으로 바꾸고⁸, 곧바로 out으로 제어를 옮긴다.

이전 timer를 등록시키는 부분이다. 먼저 expire를 현재 시간(jiffies)에 timeout값을 더한 시간으로 설정하고, 등록할 timer를 초기화 시킨다(init_timer()). Timeout이 걸리는 시간을 설정하고(timer.expires = expire), timer에 등록된 함수가 호출되었을 때 넘겨줄 데이터 값을 지정한다(timer.data = current). 이전 등록하기전의 마지막 단계로서 timeout이 걸렸을 때 호출할 함수로 process_timeout을 등록하도록 한다(timer.function = process_timeout). Timer를 등록하는 함수는 add_timer()이다. Timer의 등록이 끝났으므로 다른 프로세스에게 CPU를 양보하기 위해서 schedule() 함수를 호출하고, 이후에 다시 스케줄링이 되어 실행될 경우에는 schedule() 이하의 부분에서 계속 수행될 것이다. 따라서, 등록한 timer를 제거하고(del_timer()), timeout값을 새로 계산한 후(timeout = expire - jiffies) 복귀하게 될 것이다.

```

static inline int try_to_wake_up(struct task_struct * p, int synchronous)
{
    unsigned long flags;
    int success = 0;

    /*
     * We want the common case fall through straight, thus the goto.
     */

```

⁸ 이것은 이미 앞에서 프로세스의 상태를 TASK_INTERRUPTIBLE으로 바꿔어 있기 때문이다.

```

*/
spin_lock_irqsave(&runqueue_lock, flags);
p->state = TASK_RUNNING;
if (task_on_runqueue(p))
    goto out;
add_to_runqueue(p);
if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
    reschedule_idle(p);
success = 1;
out:
spin_unlock_irqrestore(&runqueue_lock, flags);
return success;
}

inline int wake_up_process(struct task_struct * p)
{
    return try_to_wake_up(p, 0);
}

static void process_timeout(unsigned long __data)
{
    struct task_struct * p = (struct task_struct *) __data;

    wake_up_process(p);
}

```

코드 12. process_timeout() 함수의 정의

process_timeout() 함수는 timer의 action에 해당한다. 즉, timeout이 걸렸을 경우에 실행되는 함수이다. 인자로서는 timer를 설정할 때 준 data 값으로, 프로세스의 구조체(task_struct)에 대한 포인터이다. 하는 일은 wake_up_process() 함수를 실행해서 해당하는 프로세스를 깨우는 일이다.

wake_up_process() 함수는 다시 try_to_wake_up() 함수를 호출한다. 넘겨주는 인자는 넘겨받은 task_struct에 대한 포인터와 동기화를 나타내는 값으로 0을 주었다. try_to_wake_up() 함수는 inline으로 정의된 함수로서, 먼저 interrupt가 걸리지 못하도록 lock을 설정하고(spin_lock_irqsave()), 프로세스의 상태를 TASK_RUNNING으로 바꾼 후, 이미 프로세스가 run queue에 있을 경우(task_on_runqueue())에는 곧바로 out으로 제어를 옮긴다. 그렇지 않다면, 해당 프로세스를 run queue에 넣기 위해서 add_to_runqueue() 함수를 호출한다. 만약 동기화를 나타내는 synchronous가 0이고, 프로세스 구조체의 cpus_allowed에 현재 프로세스가 수행되는 CPU ID값 만큼으로 1을 shift한 것을 AND해서 0인 경우에 대해서 reschedule_idle() 함수를 호출한다. 이것은 현재의 프로세스가 스케줄링되어 CPU에서 실행될 수 있는 가를 확인하기 위한 것으로 만약 적절한 CPU를 찾을 수 없다면, reschedule_idle() 함수를 호출해서, 새로운 CPU를 찾아야 한다. 즉, 이것은 SMP(Symmetric Multi-Processor)인 경우에 적절하게 실행시켜줄 CPU를 찾기 위한 것이다. 함수의 호출이 끝나면, 복귀값으로 사용할 success에는 1을 주고, 인터럽트를 가능하게 만든 후(spin_unlock_irqrestore()), success를 복귀 코드로 돌려주게 된다.

```

static FASTCALL(void reschedule_idle(struct task_struct * p));

static void reschedule_idle(struct task_struct * p)
{
#endif CONFIG_SMP
    int this_cpu = smp_processor_id();
    struct task_struct *tsk, *target_tsk;
    int cpu, best_cpu, i, max_prio;
    cycles_t oldest_idle;

    /*
     * shortcut if the woken up task's last CPU is

```

```

    * idle now.
    */
best_cpu = p->processor;
if (can_schedule(p, best_cpu)) {
    tsk = idle_task(best_cpu);
    if (cpu_curr(best_cpu) == tsk) {
        int need_resched;
send_now_idle:
    /*
     * If need_resched == -1 then we can skip sending
     * the IPI altogether, tsk->need_resched is
     * actively watched by the idle thread.
     */
    need_resched = tsk->need_resched;
    tsk->need_resched = 1;
    if ((best_cpu != this_cpu) && !need_resched)
        smp_send_reschedule(best_cpu);
    return;
}
}

```

코드 13. reschedule_idle() 함수의 정의

reschedule_idle() 함수는 프로세스가 깨어나서 실행 가능한 상태(TASK_RUNNING)에 있을 경우, 이를 실행시키기 위한 CPU를 찾지 못했다면 실행되는 함수이다. SMP의 구현에서는 현재 깨어나게 된 프로세스가 특정 CPU에서 수행되는 프로세스를 선점(preempt)해야 하는가를 결정하기 위한 일을 한다. SMP가 아닌 경우에는 CPU를 선점할 수 있는가 만을 따지게 된다.

this_cpu는 현재 사용하고 있는 CPU를 의미한다(smp_processor_id()). 코드에서 알 수 있듯이 CONFIG_SMP가 있을 경우에만 실행되는 부분으로 대부분이 처리되었다. best_cpu에는 스케줄링하려는 프로세스가 수행되었던 CPU의 값을 넣어준다. 즉, 가장 최근까지 사용하던 CPU가 일반적으로 context switch 오버헤드(overhead)가 적을 것이기 때문이다. 이젠 이렇게 구한 CPU(=best_cpu)에서 스케줄링하려는 프로세스를 스케줄링할 수 있는가를 보기 위해서 can_schedule() 매크로를 호출해서 알아본다.

```

static union {
    struct schedule_data {
        struct task_struct * curr;
        cycles_t last_schedule;
    } schedule_data;
    char __pad [SMP_CACHE_BYTES];
} aligned_data [NR_CPUS] __cacheline_aligned = { {{&init_task,0}} };

#define cpu_curr(cpu) aligned_data[(cpu)].schedule_data.curr
#define last_schedule(cpu) aligned_data[(cpu)].schedule_data.last_schedule

struct kernel_stat kstat;
extern struct task_struct *child_reaper;

#endif CONFIG_SMP
#define idle_task(cpu) (init_tasks[cpu_number_map(cpu)])
#define can_schedule(p,cpu) \
    ((p)->cpus_runnable & (p)->cpus_allowed & (1 << cpu))
#else
#define idle_task(cpu) (&init_task)
#define can_schedule(p,cpu) (1)
#endif

```

can_schedule() 매크로는 프로세스의 cpus_runnable flag에 설정된 CPU ID와 cpus_allowed의 값, 그리고 CPU ID값을 모두 bit-wise AND 시킨 값으로 특정 CPU에 실행가능하고, 허가되었는가를 알기위한 것이다. 만약 가능하다면 이하의 부분을 수행한다. 먼저 tsk에는 idle task의 값을 구해오도록 한다. 만약 best_cpu인 CPU에서 수행되던 작업이 idle task라면, need_resched에 idle task의 need_resched를 가져와서 저장하고, 원래의 idle task의 need_resched에는 1로 두어서, schedule() 함수를 호출해서 다시 스케줄링이 발생하도록 만든다. 만약 best_cpu가 현재 사용중인 CPU(=this_cpu)가 아니고, need_resched가 0인 경우에는 smp_send_reschedule() 함수를 호출해서 해당 CPU에 새로운 프로세스를 스케줄링 하도록 메시지를 보낸다. 즉, 스케줄링할 원래의 CPU가 아무런 일을 하고 있지 않은 경우 그 CPU에 스케줄링을 하도록 요청하는 경우가 된다. 그외의 경우에는 현재 CPU가 가장 좋은 CPU(=best_cpu)와 같거나 혹은 need_resched이 설정된 경우가 되므로, 그냥 복귀하면 schedule() 함수가 현재 사용중인 CPU에 대해서 스케줄링을 해줄 것이다.

코드 14. reschedule idle() 함수의 구현(계속)

`oldest_idle`은 가장 오래동안 idle한 시간을 가지며 -1로 초기 값을 주고, `target_tsk` 및 `max_prio`는 NULL과 0으로 설정한 후, `for()` loop를 돌면서 스케줄링 할 수 있는 적절한 CPU를 찾도록 한다. 찾게 되는 회수는 당연히 프로세서(CPU)의 개수 만큼(`smp_num_cpus`)이 될 것이다. `cpu_logical_map()`은 i386인 경우에는 아무런 일을 하지 않고, 단순히 넘겨받은 인자를 돌려줄 뿐이다. 따라서, `cpu`에는 i값이 들어갈 것이다. 이젠 스케줄링 할 수 있는 CPU만을 찾고(`can_schedule()`), 그렇지 않은 경우에는 그냥 다음번 loop를 돈다. 이렇게 찾은 CPU에서 현재의 프로세스를 구해서 `tsk`로 둔다. `tsk`와 그 CPU의 idle task가 같다면, i386인 경우에 한해서 이하를 실행한다. `smp num siblings`은 인접한 두개의 CPU가 있는가를 보기 위한 것으로

이 변수가 2로 설정된 경우에는 두개의 CPU간에 sibling인 관계가 있다고보고 sibling인 CPU의 프로세스도 idle인지를 확인한다. 만약 둘다 idle 상태였다면, `oldest_idle`에는 그 CPU의 마지막으로 스케줄링된 시점을 구해서 넣고, `target_tsk`에는 앞에서 얻은 tsk를 넣는다. `for()` loop는 여기서 끝나서 나오게 될 것이다.

만약 i386이 아닌 경우에는 `last_schedule()` 매크로를 호출해서 그 CPU에서 마지막으로 스케줄링이 일어난 시점을 구해서 `oldest_idle`과 비교한다. 작다면, `oldest_idle`을 갱신해주고, idle task를 `target_tsk`로둔다. 즉, 가장 오래동안 스케줄링이 되지 않고, idle했던 CPU를 찾는 것이다.

Idle task를 수행중이지 않은 경우에는 `oldest_idle`이 -1과 같은지를 확인한다. 즉, 앞에서 `oldest_idle`을 초기화 할 때 -1을 주었기에 idle이 아닌 CPU를 검사하게 될 경우에만 수행될 것이다. 이젠 `preemption_goodness()`를 호출해서 그 CPU에서의 선점(preempt) 가능성을 검토해 본다. 주어지는 값은 `prio`에 들어갈 것이며, `prio`가 `max_prio`보다 큰 경우에 대해서 `max_prio`의 값을 변경하고, `target_tsk`의 값을 CPU에서 수행하던 프로세스로 바꾼다. 이와 같은 과정을 반복해서 하면서 결국에는 `target_tsk`의 값이 정해질 것이라는 것을 알 수 있다.

```

tsk = target_tsk;
if (tsk) {
    if (oldest_idle != -1ULL) {
        best_cpu = tsk->processor;
        goto send_now_idle;
    }
    tsk->need_resched = 1;
    if (tsk->processor != this_cpu)
        smp_send_reschedule(tsk->processor);
}
return;
#else /* UP */
int this_cpu = smp_processor_id();
struct task_struct *tsk;

tsk = cpu_curr(this_cpu);
if (preemption_goodness(tsk, p, this_cpu) > 0)
    tsk->need_resched = 1;
#endif
}

```

코드 15. `reschedule_idle()` 함수의 정의

이전 이렇게해서 찾은 `target_tsk`를 `tsk`로 둔다. 만약 `tsk`가 어떤 값을 지닌다면 `if()`절을 수행할 것이다. `oldest_idle`과 -1이 같지 않다면, idle task가 적어도 하나는 있었다는 말이되며, `tsk`는 찾았던 idle task를 가지게 될 것이다. 따라서, `tsk`가 스케줄링 되고 있는 프로세서에서 스케줄링 하면 될 것이므로, `send_now_idle`로 제어를 옮겨서 해당 CPU에 스케줄링을 하도록 요청한다. 그렇지 않다면, `tsk`의 `need_resched` 필드를 1로 설정하고, 만약 `tsk`가 수행중이던 프로세스가 현재 수행중인 CPU와 다를 경우에는 `smp_send_reschedule()`를 호출해서 해당 CPU에 스케줄링을 요청한다. 이것을 마치고 나면 바로 복귀하게 될 것이다. 만약 SMP가 아닌 경우에는 현재 프로세서(=CPU)에서만 스케줄링 우선순위를 검사해서 이 값이 0이 아닌 경우에 새로운 스케줄링을 하도록 진행중인 프로세스의 `need_resched` 필드에 1을 넣는다.

```

static inline int preemption_goodness(struct task_struct * prev, struct task_struct * p, int cpu)
{
    return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->active_mm);
}

```

코드 16. `preemption_goodness()`

`preemption_goodness()` 함수는 `goodness()` 함수를 호출해서 해당 CPU에 대해서 어느정도의 우선도를 가지고 있는지를 검사하는 inline함수이다. 여기서 돌려받는 값에 따라 양수값을 받는다면, 해당 CPU에서 수행중인 프로세스를 중단하고, 스케줄링을 하라는 말이되며, 그렇지 않고 음수나 0인 값을 가진다면, 스케줄링 해줄 필요가 없다는 뜻이된다.

지금까지 우리는 process scheduling에 대한 이야기를 code를 보면서 살펴보았다. 중요한 것은 Uni-processor를 가진 system과 multi-processor를 가진 system이 scheduling algorithm상 차이가 있다는 점과 현재의 process에서 실행중인 것이 상대적으로 더 높은 우선순위 값을 가지도록 만들어 줌으로써, 같은 processor상에서 수행되도록 해준다는 점이다. 또한 실시간 process에 대해서 상대적으로 normal process보다 높은 우선 순위를 가지도록 해준다는 점도 눈에 띈다.

2.9. Timer Interrupt의 처리

리눅스에서는 매 clock tick마다 interrupt가 발생한다. System의 모든 중요한 연산들은 이렇게 일어나는 timer와 관련을 가지고 있으며, 빠른 처리를 요구한다. 이러한 timer는 hardware를 적절한 interval에 interrupt를 발생시키도록 해서 정해지게 되며, 리눅스에서는 이렇게 발생된 interrupt를 이용해서 system time을 유지한다. 매 clock tick은 10ms로 나타나지며, 따라서, 100번 매 초마다 발생한다고 하겠다.

```
unsigned long volatile jiffies;
```

는 이렇게 발생한 매 clock interrupt마다 증가하게 된다. timer interrupt를 설치하는 것은 `~/arch/i386/kernel/time.c`에서 `time_init()`이며, `setup_irq(unsigned int irq, struct irqaction *new)`를 사용한다. 이곳에서 등록되는 handler function은 `timer_interrupt()`이며, 이곳에서 `do_timer()`를 호출해서 jiffies에 대한 값을 증가시켜준다.

```
void do_timer(struct pt_regs *regs)
{
    (*(unsigned long *)&jiffies)++;
#ifndef CONFIG_SMP
    /* SMP process accounting uses the local APIC timer */
#endif
    update_process_times(user_mode(regs));
    mark_bh(TIMER_BH);
    if (tq_timer)
        mark_bh(TQUEUE_BH);
}
```

코드 17. timer.c(1)

또한 `do_timer`에서는 SMP machine이 아닌 경우에는 process의 accounting과 관련된 정보를 갱신하고, timer와 관련된 작업들을 처리하기 위해서 TIMER_BH⁹를 표시하며, timer queue에 수행할 작업이 있을 경우에는 TQUEUE_BH를 다시 mark한다. 이렇게 표시된 각각의 작업들은 나중에 kernel mode에서 user mode로 system의 mode가 변경될 때 깨어나서 처리를 해주게 된다.

```
void timer_bh(void)
{
    update_times();
    run_timer_list();
}
...
void tqueue_bh(void)
{
```

⁹ Bottom Half를 말하며, 급한 처리를 한 다음에 나중에 system에 여유가 있을 때 추가적인 처리를 해주기 위해서 사용하는 kernel mechanism이다. 이 개념은 리눅스만의 독특한 것은 아니다.

```

    run_task_queue(&tq_timer);
}

```

코드 18. timer.c(2)

*timer_bh()*에서 해주는 것은 현재의 system time을 update하고 timer에 등록된 함수들이 만기가 되었을 때 처리하는 역할을 한다. *tqueue_bh()*에서는 *tq_timer*에 등록된 작업들을 처리한다. 나중에 bottom-half에 대한 것을 다룰 때 이러한 queue들에 대해서 더 자세히 알아보도록 하겠다.

timer에 대한 처리 중 *update_times()*와 *update_process_times()*에 대해서 좀더 보도록 하자. 이중 *update_time()*은 bottom half의 처리 중에 호출되며, 하나는 *do_timer()*에서 직접적으로 호출된다. *update_process_times()*는 현재의 process의 user time과 system time을 update하는 역할을 한다.

```

void update_process_times(int user_tick)
{
    struct task_struct *p = current;
    int cpu = smp_processor_id(), system = user_tick ^ 1;

    update_one_process(p, user_tick, system, cpu);
    if (p->pid) {
        if (--p->counter <= 0) {
            p->counter = 0;
            p->need_resched = 1;
        }
        if (p->nice > 0)
            kstat.per_cpu_nice[cpu] += user_tick;
        else
            kstat.per_cpu_user[cpu] += user_tick;
        kstat.per_cpu_system[cpu] += system;
    } else if (local_bh_count(cpu) || local_irq_count(cpu) > 1)
        kstat.per_cpu_system[cpu] += system;
}

```

코드 19. timer.c(3)

여기서 말하는 user time과 system time은 user mode에서, 혹은 kernel mode에서 진행한 사용시간을 말하며, 이 값들은 process를 account하는 정보로 사용된다. 또한 process의 counter field를 감소시켜서 만약 process가 자신에게 할당된 time을 다 소비했다면 새로운 scheduling이 일어나도록 해준다(*need_resched* = 1). *user_mode()*라는 macro는 현재의 process가 user mode에서 진행 중인지, 아니면 kernel mode에서 진행 중이었는지를 나타내는 Boolean 값을 return한다.

```

static inline void update_times(void)
{
    unsigned long ticks;

    /*
     * update_times() is run from the raw timer_bh handler so we
     * just know that the irqs are locally enabled and so we don't
     * need to save/restore the flags of the local CPU here. -arca
     */
    write_lock_irq(&xtime_lock);

    ticks = jiffies - wall_jiffies;
    if (ticks) {
        wall_jiffies += ticks;
        update_wall_time(ticks);
    }
    write_unlock_irq(&xtime_lock);
}

```

```
    calc_load(ticks);
}
```

코드 20. timer.c(4)

*update_times()*에서는 *system*의 현재 시간(*xtime*¹⁰)을 바꾸는 역할을 한다. 또한 가장 최근에 wall time¹¹으로 사용된 jiffies값을 가지는 *wall_jiffies*에 대한 조정과 CPU의 load¹²를 계산하게 된다.

참고로 보통 process가 실행을 지속할 수 있는 time slot을 quantum이라고 하며, 만약 process가 I/O를 요구하거나 스스로 sleep 상태로 가지 않는 한, Unix에서는 quantum동안의 process의 수행을 보장한다. 따라서, time quantum이 끝나게 되면 새로이 수행할 process를 선택하게 되며, time quantum이 끝났다는 것은 timer interrupt를 처리하면서 알게 된다.

2.10. Thread in Linux

쓰레드(Thread)란 프로세스의 실행 단위이다. 쓰레드는 주소 공간과 다른 자원을 필요로 하지만, 이들 중의 많은 부분을 다른 쓰레드와 공유할 수 있다. 또한 쓰레드는 다른 쓰레드와 주소 공간과 자원(resource)을 공유하지만 각각이 독립적으로 스케줄링(scheduling)될 수 있다. 쓰레드의 종류로는 커널 쓰레드, lightweight 프로세스, 그리고, 사용자 쓰레드가 있다. 각각에 대해서 알아보도록 하자.

2.10.1. Kernel Thread

커널 쓰레드는 사용자 쓰레드와 관련을 가질 필요가 없다. 필요시에 커널에 의해서 생성 및 삭제된다. 커널의 텍스트(text)와 전역 데이터를 공유하며, 자신만의 커널 스택(stack)을 가진다. 또한 독립적으로 스케줄링(scheduling)되며, 커널에서 제공하는 sleep이나 혹은 wakeup과 같은 표준화된 동기화(synchronization)방법을 사용한다.

커널 쓰레드는 비동기적(asynchronous)인 연산을 수행할 때 유용하며, 인터럽트 등을 처리할 때도 사용될 수 있다. 커널 쓰레드를 생성하는 데는 적은 비용이 들며, 단지 커널 스택(stack)과 레지스터(register)등의 실행 환경(context)정보를 보관할 정도의 공간만을 필요로 한다. 또한 커널 쓰레드의 환경전환(context switch)은 일반적인 프로세스의 환경전환보다 빠른데, 이것은 현재 사용중인 메모리에 대한 메모리 mapping을 다시 할 필요가 없기 때문이다.

리눅스가 커널 내부에서 쓰레드를 생성하는 방법은 *kernel_thread()* 함수를 호출하는 것이다. *kernel_thread()* 함수는 *~/arch/i386/kernel/process.c*에 함수의 아래와 같이 정의 되어 있다.

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
    long retval, d0;

    __asm__ __volatile__(
/* 기계어 명령어 부분이다.*/
        "movl %%esp,%%esi\n\t"/* clone()의 시스템 콜 번호를 %eax에 가지고 있다.
                                %ebx에는 flags값이 들어가 있다.*/
        "int $0x80\n\t"          /* clone() 시스템 콜을 호출한다.*/
        "cmpl %%esp,%%esi\n\t"   /* if(retval) ? 부모 프로세스 : 자식 프로세스 */
        "je 1f\n\t"              /* 1f로 jump */
/* 자식 프로세스로 진행한다.*/
        "movl %4,%eax\n\t"      /* arg를 %eax에 넣는다.*/
        "pushl %%eax\n\t"        /* %eax를 pushl */
        "call *%5\n\t"           /* fn을 부른다.*/
    );
}
```

¹⁰ 실제로 *system*에서 사용하게 되는 시간을 나타내는 변수이다.

¹¹ 실제적으로 쓰이는 system time을 말한다.

¹² CPU의 load란 특정 시간동안 얼마나 많은 작업들이 수행되었는지를 나타내는 척도라고 생각하면 된다.

```

    "movl %3,%0\n\t" /* exit()의 시스템 콜 번호를 %eax에 넣는다.*/
    "int $0x80\n"      /* exit() 시스템 콜을 호출한다.*/
    "1:\t"             /* Forward Jump를 위한 label */.

/* 아래 부분은 위에서 쓰는 기계어 명령어의 operand로 사용된다.*/
    :"=a" (retval), "=S" (d0),
    :"0" (__NR_clone), "i" (__NR_exit),
    "r" (arg), "r" (fn),
    "b" (flags | CLONE_VM)
    : "memory");

return retval;
}

```

코드 21. kernel_thread()함수의 정의

kernel_thread() 함수가 넘겨 받는 파라미터 값은 생성하고자 하는 쓰레드가 실행할 함수의 주소와 그 함수에 넘겨줄 인자에 대한 포인터, 그리고 flags 값이다. 이 함수는 위와 같이 inline assembly 언어로 되어 있어서 보기 어렵지만, clone() 시스템 콜을 사용해서 프로세스의 이미지를 새로이 생성하고, 생성한 것이 부모라면 곧바로 생성한 프로세스 ID를 가지고 곧바로 복귀하며, 생성한 것이 자식이 될 경우에는 함수를 수행하고, exit() 시스템 콜을 사용해서 복귀하게 된다. 이렇게 생성된 각각의 커널 쓰레드는 특정한 커널의 한 함수(function)을 수행하게 되는데, 다른 일반적인 프로세스들은 커널을 접근하기 위해서 시스템 콜을 써야만 한다. 또한 커널 쓰레드는 커널 모드에서만 수행되며, 항상 PAGE_OFFSET(0xC0000000) 이상의 선형 주소 공간에 대해서만 접근한다.

이러한 커널 쓰레드를 쓰는 예로서는 boot초기 ~/init/main.c에서 start_kernel()과 같은 함수에서 init()함수를 커널 쓰레드(PID=0)로 생성해서 사용하고 있는 부분을 찾을 수 있을 것이다. 이 커널 쓰레드는 다시 기본적인 설정을 수행하고 나서, 실제 시스템의 init 프로세스(PID=1)를 실행하는 역할을 할 것이며, 원래의 부모 역할을 하는 커널 쓰레드는 계속 진행해서 cpu_idle()함수를 수행하게 될 것이다.

2.10.2. Lightweight Process

Lightweight Process(LWP)는 커널의 지원을 받는 사용자 쓰레드이다. 따라서, LWP를 지원하기 위해서는 커널에서 반드시 쓰레드를 지원해 주어야 한다. 모든 프로세스는 하나나 그 이상의 LWP를 가질 수 있으며, 또한 각각의 LWP는 커널 쓰레드의 지원을 받는다. LWP는 독립적으로 스케줄링이 되며, 프로세스의 주소공간과 자원을 공유할 수 있다. 멀티프로세스를 가진 시스템에서는 LWP를 각각의 CPU마다 분산시켜서 수행해 줄 수 있으며, I/O요구시 블록킹이 일어나는 것은 전체 프로세스가 아니라, 단지 하나의 LWP만이 블록킹이 된다.

커널의 스택과 레지스터의 실행환경을 저장할 공간 이외에 LWP는 사용자 상태(state)정보를 기록할 필요가 있으며, 이에 들어가는 정보로는 사용자 레지스터의 실행환경(context)이 있다. 즉, 사용자 수준(level)에서 수행중인 쓰레드의 실행환경이다. 커널 쓰레드들은 반드시 LWP를 가질 필요는 없으며, 특정 커널 쓰레드는 커널과 연관된 작업만을 수행할 수 있다.

[그림11]은 LWP의 구현을 보여준다. [그림11]에서 보면 프로세스들은 여러 개의 LWP를 가질 수 있으며, LWP는 또한 커널 쓰레드의 지원을 받고 있음을 볼 수 있다.

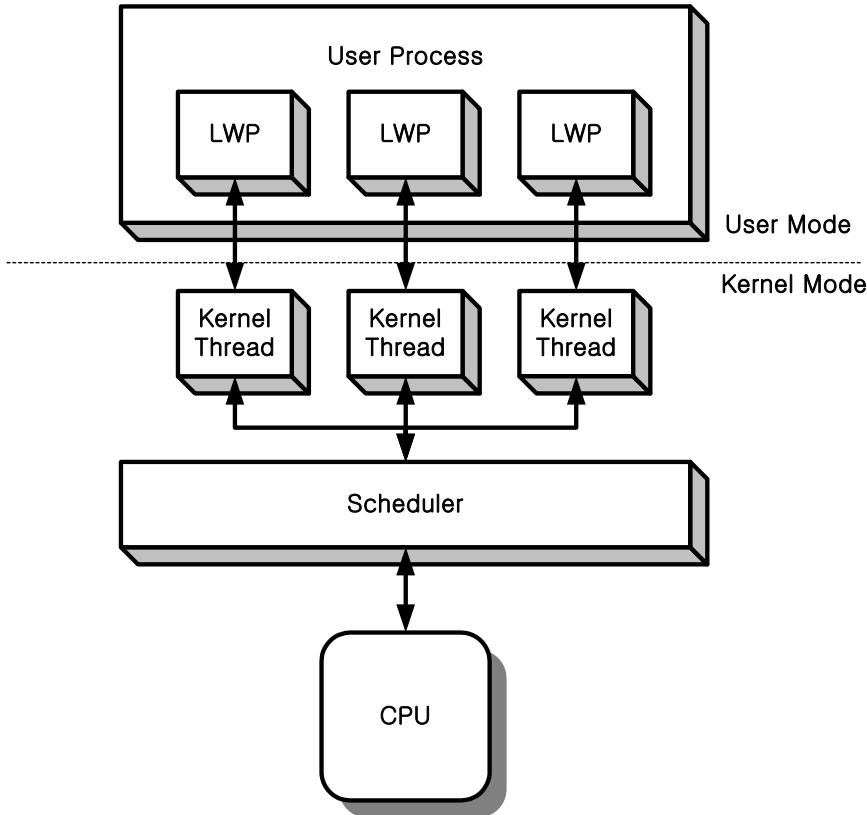


그림 13. Light Weight Process

이러한 멀티쓰레드 프로세스 환경에서는 각각의 쓰레드들이 서로 독립적일 때 성능이 가장 좋다. 또한 사용자 모드의 프로세스는 완전한 선점(preeemption)이 가능하고, 모든 프로세스 내의 LWP들이 같은 주소 공간을 사용한다. 만약 여러 LWP 간에 데이터를 공유하려면 락(lock), 상호배제(mutex), 세마포어(semaphores), 혹은 조건 변수들이 필요하게 된다. 이것에 대해서는 프로세스의 동기화(synchronization)에 대해서 살펴볼 때 자세히 보게 될 것이다.

하지만 만약 많은 수의 LWP들이 공유 데이터를 자주 접근하게 되면 쓰레드를 두어서 얻는 성능상의 이익은 별로 없게 된다. 왜냐하면 자주 자신이 사용할 자원들이 사용중이 아닌지를 확인하거나 기다려야 하기 때문이다.¹³ 하지만 이렇게 기다리는 상태는 잠시동안 다른 프로세스가 사용중일 때 기다리는 것에는 유용하지만, 그렇지 않은 경우에는 커널에서 특별한 배려를 해주어야 한다. 따라서, 이것은 오버헤드(overhead)로 작용할 수 있는 여지가 있다.

또한 각각의 LWP가 커널 스택을 사용해야 하기 때문에 시스템은 많은 양의 LWP를 제공해 주지는 못한다. 시스템은 사용자 프로세스가 적어도 하나의 LWP를 가지고 있기 때문에, LWP가 필요 없는 프로세스들에게도 쓸데없이 LWP를 생성해 주어야 한다. 이것 역시 시스템 입장에서는 부담이 될 수밖에 없다. 만약 프로세스가 많은 수의 LWP를 사용하고, 자주 생성과 삭제가 일어날 경우도 문제의 여지가 있으며, LWP는 커널의 도움을 받아서 스케줄링이 되어야 하므로, 응용프로그램에서 쓰레드들 간에 제어가 자주 이전되는 경우에도 구현하기가 쉽지 않다. 그리고, 하나의 프로세스가 많은 수의 LWP를 생성한다면, 스케줄링상 CPU를 독점하는 상황도 발생할 수 있다.

따라서, 이와 같은 LWP를 사용하는 것은 프로그램의 특별한 주의를 요한다. 많은 응용프로그램들은 다음에서 설명할 사용자 쓰레드를 사용해서 더 잘 수행될 수 있을 것이다.

2.10.3. User Thread

사용자 쓰레드란 커널의 개입 없이 완전히 사용자 수준(level)에서 사용되는 쓰레드를 말한다. 이것은 주로 라이브러리(library) 형식으로 주어지는 것에 의지해서 구현되며, 예로는 Mach의 C-쓰레드와 Posix

¹³ 이러한 것을 busy waiting이라고 한다.

쓰레드(pthread)가 있다. 이러한 라이브러리는 쓰레드의 생성과 동기화, 스케줄링 및 쓰레드의 관리를 커널의 도움 없이도 가능하도록 만든다. 또한 속도적인 측면에서도 빠르다고 볼 수 있다.¹⁴ [그림12]는 사용자 쓰레드의 구성을 간단히 나타낸 것이다. [그림12]에서 알 수 있듯이 커널과는 완전한 분리를 보인다. 스케줄러는 단지 프로세스만이 존재한다고 생각하며, 사용자 프로세스를 그런 프로세스 단위로만 스케줄링해 준다.

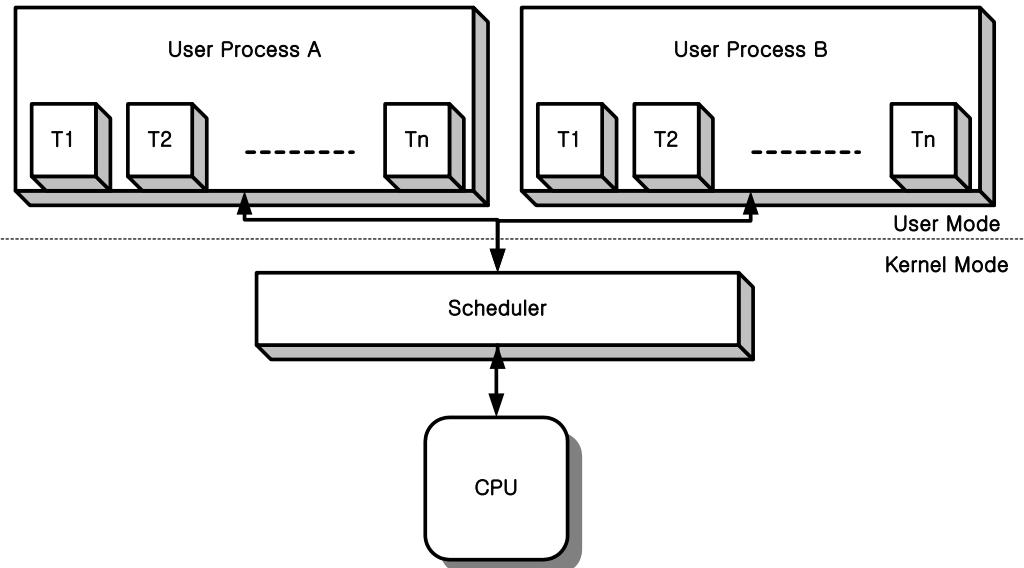


그림 14. User Thread의 구성

실제적으로 pthread 라이브러리나 C-쓰레드 라이브러리는 사용자 쓰레드의 스케줄링 및 동기화(synchronization)와 환경전환(context switch)을 책임지게 되어 커널과 같은 역할을 해준다고 볼 수 있다. 물론 이런 경우에도 커널의 개입이 없다. 구현을 위해서는 각각의 사용자 쓰레드들은 자신의 사용자 수준에서의 환경전환(context switch) 및 상태 정보의 보관과 시그널 매스크(signal mask) 등을 위해서 사용자 스택을 가지고 있어야 하며, 라이브러리는 사용자 쓰레드들 간의 환경전환을 지원하기 위해서 이들 스택을 이용하게 된다.

사용자 쓰레드가 가진 단점은 커널이 사용자 쓰레드에 대해서 아무런 정보가 없기에, 한 프로세스내에서 실행중인 쓰레드들 간의 보호(protection) 기능을 제공해 줄 수 없으며, 커널과 라이브러리간에 사용하는 스케줄링에 관련된 연산에 있어서도 문제점을 가지고 있다. 즉, 커널은 프로세스만을 스케줄링하게 되고, 라이브러리는 프로세스내의 쓰레드를 스케줄링하게 됨으로써, 프로세스를 내부의 쓰레드들 간의 우선순위(priority)는 알 수가 없다. 따라서, 높은 우선 순위의 사용자 쓰레드가 더 낮은 우선 순위를 가지는 다른 프로세스의 사용자 쓰레드에 의해서 선점(preempt)되는 상황이 생기게 된다.

리눅스에서는 pthread라는 라이브러리를 이용한 쓰레드 프로그램을 지원하고 있다. 커널 쓰레드에 대한 지원은 2.2.X version의 커널에서 가능하게 되었으며, 2.3.X version이후의 커널에서는 SMP 시스템에 대한 커널 쓰레드도 가능하게 되었다.

이상에서 우리는 세가지 쓰레드의 구현에 대해서 알아 보았으며, 리눅스에서는 쓰레드를 지원하기 위해서 고유한 커널 함수와, 커널 쓰레드 및 pthread에 대한 라이브러리의 지원을 받을 수 있다고 이야기 했다. 다음으로 볼 것은 프로세스의 생성 및 실행, 종료에 대한 것이다. 그것을 논의하기 전에 이와 같은 일을 가능하게 해주는 시스템 콜에 대해서 먼저 간단히 알아보기로 한다.

¹⁴ Kernel에서 비동기 I/O(asynchronous I/O)를 지원해서 하나의 thread가 I/O를 기다리고 있을 때, 다른 thread는 다른 작업을 진행할 수 있다. 만약 I/O가 끝나게 되면, 다시 그 thread가 스케줄링되어 실행될 때 계속적으로 진행된다.

2.11. System Call

시스템 콜(system call)이란 프로세스가 커널에 서비스를 요청하는 것이다. 즉, 프로세스와 커널간의 대화 창구 역할을 한다고 하겠다. 커널은 시스템 콜 인터페이스(interface)를 제공하며, 이것은 헤더(header) 파일 형태로 사용자 프로세스에 제공된다. 사용자 프로세스는 직접적으로 혹은 라이브러리(library)를 이용해서 커널에 접근하게 되며, 시스템 콜을 하게 되는 순간부터 사용자 모드에서 커널 모드로 전환되어 수행된다. 따라서, 시스템 콜을 시작하면 커널의 코드가 사용자 프로세스의 환경(context) 아래에서 동작하게 되는 것이다. 실제로 시스템 콜은 운영체제가 제공하는 라이브러리라고 생각하면 된다.

리눅스에서의 시스템 콜은 인터럽트를 사용해서 구현된다. 인터럽트 0x80이 이를 위해서 준비되어 있다. 리눅스에서는 ~/include/arch/unistd.h¹⁵에 있는 _syscallX 매크로(macro)를 사용해서 시스템 콜을 할 수 있도록 하고 있으며, 이곳에서 인터럽트 0x80을 호출한다. 넘겨줘야 할 파라미터(parameter)들은 레지스터(register)를 통해서 전달되며, 돌려 받는 값도 레지스터를 통한다.

~/arch/i386/kernel/traps.c의 trap_init()에서 시스템 콜에 대한 인터럽트 벡터가 초기화 설정이 되며, 여기에 들어가는 것이 system_call()함수이다. system_call()함수는 ~/arch/i386/kernel/entry.S에 어셈블리(assembly)어로 정의되어 있다. 또한 ~/arch/i386/kernel/entry.S를 보면, 모든 시스템에서 제공하는 시스템 콜은 sys_call_table[]내에 들어간다는 것을 알 수 있다.

```
ENTRY(system_call)
    pushl %eax                      # save orig_eax
    SAVE_ALL
    GET_CURRENT(%ebx)
    cmpl $(NR_syscalls),%eax
    jae badsys
    testb $0x02,tsk_ptrace(%ebx)# PT_TRACESYS
    jne tracesys
    call *SYMBOL_NAME(sys_call_table)(,%eax,4)
    movl %eax,EAX(%esp)             # save the return value
    ...

```

코드 22. entry.S(1) 시스템 콜의 시작

system_call()을 보면, 먼저 원래의 eax레지스터를 보관한 다음, 레지스터 들을 스택(stack)에 push하는 SAVE_ALL 매크로를 실행한다. 현재의 프로세스의 task_struct를 가져오기 위해서 GET_CURRENT 매크로를 실행하고, 시스템 콜 숫자가 맞는지 확인한다. 만약 올바른 값으로 시스템 콜을 했다면, 현재 프로세스가 트레이스(trace)되고 있는지 확인한 다음, 그렇지 않을 경우 sys_call_table[]에 등록된 함수를 호출하게 된다. 호출의 결과는 eax에 저장되며, 이것을 나중에 돌려주기 위해서 다시 스택에 저장한다. 만약 트레이스가 되고 있다면, 다시 ~/arch/i386/kernel/ptrace.c에 정의된 syscall_trace()를 호출해준다. 이곳에서는 현재 프로세스의 상태를 TASK_STOPPED로 만들고, 부모 프로세스에게 알려준 후 스케줄링을 요청한다.

```
...
ENTRY(ret_from_sys_call)
#ifndef CONFIG_SMP
    movl processor(%ebx),%eax
    shll $CONFIG_X86_L1_CACHE_SHIFT,%eax
    movl SYMBOL_NAME(irq_stat),%ecx          # softirq_active
    testl SYMBOL_NAME(irq_stat)+4,%ecx # softirq_mask
#else
    movl SYMBOL_NAME(irq_stat),%ecx          # softirq_active
    testl SYMBOL_NAME(irq_stat)+4,%ecx # softirq_mask
#endif

```

¹⁵ 사용자 모드에서 시스템 콜에 대한 인터페이스는 /usr/include/asm/unistd.h에 나와 있다.

```

jne    handle_softirq

ret_with_reschedule:
    cmpl $0,need_resched(%ebx)
    jne reschedule
    cmpl $0,sigpending(%ebx)
    jne signal_return
restore_all:
    RESTORE_ALL
...

```

코드 23.entry.S(2) 시스템 콜로부터의 복귀

복귀(return)는 `ret_from_sys_call`에서 처리가 된다. 소프트 인터럽트가 설정되었는지를 확인 및 처리하게 되며, 현재의 프로세스가 `need_resched` 필드가 설정되었는지를 확인, 만약 설정되었다면 스케줄링 요구를 처리한다. 또한 처리할 시그널이 있는지를 확인하고, 그것을 다시 처리하도록 한다. 그리고 나서 `RESTORE_ALL` 매크로를 실행해서 이전에 저장했던 레지스터 값을 다시 불러드린다. 이 과정에서 복귀(return) 값을 넘겨주고 `iret`를 수행한다.

여기서 한가지 주의 할 점은 만약 현재 프로세스가 `trace`가 되고 있다면, 각각의 시스템 콜의 앞뒤로 `syscall_trace()`가 호출된다는 사실이다. 이렇게 함으로써 부모 프로세스는 자식 프로세스에 대해서 완전한 제어를 할 수 있게 된다.

2.12. Interrupt의 처리

위에서 타이머 인터럽트(interrupt)의 처리에 대해서 알아보았다. 이곳에서는 전체적인 인터럽트의 처리를 보도록 하자. 인터럽트란 하드웨어가 운영체제에게 어떤 사건이 발생했음을 알리는 도구이다. 리눅스에서는 인터럽트를 크게 두 가지로 나누고 있는데, Fast 인터럽트와 slow 인터럽트가 그것이다. 여기에 시스템 콜(interrupt 0x80을 덧붙이면 세가지로도 나눌 수도 있을 것이다.

먼저 fast 인터럽트는 작고 간단한 작업을 위해서 사용되는 인터럽트이다. 즉, 빠른 처리를 요하는 인터럽트라고 할 수 있다. 만약 이러한 인터럽트가 처리중이라면 다른 인터럽트들은 전부 블록킹(blocking)이 되며, 인위적으로 다시 사용 가능(enable)하게 만들어 주어야 한다. 이것의 예로는 `keyboard` 인터럽트를 들 수 있다. slow 인터럽트는 처리가 진행 중이더라도 다른 인터럽트가 가능한 인터럽트이다. slow 인터럽트가 처리되고 나면, 나중에 시스템의 작업을 처리해 주어야 할 경우가 있게 된다.

인터럽트의 초기화는 `~/init/main.c`의 `start_kernel()`에서 부팅하게 될 때, `~/arch/i386/kernel/i8259.c`의 `init_IRQ()`를 호출하면서 초기화가 되며(intel x86의 경우), 이곳에서는 인터럽트 벡터(vector)테이블의 초기화가 이루어지며, 클럭(clock)의 `hz`를 결정해 준다. 시스템 콜 벡터 빈곳으로 남겨놓고 이곳에서는 설정하지 않는다.

일반적인 인터럽트의 처리는 `~/arch/i386/kernel/irq.c`의 `do_IRQ()`에서 일어난다. 인터럽트의 초기화를 담당하는 `init_IRQ()` 부분에서 모든 일반적인 인터럽트들의 핸들러를 이곳으로 바꾸어 줌으로써 가능하다.

```

asmlinkage unsigned int do_IRQ(struct pt_regs regs)
{
    int irq = regs.orig_eax & 0xff; /* high bits used in ret_from_code */
    int cpu = smp_processor_id();
    irq_desc_t *desc = irq_desc + irq;
    struct irqaction * action;
    unsigned int status;

    kstat.irqs[cpu][irq]++;
    spin_lock(&desc->lock);
    desc->handler->ack(irq);

    status = desc->status & ~(IRQ_REPLAY | IRQ_WAITING);

```

```
status |= IRQ_PENDING; /* we _want_ to handle it */
```

먼저 넘겨 받은 레지스터(register)를 가르키는 구조체에서 인터럽트 수를 구해오고, 어떤 CPU에서 처리를 하게 될지를 구해온다. 그리고 나서 CPU에 발행한 인터럽트의 수를 증가시켜주고, 그 인터럽트에 ack(처리했다는 표시)를 표시해 준다. status는 현재 인터럽트의 라인(line) 상태(status) 값을 나타내며, 값으로는 아래와 같은 것이 있다

이름	값	설명
IRQ_INPROGRESS	1	IRQ 핸들러(handler)가 현재 active상태이다.
IRQ_DISABLED	2	IRQ 가 현재 불가 상태이다.(disable)
IRQ_PENDING	4	현재 IRQ가 처리를 기다리고 있으며, 가능(enable)상태가 되면 처리되어야 한다.
IRQ_REPLAY	8	현재 IRQ가 재 처리(play)되고 있지만 ack(처리했음)가 되지는 않았다.
IRQ_AUTODETECT	16	IRQ가 자동 검출되고 있다.
IRQ_WAITING	32	IRQ가 자동 검출이 되어야 하지만, 아직 알려지지는 않았다.
IRQ_LEVEL	64	IRQ가 level triggering ¹⁶ 을 사용한다.
IRQ_MASKED	128	IRQ가 mask되었다.
IRQ_PER_CPU	256	IRQ가 CPU에 할당되었다.

표 5. IRQ line status.

현재 처리하고자 하는 인터럽트 기술자(descriptor)의 REPLAY와 WAITING을 꺼주고, 처리할 것이라는 것을 나타내는 PENDING값을 켜준다.

```
/*
 * If the IRQ is disabled for whatever reason, we cannot
 * use the action we have.
 */
action = NULL;
if (!(status & (IRQ_DISABLED | IRQ_INPROGRESS))) {
    action = desc->action;
    status &= ~IRQ_PENDING; /* we commit to handling */
    status |= IRQ_INPROGRESS; /* we are handling it */
}
desc->status = status;
if (!action)
    goto out;
```

만약 인터럽트 핸들러가 불가이던가 혹은 이미 진행 중이었다면, PENDING을 꺼주고 인터럽트가 진행 중이라고 표시해 준다. 만약 핸들러의 action 필드가 정의되지 않았다면 out으로 제어를 옮긴다.

```
for (;;) {
    spin_unlock(&desc->lock);
    handle_IRQ_event(irq, &regs, action);
    spin_lock(&desc->lock);

    if (!(desc->status & IRQ_PENDING))
        break;
```

¹⁶ Level triggering과 edge triggering의 두 가지가 있으며, 현재 어떤 상태로 머물러 있을 경우에 걸리는 interrupt가 level triggering이며, 그 상태로 전이하는 순간에 걸리는 interrupt가 edge triggering이다.

```

        desc->status &= ~IRQ_PENDING;
    }
    desc->status &= ~IRQ_INPROGRESS;
out:
    desc->handler->end(irq);
    spin_unlock(&desc->lock);

    if (softirq_active(cpu) & softirq_mask(cpu))
        do_softirq();
    return 1;
}

```

코드 24. ~/arch/kernel/irq.c(1)

for loop를 돌면서 인터럽트에 대한 처리 및 PENDING된 인터럽트도 처리하게 되며, 처리를 마치면 현재 인터럽트가 진행중이 아니라고 표시한다. out에서 하는 일은 핸들러에 등록된 end함수를 불러주고, 인터럽트 기술자에 대해서 lock을 해지해주며, CPU에 활성화된 소프트 IRQ를 처리해주는 일이다.

2.13. 프로세스의 생성

리눅스에서의 쓰레드의 구현은 task의 구현과 다르지 않다. 즉, 리눅스에서는 프로세스와 쓰레드를 같은 task_struct를 사용해서 나타낸다. 쓰레드를 생성한 프로세스의 주소공간과 file descriptor 및 signal handler를 등등의 정보를 공유하게 된다. 심지어 프로세스 ID까지도 공유가 가능하다. 이와 같은 것들을 가능하게 하기 위해서 리눅스에서는 fork()의 특별한 형태인 clone()라는 함수를 가지고 있다. fork()와 clone() 모두 ~/kernel/fork.c에 있는 do_fork()를 호출한다.

```

asmlinkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &regs, 0);
}

asmlinkage int sys_clone(struct pt_regs regs)
{
    unsigned long clone_flags;
    unsigned long newsp;

    clone_flags = regs.ebx;
    newsp = regs.ecx;
    if (!newsp)
        newsp = regs.esp;
    return do_fork(clone_flags, newsp, &regs, 0);
}

```

코드 25. fork와 clone의 system call interface

do_fork()에서 하는 일은 시스템 프로세스의 정보를 복사하고, 적절하게 각 레지스터들을 설정한 한다. 데이터 세그먼트의 내용을 전체를 복사하고, 스택으로 사용할 영역을 지정해주게 된다. 코드를 보면서 하나씩 설명하도록 하자.

```

int do_fork(unsigned long clone_flags, unsigned long stack_start,
           struct pt_regs *regs, unsigned long stack_top)
{
    int retval = -ENOMEM;
    struct task_struct *p;
    DECLARE_MUTEX_LOCKED(sem);

    if (clone_flags & CLONE_PID) {

```

```

/* This is only allowed from the boot up thread */
if (current->pid)
    return -EPERM;
}

```

코드 26. do_fork() 함수 – PID의 복제확인

먼저 PID(Process ID)를 복제하는지를 확인한다. 이는 시스템이 부팅할 때만 프로세스 ID를 복사할 수 있도록 하기 때문에, 만약 프로세스 ID가 0이 아닌 프로세스가 이것을 호출했다면, 에러를 돌려준다.

```

current->vfork_sem = &sem;

p = alloc_task_struct();
if (!p)
    goto fork_out;

*p = *current;

retval = -EAGAIN;
if (atomic_read(&p->user->processes) >= p->rlim[RLIMIT_NPROC].rlim_cur)
    goto bad_fork_free;
atomic_inc(&p->user->_count);
atomic_inc(&p->user->processes);
if (nr_threads >= max_threads)
    goto bad_fork_cleanup_count;

if (p->exec_domain && p->exec_domain->module)
    __MOD_INC_USE_COUNT(p->exec_domain->module);
if (p->bifmt && p->bifmt->module)
    __MOD_INC_USE_COUNT(p->bifmt->module);

p->did_exec = 0;
p->swappable = 0;
p->state = TASK_UNINTERRUPTIBLE;

```

코드 27. do_fork() 함수 (계속) – task_struct의 할당

세마포어¹⁷를 획득한 후, task_struct를 할당 받고, 현재 프로세스의 내용을 복사해 넣어둔다. 그리고 나서 자원 사용한계를 확인하고, 새로이 생성된 프로세스의 사용자와 관련된 count값을 증가 시킨다. 물론 이때는 인터럽트가 허용되지 않는 atomic increment를 사용해 주어야 한다.

실행하려는 프로세스의 실행 domain과 이진 포맷의 모듈 사용 카운터의 값을 증가 시켜주고, 아직 exec system call을 하지 않았다는 것과 swap되지 않았다는 것을 표시한 다음, 현재 프로세스의 상태를 TASK_UNINTERRUPTIBLE로 둔다. 이는 아직 완전히 생성되지 않았기에 이벤트들에 의해서 반응하지 않도록 만들어주는 것이다.

```

copy_flags(clone_flags, p);
p->pid = get_pid(clone_flags);

p->run_list.next = NULL;
p->run_list.prev = NULL;

if ((clone_flags & CLONE_VFORK) || !(clone_flags & CLONE_PARENT)) {
    p->p_opptr = current;
}

```

¹⁷ 커널의 동기화 기법 중 하나로, 프로세스가 현재 어떤 자원이 사용 가능한지를 표현하는 방법이다. 만약 다른 프로세스에 의해서 자원이 점유중이라면 사용하려는 프로세스는 그 사용이 끝 날 때까지 실행을 미루어야 한다. 물론 사용 가능한 자원의 개수는 여러 개가 될 수도 있다.

```

    if (!(p->ptrace & PT_PTRACED))
        p->p_pptr = current;
}
p->p_cptr = NULL;

```

코드 28. do_fork() 함수 (계속) – 프로세스들간의 관계설정

프로세스의 PID를 결정해주고, run list를 초기화 시켜준다. 프로세스 tree구조를 만들어 주기 위한 초기화 작업도 해준다. 원래(original)의 부모 프로세스를 가르키는 포인터를 현재의 프로세스로 설정하고, 만약 새로운 프로세스가 디버깅을 위해서 Trace되고 있지 않다면, 부모 프로세스를 가르키는 포인터도 현재의 프로세스를 가르키도록 설정한다. 그리고, 자식 프로세스는 아직 없으므로 NULL로 초기화 한다.

여기서 clone flag이 자주 나오는데, 이는 ~/include/linux/sched.h에 정의된 값으로 아래와 같은 것이 될 수 있다.

이름	값	설명
CLONE_VM	0x00000100	프로세스간에 가상 메모리를 공유한다.
CLONE_FS	0x00000200	프로세스간에 file system 정보를 공유한다.
CLONE_FILE	0x00000400	프로세스간에 열린 파일들을 공유한다.
CLONE_SIGHAND	0x00000800	프로세스간에 signal handler들과 block된 signal을 공유한다.
CLONE_PID	0x00001000	프로세스간에 프로세스 ID를 공유한다.
CLONE_PTRACE	0x00002000	새로 생성될 프로세스도 debugging을 위해 trace되기를 원할 경우에 사용한다.
CLONE_VFORK	0x00004000	자식 프로세스가 mm_release를 할 경우, 부모 프로세스가 깨어나기를 원한다면 설정한다.
CLONE_PARENT	0x00008000	새로운 생성될 프로세스가 현재 생성하고 있는 프로세스와 같은 부모를 가지도록 만들고 싶을 경우에 사용한다.
CLONE_THREAD	0x00010000	같은 Thread group에 속하게 될지를 결정한다.
CLONE_SIGNAL		CLONE_SIGHAND CLONE_THREAD

표 6. clone_flag의 값

예를 들어서 새로이 쓰레드를 생성하고자 한다면, clone에서 CLONE_THREAD로 주면 될 것이다. 즉, 어떤 것들을 새로 생성되는 프로세스와 공유할 것인가를 결정하는 역할을 한다.

```

init_waitqueue_head(&p->wait_chldexit);
p->vfork_sem = NULL;
spin_lock_init(&p->alloc_lock);
p->sigpending = 0;
init_sigpending(&p->pending);

```

코드 29. do_fork() 함수 (계속) -

그리고나서 프로세스의 wait4() 시스템 콜을 위한 wait queue 및, vfork_sem, alloc_lock, sigpending, pending 필드들에 대한 초기화를 한다.

```

p->it_real_value = p->it_virt_value = p->it_prof_value = 0;
p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0;
init_timer(&p->real_timer);
p->real_timer.data = (unsigned long) p;
p->leader = 0; /* session leadership doesn't inherit */
p->tty_old_pgrp = 0;
p->times.tms_utime = p->times.tms_stime = 0;
p->times.tms_cutime = p->times.tms_cstime = 0;
#endif CONFIG_SMP

```

```

{
    int i;
    p->has_cpu = 0;
    p->processor = current->processor;
    /* ?? should we just memset this ?? */
    for(i = 0; i < smp_num_cpus; i++)
        p->per_cpu_utime[i] = p->per_cpu_stime[i] = 0;
    spin_lock_init(&p->sigmask_lock);
}
#endif
p->lock_depth = -1;           /* -1 = no lock */
p->start_time = jiffies;

```

코드 30. do_fork() 함수 (계속) – 시간 값들에 대한 초기화 및 프로세스의 그룹에 대한 초기화

이제는 프로세스의 프로파일링(profiling)위한 각종 정보들을 초기화하는 부분이다. 또한 프로세스가 속한 session에 대한 초기화 및 SMP 기계인 경우에는 각각의 CPU당 사용자 시간(user time)과 시스템(system time)을 초기화해 준다. 이와 같은 일이 끝나면, 새로이 생성된 프로세스의 시작시간(start time)을 기록한다.

```

retval = -ENOMEM;
/* copy all the process information */
if (copy_files(clone_flags, p))
    goto bad_fork_cleanup;
if (copy_fs(clone_flags, p))
    goto bad_fork_cleanup_files;
if (copy_sighand(clone_flags, p))
    goto bad_fork_cleanup_fs;
if (copy_mm(clone_flags, p))
    goto bad_fork_cleanup_sighand;
retval = copy_thread(0, clone_flags, stack_start, stack_top, p, regs);
if (retval)
    goto bad_fork_cleanup_sighand;
p->semundo = NULL;

```

코드 31. do_fork() 함수 (계속) – 부모로부터 필요한 필드의 복제

부모 프로세스의 각종 정보들을 복사하는 부분이다. 파일 디스크립터 및, 파일 시스템에 대한 정보와 시그널 핸들러, 메모리, 스택과 레지스터 값을 복제한다. 그리고나서 아직 아무런 semaphore와 관련된 동작을 하지 않았으므로 semundo를 NULL값으로 초기화한다.

```

p->parent_exec_id = p->self_exec_id;
p->swappable = 1;
p->exit_signal = clone_flags & CSIGNAL;
p->pdeath_signal = 0;

```

코드 32. do_fork() 함수 (계속) – 실행 도메인 및 스왑, 종료에 관련된 시그널 초기화

여기까지 진행했다면, 이미 생성될 프로세스에 대해서 메모리 자원이 할당된 상태이다. 해당 실행 도메인의 초기화와 프로세스의 스왑(swap) 가능성 및, 종료상황에서 보내고자 하는 시그널(exit_signal), 부모 프로세스가 종료될 때 받을 시그널을 초기화 한다.

```

p->counter = (current->counter + 1) >> 1;
current->counter >= 1;
if (!current->counter)
    current->need_resched = 1;

```

코드 33. do_fork() 함수(계속) – 실행 우선순위의 할당

프로세스의 우선순위(counter)값을 정해주는 단계에서는 자식은 부모 프로세스가 가진 우선 순위를 반으로 나누어서 가지도록 만든다. 이것은 높은 우선순위의 프로세스가 자식 프로세스를 많이 거느리게 될 경우, 관련되지 않은 낮은 우선순위의 프로세스들에게 미치는 영향을 줄이기 위한 노력이다. 만약 부모 프로세스의 우선순위가 위와 같은 과정에서 0값을 가진다면 새로운 프로세스를 실행하도록 스케줄링을 요청한다(need_resched=1).

```

retval = p->pid;
p->tgid = retval;
INIT_LIST_HEAD(&p->thread_group);
write_lock_irq(&tasklist_lock);
if (clone_flags & CLONE_THREAD) {
    p->tgid = current->tgid;
    list_add(&p->thread_group, &current->thread_group);
}
SET_LINKS(p);
hash_pid(p);
nr_threads++;
write_unlock_irq(&tasklist_lock);

```

코드 34. PID와 thread 그룹의 초기화

자신이 돌려받을 값(프로세스의 ID)과 쓰레드그룹의 아이디를 설정한 다음, 자신이 가지는 쓰레드 그룹의 리스트를 초기화 한다. 이젠 생성한 프로세스를 태스크의 리스트에 삽입할 차례이다. SET_LINKS()가 init_task로 시작하는 태스크의 연결 리스트에 생성한 프로세스를 삽입한다. 그리고, 프로세스의 아이디를 가지고 찾을 때 사용하도록 hash 테이블에도 등록하게 되며, 생성된 프로세스의 수(nr_threads)를 증가 시켜준다.

프로세스가 생성되면, 새로 생성된 프로세스는 부모 프로세스와 같은 상황에서 수행된다. 따라서, 부모와 자식을 구분할 수 있는 방법이 필요한데, 이것은 시스템 콜에서 돌려받는 값으로 확인하게 된다. 부모 프로세스의 경우에는 자식 프로세스의 PID값을 돌려받게 되며, 자식 프로세스는 0을 받는다.

```

if (p->ptrace & PT_PTRACED)
    send_sig(SIGSTOP, p, 1);
wake_up_process(p); /* do this last */
++total_forks;
fork_out:
if ((clone_flags & CLONE_VFORK) && (retval > 0))
    down(&sem);
return retval;

```

코드 35. do_fork()함수(계속) – do_fork()함수의 종료 상황 처리

만약 프로세스가 현재 trace가 되고 있다면, 멈추도록(send_sig())한다. 그렇지 않을 경우에는 새로 생성될 프로세스를 깨운 후, 마지막으로 전체 fork한 횟수를 증가 시켜주고나서 함수를 복귀하게 된다.

```

bad_fork_cleanup_sighand:
exit_sighand(p);
bad_fork_cleanup_fs:
exit_fs(p); /* blocking */
bad_fork_cleanup_files:
exit_files(p); /* blocking */
bad_fork_cleanup:
put_exec_domain(p->exec_domain);
if (p->binfo && p->binfo->module)
    __MOD_DEC_USE_COUNT(p->binfo->module);
bad_fork_cleanup_count:
atomic_dec(&p->user->processes);

```

```

    free_uid(p->user);
bad_fork_free:
    free_task_struct(p);
    goto fork_out;
}

```

코드 36. do_fork()함수(계속) – 에러에 대한 처리

프로세스의 생성과 관련된 마지막은 앞에서 생성도중에 생긴 문제들에 대한 해결일 것이다. 즉, 원래 상태로의 복귀및 에러코드의 리턴이 되겠다. 이와 같은 에러는 주로 복사중에 일어나는 문제들로 해당하는 에러 코드를 현재의 프로세스에 알려주어야 할 것이다.

지금까지는 do_fork()함수를 이용해서 새로이 프로세스를 생성하는 과정에 대해서 알아보았다. 이것은 단순히 부모 프로세스의 복제본을 만드는 것에 지나지않다. 다음으로 볼것은 이렇게 생성된 프로세스를 이용해서 부모 프로세스와는 다른 프로그램의 실행 이미지를 만드는 것을 볼 것이다.

2.14. 프로세스의 실행

프로세스는 생성시에는 부모의 실행이미지를 그대로 가지고 실행상태에 있게된다. 따라서, 새로운 프로그램을 실행하려고 할때는 넘겨받은 부모의 실행이미지를 버리고, 다시 자신만의 실행이미지를 가져야 한다. 즉, 디스크로부터 새로이 실행될 프로그램을 불러오는 과정이 필요하다.

프로세스를 실행하기 위해서는 시스템 콜 ~/arch/i386/kernel/process.c에 있는 sys_execve()를 사용한다. 이는 다시 내부적으로 do_execve()함수를 호출하게 되어있다. 코드는 아래와 같다.

```

asmlinkage int sys_execve(struct pt_regs regs)
{
    int error;
    char * filename;

    filename = getname((char *)regs.ebx);
    error = PTR_ERR(filename);
    if (IS_ERR(filename))
        goto out;
    error = do_execve(filename, (char **)regs.ecx, (char **)regs.edx, &regs);
    if (error == 0)
        current->ptrace &= ~PT_DTRACE;
    putname(filename);
out:
    return error;
}

```

코드 37. sys_execve()함수의 정의 – exec 시스템 콜

넘겨 받은 레지스터에서 해당하는 프로그램의 파일 이름을 찾은 다음, do_execve()함수를 호출하고, 현재의 프로세스의 PT_DTRACE flag를 off시킨다. 파일 이름을 가지고 있는 dentry¹⁸의 구조체를 없애고 do_execve()의 복귀값을 돌려준다.

실제적인 실행과 관련된 일은 ~/fs/exec.c의 do_execve()에서 일어난다. 가장먼저 하는 일은 디스크 상의 파일을 여는 일일 것이다. 여기서 한가지 실제로 사용자의 입장에서 보면, 여러가지의 형태의 exec()시스템 콜이 정의되어 있으나, 이것은 프로세스가 어떤 환경에서 실행될 것인가의 차이만을 가지는 것으로 결과적으로는 do_exec()가 호출되게 된다.

```

int do_execve(char * filename, char ** argv, char ** envp, struct pt_regs * regs)
{

```

¹⁸ 이것은 나중에 파일 시스템을 보게 될 때 설명되겠지만, 우선은 디렉토리의 엔트리들을 보관하는 메모리 캐쉬라고 일단은 생각하도록 한다.

```

struct linux_binprm bprm;
struct file *file;
int retval;
int i;

file = open_exec(filename);

retval = PTR_ERR(file);
if (IS_ERR(file))
    return retval;

```

코드 38. do_execve() 함수 – 파일 열기

넘겨받은 파일이름에 대해서 열기(open_exec())를 하고, 열려진 파일에 대해서 에러가 있는지를 검사한다. 만약 에러가 있다면 곧바로 복귀하고 그렇지 않다면 아래로 진행한다.
여기서 잠시 리눅스가 바이너리를 실행하기 위해서 사용하는 linux_binprm구조체에 대해서 살펴보기로 하자. 아래와 같이 정의되어 있다.

```

struct linux_binprm{
    char buf[BINPRM_BUF_SIZE];
    struct page *page[MAX_ARG_PAGES];
    unsigned long p; /* current top of mem */
    int sh_bang;
    struct file * file;
    int e_uid, e_gid;
    kernel_cap_t cap_inheritable, cap_permitted, cap_effective;
    int argc, envc;
    char * filename; /* Name of binary */
    unsigned long loader, exec;
};

```

코드 39. linux_binprm구조체의 정의

각각의 필드들에 대한 정의는 아래와 같다.

Field	Description
char buf[BINPRM_BUF_SIZE]	실행 파일의 첫 128bytes를 가진다. 이 부분은 실행화일의 포맷(format)을 나타내는 정보를 가진다. 가령 예를 들어서 ELF파일의 포맷을 나타내는 magic number등이 있다.
struct page *page[MAX_ARG_PAGES]	실행 argument들을 보관할 page들에 대한 포인터의 배열
unsigned long p	현재 메모리의 top
int sh_bang	현재 실행하고자 하는 파일이 shell script인지의 여부를 나타냄
struct file *file	열려진 파일(실행할 파일)에 대한 포인터
int e_uid, e_gid	Effective user ID와 effective group ID(inode로 부터 넘겨 받음)
kernel_cap_t cap_inheritable, cap_permitted, cap_effective	Capability를 나타내는 필드
int argc, envc	Argument cout와 environment count
char *filename	실행하고자 하는 파일의 이름
unsigned long loader, exec	실행화일의 loader와 exec domain

표 7. linux_binprm 구조체의 필드 정의

linux_binprm구조체는 실행할 프로그램에 대한 정보를 넘겨 받아서 저장하며, 실행하기 위해서 필요한 로더(loader)나 번역기(interpreter) 프로그램을 알기위해서 사용한다.

```
bprm.p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);
memset(bprm.page, 0, MAX_ARG_PAGES*sizeof(bprm.page[0]));
bprm.file = file;
bprm.filename = filename;
bprm.sh_bang = 0;
bprm.loader = 0;
bprm.exec = 0;
```

코드 40. do_execve()함수(계속) – linux_binprm구조에의 초기화

바이너리(binary) 파일을 실행하기 위한 절차로서, 먼저 넘겨 받은 argument들을 저장하기 위한 linux_binprm 구조체를 초기화 한다. 즉, argument에 대한 저장공간의 초기화와 파일이름 및 파일 포인터, loader와 exec필드에 대한 초기화를 담당한다.

```
if ((bprm.argv = count(argv, bprm.p / sizeof(void *))) < 0) {
    allow_write_access(file);
    fput(file);
    return bprm.argv;
}
if ((bprm.envc = count(envp, bprm.p / sizeof(void *))) < 0) {
    allow_write_access(file);
    fput(file);
    return bprm.envc;
}
retval = prepare_binprm(&bprm);
if (retval < 0)
    goto out;
retval = copy_strings_kernel(1, &bprm.filename, &bprm);
if (retval < 0)
    goto out;
bprm.exec = bprm.p;
retval = copy_strings(bprm.envc, envp, &bprm);
if (retval < 0)
    goto out;
retval = copy_strings(bprm.argv, argv, &bprm);
if (retval < 0)
    goto out;
retval = search_binary_handler(&bprm,regs);
if (retval >= 0)
    /* execve success */
    return retval;
```

코드 41. do_execve()함수(계속) – 넘겨받는 변수의 검사와 linux_binprm의 초기화 및 binary format에 대한 핸들러(handler)구하기

실행을 위해서 넘겨받은 argument와 환경(environment) 변수들(argv, envp)의 count값을 저장하고, 파일과 관련된 inode로부터 실행과 관련된 정보를 넘겨받는다(prepare_binprm()). 그리고, 실행하려는 파일의 이름과 현재 최상위 메모리가 어딘가를 표시한 다음, argument 및 환경 변수들을 저장한다. 그리고나서 이진 포맷(binary format)에 대해서 핸들러를 검색해서, 현재 실행하고자 하는 이진(binary) 파일을 다루어줄 프로그램에 대한 정보를 찾는다. 여기까지 모든 과정이 제대로 되었다면 exec()는 성공했으며, 복귀하면 된다.

```
out:
allow_write_access(bprm.file);
if (bprm.file)
    fput(bprm.file);
for (i = 0 ; i < MAX_ARG_PAGES ; i++) {
```

```

    struct page * page = bprm.page[i];
    if (page)
        __free_page(page);
}
return retval;
}

```

코드 42. do_execve() 함수(계속) – 예상상황의 처리

뭔가 실행에 관련된 연산에서 에러가 있을 경우 실행되는 부분이다. 주로 하는 일은 할당받았던 자원을 운영체제로 돌려주는 것이다. 관련된 것은 실행하려는 파일과 관련된 inode 및 argument를 위한 페이지들을 놓아준다(release).

2.15. 프로세스의 종료

프로세스의 마지막이 종료이다. 이것은 exit() 시스템 콜을 사용해서 가능하다¹⁹. 프로세스의 종료는 생성시에 했던 일들에 대해서 반대로 해주면 될 것이다. 즉, 할당받은 시스템의 자원이 있다면, 이것을 놓아(release)주고, 자신이 수행상황에 대한 각종 정보를 커널에 알려주게 되며, 프로세스의 종료 정보를 부모 프로세스로 넘겨준다. 마지막으로 다른 프로세스가 수행될 수 있도록 스케줄링을 요청하게 된다.

```

NORET_TYPE void do_exit(long code)
{
    struct task_struct *tsk = current;
    if (in_interrupt())
        panic("Aiee, killing interrupt handler!");
    if (!tsk->pid)
        panic("Attempted to kill the idle task!");
    if (tsk->pid == 1)
        panic("Attempted to kill init!");
    tsk->flags |= PF_EXITING;
    del_timer_sync(&tsk->real_timer);

fake_volatile:
#endif CONFIG_BSD_PROCESS_ACCT
    acct_process(code);
#endif

```

코드 43. do_exit() 함수 – PID의 확인과 timer지우기

먼저, exit() 시스템 콜을 사용하는 프로세스가 인터럽트가 진행중인지를 확인하고, 다시 시스템의 idle task인지와 init 프로세스인지를 확인한다. 그리고 나선 프로세스가 EXIT하고 있음을 표시(PF_EXITING)한 다음, 동적 타임어 큐(dynamic timer queue)로부터 프로세스의 real_timer를 제거한다. 그리고, 만약 BSD 프로세스 account를 사용한다면 해당하는 함수를 불러준다.

```

__exit_mm(tsk);
lock_kernel();
sem_exit();
__exit_files(tsk);
__exit_fs(tsk);
exit_sighand(tsk);
exit_thread();
if (current->leader)
    disassociate_ctty(1);
put_exec_domain(tsk->exec_domain);

```

¹⁹ 모든 프로그램은 제어가 프로그램의 마지막에 도달하면, 인위적인 exit()를 호출하지 않더라도, exit() 시스템 콜이 자동으로 호출되도록 만들어준다.

```

if (tsk->binfo && tsk->binfo->module)
    __MOD_DEC_USE_COUNT(tsk->binfo->module);
tsk->exit_code = code;
exit_notify();
schedule();
BUG();
goto fake_volatile;
}

```

코드 44. do_exit()함수(계속) – 할당 받은 자원의 해제와 종료상황에 대한 보고

프로세스와 관련된 메모리와 세마포어, 파일 및 파일 시스템정보, 시그널처리, 그리고, 쓰레드에 대한 정보들을 해제(release)하고²⁰, 종료하는 현재의 프로세스가 터미널(terminal)을 가지고 있다면, 이것 역시 떼어낸다. 그리고나서 프로세스를 실행하기 위해서 필요한 실행 도메인(execution domain)을 가지고 있는 모듈에 대한 count값을 낮추는 put_exec_domain()을 호출하고나서, 다시 이진 포맷(binary format)과 관련된 모듈도 사용값을 낮춘다. 전달받은 종료코드(exit code)값을 저장한 다음, 종료 중인 프로세스와 관련된 프로세스들에게 자신이 종료하고 있음을 알려준다(exit_notify()). 프로세스의 상태도 ZOMBIE 상태로 바뀐다. 마지막으로 새로운 프로세스를 스케줄링 하도록 알려주게 되며, 현재의 프로세스는 실행리스트(run list)에서 제외되어 있으므로 더 이상 스케줄링 받지 않게 된다.

ZOMBIE상태는 부모 프로세스에 종료 상태를 알리지 않은 프로세스의 상태를 말하는 것으로 이는 부모 프로세스가 기동해서 적절한 행동을 취해야 한다. 하지만, 만약 부모 프로세스가 먼저 종료했을 경우에는 어떻게 될 것인가? 이와 같은 경우는 쉘에서 프로그램을 back ground로 실행하고 쉘을 종료하는 경우가 될 수 있는데, 이렇게 되면 남은 프로세스의 부모 프로세스는 init 프로세스가 된다. 즉, init 프로세스가 부모를 잃은 프로세스가 종료를 취급하게 된다.

이상에서 프로세스의 생성과 실행, 종료에 대해서 알아보았다. 프로세스는 실행되기 위해서 시스템의 자원을 필요로 하며, 종료 시에는 이렇게 할당된 자원들을 다시 반환해야 한다. 모든 프로세스는 부모를 가지게 되며, 종료된 프로세스를 부모로 가진 프로세스는 init 프로세스를 새로운 부모로 가진다. 또한 프로그램은 실행될 수 있는 format을 인식하는 관련된 로더(loader)가 존재하며, 이러한 포맷(format)의 예로는 ELF와 A.OUT형식 등이 존재한다.

2.16. Select and Poll

Select와 Poll은 비동기적인 입출력을 구현하기 위한 두가지의 방법이다. Select는 일반적으로 특정 디바이스나 파일과 관련된 descriptor가 입력/출력/애러 등과 같은 상황을 만날 때까지 현재의 process(혹은 thread)를 대기 시키는 것에 사용되며²¹, poll은 주기적으로 process(혹은 thread)가 앞에서 이야기한 디스크립터의 상태를 확인하는 일을 해준다. 이 두가지의 system call은 응용 프로그램에서 입출력과 관련된 blocking문제를 해결하기 위해서 많이 사용하고 있으며, 디바이스 드라이버를 구현하는데 있어서도 필요한 부분이 되기에 여기서 살펴보고 넘어가도록 하겠다. 참고로 둘다 ~/linux/fs/select.c에 구현되어 있다.

2.16.1. Poll의 구현

먼저 poll의 구현을 보도록 하겠다. 실제로 poll과 select의 구현은 그다지 다르지 않다. 즉, 내부적으로 같은 메커니즘을 사용하고 있음을 확인할 수 있을 것이다. Poll의 application interface가 되는 부분이 바로 sys_poll() 함수이다. 본격적인 분석에 들어가기에 앞어서, 먼저 poll과 select에서 사용되는 data structure에 대해서 알아보도록 하자. 아래와 같이 ~/linux/fs/select.c에 정의된 구조체와 매크로 들이 있다.

```

#define ROUND_UP(x,y) (((x)+(y)-1)/(y))
#define DEFAULT_POLLMASK (POLLIN | POLLOUT | POLLRDNORM | POLLWRNORM)

struct poll_table_entry {

```

²⁰ 물론 만약 다른 프로세스가 자료를 공유하고 있는 상태라면 실제적인 해제는 일어나지 않을 것이다.

²¹ 물론 timeout 필드를 두어서, 이와 같은 대기 시간을 조절해 줄 수도 있다.

```

    struct file * filp;
    wait_queue_t wait;
    wait_queue_head_t * wait_address;
};

struct poll_table_page {
    struct poll_table_page * next;
    struct poll_table_entry * entry;
    struct poll_table_entry entries[0];
};

#define POLL_TABLE_FULL(table) \
((unsigned long)((table)->entry+1) > PAGE_SIZE + (unsigned long)(table))

```

코드 45. select.c 파일의 macro와 structure 정의

ROUND_UP()은 x의 값을 y만큼을 올림하는 역할을 하며, DEFAULT_POLLMASK()는 기본적으로 설정될 poll에 대한 mask값이다. mask값은 다시 아래와 같은 정의를 가진다.

Flag	Value	Description
POLLIN	0x0001	파일에 대해서 read할 준비가 되었다. 즉, 파일에 대한 read는 blocking을 일으키지 않는다는 것을 보장한다.
POLLPRI	0x0002	우선 순위가 높은 데이터에 대해서 읽기가 준비되었다. Select의 경우에는 이 flag이 파일에 대해서 예외상황(exception condition)을 발생했음을 알리기 위해서 사용한다. 즉, select는 응급 데이터(out-of-band data)를 예외 상황으로 처리한다.
POLLOUT	0x0004	파일에 대해서 write할 준비가 되었다. 즉, 파일에 대한 write가 blocking을 일으키지 않는다는 것을 보장한다.
POLLERR	0x0008	파일에 대해서 error가 발생했다는 것을 알려준다. Poll이 호출된다면, 나중에 보면 알 수 있겠지만, 파일이 읽거나 혹은 쓸 수 있는 상태가라고 보고된다. 즉, read/write는 에러코드를 blocking시키지 않고 전달하기 때문이다.
POLLHUP	0x0010	End of file을 알려준다. 즉, 파일이 hang-up되었다는 것을 알려준다. Select를 호출한 프로세스의 경우에는 파일이 읽을 수 있는 상태에 있다고 알려준다. 하지만, 나중에 알 수 있듯이 읽기 연산에서 돌려받는 데이터의 길이는 0이 될 것이다.
POLLNVAL	0x0020	유효하지 않은 파일에 대한 poll을 했을 경우에 돌려준다.
POLLRDNORM	0x0040	일반적인 데이터가 읽기를 위해서 준비된 경우이다. 즉, 앞에서 설명한 POLLIN과 같이 사용되어, 일반적인 데이터가 읽기를 위해서 준비되었음을 알려준다.
POLLRDBAND	0x0080	파일로부터 응급 데이터(out-of-band data)가 읽기를 위해서 준비된 경우이다. 현재는 Linux 커널에서 잘 사용하고 있지 않다.
POLLWRNORM	0x0100	POLLRDNORM과 마찬가지로 일반적인 데이터의 쓰기가 준비된 경우이다. 마찬가지로 POLLOUT과 같이 사용되어 데이터의 쓰기가 준비되었음을 알려준다.
POLLWRBAND	0x0200	POLLRDBAND와 마찬가지로 우선순위가 있는 데이터가 파일에 쓸 수 있음을 알려준다. Datagram과 같은 구현에 있어서 사용되며, out-of-band 데이터를 전송하기 위함이다. 이것에 관련된 것을 보려면 TCP/IP에 대한 책을 보기바란다.
POLLMSG	0x0400	SIGIO signal을 구현하기 위해서 사용된다. 현재로서는 POLLIN, POLLRDNORM과 같이 사용되어, read할 데이터가 있음을 알려주는 역할만 한다.

표 8. Poll의 설정 flag에 대한 정의

따라서, DEFAULT_POLLMASK에서는 read/write에 대한 일반적인 데이터가 있음을 나타내준다고 볼 수 있겠다. poll_table_entry 구조체는 poll_table에 등록될 하나의 entry를 위한 자료구조이다. 즉, poll에서는 poll_table에 등록된 하나의 entry에 대한 연산을 처리하는 것이 주 임무이다. poll_table_page 구조체는 poll_table을 위해서 할당된 page를 구성하기 위한 자료구조이다.

poll_table_entry 구조체를 보면, polling과 관련된 file의 file 구조체에 대한 포인터와 프로세스들(혹은 thread들)의 대기(wait)를 위한 queue, 그리고, wait queue의 head를 나타내는 포인터를 가진다. poll_table_page는 poll_table을 위해서 할당된 다음 page에 대한 포인터와 poll_table의 entry의 시작을 나타내는 포인터 및 entry의 배열로 정의된다. 각각에 대한 사용은 이하에서 다시 살펴볼 것이다. POLL_TABLE_FULL() 매크로는 현재의 entry를 증가시킨 값이 하나의 page크기를 벗어나는지를 보고, poll_table을 위해서 할당된 page를 다 썼는가를 확인하는 것이다.

자, 이것으로 대략적으로 poll에 사용될 매크로와 데이터 구조체에 대해서 살펴보았다. 이제 본격적인 논의로 들어가도록 하겠다. sys_poll()을 보도록 하자.

```
asmlinkage long sys_poll(struct pollfd * ufds, unsigned int nfds, long timeout)
{
    int i, j, fdcount, err;
    struct pollfd **fds;
    poll_table table, *wait;
    int nchunks, nleft;

    /* Do a sanity check on nfds ... */
    if (nfds > NR_OPEN)
        return -EINVAL;

    if (timeout) {
        /* Careful about overflow in the intermediate values */
        if ((unsigned long) timeout < MAX_SCHEDULE_TIMEOUT / HZ)
            timeout = (unsigned long)(timeout*HZ+999)/1000+1;
        else /* Negative or overflow */
            timeout = MAX_SCHEDULE_TIMEOUT;
    }

    poll_initwait(&table);
    wait = &table;
    if (!timeout)
        wait = NULL;
```

코드 46. sys_poll() 함수의 정의

sys_poll() 함수가 넘겨받는 인자는 파일 디스크립터와 event들을 가진는 pollfd와 polling하게 될 fd의 개수를 나타내는 nfds, 그리고 마지막으로 timeout값을 나타내는 timeout이 있다. 먼저 넘겨받은 nfds가 NR_OPEN(=1024)보다 크다면 -EINVAL을 돌려준다. 즉, 프로세스나 thread가 열 수 있는 file의 개수를 초과했다는 것이다. timeout 값이 있는 경우에는 MAX_SCHEDULE_TIMEOUT(= ~0UL >> 1)을 HZ(=100)으로 나눈값과 비교해서 작다면 timeout값에 HZ를 곱하고, 다시 999를 더한후 1000으로 나눠주고 1을 더한다. 이것은 1000단위로 반올림해 주기 위한 것이다. 만약 timeout이 MAX_SCHEDULE_TIMEOUT을 2로 나눈 값보다 크다면, overflow가 날 가능성이 있기에 MAX_SCHEDULE_TIMEOUT으로 변경한다. 실제로 MAX_SCHEDULE_TIMEOUT은 커널에 있는 comment를 보면 대략 Intel 계열에서 200일 정도의 값이 된다는 것을 확인할 수 있을 것이다. poll_initwait() 함수는 poll_table_struct 구조체로 선언된 table을 초기화 시켜준다. poll_table_struct는 아래와 같은 정의를 가진다.

```
typedef struct poll_table_struct {
```

```

int error;
struct poll_table_page * table;
} poll_table;
}

```

코드 47. poll_table_struct 구조체의 정의

즉, 발생한 error와 poll_table_page 구조체를 가르키는 table 필드가 있다. 이 두 필드에 0과 NULL값을 주는 것이 poll_initwait() 함수가 하는 일이다.

wait는 이 table의 주소를 가지도록 만들어주고, 만약 timeout값이 0으로 설정된 경우에는 대기 할 필요가 없으므로 wait에는 NULL을 준도록 한다.

```

err = -ENOMEM;
fds = NULL;
if (nfd != 0) {
    fds = (struct pollfd ** )kmalloc(
        (1 + (nfd - 1) / POLLFD_PER_PAGE) * sizeof(struct pollfd *),
        GFP_KERNEL);
    if (fds == NULL)
        goto out;
}
nchunks = 0;
nleft = nfd;
while (nleft > POLLFD_PER_PAGE) { /* allocate complete PAGE_SIZE chunks */
    fds[nchunks] = (struct pollfd *)__get_free_page(GFP_KERNEL);
    if (fds[nchunks] == NULL)
        goto out_fds;
    nchunks++;
    nleft -= POLLFD_PER_PAGE;
}
if (nleft) { /* allocate last PAGE_SIZE chunk, only nleft elements used */
    fds[nchunks] = (struct pollfd *)__get_free_page(GFP_KERNEL);
    if (fds[nchunks] == NULL)
        goto out_fds;
}
}

```

코드 48. sys_poll() 함수의 정의(계속)

err에는 -ENOMEM을 설정해서 error 코드를 돌려줄 경우를 대비하고, fds에는 NULL을 두어 초기화 해준다. 만약 nfd가 0이 아닌 값을 지닌다면, poll을 위한 pollfd구조체를 nfd 만큼 할당받기 위해서 kmalloc()를 호출한다. POLLFD_PER_PAGE는 #define POLLFD_PER_PAGE ((PAGE_SIZE) / sizeof(struct pollfd))로 정의된 것으로 page당 할당될 수 있는 pollfd의 개수를 나타낸다. 만약 할당받을 수 없다면(fds == NULL) out으로 진행하고, 복귀 에러값은 앞에서 설정한 -ENOMEM이 될 것이다. nchunks는 한번에 보게될 pollfd 구조체들의 개수가 몇개가 되는지를 나타낸다. 초기에는 0으로 주고, while loop를 돌면서 차츰 그 값이 증가한다. fds[] 배열은 이미 앞에서 pollfd 구조체를 가질 수 있는 포인터의 배열로 할당되었다. 이전 그 자리에 실제적으로 pollfd구조체를 할당하기 위해서 __get_free_page()를 호출했다. 만약 할당받을 수 없다면(fds[] == NULL) out_fds로 제어를 옮긴다. 그렇지 않다면, nchunks를 증가시켜주고, nleft가 가지는 nfd의 갯수를 페이지당 pollfd(POLLFD_PER_PAGE) 만큼씩 감소시킨다. 위에서 페이지 단위로 pollfd를 할당받았고, 남은 page크기보다 작은 개수를 처리하기 위해서 다시 하나의 페이지를 할당해서 fds[] 배열을 초기화 해준다.

```

err = -EFAULT;
for (i=0; i < nchunks; i++)
    if (copy_from_user(fds[i], ufd + i * POLLFD_PER_PAGE, PAGE_SIZE))
        goto out_fds1;
if (nleft) {
    if (copy_from_user(fds[nchunks], ufd + nchunks * POLLFD_PER_PAGE,

```

```

        nleft * sizeof(struct pollfd)))
    goto out_fds1;
}

fdcount = do_poll(nfds, nchunks, nleft, fds, wait, timeout);

/* OK, now copy the revents fields back to user space. */
for(i=0; i < nchunks; i++)
    for (j=0; j < POLLFD_PER_PAGE; j++, ufds++)
        __put_user((fds[i] + j)->revents, &ufds->revents);
if (nleft)
    for (j=0; j < nleft; j++, ufds++)
        __put_user((fds[nchunks] + j)->revents, &ufds->revents);

err = fdcount;
if (!fdcount && signal_pending(current))
    err = -EINTR;

out_fds1:
if (nleft)
    free_page((unsigned long)(fds[nchunks]));
out_fds:
for (i=0; i < nchunks; i++)
    free_page((unsigned long)(fds[i]));
if (nfds != 0)
    kfree(fds);
out:
poll_freewait(&table);
return err;
}

```

코드 49. sys_poll() 함수의 정의(계속)

이전, error 코드의 값을 -EFAULT로 바꾼다. 여기서 부터 발생하는 모든 에러는 -EFAULT로 처리될 것이다. copy_from_user()는 사용자 주소 영역에서 커널 영역으로 데이터를 copy하기 위해서 사용한다. 사용자 영역에 있는 pollfd들은 ufds에 있을 것이며, 이곳에 있는 pollfd들은 page당으로 복사한다. 에러가 있다면, out_fds1으로 제어를 옮길 것이다. Page당 copy에서 남은 것들도 같이 처리한다.

실제적인 poll은 do_poll()에서 처리가 되며, 이렇게 처리된 결과값이 fdcount에 들어간다. 발생한 event를 다시 사용자 영역에 복사하기 위해서 이번에는 __put_user()를 호출한다. err에는 fdcount를 두어 에러값을 표시하게 하고, 만약 fdcount가 0인 값을 가지고, 현재의 프로세스(혹은 thread)가 처리할 시그널이 있다면, err에는 -EINTR을 넣어서 현재 프로세스가 interrupt되었다는 것을 알려준다. 나중에 -EINTR의 에러 코드를 받은 프로세스는 다시 poll하거나 실행을 진행할 것이다. out_fds1에서는 fds[]에 할당된 페이지들을 해제하기 위해서 free_page() 함수를 호출해 줄 것이며, out_fds에서는 nchunks번 만큼 free_page() 함수를 호출한다. 만약 nfds가 0이 아닌 경우에는 앞에서 할당한 fds 구조체를 해제하기 위해서 kfree() 함수를 호출한다. 마지막으로 에러코드를 돌려주기 전에 poll_freewait() 함수를 호출해서 poll_table에 있는 entry들을 제거한다.

```

static int do_poll(unsigned int nfds, unsigned int nchunks, unsigned int nleft,
                  struct pollfd *fds[], poll_table *wait, long timeout)
{
    int count;
    poll_table* pt = wait;

    for (;;) {
        unsigned int i;

```

```

        set_current_state(TASK_INTERRUPTIBLE);
        count = 0;
        for (i=0; i < nchunks; i++)
            do_pollfd(POLLFD_PER_PAGE, fds[i], &pt, &count);
        if (nleft)
            do_pollfd(nleft, fds[nchunks], &pt, &count);
        pt = NULL;
        if (count || !timeout || signal_pending(current))
            break;
        count = wait->error;
        if (count)
            break;
        timeout = schedule_timeout(timeout);
    }
    current->state = TASK_RUNNING;
    return count;
}

```

코드 50. do_poll() 함수의 정의

do_poll() 함수는 실제적인 poll을 처리하는 함수이다. 넘겨받는 인자값은 앞에서 sys_poll()에서 생성한 값들이다. for() loop를 돌면서 주기적으로 do_pollfd()를 호출해서 준비된 파일 디스크립터가 있는지를 확인한다. 먼저 for() loop에서 하는 일은 현재의 task(혹은 process 또는 thread)를 interruptible wake-up 시킬 수 있도록 TASK_INTERRUPTIBLE로 만든다. count는 0을 두어서 초기화 시키고, nchunks만큼 for() loop를 돌면서 do_pollfd() 함수를 호출한다. nleft에 대한 것도 마찬가지로 호출한다. poll_table를 가르키는 pointer인 pt는 앞에서는 wait인자를 가지지만, do_pollfd() 함수를 호출한 후에는 NULL을 가지도록 만든다. 만약 count가 0이 아니거나, timeout이 0인 경우, 혹은 현재 프로세스에 pending된 시그널이 있는 경우에는 외부에 있는 for() loop를 끝낸다. 그렇지 않다면, count에는 poll_table을 가지는 wait의 error값을 두고, count가 0이 아닌 값을 가진는 경우에 다시 for() loop를 빠져나오도록 만든다. 즉, error가 발생했음을 알려주는 것이다. schedule_timeout() 함수는 timeout값 만큼 현재의 프로세스를 대기 시키도록 만든다. for() loop를 마치고 나오면, 앞에서 설정한 task(혹은 process 또는 thread)의 상태를 스케줄링이 되도록 TASK_RUNNING으로 바꾸고, poll_table의 error값을 가지는 count를 돌려준다.

```

static void do_pollfd(unsigned int num, struct pollfd * fdpage,
                      poll_table ** pwait, int *count)
{
    int i;

    for (i = 0; i < num; i++) {
        int fd;
        unsigned int mask;
        struct pollfd *fdp;

        mask = 0;
        fdp = fdpage+i;
        fd = fdp->fd;
        if (fd >= 0) {
            struct file * file = fget(fd);
            mask = POLLNVAL;
            if (file != NULL) {
                mask = DEFAULT_POLLMASK;
                if (file->f_op && file->f_op->poll)
                    mask = file->f_op->poll(file, *pwait);
                mask &= fdp->events | POLLERR | POLLHUP;
                fput(file);
            }
        }
    }
}

```

```

        if (mask) {
            *pwait = NULL;
            (*count)++;
        }
    }
    fdp->revents = mask;
}
}

```

코드 51. do_pollfd() 함수의 정의

do_pollfd() 함수는 poll할 개수를 가지는 num과 page당 pollfd를 포인터인 fdpage, poll_table 구조체의 배열을 가지는 pwait, 그리고 마지막으로 poll되어 준비된 파일 디스크립터를 가지는 count를 넘겨받는다. for() loop를 num번 하도록 하는데, for() loop에서는 num만큼 fdpage를 증가시켜서 각각의 파일 디스크립터를 확인한다. mask는 0으로 초기화 시키고, fdp에는 fdpage를 가르키도록 해준다. 보게될 파일 디스크립터는 fd로 나타낸다. 먼저 fd와 관련된 file 구조체를 가져오기 위해서 fget() 함수를 호출한다. mask에는 기본값으로 POLLNVAL을 준다. file 구조체가 NULL 값을 가지지 않는다면, mask에는 DEFAULT_POLLMASK를 주어서 기본 poll에 대한 mask값을 설정한다. 이전 file 구조체에 있는 file operation 구조체의 poll을 호출하는 것이다. 이때 복귀 값은 mask가 가지게 된다. 여기서 중요한 부분이 바로 file->f_op->poll() 함수이다. 이것은 디바이스 드라이버와의 인터페이스가 되는 부분으로 사용되고 있으며, 만약 디바이스 드라이버와 관련을 가지지 않는 file 구조체의 경우에는 해당 method를 정의해서 poll 할 수 있도록 만들어 준다²². 복귀 값인 mask에 events bit와 POLLERR과 POLLHUP를 OR 시켜서 다시 AND 시켜준다. 즉, POLLERR과 POLLHUP이 있는지를 다시 확인하는 절차이다. 그리고나서, 이전 해당 file 구조체를 놓아주기(release) 위해서 fput() 함수를 호출하도록 한다. 만약 mask에 어떤 값이 있다면, poll_table의 배열에 대한 element에는 NULL을 주고, count값을 증가 시키도록 한다. pollfd의 revents 필드에는 발생한 event를 나타내는 mask를 넣도록 한다. 복귀값으로 사용될 것은 결국 발생한 event의 갯수가 얼마나 되는지를 나타내는 *count값이 될 것이다.

```

void poll_freequeue(poll_table* pt)
{
    struct poll_table_page * p = pt->table;
    while (p) {
        struct poll_table_entry * entry;
        struct poll_table_page *old;

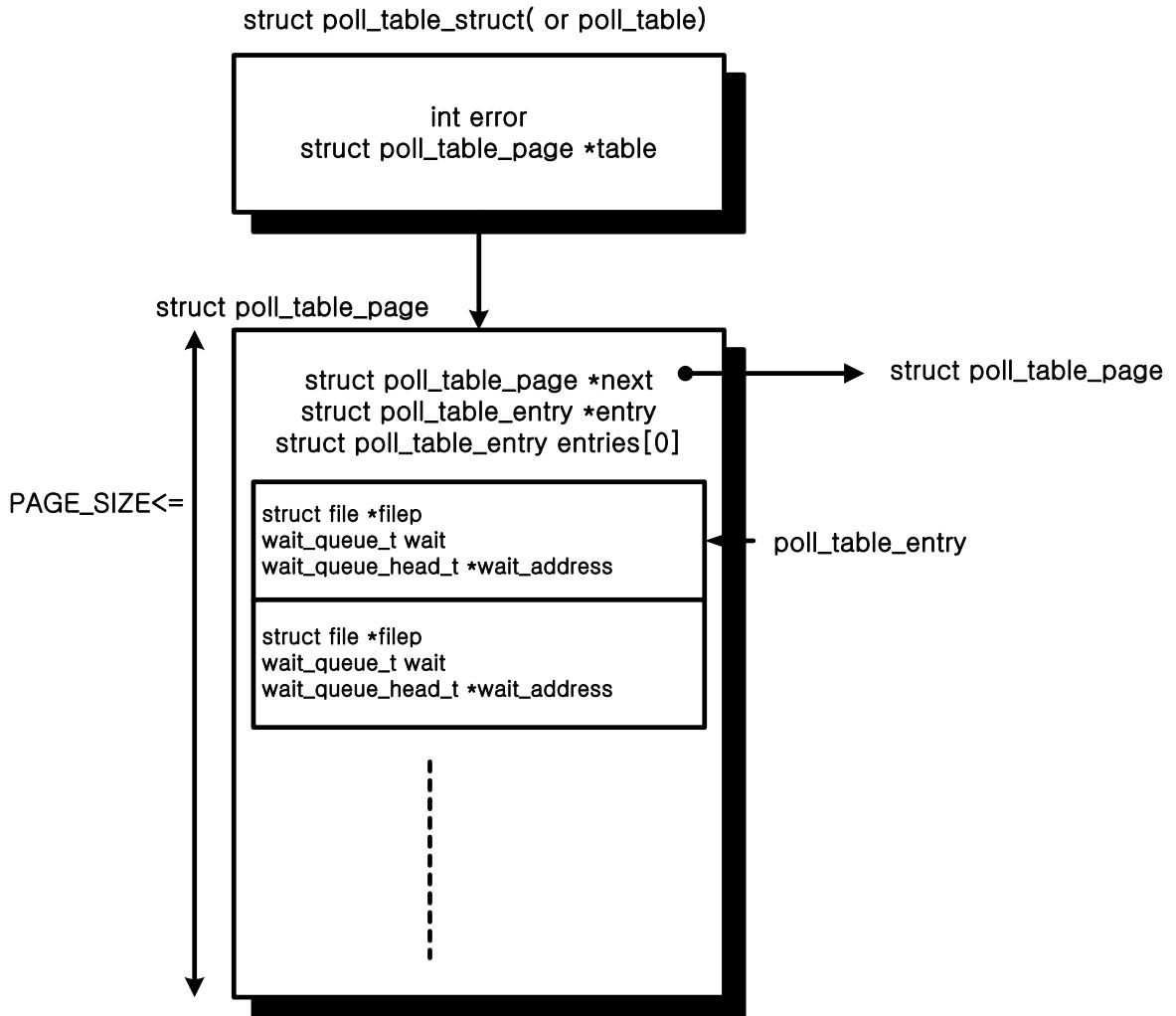
        entry = p->entry;
        do {
            entry--;
            remove_wait_queue(entry->wait_address,&entry->wait);
            fput(entry->filp);
        } while (entry > p->entries);
        old = p;
        p = p->next;
        free_page((unsigned long) old);
    }
}

```

코드 52. poll_freequeue() 함수의 정의

poll_freequeue() 함수는 poll_table에 들어있는 entry들을 제거하는 일을 한다. 즉, 하나의 poll_table_page를 취한 후, 연결 리스트를 후적하면서 wait queue에 들어가 있는 wait_address를 제거하고, 관련된 file 구조체를 해제한다(fput). 한 페이지에 할당된 모든 entry를 제거했다면, free_page() 함수를 호출해서 할당된 페이지를 해제하도록 한다.

²² Socket과 같은 경우도 이에 속한다.

그림 15. `poll_table_struct(or poll_table)` 구조체의 연관 관계

[그림 15]는 앞에서 이야기 한 자료 구조체를 정리한 것이다. 그림에서 보듯이 하나의 `poll_table` 구조체는 페이지단위로 할당되는 `poll_table_page`를 가지는 필드를 가지며, 다시 `poll_table_page` 구조체는 `poll`에 대한 `entry`를 관리하기 위한 필드들과 개개의 `entry(poll_table_entry` 구조체)를 하위의 자료 구조체로 가진다.

2.16.2. Select의 구현

Select도 거의 비슷한 구현 방식을 가진다. 내부적으로는 `poll`과 `select`가 다르지 않다는 것을 알 수 있을 것이다. `Select`의 system call 인터페이스가 되는 것이 `sys_select()` 함수이며, 아래와 같은 정의를 가진다.

```
asmlinkage long sys_select(int n, fd_set *inp, fd_set *outp, fd_set *exp, struct timeval *tvp)
{
    fd_set_bits fds;
    char *bits;
    long timeout;
    int ret, size, max_fdset;

    timeout = MAX_SCHEDULE_TIMEOUT;
    if (tvp) {
        time_t sec, usec;
```

```

if ((ret = verify_area(VERIFY_READ, tvp, sizeof(*tvp)))
    || (ret = __get_user(sec, &tvp->tv_sec))
    || (ret = __get_user(usec, &tvp->tv_usec)))
    goto out_nofds;

ret = -EINVAL;
if (sec < 0 || usec < 0)
    goto out_nofds;

if ((unsigned long) sec < MAX_SELECT_SECONDS) {
    timeout = ROUND_UP(usec, 1000000/HZ);
    timeout += sec * (unsigned long) HZ;
}
}

```

코드 53. sys_select() 함수의 정의

넘겨받는 인자는 최대 파일 디스크립터의 개수를 나타내는 n, in/out/excepiton을 나타내는 fd_set 구조체의 포인터와 timeout을 지정하는 timeval 구조체의 포인터를 가지는 tvp가 있다. 먼저 fd_set을 보면 아래와 같은 정의를 가진다.

```

#define __NFDBITS
#define __NFDBITS      (8 * sizeof(unsigned long))23

#define __FD_SETSIZE
#define __FD_SETSIZE   1024

#define __FDSET_LONGS
#define __FDSET_LONGS  (__FD_SETSIZE/__NFDBITS)
...
typedef struct {
    unsigned long fds_bits [__FDSET_LONGS];
} __kernel_fd_set;

```

코드 54. File Descriptor의 set을 나타내는 fd_set의 정의

fd_set은 실제로는 __kernel_fd_set 구조체로 정의된다. __kernel_fd_set은 __FDSET_LONGS의 크기를 가지는 fds_bit[] 배열로 정해지며, 다시 __FDSET_LONGS는 $1024/(8 \times 4)$ = 32로 정해지는 상수이다. 즉, 한 프로세스가 열수 있는 파일의 개수는 1024개로 한정되어 있으며, 이것을 unsigned long의 크기 만큼이 차지하는 bit수로 나눈 값으로 fds_bits[] 배열의 크기를 정해준 것이다.

timeout값은 기본값으로 MAX_SCHEDULE_TIMEOUT을 준다. 이전 사용자 주소 영역에서 timeval 구조체를 읽기 위해서 verify_area() 함수를 호출해서 유효한 주소인가를 확인한다. __get_user()는 사용자 주소 영역에서 데이터를 가져오는 함수이다. 이렇게 가져온 시간값은 각각 sec와 usec에 second 단위와 micro second 단위로 저장된다. 만약 에러가 있었다면, out_nofds로 제어를 옮긴다. ret에는 다시 -EINVAL을 두어 에러 코드를 초기화 시키고, sec와 usec 값이 0보다 작은 경우에는 out_nofds로 제어를 옮긴다. MAX_SELECT_SECONDS는 MAX_SCHEDULE_TIMEOUT을 초당 tick 단위로 나눈 값(=HZ)에서 -1한 것이다. 만약 sec가 이 값보다 작다면, usec를 올림한 값을 timeout에 더하고, 다시 sec를 HZ로 곱한 값을 더해준다. 이것으로 timeout값은 tick 단위로 변경되었다.

```

ret = -EINVAL;
if (n < 0)
    goto out_nofds;

/* max_fdset can increase, so grab it once to avoid race */

```

²³ 이것은 32bit machine인 경우이다. 다른 architecture에서는 크기가 달라질 수 있다.

```

max_fdset = current->files->max_fdset;
if (n > max_fdset)
    n = max_fdset;
/*
 * We need 6 bitmaps (in/out/ex for both incoming and outgoing),
 * since we used fdset we need to allocate memory in units of
 * long-words.
 */
ret = -ENOMEM;
size = FDS_BYTES(n);
bits = select_bits_alloc(size);
if (!bits)
    goto out_nofds;
fds.in      = (unsigned long *) bits;
fds.out     = (unsigned long *) (bits +   size);
fds.ex      = (unsigned long *) (bits + 2*size);
fds.res_in  = (unsigned long *) (bits + 3*size);
fds.res_out = (unsigned long *) (bits + 4*size);
fds.res_ex  = (unsigned long *) (bits + 5*size);

if ((ret = get_fd_set(n, inp, fds.in)) ||
    (ret = get_fd_set(n, outp, fds.out)) ||
    (ret = get_fd_set(n, exp, fds.ex)))
    goto out;
zero_fd_set(n, fds.res_in);
zero_fd_set(n, fds.res_out);
zero_fd_set(n, fds.res_ex);

```

코드 55. sys_select() 함수의 정의(계속)

ret에는 다시 -EINVAL을 넣어서 에러 코드를 초기화 한다. 만약 보아야 할 파일 디스크립터의 개수(=n)가 0이라면, out_nofds로 제어를 옮긴다. max_fdset은 현재 프로세스의 파일 디스크립터의 set크기를 나타내는 것으로 current->files->max_fdset을 취한다. 만약 이 값보다 큰 n값이 될 경우에는 n을 max_fdset으로 둔다.

ret는 다시 -ENOMEM으로 두고, FDS_BYTES()를 사용해서 n개의 bit에 필요한 byte의 개수를 구한다. select_bits_alloc()은 이렇게 계산된 크기의 byte 할당하는 함수이다. 할당된 메모리는 다시 bits가 가르키도록 만든다. 만약 할당 받지 못한다면, out_nofds로 제어를 옮긴다. fds는 fd_set_bits 구조체로서 아래와 같이 정의되어 있다.

```

typedef struct {
    unsigned long *in, *out, *ex;
    unsigned long *res_in, *res_out, *res_ex;
} fd_set_bits;

```

코드 56. fd_set_bits 구조체의 정의

즉, in/out/ex와 각각에 대한 결과를 저장할 res_in/res_out/res_ex 필드가 unsigned long 값을 가지는 pointer로 정의되어 있다. 따라서, 앞에서 필요에 따라서 할당된 bits 구조체의 적절한 위치를 각각의 필드가 가르키도록 만든다. 이때 두개의 필드가 가르키는 주소값의 차이는 size 만큼이 된다. get_fd_set() 함수는 사용자 영역에서 지정한 fd_set에 대한 포인터로 부터 실제 값을 읽어서 생성된 fds구조체의 in/out/ex 필드를 채우는 역할을 한다. res_in/res_out/res_ex는 0으로 초기화 하기 위해서 zero_fd_set() 함수를 사용한다.

```

ret = do_select(n, &fds, &timeout);
if (tvp && !(current->personality & STICKY_TIMEOUTS)) {
    time_t sec = 0, usec = 0;
}

```

```

if (timeout) {
    sec = timeout / HZ;
    usec = timeout % HZ;
    usec *= (1000000/HZ);
}
put_user(sec, &tvp->tv_sec);
put_user(usec, &tvp->tv_usec);
}
if (ret < 0)
    goto out;
if (!ret) {
    ret = -ERESTARTNOHAND;
    if (signal_pending(current))
        goto out;
    ret = 0;
}
set_fd_set(n, inp, fds.res_in);
set_fd_set(n, outp, fds.res_out);
set_fd_set(n, exp, fds.res_ex);
out:
select_bits_free(bits, size);
out_nofds:
    return ret;
}

```

코드 57. sys_select() 함수의 정의(계속)

실제적인 select에 대한 처리는 do_select() 함수가 한다. 나중에 보기로 하고, 일단 이 함수의 return값을 ret에 주게되며, 함수의 호출시 넘겨지는 인자는 최대 지켜볼 파일 디스크립터의 개수와 앞에서 만든 fds 변수의 주소 및 timeout값의 주소이다. 만약 timeout에 사용되는 tvp가 NULL이 아니고, 현재 프로세스의 personality를 나타내는 값에 STICKY_TIMEOUTS이 없는 경우에는 do_select() 함수에서 변경된 timeout값을 사용자에게 돌려주기 위해서 put_user() 함수를 호출한다. 넘겨지는 값은 각각 sec와 usec가 될 것이다. personality는 프로세스 시스템에 independent하게 동작하기 위해서 필요로 하는 값을 설정하기 위해서 사용하며, select의 timeout과 관련해서는 timeout 값에 대한 변경을 알려줄 것인가 그렇게 하지 않을 것인가를 구분하기 위해서 쓰였다. 즉, 프로그램을 다시 시스템에 porting하기를 원하는 사람이라면, select에서 돌려주는 timeout값에 대해서 주의해서 할 것이다.

만약 ret값이 0보다 작다면, out으로 제어를 옮긴다. 만약 ret가 0을 가진다면, ret는 -ERESTARTNOHAND를 가지게 되며, 현재 프로세스에 pending된 시그널이 있을 경우에는 out으로 진행한다. 그렇지 않다면, ret는 0을 가질 것이다. 이것은 현재 프로세스에 pending된 시그널이 있는 경우에는 -ERESTARTNOHAND라는 에러코드를 가지게 됨을 의미한다. 이전 결과 값을 가지게 되는 fds의 각 필드들(res_in/res_out/res_ex)를 사용자 주소 영역에 있는 각 파일 디스크립터의 set에 설정해 주는 일이 남았다. 이것은 set_fd_set() 함수가 처리한다. select_bits_free()는 앞에서 할당한 fds구조체를 위한 bits에 할당된 메모리를 해제(release)하는 역할을 한다. out으로 제어를 옮길 경우에는 이곳에서 수행될 것이다. out_nofds에서는 단지 ret값에 설정된 error 코드 값만을 돌려준다.

```

static void *select_bits_alloc(int size)
{
    return kmalloc(6 * size, GFP_KERNEL);
}

static void select_bits_free(void *bits, int size)
{
    kfree(bits);
}

```

코드 58. select_bits_alloc() 함수와 select_bits_free() 함수의 정의

`select_bits_alloc()` 함수와 `select_bits_free()` 함수는 크기 만큼 똑같은 필드를 6개를 커널 메모리에서 할당 받고 해제하는 일을 수행한다. 간단히 `kmalloc()` 함수와 `kfree()` 함수를 사용해서 처리하고 있다.

```
int do_select(int n, fd_set_bits *fds, long *timeout)
{
    poll_table table, *wait;
    int retval, i, off;
    long __timeout = *timeout;

    read_lock(&current->files->file_lock);
    retval = max_select_fd(n, fds);
    read_unlock(&current->files->file_lock);

    if (retval < 0)
        return retval;
    n = retval;

    poll_initwait(&table);
    wait = &table;
    if (!__timeout)
        wait = NULL;
    retval = 0;
```

코드 59. `do_select()` 함수의 정의

`do_select()` 함수는 `sys_select()` 함수에서 생성된 `fd_set_bits` 구조체의 포인터인 `fds`와 `timeout` 값의 포인터인 `timeout`, 그리고 최대 파일 디스크립터의 수를 나타내는 `n`을 넘겨 받는다. 변수의 사용에서 알 수 있듯이 실제로는 `poll`과 마찬가지로 `poll_table` 구조체를 이용한다. `__timeout`에는 넘겨받은 `timeout`값을 주도록 한다. 파일에 대한 접근을 해야 하므로, 파일에 대한 `read lock`을 설정하기 위해서 `read_lock()` 함수를 호출해서 프로세스가 `file`에 대한 `read_lock`을 설정했다는 것을 표시한다. 최대로 보아야 하는 파일 디스크립터의 위치 bit는 `max_select_fd()` 함수를 이용해서 구한 후 `retval`에 둔다. `max_select_fd()` 함수는 `unsigned long` 단위로 구분된 `fd_set_bits`에서 최대로 보아야 할 byte의 위치에 몇 번째 bit 인지를 찾는 일을 한다. 즉, 현재 프로세스가 열고 있는 파일 디스크립터들에서 넘겨받은 `fd_set_bits` 구조체의 포인터인 `fds`를 이용해서 최대로 보아야 할 byte의 index를 찾아서 몇 번째 bit를 보아야 할지를 알 수 있다. 이것이 끝나면 프로세스의 `file`에 대한 lock을 해제한다(`read_unlock()`). 만약 `max_select_fd()`가 0보다 작은 값을 돌려준다면, 파일 디스크립터의 설정에 오류가 있었음을 이야기하므로 `retval`을 돌려주고 복귀한다. 그렇지 않다면, `n`은 `retval`을 가지도록 만든다. 이전 `poll`하기 위한 과정으로 `poll_initwait()` 함수를 호출해서 `poll_table` 구조체를 초기화 시킨다. `wait`는 이렇게 초기화된 `poll_table`을 가르키는 포인터이다. 만약 `timeout` 값을 가지는 `__timeout`이 0이라면, `wait`할 필요가 없으므로 `wait`는 `NULL`을 가지도록 만든다. `retval`은 0으로 다시 초기화 한다.

```
for (;;) {
    set_current_state(TASK_INTERRUPTIBLE);
    for (i = 0 ; i < n; i++) {
        unsigned long bit = BIT(i);
        unsigned long mask;
        struct file *file;

        off = i / __NFDBITS;
        if (!(bit & BITS(fds, off)))
            continue;
        file = fget(i);
        mask = POLLNVAL;
        if (file) {
            mask = DEFAULT_POLLMASK;
            if (file->f_op && file->f_op->poll)
```

```

        mask = file->f_op->poll(file, wait);
        fput(file);
    }
    if ((mask & POLLIN_SET) && ISSET(bit, __IN(fds,off))) {
        SET(bit, __RES_IN(fds,off));
        retval++;
        wait = NULL;
    }
    if ((mask & POLLOUT_SET) && ISSET(bit, __OUT(fds,off))) {
        SET(bit, __RES_OUT(fds,off));
        retval++;
        wait = NULL;
    }
    if ((mask & POLLEX_SET) && ISSET(bit, __EX(fds,off))) {
        SET(bit, __RES_EX(fds,off));
        retval++;
        wait = NULL;
    }
}

```

코드 60. do_select() 함수의 정의(계속)

set_current_state()를 호출해서 현재 프로세스의 상태를 TASK_INTERRUPTIBLE로 만든다. 이전 n만큼 for loop를 돌면서 설정된 파일 디스크립터에 변화가 있는지를 확인하는 작업이다. BIT() 매크로는 특정 bit을 set해서 돌려주는 역할을 한다. 따라서, bit이라는 변수는 항상 특정한 bit이 하나가 설정된 값을 나타낸다고 보면 된다. __NFDBITS는 unsigned long의 크기를 가지는 변수가 가질 수 있는 bit수를 나타낸다. 따라서, off는 byte offset을 나타내기 위해서 사용한다. BITS() 매크로는 fd_set_bits 구조체의 in/out/ex에 설정된 bit이 있는지를 검출하는 역할을 하기에 bit과 BITS()를 AND한 결과가 0이 될 경우에는 이 bit이 설정되지 않았다는 말이 되므로, continue로 다음 loop를 돌도록 만든다. 파일 디스크립터의 index값으로 파일 구조체를 얻기 위해서 fget() 함수를 사용하고 있으며, mask값은 POLLNVAL을 두어서 일단은 invalid한 poll flag를 가진다고 설정한다. 만약 가져올 수 있는 파일 구조체가 존재한다면, 이 구조체에 정의된 file operation 구조체를 보고 이 구조체의 poll method를 호출해서 복귀 값을 mask에 두도록 한다. 이것이 끝나면, 해당 파일 구조체를 해제(release)하기 위해서 fput() 함수를 호출한다. 이전 돌려받은 mask값에서 특정 bit이 설정되었는지를 확인하는 일이다. 먼저 보는 것이 POLLIN_SET/POLLOUT_SET/POLLEX_SET이며, 이 값이 mask에 있는 경우에 한해서만 해당 bit이 설정되었는지를 ISSET() 매크로로 확인한다. ISSET() 매크로는 간단히 두개의 인자의 값을 AND시켜서 0인지 확인하는 일을 한다. 위에서 사용된 매크로 들을 정리하면 아래와 같다.

```

#define __IN(fds, n)          (fds->in + n)
#define __OUT(fds, n)         (fds->out + n)
#define __EX(fds, n)          (fds->ex + n)
#define __RES_IN(fds, n)      (fds->res_in + n)
#define __RES_OUT(fds, n)     (fds->res_out + n)
#define __RES_EX(fds, n)      (fds->res_ex + n)

#define BITS(fds, n)          (*__IN(fds, n)|*__OUT(fds, n)|*__EX(fds, n))
...
#define BIT(i)                (1UL << ((i)&(__NFDBITS-1)))
#define MEM(i,m)   ((m)+(unsigned)(i)/__NFDBITS)
#define ISSET(i,m)  (((i)&*(m)) != 0)
#define SET(i,m)   (*(m) |= (i))

#define POLLIN_SET (POLLRDNORM | POLLRDBAND | POLLIN | POLLHUP | POLLERR)
#define POLLOUT_SET (POLLWRBAND | POLLWRNORM | POLLOUT | POLLERR)
#define POLLEX_SET (POLLPRI)

```

코드 61. Select를 위한 매크로 들의 정의

이전 설정된 bit이 있었다면, SET() 매크로를 호출해서 결과값을 가지는 res_in/res_out/res_ex의 bit을 설정하는 일이 될 것이다. retval은 설정된 bit을 counting하기 위해서 증가 시키도록 한다. 이미 설정되었기에 기다릴 필요가 없다고 보고 wait에는 NULL을 두도록 한다.

```

wait = NULL;
if (retval || !__timeout || signal_pending(current))
    break;
if(table.error) {
    retval = table.error;
    break;
}
__timeout = schedule_timeout(__timeout);
}
current->state = TASK_RUNNING;
poll_freewait(&table);
/*
 * Up-to-date the caller timeout.
 */
*timeout = __timeout;
return retval;
}

```

코드 62. do_select() 함수의 정의(계속)

위의 for() loop를 돌고 나오면, 전체 보고 싶은 파일 디스크립터의 bit에 대한 설정 여부를 다 본 것이다. wait에는 NULL을 두도록 한다. 만약 retval이 0이 아닌 값을 가지거나, timeout이 0인 경우, 혹은 현재 프로세스에 대해서 pending된 signal이 있는 경우에는 바깥쪽의 무한 for() loop를 나오기 위해서 break한다. table의 error 필드를 확인하고, 만약 에러가 있었다면 retval에는 이 error 값을 주고, 다시 break한다. 그렇지 않다면 이제 남은 것은 특정 bit이 설정될 때까지 기다리는 일이 될 것이다. schedule_timeout()함수에 __timeout을 넘겨주어 호출한다. 여기서 프로세스는 interruptible wait 상태가 될 것이다. 깨어나게 되는 것은 signal을 받거나, 혹은 특정 bit이 설정되었을 경우에 해당한다. 무한 for() loop를 나온다면, 프로세스의 상태는 TASK_RUNNING으로 다시 복귀될 것이며, poll을 위해서 사용된 poll_table 구조체는 해제한다(poll_freewait()). __timeout값은 원래의 timeout값을 대치하기 위해서 사용하기에 *timeout으로 들어가고 retval을 돌려주고 복귀 한다.

```

static int max_select_fd(unsigned long n, fd_set_bits *fds)
{
    unsigned long *open_fds;
    unsigned long set;
    int max;

    /* handle last in-complete long-word first */
    set = ~(~0UL << (n & (_NFDBITS-1)));
    n /= _NFDBITS;
    open_fds = current->files->open_fds->fds_bits+n;
    max = 0;
    if (set) {
        set &= BITS(fds, n);
        if (set) {
            if (!(set & ~*open_fds))
                goto get_max;
            return -EBADF;
        }
    }
}

```

코드 63. max_select_fd() 함수의 정의

max_select_fd() 함수가 넘겨받는 인자는 최대 bit의 갯수인 n과 fd_set_bits 구조체에 대한 포인터인 fds이다. set은 마지막(최상위의)에 완전하지 않은 unsigned long에 해당하는 bit를 보기위한 것 변수이다. n은 다시 __NFDBITS로 나눈 뒷을 넣어서 몇번째 unsigned long 값인지를 나타낸다. open_fds에는 현재 프로세스가 open한 파일디스크립터의 bit에 앞에서 구한 n값을 더한 값으로 설정한다. max는 일단 0으로 초기화 하도록 한다. 만약 set이 0이 아닌 값을 가진다면, set에 BITS() 매크로를 이용해서 구한 bit을 AND 시키도록 한다. 즉, fds의 in/out/ex에 설정된 bit과 같이 AND해 준다. 만약 이렇게 한 값이 0이 아니라면, open_fds가 가르켜주는 값을 NOT시켜서 set값과 같이 AND시켜준다. 이 값이 0이 아니라면, 최대 값에 해당하는 것을 구한 것이 되므로 get_max로 제어를 옮긴다. 그렇지 않다면, -EBADF를 돌려주어서 잘못된 descriptor임을 알려준다.

```

while (n) {
    open_fds--;
    n--;
    set = BITS(fds, n);
    if (!set)
        continue;
    if (set & ~*open_fds)
        return -EBADF;
    if (max)
        continue;
get_max:
    do {
        max++;
        set >>= 1;
    } while (set);
    max += n * __NFDBITS;
}
return max;
}

```

코드 64. max_select_fd() 함수의 정의(계속)

이젠 n이 0이 아닌 값을 가진동안 반복적으로 수행한다. 먼저 open된 file descriptor bit들을 하나씩 보기 위해서 open_fds와 n을 감소시켜서 나간다. 만약 설정된 bit이 없다면(BITS() = 0), 계속 진행하고(continue), 그렇지 않다면 open_fds와 비교해 본다. 만약 open된 file descriptor값이 아니라면, -EBADF를 돌려준다. max가 어떤 값을 가진다면, 계속 진행해 보도록 한다. get_max에서는 이제 maximum file descriptor값을 구했다는 것이 되므로 이곳에서 수행하도록 한다. 즉, do {} while() loop를 돌면서 max값은 set에 설정된 bit이 있는 동안에 한해서 증가시켜주도록 한다. 또한 이 loop를 돌고 나오면, 이젠 max값에 n x __NFDBITS를 곱해서 unsigned long 크기에 해당하는 bit 크기만큼을 조정하도록 한다. 모든 bit를 전부다 검사했다면, 앞에서 구한 max를 돌려주고 함수는 복귀한다.

덧붙여서 디바이스 드라이버와 같은 곳에서 사용할 수 있는 함수를 한가지 더 보고 넘어가도록 하겠다. 그것이 바로 poll_wait() 함수이다. poll_wait() 함수는 ~/linux/include/linux/poll.h에 아래와 같이 정의된 inline함수이다.

```

static inline void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
{
    if (p && wait_address)
        __pollwait(filp, wait_address, p);
}

```

코드 65. poll_wait() 함수의 정의

poll_wait() 함수는 poll_table 구조체에 wait_queue와 같은 것을 더해주는 역할을 수행하는 함수이다. 가령 예를 들어서 driver함수에서 대기하고 있는 프로세스를 깨운다던가 poll 연산의 상태를 변경하는 경우에 사용할 수 있다. 예를 들어서 driver에서 사용하는 wait queue가 wait_queue라고 할 경우, 디바이스 드라이버의 poll method와 같은 곳에서 다음과 같이 사용될 것이다.

```
unsigned long poll( struct file *filp, poll_table *wait )
{
    device_private_struct *dev = filp->private_data;
    wait_queue_head_t *wait_queue = &dev->wait_queue;
    ...
    poll_wait( filp, &wait_queue, wait );
    ...
    mask |= POLLOUT | POLLIN | POLLRDNORM | POLLWRNORM; /* normal data read/write capable */
    return mask;
}
```

즉, poll() method에서 poll_wait()함수를 호출해서 관련된 file 구조체를 가르키는 filp를 넘겨주고, poll_table 구조체를 가르키는 wait와 함께, 디바이스 드라이버에서 정의해서 사용하는 wait_queue_head_t 구조체의 주소를 넘겨준다. mask값은 현재 쓰기나 읽기를 할 수 있는 일반적인(normal) 데이터가 준비되었다고 알려주기 위한 것이다.

실제적인 구현은 __pollwait() 함수가 맡고 있다. 넘겨받은 인자인 poll_table과 wait_address가 있을 경우에 __pollwait() 함수가 호출된다. 따라서, __pollwait() 함수를 보도록 하자.

```
void __pollwait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
{
    struct poll_table_page *table = p->table;

    if (!table || POLL_TABLE_FULL(table)) {
        struct poll_table_page *new_table;

        new_table = (struct poll_table_page *) __get_free_page(GFP_KERNEL);
        if (!new_table) {
            p->error = -ENOMEM;
            __set_current_state(TASK_RUNNING);
            return;
        }
        new_table->entry = new_table->entries;
        new_table->next = table;
        p->table = new_table;
        table = new_table;
    }

    /* Add a new entry */
    {
        struct poll_table_entry * entry = table->entry;
        table->entry = entry+1;
        get_file(filp);
        entry->filp = filp;
        entry->wait_address = wait_address;
        init_waitqueue_entry(&entry->wait, current);
        add_wait_queue(wait_address,&entry->wait);
    }
}
```

코드 66. __pollwait() 함수의 정의

`__pollwait()` 함수는 먼저 넘겨받은 인자인 `poll_table`을 가르키는 `p`의 `table`부분을 읽어와서 `table`로 둔다. 이 `table`은 `poll_table_page`가 될 것이다. 만약 이 `table`이 `NULL`을 가지거나 혹은 이미 `page` 전체를 다 사용하는 경우(`POLL_TABLE_FULL()`)에는 새로운 `page`를 하나 할당해 주어야 한다(`__get_free_page()`). 새로운 `page`를 할당할 수 없다면, `error code`는 `-ENOMEM`이 될 것이며, 현재 프로세스의 상태를 `TASK_RUNNING`으로 만들어주어, 프로세스가 스케줄링을 받아서 에러가 있었음을 알도록 만들어야 할 것이다. 새롭게 할당된 테이블을 초기화시키고, 이전의 테이블과 연계시켜주어야 한다.

이젠 새로운 poll entry를 추가할 차례이다. `poll_table_page`의 `entry`에 대한 포인터를 얻어서 `entry`로 두고 새로운 `entry`를 가르키기 위해서 `entry`의 포인터를 증가시키도록 한다. `file` 구조체를 사용하기에 `get_file()` 함수를 호출해서 사용 카운터를 증가시켜주고, `entry`의 `file` 구조체에 대한 포인터에 `filp`값을 저장한다. 이젠 `wait queue`에 `wait queue` 구조체를 넣기 위해서 `init_waitqueue_entry()` 함수를 호출해서 `wait queue`를 초기화 시켜주고, `add_wait_queue()` 함수를 호출해서 이 `wait queue`에 넘겨받은 `wait_queue_head_t` 포인터를 넣어주도록 한다.

2.17. ELF 파일의 형식(format)

ELF(Executable and Linking Format)은 binary file²⁴로서, Unix System Laboratory에서 개발되고 발전되어왔다. SVR4와 Solaris 2.X version의 운영체제에서는 기본적인 실행 file의 format으로 사용되고 있다. 실행 file의 format으로는 a.out과 COFF format이 있지만, ELF format이 보다 강력하며, 유연성을 가지고 있다. 적절한 tool과 같이 사용될 때 실행되는 과정을 제어 할 수 있다. 현재 리눅스는 kernel 차원에서 binary file format에 대한 지원을 가지고 있으며, binary file 자체가 가진 특정한 magic number로 실행할 method를 찾게 된다. 이번 장에서는 먼저 ELF file의 format에 대해서 알아보도록 하자²⁵.

ELF는 UNIX System Laboratory에서 Application Binary Interface(ABI)의 일부로서 개발되고 발표되었다. Tool Interface Standards committee(TIS)에서 32 bit Intel Architecture 환경에서 동작하는 portable object 파일 포맷으로 ELF 표준을 선택했다. 프로그램에게 ELF 표준은 여러 운영체제 환경으로 확장될 수 있는 binary 인터페이스 정의들의 집합을 제공한다. 따라서, 프로그램들은 binary 파일의 이러한 인터페이스만을 중심으로 프로그램을 할 수 있는 방법을 제공받을 수 있으며, 더불어 새로이 코드를 재컴파일해서 기록할 필요가 없게된다.

Object 파일에는 크게 3가지가 존재한다. 각각은 아래와 같이 정의될 수 있다.

- 재배치 가능 파일(relocatable file) – 이 파일은 링크를 쉽게 할 수 있는 코드 및 데이터와 함께, 실행 파일을 만들거나 공유 object 파일을 생성하기 위한 다른 object 파일을 가지고 있는 형태의 파일이다.
- 실행 가능 파일(executable file) – 이 파일은 실행을 위해서 적합한 프로그램을 가지는 파일로서, exec 시스템 콜이 어떻게 프로세스의 이미지를 프로그램을 이용해서 만들지를 기술하는 파일이다.
- 공유 object 파일(shared object file)은 두개의 환경(context)상에서 링크에 적합한 형태의 코드와 데이터를 가지는 파일로서, 먼저 LD와 같은 링크 애디터가 이 파일과 함께, 다른 재배치 가능 파일과 공유 object 파일을 처리해서 또 하나의 object 파일을 생성해주게 되고, 이를 다시 동적 링커(dynamic linker)가 생성된 object 파일을 실행 가능 파일과 다른 공유 object 파일을 묶어서 프로세스의 이미지를 생성해 주는 두 단계를 거치는 파일이다.

²⁴ File의 내용이 0과 1의 연속이라는 말이다. 물론 특정한 데이터를 나타내기 보다는 실행할 수 있는 machine instruction을 가지고 있다. 다른 file은 0과 1의 binary data의 연속이지만 대체로 byte stream으로 해석된다. 즉, 뭔가 읽을 수 있는 값을 줄 수 있다는 말이다. Unix에서는 file의 내용에는 관심을 두지 않으며, 어떤 내용이 들어가 있어서 상관하지 않는다. 다만 그것을 해석하는 것은 user application의 몫이다.

²⁵ 이 글에서 참조하는 것은 MIT에서 나온 ELF 파일 포맷 1.1버전에 대한 것이다. 기본적으로 32bit machine에서 사용된다는 것을 가정한다.

어셈블러나 링크 에디터에 의해서 생성된 object 파일은 프로세서에서 실행될 프로그램의 이진(binary) 형식을 가지게 될 것이며, 이곳에서의 논의는 쉘 스크립터와 같은 것은 제외할 것이다.

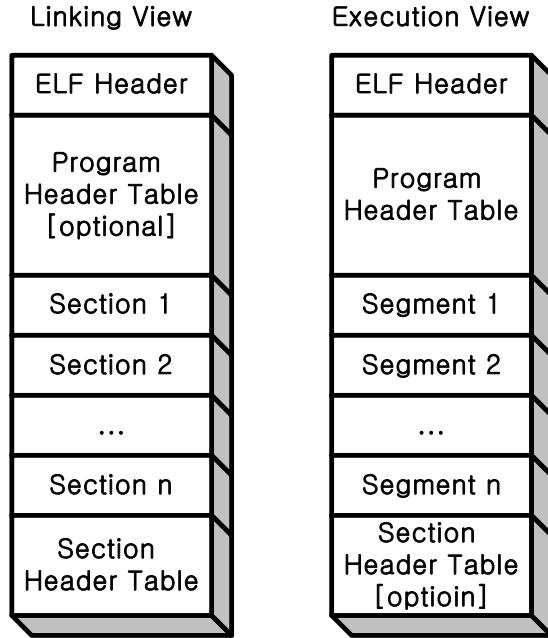


그림 16. ELF 파일의 포맷 - 관점에 따른 형식

먼저 ELF 파일의 형식(format)을 보면 [그림13]과 같다. 즉, object 파일은 크게 두가지의 관점에서 볼 수 있는데, 각각이 프로그램의 링킹(linking)과 프로그램의 실행이라는 측면이 될 것이다.

[그림13]에서 중요한 부분만 훑어보면, **ELF 헤더(header)**는 파일의 구성을 나타내는 로드맵(road map)과 같은 역할을 하며, 첫 부분을 차지한다. **섹션(section)**은 링킹을 위한 object 파일의 정보를 다양으로 가지고 있으며, 이에 해당하는 것으로는 명령(instruction), 데이터(data), 심벌 테이블(symbol table), 재배치 정보(relocation information)등등이 들어간다. **프로그램 헤더 테이블(program header table)**은 옵션(option)이며, 시스템에 어떻게 프로세스 이미지를 만들지를 지시한다. 프로세스의 이미지를 만들기 위해서 사용되는 파일은 반드시 프로그램 헤더 테이블을 가져야하며, 재배치 가능 파일의 경우에는 가지지 않아도 된다. **섹션 헤더(section header)** 테이블은 파일의 섹션들에 대해서 알려주는 정보를 가지는데, 모든 섹션은 이 테이블에 하나의 엔트리(entry)를 가져야 한다. 각각의 엔트리는 섹션 이름이나, 섹션의 크기와 같은 정보를 제공해 준다. 만약 파일이 링킹하는 동안 사용된다면, 반드시 섹션 헤더 테이블을 가져야하며, 다른 object파일은 섹션 헤더 테이블을 가지고 있지 않을 수도 있다. 이에 해당하는 정보들은 리눅스의 경우 `~/include/linux/elf.h`에서 찾을 수 있다.

코드를 직접 보기 전에 이하의 내용에서 사용하게 될 데이터 타입에 대한 정의부터 하기로 한다. 데이터 타입이 필요한 것은 ELF format이 다양한 프로세서에서 사용할 수 있으며, 이들 각각이 사용하는 기본 데이터의 길이가 달라질 가능성이 있기 때문이다. 따라서, object 파일은 기계 의존적인 특성들에 대비를 해야하는데, 기계에 의존적이지 않은 형식을 지원하기 위한 제어 데이터(control data)가 이 역할을 해주고 있다. 즉, object 파일을 인식하고, 파일의 내용을 일반적인 방법으로 해석하도록 돋는다. 리눅스에서는 아래와 같은 것을 정의해서 사용하고 있다.

```
/* 32-bit ELF base types. */
typedef __u32 Elf32_Addr;
typedef __u16 Elf32_Half;
typedef __u32 Elf32_Off;
typedef __s32 Elf32_Sword;
typedef __u32 Elf32_Word;
```

```
/* 64-bit ELF base types. */
typedef __u64 Elf64_Addr;
typedef __u16 Elf64_Half;
typedef __s16 Elf64_SHalf;
typedef __u64 Elf64_Off;
typedef __s64 Elf64_Sword;
typedef __u64 Elf64_Word;
```

코드 67. ELF 파일 형식에 사용할 데이터 타입 정의

두가지로 나누어 볼 수 있는데, 먼저 32bit 기계에 대한 것과 64bit 기계에 대한 부분이다. 각각이 기본 연산의 단위로 32bit과 64bit를 사용함을 알 수 있다. 나중에 여기서 정의된 것을 이용해서 다른 데이터 타입에 대한 정의를 하게 될 것이다. 만약 이러한 데이터 타입을 제공하게 될 때, 기본 연산의 단위로 정렬(alignment)하고 싶다면, 패딩(padding)을 삽입해서 4bytes나 혹은 8bytes 단위로 정렬할 수 있을 것이다.

2.17.1. ELF 헤더의 정의

이전 ELF 헤더부터 보기로 하자. 아래와 같이 정의된다. 32bit과 64bit 기계의 ELF 헤더 정의이다.

```
#define EI_NIDENT      16

typedef struct elf32_hdr{
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half      e_type;
    Elf32_Half      e_machine;
    Elf32_Word      e_version;
    Elf32_Addr     e_entry; /* Entry point */
    Elf32_Off       e_phoff;
    Elf32_Off       e_shoff;
    Elf32_Word      e_flags;
    Elf32_Half      e_ehsize;
    Elf32_Half      e_phentsize;
    Elf32_Half      e_phnum;
    Elf32_Half      e_shentsize;
    Elf32_Half      e_shnum;
    Elf32_Half      e_shstrndx;
} Elf32_Ehdr;

typedef struct elf64_hdr {
    unsigned char e_ident[16];           /* ELF "magic number" */
    Elf64_SHalf    e_type;
    Elf64_Half     e_machine;
    __s32          e_version;
    Elf64_Addr    e_entry;             /* Entry point virtual address */
    Elf64_Off      e_phoff;            /* Program header table file offset */
    Elf64_Off      e_shoff;            /* Section header table file offset */
    __s32          e_flags;
    Elf64_SHalf    e_ehsize;
    Elf64_SHalf    e_phentsize;
    Elf64_SHalf    e_phnum;
    Elf64_SHalf    e_shentsize;
```

```

Elf64_SHalf      e_shnum;
Elf64_SHalf      e_shstrndx;
} Elf64_Ehdr;

```

코드 68. 32bit 및 64bit ELF header

각각의 필드가 하는 역할은 아래와 같다. 주석으로 처리하기에는 조금 무리가 있을 듯 싶어서, 다시 정리했다.

Field	Description
e_ident[]	파일의 내용을 해석하고 디코딩하기 위해서 사용되는 기계 독립적인 데이터를 제공하며, 파일이 object파일임을 나타낸다.
e_type	Object 파일의 타입을 표시한다.
e_machine	이 파일을 사용하기 위해서 필요한 architecture정보를 나타낸다.
e_version	Object 파일의 버전 정보를 나타낸다.
e_entry	시스템이 실행하기 위해서 제어를 옮길때 어디로 옮겨야 하는지를 가르쳐주는 가상주소를 가진다. 만약 파일이 진입점(entry point)을 가지지 않는다면 0을 가질 것이다.
e_phoff	프로그램 헤더 테이블의 파일 옵셋을 byte단위로 나타낸다. 프로그램 헤더 테이블이 없다면 당연히 0을 가질 것이다.
e_shoff	섹션 헤더 테이블의 파일 옵셋을 byte단위로 나타낸다. 역시 섹션 헤더 테이블을 가지지 않는다면 0을 가질 것이다.
e_flags	파일과 관련되서 프로세서에 특수한(specific) 플랙을 가진다. 플랙을 나타내는 형태는 EF_[machine_flag]이 될 것이다. 이것을 알기위해서 기계 의존적인 부분을 보아야 할 것이다.
e_ehsize	ELF 헤더의 크기를 가진다.
e_phentsize	파일의 프로그램헤더 테이블에 있는 한 엔트리의 크기를 byte단위로 표시한다. 모든 엔트리는 동일한 크기를 가진다.
e_phnum	프로그램 헤더 테이블에 들어있는 모든 엔트리의 수를 나타낸다. 따라서, 전체 프로그램 헤더 테이블의 크기는 앞에 나온 e_phentsize와 e_phnum의 곱이 될 것이다. 마찬가지로 프로그램 헤더 테이블이 없다면 0을 가진다.
e_shentsize	섹션 헤더의 크기를 byte단위로 나타낸다. 섹션 헤더는 섹션 헤더 테이블의 하나의 엔트리를 차지하며, 모두 크기가 동일하다.
e_shnum	섹션 헤더 테이블에 있는 엔트리의 수를 가진다. 따라서, e_shentsize와 e_shnum의 곱으로 섹션 헤더 테이블의 전체 크기를 byte단위로 알 수 있다. 만약 섹션 헤더 테이블이 없다면, 당연히 0을 가질 것이다.
e_shstrndx	섹션의 이름을 나타내는 스트링(string)의 테이블과 관련된 엔트리의 섹션 헤더 테이블 인덱스(index)를 가지며, 만약 섹션 이름을 나타내는 테이블이 없다면 SHN_UNDEF 값을 가질 것이다.

표 9. ELF헤더의 필드 정의

e_type에 들어갈 수 있는 값으로는 다시 아래와 같은 것이 있다.

```

#define ET_NONE 0 /* 파일의 타입이 없음 */
#define ET_REL 1 /* 재배치 가능 파일 */
#define ET_EXEC 2 /* 실행 파일 */
#define ET_DYN 3 /* 공유 object 파일 */
#define ET_CORE 4 /* core 파일 */
#define ET_LOPROC 0xff00 /* 프로세서에 의존적인 파일 */
#define ET_HIPROC 0xffff /* 프로세서에 의존적인 파일 */

```

또한 e_machine에 들어갈 수 있는 값으로는 다시 아래와 같이 정의 되어 있다.

```
#define EM_NONE 0 /* 특정 machine을 구분하지 않음 */
#define EM_M32 1 /* AT&T WE32100 */
#define EM_SPARC 2 /* SPARC */
#define EM_386 3 /* Intel 80386 */
#define EM_68K 4 /* Motorola 68000 */
#define EM_88K 5 /* Motorola 88000 */
#define EM_486 6 /* 사용되지 않음 */
#define EM_860 7 /* Intel 80860 */
#define EM_MIPS /* MIPS R3000 (officially, big-endian only) */
#define EM_MIPS_RS4_BE 10 /* MIPS R4000 big-endian */
#define EM_PARISC 15 /* HPPA */
#define EM_SPARC32PLUS 18/* Sun's "v8plus" */
#define EM_PPC 20 /* PowerPC */
#define EM_SH 42 /* SuperH */
#define EM_SPARCV9 43 /* SPARC v9 64-bit */
#define EM_IA_64 50 /* HP/Intel IA-64 */
#define EM_X8664 62 /* AMD x86-64 */
#define EM_ALPHA 0x9026 /* Alpha */
#define EM_S390 0xA390 /* IBM S390 */
```

이곳에서 언급되지 않은 값은 새로운 기계에 필요에 따라서 할당될 수 있으며, 프로세서에 특정한 ELF의 이름은 각각을 구분하기 위해서 사용된다. 예를 들어서 e_flags에서와 같이 사용할 수 있을 것이다.

마지막으로 e_version으로 들어갈 수 있는 값은 아래와 같다.

```
#define EV_NONE 0 /* Invalid 버전 */
#define EV_CURRENT 1 /* 현재 버전 */
#define EV_NUM 2 /* 다음에 주어지는 버전 */
```

EV_CURRENT는 새로운 버전이 발표되면 현재 버전 번호를 반영하기 위해서 바뀌게 될 것이다.

2.17.2. ELF 포맷의 검출

앞에서는 ELF 헤더의 형식에 대해서 보았다. 이전 실제적으로 커널에서 해당하는 인터프리터를 가져오기 위해서 ELF 포맷을 검출할 수 있도록 하는 부분을 보도록 하자. ELF는 여러 프로세스들과, 여러 데이터의 인코딩, 그리고, 여러 계층의 기계들을 지원하기 위한 object 파일의 기본 윤곽(framework)을 제공한다. 따라서, 프로세서에 의존적이지 않고, 파일의 나머지 부분들에 대해 독립적인, 파일을 해석하는 방법을 기술하는 bytes들을 파일의 시작 부분에 가져야 한다. ELF 헤더의 e_ident가 이에 해당한다. e_ident[]는 아래와 같이 정의된다.

이름	값	설명
EI_MAG0	0	File identification
EI_MAG1	1	File identification
EI_MAG2	2	File identification
EI_MAG3	3	File identification
EI_CLASS	4	파일 클래스
EI_DATA	5	데이터의 인코딩 방식
EI_VERSION	6	파일의 버전
EI_PAD	7	패딩 바이트의 시작

표 10. e_ident[]의 필드

각각의 필드에 대한 설명을 덧붙이자면, 먼저 EI_MAG0에서 EI_MAG3까지는 ELF파일을 가르키는 매직 번호가 들어온다. 리눅스에서는 아래와 같이 정의하고 있다.

```
#define ELF_MAGIC0      0x7F      /* EI_MAG */
#define ELF_MAGIC1      'E'
#define ELF_MAGIC2      'L'
#define ELF_MAGIC3      'F'
```

EI_CLASS는 파일의 클래스나 용량(capacity)를 나타내며, 아래와 같은 것이 올 수 있다.

```
#define ELFCLASSNONE    0      /* Invalid class */
#define ELFCLASS32       1      /* 32 bit objects */
#define ELFCLASS64       2      /* 64 bit objects */
#define ELFCLASSNUM 3      /* ELF class number */
```

ELF 파일 포맷은 다양한 크기를 가지는 기계들에 이식성이 있도록 디자인 되었다. 즉, ELFCLASS32는 4Gbytes의 가상 메모리 공간을 사용하는 32 bit 기계에 적합하며, ELFCLASS64는 64 bit 기계를 위해서 예약된 것이다.

EI_DATA에 올 수 있는 것은 다시 아래와 같은 값들을 가질 수 있다.

```
#define ELFDATANONE 0      /* Invalid data encoding */
#define ELFDATA2LSB 1      /* LSB26가 가장 낮은 주소를 차지 한다. */
#define ELFDATA2MSB 2      /* MSG가 가장 낮은 주소를 차지 한다. */
```

파일의 데이터를 인코딩 하는 방법은 파일내에 있는 가장 기본적인 object²⁷를 해석하는 방법을 표시한다. ELFCLASS32의 경우를 예로 들면 기본 object는 1, 2, 혹은 4 bytes를 차지할 수 있을 것이다. 이것은 인코딩 방식에 따라서 아래의 [그림14]와 같이 나타낼 수 있다.

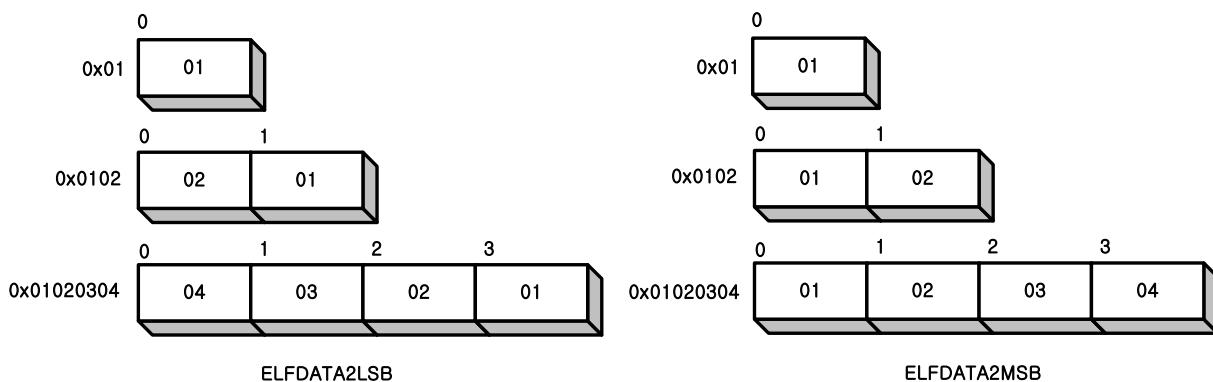


그림 17. 데이터 encoding

EI_VERSION에 들어가게 되는 값은 현재로는 EV_CURRENT이다. 또한 EI_PAD는 예약된 byte로 0으로 설정된다. 따라서, object파일을 읽어들인 프로그램은 이것을 무시할 수 있다. 만약 새로운 것이 추가된다면 EI_PAD는 바뀌게 될 것이다.

Intel의 32bit architecture를 지원하는 e_ident[]를 만들려면 EI_CLASS에는 ELFCLASS32를 EI_DATA에는 ELFDATA2LSB를 넣어야 할 것이다. 또한 이것 이외에 ELF 헤더의 e_machine에는 EM_386이 들어와야

²⁶ LSB(Least Significant Byte), MSB(Most Significant Byte).

²⁷ 여기서 말하는 object는 byte단위를 말한다. 즉, 숫자로서 표기될 값이 어떤 주소에 byte단위로 저장될지를 나타낸다고 보면 될 것이다.

할 것이며, ELF 헤더의 `e_flags`는 파일과 관련된 프로세서의 특수한 flag의 역할을 하지만, Intel 32 bit architecture에서는 이것을 사용하지 않기에 0을 가지게 될 것이다.

리눅스에서는 실행 가능한 binary 포맷에 대해서 `linux_binfmt` 구조체를 정의하고 있으며, 전체 binary 포맷에 대한 포인터는 `formats`가 가지고 있다. 정의는 `~/fs/exec.c`에 있으며, 아래와 같다.

```
/* ~/include/linux/binfmts.h에서 가져옴 */
struct linux_binfmt {
    struct linux_binfmt * next;           /* 다음 binary format에 대한 포인터 */
    struct module *module;                /* 이 binary format이 필요로 하는 모듈 */
    int (*load_binary)(struct linux_binprm *, struct pt_regs * regs); /* binary format의 로더 */
    int (*load_shlib)(struct file *);      /* 공유 라이브러리의 로더 */
    int (*core_dump)(long signr, struct pt_regs * regs, struct file * file); /* core dump파일 생성
기 */
    unsigned long min_coredump;           /* 최소 코아 덤프 크기 */
};

/* ~/fs/exec.c에서 가져옴 */
static struct linux_binfmt *formats;
```

코드 69. `linux_binfmt`의 정의

`formats`에 binary 포맷을 등록하는 것은, 해당 모듈이 초기화 될 때 `register_binfmt()` 함수를 `linux_fmt` 구조체를 넘겨주어 호출하는 것이다. ELF binary format에 대한 등록은 `~/fs/binfmt.c`에 있다. 등록을 해제하는 것은 `unregister_binfmt()` 함수가 한다. 등록 시에 넘겨주는 인자의 값으로는 `elf_format`이며, 아래와 같이 정의된다.

```
static struct linux_binfmt elf_format = {
    NULL, THIS_MODULE,
    load_elf_binary,
    load_elf_library,
    elf_core_dump,
    ELF_EXEC_PAGESIZE /* 4096 */
};
```

코드 70. ELF 포맷에 대한 `linux_binfmt`의 정의

리눅스에서는 프로그램을 실행할 때, 프로그램이 어떤 포맷으로 되어 있는지를 확인하게 되며, 실행하도록 도와주는 로더(loader) 프로그램을 선택해 준다. `~/fs/exec.c`에서 `do_execve()`²⁸를 호출하면, 이 프로그램에서 `search_binary_handler()` 함수를 호출한다. 이 함수는 binary 포맷에 맵는 등록된 핸들러를 찾는 역할을 하며, 이 함수는 다시 등록된 포맷의 `load_binary()` 함수를 차례로 호출한다. 따라서, ELF 파일에 대한 핸들러를 등록했다면, `load_elf_binary()` 함수가 호출될 것이다.

`load_elf_binary()` 함수가 넘겨받는 것은 `linux_binprm` 구조체 타입인 `bprm`과 레지스터들에 대한 포인터이다. `linux_binprm` 구조체의 버퍼 필드에는 이미 읽은 실행 파일의 시작부분에 있는 128bytes를 가지고 있으므로 이곳에서 ELF MAG 값을 가지고 있는지를 확인하게 된다. 만약 ELF 파일 포맷이 아니라면, 다음에 등록된 binary handler로 이동한다. 또한 `load_elf_binary()` 함수에서는 현재 프로세스(`current`)의 `task_struct` 필드에서 mm에 있는 부분들에 대한 초기화 및 실행에 필요한 `interpreter` 프로그램의 적재와 프로그램의 실행을 위한 메모리 할당이 있게 된다.

2.17.3. Section

파일 내에 있는 모든 섹션은 파일의 섹션 헤더 테이블을 통해서 접근할 수 있다. 몇몇 섹션 헤더 테이블의 인덱스는 이미 특수 목적으로 정의되어서 사용되고 있는데, 아래와 같은 것들이 있다.

²⁸ 실제적으로 `sys_execve()` 시스템 콜에 대한 처리를 맡고 있다.

```
/* special section indexes */
#define SHN_UNDEF          0
#define SHN_LORESERVE      0xff00
#define SHN_LOPROC          0xff00
#define SHN_HIPROC          0xff1f
#define SHN_ABS             0xffff1
#define SHN_COMMON          0xffff2
#define SHN_HIRESERVE      0xfffff
/* 위의 부분은 ELF 1.1에서 정의되어 있는 부분이며, 아래의 것은 LINUX에서 확장된 부분이다. */
#define SHN_MIPS_ACCOMON 0xff00
```

코드 71. 특수 목적의 섹션 인덱스 정의

이들 각각에 대한 의미를 살펴보면, 아래와 같은 표로 정리 된다.

이름	설명
SHN_UNDEF	의미없거나, 정의되지 않은 경우, 또는 잃어버렸거나 관련이 없는 섹션에 대한 언급을 표시한다. 예를 들어서 ‘defined’라는 것이 섹션 번호 SHN_UNDEF에 관련되어 있다면 정의되지 않은 심벌이다.
SHN_LORESERVE	예약된 인덱스의 하한선(lower bound)를 나타낸다.
SHN_LOPROC	SHN_LOPROC에서 SHN_HIPROC 사이에 들어있는 영역(range)에 대해서는 프로세서 의존적인 의미를 가지도록 예약되었다.
SHN_HIPROC	위와 같은 의미이다.
SHN_ABS	해당하는 언급(reference)에 대해서 절대값을 표시한다. 예를 들어서 SHN_ABS 섹션 번호에 관련되어 정의된 심벌들은 절대 값을 가지며, 재배치에 영향을 받지 않는다.
SHN_COMMON	이 섹션에 관련되어 정의된 심벌들은 일반(common) 심벌이다. 예를 들어 FORTRAN COMMON이나 할당되지 않는 C의 external 변수들이 있다.
SHN_HIRESERVE	이 값은 예약된 인덱스의 영역에 대한 상한선(upper bound)를 나타낸다. 시스템에서는 SHN_LORESERVE와 SHN_HIRESERVE 사이의 인덱스들을 예약하고 있으며, 이 값들은 섹션 헤더 테이블을 언급하지 않는다. 즉, 섹션 헤더 테이블은 예약된 인덱스들에 대해서 엔트리를 가지지 않는다.

표 11. 특수 목적의 섹션 인덱스

섹션들은 ELF header와 프로그램 헤더, 그리고, 섹션 헤더 테이블을 제외한 모든 object 파일에 있는 정보들을 포함하고 있으며, 다음과 같은 여러 조건들을 맞춘다.

- Object 파일내의 모든 섹션은 그것을 기술하는 정확히 하나의 섹션 헤더를 가진다. 하지만, 섹션 헤더는 섹션을 가지지 않을 수도 있다.
- 각각의 섹션은 파일내에서 하나의 연속된(혹은 빈(empty)) byte들을 점유한다.
- 파일내의 섹션은 겹쳐질 수 없으며, 파일내의 어떠한 byte도 둘 이상의 섹션에 존재하지 못한다.
- Object 파일은 비활성(inactive)인 공간을 가질 수 있으며, 다양한 헤더나 섹션들도 object파일의 모든 byte를 다 커버(cover)하지 않을 수 있다. 비활성인 데이터의 내용은 명시되지 않는다.

섹션 헤더 테이블은 Elf32_Shdr구조체의 배열로 정의되며, 아래와 같다.

```
typedef struct {
    Elf32_Word    sh_name;
    Elf32_Word    sh_type;
    Elf32_Word    sh_flags;
```

```

Elf32_Addr    sh_addr;
Elf32_Off     sh_offset;
Elf32_Word    sh_size;
Elf32_Word    sh_link;
Elf32_Word    sh_info;
Elf32_Word    sh_addralign;
Elf32_Word    sh_entsize;
} Elf32_Shdr;

```

코드 72. 섹션 헤더 구조체의 정의

앞에서 본 ELF 헤더의 `e_shoff`은 섹션 헤더 테이블의 움셋을 파일의 처음에서 부터 계산한 값이다. 또한 `e_shnum`은 섹션 헤더 테이블에 얼마나 많은 수의 엔트리가 있는지를 말해준다. 섹션 헤더 테이블에 있는 한 엔트리의 크기는 `e_shentsize`로 알 수 있다.

각각의 필드가 가지는 의미는 아래와 같다.

이름	의미
<code>sh_name</code>	섹션의 이름을 가진다. 이 값을 섹션 헤더 스트링 테이블 섹션에 대한 인덱스로 사용할 수 있으며, NULL로 끝나는 스트링의 위치를 알려준다.
<code>sh_type</code>	섹션의 내용들이나 혹은 의미(semantics)를 구분짓는다(categorize).
<code>sh_flags</code>	여러가지 특성을 나타내는 플랙으로 사용된다.
<code>sh_addr</code>	만약 섹션이 프로세스의 메모리 이미지로서 나타나게 된다면, 섹션의 첫번째 byte가 위치해야하는 주소를 알려준다. 그렇지 않다면 0을 가진다.
<code>sh_offset</code>	섹션의 첫 byte주소를 파일의 처음에서 어디에 위치해 있는지를 알려준다. SHT_NOBIT와 같은 타입을 가지는 섹션은 파일에서 자리를 차지 하지 않으므로 <code>sh_offset</code> 은 단지 파일에서 개념적인 위치만을 알려준다.
<code>sh_size</code>	섹션의 크기를 byte단위로 알려준다. SHT_NOBITS가 아닌 섹션 타입에 대해서 <code>sh_size</code> 만큼을 파일에서 차지한다. SH_NOBITS에 대해서 0이 아닌 값을 가질 수 있으나, 실제적으로는 파일에서 아무런 자리를 차지 하지 않는다.
<code>sh_link</code>	섹션 헤더 테이블 인덱스의 링크를 가지는 부분으로 이 필드에 대한 해석은 섹션 헤더 타입에 따른다.
<code>sh_info</code>	부가적인 정보를 가지는 부분으로 역시 섹션 타입에 의존적이다.
<code>sh_addralign</code>	어떤 섹션에 대해서는 주소가 정렬(alignment)되어야 하는데, 예를 들어서 섹션이 doubleword를 가진다면 전체 섹션에 대해서 doubleword로 시스템이 정렬해 주어야 한다. 즉, <code>sh_addralign</code> 값의 modulo값인 0에 <code>sh_addr</code> 이 맞춰져야 할 것이다. 현재 0이나 2의 제곱승 ²⁹ 에 대해서만 이 값을 나타낼 수 있으며, 0이나 1은 섹션이 정렬될 필요가 없음을 의미한다.
<code>sh_entsize</code>	싱글 테이블과 같은 섹션들은 고정된 크기의 엔트리만을 가진다. 그러한 섹션들에 대해서 <code>sh_entsize</code> 는 각각의 엔트리의 크기를 byte단위로 표시한다. 만약 고정된 크기의 엔트리를 가지지 않는다면 0 값을 가진다.

표 12. 섹션 헤더의 필드 정의

`sh_type`은 섹션의 의미(semantics)를 나타내면 아래와 같은 값을 가질 수 있다.

```

/* sh_type */
#define SHT_NULL          0
#define SHT_PROGBITS      1
#define SHT_SYMTAB        2

```

²⁹ 2, 4, 8, 16...

```

#define SHT_STRTAB      3
#define SHT_REL          4
#define SHT_HASH         5
#define SHT_DYNAMIC      6
#define SHT_NOTE          7
#define SHT_NOBITS        8
#define SHT_REL           9
#define SHT_SHLIB        10
#define SHT_DYNSYM       11
#define SHT_NUM          12
#define SHT_LOPROC      0x70000000
#define SHT_HIPROC      0x7fffffff
#define SHT_LOUSER      0x80000000
#define SHT_HIUSER      0xffffffff
/* 이상의 부분은 ELF 1.1에서 정의한 타입이며, 이하의 것은 확장된 것이다. */
#define SHT_MIPS_LIST 0x70000000
#define SHT_MIPS_CONFLICT 0x70000002
#define SHT_MIPS_GPTAB 0x70000003
#define SHT_MIPS_UCODE 0x70000004

```

코드 73. 섹션 타입의 정의

각각이 의미하는 바는 아래와 같다.

이름	설명
SHT_NULL	섹션 헤더가 활성이 아니다(inactive). 즉, 관련된 섹션을 가지고 있지 않음을 알려준다. 다른 섹션 헤더의 멤버는 정의되지 않은 값을 가진다.
SHT_PROGBITS	프로그램에서 정의한 정보를 가지는 부분이다. 형식과 의미 역시 프로그램이 어떻게 하느냐에 따른다.
SHT_SYMTAB	심벌의 테이블을 가지는 섹션이다. 현재 object파일은 각각의 타입에 대해서 단지 하나의 섹션을 가질 수 있지만 나중에 가서는 이것이 풀릴 것이다. SHT_SYMTAB는 링크 에디터를 위해서 심벌들을 제공하고 있지만, 동적인 링킹에도 사용될 수 있다. 완전한 심벌 테이블은 동적 링킹을 위해서 필요치 않은 많은 심벌들을 가질 수 있다.
SHT_STRTAB	스트링 테이블을 가진다. Object파일은 여러개의 스트링 심벌 테이블을 가질 수 있다.
SHT_REL	섹션이 명시적(explicit)인 가수(addend)를 가지는 재배치 엔트리를 가지고 있다. Object 파일은 여러개의 재배치 섹션을 가질 수 있다.
SHT_HASH	섹션이 심벌 해쉬 테이블을 가진다. 동적인 링킹에 참여하는 모든 object들은 심벌 해쉬 테이블을 가져야 하며, 현재 object 파일은 단지 하나의 해쉬 테이블을 가지지만, 나중에는 이러한 제약이 풀릴 것이다.
SHT_DYNAMIC	섹션이 동적인 링킹을 위한 정보를 가진다. 현재 object파일은 단지 하나의 동적인 섹션을 가질 수 있지만, 이 역시 제약사항이 풀릴 것이다.
SHT_NOTE	파일에 대한 표식(note)을 가지는 섹션을 말한다.
SHT_NOBITS	아무런 공간을 차지 하지 않는 섹션이다. 아무런 공간도 가지지 않는 섹션이지만 sh_offset에는 파일의 개념적인 옵셋을 가질 수 있다.
SHT_REL	명시적인 가수(addend)가 없는 재배치 엔트리에 대한 정보를 가지는 섹션이다. Object 파일은 여러개의 재배치 섹션을 가질 수 있다.
SHT_SHLIB	명확한 의미를 가지지는 않는 섹션 타입으로 프로그램은 이 타입에 대해서 ABI가 정하는 것을 따르지 않아도 상관없다.
SHT_DYNSYM	Object 파일은 SHT_SYMTAB이외에 공간(space)을 절약하기 위해서, 동적인

	링킹 심벌들의 최소 집합을 가지는 SHT_DYNSYM을 가질 수 있다.
SHT_LOPROC	프로세서에 의존적인 의미를 가진다.
SHT_HIPROC	역시 프로세서에 의존적인 정보를 가진다.
SHT_LOUSER	응용 프로그램을 위해서 예약된 인덱스의 하위 한도(lower bound)를 명시한다.
SHT_HIUSER	응용 프로그램을 위해서 예약된 인덱스의 상위 한도(upper bound)를 명시하는 부분으로 SHT_LOUSER와 SHT_HIUSER의 사이에 있는 섹션 타입들은 응용 프로그램에서, 현재나 미래의 시스템에서 정의한 섹션 타입들과 상충없이 사용될 수 있다.

표 13. 섹션 타입

나머지 섹션 타입에 대해서는 현재로서는 정확한 의미를 찾을 수 없으며, 다음 그 이름이 의미하는 바로 추측해 볼 수는 있을 것이다. 즉, MIPS CPU에 해당하는 사항이다.

현재로선 다른 섹션 타입은 예약되어 있다. 앞에서 말했듯이, 인덱스가 정의 되지 않은 섹션 레퍼런스(reference)를 표시하더라고, 인덱스 0(SHN_UNDEF)에 해당하는 섹션 헤더는 존재한다. 아래와 같은 것을 그 엔트리가 가질 수 있다. 즉, 이것은 섹션 헤더 테이블의 엔트리로 인덱스 0에 해당한다.

이름	값	설명
sh_name	0	아무런 이름이 없음
sh_type	SHT_NULL	비활성임
sh_flags	0	아무런 flag이 없음
sh_addr	0	아무런 주소가 없음
sh_offset	0	파일 옵셋도 가지지 않음
sh_size	0	섹션의 크기도 없음
sh_link	SHN_UNDEF	링크 정보도 가지지 않음
sh_info	0	보조적인 정보도 가지지 않음
sh_addralign	0	정렬할 필요없음
sh_entsize	0	해당 엔트리의 크기도 없음

표 14. 섹션 헤더 테이블 엔트리 0에 대한 설정예

섹션 헤더의 sh_flags로 올 수 있는 것은 섹션의 특성들(attributes)을 나타내는 값으로 아래와 같은 정의된 값이 사용된다.

```
/* sh_flags */
#define SHF_WRITE          0x1
#define SHF_ALLOC           0x2
#define SHF_EXECINSTR       0x4
#define SHF_MASKPROC        0xf0000000
/* 위의 것은 ELF 1.1에서 정의된 것이며, 아래는 LINUX에서 추가된 것임 */
#define SHF_MIPS_GPREL      0x10000000
```

코드 74. 섹션 헤더의 flag값들

각각이 의미하는 바는 아래의 표와 같다.

이름	의미
SHF_WRITE	프로세스의 실행중에 write 가능한 데이터를 섹션이 포함하고 있다는 뜻이다.
SHF_ALLOC	프로세스 실행중에 차지하는 메모리를 나타내는 섹션이다. 어떤 제어 섹션들은 object 파일의 메모리 이미지에 있지 않을 수도 있는데, 이러한

	경우 이 특성(attribute)는 꺼져(off) 있을 것이다.
SHF_EXECINSTR	이 섹션은 실행가능한 기계어 명령(instruction)들을 가진다.
SHF_MASKPROC	프로세서에 의존적인 의미를 가지며 예약되어 있다.

표 15. sh_flags필드의 bit정의

섹션 헤더의 필드중에서 sh_link와 sh_info는 섹션 타입에 따라 특별한 의미를 지닌다. 아래의 표를 보록 하자.

sh_type	sh_link	sh_info
SHT_DYNAMIC	섹션에 있는 엔트리들에 의해서 사용되는 스트링 테이블의 섹션 헤더 인덱스이다.	0
SHT_HASH	해쉬 테이블이 적용되는 심벌 테이블의 섹션 헤더 인덱스이다.	0
SHT_REL SHT_RELA	관련이 있는 심벌 테이블의 섹션 헤더 인덱스이다.	재배치(relocation)이 적용되는 섹션의 섹션 헤더 인덱스이다.
SHT_SYMTAB SHT_DYNSYM	관련이 있는 스트링 테이블의 섹션 헤더 인덱스이다.	마지막 local 심벌의 심벌 테이블 인덱스 보다 하나 큰 것을 나타낸다.(STB_LOCAL을 연결함(binding))
Other	SHN_UNDEF	0

표 16. 섹션 타입에 따른 sh_link와 sh_info의 해석

물론 이곳에서 전부를 이해하기는 힘들 것이다. 여기서는 이러한 것들이 있다는 것만 알면 될 것이다. 이전 특수한 의미를 가지는 섹션을 보도록 하자.

2.17.4. 특별한 의미를 가지는 섹션들

다양한 섹션들이 프로그램과 제어 정보를 포함하고 있다. 이들 중에서 아래에 보이는 섹션들은 시스템에서 사용되며, 타입과 특성(attribute)을 지니고 있다.

이름	타입	특성
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	none
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.debug	SHT_PROGBITS	none
.dynamic	SHT_DYNAMIC	아래를 보라.
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.got	SHT_PROGBITS	아래를 보라.
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	아래를 보라.
.line	SHT_PROGBITS	none
.note	SHT_PROGBITS	none
.plt	SHT_PROGBITS	아래를 보라.
.relname	SHT_REL	아래를 보라.
.relaname	SHT_REL_A	아래를 보라.
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC

.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	아래를 보라.
.symtab	SHT_SYMTAB	아래를 보라.
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

표 17. 특별한 의미를 가지는 섹션들

이들에 대한 설명을 보기 전에 간단한 C로 작성한 프로그램을 보도록 하자. 이 프로그램은 GCC를 이용해서 옵션으로 -S를 줘서 어셈블리 코드를 생성하도록 했을 때, 아래와 같은 코드를 가진다.

main.c	main.s
<pre>#include<stdio.h> int i=1; int main(int argc, char** argv) { int j; char k='k'; printf("This is a test program!!!\n"); }</pre>	<pre>.file "main1.c" .version "01.01" gcc2_compiled.: .globl i .data .align 4 .type i,@object .size i,4 i: .long 1 .section .rodata .LC0: .string "This is a test program!!!\n" .text .align 4 .globl main .type main,@function main: .pushl %ebp .movl %esp, %ebp .subl \$8, %esp .movb \$107, -5(%ebp) .subl \$12, %esp .pushl \$.LC0 .call printf .addl \$16, %esp .leave .ret .Lfe1: .size main,.Lfe1-main .ident "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.0)"</pre>

코드 75. C파일의 어셈블링

위의 프로그램에서 보고자 하는 것은 앞에서 본 섹션들이 실제 어셈블링된 파일에 어떻게 있는가를 보여주기 위한 것이다. 앞에서 본 내용 중에서 어셈블링 된 파일에서 찾을 수 있는 것은 .data와 .rodata, .text를 찾을 수 있을 것이다. C 코드를 보면 해당하는 부분이 어떤 정의를 가지고 있는지 확인할 수 있는데, 먼저 사용되지 않는 변수들에 대해서는 최적화 과정에서 제외된 것을 볼 수 있다(j와 k). 하지만 전역 변수로 사용된 변수(i)는 .data 섹션에서 정의되어 있으며, 나머지 상수 문자열도 .data 섹션에서 찾을 수 있다. 다만 이곳에는 .rodata로 정의된 읽기 전용의 문자열이

들어있다. .text에는 실제로 실행될 코드가 들어가며, .data와 마찬가지로 4bytes로 정렬하고 있다. 전체 프로그램의 크기는 .size로 나타내지며, .Lfe1-main의 값을 가진다.

이젠 위에서 정의한 섹션들의 의미를 보도록 하자.

섹션 이름	설명
.bss	초기화 되지 않은 데이터를 가지는 섹션으로, 프로그램의 메모리 이미지에 기여하고 있다. 정의에 의해서, 시스템은 프로그램이 수행될 때, 데이터를 0으로 초기화 한다. 파일의 공간은 차지하지 않기에 SHT_NOBITS의 섹션 타입을 가진다.
.comment	섹션이 버전 제어 정보를 가진다.
.data	초기화 된 데이터를 가지는 섹션으로 프로그램의 메모리 이미지에 기여한다.
.data1	
.debug	심벌의 디버깅을 위한 정보를 가지는 섹션이다. 내용은 명시되지 않는다.
.dynamic	동적인 링킹 정보를 가지는 섹션이다. 섹션의 성질은 SHF_ALLOC bit을 가질 것이다. SHF_WRITE bit이 설정되는 것은 프로세서 의존적이다.
.dynstr	동적인 링킹을 위해서 필요한 스트링들을 가지는 섹션이다. 심벌 테이블의 엔트리와 관련된 이름을 표시하는 스트링이 가장 일반적이다.
.dynsym	동적인 링킹 심벌 테이블을 가지는 섹션이다.
.fini	프로세스의 종료 코드에 기여하는 실행가능 명령들을 가지는 섹션이다. 즉, 프로그램이 일반적인 종료를 할 때, 시스템은 이 섹션에 있는 코드를 실행하기 위해서 준비할 것이다.
.got	전역 옵셋 테이블을 가지는 섹션이다.
.init	프로세스의 초기화 코드에 기여하는 실행가능 명령들을 가지는 섹션이다. 즉, 프로그램이 실행을 시작할 때, 시스템은 주(main) 프로그램의 진입점(entry point)을 부르기 전에 이 섹션에 있는 코드를 수행하기 위해서 준비할 것이다.
.interp	프로그램의 해석기(interpreter)의 경로 명(path name)을 가지는 섹션이다. 만약 파일이 적재 가능한 세그먼트를 가지고 있고, 이 세그먼트가 섹션을 가질 때, 섹션의 성질(attributes)은 SHF_ALLOC bit을 포함할 것이며, 그렇지 않다면 이 bit이 OFF될 것이다.
.line	심벌 디버깅을 위한 라인 정보를 가지는 섹션으로, 기계어 코드와 프로그램의 소스사이의 일치하는 부분을 기술한다. 섹션의 내용은 정해지지 않았다.
.note	“Note Section” 형식으로 들어있는 정보를 가지는 섹션이다.
.plt	프로시저(procedure) ³⁰ 의 링크 테이블을 가지는 섹션이다.
.relname .relaname	재배치 정보를 가지는 섹션이다. 만약 파일이 적재 가능한 세그먼트를 가지며, 이 세그먼트가 재배치 및 섹션의 성질을 포함하고 있다면, SHF_ALLOC bit이 설정될 것이다. 그렇지 않다면, 이 bit는 OFF될 것이다. 일반적으로 name은 재배치가 적용되는 섹션에 대해서 제공될 것이다. 예를 들어 .text에 대한 재배치 섹션은 일반적으로 .rel.text나 .rela.text라는 이름을 가지게 된다.
.rodata .rodata1	읽기 전용 데이터를 가지는 섹션으로 프로세스의 쓰기가 되지 않는 세그먼트 이미지를 만드는데 기여한다.
.shstrtab	섹션의 이름들을 가지는 섹션이다.
.symtab	이 섹션에 있는 “Symbol Table”이 기술하는 것과 같은 심벌 테이블을 가지는 섹션이다. 만약 파일이 적재 가능한 세그먼트를 가지고, 그 세그먼트가 심벌 테이블을 포함한다면, 섹션의 성질들은 SHF_ALLOC bit를 포함할 것이다. 그렇지 않다면 이 bit는 OFF된다.
.text	“text”를 가지는 섹션으로 프로그램의 실행가능한 명령을 지니고 있다.

표 18. 특별한 의미를 가지는 섹션들

점(dot: .)을 섹션의 이름에 가지는 섹션들은 시스템에 미리 예약되어 있지만, 응용 프로그램에서는 이미 존재하는 의미만 만족시킨다면 이러한 섹션들을 사용할 수 있다. 시스템의 섹션 이름과 상충되지

³⁰ Procedure는 함수와 동일하게 생각하면 될 것이다. 여러곳에 있는 정보를 얻다보니, 말이 조금씩 차이가 난다.

않는다면 점(dot..)을 빼고서 이러한 이름들을 사용할 수도 있다. Object 파일의 포맷은 위에서 정의되지 않은 이름에 대해서 사용자가 섹션을 정의할 수 있도록 하고 있다. 또한 Object 파일은 같은 이름을 가진 둘 이상의 섹션들도 가질 수 있다.

특정 프로세서의 architecture에 예약된 섹션의 이름들은 섹션의 이름 앞에 architecture의 이름의 약자를 붙여서 만들어지며, e_machine을 위해서 사용된 architecture 이름들로 부터 취해야 한다. 예를 들어, .FOO.psect는 FOO architecture에 의해 정의된 psect 섹션을 나타낸다. 현재 존재하는 확장명(extension)들은 그들의 역사적인(historical) 이름들에서 따왔다. 예를 들어 이미 존재하는 확장명에는 .sdata, .tdesc, .sbss, .lit4, .list8, .reginfo, .gptab, .liblist, .conflict 등이 있다.

커널도 일종의 실행 화일이다. 커널이 링킹 되는데 사용하는 스크립터 화일을 보면 어느정도 어떻게 각각의 섹션들이 배치가 되는지를 유추할 수 있다. 커널에 대한 링크는 LD가 처리하며, 아래에 보이는 것은 링크시에 사용되는 스크립트(script)인 ~/arch/i386/vmlinux.lds이다.

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
SECTIONS
{
    . = 0xC0000000 + 0x100000;
    _text = .;                                /* Text and read-only data */
    .text : {
        *(.text)
        *(.fixup)
        *(.gnu.warning)
    } = 0x9090
    .text.lock : { *(.text.lock) } /* out-of-line lock text */

    _etext = .;                                /* End of text section */
}
```

코드 76. vmlinux.lds 스크립트

커널의 최종적인 이미지의 output포맷은 ELF32_i386이다. 또한 architecture로는 i386을 가진다. 커널의 시작점(entry point)은 _start이며, 0xC0000000 + 0x100000이다. 즉, 커널은 최소 0xC0100000(3Gbytes + 1Mbytes)에서 시작점을 가진다. 이후에는 .text (코드) 섹션이 오게되며, _etext가 .text 섹션의 끝을 나타낸다.

```
.rodata : { *(.rodata) }
.kstrtab : { *(.kstrtab) }
.= ALIGN(16);                                /* Exception table */
_start__ex_table = .;
__ex_table : { *(__ex_table) }
_stop__ex_table = .;
_start__ksymtab = .;                          /* Kernel symbol table */
__ksymtab : { *(__ksymtab) }
_stop__ksymtab = .;
.data : {                                     /* Data */
    *(.data)
    CONSTRUCTORS
}
_edata = .;                                /* End of data section */
```

코드 77. vmlinux.lds 스크립트 – continued

.text 섹션 다음에는 커널에서 사용할 데이터의 섹션들이 나온다. .rodata와 .kstrtab은 각각 읽기 전용 섹션과 커널의 스트링 테이블을 가지는 섹션이다. 이하에는 16bytes 단위로 정렬되는 섹션들이 오는데, __start__ex_table과 __ex_table, __stop__ex_table은 Exception table을 가지며, __ksymtab는 커널 심벌

테이블의 섹션이다. `_stop_ksymtab` 역시 커널의 심벌 테이블 섹션이다. 데이터 섹션은 `.data`로 시작해서 `_edata`로 끝난다. 따라서, 커널의 코드와 데이터의 끝은 각각 `_etext`와 `_edata`를 보면 될 것이다.

```
. = ALIGN(8192);           /* init_task */
.data.init_task : { *(.data.init_task) }

.= ALIGN(4096);           /* Init code and data */
._init_begin = .;
.text.init : { *(.text.init) }
.data.init : { *(.data.init) }
.= ALIGN(16);
._setup_start = .;
.setup.init : { *(.setup.init) }
._setup_end = .;
._initcall_start = .;
.initcall.init : { *(.initcall.init) }
._initcall_end = .;
.= ALIGN(4096);
._init_end = .;

.= ALIGN(4096);
.data.page_aligned : { *(.data.idt) }

.= ALIGN(32);
.data.cacheline_aligned : { *(.data.cacheline_aligned) }
```

코드 78. vmlinux.lds 스크립트 – continued

이젠 커널의 최기화에서 사용하는 데이터 및 함수들을 가지는 섹션들에 대한 정의가 나온다. `.data.init_task`는 `init_task`들의 섹션들로, 아래에 보이는 `~/include/linux/init.h`에서 정의된 타입으로 선언되면 이곳을 차지한다.

```
#define __init          __attribute__ ((__section__ ("text.init")))
#define __exit          __attribute__ ((unused, __section__ ("text.exit")))
#define __initdata      __attribute__ ((__section__ ("data.init")))
#define __exitdata      __attribute__ ((unused, __section__ ("data.exit")))
#define __initsetup     __attribute__ ((unused, __section__ ("setup.init")))
#define __init_call     __attribute__ ((unused, __section__ ("initcall.init")))
#define __exit_call     __attribute__ ((unused, __section__ ("exitcall.exit")))
```

코드 79. init.h의 섹션과 관련된 정의

마지막으로 보이는 것이 IDT(Interrupt Descriptor Table)을 가지는 데이터 섹션과 데이터 캐쉬의 섹션들에 대한 정의이다. 각각 페이지 단위와 캐쉬라인(cache line: 16bytes) 단위의 정렬을 가진다.

```
__bss_start = .;           /* BSS */
.bss : {
    *(.bss)
}
._end = .;
/* Sections to be discarded */
/DISCARD/ : {
    *(.text.exit)
    *(.data.exit)
    *(.exitcall.exit)
}
/* Stabs debugging sections. */
```

```
.stab 0 : { *(.stab) }
.stabstr 0 : { *(.stabstr) }
.stab.excl 0 : { *(.stab.excl) }
.stab.exclstr 0 : { *(.stab.exclstr) }
.stab.index 0 : { *(.stab.index) }
.stab.indexstr 0 : { *(.stab.indexstr) }
.comment 0 : { *(.comment) }
}
```

코드 80vmlinux.lds 스크립트 – continued

이전 초기화되지 않은 커널 데이터에 대한 섹션 정의이다. `_bss_start`가 시작을 가르키며, `.bss` 섹션이 오고, `_end`가 그 끝을 나타낸다. 그 다음으로 오는 것은 버려도 되는 섹션들이다. `.text.exit`, `.data.exit`, `.exitcall.exit` 섹션이 이에 해당한다. 마지막으로 오는 섹션은 디버깅을 위한 스트링 테이블을 가지는 섹션들이다. 제일 마지막은 `.comment` 섹션이 차지하고 있다.

이와 같은 LD 스크립터로 링크된 커널은 objdump 프로그램으로 섹션들이 어떻게 object 파일 내에 정의되어 있는지를 볼 수 있는데, 이것을 보기 위해서는 압축되지 않은 커널을 보아야 한다. 커널 컴파일 시에 압축되지 않은 커널은 `~/arch/i386/boot/compressed/bvmlinux`가 있으므로 이것을 다음과 같이 보면 된다.

```
#objdump -h bvmlinux
bvmlinux:      file format elf32-i386

Sections:
Idx Name      Size    VMA     LMA   File off  Algn
 0 .text       00002075 00100000 00100000 00001000 2**2
              CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .rodata     000004b8 00102080 00102080 00003080 2**5
              CONTENTS, ALLOC, LOAD, READONLY, DATA
 2 .data        000c3594 00103538 00103538 00003538 2**2
              CONTENTS, ALLOC, LOAD, DATA
 3 .sbss        00000000 001c6acc 001c6acc 000c6ae0 2**0
              CONTENTS
 4 .bss         0000c480 001c6ae0 001c6ae0 000c6ae0 2**5
              ALLOC
 5 .comment    0000002e 00000000 00000000 000c6ae0 2**0
              CONTENTS, READONLY
 6 .note        00000014 00000000 00000000 000c6b0e 2**0
              CONTENTS, READONLY
```

코드 81. 커널에 대한 objdump의 실행 예

보는 방법은 섹션의 이름이 가장 먼저오며 섹션의 크기와 가상주소 및 선형주소, 파일내에서의 옵셋과 정렬 형태이다. 각 섹션의 두번째 라인은 세션의 성질을 나타낸다.

이상에서 우린 간단히 ELF 파일의 포맷에 대해서 알아보았다. 실제로 GCC와 같은 컴파일러로 프로그램을 컴파일 했다면, 이 ELF 파일의 포맷을 따르게 될 것이며, ELF 이외의 리눅스에서 정의하는 실행 파일 포맷에는 em86과 a.out, misc, script 형식 등이 있다. 각각은 `~/fs/binfmt_XXX.c`로 코드를 가지고 있으므로 참고하기 바란다.

2.18. Process간 통신(Interprocess Communication)

이번 장에서 다루고자 하는 것은 프로세스간 통신 방법이다. 프로세스간 통신(Interprocess Communication)은 간단히 IPC라고도 한다. 즉, 프로세스간의 의사전달 수단인 것이다. 하지만, 반드시 어떠한 것을 전달한다고 보기보다는 프로세스간에 실행을 동기적으로 맞출 수 있는 방법을 제공한다고

생각하는 것이 좋을 것이다. 먼저, 리눅스에서 지원하는 프로세스간 통신은 시그널(signal), 파이프(pipe), FIFO(First In First Out), 소켓(socket), 세마포어와 메시지 큐, 그리고, 공유 메모리를 사용하는 것이며, 이중에서 세마포어와 메시지 큐, 공유메모리는 System V 계열에서 지원하는 IPC(Inter Process Communication)방법이다. 이들 중에서 소켓을 이용하는 방법은 리눅스의 네트워크 부분에서 다루기로 하고, 시그널과 파이프, FIFO, 세마포어(semaphore)와 메시지(message), 그리고, 공유메모리(shared memory)를 이곳에서 다루기로 한다. 참고로, 공유메모리는 현재에 있어서 두개의 프로세스가 같은 메모리를 사용할 수 있도록 하는 유일한 방법이다.

리눅스 커널에서는 시그널은 단순히 프로세스의 task_struct 구조체를 이용해서 주고 받을 수 있도록 하고 있으며, 파이프와 FIFO는 파일 시스템과 관련된 inode를 대상으로 프로세스간에 정보를 교환할 수 있도록 만들고 있다.

리눅스 커널에서는 시스템 V계열의 IPC를 지원하기 위해서 IPC방법의 각각에 대해서 객체(object)들을 정의하고 있으며, 자원관리의 대상이 된다. 또한 각각의 객체들은 고유한 ID를 가지고 있고, 동적으로 생성된다. 따라서, 이러한 객체들을 사용하기를 원하는 프로세스들은 적어도 ID는 알고 있어야 할 것이다. 커널에서는 또한 생성된 객체들에 대한 접근 권한(access permission)을 두고 있으며, 아래와 같이 정의된다.

```
/* used by in-kernel data structures */
struct kern_ipc_perm
{
    key_t          key;      /*IPC를 사용하기 위해서 생성할때 커널에서 주어진 key값*/
    uid_t          uid;      /*사용자 ID*/
    gid_t          gid;      /*그룹 ID.*/
    uid_t          cuid;     /*생성한 사용자의 ID*/
    gid_t          cgid;     /*생성한 그룹의 ID*/
    mode_t         mode;     /*접근허가 모드*/
    unsigned long seq;      /*ID를 계산하기 위해서 사용되는 counter값*/
};
```

코드 82. IPC 접근허가(permission) 구조체의 정의

만약 프로세스가 커널에서 생성한 IPC 객체들을 접근하고자 한다면, 위에서 정의한 kern_ipc_perm을 확인하도록 하며, ipcperm() 함수가 맡고 있다. ipcperm() 함수에서는 넘겨받은 kern_ipc_perm과 flag(접근하고자 하는 모드)를 현재의 모드와 비교하고, 사용자의 ID들에 대한 접근 권한을 확인하는 절차를 밟게 된다. 물론 super user에 대한 경우에는 모든 권한이 다 허가가 될 것이다. 관련된 파일로는 ~/include/linux/ipc.h와 ~/ipc이하에 나와있는 파일들이 있다. 시스템 V계열의 IPC를 다룰 때 좀더 자세히 보게 될 것이다.

2.18.1. Signal

시그널은 단순히 어떤 프로세스의 의사를 전달하는 목적만을 가진다. 즉, 특정한 사건이 일어났음을 알릴 목적으로 사용하는 IPC방법이다. 많은 데이터를 전송하지는 못하며, 단순히 task_struct에 signal을 나타내는 bit를 설정하는 일만 할 뿐이다. 나중에 이를 위한 처리는 시그널을 받는 프로세스에 전적으로 의존한다. i386상에서 정의된 리눅스에서 사용하는 시그널에는 아래와 같은 것들이 있다.

#define SIGHUP	1	/* 프로세스나 제어 터미널을 HANGUP 시킨다. */
#define SIGINT	2	/* 키보드로 부터의 인터럽트 */
#define SIGQUIT	3	/* 키보드로 부터의 QUIT */
#define SIGILL	4	/* 잘못된 명령 */
#define SIGTRAP	5	/* 디버깅을 위한 break point */
#define SIGABRT	6	/* 비정상적인 종료 */
#define SIGIOT	6	/* SIGABRT와 동일함 */
#define SIGBUS	7	/* Bus 에러 */
#define SIGFPE	8	/* Floating point 에러 */

#define SIGKILL	9	/* 프로세스를 종료 */
#define SIGUSR1 10		/* 사용자 정의 시그널 */
#define SIGSEGV 11		/* 잘못된 메모리에 대한 접근 */
#define SIGUSR2 12		/* 사용자 정의 시그널 */
#define SIGPIPE 13		/* 읽는 프로세스가 없는 상황에서의 PIPE에 대한 쓰기 */
#define SIGALRM 14		/* 시간의 경과 */
#define SIGTERM 15		/* 프로세스의 종료 */
#define SIGSTKFLT 16		/* 보조 프로세스의 stack 에러 */
#define SIGCHLD 17		/* 자식 프로세스의 종료나 멈춤 */
#define SIGCONT 18		/* 현재 프로세스가 멈추었다면, 실행을 다시 시작 */
#define SIGSTOP 19		/* 프로세스의 실행을 멈춤 */
#define SIGTSTP 20		/* TTY에서 발생한, 프로세스의 실행 중단 */
#define SIGTTIN 21		/* Background 프로세스가 input을 필요로 함 */
#define SIGTTOU 22		/* Background 프로세스가 output을 필요로 함 */
#define SIGURG 23		/* Socket에 urgent한 상황이 발생함 */
#define SIGXCPU 24		/* CPU 시간의 한계를 초과했음 */
#define SIGXFSZ 25		/* 파일 크기의 한계를 초과했음 */
#define SIGVTALRM 26		/* Virtual timer의 clock */
#define SIGPROF 27		/* Profile을 위한 timer의 clock */
#define SIGWINCH 28		/* 윈도우의 크기 변화 */
#define SIGIO 29		/* 현재 I/O처리가 가능함 */
#define SIGPOLL SIGIO		/* SIGIO와 동일함 */
#define SIGPWR 30		/* 파워 공급장치의 오류 */
#define SIGSYS 31		/* 사용되지 않음 */
#define SIGUNUSED 31		/* 사용되지 않음 */

코드 83. 시그널의 정의

총 31개의 시그널을 정의해서 사용하고 있으며, POSIX³¹에서 지원하지 않는 시그널도 존재한다. 리눅스에서는 기본적으로 이들 시그널에 대해서 프로세스가 어떤 식으로 반응할지를 미리 정의해 두었는데, 프로세스의 실행을 중단하거나, 혹은 core dump를 만들기도 한다. 물론 전달 받은 시그널을 기본적으로 무시할 수도 있으며, 아니면 시그널 핸들러(signal handler)를 만들어, 해당하는 시그널을 받았을 경우에 프로세스가 어떤 행동을 취할지를 프로그램마다 결정할 수 있도록 하고 있다.

시그널이 pending상태로 있다는 것은 시그널이 전송되었지만, 아직 받지는 않은 상태를 말한다. 즉, 프로세스가 시그널을 처리하지 않아다는 것을 의미한다. 시그널은 프로세스에 대해서 단지 한번에 시그널 타입 당 하나의 시그널만이 pending상태에 있을 수 있다. 만약 같은 타입의 시그널이 이미 받는 프로세스에 대해서 pending된 상태로 있다면, 보내려는 시그널은 없어질 것이다.

시그널 핸들러(signal handler)는 해당하는 시그널을 받았을 때 처리를 맡는 함수로서, 프로세스가 등록할 수 있으며, 없을 경우에는 시스템이 디폴트(default)로 지정된 핸들러를 이용해서 위에서 보았던 중단, core dump, 무시(ignore)와 같은 행동들을 취할 것이다. 시그널 핸들러는 먼저 처리하고자 하는 시그널이 더이상 발생하지 못하도록 만든다(masking). 따라서, 또 다른 동일한 핸들러가 진행하지 못하도록 만들어주고, 자신이 처리하고자 하는 일을 한다. 처리를 마치면 이전에 자신이 했던 일을 되돌려주어서, 동일한 시그널을 다시 받아서 처리될 수 있다(unmasking).

또한 시그널은 반드시 실행중인 프로세스에 의해서만 처리될 수 있다. 즉, 프로세스가 실행중이 아니라면, 보내진 시그널은 프로세스의 주의를 끌 수 없다. 이와 같은 이유로 인해서, 시그널을 보낸 시점과 처리되는 시점 사이에는 얼마만큼의 시간이 지날지 알 수 없게 된다.

³¹ Unix의 표준화를 위해서 만들어진 표준규격. 즉, 사용자의 입장에서 동일한 인터페이스로 프로그램의 이식성(portability)을 높이기 위해서, 하위의 운영체제에서 지원해야하는 기능에 대해서 표준화 시킨 것.

커널은 항상 커널 모드에서 사용자 모드로 바뀔 때, 전달을 기다리는 시그널이 있는지를 확인하게 되며, 관련된 프로세스의 `task_struct`의 필드를 보고, 시그널을 어떻게 처리할지를 결정하게 된다. 만약 `blocking`이 되어있다면, 커널은 시그널을 무시할 것이다.

시그널을 받은 프로세스는 만약 시그널 핸들러가 처리하고자 하는 시그널일 경우에는 핸들러를 즉시 호출한다. 따라서, 프로세스는 어느 때이고 간에 시그널을 받을 수 있기 때문에, 이 핸들러를 부르기 위해서는 반드시 자신의 현재 상태를 저장해 두어야 하며, 나중에 핸들러 처리를 마쳤을 때, 원래의 상태로 다시 복귀하게 된다.

시그널과 관련된 시스템 콜들을 보기로 하자. 이 함수들은 프로세스와 관련해서 시그널을 보내거나, 시그널과 관련된 처리를 바꾼다던가 등의 일을 한다. 아래와 같다. `~/kernel/signal.c`를 참조하도록 하자.

시스템 콜	설명
<code>sys_rt_sigprocmask()</code>	블록킹된 리얼타임 시그널의 집합(set)을 바꾼다(modify).
<code>sys_rt_sigpending()</code>	리얼타임 시그널이 pending되어 있는지 확인한다.
<code>sys_rt_sigtimedwait()</code>	리얼타임 시그널을 timeout을 설정해서 기다린다.
<code>sys_kill()</code>	프로세스에 시그널을 전달한다.
<code>sys_rt_sigqueueinfo()</code>	프로세스에 리얼타임 시그널을 보낸다.
<code>sys_sigpending()</code>	Pending된 시그널이 있는지 확인한다.
<code>sys_sigprocmask()</code>	블록킹된 시그널의 집합을 바꾼다.
<code>sys_rt_sigaction()</code>	리얼타임 시그널과 관련된 취해야 할 행동(action)을 바꾼다.
<code>sys_sgetmask()</code>	블록킹된 시그널의 mask값을 가져온다.
<code>sys_ssetmask()</code>	블록킹된 시그널의 mask값을 설정한다.
<code>sys_signal()</code>	시그널과 관련된 취해야 할 행동을 바꾼다.

표 19. 시그널과 관련된 시스템 콜 인터페이스

이전 버전에서 지원하는 것과의 호환을 위해서 `sys_sigaction()`은 `sys_signal()`에서 처리하고, `sys_procmask()`는 `sys_sgetmask()`와 `sys_ssetmask()`가 각각 나누어서 처리한다.

커널의 내부에서 시그널을 보내는 것은 `send_signal()` 함수가 처리한다. 코드는 아래와 같으며, `~/kernel/signal.c`에 정의되어 있다

```
static int send_signal(int sig, struct siginfo *info, struct sigpending *signals)
{
    struct sigqueue * q = NULL;
    if (atomic_read(&nr_queued_signals) < max_queued_signals) {
        q = kmem_cache_alloc(sigqueue_cachep, GFP_ATOMIC);
    }

    if (q) {
        atomic_inc(&nr_queued_signals);
        q->next = NULL;
        *signals->tail = q;
        signals->tail = &q->next;
        switch ((unsigned long) info) {
            case 0:
                q->info.si_signo = sig;
                q->info.si_errno = 0;
                q->info.si_code = SI_USER;
                q->info.si_pid = current->pid;
                q->info.si_uid = current->uid;
                break;
            case 1:

```

```

        q->info.si_signo = sig;
        q->info.si_errno = 0;
        q->info.si_code = SI_KERNEL;
        q->info.si_pid = 0;
        q->info.si_uid = 0;
        break;
    default:
        copy_siginfo(&q->info, info);
        break;
}

```

코드 84. send_signal()함수 – 시그널 큐의 생성

먼저 시그널 큐의 캐시(sigqueue_cachep)로부터 시그널 큐(sigqueue)를 생성한다. 만약 정확히 할당 받았다면, 이하의 부분을 수행한다.

큐잉된 시그널의 수를 증가시키고, q를 초기화하며 pending된 시그널(signals)의 꼬리(tail)부분에 시그널 큐를 넣는다. 그리고, 사용자 프로세스에서 전송된 시그널인지, 아니면 커널에서 전송한 시그널인지를 확인한 다음 할당 받은 시그널 큐를 초기화 해준다.

```

} else if (sig >= SIGRTMIN && info && (unsigned long)info != 1
          && info->si_code != SI_USER) {
    /*
     * Queue overflow, abort. We may abort if the signal was rt
     * and sent by user using something other than kill().
     */
    return -EAGAIN;
}
sigaddset(&signals->signal, sig);
return 0;
}

```

코드 85. send_signal()함수(continued) – 시그널 큐의 생성 확인

만약 제대로 시그널 큐를 할당 받지 못했다면, 큐가 이미 많이 있는 상태가 된다. 따라서, 다시 시도하도록 에러(-EAGAIN)를 돌려주고 복귀한다. 함수의 복귀 시에 pending된 시그널의 bit를 표시하고 0을 돌려준다.

여기서, siginfo구조체와 sigpending구조체 및 sigqueue구조체를 보기로 하자. 정의는 뒤의 두가지는 ~/include/linux/signal.h에 있으며, 나머지 하나는 ~/include/asm/siginfo.h에 있다.

```

#define _NSIG          64
#define _NSIG_BPW      32
#define _NSIG_WORDS    (_NSIG / _NSIG_BPW)
...
typedef struct {
    unsigned long sig[_NSIG_WORDS];
} sigset_t;
...
struct sigqueue {
    struct sigqueue *next; /* 다음의 sigqueue에 대한 포인터 */
    siginfo_t info;        /* 관련된 siginfo구조체 */
};
struct sigpending {
    struct sigqueue *head, **tail; /* pending된 시그널의 큐에 대한 큐*/
    sigset_t signal;           /* pending된 시그널의 bitmap(32bit) */
};

```

```
};
```

코드 86. sigqueue와 sigpending 구조체의 정의

즉, 시그널이 pending된 경우에는 sigqueue구조체를 할당 받아서 sigpending구조체의 tail부분에 들어가게 되며, pending된 시그널의 정보는 sigqueue의 info필드를 차지하게 된다. 또한 pending 시그널의 bit는 sigpending구조체의 signal필드에 설정(set)된다. 여기서 signal필드는 sigset_t의 _NSIG_WORDS 갯수 만큼의 unsigned long값을 가진다.

```
typedef union sigval {
    int sival_int;
    void *sival_ptr;
} sigval_t;

#define SI_MAX_SIZE      128
#define SI_PAD_SIZE     ((SI_MAX_SIZE/sizeof(int)) - 3)

typedef struct siginfo {
    int si_signo;           /* 시그널의 번호 */
    int si_errno;           /* 에러 번호 */
    int si_code;            /* 사용자 프로세스 혹은 커널 */
    union {
        int _pad[SI_PAD_SIZE]; /* Padding */
        /* kill() */
        struct {
            pid_t _pid;          /* 보내는 프로세스의 ID */
            uid_t _uid;          /* 보내는 프로세스의 사용자 ID */
        } _kill;
        /* POSIX.1b timers */
        struct {
            unsigned int _timer1; /* 시그널과 관련된 타임머들 */
            unsigned int _timer2;
        } _timer;
        /* POSIX.1b signals */
        struct {
            pid_t _pid;          /* 보내는 프로세스의 ID */
            uid_t _uid;          /* 보내는 프로세스의 사용자 ID */
            sigval_t _sigval;
        } _rt;
        /* SIGCHLD 시그널을 위한 구조체 */
        struct {
            pid_t _pid;          /* 어느 자식 프로세스로 보낼 것인가? */
            uid_t _uid;          /* 보내는 프로세스의 사용자 ID */
            int _status;          /* 프로세스의 종료 상태 */
            clock_t _utime;       /* 프로세스의 사용자 모드에서 사용한 시간 */
            clock_t _stime;       /* 프로세스의 시스템 모드(커널 모드)에서 사용한 시간
        } _sigchld;
        /* SIGILL, SIGFPE, SIGSEGV, SIGBUS */
        struct {
            void *_addr; /* 메모리에 대한 언급과 관련된 에러일 경우, 언급한 주소 */
        } _sigfault;
        /* SIGPOLL */
        struct {
            int _band; /* POLL_IN, POLL_OUT, POLL_MSG */
        } _sigpoll;
    };
}
```

```

        int _fd;      /* 파일 디스크립터 설정. Polling과 관련서 어느 FD를 볼 것인가?
*/
    } _sigpoll;
} _sifields;
} siginfo_t;

```

코드 87. siginfo구조체의 정의

siginfo구조체의 정의는 위에서와 같이 전달하고자 하는 시그널의 특성에 따라서 각각 보관하는 정보가 달라진다. 이것은 시그널을 보낼 때 상대방에 어떤 정보를 보낼지를 결정하고, 각각에 해당하는 siginfo구조체 필드를 완성해 주어야 할 것이다.

시그널을 받는 것은 ret_from_intr() 함수에서 실행된다. 커널은 모든 인터럽트와 예외(exception)상황이 발생한 후에 커널 모드에서 사용자 모드로 진입하기 전에, 블록킹 되지 않은 pending 시그널이 존재하는지를 확인한다. 아래와 같다. ~/arch/i386/kernel/entry.S에 정의되어 있다.

```

ENTRY(ret_from_intr)
    GET_CURRENT(%ebx)
    movl EFLAGS(%esp),%eax          # mix EFLAGS and CS
    movb CS(%esp),%al
    testl $(VM_MASK | 3),%eax # return to VM86 mode or non-supervisor?
    jne ret_with_reschedule
    jmp restore_all
...
ret_with_reschedule:
    cmpl $0,need_resched(%ebx)
    jne reschedule
    cmpl $0,sigpending(%ebx)
    jne signal_return
restore_all:
    RESTORE_ALL

    ALIGN
signal_return:
    sti                                # we can get here from an interrupt handler
    testl $(VM_MASK),EFLAGS(%esp)
    movl %esp,%eax
    jne v86_signal_return
    xorl %edx,%edx
    call SYMBOL_NAME(do_signal)
    jmp restore_all

    ALIGN
v86_signal_return:
    call SYMBOL_NAME(save_v86_state)
    movl %eax,%esp
    xorl %edx,%edx
    call SYMBOL_NAME(do_signal)
    jmp restore_all

```

코드 88. ret_from_intr()의 수행

ret_from_intr()은 어셈블리어로 되어있다. 먼저 현재의 프로세스의 task_struct를 구해온다(GET_CURRENT). 그리고, 어떤 모드에서 인터럽트가 발생했는지를 확인한다. 즉, VM86 모드나 사용자 모드였는지, 아니면 커널 모드였는지를 확인한다. VM86모드나 사용자 모드였다면 retore_all로 제어를 옮겨서 저장했던 레지스터들을 복구하고 바로 복귀하고, 그렇지 않다면 ret_with_reschedule로 제어를 옮긴다.

`ret_with_reschedule`로 제어가 옮겨왔다면, 먼저 현재의 프로세스의 `task_struct`에(%ebx) `need_resched`필드가 설정되었는지를 확인하고, 만약 설정되었다면 `reschedule`로 제어를 옮긴다. 그리고 나서, 현재 프로세스 `task_struct`의 `sigpending`필드가 설정되었는지 확인하고, 만약 설정되었다면 `signal_return`으로 제어를 옮긴다. 설정되지 않았다면, 위에서와 같이 저장했던 레지스터들을 복구(`restore_all`)하고 복귀한다.

이전 `signal_return`으로 넘어갔다고 보면, 여기서는 진행하기 전에 인터럽트들을 발생할 수 있도록 만들어주고(sti), 어떤 모드에서 인터럽트가 발생했는지를 확인한다. 만약 VM86 모드에서 발생했다면, 다시 `v86_signal_return`으로 제어를 옮기게 된다. 그렇지 않다면, 이전 `do_signal()`함수를 호출하게 된다. `do_signal()`함수가 끝나면, 모든 레지스터의 내용을 복구하고 복귀할 것이다.

`v86_signal_return`으로 제어가 옮겨간 상태에서는 `save_v86_state`를 호출해서 VM86 모드에서 사용하던 레지스터의 내용을 저장하고, `do_signal()`함수를 호출한 후 레지스터의 내용을 복구하고 복귀할 것이다. 따라서, 실제 시그널의 확인만이 `ret_from_intr()` 함수에서 행해지며, 처리는 `do_signal()`함수를 불러서 하고 있다.

`do_signal()`함수는 `~/arch/i386/kernel/signal.c`에 정의되어 있다. 코드는 아래와 같다.

```
int do_signal(struct pt_regs *regs, sigset_t *oldset)
{
    siginfo_t info;
    struct k_sigaction *ka;

    if ((regs->xcs & 3) != 3)
        return 1;
    if (!oldset)
        oldset = &current->blocked;
```

코드 89. `do_signal()`함수 – 파라미터 값의 확인

먼저 넘겨받는 것들을 보도록 하자. 넘겨받는 정보에는 레지스터(regs)들과 `sigset_t`의 시그널 set정보가 있다. 레지스터의 정보에는 현재 프로세스의 사용자 모드에서 사용 중이던 레지스터들이 들어있는 스택이 넘어오게 되며, `oldset`은 블록킹된 시그널의 bit mask를 저장할 공간이 된다. 실제로 `oldset`은 `ret_from_intr()` 함수에서 진행되어왔다면, NULL값을 가질 것이다. 위에서 보면 NULL값을 때는 현재 프로세스의 블록킹 시그널의 bit mask의 주소를 가진다. 만약 사용자 모드에서 진행 중이었다면 그대로 1을 돌려주고 복귀한다.

```
for (;;) {
    unsigned long signr;

    spin_lock_irq(&current->sigmask_lock);
    signr = dequeue_signal(&current->blocked, &info);
    spin_unlock_irq(&current->sigmask_lock);
    if (!signr)
        break;
    if ((current->ptrace & PT_PTRACED) && signr != SIGKILL) {
        /* Let the debugger run. */
        current->exit_code = signr;
        current->state = TASK_STOPPED;
        notify_parent(current, SIGCHLD);
        schedule();
        /* We're back. Did the debugger cancel the sig? */
        if (!(signr == current->exit_code))
            continue;
        current->exit_code = 0;
        /* The debugger continued. Ignore SIGSTOP. */
        if (signr == SIGSTOP)
            continue;
        /* Update the siginfo structure. Is this good? */
        if (signr != info.si_signo) {
```

```

        info.si_signo = signr;
        info.si_errno = 0;
        info.si_code = SI_USER;
        info.si_pid = current->p_pptr->pid;
        info.si_uid = current->p_pptr->uid;
    }
    /* If the (new) signal is now blocked, requeue it. */
    if (sigismember(&current->blocked, signr)) {
        send_sig_info(signr, &info, current);
        continue;
    }
}

```

코드 90. do_signal()함수(continued) – Pending된 시그널의 처리

무한 루프(loop)를 돌면서 현재 프로세스에 대해서 pending된 시그널들을 처리하는 부분이다. 먼저 현재의 프로세스의 시그널 마스크(signal mask)에 대한 lock을 구한다. 그리고 나서 pending된 시그널을 하나 가져온다(dequeue_signal()). 걸었던 lock을 해제하고, 만약 더 이상 가져올 시그널이 존재하지 않는다면 loop를 나간다.

현재 프로세스에 대해서 PT_PTRACED가 설정되어있고, 가져온 시그널이 SIGKILL이 아닌경우에는 debugging을 위해서 debugger가 동작하는 상황이므로, 프로세스의 exit_code를 시그널 번호로 놓고, 프로세스를 멈춘다(TASK_STOPPED). 그리고 나서 부모 프로세스에게 시그널(SIGCHLD)을 보내고, 프로세스의 스케줄링을 요구한다(schedule()). 부모 프로세스로부터 계속 진행을 명령 받고, 스케줄링이 되었을 경우에 실행되는 부분이다. 먼저 현재 프로세스의 종료코드로 저장해 두었던 시그널 번호를 가져온다. 만약 없다면 다음 번 루프로 들어가게 된다(continue).

프로세스의 종료코드를 다시 0으로 두고, 만약 이전에 가졌던 시그널이 SIGSTOP일 경우에는 다시 루프를 돈다(continue). 만약 siginfo구조체에 있는 시그널과 현재 처리중인 시그널의 번호가 일치하지 않을 경우에는 siginfo구조체를 갱신한다. 만약 처리중인 시그널이 블록킹된 시그널에 속한다면 현재의 프로세스로 siginfo구조체를 보내준다(send_sig_info()).

```

ka = &current->sig->action[signr-1];
if (ka->sa.sa_handler == SIG_IGN) {
    if (signr != SIGCHLD)
        continue;
    /* Check for SIGCHLD: it's special. */
    while (sys_wait4(-1, NULL, WNOHANG, NULL) > 0)
        /* nothing */;
    continue;
}

```

코드 91. do_signal()함수(continued) – 시그널 액션의 처리

여기서부터는 시그널의 처리에 관련된 것이다. 현재 프로세스의 시그널에 해당하는 행동(action)을 실행해주는 부분이다. 만약 핸들러(handler)가 SIG_IGN(ignore: 무시)라고 설정된 경우에는, 시그널 번호가 SIGCHLD가 아니라면 루프를 계속 진행하고, 그렇지 않다면 sys_wait4()를 호출해서 현재 프로세스를 자식 프로세스의 종료를 기다리도록 만든다. 만족되면 다시 루프를 진행한다.

```

if (ka->sa.sa_handler == SIG_DFL) {
    int exit_code = signr;
    /* Init gets no signals it doesn't want. */
    if (current->pid == 1)
        continue;
    switch (signr) {
    case SIGCONT: case SIGCHLD: case SIGWINCH:
        continue;
    case SIGTSTP: case SIGTTIN: case SIGTTOU:

```

```

        if (is_orphaned_pgrp(current->pgrp))
            continue;
        /* FALLTHRU */
    case SIGSTOP:
        current->state = TASK_STOPPED;
        current->exit_code = signr;
        if (!!(current->p_pptr->sig->action[SIGCHLD-1].sa.sa_flags &
SA_NOCLDSTOP))
            notify_parent(current, SIGCHLD);
        schedule();
        continue;
    case SIGQUIT: case SIGILL: case SIGTRAP:
    case SIGABRT: case SIGFPE: case SIGSEGV:
    case SIGBUS: case SIGSYS: case SIGXCPU: case SIGXFSZ:
        if (do_coredump(signr, regs))
            exit_code |= 0x80;
        /* FALLTHRU */
    default:
        sigaddset(&current->pending.signal, signr);
        recalc_sigpending(current);
        current->flags |= PF_SIGNALLED;
        do_exit(exit_code);
        /* NOTREACHED */
    }
}

```

코드 92. do_signal() 함수(continued) – 시그널 액션의 처리

만약 핸들러가 SIG_DFL(default: 기본 설정)으로 되어있다면, 해당하는 기본 연산을 수행한다. 종료코드로 시그널 번호를 두고, 만약 init 프로세스(PID==1)인 경우에는 루프를 계속 실행한다. 그 외의 SIGCONT, SIGCHLD, SIGWINCH는 루프의 계속 진행을, SIGTSTP, SIGTTIN, SIGTTOU은 현재 프로세스가 속한 프로세스 그룹이 orphaned³² 프로세스 그룹이 되는지를 확인하고 계속 진행을 하거나, 아니면 SIGSTOP를 받도록 만들며, SIGSTOP은 프로세스를 멈추고 부모에 SIGCHLD 시그널을 보내며, SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGFPE, SIGSEGV, SIGBUS, SIGSYS, SIGCPU, SIGXFSZ는 코아(core) 덤프(do_coredump())를 생성하고, 종료코드를 0x80과 OR시켜서 복귀코드를 넘긴다.

이하의 부분도 위에서 코아 덤프를 생성한 것과 연결되어서 실행되어지며, 정의된 시그널이 없을 때에 해당한다. 먼저 현재 프로세스의 pending된 시그널에 추가하고, 현재 프로세스의 pending 시그널을 새로 계산한다(recalc_sigpending()). 그리고, 프로세스의 flag값을 PF_SIGNALLED(시그널을 받았음)을 표시한 후에 복귀한다(do_exit()).

```

__asm__("movl %0,%%db7" : : "r" (current->thread.debugreg[7]));
/* Whee! Actually deliver the signal. */
handle_signal(signr, ka, &info, oldset, regs);
return 1;
}
/* Did we come from a system call? */
if (regs->orig_eax >= 0) {
    /* Restart the system call - no handlers present */
    if (regs->eax == -ERESTARTNOHAND ||
        regs->eax == -ERESTARTSYS ||
        regs->eax == -ERESTARTNOINTR) {
        regs->eax = regs->orig_eax;
        regs->eip -= 2;
    }
}

```

³² 프로세스의 그룹 리더가 종료한 상태.

```

    }
    return 0;
}

```

코드 93. do_signal() 함수(continued) – 시그널 핸들러의 호출

이전 기본 행동(default action)이나 무시(ignore)가 아닌 경우에 프로세스의 등록된 시그널 핸들러를 호출하는 부분이다(handle_signal()). 여기까지 진행했다면 1을 돌려주고 복귀한다. 만약 현재의 진행된 상황이 시스템 콜에서 왔고, 핸들러가 정해지지 않은 경우에는 시스템 콜을 다시 실행하도록 한다.

여기서 한가지 의문이 든다. 현재의 실행모드는 커널 모드이고, 시그널 핸들러는 사용자 모드에서 설정되었는데, 어떻게 커널에서 사용자 모드의 핸들러를 실행해 줄 것인가이다. 즉, 커널은 자신의 스택을 사용자 모드로 전환하게 되면 다 잊어먹기 때문이다. 따라서, 핸들러를 수행하고 난 이후에는 사용자 모드에서 이전에 커널 모드 상황으로 진행할 수 없게 된다.

이것을 해결해 주는 방법이 커널모드의 스택에 저장된 하드웨어 정보를 잠시동안 현재 프로세스의 사용자 모드 스택에 저장하는 방법이다. 나중에 사용자 모드의 시그널 핸들러가 수행을 마치면 sigreturn() 함수가 실행되어, 현재 프로세스의 사용자 모드 스택에 저장된 커널 스택의 하드웨어 정보를 다시 커널 스택으로 자동적으로 복사한다. 그리고, 사용자 모드 스택은 원래의 것으로 복구한다.

시그널의 핸들러에 대해서 좀더 보기로 하자. 먼저 시그널 핸들러를 위해서 정의된 sigaction 구조체를 보도록 하자. ~/include/asm/signal.h를 참고하자.

```

typedef void (*__sighandler_t)(int); /* 시그널 핸들러 타입의 정의 */
...
struct sigaction {
    __sighandler_t sa_handler; /* 시그널 핸들러의 포인터 */
    unsigned long sa_flags; /* 시그널 action의 flag값 */
    void (*sa_restorer)(void); /* 복구 처리 핸들러의 포인터 */
    sigset_t sa_mask; /* 시그널 mask */
};

...
struct k_sigaction {
    struct sigaction sa;
};

...
/* ~/include/linux/sched.h에 정의된 task_struct의 시그널처리를 위한 구조체 */
struct signal_struct {
    atomic_t count; /* 현재 프로세스가 가지고 있는 시그널 핸들러의 수 */
    struct k_sigaction action[_NSIG]; /* 시그널 action의 배열 */
    spinlock_t siglock; /* signal_struct를 다루는데 사용되는 lock */
};

```

코드 94. 시그널 핸들러 구조체의 정의

시그널 핸들러를 설정하는 것은 sys_signal() 시스템 콜이다. ~/kernel/signal.c에 정의되어 있다.

```

asmlinkage unsigned long sys_signal(int sig, __sighandler_t handler)
{
    struct k_sigaction new_sa, old_sa;
    int ret;

    new_sa.sa.sa_handler = handler;
    new_sa.sa.sa_flags = SA_ONESHOT | SA_NOMASK;
    ret = do_sigaction(sig, &new_sa, &old_sa);
    return ret ? ret : (unsigned long)old_sa.sa.sa_handler;
}

```

{}

코드 95. sys_signal() 시스템 콜

sys_signal() 시스템 콜은 사용자 프로세스로부터 시그널 번호와 시그널 핸들러에 포인터를 넘겨받는다. 새로운 시그널 핸들러와 이전에 있던 시그널 핸들러에 대한 k_sigaction구조체를 할당 받아서, 새로운 시그널 핸들러의 flags값을 SA_ONESHOT과 SA_NOMASK를 설정한 다음 do_sigaction() 함수를 호출한다. 잘되었다면 이전에 있던 시그널 핸들러의 값을 돌려주고, 그렇지 않으면 do_sigaction() 함수의 복귀 값을 돌려준다.

이때 sigaction구조체의 flag값으로 줄 수 있는 것으로는 아래와 같은 것이 있다.

Flag값	설명
SA_NOCLDSTOP	프로세스가 멈추었을 경우, SIGCHLD시그널을 부모 프로세스로 전달하지 마라.
SA_NOCLDWAIT	아직 지원되지 않음(커널 버전 2.4.0에서)
SA_SIGINFO	시그널 핸들러에 부가적인 정보를 제공함
SA_ONSTACK	시그널 핸들러를 위해서 다른(alternate) 스택을 사용하라.
SA_RESTART	인터럽트된 시스템 콜을 자동적으로 다시 시작하라.
SA_NODEFER	시그널 핸들러를 실행중일 때는 시그널을 masking하지 마라.
SA_RESETHAND	시그널 핸들러를 실행한 후에는 default action으로 재 설정하라.
SA_NOMASK	SA_NODEFER
SA_ONESHOT	SA_RESETHAND
SA_INTERRUPT	사용되지 않음(무시)
SA_RESTORER	사용자 모드 스택을 복구하는 복구(restor)처리 핸들러를 설치한다.

표 20. sigaction구조체의 flag설정 값

따라서, 위에서 설정한 flag는 한번만 실행하고, 실행도중에 시그널을 masking하지 않는 시그널 핸들러를 설치한다는 의미가 된다.

do_sigaction()함수는 ~/kernel/signal.c에 정의되어 있다. 이 함수는 주어진 핸들러로 시그널의 핸들러를 설정한다.

```
int do_sigaction(int sig, const struct k_sigaction *act, struct k_sigaction *oact)
{
    struct k_sigaction *k;

    if (sig < 1 || sig > _NSIG ||
        (act && (sig == SIGKILL || sig == SIGSTOP)))
        return -EINVAL;
    k = &current->sig->action[sig-1];
    spin_lock(&current->sig->siglock);
    if (oact)
        *oact = *k;
    if (act) {
        *k = *act;
        sigdelsetmask(&k->sa.sa_mask, sigmask(SIGKILL) | sigmask(SIGSTOP));
        if (k->sa.sa_handler == SIG_IGN
            || (k->sa.sa_handler == SIG_DFL
                && (sig == SIGCONT ||
                     sig == SIGCHLD ||
                     sig == SIGWINCH))) {
            spin_lock_irq(&current->sigmask_lock);
            if (rm_sig_from_queue(sig, current))
                recalc_sigpending(current);
            spin_unlock_irq(&current->sigmask_lock);
        }
    }
}
```

```

    }
    spin_unlock(&current->sig->siglock);
    return 0;
}

```

코드 96. do_sigaction()함수

넘겨 받는 값은 시그널의 번호와 k_sigaction구조체의 포인터, 원래의 k_sigaction구조체를 가지게 될 포인터이다. 시그널 번호가 올바르지 않거나, SIGKILL이나 SIGSTOP에 대한 핸들러를 설정하려고 한다면, 에러(-EINVAL)를 돌려준다.

현재 프로세스의 시그널 action배열을 검사해서 새로 설치하고자 하는 번호에 해당하는 핸들러를 찾아온 다음, lock을 설정한다(spin_lock()). 만약 원래(original)의 시그널 핸들러에 대한 k_sigaction구조체를 보관하고 싶다면, 이것을 넘겨받은 oact에 저장한다.

새로이 설정될 k_sigaction구조체 포인터가 있다면, 원래의 자리에 이 핸들러로 설정하고(*k=*act), 설정하고자 하는 k_sigaction의 시그널 매스크에서 SIGKILL과 SIGSTOP을 빼도록 한다(sigdelsetmask()). 만약 설정하고자 하는 시그널 핸들러가 SIG_IGN(ignore)로 설정된 경우이거나, 혹은, SIG_DFL이고 현재 설정하고자 하는 시그널의 번호가 SIGCONT나 SIGCHLD, SIGWINCH이 경우에는 아래와 같이 진행한다. 현재 프로세스의 시그널 mask에 lock을 걸어놓고, 설정하고자 하는 시그널이 이미 현재 프로세스의 pending된 시그널인지를 확인한다(rm_sig_from_queue()). 만약 pending된 시그널이라면, 이 시그널을 떼어내고, 현재 프로세스에 대해서 pending된 시그널을 새로 계산해준다(rm_sig_from_queue()). 마지막으로 설정된 lock들을 풀어서 복귀한다.

이상에서 우리는 간단하게 나마 시그널이 어떤 식으로 구현되어 있으며, 어떻게 시그널 핸들러를 설정, 시그널이 도착했을 때 취할 행동(action)을 바꾸는지를 알아보았다. 이 같은 시그널은 데이터를 전달하기보다는, 연산이나 시스템의 사건(event)을 프로세스에 알려주는데 효과적인 방법이다. 아래에서는 이젠 데이터를 상호 교환하는 방법과 공유 영역에 대한 접근을 제어하는 방법에 대해서 논한다.

2.18.2. Pipe

파이프란 Unix상에서 가장 고전적인 IPC방법으로, 한 프로세스의 output을 다른 프로세스의 input으로 보내는 방법이다. 이것과 유사한 방법으로는 나중에 살펴볼 FIFO(First In First Out)이란 방법도 있다. 파이프는 실제적인 파일 시스템에 이미지를 가지지 않는다는 점에서 FIFO와 다르다. 즉, FIFO는 실제로 파일 시스템 상에 이미지를 가진다.

파이프는 pipe()시스템 콜로 생성된다. 이는 다시 커널의 sys_pipe()를 호출하며, sys_pipe()함수는 아래와 같다. 코드는 ~/arch/i386/kernel/sys_i386.c에 있다.

```

asmlinkage int sys_pipe(unsigned long * fildes)
{
    int fd[2];
    int error;

    error = do_pipe(fd);
    if (!error) {
        if (copy_to_user(fildes, fd, 2*sizeof(int)))
            error = -EFAULT;
    }
    return error;
}

```

코드 97. sys_pipe() 시스템 콜

sys_pipe()는 두개의 파일 디스크립터를 가지고 do_pipe()함수를 호출한 후, 결과를 프로세스의 파일 디스크립터 필드에 복사한 후에 에러 값과 함께 복귀한다. 다시 do_pipe()함수를 보도록 하자. 이 함수는 ~/fs/pipe.c에 정의되어 있다.

```

int do_pipe(int *fd)
{
    struct qstr this;
    char name[32];
    struct dentry *dentry;
    struct inode * inode;
    struct file *f1, *f2;
    int error;
    int i,j;

    error = -ENFILE;
    f1 = get_empty_file();
    if (!f1)
        goto no_files;
    f2 = get_empty_file();
    if (!f2)
        goto close_f1;
    inode = get_pipe_inode();
    if (!inode)
        goto close_f12;
    error = get_unused_fd();
    if (error < 0)
        goto close_f12_inode;
    i = error;
    error = get_unused_fd();
    if (error < 0)
        goto close_f12_inode_i;
    j = error;
}

```

코드 98. do_pipe() 함수의 정의 – 사용되지 않는 파일 객체 찾기와 파일 디스크립터 만들기

먼저 빈 파일 객체 2개를 찾는다(get_empty_file()). 그리고, 다시 파일을 위해서 빈 inode object를 하나 찾고(get_pipe_inode()), 복제할 파일 디스크립터(fd)를 두개를 만들 현재의 프로세스에서 찾는다(get_unused_fd()).

```

error = -ENOMEM;
sprintf(name, "[%lu]", inode->i_ino);
this.name = name;
this.len = strlen(name);
this.hash = inode->i_ino; /* will go */
dentry = d_alloc(pipe_mnt->mnt_sb->s_root, &this);
if (!dentry)
    goto close_f12_inode_i_j;
dentry->d_op = &pipefs_dentry_operations;
d_add(dentry, inode);
f1->f_vfsmnt = f2->f_vfsmnt = mntget(mntget(pipe_mnt));
f1->f_dentry = f2->f_dentry = dget(dentry);

```

코드 99. do_pipe() 함수(continued) – 파일 시스템에 PIPE 생성

할당받은 inode의 번호를 파일의 이름으로 하는 하나의 디렉토리 entry를 생성하고나서(d_alloc), 파일에 대한 디렉토리 연산(operation)을 새로 생성된 디렉토리 연산으로 만들어준다(pipefs_dentry_operations). 위의 과정이 끝나면, 디렉토리 해쉬(hash) 큐로 넣어준다(d_add()). 파일 object f1과 f2의 vfsmnt와 dentry필드를 pipe_mnt와 새로 할당한 디렉토리 entry로 만들어준다. pipe_mnt는 마운트 포인트에 해당하는 것으로 struct vfsmount로 정의되어 있다. 나중에 VFS 파일 시스템을 보게 될 때 좀더 자세히 알아보도록 하겠다.

```

/* read file */
f1->f_pos = f2->f_pos = 0;
f1->f_flags = O_RDONLY;
f1->f_op = &read_pipe_fops;
f1->f_mode = 1;
f1->f_version = 0;

/* write file */
f2->f_flags = O_WRONLY;
f2->f_op = &write_pipe_fops;
f2->f_mode = 2;
f2->f_version = 0;

fd_install(i, f1);
fd_install(j, f2);
fd[0] = i;
fd[1] = j;
return 0;

```

코드 100. do_pipe()함수(continued) – 파일 객체의 초기화

이전 앞에서 할당 받은 각각의 파일 객체에 대한 초기화와 파일 연산을 정해주는 부분이다. 하나는 read로 다른 하나는 write로 각각 파일 연산을 초기화하고, read only 및 write only로 flag를 설정한다. 위의 과정이 끝나면, fd_install()함수를 호출해서 현재 프로세스의 이전에서 할당 받은 파일 디스크립터 자리에 파일 객체에 대한 포인터를 넣고, 파라미터 값으로 넘겨받은 파일 디스크립터의 0번에 read를 위한 file 디스크립터와 1번에 write를 위한 파일 디스크립터 index값을 넣고 복귀한다.

```

close_f12_inode_i_j:
    put_unused_fd(j);
close_f12_inode_i:
    put_unused_fd(i);
close_f12_inode:
    free_page((unsigned long) PIPE_BASE(*inode));
    kfree(inode->i_pipe);
    inode->i_pipe = NULL;
    iput(inode);
close_f12:
    put_filp(f2);
close_f1:
    put_filp(f1);
no_files:
    return error;
}

```

코드 101. do_pipe()함수(continued) – 복귀

나머지 코드는 예외 상황에 대한 처리로 그때까지 할당 받았던 커널의 자원을 놓아주고(release) 에러코드를 넘겨주는 것으로 시스템 콜을 마치게 된다.

프로세스는 이후에 파일 디스크립터에 대한 연산만을 하게 되므로, 커널의 내부에서는 do_pipe()에서 해준 파일 연산이나 디렉토리 연산에 대한 포인터를 가지고 프로세스의 요구를 만족 시켜주게 된다. 이와 같은 것을 위해서, pipefs_dentry_operations와 read_pipe_fops, write_pipe_fops가 사용된다. 각각은 아래와 같은 정의를 가진다.

```

static struct dentry_operations pipefs_dentry_operations = {
    d_delete: pipefs_delete_dentry,
};

```

```

struct file_operations read_pipe_fops = {
    llseek:          pipe_llseek,
    read:           pipe_read,
    write:          bad_pipe_w,
    poll:           pipe_poll,
    ioctl:          pipe_ioctl,
    open:           pipe_read_open,
    release:        pipe_read_release,
};

struct file_operations write_pipe_fops = {
    llseek:          pipe_llseek,
    read:           bad_pipe_r,
    write:          pipe_write,
    poll:           pipe_poll,
    ioctl:          pipe_ioctl,
    open:           pipe_write_open,
    release:        pipe_write_release,
};

```

코드 102. 파일에 대한 연산의 정의

정의 되지 않은 연산에 대해서는 VFS에서 내부적으로 처리하게 되며, read의 경우에는 bad_pipe_w()함수가, write경우에는 bad_pipe_r()함수가 각각 정의된다.

get_pipe_inode()함수에 대해서 좀더 살펴보기로 하자. 이 함수는 pipe를 위해서 사용할 inode를 돌려주는 함수로 아래와 같이 정의된다.

```

static struct inode * get_pipe_inode(void)
{
    struct inode *inode = get_empty_inode();

    if (!inode)
        goto fail_inode;
    if (!pipe_new(inode))
        goto fail_input;
    PIPE_READERS(*inode) = PIPE_WRITERS(*inode) = 1;
    inode->i_fop = &rdwr_pipe_fops;
    inode->i_sb = pipe_mnt->mnt_sb;
    inode->i_state = I_DIRTY;
    inode->i_mode = S_IFIFO | S_IRUSR | S_IWUSR;
    inode->i_uid = current->fsuid;
    inode->i_gid = current->fsgid;
    inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
    inode->i_blksize = PAGE_SIZE;
    return inode;

fail_input:
    iput(inode);
fail_inode:
    return NULL;
}

```

코드 103. get_pipe_inode()함수

get_pipe_inode()함수는 시스템에서 먼저 빈 inode를 하나를 얻는다. 그리고 나서 pipe_new()를 호출해서 inode의 i_pipe 및 파일과 관련된 자료구조를 초기화 한 후, inode에 대한 파일 연산을 초기화 한다 (rdwr_pipe_fops). inode에 대한 슈퍼블록 또한 pipe_mnt의 마운트 파일시스템의 슈퍼블록을 가리키도록

한다. 여기서 inode의 상태를 `I_DIRTY`로 두어서 나중에 `mark_inode_dirty()`함수가 이미 dirty list에 이 inode가 존재한다고 여기도록 만들어서 inode dirty list로 옮기는 것을 방지하도록 한다. 또한 pipe에 대한 연산의 최대크기는 `i_blksize`로 `PAGE_SIZE`(i386에서 4Kbytes)를 가리키도록 만든다. `rdwr_pipe_fops`연산은 아래와 같은 정의를 가진다.

```
struct file_operations rdwr_pipe_fops = {
    llseek:          pipe_llseek,
    read:           pipe_read,
    write:          pipe_write,
    poll:           pipe_poll,
    ioctl:          pipe_ioctl,
    open:           pipe_rdwr_open,
    release:        pipe_rdwr_release,
};
```

코드 104. 파일에 대한 파일 연산의 정의

`pipe_mnt`구조체는 파일에 대한 전역변수로서 파일 파일시스템을 등록할 때 초기화가 이루어진다. 이것은 `init_pipe_fs()` 함수가 하는 역할이다. 나중에 이 함수를 보기로 하자.

`pipe_new()`함수는 하나의 새로운 페이지를 커널의 메모리에서 할당 받고, 그것을 inode와 연결한 후, 각종 초기화 작업을 마친 후 돌려주는 역할을 하고 있다. 코드는 아래와 같다. `~/fs/pipe.c`를 참조하기 바란다.

```
struct inode* pipe_new(struct inode* inode)
{
    unsigned long page;

    page = __get_free_page(GFP_USER);
    if (!page)
        return NULL;
    inode->i_pipe = kmalloc(sizeof(struct pipe_inode_info), GFP_KERNEL);
    if (!inode->i_pipe)
        goto fail_page;
    init_waitqueue_head(PIPE_WAIT(*inode));
    PIPE_BASE(*inode) = (char*) page;
    PIPE_START(*inode) = PIPE_LEN(*inode) = 0;
    PIPE_READERS(*inode) = PIPE_WRITERS(*inode) = 0;
    PIPE_WAITING_READERS(*inode) = PIPE_WAITING_WRITERS(*inode) = 0;
    PIPE_RCOUNT(*inode) = PIPE_WCOUNT(*inode) = 1;
    return inode;
fail_page:
    free_page(page);
    return NULL;
}
```

코드 105. `pipe_new()`함수

즉, 커널로부터 새로운 사용자 프로세스를 위한 프로세스를 할당 받고나서(`__get_free_page()`), inode에 `pipe`을 나타내는 `i_pipe`구조체에 `pipe_inode_info`구조체를 할당해서 넣어준다. 또한 inode의 wait queue를 초기화하고(`init_waitqueue_head()`), inode에 대한 각종 초기화 작업을 담당한다. 이곳에서 사용되는 매크로들의 정의는 아래와 같다. `~/include/linux/pipe_i_fs.h`를 참고하기 바란다.

```
#define PIPE_SIZE      PAGE_SIZE
#define PIPE_SEM(inode)  (&(inode).i_sem)
#define PIPE_WAIT(inode) (&(inode).i_pipe->wait)
#define PIPE_BASE(inode) ((inode).i_pipe->base)
```

```
#define PIPE_START(inode) ((inode).i_pipe->start)
#define PIPE_LEN(inode) ((inode).i_size)
#define PIPE_READERS(inode) ((inode).i_pipe->readers)
#define PIPE_WRITERS(inode) ((inode).i_pipe->writers)
#define PIPE_WAITING_READERS(inode) ((inode).i_pipe->waiting_readers)
#define PIPE_WAITING_WRITERS(inode) ((inode).i_pipe->waiting_writers)
#define PIPE_RCOUNTER(inode) ((inode).i_pipe->r_counter)
#define PIPE_WCOUNTER(inode) ((inode).i_pipe->w_counter)
#define PIPE_EMPTY(inode) (PIPE_LEN(inode) == 0)
#define PIPE_FULL(inode) (PIPE_LEN(inode) == PIPE_SIZE)
#define PIPE_FREE(inode) (PIPE_SIZE - PIPE_LEN(inode))
#define PIPE_END(inode) ((PIPE_START(inode) + PIPE_LEN(inode)) & (PIPE_SIZE-1))
#define PIPE_MAX_RCHUNK(inode) (PIPE_SIZE - PIPE_START(inode))
#define PIPE_MAX_WCHUNK(inode) (PIPE_SIZE - PIPE_END(inode))
```

코드 106. PIPE에 대한 매크로 정의들

매크로에 대한 정의는 메모리상의 inode의 구조에 대해서 알고 있다면 자명할 것이다. 이상에서 설명하지 않은 pipe_inode_info 데이터 구조는 아래와 같은 정의를 가진다. ~/include/linux/pipe_fs_i.h를 참고하기 바란다.

```
struct pipe_inode_info {
    wait_queue_head_t wait;           /* pipe/FIFO의 wait queue*/
    char *base;                      /* 커널 버퍼의 주소*/
    unsigned int start;              /* 커널 버퍼에서의 read가 일어나는 자리*/
    unsigned int readers;             /* read를 하는 프로세스를 위한 flag 혹은 그 프로세스의 수*/
    unsigned int writers;             /* write를 하는 프로세스를 위한 flag 혹은 그 프로세스의 수*/
    unsigned int waiting_readers;     /* read를 기다리는 프로세스의 수*/
    unsigned int waiting_writers;     /* write를 기다리는 프로세스의 수*/
    unsigned int r_counter;           /* read를 한 횟수*/
    unsigned int w_counter;           /* write를 한 횟수*/
};
```

코드 107. PIPE를 가지는 inode info구조체의 정의

만약 pipe를 사용해서 프로세스간의 통신을 하게 될 때, 읽으려 하는 프로세스는 쓰는 프로세스가 없으며, wait 큐에서 대기 상태로 머물러 있게 될 것이다. 또한 커널의 버퍼를 통해서 데이터를 전달하기 때문에, 디스크 상에는 그 데이터의 이미지가 없을 것이다. inode역시 커널에서 할당 받은 inode object를 사용하고 있다.

파이프에 대한 초기화는 init_pipe_fs() 함수의 호출을 통한다. ~/fs/pipe.c를 참고하기 바란다. 이 함수는 커널의 초기화에서 호출되며, 파일 시스템에 pipe를 등록하는 역할을 한다.

```
static int __init init_pipe_fs(void)
{
    int err = register_filesystem(&pipe_fs_type);
    if (!err) {
        pipe_mnt = kern_mount(&pipe_fs_type);
        err = PTR_ERR(pipe_mnt);
        if (IS_ERR(pipe_mnt))
            unregister_filesystem(&pipe_fs_type);
        else
            err = 0;
    }
    return err;
}
```

코드 108. init_pipe_fs() 함수 – 파일의 등록

이 함수에서 중요한 부분은 역시 register_filesystem()과 kern_mount가 될 것이다. register_filesystem()함수는 파일 시스템 타입(type)의 하나로 커널에 등록시켜주는 역할을 하며, kern_mount()함수는 해당하는 파일 시스템으로부터 슈퍼블록을 읽고, 커널의 마운트된 파일 시스템의 정보를 갱신하는 역할을 한다.

파일 시스템의 삭제는 exit_pipe_fs() 함수를 통한다. ~/fs/pipe.c를 참고하기 바란다. 단순히 등록된 파일 시스템을 해제하고, 커널에서 umount를 호출하도록 한다.

```
static void __exit exit_pipe_fs(void)
{
    unregister_filesystem(&pipe_fs_type);
    kern_umount(pipe_mnt);
}
```

이상에서 파일에 대해서 살펴보았다. 이와는 달리 FIFO는 실제로 디스크 상에 이름을 가지는 특수한 형태의 파일이다. 이름을 가진다는 것은 즉, 이를 관리하기 위해서 디스크 상에 어떤 데이터 구조를 가진다는 말이다.

2.18.3. FIFO(First In First Out)

FIFO는 파일보다 더 많은 장점을 준다. PIPE가 사용상의 편의성과 간단함을 주는 반면에 이미 열려져 있는 pipe에 대해서 open할 수 없다는 점이다. 즉, 서로 관련성을 가지지 않는 프로세스간에는 데이터를 전송할 수 없다는 단점이 생긴다. 반면에 FIFO는 이러한 프로세스들도 데이터를 서로간에 주고 받을 수 있는 디스크 상의 inode구조를 제공하고 있다. 이와 같은 점에서 FIFO를 named PIPE라고 부르기도 한다. FIFO를 생성하는 방법은 일반적으로 디바이스 노드를 생성하는 방법인 mknod와 mkfifo가 있다. 사용법은 아래와 같다.

```
mkfifo fifo_pathname
mknod fifo_pathname p
```

코드 109. FIFO의 사용법

이와 같이 생성된 FIFO를 사용하는 방법은 아래와 같다.

```
#mknod FIFO p
#ls -al
...
prw-rw-r-- 1 shkwon shkwon 0 Feb 6 15:09 FIFO
#vi FIFO(여기서 프로세스가 blocking이 된다.)
```

(다른 쉘에서)

#ls -al > FIFO (이때 이전에 blocking이 된 쉘이 다시 가동되면서 ls -al의 결과를 보여줄 것이다.)

코드 110. FIFO 사용의 예

기본적으로 FIFO에 대한 연산이나 구조는 파일과 거의 동일한 구조를 지닌다. 이것은 물론 Linux에서의 구현상의 동일함이며, pipe_inode_info 자료구조체를 위에서 설명할 때 inode에 들어가는 정보가 많은 공통점을 지닌다는 것을 통해서 알 수 있다.

FIFO에 대한 시스템 콜 인터페이스는 mknod() 함수이다. 이것은 sys_mknod() 시스템 콜을 호출하며, 이 함수에서 다시 mode값이 S_IFIFO일 경우 vfs_mknod() 함수를 호출한다. vfs_mknod() 함수는 결국 파일 시스템에 의존적인 inode의 연산인 mknod() 함수를 호출해서 해당하는 inode를 디스크 상에 만들고, 이것을 FIFO라고 표시한다. 만약 파일 시스템으로 EXT2를 사용한다면 ext2_mknod() 함수가 호출될 것이며, 이 함수는 init_special_inode() 함수를 호출해서 inode 파일 연산 필드인 i_fop을 def_fifo_ops를

가르키도록 만들 것이다. 이후에 FIFO에 대한 연산은 이 `def_fifo_fops`에 정의된 연산을 사용할 것이다. `def_fifo_fops`는 아래와 같이 정의된다. `~/fs/fifo.c`를 참조하기 바란다.

```
struct file_operations def_fifo_fops = {
    open:           fifo_open,          /* will set read or write pipe_fops */
};
```

코드 111. FIFO에 대한 파일 연산의 정의

`def_fifo_fops`는 단지 `fifo`에 대한 `open`연산만을 지정하고 있다. `fifo_open()`함수를 보도록 하자. `~/fs/fifo.c`에 역시 정의되어 있다.

```
static int fifo_open(struct inode *inode, struct file *filp)
{
    int ret;

    ret = -ERESTARTSYS;
    lock_kernel();
    if (down_interruptible(PIPE_SEM(*inode)))
        goto err_nolock_nocleanup;
    if (!inode->i_pipe) {
        ret = -ENOMEM;
        if (!pipe_new(inode))
            goto err_nocleanup;
    }
    filp->f_version = 0;
```

코드 112. `fifo_open()` 함수 – FIFO를 위한 메모리의 할당 및 `inode`의 초기화

`fifo_open()`함수는 FIFO에 대한 읽기에서 `inode`를 읽으려고 할 때 호출된다. 먼저 kernel에 대한 `lock`을 구한다음, 새로이 커널로부터 메모리를 할당 받고 관련된 `inode`를 초기화한다(`pipe_new()`). 초기화가 끝나면 FIFO를 나타내는 파일 객체의 버전을 0으로 초기화 한다.

```
switch (filp->f_mode) {
    case 1:
        filp->f_op = &read_fifo_fops;
        PIPE_RCOUNT(*inode)++;
        if (PIPE_READERS(*inode)++ == 0)
            wake_up_partner(inode);
        if (!PIPE_WRITERS(*inode)) {
            if ((filp->f_flags & O_NONBLOCK)) {
                /* suppress POLLHUP until we have
                 * seen a writer */
                filp->f_version = PIPE_WCOUNTER(*inode);
            } else
            {
                wait_for_partner(inode, &PIPE_WCOUNTER(*inode));
                if(signal_pending(current))
                    goto err_rd;
            }
        }
    break;
```

코드 113. `fifo_open()` 함수(continued) – 읽기 전용을 위한 초기화

만약 FIFO와 관련된 파일 객체의 모드가 O_RDONLY일 경우에는 읽기 연산만 가능한 것으로, 파일 연산을 read_fifo_fops로 설정하고, pipe_inode_info의 r_counter필드를 증가시킨다(PIPE_RCOUNT(*inode)++). 만약 FIFO를 읽는 프로세스가 없다면 wake_up_partner()를 호출해서 inode상에서 기다리고 있을 프로세스들을 깨우도록 한다. FIFO를 쓰는 프로세스가 없다면(PIPE_WRITERS(*inode)), NONBLOCK으로 f_flag가 설정되었을 경우에는 파일 객체의 f_version필드를 inode의 w_counter로 두고, 그렇지 않을 경우에는 쓰기가 있을 때까지 기다리게 된다(wait_for_partner()). 이 상태에서 inode에서 프로세스는 잠들게 된다.

```
case 2:
    ret = -ENXIO;
    if ((filp->f_flags & O_NONBLOCK) && !PIPE_READERS(*inode))
        goto err;
    filp->f_op = &write_fifo_fops;
    PIPE_WCOUNT(*inode)++;
    if (!PIPE_WRITERS(*inode))
        wake_up_partner(inode);
    if (!PIPE_READERS(*inode)) {
        wait_for_partner(inode, &PIPE_RCOUNT(*inode));
        if (signal_pending(current))
            goto err_wr;
    }
    break;
```

코드 114. fifo_open()함수(continued) – FIFO의 쓰기 전용을 위한 초기화

만약 FIFO와 관련된 파일 객체의 모드가 O_WRONLY일 경우에는 쓰기만 가능한 것으로, 파일 객체의 f_flag필드가 O_NONBLOCK이고, 읽는 프로세스가 없다면 에러를, 그렇지 않다면 write_fifo_fops를 파일 연산으로 설정하고, inode의 w_counter를 증가시킨다(PIPE_WCOUNT(*inode)++). 만약 write하고 있는 프로세스가 없다면, 읽기를 기다리고 있는 프로세스들을 깨우게 된다(wake_up_partner()). 만약 FIFO에 대해서 read하고 있는 프로세스가 없다면, read할 프로세스를 기다리기 위해서 잠든다(wait_for_partner()). 이 경우 다시 inode에서 프로세스는 잠들게 된다.

```
case 3:
    filp->f_op = &rdwr_fifo_fops;
    PIPE_READERS(*inode)++;
    PIPE_WRITERS(*inode)++;
    PIPE_RCOUNT(*inode)++;
    PIPE_WCOUNT(*inode)++;
    if (PIPE_READERS(*inode) == 1 || PIPE_WRITERS(*inode) == 1)
        wake_up_partner(inode);
    break;
default:
    ret = -EINVAL;
    goto err;
}
/* Ok! */
up(PIPE_SEM(*inode));
unlock_kernel();
return 0;
```

코드 115. fifo_open()함수(continued) – FIFO에 대한 읽기 쓰기의 초기화

만약 FIFO와 관련된 파일 객체의 모드가 O_RDWR일 경우에는 읽기와 쓰기가 가능한 것으로, 파일 객체의 파일 연산을 rdwr_fifo_fops로 초기화하고, inode의 readers와 writers, r_counter와 w_counter를 하나씩 증가시켜준다. 만약 현재의 inode의 readers와 writer필드값이 10이라면, inode상에서 기다리고 있던

프로세스들을 깨워준다(wake_up_partner()). 이외의 경우에는 전부 에러(-EINVAL)로 처리하게 되며, 앞에서 사용했던 세마포어와 커널에 대한 lock을 풀어주고 복귀한다.

```

err_rd:
    if (!--PIPE_READERS(*inode))
        wake_up_interruptible(PIPE_WAIT(*inode));
    ret = -ERESTARTSYS;
    goto err;
err_wr:
    if (!--PIPE_WRITERS(*inode))
        wake_up_interruptible(PIPE_WAIT(*inode));
    ret = -ERESTARTSYS;
    goto err;
err:
    if (!PIPE_READERS(*inode) && !PIPE_WRITERS(*inode)) {
        struct pipe_inode_info *info = inode->i_pipe;
        inode->i_pipe = NULL;
        free_page((unsigned long)info->base);
        kfree(info);
    }
err_nocleanup:
    up(PIPE_SEM(*inode));
err_nolock_nocleanup:
    unlock_kernel();
    return ret;
}

```

코드 116. fifo_open()함수(continued) – 종료

이 부분은 이전 상황에서 에러가 있을 경우에 처리를 맡는 부분이다. read나 write를 위해서 FIFO를 open한 경우에 대해서 inode상에서 잠들어 있을지도 모를 프로세스들을 깨우고(wake_up_interruptible()), 에러 값으로 -ERESTARTSYS를 돌려준다. 또한 read나 write하는 프로세스가 없는 경우에 대해서는 할당 받은 inode의 i_pipe필드를 NULL로 만든 후, 커널에서 할당 받은 메모리를 놓아준다(free_page()). 마지막으로 FIFO를 생성하는데 사용했던, pipe_inode_info 구조체를 놓아준다(kfree()). 복귀하기 전에 inode의 세마포어 값과 lock된 커널을 unlock(unlock_kernel())시켜주며, 에러코드를 돌려준다.

```

static void wake_up_partner(struct inode* inode)
{
    wake_up_interruptible(PIPE_WAIT(*inode));
}

```

코드 117. wake_up_partner()함수

읽기나 쓰기를 위해서 대기하고 있는 프로세스들은 wake_up_partner()함수를 사용하는데, 이는 inode상에서 잠들어 있는 모든 프로세스를 다깨운다. 따라서, 읽기든 쓰기든 상관없이 프로세스들은 깨어나게 된다.

```

static void wait_for_partner(struct inode* inode, unsigned int* cnt)
{
    int cur = *cnt;
    while(cur == *cnt) {
        pipe_wait(inode);
        if(signal_pending(current))
            break;
    }
}

```

코드 118. wait_for_partner()함수

wait_for_partner()함수는 현재의 프로세스가 시그널을 받던가 아니면, inode의 r_counter나 w_counter가 변화가 없을 때까지 계속적으로 loop를 돌면서 pipe_wait()함수를 호출해서 기다리게 된다.

FIFO와 관련된 연산으로는 이전에 fifo_open()함수에서 잠시 보았던 read_fifo_fops, write_fifo_fops, rdwr_fifo_fops가 있으며, 아래와 같은 정의를 가진다. ~/fs/pipe.c를 참고하기 바란다.

```
struct file_operations read_fifo_fops = {
    llseek:          pipe_lseek,
    read:            pipe_read,
    write:           bad_pipe_w,
    poll:            fifo_poll,
    ioctl:           pipe_ioctl,
    open:            pipe_read_open,
    release:         pipe_read_release,
};

struct file_operations write_fifo_fops = {
    llseek:          pipe_lseek,
    read:            bad_pipe_r,
    write:           pipe_write,
    poll:            fifo_poll,
    ioctl:           pipe_ioctl,
    open:            pipe_write_open,
    release:         pipe_write_release,
};

struct file_operations rdwr_fifo_fops = {
    llseek:          pipe_lseek,
    read:            pipe_read,
    write:           pipe_write,
    poll:            fifo_poll,
    ioctl:           pipe_ioctl,
    open:            pipe_rdwr_open,
    release:         pipe_rdwr_release,
};
...
#define fifo_poll pipe_poll
```

코드 119. FIFO에 대한 파일 연산의 정의

파이프의 파일 연산자의 정의에서 보듯이, 이는 파이프와 같은 연산을 사용하고 있다는 것을 알 수 있다.

이상에서 우린 기본적인 IPC방법인 시그널과 파이프, 그리고, FIFO에 대해서 보았다. 이하에서는 시스템 V계열에서 사용하는 IPC방법인 세마포어와 메시지 큐, 그리고, 공유 메모리에 대해서 알아보기로 한다.

2.18.4. Semaphore

시스템 V계열의 IPC중에서 가장 먼저 볼 것은 세마포어이다. 주로 세마포어는 동기화를 위한 방법으로 제공되는 것으로 프로세스간에 중요한 데이터를 접근하는데 있어서, 서로간에 접근하는 순서를 정하는 방법이다. 예를 들어서 연결 리스트에 대한 자료구조를 접근하고자 할 때, 임의의 프로세스 2개가 동시에 접근하는 경우를 생각하면 될 것이다. 이 경우 하나의 프로세스가 자료구조를 고치는 도중에 스케줄링 아웃(out)이 되어 있다면, 다른 프로세스가 실행될 기회를 얻게 되며, 같은 자료구조를 고칠 가능성을 가질 수 있다. 이 경우 다시 원래의 프로세스가 실행될 기회를 얻게 되어 재 시작하게 되면, 자료구조는 깨질 것이다.

가장 간단한 방법으로는 이진(binary) 세마포어를 사용하는 것이다. 즉, 0와 1의 값을 사용해서 하나의 변수에 저장하는 것이다. 접근하는 프로세스는 1일 경우에만 접근할 수 있으며, 그렇지 않을 경우에는 대기(wait)상태로 기다려야 한다. 만약 접근할 수 있다면 변수를 0으로 만들어주고, 자신이 원하는 연산을 수행한 후에, 다시 이 변수를 1로 만들어 주어야 할 것이다. 이때 중요한 것은 변수에 대한 테스트와 증가, 혹은 감소의 연산이 다른 커널의 활동에 대해서 보호되어야 한다는 것이다. 즉, 한번에 이 모든 연산이 끝나야 한다(atomic). 이것은 간단히 세마포어를 설명하는 예이며, 리눅스에서는 이진 세마포어만을 지원하는 것이 아니라 더 많은 수의 세마포어를 지원한다.

세마포어를 위해서 커널에서 정의하는 데이터 구조는 아래와 같다. 각각의 필드별로 하는 역할을 설명해 준다.

```
/* 이전 버전과의 호환성 때문에 있는 semid_ds의 정의 */
struct semid_ds {
    struct ipc_perm    sem_perm;           /* 세마포어의 허가모드 */
    __kernel_time_t   sem_otime;          /* 마지막 semop()연산의 시간 */
    __kernel_time_t   sem_ctime;          /* 마지막으로 변화가 있었던 시간 */
    struct sem        *sem_base;          /* 세마포어를 관리하는 array에서 첫번째 세마포어에
    대한 주소 */
    struct sem_queue *sem_pending;        /* 아직 세마포어에 대해서 처리되지 않은 연산들 */
    struct sem_queue **sem_pending_last; /* 마지막에 있는 처리되지 않은 연산 */
    struct sem_undo   *undo;              /* 세마포어 배열에 대한 undo요청들에 대한 포인터 */
    unsigned short    sem_nsems;          /* 세마포어 배열에 있는 세마포어들의 수 */
};
```

코드 120. semid_ds구조체의 정의

현재 이것은 낡은 것으로 이전 버전과의 호환성을 위해서 존재하며, 아래와 같이 sem_array라는 구조체로 다시 정의 된다. 물론 각각의 필드들이 명시하는 것은 동일하다고 보면 될 것이다.

```
struct sem_array {
    struct kern_ipc_perm    sem_perm;
    time_t                  sem_otime;
    time_t                  sem_ctime;
    struct sem               *sem_base;
    struct sem_queue         *sem_pending;
    struct sem_queue         **sem_pending_last;
    struct sem_undo          *undo;
    unsigned long            sem_nsems;
};
```

코드 121. sem_array구조체의 정의

커널은 세마포어의 object들을 세마포어의 배열로 나타내며, 배열 내에 있는 세마포어들은 sem구조체로 나타낸다. sem 구조체는 아래와 같이 정의 된다.

```
struct sem {
    int      semval;           /* 현재의 세마포어 값 */
    int      sempid;          /* 마지막 연산을 가한 프로세스의 ID */
};
```

코드 122. sem구조체의 정의

이것은 각각의 배열에서 세마포어가 가지는 값과 마지막 semop연산을 가한 프로세스의 ID를 보관하는 자료 구조이다. 그리고, 만약 세마포어에 대한 연산을 가했을 경우에 문제가 될 수 있는 dead lock에서 회복하기위한 구조로서 sem_undo구조체를 두고 있다.

```
struct sem_undo {
    struct sem_undo * proc_next;          /* next entry on this process */
    struct sem_undo * id_next;            /* next entry on this semaphore set */
    int               semid;              /* semaphore set identifier */
    short *           semadj;             /* array of adjustments, one per semaphore */
};
```

코드 123. sem_undo구조체의 정의

만약 하나의 프로세스가 세마포어 연산을 한 이후에 종료했다면, 이 세마포어는 결코 양의 값을 가질 수 없게 된다. 이 경우 다른 프로세스들은 세마포어가 놓여지기를 기다리면서 계속 수행을 멈추는 상황이 되며, 회복이 되지 않는다. 이를 위해서 사용하는 것이 바로 sem_undo 자료 구조이다.

세마포어의 초기화는 ~/init/main.c의 start_kernel()함수에서 ipc_init()함수를 호출하면서 초기화 된다. 이곳에서는 sem_init(), msg_init(), shm_init()함수를 각각 호출하며, 다시 ~/ipc/semc의 sem_init()함수는 사용된 세마포어의 개수를 0으로 만들고, ipc_init_ids()함수를 호출한다. 넘겨주는 파라미터로는 ipc_ids구조체로 정의된 sem_ids와 생성할 세마포어 ID의 개수인 SEMMNI(128)을 준다. ipc_init_ids() 함수에서는 전체 시스템에서 사용할 IPC를 위한 object들의 상한선을 가지고 있으며, 이를 넘겨서 생성할 수는 없다. 함수가 에러 없이 복귀하게 되면 sem_ids는 세마포어들을 보관할 수 있는 메모리를 할당 받을 수 있게 된다.

ipc_ids구조체는 아래와 같이 정의 된다. 각각의 필드들이 하는 역할을 보도록 하자.

```
struct ipc_id {
    struct kern_ipc_perm* p;
};

...
struct ipc_ids {
    int size;                      /*ipc_ids의 크기*/
    int in_use;                    /*사용 카운트*/
    int max_id;                   /*최대 ID*/
    unsigned short seq;           /*kern_ipc_perm의 seq값*/
    unsigned short seq_max;        /*위의 필드가 가질 수 있는 최대값(unsigned short)*/
    struct semaphore sem;         /*semaphore 구조체*/
    spinlock_t ary;               /*ipc_ids구조체에 대한 lock flag*/
    struct ipc_id* entries;       /*ipc_id들의 배열에 대한 포인터*/
};
```

코드 124. ipc_id 및 ipc_ids구조체의 정의

위에서 entries필드는 ipc_init_ids()함수를 호출할 때 배열로 할당된다. 한가지 덧붙여서 보아야 할 것은 시스템에 의존적인 구조인 semaphore구조체이다. i386의 경우에는 ~/i386semaphore.h에 정의되어 있으며, 아래와 같은 필드를 가진다.

```
struct semaphore {
    atomic_t count;
    int sleepers;
    wait_queue_head_t wait;
#ifndef WAITQUEUE_DEBUG
    long __magic;
#endif
```

```
};
```

코드 125. 하위의 시스템 semaphore의 구조체의 정의

이 구조체는 실제로 세마포어에 대한 연산을 적용할 때, 시스템에서 사용하는 세마포어이다. 기다리는 프로세스들을 위한 wait queue(wait)와 세마포어 카운트(count) 값, 현재 sleep된 상태로 있는 프로세스들의 수를 나타내는 sleepers필드로 구성된다(즉, 실제로 세마포어 연산을 하고자 할 때, 프로세스들이 기다리기 위한 큐와 세마포어 (이진: 0혹은 1), 기다리는 프로세스들의 수를 가진다.). 시스템에 있는 세마포어에 대한 초기화도 이전에 설명한 ipc_init_ids()함수에서 ~/include/arch/semaphore.h 에 있는 sema_init()를 호출에서 행해진다. 간단히 count과 sleepers를 0으로 놓고, wait queue를 초기화 한다

ipc_init_ids()함수는 나중에 메시지 큐와 공유 메모리의 초기화에도 나오게 되므로 이곳에서 살펴보기로 하자.

```
void __init ipc_init_ids(struct ipc_ids* ids, int size)
{
    int i;
    sema_init(&ids->sem,1);

    if(size > IPCMNI)
        size = IPCMNI;
    ids->size = size;
    ids->in_use = 0;
    ids->max_id = -1;
    ids->seq = 0;
    {
        int seq_limit = INT_MAX/SEQ_MULTIPLIER;
        if(seq_limit > USHRT_MAX)
            ids->seq_max = USHRT_MAX;
        else
            ids->seq_max = seq_limit;
    }

    ids->entries = ipc_alloc(sizeof(struct ipc_id)*size);

    if(ids->entries == NULL) {
        printk(KERN_ERR "ipc_init_ids() failed, ipc service disabled.\n");
        ids->size = 0;
    }
    ids->ary = SPIN_LOCK_UNLOCKED;
    for(i=0;i<size;i++)
        ids->entries[i].p = NULL;
}
```

코드 126. ipc_init_ids()함수

ipc_init_ids()함수는 ipc_ids구조체의 포인터와 얼마 만큼의 개수를 가질 것인가를 넘겨받는다. 해당 ipc_ids구조체의 세마포어를 초기화하고, 크기의 한계를 정한다(IPCMNI=32768). 또한 ipc_ids의 in_use필드와 max_id, seq필드를 초기화 하고, 가질 수 있는 sequence의 최대값도 정한다. 이와 같은 것이 끝나면 이젠 해당하는 ipc_id의 구조체를 할당하는 것이다. 할당 받은 ipc_id구조체들은 ipc_ids의 entries필드를 차지하게 된다. 즉, 하나의 배열구조를 이루게 된다. 마지막으로 할당 받은 ipc_id구조체의 접근허가 모드를 가리키는 포인터를 NULL로 설정한다.

세마포어에 대한 시스템 콜 인터페이스 들은 아래와 같이 정의된다.

```
asmlinkage long sys_semget (key_t key, int nsems, int semflg);
```

```
asmlinkage long sys_semop (int semid, struct sembuf *sops, unsigned nsops);
asmlinkage long sys_semctl (int semid, int semnum, int cmd, union semun arg);
```

코드 127. 세마포어의 시스템 콜 인터페이스

세마포어를 새로이 생성하는 것은 sys_semget() 시스템 콜이다. 코드는 ~/ipc/sem.c에 있으며, 아래와 같다.

```
asmlinkage long sys_semget (key_t key, int nsems, int semflg)
{
    int id, err = -EINVAL;
    struct sem_array *sma;

    if (nsems < 0 || nsems > sc_semmsl)
        return -EINVAL;
    down(&sem_ids.sem);

    if (key == IPC_PRIVATE) {
        err = newary(key, nsems, semflg);
    } else if ((id = ipc_findkey(&sem_ids, key)) == -1) { /* key not used */
        if (!(semflg & IPC_CREAT))
            err = -ENOENT;
        else
            err = newary(key, nsems, semflg);
    } else if (semflg & IPC_CREAT && semflg & IPC_EXCL) {
        err = -EEXIST;
    } else {
        sma = sem_lock(id);
        if(sma==NULL)
            BUG();
        if (nsems > sma->sem_nsems)
            err = -EINVAL;
        else if (ipcperms(&sma->sem_perm, semflg))
            err = -EACCES;
        else
            err = sem_buildid(id, sma->sem_perm.seq);
        sem_unlock(id);
    }

    up(&sem_ids.sem);
    return err;
}
```

코드 128. sys_semget() 시스템 콜

넘겨 받는 파라미터 값으로는 key값과 세마포어의 수, 그리고, 세마포어의 flag를 받아들인다. 먼저 원하는 세마포어의 개수를 확인한 다음, 전역 변수인 sem_ids에 대해서 semaphore값을 down(decrement)시킨다. 만약 IPC_PRIVATE로 생성하려고 한다면 새로운 semaphore 배열(newary())을 생성하고, 그렇지 않다면 key를 가지고 해당하는 세마포어 ID를 찾는다(ipc_findkey()). key를 가지고 해당하는 세마포어 ID를 찾을 수 없다면, 생성 요구일 경우에는 새로이 세마포어 배열을 생성하고, 그렇지 않다면 에러(ENOENT)를 돌려준다. 만약 찾았다면, 요구가 배타적으로 생성을 요구한다면, 다시 에러를 돌려주고(EEXIST), 그렇지 않다면 원래 있던 세마포어 ID를 가지고 연산을 행한다. sem_lock()으로 먼저 ID를 가지고 세마포어 배열을 가지고 와서 세마포어의 개수 및 허가모드(ipcperms())를 확인한 후(EACCES), sequence 번호를 생성한다.(sem_buildid()). 모든 것이 제대로 되었다면, 위에서 down()시킨 semaphore를 다시 up(increment)시킨 후에 복귀한다. 이 함수에서 주의해서 보아야 하는 부분은 새로운 세마포어 배열의 생성 부분인데(newary()), 이것을 좀더 자세히 보도록 하자. 코드는 ~/ipc/util.c에 있다.

```
static int newary (key_t key, int nsems, int semflg)
```

```

{
    int id;
    struct sem_array *sma;
    int size;

    if (!nsems)
        return -EINVAL;
    if (used_sems + nsems > sc_semmns)
        return -ENOSPC;
    size = sizeof (*sma) + nsems * sizeof (struct sem);
    sma = (struct sem_array *) ipc_alloc(size);
    if (!sma) {
        return -ENOMEM;
    }
    memset (sma, 0, size);
    id = ipc_addid(&sem_ids, &sma->sem_perm, sc_semmni);
    if(id == -1) {
        ipc_free(sma, size);
        return -ENOSPC;
    }
    used_sems += nsems;
    sma->sem_perm.mode = (semflg & S_IRWXUGO);
    sma->sem_perm.key = key;
    sma->sem_base = (struct sem *) &sma[1];
    /* sma->sem_pending = NULL; */
    sma->sem_pending_last = &sma->sem_pending;
    /* sma->undo = NULL; */
    sma->sem_nsems = nsems;
    sma->sem_ctime = CURRENT_TIME;
    sem_unlock(id);
    return sem_buildid(id, sma->sem_perm.seq);
}

```

코드 129. newary() 함수

newary()가 하는 역할은 넘겨받은 key값과 세마포어의 개수, 그리고, flag값으로 새로운 세마포어 배열을 생성하는 것이다. 세마포어에 대해서 사용되는 상수 값에 대해서 간략히 알아보면, 아래와 같다.

#define SEMMNI 128	/* <= IPCMNI : 최대 세마포어 ID의 수 */
#define SEMMSL 250	/* <= 8 000 : ID당 최대 세마포어의 수 */
#define SEMMNS (SEMMNI*SEMMSL)	/* <= INT_MAX : 시스템의 최대 세마포어의 수 */
#define SEMOPM 32	/* <= 1 000 : semop시스템 콜당 최대 연산(operation)의 수 */
#define SEMVMX 32767	/* <= 32767 : 세마포어의 최대 값 */

코드 130. 세마포어에서 사용되는 상수 값

따라서, 먼저 시스템에서 사용하는 최대 세마포어의 수와 요청된 세마포어의 수를 더해서 시스템에서 허락하는 최대 세마포어의 수와 비교한다. 값을 넘어서면 에러(ENOSPC)를 돌려주고, 그렇지 않다면 세마포어 배열에의 크기와 struct sem의 크기를 가지는 메모리를 할당해서(ipc_alloc())이를 초기화(0으로 만듬) 시킨다. 할당된 세마포어 배열의 헤더(sem_perm)을 전역변수 sem_ids에 덧붙이고(ipc_addid()), 사용된 세마포어의 수에 할당된 만큼의 세마포어를 더한다(used_sems+=nsems). 나머지는 할당된 세마포어 배열에 대한 초기화이다. 마지막으로 생성된 세마포어의 id값을 계산해서 넘겨주고 복귀한다. 다시 이 함수에서 조금만 더 살펴본다면 ipc_addid() 부분이다. 이것은 세마포어 ID에 사용하고자 하는 세마포어 배열을 연관짓는 부분이기 때문에 중요하다. 코드는 ~/ipc/util.c에 있다.

```
int ipc_addid(struct ipc_ids* ids, struct kern_ipc_perm* new, int size)
```

```
{
    int id;

    size = grow_ary(ids, size);
    for (id = 0; id < size; id++) {
        if(ids->entries[id].p == NULL)
            goto found;
    }
    return -1;
found:
    ids->in_use++;
    if (id > ids->max_id)
        ids->max_id = id;
    new->cuid = new->uid = current->euid;
    new->gid = new->cgid = current->egid;
    new->seq = ids->seq++;
    if(ids->seq > ids->seq_max)
        ids->seq = 0;
    spin_lock(&ids->ary);
    ids->entries[id].p = new;
    return id;
}
```

코드 131. ipc_addid() 함수

ipc_addid()는 새로운 ID를 ID의 배열에 더하는 역할을 한다. 이 경우 배열의 크기가 작을 때는 크기를 늘려주고(grow_ary()), 빈 ID를 찾는다. 그리고나서 ID 구조체에 대한 초기화 작업을 하게 되며, lock을 설정한채로 ID를 돌려주고 복귀한다. grow_ary() 함수는 단순히 ID의 배열을 새로이 할당하고, 이전에 있던 ID의 배열을 복사한 후, 새로이 할당된 ID의 배열에 대해서 NULL로 초기화해서 돌려준다.

이상에서 전체적인 새로운 세마포어의 할당에 대해서 알아보았다. 이렇게 해서 할당된 세마포어의 전체적인 구조는 [그림15]와 같다. 물론 [그림15]에서는 여러 번의 세마포어 연산이 있었던 후의 구조를 보여주는 것이다.

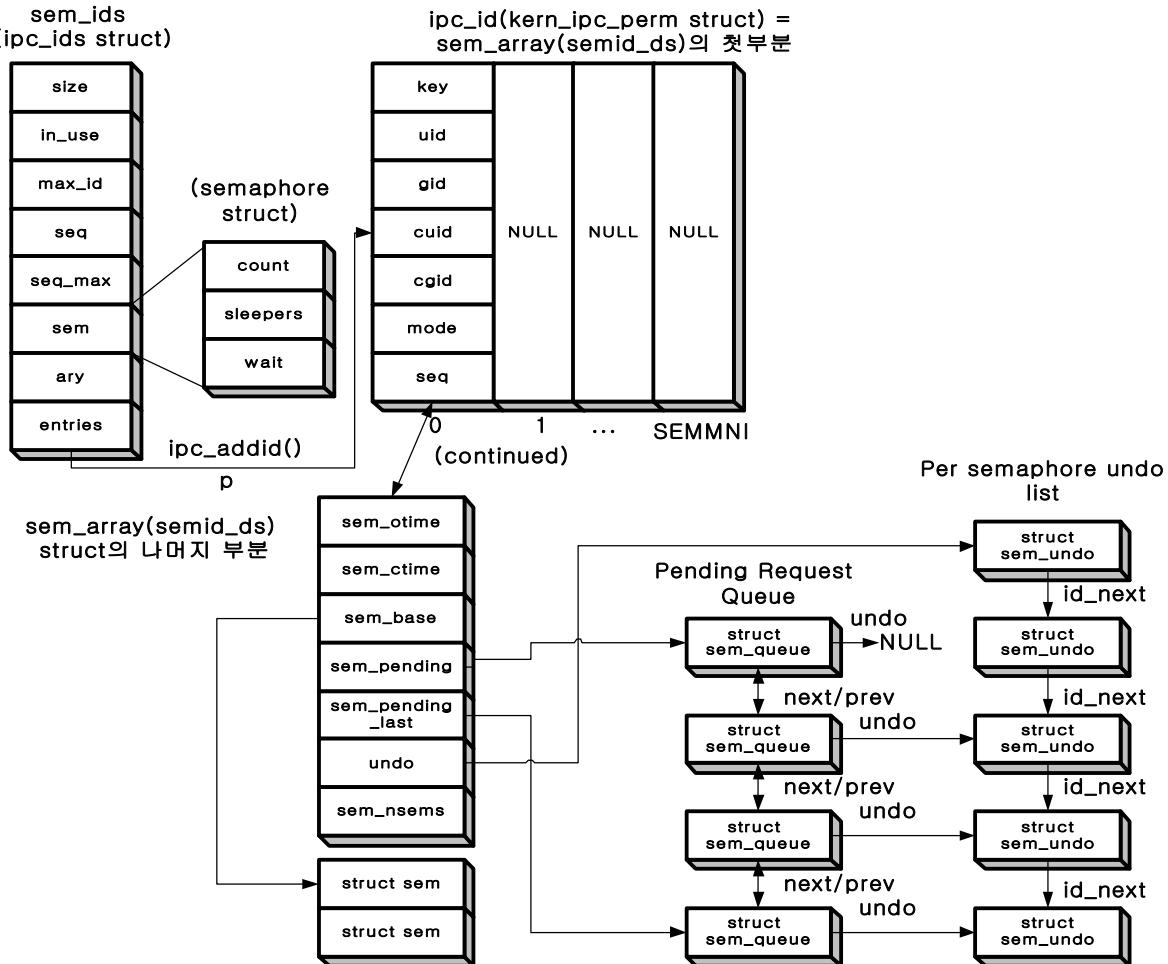


그림 18. 세마포어의 커널내 구조

[그림15]에서 세마포어의 초기화에서 할당되는 부분은 단지 ipc_id에 대한 부분만이 배열로서 할당된다. 나중에 sys_semget()이라는 호출을 통해서 이곳에 세마포어 배열이 할당되며, 사용될 때는 이렇게 할당된 구조를 sem_lock()에서 형 변환(type casting)을 해서 세마포어 배열로 사용한다.

세마포어들은 연산(혹은 요청)을 행할 때, sem_queue라는 자료구조를 만들어서 사용한다. 아래와 같이 정의된다.

```
/* One queue for each sleeping process in the system. */
struct sem_queue {
    struct sem_queue * next;      /* 세마포어 큐의 다음을 가르키는 포인터 */
    struct sem_queue ** prev;     /* 세마포어 큐의 이전을 가르키는 포인터 */
    struct task_struct* sleeper;  /* 세마포어에 대해서 잠들어있는 프로세스에 대한 포인터 */
    struct sem_undo * undo;       /* undo 구조체 포인터 */
    int pid;                     /* 세마포어를 요청하고 있는 프로세스의 ID */
    int status;                  /* 세마포어 연산의 종료상태 */
    struct sem_array * sma;       /* 세마포어 연산의 배열에 대한 포인터 */
    int id;                      /* 내부적인 세마포어의 ID */
    struct sembuf * sops;        /* 아직 적용되지 않은 세마포어 연산의 배열에 대한
포인터 */
    int nsops;                   /* 연산의 갯수 */
    int alter;                   /* 세마포어를 변경(alter)할 연산임을 표시 */
}
```

```
};
```

코드 132. sem_queue구조체의 정의

자료구조에서 next는 연결리스트의 다음을 가리키며, prev는 이전을 가리킨다. sleeper는 현재 wait queue에서 대기중인 프로세스를 나타내며, undo는 sem_undo구조체를 가리킨다. pid는 이 세마포어의 요청과 관련된 프로세스의 ID를 status는 연산의 결과로 가지게 되는 상태이며, sma는 세마포어 배열에 대한 포인터를, sops와 nsops는 아직 처리되지 못한 세마포어 연산의 포인터와 개수를, 그리고, alter는 세마포어를 변경할 것이라 flag값을 나타낸다.

세마포어에 대한 연산은 sys_semop() 시스템 콜을 통한다. 이것을 좀더 자세히 보도록 하자. 코드는 ~/ipc/sem.c를 참조하기 바란다.

```
asmlinkage long sys_semop (int semid, struct sembuf *tsops, unsigned nsops)
{
    int error = -EINVAL;
    struct sem_array *sma;
    struct sembuf fast_sops[SEMOPM_FAST];
    struct sembuf* sops = fast_sops, *sop;
    struct sem_undo *un;
    int undos = 0, decrease = 0, alter = 0;
    struct sem_queue queue;

    if (nsops < 1 || semid < 0)
        return -EINVAL;
    if (nsops > sc_semopm)
        return -E2BIG;
    if(nsops > SEMOPM_FAST) {
        sops = kmalloc(sizeof(*sops)*nsops,GFP_KERNEL);
        if(sops==NULL)
            return -ENOMEM;
    }
    if (copy_from_user (sops, tsops, nsops * sizeof(*tsops))) {
        error=-EFAULT;
        goto out_free;
    }
    sma = sem_lock(semid);
    error=-EINVAL;
    if(sma==NULL)
        goto out_free;
    error = -EIDRM;
    if (sem_checkid(sma,semid))
        goto out_unlock_free;
    error = -EFBIG;
```

코드 133. sys_semop()시스템 콜

세마포어 ID와 세마포어 버퍼, 그리고 세마포어 연산의 개수를 넘겨받는다. 먼저 넘겨받은 세마포어 연산의 수와 세마포어 ID의 값이 올바른 값을 가지는지 확인한다. 넘겨받는 세마포어 버퍼 구조체에 대한 보관을 위해서 메모리 공간을 할당하고, 그곳에 복사한다. 그리고 나서 세마포어 ID값으로 현재 사용하는 세마포어가 있는지를 확인하고(sem_lock()), 또한 세마포어의 sequence ID값도 확인한다(sem_checkid()). 세마포어 버퍼란 세마포어 연산을 위한 것으로 아래와 같은 구조를 가진다. (~include/linux/sem.h를 참조하라.)

```
/* semop system calls takes an array of these.*/
struct sembuf {
```

```

unsigned short sem_num; /* 배열에 있는 세마포어에 대한 인덱스 */
short      sem_op;      /* 세마포어 연산값 */
short      sem_flg;     /* 세마포어 연산의 flag값 */
};

```

코드 134. sembuf구조체의 정의

세마포어 배열에 대한 index값과 연산의 종류, 그리고 연산 flag로 구성된다. 세마포어 연산의 flag값으로 올 수 있는 것은 IPC_NOWAIT와 SEM_UNDO정도가 있다. IPC_WAIT는 세마포어 연산을 할 때, 기다리게 되면 에러를 돌려주고 복귀하라는 것이며, SEM_UNDO는 프로세스의 종료 시에 세마포어에 대해서 행한 연산을 되돌려놓으라는(undo) 것이다.

```

for (sop = sops; sop < sops + nsops; sop++) {
    if (sop->sem_num >= sma->sem_nsems)
        goto out_unlock_free;
    if (sop->sem_flg & SEM_UNDO)
        undos++;
    if (sop->sem_op < 0)
        decrease = 1;
    if (sop->sem_op > 0)
        alter = 1;
}
alter |= decrease;

```

코드 135. sys_semop() – continued

복제해둔 세마포어 연산들에서 각각이 올바른 값을 가지고 있는지와 UNDO flag를 사용했는지, 그리고 연산이 decrease를 할지 아니면 increase를 할지를 결정한다. 그리고 연산이 세마포어에 대해서 변화(alter)를 일으키는지를 확인한다.

```

error = -EACCES;
if (ipcperms(&sma->sem_perm, alter ? S_IWUGO : S_IRUGO))
    goto out_unlock_free;
if (undos) {
    un=current->semundo;
    while(un != NULL) {
        if(un->semid==semid)
            break;
        if(un->semid== -1)
            un=freeundos(sma,un);
        else
            un=un->proc_next;
    }
    if (!un) {
        error = alloc_undo(sma,&un,semid,alter);
        if(error)
            goto out_free;
    }
} else
    un = NULL;
error = try_atomic_semop (sma, sops, nsops, un, current->pid, 0);
if (error <= 0)
    goto update;

```

코드 136. sys_semop() – continued

ipcperms() 함수를 호출해서 세마포어가 연산에 대한 접근을 허가하는지를 확인한 다음(S_IWUGO or S_IRUGO: write 혹은 read), undo flag가 설정되었는지를 확인한다. undo가 설정되어 있다면, 현재 프로세스의 semundo를 통해서 세마포어 id와 같은 값이 있는지를 확인하고, 만약 세마포어 ID가 -1을 가지는 것이 발견된다면, 세마포어 undo구조체를 전부 free시켜준다(freeundos()). 찾은 세마포어 ID가 없을 경우에는 새로이 세마포어 undo구조체를 할당(alloc_undo())한다.

여기까지 일단 진행하면 undo는 어쨌든 undo구조체를 가지고 있게 된다. 이젠 실제로 세마포어 연산을 적용하는 단계이다. 이것은 try_atomic_semop() 함수를 통해서 한번에 적용 가능하지를 알아보는 보는 것이다.

try_atomic_semop() 함수는 세마포어 연산의 연속이 한번에 성공할 수 있다면 0을 돌려주고, 만약 sleep해야 한다면 1을 돌려줄 것이다. 예러가 생길 경우에는 다시 예러에 해당하는 값을 돌려준다. 즉, 세마포어 연산의 적용 가능성을 점검하는 함수이다. 따라서, 0보다 큰 값을 가지게 된다면 sleep해야 한다. 이때 세마포어 queue에 대한 자료구조를 사용하게 된다. 만약 0이나 그 이하의 값을 가진다면, update로 바로 진행한다.

```

queue.sma = sma;
queue.sops = sops;
queue.nsops = nsops;
queue.undo = un;
queue.pid = current->pid;
queue.alter = decrease;
queue.id = semid;
if (alter)
    append_to_queue(sma ,&queue);
else
    prepend_to_queue(sma ,&queue);
current->semsleeping = &queue;

```

코드 137. sys_semop() – continued

sleep상태로 가기 위해서 세마포어 queue의 구조체를 초기화 하는 과정이다. 세마포어 배열 및 연산과, 연산의 개수 및 undo를 위한 것을 저장하고, 현재 프로세스의 ID와 값의 변화를 명시한다. 그리고, 어떤 세마포어 ID와 관련되었는지를 저장하게 된다. 이와 같은 과정을 마치면 큐를 세마포어 배열이 가르키는 pending 세마포어 연산의 앞이나 혹은 뒤에 붙여두게 된다 append_to_queue(), or prepend_to_queue(). 그리고 나서 현재 프로세스가 세마포어의 획득을 위해서 잠들어 있는 세마포어 큐를 현재의 프로세스의 semsleeping구조에 넣어둔다.

```

for (;;) {
    struct sem_array* tmp;
    queue.status = -EINTR;
    queue.sleeper = current;
    current->state = TASK_INTERRUPTIBLE;
    sem_unlock(semid);
    schedule();
    tmp = sem_lock(semid);
    if(tmp==NULL) {
        if(queue.status != -EIDRM)
            BUG();
        current->semsleeping = NULL;
        error = -EIDRM;
        goto out_free;
    }
    if (queue.status == 1)
    {
        error = try_atomic_semop (sma, sops, nsops, un,
                                current->pid,0);
        if (error <= 0)

```

```

        break;
    } else {
        error = queue.status;
        if (queue.prev) /* got Interrupt */
            break;
        /* Everything done by update_queue */
        current->semsleeping = NULL;
        goto out_unlock_free;
    }
}
current->semsleeping = NULL;
remove_from_queue(sma,&queue);

```

코드 138. sys_semop() – continued

무한 loop를 돌면서 계속적으로 세마포어 연산이 한번에 가능한지를 계속 점검한다. 이때 현재 확인 중이 프로세스가 인터럽트될 수 있다고 표시한 다음(TASK_INTERRUPTIBLE), lock했던 세마포어 id를 놓아준다. 그리고 나서 새로운 프로세스가 스케줄링 되도록 요구한다(schedule()). 만약 다시 실행기회를 얻는다면, 다시 세마포어에 대한 lock연산을 수행하고(sem_lock()), 세마포어 큐의 상태를 확인한다. 만약 세마포어 큐의 상태가 1(아직도 pending되어 있다.)라면, try_atomic_semop() 함수를 호출해서 세마포어 연산이 가능한지를 확인하게 되고, 그렇지 않다면 현재 세마포어에 대한 요구가 interrupt되어 있는지를 확인하게 된다. 무한 loop를 나오게 되면, 일단 어떤 식으로든 세마포어에 대한 연산이 행해졌다는 것을 나타내며, 이젠 현재 프로세스의 세마포어 sleep를 NULL로 만들어준 후, 세마포어 배열로부터 세마포어 큐를 지운다(remove_from_queue()).

```

update:
    if (alter)
        update_queue (sma);
out_unlock_free:
    sem_unlock(semid);
out_free:
    if(sops != fast_sops)
        kfree(sops);
    return error;
}

```

코드 139. sys_semop() – continued

세마포어의 큐를 update(update_queue())하고, 세마포어 ID를 lock했으므로 unlock시켜준다. 그리고, 할당 받았던 세마포어 연산에 대한 구조체를 해제(free)한 다음 에러코드를 돌려주고 복귀한다. 여기서 update_queue()함수를 분석해 보면 아래와 같다.

```

static void update_queue (struct sem_array * sma)
{
    int error;
    struct sem_queue * q;

    for (q = sma->sem_pending; q; q = q->next) {
        if (q->status == 1)
            continue; /* this one was woken up before */
        error = try_atomic_semop(sma, q->sops, q->nsops,
                                q->undo, q->pid, q->alter);
        if (error <= 0) {
            wake_up_process(q->sleeper);
            if (error == 0 && q->alter) {
                /* if q-> alter let it self try */

```

```

        q->status = 1;
        return;
    }
    q->status = error;
    remove_from_queue(sma,q);
}
}

```

코드 140. update_queue()함수

update_queue()함수는 모든 pending상태에 있는 세마포어 queue에 대해서 작용한다. 이 함수는 pending되어있는 큐를 살펴서 적용 가능한 세마포어 연산이 있다면 이를 행하고, 연산이 올바르다면 해당하는 큐의 sleeper에 잠들어있는 프로세스들을 깨우게 된다. 만약 연산이 적용가능하고 세마포어에 변화(alter)를 일으킨다면 큐의 상태는 1값을 가지게 되며, 세마포어 배열에서 큐는 제거될 것이다.

세마포어의 제어에 대한 연산은 sys_semctl() 시스템 콜이 적용된다. 코드는 ~/ipc/sem.c를 참고하기 바란다.

```

asmlinkage long sys_semctl (int semid, int semnum, int cmd, union semun arg)
{
    int err = -EINVAL;
    int version;

    if (semid < 0)
        return -EINVAL;
    version = ipc_parse_version(&cmd);
    switch(cmd) {
    case IPC_INFO:
    case SEM_INFO:
    case SEM_STAT:
        err = semctl_nolock(semid,semnum,cmd,version,arg);
        return err;
    case GETALL:
    case GETVAL:
    case GETPID:
    case GETNCNT:
    case GETZCNT:
    case IPC_STAT:
    case SETVAL:
    case SETALL:
        err = semctl_main(semid,semnum,cmd,version,arg);
        return err;
    case IPC_RMID:
    case IPC_SET:
        down(&sem_ids.sem);
        err = semctl_down(semid,semnum,cmd,version,arg);
        up(&sem_ids.sem);
        return err;
    default:
        return -EINVAL;
    }
}

```

코드 141. sys_semctl() 시스템 콜

넘겨받는 파라미터 값으로는 세마포어의 ID와 세마포어의 개수, 명령과 sys_semtl() 시스템 콜을 위한 argument구조체로 이루어진다. 주로, 세마포어에 대한 정보와 상태를 얻는 연산과 세마포어를 삭제하는 연산을 나누어진다.

시스템 콜을 위한 argument구조체는 아래와 같이 정의된다. 해당하는 정보를 돌려주기 위한 것이다.

```
union semun {
    int val;                                /* 설정할 값 */
    struct semid_ds *buf;                    /* IPC_STAT와 IPC_SET을 위해서 필요한 버퍼 포인터 */
    unsigned short *array;                  /* GETALL이나 SETALL을 위한 배열 */
    struct seminfo *__buf;                 /* IPC_INFO를 위한 버퍼 */
    void *__pad;
};
```

코드 142. semun구조체의 정의

다시 seminfo구조체는 다시 아래와 같은 정의를 가진다.

```
struct seminfo {
    int semmap;                            /* 최대 세마포어 ID 갯수의 값 */
    int semmni;                            /* 최대 시스템에 있을 수 있는 세마포어의 갯수 */
    int semmns;                            /* 시스템내의 undo 구조체의 갯수 */
    int semmnu;                            /* ID당 최대 세마포어 갯수 */
    int semmsl;                            /* 세마포어 연산당 최대 연산의 갯수 */
    int semopm;                            /* 세마포어 연산당 최대 연산의 갯수 */
    int semume;                            /* 프로세스당 undo entry를 의 갯수 */
    int semusz;                            /* sem_undo구조체의 크기 */
    int semvmx;                            /* 세마포어의 최대 크기값 */
    int semaem;                            /* 종료시 정정(adjust)할수 있는 최대크기 */
};
```

코드 143. seminfo구조체의 정의

이 구조체는 현재 시스템의 세마포어 설정 및 사용정보를 알려주기 위해서 사용한다. 이 구조체는 sem_ctl() 함수에서 사용되며, 사용자 주소공간으로 세마포어 정보를 돌려주는데 이용된다. sem_ctl() 함수를 보면 크게 세가지 함수가 있다. semctl_nolock() 함수는 시스템내의 세마포어나 특정 세마포어의 정보를 알려주게 되며, semctl_main()는 사용중인 모든 세마포어의 값을 가져오거나 설정할 때와 특정한 세마포어에 값을 설정하는 역할을 한다. semctl_down()함수는 특정한 세마포어를 제거하거나 값을 새로이 재설정할 때 사용한다. 이 부분에 대한 코드 역시 ~/ipc/sem.c에서 찾을 수 있다.

이상에서 우린 세마포어에 대해서 아주 깊숙이 내부적인 함수까지를 살펴보았다. 이와 같이 했던 이유는 세마포어가 시스템에서 아주 중요한 부분을 차지하고 있으며, 프로세스간 동기화를 위해서 많은 작용을 하고 있기 때문이다. 이것을 잘 이해하는 것이 나머지 IPC에 대한 방법을 이해하는데 많은 도움을 줄 것이다. 앞으로 설명할 것은 이전 세마포어와 기본적으로 동일한 구조를 가지면서 자신만의 고유한 기능을 가지는 System V계열의 IPC 방법 중 메시지 큐와 공유메모리에 대해서 보기로 하겠다. 기본적인 데이터 구조는 세마포어에서 사용한 것과 동일하므로 세마포어에 대한 그림을 참조해도 될 것이다.

2.18.5. Message Queue

메시지는 byte의 연속이다. 즉, 메시지에 들어있는 내용은 관여하지 않는다는 말이다. 프로세스는 메시지를 메시지 큐로 보낼 수도 있고 메시지 큐로부터 메시지를 받을 수도 있다. 여기서 큐(queue)라는 말은 메시지가 들어간 순서대로 메시지 큐로부터 하나씩 꺼낸다는 말이다. 리눅스에서의 메시지 큐의 구현은 아래와 같다. 정의는 ~/ipc/msg.c에 있다.

```
/* one msq_queue structure for each present queue on the system */
struct msg_queue {
    struct kern_ipc_perm q_perm;          /* 접근 허가 설정 */
    time_t q_stime;                      /* 마지막 메시지 전송 시간 */
    time_t q_rtime;                      /* 마지막 메시지 도착(receive) 시간 */
    time_t q_ctime;                      /* 메시지 큐에 대한 마지막 변화(change) 시간 */
    unsigned long q_cbytes;              /* 큐에 저장된 현재 메시지의 길이(byte단위) */
    unsigned long q_qnum;                /* 큐에 있는 메시지의 수 */
    unsigned long q_qbytes;              /* 큐에 있을 수 있는 최대 전체 메시지의 길이(byte단위) */
    pid_t q_lspid;                      /* 마지막으로 메시지를 보낸 프로세스 ID */
    pid_t q_lrpid;                      /* 마지막으로 메시지를 받은 프로세스 ID */
    struct list_head q_messages;         /* 큐에 있는 메시지의 리스트 */
    struct list_head q_receivers;        /* 메시지를 받는 리스트 */
    struct list_head q_senders;          /* 메시지를 보내는 리스트 */
};
```

코드 144. 메시지 큐의 정의

또한 커널에서 메시지 큐를 관리하기 위한 데이터 구조로는 msqid_ds 구조체가 있다. 정의는 ~/ipc/msg.h에 있다.

```
/* 버전 호환때문에 존재하는 msqid_ds구조체의 정의 */
struct msqid_ds {
    struct ipc_perm msg_perm;           /* 접근 허가 설정 */
    struct msg *msg_first;              /* 첫번째 큐상에 존재하는 사용되지 않은 메시지 포인터 */
    struct msg *msg_last;               /* 큐의 마지막에 위치하는 사용되지 않은 메시지 포인터 */
    __kernel_time_t msg_stime;          /* 마지막으로 메시지를 보낸 시간 */
    __kernel_time_t msg_rtime;          /* 마지막으로 메시지를 받은 시간 */
    __kernel_time_t msg_ctime;          /* 마지막 변화(change)시간 */
    unsigned long msg_lcbytes;          /* 사용되지 않는 필드 */
    unsigned long msg_lqbytes;          /* 사용되지 않는 필드 */
    unsigned short msg_cbytes;          /* 현재 큐가 가진 데이터의 크기(byte) */
    unsigned short msg_qnum;             /* 큐에 있는 메시지의 수 */
    unsigned short msg_qbytes;           /* 큐가 가질 수 있는 최대 데이터 크기(byte) */
    __kernel_ipc_pid_t msg_lspid;        /* 마지막으로 메시지를 보낸 프로세스의 ID */
    __kernel_ipc_pid_t msg_lrpid;        /* 마지막으로 메시지를 받은 프로세스의 ID */
};

/* 실제 i386를 위한 msgid_ds정의 */
struct msqid64_ds {
    struct ipc64_perm msg_perm;
    __kernel_time_t msg_stime;
    unsigned long __unused1;
    __kernel_time_t msg_rtime;
    unsigned long __unused2;
    __kernel_time_t msg_ctime;
    unsigned long __unused3;
    unsigned long msg_cbytes;
    unsigned long msg_qnum;
    unsigned long msg_qbytes;
    __kernel_pid_t msg_lspid;
    __kernel_pid_t msg_lrpid;
    unsigned long __unused4;
    unsigned long __unused5;
};
```

```
...
static struct ipc_ids msg_ids;
```

코드 145. 메시지 ID를 위한 구조체

전역 변수 `msg_ids`는 전체 메시큐를 관리할 목적으로 사용된다. 또한 메시지 큐에서 사용하는 상수 정의로는 아래와 같은 것이 있다.

```
#define MSGMNI 16 /* <= IPCMNI */ /* 최대 메시지 큐의 ID값 */
#define MSGMAX 8192 /* <= INT_MAX */ /* 최대 메시지 크기 */
#define MSGMNB 16384 /* <= INT_MAX */ /* 메시지 큐의 기본(default) 최대 크기 */
```

코드 146. 메시지 큐와 관련된 상수 정의

전체적인 메시지 큐의 구조는 [그림16]와 같이 된다. `ipc_ids`구조체와 `ipc_id`구조체를 동일하게 사용하고 있으므로 세마포어 구조에 대한 [그림15]도 참조 하도록하자.

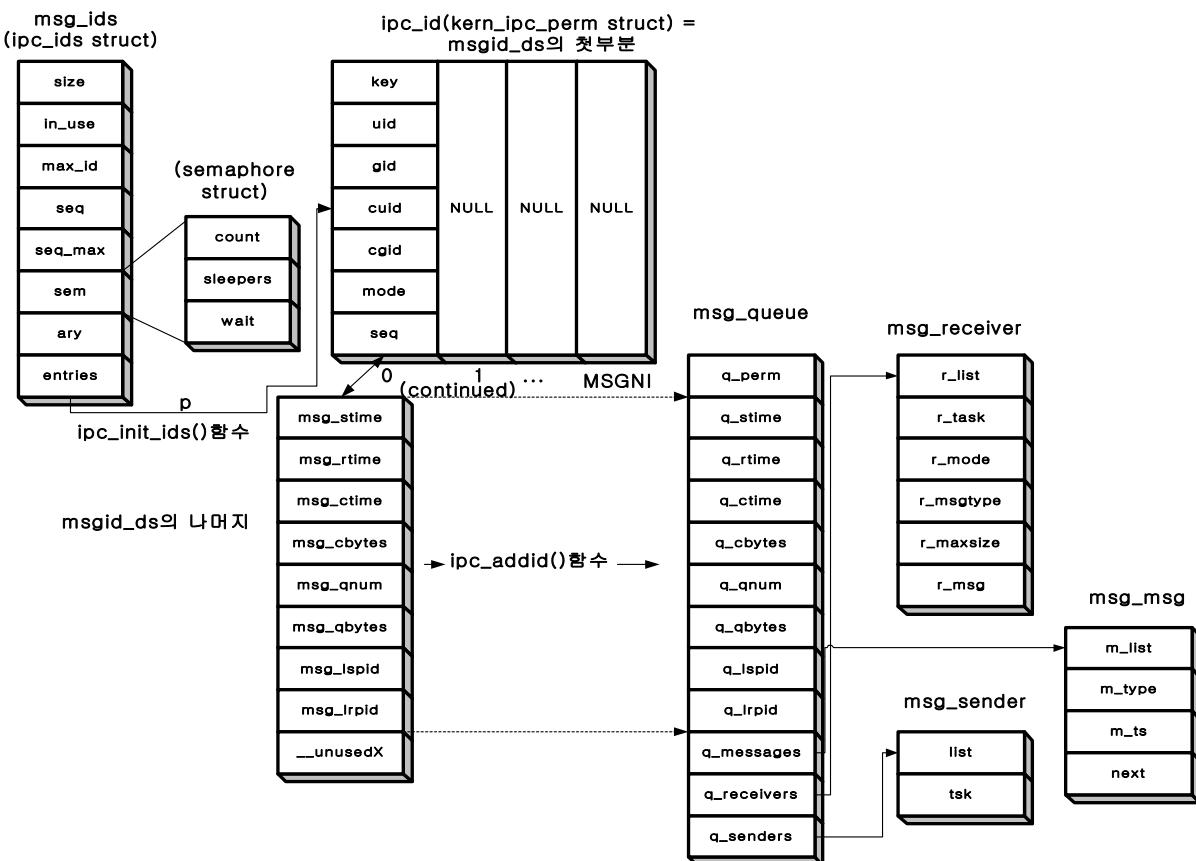


그림 19. 메시지 큐의 전체 구조

메시지 큐의 초기화(`msg_init()`) `ipc_init_ids()` 함수를 호출해서 `msg_ids`구조체와 `msgid_ds`의 배열을 할당한다. 이후에 다시 `msg_get()` 함수에서 `newque()` 함수를 호출하게 될 때, `msg_queue` 구조체가 할당되고 이것이 다시 `msg_ids` 구조체의 `entries`를 통해서 배열의 한부분으로 들어간다. 즉, `msgid_ds` 구조가 있던 곳에 `msg_queue` 가 들어가게 된다. 메시지 큐에서 메시지를 받은 리스트는 `msg_receiver`로 들어가게 되며, 메시지는 `q_messages`에, 보내는 리스트는 `msg_sender`로 들어가게 된다.

메시지 큐에 관련된 시스템 콜로는 아래와 같은 것이 있다. 이하에서 이를 각각에 대해서 알아보도록 하겠다.

```
asmlinkage long sys_msgget (key_t key, int msgflg);
asmlinkage long sys_msghnd (int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
asmlinkage long sys_msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg);
asmlinkage long sys_msqclt (int msqid, int cmd, struct msqid_ds *buf);
```

코드 147. 메시지 큐와 관련된 시스템 콜

먼저 시스템 콜의 정의를 보기전에, 넘겨주는 파라미터 값으로 들어가는 msgbuf구조체를 보기로하자. 이것이 바로 전송할 메시지가 된다. 아래와 같이 정의된다.

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
    long mtype;          /* 메시지 타입 */
    char mtext[1];        /* 메시지 내용 */
};
```

코드 148. 메시지 버퍼의 정의

msgbuf구조체에서 메시지 타입 필드는 사용자 프로세스가 정의할 수 있으며, mtext는 메시지의 내용을 담는다.

```
asmlinkage long sys_msgget (key_t key, int msgflg)
{
    int id, ret = -EPERM;
    struct msg_queue *msq;

    down(&msg_ids.sem);
    if (key == IPC_PRIVATE)
        ret = newque(key, msgflg);
    else if ((id = ipc_findkey(&msg_ids, key)) == -1) { /* key not used */
        if (!(msgflg & IPC_CREAT))
            ret = -ENOENT;
        else
            ret = newque(key, msgflg);
    } else if (msgflg & IPC_CREAT && msgflg & IPC_EXCL) {
        ret = -EXIST;
    } else {
        msq = msg_lock(id);
        if(msq==NULL)
            BUG();
        if (ipcperms(&msq->q_perm, msgflg))
            ret = -EACCES;
        else
            ret = msg_buildid(id, msq->q_perm.seq);
        msg_unlock(id);
    }
    up(&msg_ids.sem);
    return ret;
}
```

코드 149. sys_msgget() 함수의 정의 - 메시지 큐를 생성하는 함수

sys_msgget()함수는 key값과 메시지의 flag값을 받아들여서 새로운 메시지 큐를 생성하고, 생성된 메시지 큐의 ID를 돌려주는 역할을 한다. 새로이 메시지 큐를 생성하는 것은 newque()함수가 하는 일이다. 먼저 전역 변수인 msg_ids에 대해서 세마포어를 얻는다(down()). 만약 key값이 IPC_PRIVATE로 설정되었다면, 새로이 큐를 생성(newqueue())하고, 그렇지 않다면 해당하는 메시지 큐가 있는지를 key값으로 검색한다. 찾지못했다면, 새로운 메시큐에 대한 생성요구인지를 확인하고 그렇지 않다면 에러(-ENOENT)를

돌려준다. 만약 새로운 메시지 큐를 생성하는 요구라면 newqueue()를 호출한다. Key값에 해당하는 메시지 큐를 찾았지만 msgflg값이 IPC_CREATE이고, IPC_EXCL일 경우에는 에러(-EEXIST)를 돌려준다. 그렇지 않다면 key에 해당하는 메시지 큐를 찾았다면, 메시지 큐의 해당 ID에 대해서 lock을 설정하고, 접근허가를 확인한 후, 새로운 메시지의 ID를 생성해서 복귀값으로 둔다. 이전에 설정한 lock을 해제하고, 세마포어를 높아준(up)주고 복귀한다.

```
asmlinkage long sys_ms snd (int msqid, struct msgbuf *msgp, size_t msgs z, int msgflg)
{
    struct msg_queue *msq;
    struct msg_msg *msg;
    long mtype;
    int err;

    if (msgsz > msg_ctlmax || (long) msgsz < 0 || msqid < 0)
        return -EINVAL;
    if (get_user(mtype, &msgp->mtype))
        return -EFAULT;
    if (mtype < 1)
        return -EINVAL;
    msg = load_msg(msgp->mtext, msgsz);
    if (IS_ERR(msg))
        return PTR_ERR(msg);
    msg->m_type = mtype;
    msg->m_ts = msgsz;
    msq = msg_lock(msqid);
    err = -EINVAL;
    if (msq == NULL)
        goto out_free;
retry:
    err = -EIDRM;
    if (msg_checkid(msq, msqid))
        goto out_unlock_free;
    err = -EACCES;
    if (ipcperms(&msq->q_perm, S_IWUGO))
        goto out_unlock_free;
```

코드 150. sys_ms snd() 함수의 정의 - 메시지를 전송 함수

sys_ms snd()함수는 메시지를 보내기 위해서 호출된다. 넘겨받는 값으로는 메시지 큐의 ID와 메시지 버퍼, 메시지 크기와 메시지에 대해서 설정된 flag값이 온다. 함수의 초반은 넘겨받은 값들이 옳바른 값을 가지는지를 확인하는 것이다. 중요한 것으로는 메시지를 가져오는 일(load_msg), 메시지 큐에서 ID를 체크(check)하는 일(msg_checkid()), 그리고, 접근허가를 확인하는 일(ipcperms())이다.

여기서 잠시 msg_msg구조체를 보도록 하자. 정의는 ~/ipc/msg.c에 나와 있으며 아래와 같다.

```
struct msg_msgseg {
    struct msg_msgseg *next;
    /* the next part of the message follows immediately */
};

/* one msg_msg structure for each message */
struct msg_msg {
    struct list_head m_list; /* 메시지 큐의 연결 리스트 */
    long m_type;           /* 메시지의 type */
    int m_ts;              /* 메시지 텍스트의 길이 */
    struct msg_msgseg *next; /* 다음 메시지를 가르키는 포인터 */
    /* 이하의 부분에 바로 실제적인 메시지가 따라나온다. */
```

```
};
```

코드 151. msg_msg구조체의 정의

먼저 msg_msg구조체는 첫필드로 메시지 큐에 들어가는 연결 리스트를 가지며, 메시지의 타입과 길이, 다음 메시지가 어디에 있는지와 실제 메시지가 따라서 나온다.

```
if(msgsz + msq->q_cbytes > msq->q_qbytes ||
   1 + msq->q_qnum > msq->q_qbytes) {
    struct msg_sender s;

    if(msgflg&IPC_NOWAIT) {
        err=-EAGAIN;
        goto out_unlock_free;
    }
    ss_add(msq, &s);
    msg_unlock(msqid);
    schedule();
    current->state= TASK_RUNNING;
    msq = msg_lock(msqid);
    err = -EIDRM;
    if(msq==NULL)
        goto out_free;
    ss_del(&s);
    if (signal_pending(current)) {
        err=-EINTR;
        goto out_unlock_free;
    }
    goto retry;
}
```

코드 152. sys_msgsnd() 함수의 정의(계속) - 메시지의 도착을 알려줌.

보내려는 메시지의 크기(msgsz)와 이전에 큐에 들어있는 메시지(q_cbytes)의 합이 최대 메시지 큐의 크기(q_qbytes)보다 크고, 큐에 들어있는 메시지의 수(q_qnum)와 1을 더 한 값이 최대 메시지 큐의 크기(q_qbytes)보다 클 경우에 한해서 아래와 같은 일을 한다.

먼저 메시지에 설정된 flag가 IPC_NOWAIT라면 에러(-EAGAIN)를 복귀 값으로 설정한 후, out_unlock_free로 제어를 옮긴다. 즉, 메시지가 이미 많이 들어있는 경우가 되므로 기다리지 못할 경우 에러가 된다. 그렇지 않다면, 현재 프로세스의 상태를 TASK_INTERRUPTIBLE로 만든 후, 메시지 큐의 보내는 프로세스의 큐에 현재 프로세스를 집어넣고(ss_add()), lock을 끈 후 스케줄링(schedule())을 요청한다.

다시 스케줄링이 되어서 처리를 계속하게 되면, 현재 프로세스의 상태는 다시 TASK_RUNNING이 되며, 메시지 큐 ID에 해당하는 메시지 큐에 대해서 lock을 설정한 후, 보내는 프로세스의 큐에서 현재 프로세스를 제거한다(ss_del()). 만약 현재 프로세스가 pending된 시그널이 있다면 에러(-EINTR)를 복귀 값으로 설정하고, out_unlock_free로 제어를 옮긴다. 그렇지 않을 경우에는 다시 한번 메시지를 보내려는 시도를 하게 된다(goto retry).

```
if(!pipelined_send(msq,msg)) {
    /* noone is waiting for this message, enqueue it */
    list_add_tail(&msg->m_list,&msq->q_messages);
    msq->q_cbytes += msgsz;
    msq->q_qnum++;
    atomic_add(msgsz,&msg_bytes);
    atomic_inc(&msg_hdrs);
}
err = 0;
```

```

msg = NULL;
msq->q_lspid = current->pid;
msq->q_stime = CURRENT_TIME;
out_unlock_free:
    msg_unlock(msqid);
out_free:
    if(msg!=NULL)
        free_msg(msg);
    return err;
}

```

코드 153. sys_msgsnd() 함수의 정의(계속) - 메시지를 저장하고 복귀함.

pipelined_send() 함수는 메시지 큐의 receivers에 메시지가 오기를 대기하고 있는 프로세스에 메시지가 도착했음을 알리고, 프로세스를 깨워주는 역할을 한다. 제대로 수행이 되었다면, 메시지 큐에 메시지를 넣고(list_add_tail()), 현재 메시지 큐가 가지고 있는 데이터 크기(q_cbytes), 큐에 들어있는 메시지 수(q_qnum)을 증가 시켜준다. 또한 전역 변수인 msg_bytes와 msg_hdrs³³에 대해서도 증가 시켜준다. 복귀하기전에 설정했던 lock은 해제하고, 메시지를 위해서 할당된 메모리는 해제한다(free_msg()).

```

asmlinkage long sys_msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz,
                           long msgtyp, int msgflg)
{
    struct msg_queue *msq;
    struct msg_receiver msr_d;
    struct list_head* tmp;
    struct msg_msg* msg, *found_msg;
    int err;
    int mode;

    if (msqid < 0 || (long) msgsz < 0)
        return -EINVAL;
    mode = convert_mode(&msgtyp,msgflg);
    msq = msg_lock(msqid);
    if(msq==NULL)
        return -EINVAL;
retry:
    err=-EACCES;
    if (ipcperms (&msq->q_perm, S_IRUGO))
        goto out_unlock;

```

코드 154. sys_msgrcv() 함수의 정의 - 메시지를 받기 위한 검사.

자, 이젠 메시지의 보내기에 대해서 볼 차례이다. 해당하는 시스템 콜로는 sys_msgrcv() 함수가 있다. 먼저 메시지 ID와 메시지 크기가 올바른지 확인한다. 바르지 않다면 에러(-EINVAL)를 넘겨주고 복귀한다. 넘겨받은 메시지 flag과 메시지 타입에서 받으려는 메시지에 대한 모드를 결정하게 되는데. 아래와 같은 것이 올 수 있다.

```

#define SEARCH_ANY          1 /* 메시지 타입이 0을 가지는 경우, 첫번째로 들어오는 메시지 */
#define SEARCH_EQUAL         2 /* 메시지 타입이 0이상을 가지는 경우, 메시지 타입과 같은 메시지 */
#define SEARCH_NOTEQUAL      3 /* 메시지 타입과 같지 않은 메시지 */
#define SEARCH_LESSEQUAL      4 /* 메시지 타입이 음의 값을 가지는 경우, 메시지 타입의 절대값보다 작은 값을 메시지 타입으로 가지는 메시지 */

```

코드 155. 메시지의 모드 값

³³ msg_bytes는 현재 큐된 메시지들의 총 크기를 말하며, msg_hdrs는 몇개의 메시지가 큐되었는지를 나타낸다.

이러한 값으로의 변환은 `convert_mode()` 함수가 처리한다. 즉, 사용자 프로세스는 받으려는 메시지의 타입을 결정해 주어야 할 것이다.

주 함수 부분으로 들어가기 전에 메시지 ID에 해당하는 메시지 큐에 lock을 설정하고(`msg_lock()`), 만약 없다면 에러(-EINVAL)를 돌려준다. 접근제어(ipcperms())도 살펴서 해당하는 접근허가 모드를 가졌는지 확인한다.

```
tmp = msq->q_messages.next;
found_msg=NULL;
while (tmp != &msq->q_messages) {
    msg = list_entry(tmp,struct msg_msg,m_list);
    if(testmsg(msg,msgtyp,mode)) {
        found_msg = msg;
        if(mode == SEARCH_LESSEQUAL && msg->m_type != 1) {
            found_msg=msg;
            msgtyp=msg->m_type-1;
        } else {
            found_msg=msg;
            break;
        }
    }
    tmp = tmp->next;
}
```

코드 156. sys_msgrcv() 함수의 정의(계속) - 실제적으로 메시지를 얻어오는 온다.

이전 해당하는 받으려는 메시지가 있는지를 메시지 큐에서 찾는 일이 될 것이다. 먼저 큐에 저장된 메시지(q_messages)를 하나 가져온다. 이것을 가지고 전체 큐에 저장된 메시지를 검색해서 찾으려는 메시지 타입을 가진 메시지를 찾는다. 이때 찾기에 사용되는 것은 모드(mode) 값이 들어가며, 해당하는 메시지를 찾는 것은 `testmsg()` 함수이다. 있다면 찾은 메시지를 `found_msg`에 넣는다.

```
if(found_msg) {
    msg=found_msg;
    if ((msgsz < msg->m_ts) && !(msgflg & MSG_NOERROR)) {
        err=-E2BIG;
        goto out_unlock;
    }
    list_del(&msg->m_list);
    msq->q_qnum--;
    msq->q_rtime = CURRENT_TIME;
    msq->q_lpid = current->pid;
    msq->q_cbytes -= msg->m_ts;
    atomic_sub(msg->m_ts,&msg_bytes);
    atomic_dec(&msg_hdrs);
    ss_wakeup(&msq->q_senders,0);
    msg_unlock(msqid);
out_success:
    msgsz = (msgsz > msg->m_ts) ? msg->m_ts : msgsz;
    if (put_user (msg->m_type, &msgp->mtype) ||
        store_msg(msgp->mtext, msg, msgsz)) {
        msgsz = -EFAULT;
    }
    free_msg(msg);
    return msgsz;
```

코드 157. sys_msgrcv() 함수의 정의(계속) - 메시지 큐에서 메시지를 꺼낸다.

찾는 메시지가 맞다면, 메시지의 크기가 맞는지를 확인한다. 잘못된 크기를 가진다면 에러(-E2BIG)를 복귀 값으로 놓고, `out_unlock`으로 제어를 옮긴다. 메시지를 메시지의 리스트에서 떼어내고(`list_del()`), 떼어낸 메시지 큐의 필드 값을 갱신한다. 이때 전역 변수 `msg_bytes`와 `msg_hdrs`에 대해서 감소시켜준다. 또한 보내는 측의 프로세스들을 깨워주고(`ss_wakeup()`), 메시지 큐에 대한 lock을 해제한다(`msg_unlock`). 여기서부터는 이제 사용자 주소 공간으로 메시지에 있는 데이터를 복제하는 일이다(`put_user()`). 잘 되었다면 떼어낸 메시지를 위한 커널 메모리를 해제(`free_msg()`)하고, 메시지 크기를 복귀 값으로 넘겨준다.

```

} else
{
    struct msg_queue *t;

    if (msgflg & IPC_NOWAIT) {
        err=-ENOMSG;
        goto out_unlock;
    }
    list_add_tail(&msr_d.r_list,&msq->q_receivers);
    msr_d.r_tsk = current;
    msr_d.r_msctype = msgtyp;
    msr_d.r_mode = mode;
    if(msgflg & MSG_NOERROR)
        msr_d.r_maxsize = INT_MAX;
    else
        msr_d.r_maxsize = msgsz;
    msr_d.r_msg = ERR_PTR(-EAGAIN);
    current->state = TASK_INTERRUPTIBLE;
    msg_unlock(msqid);
    schedule();
    current->state = TASK_RUNNING;
    msg = (struct msg_msg*) msr_d.r_msg;
    if(!IS_ERR(msg))
        goto out_success;
    t = msg_lock(msqid);
    if(t==NULL)
        msqid=-1;
    msg = (struct msg_msg*)msr_d.r_msg;
    if(!IS_ERR(msg)) {
        if(msqid!=-1)
            msg_unlock(msqid);
        goto out_success;
    }
    err = PTR_ERR(msg);
    if(err == -EAGAIN) {
        if(msqid==-1)
            BUG();
        list_del(&msr_d.r_list);
        if (signal_pending(current))
            err=-EINTR;
        else
            goto retry;
    }
}

```

코드 158. `sys_msgrcv()` 함수의 정의(계속) - 메시지가 들어오길 대기한다.

이제는 해당하는 메시지를 찾지 못한 것으로 받는 프로세스가 잠들게 되는 부분이다. 만약 받는 프로세스가 IPC_NOWAIT로 메시지 플랙을 설정하고 함수를 호출했다면 에러(-ENOMSG)를 돌려주고 out_unlock으로 제어를 옮기게 된다. 현재의 프로세스는 메시지 큐의 받는 프로세스가 대기하는 큐(q_receivers)에 들어가게 되며, 메시지 타입과 모드, task_struct를 보관한다. 현재의 프로세스의 상태는 TASK_INTERRUPTIBLE이 되며, 설정했던 lock은 풀어준다(msg_unlock()). 마지막으로 schedule()함수를 호출해서 다른 프로세스가 스케줄링이 되도록 요구한다.

이하는 깨어났을 때 하는 부분이다. 먼저 현재의 프로세스 상태를 TASK_RUNNING으로 바꾸고, 받은 메시지를 가져온다. 만약 에러가 없다면, out_success로 제어를 옮기게 된다. 에러가 있는 경우에는 다시 메시지 큐에 대한 lock을 걸고, 다시 한번 메시지를 가져와서 에러검사를 한다. 에러가 없고, 메시지 큐의 ID가 -1아닐 경우에는 메시지 큐에 걸린 락(lock)을 풀게 되며, 제어는 out_success로 다시 옮겨간다.

에러가 있다면, 에러 상황을 확인하게 되며(PTR_ERR()), 에러가 -EAGAIN을 가리키면 도착 메시지를 기다리는 큐에서 자신이 설정한 것을 제거하고(list_del()), pending된 시그널이 있는지를 본다. Pending된 시그널이 있다면, 에러(-EINTR)를 설정하고, 그렇지 않다면 다시 한번 시도하게 된다.

```
out_unlock:
    if(msqid!= -1)
        msg_unlock(msqid);
    return err;
}
```

코드 159. sys_msgrcv() 함수의 정의(계속) – 복귀.

이부분은 lock을 해제하고 에러코드를 돌려준다.

이제 남은 것은 메시지 큐에 대한 제어(control)을 처리하는 것이다. 아래와 같다.

```
asmlinkage long sys_msgctl (int msqid, int cmd, struct msqid_ds *buf)
{
    int err, version;
    struct msg_queue *msq;
    struct msq_setbuf setbuf;
    struct kern_ipc_perm *ipcperm;

    if (msqid < 0 || cmd < 0)
        return -EINVAL;
    version = ipc_parse_version(&cmd);
    switch (cmd) {
    case IPC_INFO:
    case MSG_INFO:
    {
        /* msginfo구조체를 만들어서 시스템의 메시지 큐를 위한 설정 사항을 돌려주며,
        만약 MSG_INFO 명령을 받았다면 시스템 전체의 메시지 큐 사용량을 넘겨준다.*/
    }
    case MSG_STAT:
    case IPC_STAT:
    {
        /* 현재 메시지 큐의 정보를 알려준다.*/
    }
    case IPC_SET:
        /* 사용자 프로세스로 부터 msg_setbuf의 내용을 가져온다.*/
        break;
    case IPC_RMID:
        break;
    default:
```

```
    return -EINVAL;
}
```

코드 160. sys_msgctl() 함수의 정의 - 메시지 큐에 대한 제어.

IPC_SET과 IPC_RMID에 대해서는 처리가 끝나면 switch문을 빠져 나와서 계속 진행되지만, 그렇지 않은 경우에는 즉각 복귀(return)한다.

```
down(&msg_ids.sem);
msq = msg_lock(msqid);
err=-EINVAL;
if (msq == NULL)
    goto out_up;
err = -EIDRM;
if (msg_checkid(msq,msqid))
    goto out_unlock_up;
ipcp = &msq->q_perm;
err = -EPERM;
if (current->euid != ipcp->cuid &&
    current->euid != ipcp->uid && !capable(CAP_SYS_ADMIN))
    goto out_unlock_up;
```

코드 161. sys_msgctl() 함수의 정의(계속) - 메시지 큐의 접근에 대한 검사.

전역 변수 msg_ids에 대해서 세마포어를 얻고(down()), 다시 메시지 큐에 대한 lock을 설정한다(msg_lock()). 만약 메시지 ID에 해당하는 값이 없을 경우에는 에러(-EINVAL)를 설정하고, out_up으로 제어를 옮긴다. 메시지 큐에서 해당하는 메시지를 가져와서 올바른 ID를 가지는지 확인한다(msg_checkid()). 큐의 접근제어 정보는 이하에서 이용하게 됨으로 보관하도록 한다. 만약 현재 프로세스의 신용정보(credential)가 큐에 설정된 값과 다르다고 할 경우에는 out_unlock_up으로 제어를 옮긴다.

```
switch (cmd) {
case IPC_SET:
{
    if (setbuf.qbytes > msg_ctlmnb && !capable(CAP_SYS_RESOURCE))
        goto out_unlock_up;
    msq->q_qbytes = setbuf.qbytes;
    ipcp->uid = setbuf.uid;
    ipcp->gid = setbuf.gid;
    ipcp->mode = (ipcp->mode & ~S_IRWXUGO) |
        (S_IRWXUGO & setbuf.mode);
    msq->q_ctime = CURRENT_TIME;
    /* sleeping receivers might be excluded by
     * stricter permissions.
     */
    expunge_all(msq,-EAGAIN);
    /* sleeping senders might be able to send
     * due to a larger queue size.
     */
    ss_wakeup(&msq->q_senders,0);
    msg_unlock(msqid);
    break;
}
case IPC_RMID:
    freeque (msqid);
    break;
}
```

```
err = 0;
```

코드 162. sys_msgctl() 함수의 정의(계속) - 실제적인 메시지 큐에 대한 제어.

앞에서 수행하다 만 IPC_SET과 IPC_RMID에 대한 처리를 계속 실행하는 부분이다. 해당하는 메시지 큐의 접근허가(permission)와 큐에 존재하는 메시지의 크기(q_qbytes) 및 변화(change)시간(q_ctime)을 갱신하고, 모든 메시지 큐의 받는 역할을 하는 프로세스들에게 -EAGAIN을 돌려주고 깨운다(expunge_all()). 그리고, 마지막으로 보내는 프로세스를 깨운 다음 메시지에 설정한 lock을 해제한다.

IPC_RMID의 경우에는 메시지 큐의 ID에 해당하는 메시지 큐를 지운다(freeque()).

```
out_up:
    up(&msg_ids.sem);
    return err;
out_unlock_up:
    msg_unlock(msqid);
    goto out_up;
out_unlock:
    msg_unlock(msqid);
    return err;
}
```

코드 163. sys_msgctl() 함수의 정의(계속) - 복귀.

복귀하기 전에 세마포어를 높아주고(up), 메시지에 대한 lock이 설정된 경우에는 풀어준다. 에러코드는 제어가 옮겨오기 전에 설정되어 있으므로 복귀 값으로 넘겨주면 된다.

2.18.6. Shared Memory

공유 메모리(shared memory)는 IPC의 여러 형태 중에서 가장 빠른 것으로, 다른 프로세스가 사용하는 메모리 영역에 대해서 직접적으로 접근해서 데이터를 읽거나 쓸 수 있다. 이것을 사용하기 위해서는 프로세스는 반드시 동기적(synchronously)으로 원하는 메모리 영역에 대해서 접근해야 해야 할 것이다.

공유 메모리 역시 특정한 번호를 공유되는 부분에 대한 ID로 사용하고 있으며, 이 번호는 shmid_ds 구조체를 가르키는 역할을 한다. 공유되는 메모리에 대해서, 공유하고자 하는 프로세스는 붙이기(attach)와 떼내기(detach) 연산을 행하게 되며, 붙이기를 했다면 프로세스의 공간 내에 그 메모리 영역이 있다고 생각하게 된다. shmid_ds 자료구조는 아래와 같다.

```
/* ~/include/linux/shm.h에 정의된 shmid_ds 자료구조 */
struct shmid_ds {
    struct ipc_perm          shm_perm;      /* 공유메모리에 대한 접근허가 모드 */
    int                      shm_segsz;     /* 세그먼트의 크기(bytes) */
    __kernel_time_t          shm_atime;     /* 마지막 attach연산을 한 시간 */
    __kernel_time_t          shm_dtime;     /* 마지막 detach연산을 한 시간 */
    __kernel_time_t          shm_ctime;     /* 마지막 변화(change)된 시간 */
    __kernel_ipc_pid_t       shm_cpid;      /* 생성한 프로세스의 pid */
    __kernel_ipc_pid_t       shm_lpid;      /* 마지막 연산을 한 프로세스의 pid */
    unsigned short           shm_nattch;    /* 현재 attach를 한 수 */
    unsigned short           shm_unused;    /* 호환성 때문에 있는 부분들 */
    void                     *shm_unused2;
    void                     *shm_unused3;
};

/* ~/include/asm/shmbuf.h에 정의된 Intel i386 architecture를 위한 shmid64_ds 자료구조 */
struct shmid64_ds {
    struct ipc64_perm  shm_perm;
    size_t              shm_segsz;
    __kernel_time_t     shm_atime;
```

```

unsigned long          __unused1;
__kernel_time_t       shm_dtime;
unsigned long          __unused2;
__kernel_time_t       shm_ctime;
unsigned long          __unused3;
__kernel_pid_t        shm_cpid;
__kernel_pid_t        shm_lpid;
unsigned long          shm_nattch;
unsigned long          __unused4;
unsigned long          __unused5;
};

```

코드 164. 메시지 큐에서 사용하는 자료구조

공유 메모리에서 사용하는 상수들에 대한 값은 아래와 같은 것들이 있다.

```

#define SHMMAX 0x2000000          /* 최대 공유메모리 세그먼트의 크기(byte단위)*/
#define SHMMIN 1                  /* 최소 공유메모리 세그먼트의 크기(byte단위)*/
#define SHMMNI 4096              /* 최대 시스템 내의 공유메모리 세그먼트의 갯수*/
#define SHMALL (SHMMAX/PAGE_SIZE*(SHMMNI/16)) /* 최대 공유메모리의 시스템내의 크기(page단위)*/
#define SHMSEG SHMMNI /* 프로세스당 공유메모리 세그먼트의 갯수*/

```

코드 165. 메시지 큐에서 사용하는 상수 값

공유메모리에 대한 연산으로는 시스템 콜로 sys_shmget(), sys_shmat(), sys_shmdt(), sys_shmctl()이 있으며 아래와 같이 정의 된다.

```

asmlinkage long sys_shmget (key_t key, size_t size, int flag);
asmlinkage long sys_shmat (int shmid, char *shmaddr, int shmflg, unsigned long *addr);
asmlinkage long sys_shmdt (char *shmaddr);
asmlinkage long sys_shmctl (int shmid, int cmd, struct shmid_ds *buf);

```

코드 166. 공유메모리에 대한 시스템 콜 인터페이스

먼저 공유메모리 영역을 획득하는 sys_shmget() 함수에 대해서 보도록 하자. 정의는 ~/ipc/shm.c에 나와 있다.

```

asmlinkage long sys_shmget (key_t key, size_t size, int shmflg)
{
    struct shmid_kernel *shp;
    int err, id = 0;

    down(&shm_ids.sem);
    if (key == IPC_PRIVATE) {
        err = newseg(key, shmflg, size);
    } else if ((id = ipc_findkey(&shm_ids, key)) == -1) {
        if (!(shmflg & IPC_CREAT))
            err = -ENOENT;
        else
            err = newseg(key, shmflg, size);
    } else if ((shmflg & IPC_CREAT) && (shmflg & IPC_EXCL)) {
        err = -EEXIST;
    } else {
        shp = shm_lock(id);
        if(shp==NULL)
            BUG();
        if (shp->shm_segsz < size)

```

```

        err = -EINVAL;
    else if (ipcperms(&shp->shm_perm, shmflg))
        err = -EACCES;
    else
        err = shm_buildid(id, shp->shm_perm.seq);
        shm_unlock(id);
    }
    up(&shm_ids.sem);
    return err;
}

```

코드 167. sys_shmget() 함수의 정의 - 공유 메모리의 생성.

sys_shmget() 함수는 key와 size, 그리고 공유메모리 영역에 대한 flag를 argument로 받는다. 먼저 공유 메모리 ID에 대한 세마포어를 낮추어서 다른 프로세스들의 접근을 막고, IPC_PRIVATE로 key가 설정되었을 경우에 newseg() 함수를 호출해서 새로운 공유메모리 영역에 대한 요구를 처리한다. 그렇지 않다면, 넘겨받은 key를 가지고, 해당하는 공유메모리 ID를 찾는다(ipc_findkey). 다시 찾은 ID값이 없고, IPC_CREAT요구가 아니라면, 에러(-ENOENT)를 돌려주고, 그렇지 않다면 새로이 공유메모리를 요구한다(newsag()).

여기까지 진행하면 일단 해당하는 key값을 찾은 상태가 된다. 이때 만약 IPC_CREAT를 가지고 있으면서 IPC_EXCL로 설정되었다면, 당연히 에러(-EXEXIST)가 될 것이며, 만약 그렇지 않다면, 이젠 ID를 가지고 공유메모리를 가져오는 과정이 된다. 요구한 크기보다 작은 공유메모리를 가진다면 에러(-EINVAL)가 되고, 공유메모리에 대한 허가(permission)가 일치하지 않는다면 또한 에러(-EACCESS)가 될 것이다. 모든 요구를 만족시킨다면 ID를 가지고, 공유메모리에 대한 ID를 생성해서 돌려줄 것이다(shm_buildid()).

이 함수에서 shm_ids구조체는 전역 변수로서 시스템내의 모든 공유메모리에 대한 ID들을 가지는 ipc_ids구조체로 선언된 것이다. 또한 shmid_kernel구조체는 아래와 같은 정의를 가진다.

```

struct shmid_kernel /* private to the kernel */
{
    struct kern_ipc_perm    shm_perm;      /* IPC의 허가모드(permission mode) */
    struct file *            shm_file;     /* 공유메모리에 관련된 file구조체의 포인터 */
    int                      id;           /* 공유메모리 ID */
    unsigned long             shm_nattch;   /* 현재 attach한 수 */
    unsigned long             shm_segsz;    /* 공유된 메모리의 크기(세그먼트 크기) */
    time_t                   shm_atim;     /* 마지막 attach 시간 */
    time_t                   shm_dtim;     /* 마지막 detach 시간 */
    time_t                   shm_ctim;     /* 마지막 change 시간 */
    pid_t                     shm_cprid;    /* 생성한 프로세스의 PID */
    pid_t                     shm_lprid;    /* 마지막으로 연산을 한 프로세스의 PID */
};

```

코드 168. shmid_kernel구조체의 정의

이것은 커널 내에서만 사용하는 데이터 구조로 shm_file필드부분을 제외하고는 shmid_ds구조체와 동일하다.

여기서 이야기를 더 진행하기 전에, 전체적인 공유메모리의 구조를 보도록 하자. [그림17]과 같다. ipc_init()함수에서 shm_init()함수를 호출하고, 다시 ipc_init_ids()를 호출해서 shmid_ds구조체의 배열³⁴을 할당 받는다. 그리고나서 새로운 공유메모리를 생성하고자 하면, shmid_kernel구조체를 할당 받아서, shmid_ds 구조체(ipc_ids구조체)의 entries가 가르키는 포인터의 배열에 넣어주게 된다. 배열에 들어가게 될 때는 공유메모리의 id를 가지고 자리를 찾게 된다(ipc_addid()).

³⁴ 공유메모리에 대한 배열은 초기화에서 하나만을 할당 받는다.

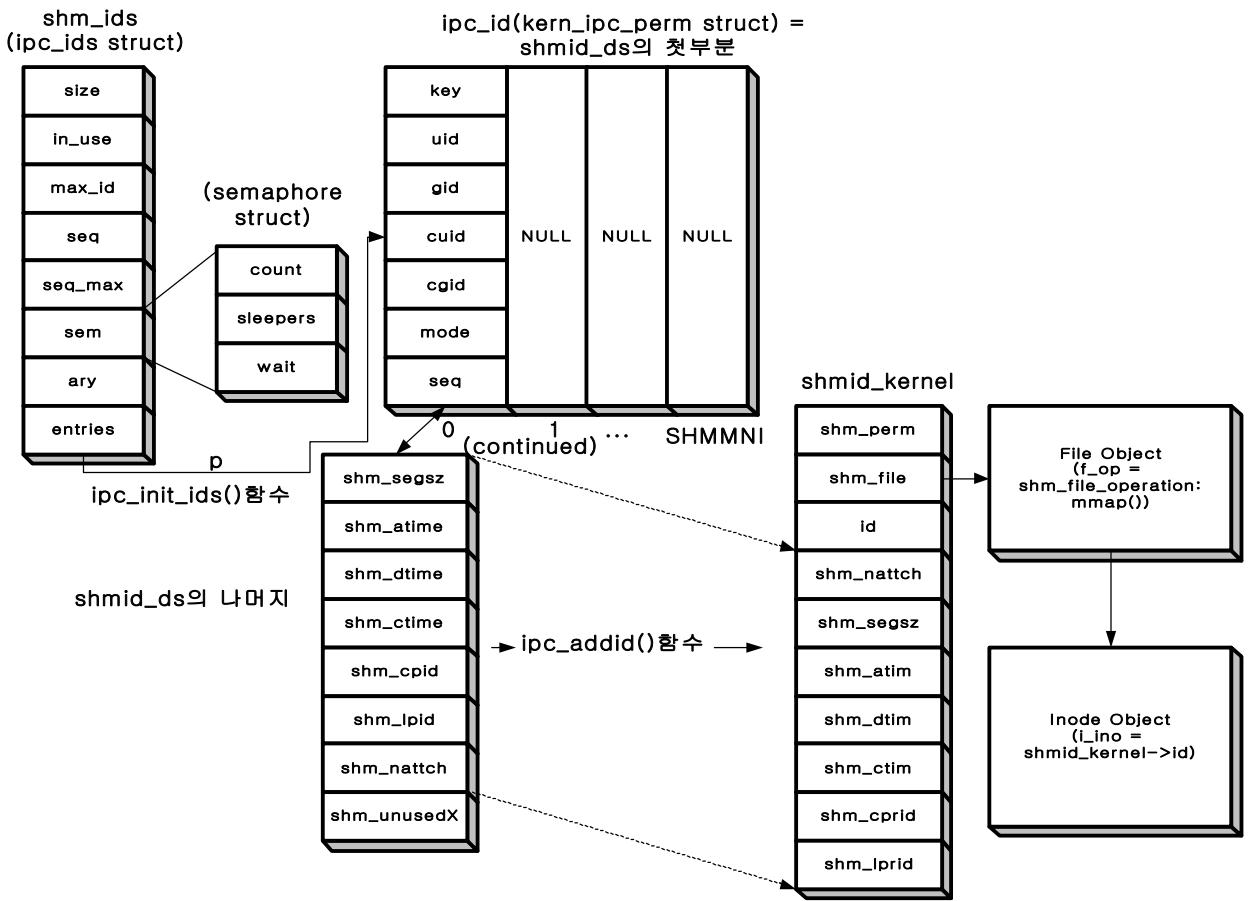


그림 20. 공유 메모리의 전체 구조

시스템 콜의 이야기로 다시 돌아가서, sys_shmat() 함수는 아래와 같이 정의된다. 역시 ~ipc/shm.c에 정의되어 있다.

```
smlinkage long sys_shmat (int shmid, char *shmaddr, int shmflg, ulong *raddr)
{
    struct shmid_kernel *shp;
    unsigned long addr;
    struct file * file;
    int err;
    unsigned long flags;
    unsigned long prot;
    unsigned long o_flags;
    int acc_mode;
    void *user_addr;

    if (shmid < 0)
        return -EINVAL;
    if ((addr = (ulong)shmaddr)) {
        if (addr & (SHMLBA-1)) {
            if (shmflg & SHM_RND)
                addr &= ~(SHMLBA-1); /* round down */
            else
                return -EINVAL;
        }
    }
}
```

```

        }
        flags = MAP_SHARED | MAP_FIXED;
    } else
        flags = MAP_SHARED;
    if (shmflg & SHM_RDONLY) {
        prot = PROT_READ;
        o_flags = O_RDONLY;
        acc_mode = S_IRUGO;
    } else {
        prot = PROT_READ | PROT_WRITE;
        o_flags = O_RDWR;
        acc_mode = S_IRUGO | S_IWUGO;
    }
/*
 * We cannot rely on the fs check since SYSV IPC does have an
 * additional creator id...
 */
shp = shm_lock(shmid);
if(shp == NULL)
    return -EINVAL;
if (ipcperms(&shp->shm_perm, acc_mode)) {
    shm_unlock(shmid);
    return -EACCES;
}

```

코드 169. sys_shmat() 함수의 정의 - 공유메모리를 덧붙이는(attach)함수.

이 함수는 공유메모리를 현재의 프로세스의 주소공간에 더 붙이는 역할을 한다. 먼저 공유메모리 ID의 값을 확인하고(-EINVAL), 공유메모리의 주소를 가져온다. 이때 주소가 0이 아닌 값을 가진다고 있다면, 페이지 크기(SHMLBA=PAGE_SIZE in i386)에서 1을 뺀 값(0xFFFF)에 AND시킨 값을 본다. 만약 어떤 값이 있고, flag이 round가 설정되어(SHM_RND) 있으면, 페이지 단위로 만들어주고, flag이 설정되어 있지 않다면 에러(-EINVAL)를 돌려준다. 이하에서는 flag값을 MAP_SHARED와 MAP_FIXED값을 OR시켜서 넣어준다. 주소 값으로 아무 값도 없다면 flag값은 MAP_SHARED만 설정한다. MAP_SHARED는 메모리의 내용이 공유됨(변화가 다른 프로세스에도 반영됨)을 나타내고, MAP_FIXED는 주소를 정확하게 해석하라는 표시다.

이젠 공유메모리의 flag값을 확인하는 일이다. 만약 읽기 전용(SHM_RDONLY)인 경우와 그럴지 못한 경우에 대해서 각각 해당하는 값을(protection, file open flag, access mode)을 설정한다. 그리고 나서, 공유메모리에 대한 연산을 시작하기 전에 lock과 접근허가(permission)를 검사한다(shm_lock(), ipcperms())..

```

file = shp->shm_file;
shp->shm_nattch++;
shm_unlock(shmid);
down(&current->mm->mmap_sem);
user_addr = (void *) do_mmap(file, addr, file->f_dentry->d_inode->i_size, prot, flags, 0);
up(&current->mm->mmap_sem);
down (&shm_ids.sem);
if(!(shp = shm_lock(shmid)))
    BUG();
shp->shm_nattch--;
if(shp->shm_nattch == 0 &&
   shp->shm_flags & SHM_DEST)
    shm_destroy (shp);
shm_unlock(shmid);
up (&shm_ids.sem);
*raddr = (unsigned long) user_addr;
err = 0;
if (IS_ERR(user_addr))

```

```

    err = PTR_ERR(user_addr);
    return err;
}

```

코드 170. sys_shmat() 함수의 정의(계속) - 사용자 주소 공간에 공유메모리 공간에 mapping하기.

공유메모리에 대한 파일 object를 얻고, shmid_kernel의 shm_nattach필드를 갱신하고 나면 shmid의 lock을 푼다. 이젠 현재 프로세스의 주소공간을 고치는 일이 남는다. 들어가기 전에 현재 프로세스의 mm필드에 세마포어를 down시킨다. do_mmap() 함수는 프로세스의 사용자 공간에 파일 영역을 mapping하는 함수이다. 끝나면 현재 프로세스의 세마포어 값을 다시 up시킨다.

여기까지 진행되면 전역 변수인 shm_ids에 세마포어를 획득(down)하고, 이젠 공유메모리의 shmid의 shm_nattach횟수를 감소 시킨다. 만약 이미 공유메모리가 삭제되었을 경우를 대비해서, shm_nattach가 0을, shm_flags에 SHM_DEST(공유메모리를 제거(destroy)함)가 설정되어 있다면, shm_destroy()함수를 호출해서 공유메모리를 제거한다. 물론 이것 역시 lock을 shmid에 걸어서 확인하는 작업이 될 것이다. 이젠 전역 변수인 shm_ids에 대해서 다시 세마포어를 놓아준다. 마지막으로 할 일은 attach된 공유메모리 공간의 주소 값을 확인하는 것이다. 만약 에러가 있다면 복귀 값을 돌려준다.

```

asmlinkage long sys_shmdt (char *shmaddr)
{
    struct mm_struct *mm = current->mm;
    struct vm_area_struct *shmd, *shmdnext;

    down(&mm->mmap_sem);
    for (shmd = mm->mmap; shmd; shmd = shmdnext) {
        shmdnext = shmd->vm_next;
        if (shmd->vm_ops == &shm_vm_ops
            && shmd->vm_start - (shmd->vm_pgoff << PAGE_SHIFT) == (ulong) shmaddr)
            do_munmap(mm, shmd->vm_start, shmd->vm_end - shmd->vm_start);
    }
    up(&mm->mmap_sem);
    return 0;
}

```

코드 171. sys_shmdt() 함수의 정의 - 공유 메모리를 떼어내는(detach) 함수.

sys_shmdt() 함수는 공유메모리 영역을 프로세스로부터 떼어내는 역할을 한다. 먼저 현재 프로세스의 mm(memory management)필드 값을 구해와서 세마포어를 얻는다(down). 프로세스의 전체 mapping된 메모리 영역에 대해서, 해당하는 연산이 shm_vm_ops와 같고, 넘겨받은 부분에 해당하는 공유메모리 영역(shmaddr)과 같다면, do_munmap() 함수를 호출해서 공유메모리 mapping을 해제한다. 끝나면 세마포어를 원래의 값을 돌려놓고(up) 복귀한다.

sys_shmctl()은 공유메모리 영역에 대한 제어 요구를 처리한다. ~/ipc/shm.c를 참조하자.

```

asmlinkage long sys_shmctl (int shmid, int cmd, struct shmid_ds *buf)
{
    struct shm_setbuf setbuf;
    struct shmid_kernel *shp;
    int err, version;

    if (cmd < 0 || shmid < 0)
        return -EINVAL;

    version = ipc_parse_version(&cmd);

```

코드 172. sys_shmctl() 함수의 정의 - 공유메모리에 대한 제어(control)를 담당하는 함수.

먼저 지역적으로 사용하게될 변수들을 정의한다. 그리고, 넘겨받은 공유메모리 ID와 제어요구가 옳바른 값을 가져는지를 확인하고, 제어 요구 명령어로부터 버전(version)정보(64bit인지)를 넘겨받는다.

```

switch (cmd) { /* replace with proc interface ? */
case IPC_INFO:
{
    /* shminfo64의 구조체를 만들어서 현재 공유 메모리 사용과 관련된 시스템의 정보를
    넘겨준다.*/
}
case SHM_INFO:
{
    /* shm_info의 구조체를 만들어서 공유메모리의 현재 사용 현황을 알려준다.*/
}
case SHM_STAT:
case IPC_STAT:
{
    /* 특정 공유메모리의 사용현황을 알려준다.*/
}
case SHM_LOCK:
case SHM_UNLOCK:
{
    /* 특정 공유메모리에 대한 lock을 설정하거나 lock설정을 풀다.*/
}
case IPC_RMID:
{
    /* 특정 ID에 해당하는 공유메모리의 ID를 삭제한다.*/
}
case IPC_SET:
{
    /* 공유 메모리에 대한 정보(사용자 ID나 그룹 사용자 ID, 혹은 flag값)를 설정한다.*/
}
default:
    return -EINVAL;
}

```

코드 173. sys_shmctl() 함수의 정의(계속) – 공유 메모리에 대한 제어 값을.

명령어 형태에 따라서 처리하는 부분이다. 각각에 해당하는 연산을 정의하는 것으로 끝마치며 예러가 있을 경우에는 이하에 나오는 곳으로 제어를 옮긴다.³⁵

```

err = 0;
out_unlock_up:
    shm_unlock(shmid);
out_up:
    up(&shm_ids.sem);
    return err;
out_unlock:
    shm_unlock(shmid);
    return err;
}

```

코드 174. sys_shmctl() 함수의 정의(계속) - 복귀.

명령(command)에 해당하는 연산을 하는 도중에 생긴 lock을 해제하거나, 세마포어를 증가(up)시키는 등의 연산과 예러 코드를 돌려주는 부분이다.

³⁵ 여기서 필요 없이 코드가 길어질 것 같아서 간단히 요약 정리만 하였다.

이상에서 우린 기본적인 IPC방법과 System V계열에서 지원하는 IPC방법들에 대해서 살펴보았다. 가장 간단한 방법은 시그널을 보내는 방법이며, 둘 간의 적은 데이터를 보내는 방법으로는 PIPE와 FIFO가 있다는 것을 배웠다. 또한 좀더 고급화된 IPC방법으로 System V계열에서 지원하는 세마포어, 메시지, 공유메모리를 사용하는 방법도 알아보았다. 실제에 있어서 System V계열의 IPC는 Solaris(v2.5이상)와 같은 시스템에서 사용되고 있으며, 리눅스에서는 이와 동일한 인터페이스를 제공하고 있다. 따라서, 이 글에서 보여준 IPC방법은 리눅스에서만 독특한 것은 아니며, 이미 일반화된 프로세스간 통신 방법이라고 하겠다. 나중에 운영체제를 새로이 개발하거나 이식하는 일을 하게 될 때, IPC를 사용하는 상위의 응용프로그램이 코스(source) 레벨에서 호환성을 갖기 위해서는 반드시 이 글에서 이야기 하는 기능 정도는 지원해 주어야 할 것이다. 참고로 solaris의 경우에는 이 글에서 보여준 IPC 방법 이외에 좀더 빠른 IPC방법으로 door라는 방법도 제공하고 있다.

2.19. Process의 Memory Image Layout

보통의 경우 process는 자신의 주소공간(address space)을 32bit machine인 경우에 4Gigabytes로 인식한다. 즉, 자신이 사용할 수 있는 총 주소공간을 4Gigabytes가 있다고 여기는 것이다. 리눅스에서는 이러한 주소공간을 [그림18]과 같이 사용하고 있다.

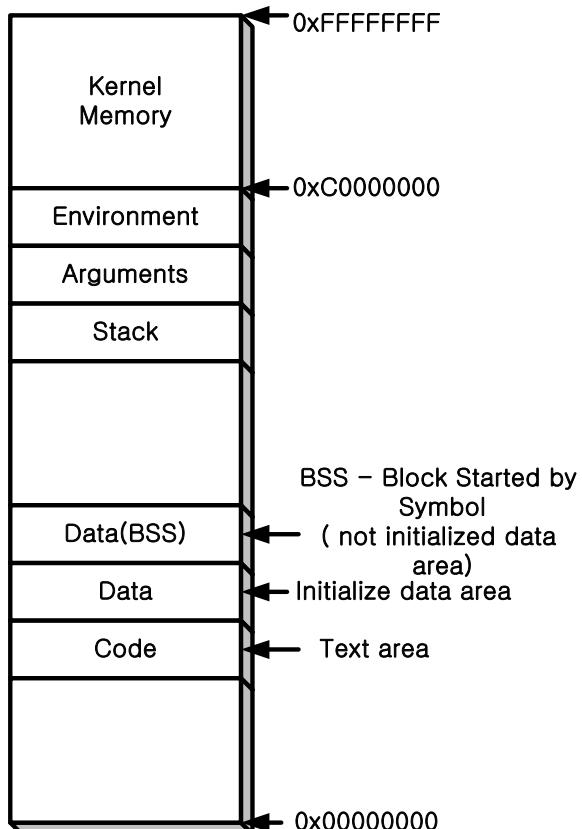


그림 21. Process의 memory내의 Image

즉, 1에서 3Gigabytes까지는 자신의 address space에 있다고 보고, 나머지 1Gigabytes의 공간을 kernel에서 사용한다고 생각한다. kernel공간으로의 직접적인 접근은 불가능하며, system call과 같은 것을 통해서 접근한다. 또한, 모든 process는 [그림18]과 같은 공간을 자신이 가지고 있다고 생각하므로, 각각의 process가 자신만이 공간을 access하도록 하는 mechanism을 kernel에서 제공해 주어야 한다. 이 부분은 나중에 가상 메모리(Virtual Memory)에 대해서 알아볼 때 좀더 자세히 볼 것이다. 참고로 window의 경우에는 1에서 2Gigabyte영역을 process가 쓴다고 생각하며, 나머지 1Gigabytes를 DLL과 같은 library가 나머지 1Gigabytes를 kernel에서 사용한다고 본다.

그리고, [그림18]에서 BSS라고 되어있는 부분은 block started by symbol의 약자로 프로그램에서 초기화되지 않은 변수들이 저장되는 공간이다. 이것은 전통적인 Unix환경에서의 어휘를 따온 것으로 프로그램을 작성할 때 변수에 값을 주지 않고, 선언(declaration)만 해놓은 경우에 이곳에 그 변수가 들어가게 된다. 물론 이것은 compiler가 적절히 해 주어야 할 것이다.

User process는 자신만의 공간을 access할 수 있으므로 만약 특정의 system 영역을 access하려고 할 때는 kernel에 이것을 요청해서 자신의 주소 공간상에 mapping시켜야 한다. 이러한 것의 예로 들 수 있는 것이 frame Buffer를 사용하는 것이다. 즉, video memory buffer를 user process address공간으로 mapping해서 직접적으로 해당 memory를 access할 수 있게 된다. 또한 kernel mode로 진행하게 될 경우에는 trap을 발생시켜서 kernel mode로 동작하게 되며, 이럴 경우 자신이 mapping하고 있는 address공간도 달라지게 된다.

2.20. 커널에서 제공하는 기능들

커널은 프로세스의 실행을 원활히 하기위한 여러가지 기능들을 가지고 있다. 예를 들어 현재 프로세스가 진행중인 상황에서 커널이 처리할 critical한 작업을 수행하고 이를 프로세스들의 상태에 반영하기 위한 일련의 기능을 갖추고 있다. 또한 인위적으로 프로세스의 실행을 나중에 재개시키는 방법과 timer의 설정 및 해지, 프로세스들이 대기하는 큐를 만들 수 있는 메커니즘을 제공한다. 여기서는 이것과 관련되는 부분들을 보기로 하자.

2.20.1. Bottom Half

Bottom half란 interrupt의 결과로 불려지는 커널내의 함수(routine)들의 집합이다. 프로세스의 상태에 의존하지 않으며, sleep()과 같은 함수를 불러서 진행을 블록킹(blocking) 할 수 없다. 참고로 top half란 시스템 콜이나 트랩(trap)³⁶의 결과로 생기며, 동기적(synchronous)으로 호출되는 커널내의 함수(routine)들이다. 프로세스와 상태에 의존적이며, sleep()함수를 부름으로써 블록킹 할 수 있다.

인터럽트의 발생시 이를 처리하는 모든 함수들이 불려질 필요는 없다. 즉, 바쁘고 중요한 일을 처리한 다음 나중에 덜 바쁜 일을 처리해 주도록 만들어줄 수 있다. 이와 같은 대표적인 경우로 네트워크(network)에서 발생하는 패킷(packet)의 처리를 나중으로 미루어 둘 수도 있을 것이다. 되도록이면 많은 패킷을 놓치지 않고 빨리 받아서 큐에 넣어둔 다음 네트워크 인터페이스 인터럽트를 처리했다고 알리고, 나중에 이 패킷들에 대한 처리를 다시 조금 한가한 시간에 해주게 된다. 이럴 때 사용할 수 있는 것이 바로 bottom half이다. 따라서, 상대적으로 처리시간이 긴 것들은 이것을 이용해서 나중에 시스템에서 처리해 준다.

커널 버전(version) 2.4에서는 bottom half에 대한 처리가 많이 변경되었다. 즉, 소프트웨어 IRQ의 일환으로 처리된다. Bottom half의 초기화는 ~/kernel/softirq.c의 softirq_init()에서 bh_base 배열(array)에 init_bh()함수가 bottom half 핸들러 함수를 등록시켜주는 곳에서 일어나며, bh_acton()함수에서 bottom half에 대한 처리를 해준다. 제거는 remove_bh() 함수가 처리한다. 모든 시스템 콜이 복귀하기 전에 softirq와 mask를 가지고 softirq가 활성(active)인지를 확인하게 되며, 만약 그럴 경우에는 do_softirq() 함수를 불러서 softirq를 처리한다.

do_softirq() 함수는 소프트웨어 인터럽트를 처리하기 위한 함수이다.

```
asmlinkage void do_softirq()
{
    int cpu = smp_processor_id();
    __u32 active, mask;

    if (in_interrupt())
        return;
    local_bh_disable();
    local_irq_disable();
    mask = softirq_mask(cpu);
    active = softirq_active(cpu) & mask;
```

³⁶ 시스템에 오류(fault)가 발생할 때 발생한다.

먼저 현재 CPU의 ID값을 가져오고, 인터럽트가 진행 중인지를 확인한다. 만약 진행 중이라면 곧바로 복귀하며, 그렇지 않을 경우에는 bottom half 및 인터럽트를 일어나지 못하게 만든다. 그리고 나서 softirq의 mask와 현재 CPU에서 활성화되어 있는 softirq를 가져온다.

```

if (active) {
    struct softirq_action *h;
restart:
    /* Reset active bitmask before enabling irqs */
    softirq_active(cpu) &= ~active;
    local_irq_enable();
    h = softirq_vec;
    mask &= ~active;
    do {
        if (active & 1)
            h->action(h);
        h++;
        active >>= 1;
    } while (active);
    local_irq_disable();
    active = softirq_active(cpu);
    if ((active &= mask) != 0)
        goto retry;
}

```

만약 활성화된 softirq가 있다면, 이것이 처리가 되었다는 것을 나타내주고(&~active), 현재의 CPU에서 IRQ가 다시 발생할 수 있도록 만들어준다. do ~ while을 돌면서 설정된 함수들을 하나하나 처리해나가고, 처리가 끝나면 다시 IRQ가 발생하지 못하도록 만들어준다. 다시 처리하는 동안에 발생하였을지 모를 인터럽트에 대한 부가적인 softirq들이 활성화 되었는지를 확인한 다음, 만약 더 있다면 다시 이것을 위한 처리 절차를 밟게 된다.

```

local_bh_enable();
return;
retry:
    goto restart;
}

```

다시 bottom half를 가능(enable)하도록 만들고 복귀한다. 이때 무한 반복을 막기 위해서 국지적인(local) 하드웨어 인터럽트를 불가(disable)로 놓고 복귀함에 유의하자.

Bottom half를 위한 우선 순위는 아래와 같다. 즉, TIMER_BH가 가장 높은 우선순위를 가지고, 이어서 TQUEUE_BH가 오게 된다. 유념해서 봐야 할 부분은 TIMER_BH, TQUEUE_BH, IMMEDIATE_BH정도가 될 것이다.

```

enum {
    TIMER_BH = 0,
    TQUEUE_BH,
    DIGI_BH,
    SERIAL_BH,
    RISCOM8_BH,
    SPECIALIX_BH,
    AURORA_BH,
    ESP_BH,
    SCSI_BH,
    IMMEDIATE_BH,
    CYCLADES_BH,
    CM206_BH,
}

```

```
JS_BH,
MACSERIAL_BH,
ISICOM_BH
};
```

- TIMER_BH는 시스템에 주기적으로 발생하는 타이머 인터럽트가 발생할 때마다 활성화된다. (mark_bh() 함수를 이용해서)
- TQUEUE_BH는 시스템에 주기적으로 발생하는 타이머 인터럽트에서, 테스크 큐인 tq_timer에 어떤 값이 들어있을 경우에만 do_timer()루틴에서 활성화된다. (mark_bh() 함수를 이용해서)
- IMMEDIATE_BH는 Immediate 큐(queue)에 들어있는 작업들을 처리하기 위한 bottom half핸들러들을 보관한다.

나중에 이러한 값들은 Bottom half핸들러를 설치할 때 사용하게 되므로 기억하고 있는 것이 중요하다. 이전 커널 버전인 2.2.X에 있던 CONSOLE_BH와 NET_BH가 없어졌다. 또한 do_bottom_half()같은 함수도 더 이상 보이지 않는다.

2.20.2. Task Queue

테스크 큐는 하고자 하는 일을 나중으로 미루어 둘 때 사용된다. bottom half와 비슷한 역할을 하지만, bottom half가 32개까지의 한계를 가지는 반면 테스크 큐는 연결 리스트로 구성되어 사용된다. 아래의 데이터 구조를 가지고 테스크 큐를 정의 한다.

```
struct tq_struct {
    struct tq_struct *next;           /* linked list of active bh's */
    unsigned long sync;              /* must be initialized to zero */
    void (*routine)(void *);         /* function to call */
    void *data;                      /* argument to function */
};

typedef struct tq_struct * task_queue;
#define DECLARE_TASK_QUEUE(q) task_queue q = NULL
extern task_queue tq_timer, tq_immediate, tq_scheduler, tq_disk;
```

큐의 구조는 [그림19]와 같다. task_queue라는 것을 head로 두고, 나머지 실행할 루틴(routine)과 그 루틴으로 넘겨 주어야 할 데이터 값을 자료 구조로 가진다. 테스크 큐의 종류로는 tq_timer와 tq_immediate, tq_scheduler, tq_disk등이 있다. 각각의 역할은 다음과 같다.

- tq_timer : 이 큐는 다음 시스템 클럭 틱 인터럽트가 발생할 때 수행되어야 할 작업들을 넣어두는 큐이다. 각각의 시스템 클럭 틱 인터럽트가 발생할 때 do_timer()에서 tq_timer를 검사해서 만약 이곳에 어떤 작업이 있다고 판단되면 bottom half를 스케줄링 해준다.(mark_bh(TQUEUE_BH)로 이러한 작업을 한다.) 처리되는 시점은 ~/kernel/timer.c에서 tqueue_bh()에서 처리된다.
- tq_immediate : 이 큐는 스케줄러(scheduler)가 현재 활성화된 bottom half를 처리할 때 같이 처리된다. 우선 순위에 있어서, tq_timer보다는 아래에 있으며, 즉각적인 처리를 요하는 작업을 큐잉(queuing)할 때 사용한다.
- tq_scheduler : 이 큐는 스케줄러가 직접적으로 처리해 준다. schedule()함수에서 tq_scheduler 큐를 확인하고 만약 비어있지 않다면, 즉각 적으로 run_task_queue()를 호출해서 처리한다. 주로 시스템에 있는 다른 작업 큐를 지원할 목적을 하지고 사용되며, 대체로 디바이스 드라이버와 같은 것의 테스크 큐를 처리한다.
- tq_disk : 이 큐는 주로 디스크의 입출력과 요구의 처리를 위해서 사용된다.

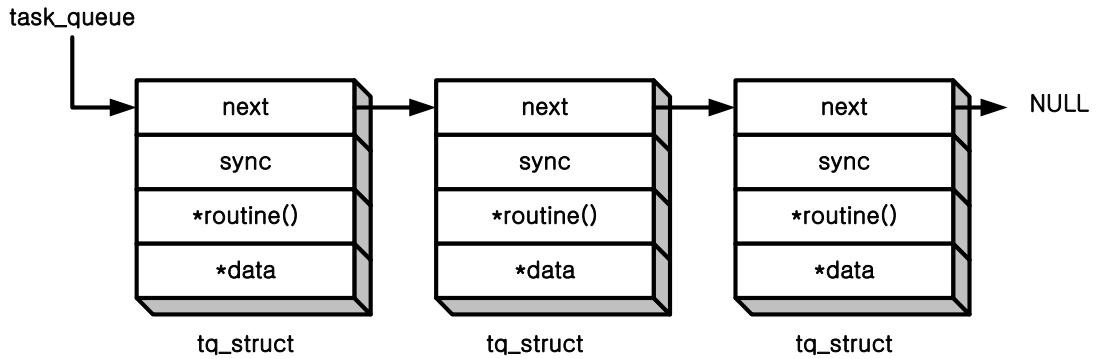


그림 22. Task Queue의 구조.

큐를 처리하는 함수들을 보도록 하자. 처리는 run_task_queue() 함수에서 일어나며, 작업을 큐에 저장하는 것은 queue_task()가 맡는다. run_task_queue()를 실행하는 시점은 ~/kernel/sched.c의 schedule()에서 tq_scheduler가 설정된 경우와 ~/kernel/timer.c의 tqueue_bh(), immediate_bh()이다. 각각의 경우에 대해서 매개변수로 관련된 테스크 큐를 넘겨 준다.

```

extern spinlock_t tqueue_lock;
static inline void queue_task(struct tq_struct *bh_pointer,
                             task_queue *bh_list)
{
    if (!test_and_set_bit(0, &bh_pointer->sync)) {
        unsigned long flags;
        spin_lock_irqsave(&tqueue_lock, flags);
        bh_pointer->next = *bh_list;
        *bh_list = bh_pointer;
        spin_unlock_irqrestore(&tqueue_lock, flags);
    }
}
static inline void run_task_queue(task_queue *list)
{
    if (*list) {
        unsigned long flags;
        struct tq_struct *p;

        spin_lock_irqsave(&tqueue_lock, flags);
        p = *list;
        *list = NULL;
        spin_unlock_irqrestore(&tqueue_lock, flags);

        while (p) {
            void *arg;
            void (*f)(void *);
            struct tq_struct *save_p;
            arg = p -> data;
            f = p -> routine;
            save_p = p;
            p = p -> next;
            smp_mb();
            save_p -> sync = 0;
            if (f)
                (*f)(arg);
        }
    }
}
  
```

{

코드 175. Task Queue를 다루는 함수들.

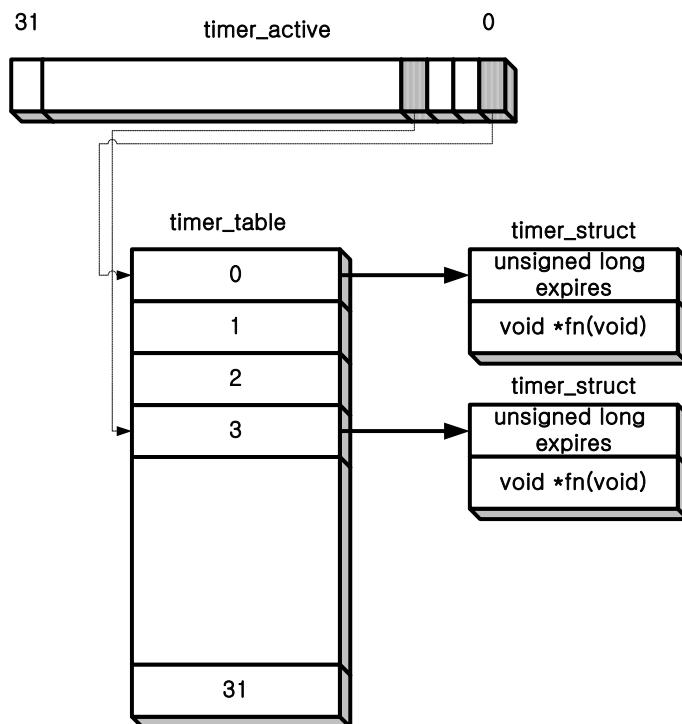
`queue_task()`은 테스크 큐에 테스크를 집어넣는 일을 하며, `run_task_queue()`은 이미 만들어져 있는 테스크 큐에서 작업들을 하나씩 꺼집어내서 실행하는 역할을 한다. 둘 다 리스트의 연산에서 인터럽트가 발생하지 않도록 한다.

코드를 보면 어떤 부분에서도 테스크 큐에 등록된 작업이나 데이터들을 해제해 주지 않는 것을 볼 수 있다. 따라서, 프로그램머가 나중에 이 부분을 처리해 주어야 한다. 위에서 나열한 테스크 큐 이외에도 사용자의 의도에 따라서 새로운 테스크 큐를 정의하는 것은 가능하며, 원하는 목적에 맞도록 코딩하면 될 것이다. 참고로 `tq_disk`는 블록 디바이스의 버퍼(buffer)연산에서 사용되며, 이와 관련된 `code`는 파일 시스템과 메모리 관리쪽에서 살펴보기 바란다.

2.20.3. Timer

타이머는 얼마 만큼의 시간이 흐른 뒤에 주기적으로 혹은 한번만 수행되기를 원할 때 사용한다. 커널에서 일어나는 일들은 시간에 관련된 작업들이 많은데, 이런 작업들을 지원하기 위해선 타이머가 꼭 필요할 것이다. 리눅스에서의 시간은 jiffies에 의해서 표현된다. 한번의 clock tick 인터럽트가 발생될 때마다 jiffies는 하나씩 증가하게 된다. 따라서, 주기적으로 clock tick 인터럽트가 발생하도록 해놓게 되면 현재 시스템의 시간을 그것에 관련 지어서 알 수 있게 된다.

커널 버전 2.2.X에서는 두 가지 종류의 타이머를 두었으나, 2.4.X버전의 커널에서는 이를 합쳐서 하나로 두고 있다. 먼저 커널 2.2.X버전에서의 타이머를 [그림20]에서 확인해 보도록 하자. 버전 2.2.X 커널에는 `timer_table`을 이용한 타이머와 `timer_list`을 이용한 타이머 두 가지가 있다. `time_table`을 이용한 것은 32개의 엔트리(entry)를 가지고 각각이 하나의 `timer_struct`를 가리키도록 해주는 방법이다. 이는 한정된 수의 타이머를 가지고 있으며, 각각의 엔트리가 유효한지를 점검하기 위해선 `timer_active`값을 비트(bit)단위로 검사해야 한다. 이와 같은 검사 및 타이머를 실행하는 것은 타이머 bottom half에서(~/kernel/timer.c의 `timer_bh()`함수) `run_old_timer()`를 수행하면서 된다.

**그림 23. Timer Table의 구조**

[그림21]은 timer_list를 사용한 타이머의 구현을 나타낸 것이다. [그림21]에서 보듯이 각각의 타이머는 양방향으로 연결된 구조를 가지며, add_timer()와 del_timer()와 같은 함수로 타이머에 관련된 작업들을 추가 혹은 삭제 할 수 있다. 타이머의 수행은 타이머 bottom half의(timer_bh()함수)에서 run_timer_list()를 호출하는 것이다.

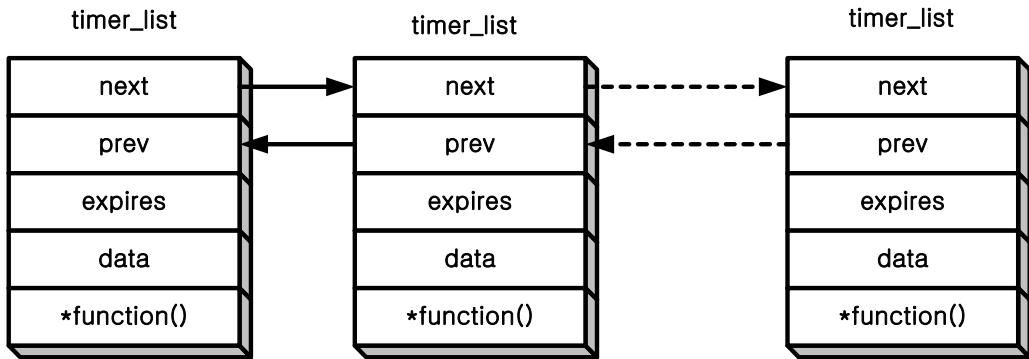


그림 24. Timer List의 구조

timer_list는 실제로 timer_vec구조를 통해서 전체적인 연결을 유지하게 되며, 각각의 timer_vec구조는 5개로 되어 있다. 각각의 timer_vec구조체에는 64개와 256개까지의 두 가지 크기를 가지는 timer_list들이 들어갈 수 있다. timer_list구조체가 각각의 timer_vec에 들어가기 위해서는 먼저 인덱스(index) 값을 jiffies와 expires값을 이용해서 구하게 되며, 구한 index값에 맞게 timer_vec의 timer_list에 들어가게 된다.

2.4 버전의 커널에서는 이전 버전에서 보이는 timer_table이 없어졌다. 모든 타이머는 timer_list와 timer_vec로 관리된다. timer_bh()를 보더라도 그곳에서 보이던 run_old_timer()부분이 사라졌음을 확인할 수 있을 것이다. 하지만, run_timer_list()는 여전히 남아 있다.

2.20.4. Wait Queue

대기 큐(wait queue)는 프로세스가 자신이 사용하고 싶은 자원이 이미 점유되어 있을 경우에 그 자원을 기다리기 위해서 잠시 기다릴 수 있는 환경을 제공한다. 즉, 대기상태로 프로세스의 상태를 바꾸게 된다. 나중에 이 대기 큐에 있는 프로세스는 자원이 사용 가능하게 되었을 때 다시 깨어나게 되며, 또다시 그 자원이 사용 가능한지를 확인하고, 만약 사용할 수 있다면 점유하게 되며, 그렇지 못할 경우에는 예외를 돌려주거나(return) 다시 대기 큐에서 대기상태로 있게 된다.

대기 중인 프로세스는 TASK_INTERRUPTIBLE이나 TASK_UNINTERRUPTIBLE의 상태가 되며, TASK_INTERRUPTIBLE인 경우에는 타이머 만료나 시그널(signal), 혹은 이벤트(event)등에 의해서 인터럽트(대기 상태에서 나오게 됨)가 될 수 있다. TASK_UNINTERRUPTIBLE인 경우에는 그러한 일이 허용되지 않는다. 반드시 다음 번에 프로세스가 정상적인 동작을 하기 위해서는 반드시 그 자원이 사용 가능해야 할 것이다.

대기 큐에 있는 프로세스들이 대기 큐가 처리가 되면 다시 실행가능 상태(TASK_RUNNING)로 바뀌며, 만약 실행 큐(run queue)에서 제거된 프로세스라면 다시 실행 큐에 삽입된다. 나중에 스케줄러가 다시 그 프로세스를 선택해서 실행을 해주게 되며, 이때 프로세스는 자신이 자원을 획득했는지 아니면, 여전히 그 자원이 점유 중인지를 알 수 있게 된다. 이러한 대기 큐는 시스템이 프로세스들을 자원에 대해서 동기적으로(synchronous) 접근할 수 있는 방법을 제공해 주기에 리눅스 상에서 세마포어 등을 구현하는데 사용될 수 있다. [그림22]는 대기 큐의 구성을 보여준다.

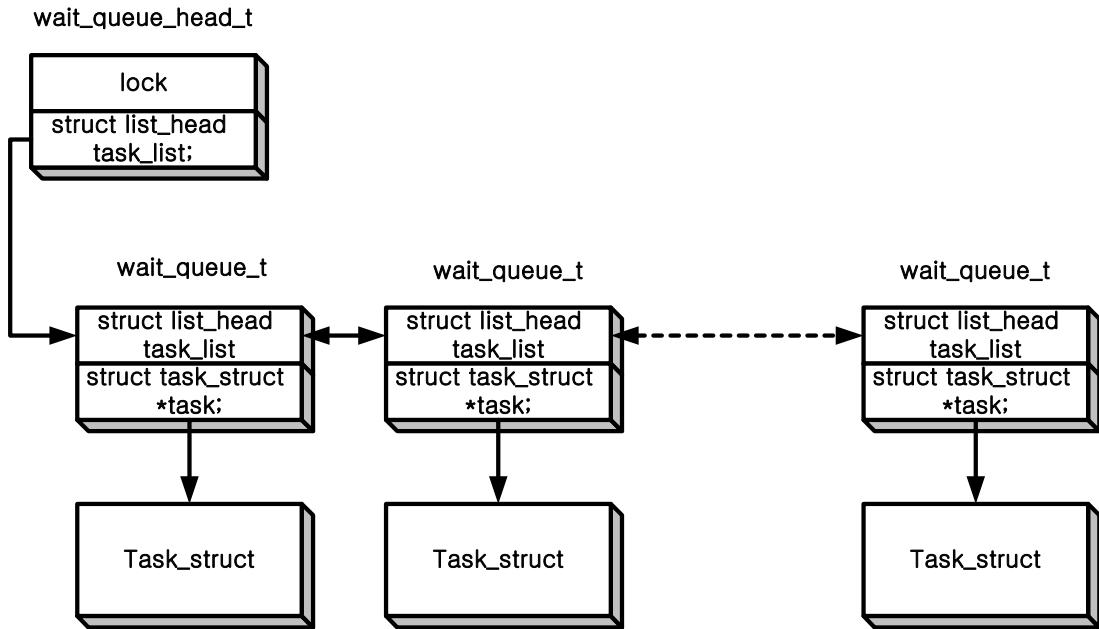


그림 25. Wait Queue의 구조

대기 큐와 관련된 연산으로 커널에서 제공하고 있는 함수로는 `add_wait_queue()`, `add_wait_queue_exclusive()`, `remove_wait_queue()` 등이 있다. 앞의 두 함수간의 차이는 대기 큐의 꼬리(tail)에 삽입 하지 않는가와 삽입하는가이다.

대기 큐와 연관된 스케줄링 쪽에서 동작하는 코드를 잠시 보도록 하자. 여기서 보여줄 것은 어떻게 프로세스가 대기 큐에 삽입되고 제거 되는지에 대한 것이다. 예제로 보여주는 코드는 `~/kernel/sched.c`이다.

```

#define SLEEP_ON_VAR \
    unsigned long flags; \
    wait_queue_t wait; \
    init_waitqueue_entry(&wait, current); \
#define SLEEP_ON_HEAD \
    wq_write_lock_irqsave(&q->lock,flags); \
    __add_wait_queue(q, &wait); \
    wq_write_unlock(&q->lock); \
#define SLEEP_ON_TAIL \
    wq_write_lock_irq(&q->lock); \
    __remove_wait_queue(q, &wait); \
    wq_write_unlock_irqrestore(&q->lock,flags);
  
```

먼저 대기 큐를 사용하기 위해서는 선언을 해야 할 것이다. 이는 `SLEEP_ON_VAR`을 사용해서 현재의 프로세스를 가지고 대기 큐를 초기화 한다. `SLEEP_ON_HEAD`는 대기 큐에 하나의 대기 작업을 삽입하며, `SLEEP_ON_TAIL`은 대기 큐에서 대기 작업을 하나를 제거한다.

```

void interruptible_sleep_on(wait_queue_head_t *q)
{
    SLEEP_ON_VAR

    current->state = TASK_INTERRUPTIBLE;
    SLEEP_ON_HEAD
    schedule();
    SLEEP_ON_TAIL
}
  
```

```

long interruptible_sleep_on_timeout(wait_queue_head_t *q, long timeout)
{
    SLEEP_ON_VAR

    current->state = TASK_INTERRUPTIBLE;
    SLEEP_ON_HEAD
    timeout = schedule_timeout(timeout);
    SLEEP_ON_TAIL
    return timeout;
}
void sleep_on(wait_queue_head_t *q)
{
    SLEEP_ON_VAR

    current->state = TASK_UNINTERRUPTIBLE;
    SLEEP_ON_HEAD
    schedule();
    SLEEP_ON_TAIL
}
long sleep_on_timeout(wait_queue_head_t *q, long timeout)
{
    SLEEP_ON_VAR

    current->state = TASK_UNINTERRUPTIBLE;
    SLEEP_ON_HEAD
    timeout = schedule_timeout(timeout);
    SLEEP_ON_TAIL
    return timeout;
}

```

코드 176. Wait Queue의 사용법.

`interruptible_sleep_on()`을 호출하면 프로세스의 현재 상태를 `TASK_INTERRUPTIBLE`로 바꾸고, 대기 큐에 작업을 삽입한다. 그리고 나서 스케줄러를 호출해서 다른 작업의 수행을 계속하도록 요청한다. 나중에 다시 이 작업이 대기 큐에서 수행을 재개하게 되면, 상태는 `TASK_RUNNING`으로 바뀌어 있을 것이며, 가장 먼저 해주는 일이 작업을 대기 큐에서 제거하는 일이다. 나머지 `interruptible_sleep_on_timeout()`과 `sleep_on()`, `sleep_on_timeout()` 등도 비슷한 일을 하게 되며, 차이점은 타이머를 가지는가와 `interruptible`인지 아닌지의 차이일 것이다.

2.21. Tasklet, Bottom Half, Software IRQ and Task Queue

앞에서 이미 bottom half와 task queue 및 timer에 대한 것들을 보았다. 여기서 보고 싶은 것은 2.4 버전에 대한 것이다. 중복된 부분이 있기는 하지만, 어쨌든 의미 있는 일이라고 보고 이야기 하도록 하겠다. Linux 2.4 버전의 커널에서 두드러진 부분으로 tasklet이라는 개념이 보인다. 이것은 일종의 delayed execution을 지원하는 메커니즘으로 그전에 있던 bottom half의 기능을 포함하며, 기타 다른 목적으로 사용하기 위해서 고안되었다. Tasklet이란 task의 작은 조각이다. 즉, task의 일부로서 실행되어지는 하나의 함수(function)이다. 이러한 것의 예로는 network driver의 구현이 가장 대표적인 경우이다. 물론 이외에도 많은 부분이 있지만, network driver에서 데이터 packet을 받아서 처리하는 것이 좋은 예가 될 것이다.

Network driver는 자신이 등록하는 ISR(Interrupt Service Routine)은 크게 두 가지의 일을 한다. 그 한 가지가 바로 데이터 packet을 받는 것이며, 다른 한 가지가 Tx에서 사용된 소켓 버퍼(sk_buff)를 처리하는 일이다. Tx에서 사용된 소켓 버퍼는 Tx의 상태에 따라서 에러인지 아닌지를 구분하는 것과 사용된 소켓 버퍼를 해제하는 일이다. 물론 이 과정에서 Tx queue를 관리하는 일도 포함한다. 그럼, Rx시에는 어떤 일이 발생할까? Rx에서는 받은 데이터를 소켓 버퍼의 형태로 변환하고, 이를 상위의 protocol 모듈에게 알려주는 일이다. 즉, ISR 내에서 모든 protocol에서 해야 할 일을 처리하기보다는 소켓 버퍼를

전달하고, 이를 알려주는 메커니즘을 사용한다. 이때가 바로 bottom half³⁷를 사용하게 되는 가장 대표적인 경우가 된다. 즉, top half(ISR)은 interrupt가 disable된 상황에서 동작하기에, 가능하면 빨리 수행되어야 한다. 따라서, 이러한 top half에서는 protocol과 관련된 수행을 할 수 없으므로, 이를 delay시켜서 적어도 사용자 process가 수행되기전에 protocol layer가 수행되도록 만들어 준다. 이것은 우선 순위의 문제로서 가장 빠르게 수행되어야 하는 것은 ISR이며, 그보다 느린 부분은 커널 내에서 수행되는 일이며, 그보다 우선 순위가 더 낮을 수 있는 것이 바로 사용자 process라는 것이다.

Lower Priority : Non-critical

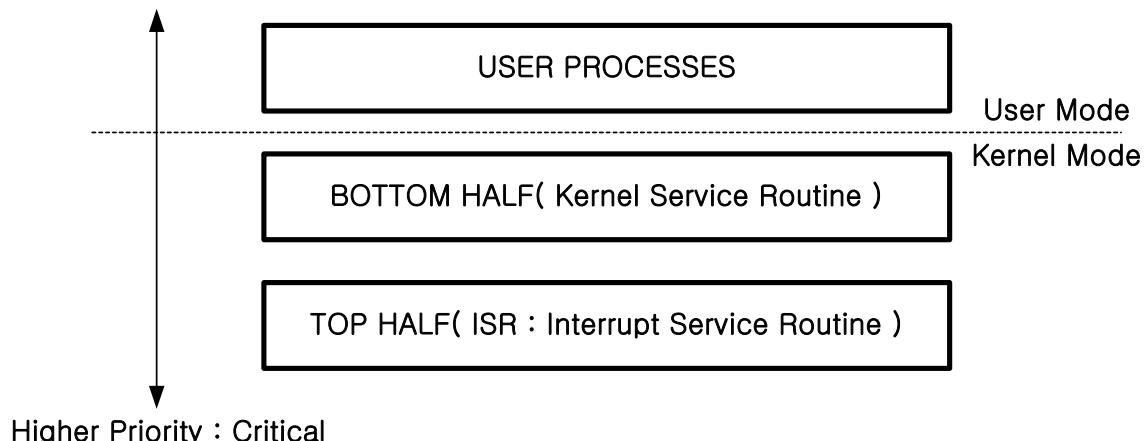


그림 26. Top Half와 Bottom Half

그럼 이렇게 함으로써 얻는 이점은 무엇인가? 즉, 이점이 없다면 이렇게 할 이유가 없는 것이다. 여기서 생기는 이점이라고 할 수 있는 부분이 바로 데이터의 throughput이라고 할 수 있겠다. 즉, 처리할 수 있는 interrupt의 횟수가 늘어나기에(즉, interrupt가 disable되어있는 순간이 짧기 때문에) 외부로 부터 발생하는 event에 대해서 놓치지 않고, 처리해 줄 수 있다는 것이다.

하지만, 유념해야 할 점이 한가지 있다. 즉, bottom half에서 수행중인 코드는 interrupt의 일환으로 수행되기에 현재 blocking되어 있는 사용자 process가 무엇인가에 가정을 해서는 안된다는 점과 sleeping과 같은 일을 수행해서는 안되며, 또한 scheduler(schedule())도 호출해서는 안된다. 단지 scheduling이 필요하면, need_reschedule과 같은 값을 이용해서 나중에 커널 모드에서 사용자 모드를 진입할 때 스케줄링이 발생하도록 만들어야 한다.

Linux 커널 2.4 이전의 버전에서는 bottom half에 관한 것만이 보인다. 하지만, Linux 커널 2.4버전 이후에서는 이전 bottom half를 좀더 일반화된 개념으로 생각하기 시작해서 모든 delay된 operation들을 처리하기 위한 tasklet이라는 개념이 도입되었다.

이번 장에서는 이러한 tasklet에 대해서 좀더 자세히 알아보도록 하자. 자주 실행되지만, 제대로 본적은 없는 것이며, 자칫하면 혼동할 수 있는, Linux 커널의 중요한 mechanism이기 때문이다.

2.21.1. Tasklet의 데이터 구조

```
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

³⁷ Bottom half를 수행하는 도중에는 interrupt가 enable되어 있으므로, ISR을 수행할 수 있다.

코드 177. tasklet_struct 구조체의 정의

Tasklet의 데이터 구조는 ~/include/linux/interrupt.h에 있는 tasklet_struct로 정의된다. tasklet_struct를 보면 알 수 있듯이 tasklet_struct는 연결 리스트로 관리되며, 상태(state)와 싱크(sync)를 나타내는 count, 그리고 task로서 수행될 함수(function)과 그 함수가 사용할 데이터(혹은 포인터 : unsigned long으로 정의되어 있다.)를 포함한다.

```
enum
{
    TASKLET_STATE_SCHED,           /* Tasklet is scheduled for execution */
    TASKLET_STATE_RUN             /* Tasklet is running (SMP only) */
};
```

코드 178. Tasklet의 상태를 나타내는 값

다시 tasklet_struct구조체의 state 필드는 현재 tasklet이 이미 실행되기 위해서 scheduling이 되었다면, TASKLET_STATE_SCHED를 가지게 되며, 특정 CPU(SMP : Symmetric Multi-Process 상에서)에서 이미 실행중인 경우에는 TASKLET_STATE_RUN이라는 값을 가지게 된다.

```
#define DECLARE_TASKLET(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }

#define DECLARE_TASKLET_DISABLED(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data }
```

코드 179. Tasklet을 선언하는 macro의 정의

위와 같은 tasklet은 먼저 사용되기 전에 초기화 되어야 한다. 이것을 돋기 위해서 DECLARE_TASKLET() 매크로와 DECLARE_TASKLET_DISABLED() 매크로가 정의되어 있다. 둘다 tasklet_struct 구조체의 필드를 채워주는 일을 하며, 두번째에 있는 매크로는 count값을 1로 만들어서 disabled(실행불가)로 초기 설정을 하도록 만든다.

```
struct tasklet_head
{
    struct tasklet_struct *list;
} __attribute__ ((aligned_(SMP_CACHE_BYTES)));

extern struct tasklet_head tasklet_vec[NR_CPUS];
extern struct tasklet_head tasklet_hi_vec[NR_CPUS];
...
extern struct tasklet_struct bh_task_vec[];
```

코드 180. tasklet_vec[]와 tasklet_hi_vec[] 및 bh_task_vec[] 배열의 정의

tasklet의 연결리스트를 유지하기 위해서는 tasklet_head라는 구조체를 사용한다. 또한 tasklet_head 구조체의 배열을 가지는 tasklet_vec[]와 tasklet_hi_vec[]가 전역 변수로서 정의되어 있다. 즉, 이 두개의 변수를 통해서 모든 tasklet에 대한 연산이 발생할 것이다. 또한, bottom half의 구현을 tasklet으로 하기위해서 bh_task_vec[]라는 배열도 정의하고 있다.

```
enum
{
    HI_SOFTIRQ=0,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    TASKLET_SOFTIRQ
};
```

코드 181. Software IRQ의 값 정의

tasklet_vec[]와 tasklet_hi_vec[] 배열은 각각 TASKLET_SOFTIRQ와 HI_SOFTIRQ에 해당하는 tasklet들의 header를 가지는 배열로 정의된다. NET_TX_SOFTIRQ와 NET_RX_SOFTIRQ는 각각 network에 관련해서 Tx쪽과 Rx쪽에서의 bottom half의 처리를 담당하는 값으로 사용된다.

앞에서 정의된 것들은 모두 커널에서 software IRQ를 담당하는 ~/linux/kernel/softirq.c 파일에서 사용하게 되며, 이제부터는 이 파일에 대한 이야기를 하도록 하겠다. Tasklet의 처리를 담당하는 kernel thread인 ksoftirqd(Kernel Software IRQ daemon)부터 보기로 하자³⁸.

2.21.2. Kernel Software IRQ Daemon

이 커널 thread는 커널이 초기화 되면서 생성된다. 생성을 담당하고 있는 코드는 아래와 같이 kernel_thread() 함수를 사용해서 생성한다.

```
static __init int spawn_ksoftirqd(void)
{
    int cpu;

    for (cpu = 0; cpu < smp_num_cpus; cpu++) {
        if (kernel_thread(ksoftirqd, (void *) (long) cpu,
                          CLONE_FS | CLONE_FILES | CLONE_SIGNAL) < 0)
            printk("spawn_ksoftirqd() failed for cpu %d\n", cpu);
        else {
            while (!ksoftirqd_task(cpu_logical_map(cpu))) {
                current->policy |= SCHED_YIELD;
                schedule();
            }
        }
    }
    return 0;
}
__initcall(spawn_ksoftirqd);
```

코드 182. spawn_ksoftirqd() 함수의 정의

__initcall() 매크로는 spawn_ksoftirqd() 함수가 커널이 초기화 될 때(do_initcall()함수에서), 호출되도록 만들어준다. 즉, 이 함수는 커널이 가장 먼저 생성하는 init 커널 thread에서 만들어 지므로, init 커널 thread의 특성들을 그대로 사용할 것이다. kernel_thread() 함수에서 인자로 사용한 CLONE_FS, CLONE_FILES, CLONE_SIGNAL에 따라, init 커널 thread의 file system 관련 구조체와 file 관련 구조체 및 signal 등을 그대로 상속할 것이다. 만약 생성에 실패한다면, 그 CPU에 대해서 현재 진행중인 process의 상태를 SCHED_YIELD(CPU를 release해서 다른 프로세스에게 실행될 기회를 제공한다.)바꾸고 나서 schedule() 함수를 호출해 다른 process가 실행될 수 있는 기회를 주도록 한다. ksoftirqd_task() 매크로는 특정 CPU의 번호로 부터 그 CPU에서 실행중인 ksoftirqd의 task_struct 구조체의 포인터를 돌려준다. 즉, while() loop를 돌면서 ksoftirqd가 생성될 때까지 기다리도록 만드는 것이다. 이러한 일은 전체 시스템에 있는 CPU의 개수 만큼 반복적으로 수행된다.

```
static int ksoftirqd(void * __bind_cpu)
{
    int bind_cpu = (int) (long) __bind_cpu;
    int cpu = cpu_logical_map(bind_cpu);

    daemonize();
    current->nice = 19;
```

³⁸ 물론 software IRQ의 자료구조를 초기화 하는 곳은 softirq_init() 함수에서 처리한다.

```

sigfillset(&current->blocked);
/* Migrate to the right CPU */
current->cpus_allowed = 1UL << cpu;
while (smp_processor_id() != cpu)
    schedule();
sprintf(current->comm, "ksoftirqd_CPU%d", bind_cpu);
__set_current_state(TASK_INTERRUPTIBLE);
mb();
ksoftirqd_task(cpu) = current;
for (;;) {
    if (!softirq_pending(cpu))
        schedule();
    __set_current_state(TASK_RUNNING);

    while (softirq_pending(cpu)) {
        do_softirq();
        if (current->need_resched)
            schedule();
    }
    __set_current_state(TASK_INTERRUPTIBLE);
}
}

```

코드 183. ksoftirqd() 함수의 정의

자, 그럼 이젠 커널 software IRQ daemon에 대해서 보기로 하자. ksoftirqd() 함수이다. __bind_cpu는 앞에서 커널 thread를 생성하면서 주어진 CPU의 번호이며, 현재 이 daemon이 실행되고 있는 CPU를 나타낸다. cpu_logical_map() 매크로는 i386 계열의 경우에는 그냥 인자값을 그대로 돌려주는 값이며, sparc과 같은 경우에는 논리적으로 mapping된 값을 돌려준다.

daemonize() 함수는 자신이 사용하는 시스템의 자원을 다시 커널에 돌려주고, 프로세스를 session 번호 1과 group 번호 1로 설정하게 되며, console를 프로세스로 부터 떼어내고, init process가 가진 자원을 공유해서 사용하도록 만들어주는 일을 한다. 즉, 이 함수를 호출한 이후에는 하나의 backgroup로 동작하는 프로세스로 변화하게 된다. Nice값은 scheduling에 관련된 값으로 우선순위의 계산에 합(addition)으로 들어가게 되는데, 이 값을 19로 설정했다. Nice값은 -20에서 +19까지를 Linux에서 사용하며, -20이 우선순위가 제일 높다는 표현이며, 19로 설정했으므로 다른 프로세스보다는 우선 순위가 낮게 만들어 주고 있음을 알 수 있다. sigfillset() 함수는 프로세스의 signal set을 전체 1로 설정하는 일을 해주며, 여기서는 blocking되어야 할 signal(current->blocked)를 전체 선택해서 signal을 받지 않도록 만들어 주었다.

프로세스가 실행이 허가된(cpus_allowed) CPU의 값을 표시하고, 이젠 이 프로세스를 실행하고 있는 CPU가 실행될 기회를 얻도록 만들기 위해서 while() loop를 돌면서 schedule() 함수를 호출한다. comm은 command line을 나타내는 task_struct의 필드로서 프로그램의 이름을 나타내는 값이다. 이곳에 현재 실행중인 프로세스의 이름과 할당된 CPU번호를 넣느다. 즉, 나중에 ps와 같은 명령어를 사용해서 실행되고 있는 상황을 보기 위한 것이다. __set_current_state() 함수는 프로세스의 상태를 바꾸는 것으로 TASK_INTERRUPTIBLE로 만들어주었다. mb()는 단지 동기화를 위한 방법으로 사용되는 것으로 정확히 수행순서를 맞추기 위해서 사용했다. 이젠 현재의 process(= ksoftirqd 커널 thread)를 CPU의 softirq task로 저장하고(ksoftirqd_task()), 무한 loop를 돌면서 pending된 software IRQ가 있는지를 확인한다(softirq_pending()). 만약 대기하고 있는 software IRQ가 없다면, 곧바로 schedule() 함수를 호출해서 다른 프로세스에게 CPU를 놓아주고, 그렇지 않다면 __set_current_state() 매크로를 호출해서 ksoftirqd 커널 thread를 실행 상태로 바꾼다(TASK_RUNNING). 이젠 pending된 software IRQ가 있는 동안 do_softirq() 함수를 호출해서 처리하는 일만 남았다. 만약 software IRQ를 처리하는 도중에 need_resched 필드가 설정된 경우에는 schedule()를 호출해서 IRQ 처리의 결과가 곧바로 다른 process에게 전달될 수 있도록 해준다. while() loop를 마치면 이젠 다시 ksoftirqd 커널 thread의 상태를 TASK_INTERRUPTIBLE로 바꿔서 어떤 event가 발생하기를 기다리게 된다.

2.21.3. Tasklet의 초기화

```
void __init softirq_init()
{
    int i;

    for (i=0; i<32; i++)
        tasklet_init(bh_task_vec+i, bh_action, i);

    open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
    open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
}
```

코드 184. `softirq_init()` 함수의 정의

앞에서 정의한 `tasklet_head` 구조체들에 대한 초기화는 `softirq_init()` 함수를 통해서 이루어진다. 이 함수는 `start_kernel()` 함수가 호출해 주는 것으로 `bh_task_vec[]`를 초기화 하며, `tasklet`의 `action`³⁹에 대한 정의를 가지는 `softirq_vec[]` 배열의 `TASKLET_SOFTIRQ`와 `HI_SOFTIRQ` index를 가지는 `entry`의 `action`을 초기화 한다.

```
void open_softirq(int nr, void (*action)(struct softirq_action*), void *data)
{
    softirq_vec[nr].data = data;
    softirq_vec[nr].action = action;
}
```

코드 185. `open_softirq()` 함수의 정의

`open_softirq()` 함수는 `softirq_vec[]` 배열에 대한 index값과 취해야 할 `action`(`struct softirq_action`)에 대한 포인터와 그 함수가 수행될 때 넘겨줄 데이터에 대한 포인터를 인자로 받는다. 간단히 `softirq_vec[]`의 각 필드에 해당 값을 넣는 것으로 충분하다. 이렇게 정해진 `action`은 각 software IRQ 번호에 맞는 일이 발생될 때 호출될 것이다.

```
struct softirq_action
{
    void     (*action)(struct softirq_action *);
    void     *data;
};
```

코드 186. `softirq_action` 구조체의 정의

여기서 잠시 앞으로 계속 보게될 `softirq_action` 구조체에 대해서 보고 넘어가도록 하자. 특이한 점은 `softirq`에서 처리할 `action`을 `action` 함수의 인자값으로 가진다는 점이다. 자, 그럼 이렇게 정의된 `softirq_action` 구조체로 앞에서 `open_softirq()`에서 사용한 것들을 보기로 하자. 각각은 `tasklet_action`과 `tasklet_hi_action`이며, `data` 필드는 둘다 `NULL`을 주어 넘겨받는 인자가 없음을 나타냈다.

```
static void tasklet_action(struct softirq_action *a)
{
    int cpu = smp_processor_id();
    struct tasklet_struct *list;

    local_irq_disable();
    list = tasklet_vec[cpu].list;
```

³⁹ Tasklet이 수행해야 하는 함수라고 생각하도록 하자.

```

tasklet_vec[cpu].list = NULL;
local_irq_enable();

while (list) {
    struct tasklet_struct *t = list;
    list = list->next;
    if (tasklet_trylock(t)) {
        if (!atomic_read(&t->count)) {
            if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
                BUG();
            t->func(t->data);
            tasklet_unlock(t);
            continue;
        }
        tasklet_unlock(t);
    }
    local_irq_disable();
    t->next = tasklet_vec[cpu].list;
    tasklet_vec[cpu].list = t;
    __cpu_raise_softirq(cpu, TASKLET_SOFTIRQ);
    local_irq_enable();
}
}

```

코드 187. tasklet_action() 함수의 정의

tasklet_action() 함수는 호출되었을 때 실행중이던 CPU의 ID를 구해서 cpu에 저장한다. 그리고나서, local_irq_disable() 함수를 호출해서 현재 실행중인 CPU상에서 interrupt가 발생하지 않도록 한 후, tasklet_vec[]의 list field에 있을 tasklet들에 대한 포인터를 얻는다(list). 이것을 마치고 나면, tasklet_vec[]의 list에는 NULL로 두어 다시 tasklet 리스트를 초기화 한다. 이젠 커널 데이터 구조에 대한 연산을 마쳤으므로 다시 interrupt를 enable시키기 위해서 local_irq_enable()을 호출한다. 나머지는 이전 얻어온 tasklet들을 차례로 실행하는 일이다. while() loop가 사용된다.

tasklet_trylock() 함수는 tasklet의 상태(state)를 TASKLET_STATE_RUN으로 atomic하게 바꾸는 일을 한다. 이전 tasklet 구조체의 count를 읽어서 0인 경우에(enable된 경우 : 앞에서 disable를 1로 처리했다. - DECLARE_TASKLET_DISABLE() 매크로를 보도록 하자.)만 다시 tasklet이 이미 scheduling이 되었는가를 확인한다. 만약 이미 scheduling이 되었다면, 이는 disable된 상황에서 실행되었다는 말이되므로 BUG()을 호출해서 문제가 있음을 알려준다. 이제 tasklet을 수행하기 위해서 tasklet의 함수(func필드)에 tasklet의 데이터를 가르키는 포인터(data)를 넘겨주어서 호출한다. 이것을 마치면, 다음 tasklet으로 진행하기에 앞서 tasklet_unlock()을 호출해서 tasklet에 대한 lock을 해제한다.

하지만, 만약 tasklet에 대한 lock을 걸수 없는 경우가 발생된다면, 그 tasklet을 떼어내서(list가 이미 다음번 tasklet을 가지고 있다.), tasklet_vec[].list에 다시 연결해 둔고, 다음번 실행을 위해서 __cpu_raise_softirq()를 호출해, 현재 CPU에서 수행할 tasklet이 있음을 표시한다. 이 과정에서는 역시 커널 데이터 구조를 다루게 되므로, 실행중인 CPU에서만 interrupt가 발생하지 않도록 local_irq_disable()과 local_irq_enable()을 호출한다.

```

static void tasklet_hi_action(struct softirq_action *a)
{
    int cpu = smp_processor_id();
    struct tasklet_struct *list;

    local_irq_disable();
    list = tasklet_hi_vec[cpu].list;
    tasklet_hi_vec[cpu].list = NULL;
    local_irq_enable();
    while (list) {
        struct tasklet_struct *t = list;

```

```

list = list->next;
if (tasklet_trylock(t)) {
    if (!atomic_read(&t->count)) {
        if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
            BUG();
        t->func(t->data);
        tasklet_unlock(t);
        continue;
    }
    tasklet_unlock(t);
}
local_irq_disable();
t->next = tasklet_hi_vec[cpu].list;
tasklet_hi_vec[cpu].list = t;
__cpu_raise_softirq(cpu, HI_SOFTIRQ);
local_irq_enable();
}
}

```

코드 188. tasklet_hi_action() 함수의 정의

tasklet_hi_action() 함수도 역시 tasklet_action()과 마찬가지다. 달라지는 부분은 tasklet_vec[] 대신에 tasklet_hi_vec[]를 사용한다는 점이다. 따라서, tasklet_action()과 tasklet_hi_action()은 기능에서의 차이가 아니라, 단지 사용되는 tasklet의 종류가 무엇인가에 따라서 구분되어 진다는 것이다.

자, 다시 앞으로 돌아가서 softirq_init() 함수를 보도록 하자. 이 함수에서는 tasklet을 초기화 하기 위해서 tasklet_init() 함수를 호출하고 있다.

```

void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data)
{
    t->next = NULL;
    t->state = 0;
    atomic_set(&t->count, 0);
    t->func = func;
    t->data = data;
}

```

코드 189. tasklet_init() 함수의 정의

tasklet_init() 함수는 tasklet_struct 구조체와 tasklet으로 호출할 함수 및 함수의 인자를 넘겨받는다. 이 함수는 단지 tasklet_struct 구조체를 하나 설정하는 일을 수행한다. next 필드에는 NULL을 주고, state에는 0으로, count필드도 역시 0으로 두게되며, func와 data에 각각 수행할 tasklet과 인자로 사용할 data값을 준다.

따라서, 우린 앞에서 tasklet_init() 함수를 호출해서 32개의 bottom half entry(bh_tasklet_vec[])에 대해서 기본적으로 bh_action()이라는 함수를 설정한다는 것을 알 수 있으며, 인자로 사용할 값으로는 index값을 준다고 볼 수 있다. 자, 그럼 우리는 이제 bottom half에 대해서 보기 위해서 bh_action()이라는 함수를 알 필요가 있다.

```

static void bh_action(unsigned long nr)
{
    int cpu = smp_processor_id();

    if (!spin_trylock(&global_bh_lock))
        goto resched;

    if (!hardirq_trylock(cpu))
        goto resched_unlock;
}

```

```

if (bh_base[nr])
    bh_base[nr]();

hardirq_endlock(cpu);
spin_unlock(&global_bh_lock);
return;

resched_unlock:
    spin_unlock(&global_bh_lock);
resched:
    mark_bh(nr);
}

```

코드 190. bh_action() 함수의 정의

bh_action() 함수는 먼저 자신이 수행되는 CPU의 ID를 구한다(smp_processor_id()). 그리고, bottom half에 대한 global lock을 나타내기 위한 변수인 global_bh_lock에 lock을 설정하려고 시도한다(spin_trylock()). 만약 lock을 설정할 수 없다면, 다시 이 bottom half를 수행되도록 하기 위해서 mark_bh()를 호출하고 바로 복귀하게 된다. 만약 lock을 설정할 수 있다면, hardware IRQ를 현재 CPU에 대해서 설정할 수 있는지 시도하게된다(hardirq_trylock()). 이것은 단지 현재 수행중인 CPU에서 실행중인 IRQ가 있는지만을 확인하는 일을 한다. 만약 수행중인 IRQ가 있다면, 앞에서 설정한 global_bh_lock의 lock을 해제하고(spin_unlock()), bottom half를 marking한 후 나중에 다시 처리하도록 만들어준다. IRQ가 없다는 것을 확인하면, 이전 bottom half를 service하는 일이 남았다. 간단히 해당 bottom half의 index(= nr)에 있는 bottom half 함수를 수행하기만 하면 될 것이다(bh_base[nr]). 이것을 마치면 앞에서 설정했던 lock들을 해제한다. 여기서 bh_base[]는 bottom half로 실행할 함수들에 대한 포인터를 기록하고 있는 배열이다. 함수들은 void 값을 인자와 복귀 값으로 돌려주면 되는 것들이다.

```

void init_bh(int nr, void (*routine)(void))
{
    bh_base[nr] = routine;
    mb();
}

void remove_bh(int nr)
{
    tasklet_kill(bh_task_vec+nr);
    bh_base[nr] = NULL;
}

```

코드 191. init_bh() 함수와 remove_bh() 함수의 정의

init_bh() 함수는 앞에서 본 bh_base[] 배열에 값을 넣어주는 함수이며, remove_bh() 함수는 tasklet_struct 구조의 배열인 bh_task_vec[]에서 해당 index(= nr)에 있는 tasklet들을 제거하고(tasklet_kill()), bottom half의 해당 index에 NULL를 넣어서 bottom half로서 수행될 함수를 제거하는 일을 한다.

```

void tasklet_kill(struct tasklet_struct *t)
{
    if (in_interrupt())
        printk("Attempt to kill tasklet from interrupt\n");

    while (test_and_set_bit(TASKLET_STATE_SCHED, &t->state)) {
        current->state = TASK_RUNNING;
        do {
            current->policy |= SCHED_YIELD;
            schedule();
        }
    }
}

```

```

        } while (test_bit(TASKLET_STATE_SCHED, &t->state));
    }
tasklet_unlock_wait(t);
clear_bit(TASKLET_STATE_SCHED, &t->state);
}

```

코드 192. tasklet_kill() 함수의 정의

tasklet_kill() 함수는 현재 스케줄링된 tasklet의 수행이 끝나도록 대기하는 일을 한다. in_interrupt()는 인터럽트 내에서 이것이 호출되었는가를 알아보기 위한 것이며, 죽이려는(kill) task의 상태가 TASKLET_STATE_SCHED인 경우 동안 while() loop를 수행한다. 현재 수행중이던 process의 상태를 TASK_RUNNING으로 바꾸고, scheduling policy를 나타내는 필드에 SCHED_YIELD를 선언한 후 다시 scheduling을 요청한다(schedule()). 이렇게 한 후 TASKLET_STATE_SCHED가 tasklet의 상태 필드(state)에 아직 있다면, do {} while() loop를 순회하게 될 것이다. remove_bh() 함수가 수행되는 때는 특정 디바이스 드라이버내에서 호출된다. 따라서, 이 함수는 반드시 특정 process의 context를 지니고 있다고 볼 수 있으며, 이럴 경우 해당 프로세스의 실행을 연기시켜서 tasklet이 수행될 기회를 빨리 얻도록 만들어주는 것이다. tasklet_unlock_wait()함수는 tasklet이 TASKLET_STATE_RUNNING인 동안 기다리는 역할을 하는 inline으로 정의된 함수이며, clear_bit()을 호출해서 tasklet의 상태에서 TASKLET_STATE_SCHED를 지우도록 한다.

앞에서 설명한 init_bh() 함수는 각각의 bottom half를 필요로 하는 부분에서 호출되어 bottom half routine을 등록하는데 사용한다. 그럼 이러한 bottom half에는 어떤 것들이 있는가? ~/linux/include/linux/interrupt.h에서 아래와 같은 정의를 찾을 수 있을 것이다.

```

enum {
    TIMER_BH = 0,
    TQUEUE_BH,
    DIGI_BH,
    SERIAL_BH,
    RISCOM8_BH,
    SPECIALIX_BH,
    AURORA_BH,
    ESP_BH,
    SCSI_BH,
    IMMEDIATE_BH,
    CYCLADES_BH,
    CM206_BH,
    JS_BH,
    MACSERIAL_BH,
    ISICOM_BH
};

```

코드 193. Bottom 들에 대한 정의

이러한 bottom half에서 사용자(디바이스 드라이버 같은 것을 programming하는 사람일 것이다.)가 사용할 수 있는 bottom half에는 TQUEUE_BH와 IMMEDIATE_BH가 있으며, 이외에도 중요한 bottom half로는 TIMER_BH가 있을 것이다. 예전 버전의 커널에는 NET_BH와 같은 network bottom half 부분도 있었으나, 현재는 이러한 것이 tasklet으로 따로 정의되어 있다⁴⁰.

2.21.4. Bottom Half 설정

Bottom half를 active하게 만들어서 bottom half를 처리하도록 알려주는 것이 바로 mark_bh() 함수이다. 이 함수는 inline으로 정의된 것으로 아래와 같다.

⁴⁰ ~/net/core/dev.c를 참조하기 바란다. 이곳에 NET_TX_SOFTIRQ와 NET_RX_SOFTIRQ를 초기화 하는 부분을 찾을 수 있을 것이다. 함수의 이름은 net_dev_init()이다.

```
static inline void mark_bh(int nr)
{
    tasklet_hi_schedule(bh_task_vec+nr);
}
```

코드 194. mark_bh() 함수의 정의

mark_bh() 함수는 예전에는 bottom half를 직접적으로 active한 상태로 바꾸는 역할을 했지만, 2.4 버전의 커널에서는 이제 tasklet으로 bottom half를 구현하고 있기에 tasklet_hi_schedule()를 호출해서 bottom half tasklet이 수행되도록 scheduling을 요청하도록 처리되어 있다.

```
static inline void tasklet_hi_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
        __tasklet_hi_schedule(t);
}
```

코드 195. tasklet_hi_schedule() 함수의 정의

tasklet_hi_schedule() 함수는 tasklet_struct 구조체의 포인터를 넘겨받아서, 이 tasklet의 상태(state)를 TASKLET_STATE_SCHED로 바꾼다. 만약 이미 TASKLET_STATE_SCHED로 설정이 되어있지 않았다면, 다시 __tasklet_hi_schedule() 함수를 호출하도록 한다.

```
void __tasklet_schedule(struct tasklet_struct *t)
{
    int cpu = smp_processor_id();
    unsigned long flags;

    local_irq_save(flags);
    t->next = tasklet_vec[cpu].list;
    tasklet_vec[cpu].list = t;
    cpu_raise_softirq(cpu, TASKLET_SOFTIRQ);
    local_irq_restore(flags);
}

void __tasklet_hi_schedule(struct tasklet_struct *t)
{
    int cpu = smp_processor_id();
    unsigned long flags;

    local_irq_save(flags);
    t->next = tasklet_hi_vec[cpu].list;
    tasklet_hi_vec[cpu].list = t;
    cpu_raise_softirq(cpu, HI_SOFTIRQ);
    local_irq_restore(flags);
}
```

코드 196. __tasklet_schedule() 함수와 __tasklet_hi_schedule() 함수의 정의

__tasklet_hi_schedule() 함수와 __tasklet_schedule() 함수는 기본적으로 같은 함수이며, 다만 처리하고자 하는 tasklet_head 구조체가 어떤 것인가에 따라서만 달라진다. 먼저 CPU의 ID를 구하고(smp_processor_id()), 이 CPU에 대한 IRQ lock을 설정한 후(local_irq_save()), tasklet_hi_vec[]나 tasklet_vec[]에서 CPU의 ID값에 따른 list값을 구한 후 tasklet 연결리스트의 가장 앞부분에 tasklet_struct 구조체를 넣는 일을 한다. 마지막으로 해당 CPU에 어떤 SOFTIRQ를 처리해야 하는지를 알려주기 위해서 cpu_raise_softirq() 함수를 호출한다. 넘겨주는 인자는 CPU의 ID와 TASKLET_SOFTIRQ/HI_SOFTIRQ가 사용된다. 이것을 마치고 나면 local_irq_restore()를 호출해서

interrupt에 설정했던 lock을 풀도록 한다. 따라서, 우리는 mark_bh()를 호출해서 bottom half를 수행시키고자 하면, tasklet_hi_vec[]의 list로서 bottom half의 tasklet이 들어감을 알 수 있다.

```
#define __cpu_raise_softirq(cpu, nr) do { softirq_pending(cpu) |= 1UL << (nr); } while (0)
...
inline void cpu_raise_softirq(unsigned int cpu, unsigned int nr)
{
    __cpu_raise_softirq(cpu, nr);

    /*
     * If we're in an interrupt or bh, we're done
     * (this also catches bh-disabled code). We will
     * actually run the softirq once we return from
     * the irq or bh.
     *
     * Otherwise we wake up ksoftirqd to make sure we
     * schedule the softirq soon.
     */
    if (!(local_irq_count(cpu) | local_bh_count(cpu)))
        wakeup_softirqd(cpu);
}
```

코드 197. cpu_raise_softirq() 함수의 정의

cpu_raise_softirq() 함수는 먼저 __cpu_raise_softirq()를 호출한다. __cpu_raise_softirq()는 해당 CPU의 상태를 나타내는 정보에 software IRQ가 발생했음을 단순히 marking하는 일만한다. 만약 현재 진행중인 interrupt가 해당 CPU에 대해서 없고, 진행중인 bottom half도 없다면, ksoftirqd 커널 thread가 깨어나서 software IRQ를 처리하도록 만들어 주기 위해서 wakeup_softirqd() 함수를 호출한다. 이 함수를 호출하고나면, ksoftirqd가 결국 scheduling되어 해당 software IRQ의 처리를 맡을 것이다.

```
static inline void wakeup_softirqd(unsigned cpu)
{
    struct task_struct * tsk = ksoftirqd_task(cpu);

    if (tsk && tsk->state != TASK_RUNNING)
        wake_up_process(tsk);
}
```

코드 198. wakeup_softirqd() 함수의 정의

wakeup_softirqd() 함수는 해당 CPU의 software IRQ를 처리할 ksoftirqd 커널 데몬 thread를 깨우는 일을 한다. 단지 프로세스(task)의 상태가 TASK_RUNNING이 아닌 경우에 대해서 wake_up_process()를 호출하도록 한다. wake_up_process() 함수는 ~/linux/kernel/sched.c에 정의된 함수로서 프로세스를 깨워서 run queue에 넣는 일을 해준다. 이렇게 함으로서, 나중에 scheduler에 의해서 수행될 기회를 얻게 된다. 깨어나서 처리하는 동작을 처리하는 부분은 앞에서 보았듯이 ksoftirqd() 함수의 코드가 될 것이다. 이중에서도 특히 바로 do_softirq() 함수를 호출하는 부분이 된다. 이 do_softirq() 함수가 설정된 tasklet을 수행할 것이다.

2.21.5. Tasklet의 수행

do_softirq() 함수가 호출되는 context는 이미 말했듯이 ksoftirqd 커널 데몬 thread가 수행되는 환경이 될 것이다. 물론, 이 이외에도 local_bh_enable()이라는 매크로 내에서 직접적으로 호출이 되지만, 이는 networking과 관련된 code에서 직접적으로 해당 network bottom half를 호출하기 위한 것이다.

```
asmlinkage void do_softirq()
{
    int cpu = smp_processor_id();
```

```

__u32 pending;
long flags;
__u32 mask;

if (in_interrupt())
    return;
local_irq_save(flags);
pending = softirq_pending(cpu);
if (pending) {
    struct softirq_action *h;

    mask = ~pending;
    local_bh_disable();

restart:
    /* Reset the pending bitmask before enabling irqs */
    softirq_pending(cpu) = 0;
    local_irq_enable();
    h = softirq_vec;
    do {
        if (pending & 1)
            h->action(h);
        h++;
        pending >>= 1;
    } while (pending);
    local_irq_disable();
    pending = softirq_pending(cpu);
    if (pending & mask) {
        mask &= ~pending;
        goto restart;
    }
    __local_bh_enable();
    if (pending)
        wakeup_softirqd(cpu);
}
local_irq_restore(flags);
}

```

코드 199. do_softirq() 함수의 정의

do_softirq() 함수는 먼저 자신이 실행되고 있는 CPU의 ID를 구해서 cpu에 놓는다. Interrupt가 처리되고 있는 중이었다면(in_interrupt()), 바로 복귀하도록 한다. 이는 interrupt가 bottom half를 처리하는 동안 커널 자료 구조에 접근하는 관계로 interrupt가 없는 상황에서만 수행되도록 만들어주기 위함이다. 이전 interrupt가 없다면, 현재 실행되고 있는 CPU에서는 인터럽트를 service하지 않기 위해서 local_irq_save()를 호출해서 lock을 설정한다. 그런 후, pending된 software IRQ가 있는지를 확인하기 위해서 softirq_pending()을 실행중인 CPU에 대해서 호출한다. 이 값을 pending 변수로 놓고, mask에는 pending된 값 이외의 값들로 설정하기 위해서 ~pending으로 주었다. local_bh_disable()은 매크로로 정의된 것으로 bottom half를 실행한다고 설정하는 역할을 한다. 따라서, 현재 CPU상에서는 이미 bottom half가 실행 중이니 다른 bottom half를 실행할 일이 있으면, bottom half의 실행을 마칠 때까지 실행하지 못하도록 만들것이다. 따라서, disable시킨다고 생각하도록 하자. softirq_pending()에는 이미 값을 다 얻어왔으므로 0을 주어 다시 초기화(re-initialize)시키고, local_irq_enable()로 interrupt가 발생되도록 만든다. 이는 커널의 critical section을 보호하기 위해서 앞에서 설정했던 lock을 해제하는 일을 한다. softirq_vec는 software IRQ의 action들을 가지고 있음을 앞에서 이미 보았다. 이것을 h가 가르키게 만든 후, pending된 software IRQ가 있는 것만을 선별적으로 선택해서 호출하도록 만들어 준다.

다시 local_irq_disable()은 software IRQ를 처리하는 동안에 생겼을 지도 모를 pending된 software IRQ를 가져오기 위해서 설정했다. pending된 software IRQ가 있다면, 이 값을 앞에서 이미 service 했을지도 모르기에 mask값과 AND시켜서, 앞에서 처리한 software IRQ가 없을 경우에만 restart로 돌아가서

pending된 software IRQ를 처리하도록 해준다. 이 경우에는 다시 mask값도 경신될 것이다. 만약 pending된 IRQ가 없다면, `_local_bh_enable()`을 호출해서 bottom half가 수행을 다 했으므로, 다른 bottom half들을 실행할 수 있도록 만들어준다. 만약 앞에서 이미 처리한 pending된 software IRQ가 다시 다른 software IRQ를 처리하는 동안 발생했다면, `wakeup_softirqd()`를 호출해서 다음번 ksoftirqd 커널 데몬 thread를 스케줄링 하도록 `wakeup_softirqd()`를 호출하도록 한다. 마지막으로 이전 local interrupt를 다시 발생 가능하도록 만들어 주기 위해서 `local_irq_restore()`를 호출한다. 즉, 이전에 저장했던 interrupt와 관련된 flag값들을 복구하는 것이다.

이젠 지금까지 보았던 tasklet과 bottom half 및 software IRQ에 대한 정리를 좀 하기로 하겠다. 먼저 [그림 27]을 보기로 하자.

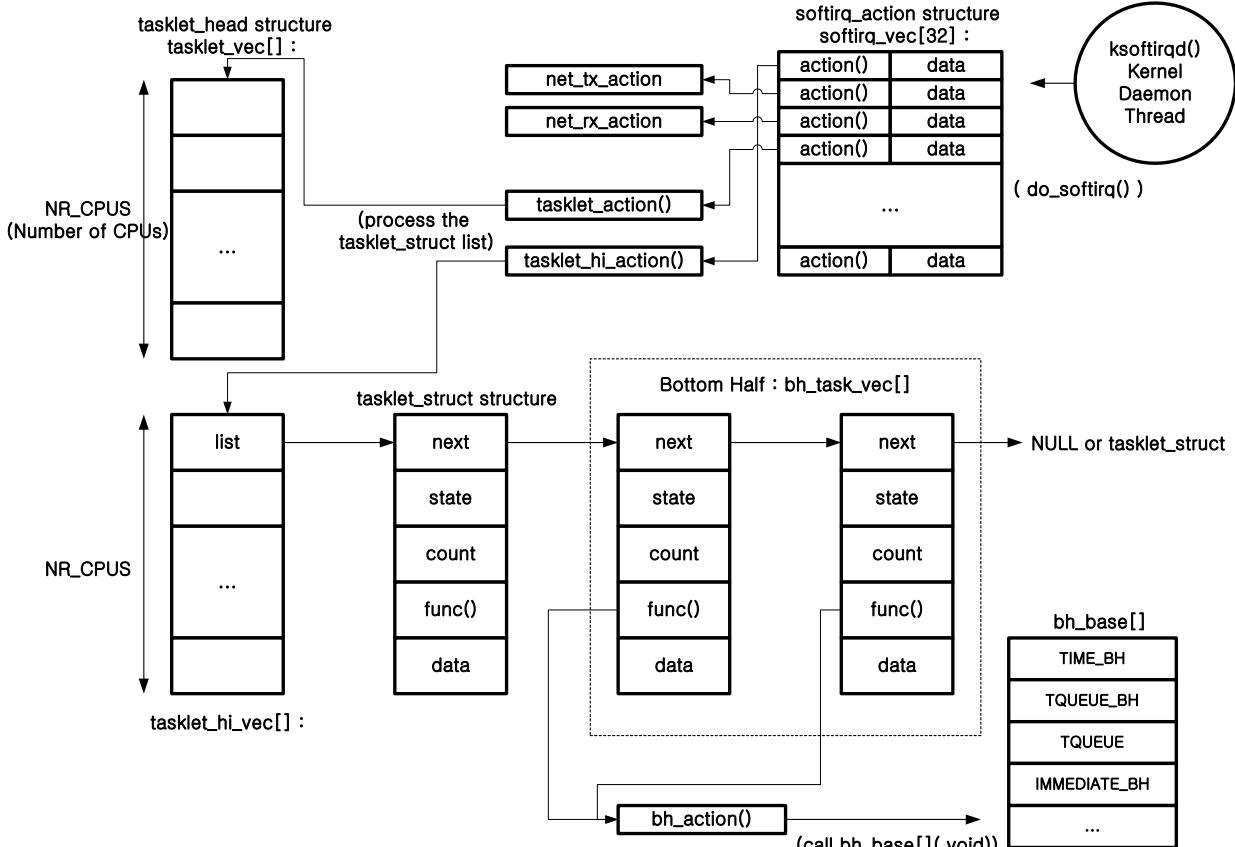


그림 27. Tasklet, Bottom Half, Software IRQ의 구조

[그림 27]에서 보듯이 모든 software IRQ에 대한 처리는 ksoftirqd 커널 데몬 thread에 의해서 처리된다. 이 커널 thread는 `softirq_vec[]`에 정의된 action에 따라서, 각각 `tasklet_hi_action()`, `tasklet_action()` 및 `net_tx_action()`과 `net_rx_action()`을 호출하게 되며, 이러한 action들 내에서 `tasklet_head`에 연결된 `tasklet_struct` 구조체들을 추적하면서 하나씩 처리해 나가게 된다. 또한 bottom half 역시 실제로 `tasklet_struct`의 배열로 정의되며, 이것이 `tasklet_hi_vec[]`의 CPU를 index로 한 `tasklet_struct`의 리스트에 자신의 `tasklet_struct`를 등록하는 것으로 처리된다. Bottom half는 `tasklet_struct` 구조체의 function을 가르키는 부분에 다시 `bh_action()`를 두고 있으며, `bh_action()` 함수에서 `bh_base[]` 배열을 접근해서 active된 bottom half들을 처리해 나가게 된다. `bh_base[]`에서 자주 사용되는 부분으로는 TIME_BH(Timer Bottom Half)와 TQUEUE_BH(Task Queue Bottom Half) 및 IMMEDIATE_BH(Immediate Bottom Half) 등이 있다.

2.21.6. Task Queue

지금부터 볼 것은 bottom half의 일환으로 수행되어야 할 일(task)를 가지는 큐(queue)에 대한 것을 보도록 하겠다. 즉, 이 queue는 처리해야 할 함수들의 list로 만들어진 것이며, 사용되는 자료구조들은 `~/linux/include/linux/tqueue.h`에 있다.

```
struct tq_struct {
    struct list_head list;          /* linked list of active bh's */
    unsigned long sync;             /* must be initialized to zero */
    void (*routine)(void *);        /* function to call */
    void *data;                     /* argument to function */
};
```

코드 200. tq_struct 구조체의 정의

tq_struct 구조체는 실행할 task(혹은 function)을 나타내기 위해서 사용한다. 즉, 이 tq_struct 구조체의 연결리스트로 이루어진 것을 task queue로 말하고, 이 연결된 task queue로 부터 tq_struct를 하나씩 떼어내서 수행하는 것이 바로 task queue를 실행하는 것이다.

tq_struct 구조체의 sync 필드는 0으로 항상 초기화되어야 하며, 이 값을 가지고 현재 task가 이미 task queue에 들어있는지를 나타내기 위해서 사용한다. Routine필드는 수행할 함수의 포인터를 가지며, data는 함수의 인자(argument)로 사용된다.

```
/*
 * Emit code to initialise a tq_struct's routine and data pointers
 */
#define PREPARE_TQUEUE(_tq, _routine, _data)
    do {
        (_tq)->routine = _routine;
        (_tq)->data = _data;
    } while (0)
/*
 * Emit code to initialise all of a tq_struct
 */
#define INIT_TQUEUE(_tq, _routine, _data)
    do {
        INIT_LIST_HEAD(&(_tq)->list);
        (_tq)->sync = 0;
        PREPARE_TQUEUE(_tq, _routine, _data);
    } while (0)

typedef struct list_head task_queue;

#define DECLARE_TASK_QUEUE(q)           LIST_HEAD(q)
#define TQ_ACTIVE(q)                  (!list_empty(&q))

extern task_queue tq_timer, tq_immediate, tq_disk;
```

코드 201. Task queue의 생성과 초기화

Task queue로서 중요하게 사용되는 것으로는 tq_timer, tq_immediate, tq_disk 등이 있다. 이중에서 tq_disk는 사용할 disk에 대한 접근을 요하는 일에 사용되기에, 파일 시스템이나 메모리 관리, 혹은 block 모드 디바이스 드라이버등에서 사용한다. tq_timer의 경우에는 시스템의 timer와 연동되어 매 time tick마다 수행되는 task queue이며, tq_immediate는 스케줄러나 시스템 call의 return 시에 수행되는 큐이다.

이것들을 앞에서 설명한 bottom half들과 연관시키면, IMEDIATE_BH는 tq_immediate에 task들을 연결하게 되며, TQUEUE_BH는 tq_timer에 task를 넣게 된다. TIMER_BH는 시스템의 timer를 유지할 목적으로 사용하기에 특별한 task queue를 가지지 않는다.

하나의 task queue를 선언하기 위해서는 DECLARE_TASK_QUEUE()라는 매크로를 사용한다. 이 매크로는 주어지는 인자의 이름으로 하나의 연결리스트를 생성하기 위한 head를 초기화 하는 일을 한다. 해당 task queue에 실행할 task가 있는가를 확인하기 위해서는 TQ_ACTIVE() 매크로를 사용한다. 단순히 list에 연결된 entry가 있는가를 확인하는 일을 한다. 그리고, 마지막으로 task queue에 들어갈 entry를 생성하기

위해서 INIT_TQUEUE() 매크로를 사용한다. 이 매크로는 tq_struct 구조체의 각 필드를 초기화 하는 일을 한다.

```
/* 아래의 부분은 ~/linux/include/linux/list.h에서 가져온 내용이다.*/
struct list_head {
    struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)

#define INIT_LIST_HEAD(ptr) do { \
    (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)
```

코드 202. LIST의 초기화

list_head 구조체는 Linux 커널에서 연결 리스트의 head 역할을 하는 node를 생성하는데 사용한다. INIT_LIST_HEAD() 매크로는 인자로 받은 list_head 구조체의 next와 prev 필드를 자기 자신을 가리키는 포인터로 초기화 시켜주는 일을 하며, LIST_HEAD는 하나의 list_head 구조체를 선언하고, 이 list_head 구조체의 next와 prev에 자기 자신의 주소를 주는 일을 한다. 즉, LIST_HEAD() 매크로는 list_head로 사용할 변수를 선언하는데 사용하며, INIT_LIST_HEAD() 매크로는 이미 list_head로 선언된 변수를 초기화 시켜주기 위해서 사용한다.

자, 그럼 이러한 task queue를 처리하는 함수를 등록하는 부분을 보도록 하자. 이 함수를 등록하는 것은 당연히 bottom half에 하나의 entry를 등록하는 것이다. init_bh() 함수가 이를 위해서 사용된다. 호출되는 부분은 ~/linux/kernel/sched.c의 sched_init()에서 예를 찾아볼 수 있다.

```
void __init sched_init( void )
{
    ...
    init_bh(TIMER_BH, timer_bh);
    init_bh(TQUEUE_BH, tqueue_bh);
    init_bh(IMMEDIATE_BH, immediate_bh);
    ...
}
```

코드 203. init_bh() 함수의 사용 예

sched_init() 함수에서는 TIMER_BH와 TQUEUE_BH, IMMEDIATE_BH에 해당하는 bottom half에 대해, action 함수들로서 timer_bh(), tqueue_bh(), immediate_bh()를 두어서 초기화 하고 있다. 따라서, mark_bh()와 같은 함수로 TIMER_BH, TQUEUE_BH, IMMEDIATE_BH를 설정한다면, timer_bh(), tqueue_bh(), immediate_bh()라는 함수들이 bottom half에 대한 처리시에 호출될 것이다.

```
void timer_bh(void)
{
    update_times();
    run_timer_list();
}
```

코드 204. timer_bh() 함수의 정의

timer_bh() 함수는 시스템의 timer를 업데이트(update)하고, 시스템의 timer list를 처리한다(run_timer_list()). 여기서 주의할 점은 timer list는 앞에서 본 task queue와는 다르다는 점이다. 여기서 처리될 timer list는 아래와 같은 자료 구조를 가지며, 시스템에서 관리하는 5개의 timer list queue 중의 하나에 들어가게 된다.

```
struct timer_list {
    struct list_head list;
    unsigned long expires;
    unsigned long data;
    void (*function)(unsigned long);
};
```

코드 205. timer_list 구조체의 정의

즉, timer list queue에 연결될 list 필드와 언제 timeout이 될지를 나타내는 expires, timeout이 되었을 때 수행할 함수를 나타내는 function, 그 함수에 전달될 데이터를 나타내는 data로 구성되어 있다. 따라서, 이 list 중에서 timeout된 것을 수행하기 위해서 run_timer_list() 함수를 호출하는 것이다.

```
void do_timer(struct pt_regs *regs)
{
    (*(unsigned long *)&jiffies)++;
#ifndef CONFIG_SMP
    /* SMP process accounting uses the local APIC timer */
#endif
    update_process_times(user_mode(regs));
    mark_bh(TIMER_BH);           /* TIMER_BH를 활성화(active) 시킴 */
    if (TQ_ACTIVE(tq_timer))     /* tq_timer task queue에 tq_struct 연결 리스트가 있는지 확인 */
        mark_bh(TQUEUE_BH);      /* TQUEUE_BH를 활성화 시킴 */
}
```

코드 206. do_timer() 함수의 정의

Timer interrupt의 일환으로 수행되는 do_timer() 함수는 앞에서 본 TIMER_BH와 TQUEUE_BH를 enable 시켜준다. 따라서, 만약 드라이버와 같은 것을 만드는 프로그래머라면, TIMER_BH와 TQUEUE_BH를 사용하는데 있어서 IMMEDIATE_BH와는 다르다는 점을 인식해야 할 것이다. TIMER_BH는 앞에서 보았듯이 timer_list라는 구조체를 시스템의 timer_list queue에 넣기 위해서 add_timer()와 같은 것을 사용해야 하며, 다시 제거하기 위해서는 del_timer()와 같은 함수를 사용해야 한다. 그리고, TQUEUE_BH를 사용하기 위해서는 단순히 tq_timer에 실행하고자 하는 tq_struct를 만들어서 넣어주기만 해야 한다. mark_bh()는 tq_timer가 active(즉, tq_timer list에 연결된 tq_struct가 있는 경우)인 경우에 자동으로 do_timer() 함수에서 mark_bh()를 해주기 때문이다. 따라서, IMMEDIATE_BH는 직접적으로 mark_bh()를 호출해서 IMMEDIATE_BH를 enable 시켜주고, tq_immediate task queue에 처리할 tq_struct가 있음을 알려주어야 할 것이다.

```
void tqueue_bh(void)
{
    run_task_queue(&tq_timer);
}

void immediate_bh(void)
{
    run_task_queue(&tq_immediate);
}
```

코드 207. tqueue_bh() 및 immediate_bh() 함수의 정의

tqueue_bh()와 immediate_bh() 함수는 앞에서 sched_init() 함수에서 보았듯이 TQUEUE_BH와 IMMEDIATE_BH bottom half를 처리하는 함수들이다. 이 함수들은 단순히 run_task_queue() 함수에 자신들이 관리하는 tq_timer 혹은 tq_immediate task queue의 포인터를 넘겨주는 것으로 끝난다. 즉, run_task_queue() 함수가 이러한 task queue에 연결된 tq_struct 구조체들을 하나씩 처리해 준다는 것이다.

```
extern void __run_task_queue(task_queue *list);

static inline void run_task_queue(task_queue *list)
{
    if (TQ_ACTIVE(*list))
        __run_task_queue(list);
}
```

코드 208. run_task_queue() 함수의 정의

run_task_queue() 함수는 인자로 넘겨받은 task_queue 구조체의 포인터를 보고, 이 task queue에 연결된 tq_struct가 있는지 확인한다(TQ_ACTIVE()). 만약 있다면, 다시 __run_task_queue() 함수를 호출해서 처리를 맡긴다.

```
void __run_task_queue(task_queue *list)
{
    struct list_head head, *next;
    unsigned long flags;

    spin_lock_irqsave(&tqueue_lock, flags);
    list_add(&head, list);
    list_del_init(&head);
    spin_unlock_irqrestore(&tqueue_lock, flags);

    next = head.next;
    while (next != &head) {
        void (*f)(void *);
        struct tq_struct *p;
        void *data;

        p = list_entry(next, struct tq_struct, list);
        next = next->next;
        f = p->routine;
        data = p->data;
        wmb();
        p->sync = 0;
        if (f)
            f(data);
    }
}
```

코드 209. __run_task_queue() 함수의 정의

__run_task_queue() 함수는 ~/linux/kernel/softirq.c에 정의된 함수이다. 이 함수의 먼저 task queue에 대한 lock을 걸어서 task queue에 대해서 동기화 시킨다. 그리고나서, head라는 변수를 선언해 이 변수로 해당 task queue의 연결 리스트를 모두 가져오도록 한다(list_add()). 이젠 task queue를 원래대로 비워두기 위해서 list_del_init()를 호출하고, 앞에서 설정한 lock을 해제한다.

이제부터는 head에 있는 list를 차례대로 수행하기만 하면 될 것이다. while() loop를 돌면서 list_entry()로 하나의 list entry를 구해서 그것이 가지고 있는 routine 필드에 연결된 함수와 data 필드에 연결된 함수의 파라미터 값을 읽어, 함수를 호출하면 된다. Sync 필드는 원래의 초기화 값인 0을 넣는다. wmb()는

`mb()`와 마찬가지로 실행의 순서를 동기화 시키기 위해서 사용했다. 프로그램의 문맥을 이해하는데는 생략해도 상관없다.

```
static inline int queue_task(struct tq_struct *bh_pointer, task_queue *bh_list)
{
    int ret = 0;
    if (!test_and_set_bit(0, &bh_pointer->sync)) {
        unsigned long flags;

        spin_lock_irqsave(&tqueue_lock, flags);
        list_add_tail(&bh_pointer->list, bh_list);
        spin_unlock_irqrestore(&tqueue_lock, flags);
        ret = 1;
    }
    return ret;
}
```

코드 210. `queue_task()` 함수의 정의

자, 이제 `task queue`에 대해서 남은 것은, `tq_struct` 구조체를 생성해서 어떻게 `task queue`에 넣을 수 있는 것이다. 이것을 위해서 `queue_task()`라는 `inline` 함수가 있다. 이 함수를 호출하기에 앞서 먼저 `tq_struct` 구조체를 하나 생성해야 하며, 또한 이 `tq_struct` 구조체가 들어가야 할 `task queue`를 알고 있거나 만들어야 할 것이다. 먼저 `tq_struct`의 `sync` 필드를 검사해서 이 필드가 0인 경우에는 1로 바꾼다. 그렇지 않다면, 이미 `task queue`에 들어가 있다는 말이 되므로 바로 복귀 할 것이다. 이전 중요한 자료구조를 갱신하는 것으로 `spin_lock_irqsave()`를 호출해서, 시스템의 전역 interrupt lock을 설정한다. 그리고나서, 주어진 `task queue`에 `tq_struct`를 끝에 추가 한다(`list_add_tail()`). 추가하는 것으로 일은 끝나게 되며, 앞에서 설정한 `lock`을 해제하고 1을 복귀값으로 넘겨준다. `task queue`에 추가된 `tq_struct`는 나중에 `run_task_queue()`와 같은 함수에서 수행될 것이다.

여기까지 해서 `tasklet`이 무엇인지, `bottom half`는 `tasklet`을 이용해서 어떻게 구현되는지, 그리고, `tasklet`을 처리하는 software IRQ의 구조와 `task queue`에 대해서 알아보았다. 한가지 짚고 넘어가고 싶은 것은 `tasklet_hi_vec[]` 배열은 주로 `bottom half`를 처리하는 `tasklet` list이므로, 될 수 있으면 `tasklet_vec[]`를 이용해서 사용자(커널 프로그래머)가 하고 싶은 일을 해야 한다는 것이다. 즉, `DECLARE_TASKLET()`를 사용해서 하나의 `tasklet`을 만들고, 이를 수행되게 하기 위해서 `tasklet_schedule()`로 `tasklet_vec[]`에 추가해둔다면, 결국 이 `tasklet`은 나중에 `ksoftirqd` 커널 데몬 `thread`에 의해서 처리될 것이다. 이 방법 이외에도 `task queue`를 이용한 것으로, 앞에서 보았던 `tq_struct` 구조체를 생성한 후 `queue_task()`를 이용해서 적절한 `task queue`에 넣어두고, 만약 `tq_immediate` `task queue`를 이용한다면, `mark_bh(IMMEDIATE_BH)`를 호출해서 이를 실행되도록 하는 방법이 있다. 그렇게 하지 않고 자신이 인위적으로 `task queue`를 생성하고자 한다면, `DECLARE_TASK_QUEUE()`를 호출한 후, 수행하고자 하는 `task`들을 `tq_struct`로 만들고, 이를 생성한 `task queue`에 `queue_task()`를 이용해서 넣고, 적절한 순간에 `run_task_queue()` 함수를 호출해서 수행되도록 만들어 줄 수도 있을 것이다. 어쨌든, 이것은 프로그래머의 선택이며, 자신이 하고자 하는 일에 적절한 형태로 사용하면 될 것이다. 이것으로 이번장의 이야기를 마치도록 하겠다.

2.22. Linux O(1) Scheduler

Linux 커널 버전 2.4까지의 공식 release에서는 스케줄러(scheduler)의 구현에 있어서 문제점이 있다. 즉, 프로세스의 갯수가 늘어나면, 스케줄링을 위해서 필요한 시간도 늘어난다는 점이다. 즉, $O(n)$ 이라는 시간이 필요하다는 말이다. 이때 n 은 프로세스의 개수가 되며, 이에 비례한다는 뜻이다. $O(1)$ 스케줄러는 이러한 것을 개선해서 프로세스의 개수에 상관없이, 항상 상수(constant)시간안에 스케줄링 하는 것을 보장하겠다는 것이다. 커널 버전 2.5에 들어가 있으며, 현재는 2.4 버전에 대한 patch도 제공하고 있다. 그래서, 차후에 발표될 2.6버전의 커널에 제공될 가능성성이 많은, $O(1)$ 스케줄링에 대한 분석을 이번장에서 하도록 하겠다.

2.22.1. 기존의 Linux 스케줄링 메커니즘

스케줄링이란 특정 프로세스에게 CPU라는 자원을 배정하는 일을 말한다⁴¹. 스케줄링이 일어나는 시점은 직접적으로 schedule()이라는 함수를 호출하는 곳⁴²과 Interrupt와 같은 외부 event에 의해서, 실행중인 현재 프로세스의 task_struct 구조체에 있는 need_resched이라는 필드가 설정되어, 사용자 모드로 전이하기 전에 일어나게 된다.

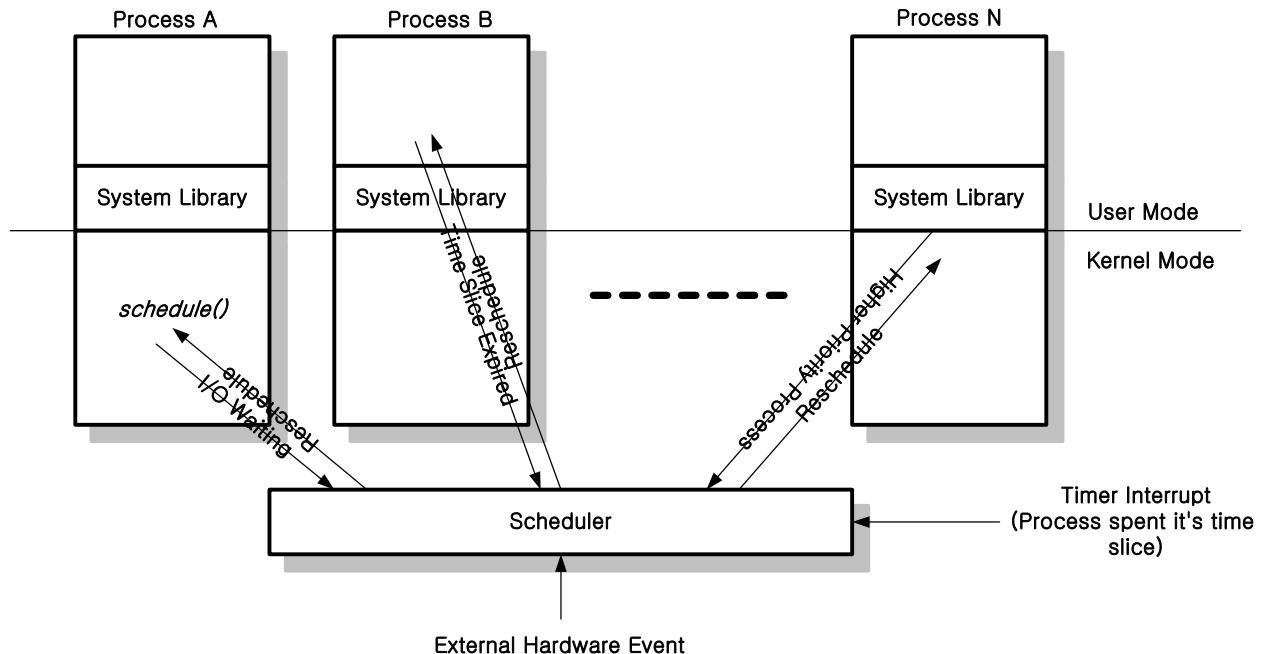


그림 28. Linux에서의 Scheduler 호출

따라서, 이러한 스케줄러는 커널의 핵심 함수중의 하나이며, 가장 많이 호출되는 함수일 것이다. 즉, 함수의 성능(performance)이 전체 시스템의 성능에 미치는 영향은 아주 클 것이다.

Linux에서는 한번의 스케줄링이 일어날 경우, 가장 우선 순위를 가지는 프로세스를 찾기 위해서 init_task라는 전체 프로세스의 연결 리스트의 head에서부터 goodness()라는 함수를 사용해서 조사해 나간다. 이렇게 조사한 goodness() 값이 가장 큰 프로세스에게 CPU를 부여하는 것이다. 따라서, 프로세스의 개수가 증가하면, 당연히 계산하여야 하는 양은 커지게 되며, 효율성이 떨어지게 된다는 것이다.

2.22.2. 커널 버전 2.5의 Scheduler의 특징

Linux 커널 버전 2.5에서의 Scheduler는 크게 보면 다음과 같은 특성을 가지고 있다. 즉, Priority에 기반한 queue를 가지고 있다는 것과 Preemptive Kernel을 적용해, 커널내에서도 preemption이 가능하도록 만들었다는 점이다. Preemptive kernel의 경우는 이미 project의 형태로 제공되고 있는 2.4버전에 대한 patch도 있으며, 현재는 RedHat에서 site에서 얻을 수 있다. O(1) scheduler를 구현하기 위한 방법으로 사용하는 것이, 바로 priority에 기반한 multi-level feed back queue이다. 즉, 기존의 OS에서 스케줄링을 위해서 흔히 사용하는 방법중의 하나를 차용한 것이라고 생각할 수 있을 것이다⁴³.

⁴¹ 물론 이것은 CPU Scheduling을 말하는 것이다. Disk 스케줄링이라고 할 때는 디스크에 대한 I/O 요청을 어떤 순으로 처리할 것인가를 정하는 것이 될 수 있을 것이다.

⁴² 이것은 프로세스가 I/O를 대기하기 위해서 wait queue에 자신의 process를 추가하고, 상태를 변화시킨 후, 호출하게 된다.

⁴³ 실제로 FreeBSD의 경우에는 오래전부터 Multi-Level FeedBack Queue를 사용해서 Ready Queue를 구현하고 있다. Queue내에서 우선 순위가 높은 프로세스들은 Round Robin 방법으로 수행된다. 항상 같은

Real-time을 제공하기 위한 노력의 일환으로 시작된 preemptive kernel은, 리눅스가 커널 모드에서 preemption을 지원하지 못하는 문제로 인해, real-time 성을 가지는 프로세스가 자신이 가진 dead-line을 맞추지 못하는 것을 개량하기 위한 것이다. 이와 관련해서 이미 RTLinux라는 것이 FSMLab에서 개발되었지만, 이는 Linux 커널을 일종의 우선순위가 가장 낮은 real-time 프로세스로 생각해서 RTLinux가 제공하는 스케줄러를 이용해서 스케줄링하겠다는 것이다. 따라서, 이와 같은 경우에는 RTLinux가 제공하는 커널 layer가 하나더 Linux 커널 이하에 있다고 보는 것이 정확할 것이다. 물론 이와 같이 했을 경우에는 Hardware에서 발생하는 모든 event를 RTLinux 커널이 먼저 처리하게 되므로, Hard Real-time을 구현할 수 있는 방법을 제공한다. 하지만, Preemptive Kernel은 이와는 달리, Linux 커널 자체의 코드를 바꾼 것으로, real-time event 자체는 Linux 커널이 먼저 처리하게 되는 것이다. Real-time Linux 커널을 구현하는 또 다른 한가지 방법으로는 Low Latency 커널이 있다. 이것은 spin-lock을 사용해서 커널 모드에서의 preemption이 일어나지 못하게 하는 부분에 대해서, lock을 깨고(break) 나올 수 있는 방법을 제공하자는 것이다. 따라서, 이와 같은 approach에서는 모든 커널의 source code를 분석해, 이와 같은 부분이 있는가를 보고, 그 부분에 대해서 추가적으로 코드를 더 작성하는 일이 필요하므로, 아주 많은 커널 코드의 변경이 필요하다.⁴⁴

현재로서는 2.5버전에서 적용된 커널은 Low Latency 커널을 적용한 것이 아니라, Peemptive Kernel을 적용한 것을 확인할 수 있었기에, 이후의 부분에서 이를 보도록 하겠다.

2.22.3. Preemptive Kernel의 분석

2.5 버전의 커널을 논하기 위해서는 preemptive kernel에 대한 이야기를 먼저 하여야 할 것이다. 즉, 2.5 버전의 커널에서의 스케줄링은 커널내부에서도 preemption을 지원하기에, 더 빠른 real-time event에 대한 반응을 보장하려는 노력을 기울이고 있기 때문이다. 먼저 preemptive kernel을 지원하기 위해서는 커널을 compile하는 과정에서 CONFIG_PREEMPT를 설정해 주어야 한다. 이렇게 했을 경우 config.h에 자동으로 CONFIG_PREEMPT가 정의(define)되어 커널을 컴파일하는 과정에 관련된 부분이 자동으로 포함될 것이다. 하지만, preemptive kernel의 특정 부분은 CONFIG_PREEMPT 설정이 없더라도, 기본 커널의 일부로 동작될 것이다. ~/include/linux/preempt.h를 보도록 하자.

```
#define preempt_count() (current_thread_info()->preempt_count)
#define inc_preempt_count() \
do { \
    preempt_count();++; \
} while (0)
#define dec_preempt_count() \
do { \
    preempt_count();--; \
} while (0)

#ifndef CONFIG_PREEMPT
extern void preempt_schedule(void);
#define preempt_disable() \
do { \
    inc_preempt_count(); \
    barrier(); \
} while (0)

#define preempt_enable_no_resched() \
do { \
    dec_preempt_count(); \
    barrier(); \
} while (0)

```

우선 순위에서 먼저 수행되어야 할 프로세스가 queue의 첫부분에 위치하도록 만든다. 실행을 마친 프로세스는 우선 순위가 새롭게 계산되어 Multi-Level FeedBack queue에 들어갈 위치를 찾게 된다.

⁴⁴ Full time job이라는 용어를 사용해서 이와 같은 일을 하고 있다고 표현하고 있다. 원래 처음 이를 진행한 사람에게 현재는 다른 사람으로 바뀌어서 일을 진행하고 있다고 한다(2002년 11월 현재).

```
#define preempt_check_resched() \
do { \
    if (unlikely(test_thread_flag(TIF_NEED_RESCHED))45) \
        preempt_schedule(); \
} while (0)

#define preempt_enable() \
do { \
    preempt_enable_no_resched(); \
    preempt_check_resched(); \
} while (0)

#else
#define preempt_disable()          do { } while (0)
#define preempt_enable_no_resched() do { } while (0)
#define preempt_enable()           do { } while (0)
#define preempt_check_resched()    do { } while (0)
#endif
```

코드 211. preempt.h 파일

preempt_count() 매크로는 현재 프로세스의 thread_info 구조체⁴⁶ 필드에 있는 preempt_count라는 값을 가르킨다. 이 값은 preempt가 된 회수를 나타낸다. 따라서, 만약 preempt가 된다면, 이 값은 증가될 것이다. 이 값이 0이상의 값을 가지고 있다면, 이미 preemption이 된 상황이므로, preemption하지 못하도록 하는데 사용된다. 증가되는 것은 inc_preempt_count()가 수행하며, 반대 역할을 하는 것이 dec_preempt_count()이다. 커널의 컴파일 옵션으로 CONFIG_PREEMPT가 설정된 경우에는, preempt_schedule()이라는 함수를 사용할 수 있으며, preempt_disable()은 현재 프로세스의 preemption count 값을 증가시키는 역할을 수행하고, 동기화 되도록 기다린다(barrier()). 반대 역할을 수행하는 것은 preempt_enable()이 맡고 있다. 이 매크로는 preempt_enable_no_resched()을 사용해서 thread_info 구조체의 preempt_count를 감소시키고, preemption이 필요한가를 알아보기 위해서 preempt_check_resched()를 호출한다. 만약 preemption이 필요하다고 여겨질 때는 preempt_check_resched()를 사용해서 TIF_NEED_RESCHED가 thread_info 구조체에 설정되었는지를 확인하게 되며, 이 flag이 설정된 경우에는 rescheduling을 하기 위해서 preempt_schedule()을 호출하게 된다. CONFIG_PREEMPT가 컴파일 옵션으로 주어진 경우에는 아무런 일도 하지 않는 NULL macro가 될 것이다.

```
/* ~/include/asm-i386/thread_info.h에서 */
...
/* how to get the thread information struct from C */
static inline struct thread_info *current_thread_info(void)
{
    struct thread_info *ti;
    __asm__ ("andl %%esp,%0; ":"=r" (ti) : "0" (~8191UL));
    return ti;
}
...
```

코드 212. current_thread_info() 함수의 정의

⁴⁵ likely()와 unlikely()는 GCC compiler에서 정의되어 있는 __builtin_expect(x, 1)과 __builtin_expect(x, 0)으로 ~/include/linux/compile.h에 정의되어 있다. GCC 2.96이상의 버전에서만 사용 가능하기에, 이하의 버전에서는 단순히 원래의 값을 돌려주는 일만한다.

⁴⁶ Linux 커널 버전 2.5.X에서는 task_struct 구조체에 thread_info라는 구조체를 가르키는 필드가 존재한다. thread_info 구조체는 process를 실행하는데 필요한 정보를 담는 구조체로 나중에 task_struct를 보게 될 때 설명될 것이다.

`current_thread_info()` 함수는 항상 현재 프로세스의 `thread_info` 구조체 필드에 대한 포인터를 얻기 위해서 사용한다. 이 포인터는 커널 모드에서는 ESP(Stack Pointer)의 8191(=0x1FFF)를 NOT시킨 값과 OR시킨 값을 사용한다. 즉, 현재 프로세스의 커널 stack에서 8192만큼 떨어진 위치에 `task_struct`가 위치하며, 첫번째 필드에는 `state`가 4 bytes를 차지하기 때문에, 그 다음에 있는 `thread_info` 구조체의 포인터 필드를 얻기 위한 것이다.

```
/* ~/include/linux/thread_info.h 파일에서 */
...
static inline int test_thread_flag(int flag)
{
    return test_bit(flag,&current_thread_info()->flags);
}
...
```

코드 213. `test_thread_flag()` 함수의 정의

`test_thread_flag()` 함수는 `test_bit()`을 사용해서 현재 `thread_info` 구조체의 `flag` 필드에 특정 값(=flag)이 설정되었는지를 확인하기 위해서 호출되는 함수이다. 이 함수 이외에도 `~/include/linux/thread_info.h` 파일에는 다음과 같은 것들이 정의되어 있다.

- static inline void `set_thread_flag(flag)` : 현재 프로세스의 `thread_info` 구조체의 `flag`에 특정 bit을 설정한다.
- static inline void `clear_thread_flag(flag)` : 현재 프로세스의 `thread_info` 구조체의 `flag`에 특정 bit을 clear한다.
- static inline void `test_and_set_thread_flag(flag)` : 현재 프로세스의 `thread_info` 구조체의 `flag`에 특정 bit을 검사해서 설정한다. 이때 이전에 설정된 값을 돌려준다.
- static inline void `test_and_clear_thread_flag(flag)` : 현재 프로세스의 `thread_info` 구조체의 `flag`에 특정 bit을 검사해서 이를 지운다. 이때 이전에 설정된 값을 돌려준다.
- static inline void `set_ti_thread_flag(struct thread_info *ti, int flag)` : 넘겨받은 `thread_info` 구조체의 `flag`에 특정 bit을 설정한다.
- static inline void `clear_ti_thread_flag(struct thread_info *ti, int flag)` : 넘겨받은 `thread_info` 구조체의 `flag`에 특정 bit을 지운다.
- static inline int `test_and_set_ti_thread_flag(struct thread_info *ti, int flag)` : 넘겨받은 `thread_info` 구조체의 `flag`에 특정 bit이 설정되었는지를 확인하고, 이를 설정한다. 이때 이전에 설정된 값을 돌려준다.
- static inline int `test_and_clear_ti_thread_flag(struct thread_info *ti, int flag)` : 넘겨받은 `thread_info` 구조체의 `flag`에 특정 bit이 설정되었는지를 확인하고, 이를 지운다. 이때 이전에 설정된 값을 돌려준다.
- static inline int `test_ti_thread_flag(struct thread_info *ti, int flag)` : 넘겨받은 `thread_info` 구조체의 `flag`에 특정 bit이 설정되어 있는지를 확인한다. 설정된 값을 돌려주기만 한다.
- static inline void `set_need_resched(void)` : 현재 프로세스의 `thread_info` 구조체에 있는 `flag` 필드에 scheduling을 새로하라는 의미를 가지는 TIF_NEED_RESCHED bit을 설정한다.
- static inline void `clear_need_resched(void)` : 현재 프로세스의 `thread_info` 구조체에 있는 `flag` 필드에 scheduling을 새로하지 않아도 좋다는 것을 나타내기 위해서 TIF_NEED_RESCHED bit을 지운다.

설정될 수 있는 bit들에 대한 정의는 CPU별로 나누어져 있으며, i386의 경우에는 아래와 같이 `~/include/asm-i386/thread_info.h`에 정의되어 있다.

```
/*
 * thread information flags
 * - these are process state flags that various assembly files may need to access
 * - pending work-to-be-done flags are in LSW
 * - other flags in MSW
```

```

/*
 *      Bit field name      Position : Comment from shkwon */
#define TIF_SYSCALL_TRACE    0      /* syscall trace active : 현재 system call trace가 되고 있다.*/
#define TIF_NOTIFY_RESUME   1      /* resumption notification requested : 재수행 요청을 받았다.*/
#define TIF_SIGPENDING     2      /* signal pending : 처리되지 않은 signal이 있다.*/
#define TIF_NEED_RESCHED   3      /* rescheduling necessary : 스케줄링할 필요가 있다.*/
#define TIF_USEDFPU        16     /* FPU was used by this task this quantum (SMP) : FPU를
사용한다.*/
#define TIF_POLLING_NRFLAG 17      /* true if poll_idle() is polling TIF_NEED_RESCHED */
                                /* TIF_NEED_RESCHED가 polling된 경우에 ON 됨47 */
#define _TIF_SYSCALL_TRACE (1<<TIF_SYSCALL_TRACE)
#define _TIF_NOTIFY_RESUME (1<<TIF_NOTIFY_RESUME)
#define _TIF_SIGPENDING   (1<<TIF_SIGPENDING)
#define _TIF_NEED_RESCHED (1<<TIF_NEED_RESCHED)
#define _TIF_USEDFPU      (1<<TIF_USEDFPU)
#define _TIF_POLLING_NRFLAG (1<<TIF_POLLING_NRFLAG)

#define _TIF_WORK_MASK          0x0000FFFF /* work to do on interrupt/exception return :
Interrupt나 Exception이 return하게 될 때 해야 할 일을 나타내는 MASK 값*/
#define _TIF_ALLWORK_MASK       0x0000FFFF /* work to do on any return to u-space : 모든 User
mode로의 전환시에 해야 할 일을 나타내는 MASK 값 */

```

코드 214. **thread_info** 구조체에서의 flag값 정의

나중에 자세히 **thread_info** 구조체에 대해서 보게 되겠지만, 이곳에서는 우선 flag에 설정될 수 있는 값부터 보도록 하자. 이전 버전의 커널에서는 `need_resched`라는 `task_struct`의 필드를 설정하는 것으로, 현재 프로세스를 실행하는 동안에 `rescheduling`을 해야 할 event가 생겼다는 것을 알려주었다. 2.5.X버전의 Linux 커널에서는 이러한 일이 `thread_info` 구조체의 flag에 `TIF_NEED_RESCHED`를 설정하는 것으로 이루어진다.

```

struct thread_info {
    struct task_struct *task;           /* main task structure */
    struct exec_domain *exec_domain;   /* execution domain */
    unsigned long flags;               /* low level flags */
    __u32 cpu;                      /* current CPU */
    __s32 preempt_count;             /* 0 => preemptable, <0 => BUG */
    mm_segment_t addr_limit;         /* thread address space:
                                         0-0xBFFFFFFF for user-thread
                                         0-0xFFFFFFFF for kernel-thread
                                         */
    __u8 supervisor_stack[0];
};

```

코드 215. i386 CPU에서의 **thread_info** 구조체의 정의

`thread_info`는 CPU에 의존적인 자료구조이다. `task_struct` 필드는 `thread_info`가 관련을 가지고 있는 `process`의 `task_struct`구조체에 대한 포인터이며, `exec_domain`은 예전에 `task_struct`내의 `exec_domain`을 그대로 유지한다. `flags`필드는 앞에서 보았던 `TIF_XXX_XXX`라는 정보를 가지는 필드이며, 현재 `process`를 수행하고 있는 CPU를 가르키기 위한 `cpu` 필드와 실행중인 `process`가 `preemption`이 가능한가를 알려주기 위한 `preempt_count` 필드를 가지고 있다. 또한 `addr_limit`은 프로세스의 주소공간(address space)을

⁴⁷ `poll_idle()`이라는 함수에서 현재 프로세스의 `thread_info` 구조체에 `TIF_NEED_RESCHED`가 설정되었는지를 확인하고, 설정되지 않았을 경우에만 현재 프로세스의 `thread_info` 구조체의 flag에 설정된다. 이후에 `poll_idle()` 함수는 아무런 일도 하지 않고, `TIF_NEED_RESCHED`가 설정될 때까지 기다리게 된다.

나타낸다. 이때, 사용자 프로세스인 경우에는 0에서 0xFFFFFFFF까지의 값을 나타내게 되며, 커널 모드의 프로세스(여기서는 커널 thread)인 경우에는 0에서 0xFFFFFFFF까지의 값을 가진다. supervisor_stack[]은 현재로서는 별다른 쓰임새를 찾지 못했다. 하지만, init_thread의 선언에서 이곳에 KERNEL_DS(Kernel의 data segment) 부분이 위치하게 된다. thread_info 구조체를 사용하는 예를 보면 다음과 같은 init_task를 생성하는 것을 들 수 있겠다.

```
#ifndef __ASSEMBLY__
#define INIT_THREAD_INFO(tsk)
{
    .task          = &tsk,
    .exec_domain   = &default_exec_domain,
    .flags         = 0,
    .cpu           = 0,
    .preempt_count = 1,
    .addr_limit    = KERNEL_DS,
}
```

코드 216. init_task의 thread_info 구조체에 대한 정의

INIT_THREAD_INFO()라는 매크로를 가지고 init_task에 대한 thread_info 구조체를 초기화하는 부분이다. 즉, task, exec_domain, flags, cpu 및 preempt_count를 초기화 시켜주고, addr_limit에는 KERNEL_DS(=0xFFFFFFFF)을 넣어주고 있다. INIT_THREAD_INFO() 매크로는 나중에 init_task를 정의하는데 사용된다. 이후, 이것을 사용해서 init kernel thread(PID=0)가 생성될 것이다. 이전 커널 모드에서의 preemption을 책임지고 있는 preempt_schedule() 함수를 보도록 하자. 정의는 ~/kernel/sched.c에 아래와 같이 되어있다.

```
#ifdef CONFIG_PREEMPT
/*
 * this is the entry point to schedule() from in-kernel preemption
 * off of preempt_enable. Kernel preemptions off return from interrupt
 * occur there and call schedule directly.
 */
asmlinkage void preempt_schedule(void)
{
    struct thread_info *ti = current_thread_info();

    /*
     * If there is a non-zero preempt_count or interrupts are disabled,
     * we do not want to preempt the current task. Just return..
     */
    if (unlikely(ti->preempt_count || irqs_disabled()))
        return;
need_resched:
    ti->preempt_count = PREEMPT_ACTIVE;
    schedule();
    ti->preempt_count = 0;
    /* we could miss a preemption opportunity between schedule and now */
    barrier();
    if (unlikely(test_thread_flag(TIF_NEED_RESCHED)))
        goto need_resched;
}
#endif /* CONFIG_PREEMPT */
```

코드 217. preempt_schedule() 함수의 정의

preempt_schedule() 함수는 현재 진행중인 프로세스의 thread_info 구조체를 얻기 위해서 current_thread_info() 함수를 사용한다. 이렇게 얻어온 thread_info 구조체의 preempt_count 값이 0이

아니거나 IRQ가 disable되어 있는 상황이라면, preemption을 하지 못하도록 그냥 return하게 된다. 즉, 이미 preemption이 일어난 상황이거나, IRQ를 빨리 처리해주어야 하는 상황이라면, preemption을 일어나지 못하게 하는 것이다. 너무 잦은 preemption은 실행중인 프로세스로부터 CPU를 빼앗는 것으로, 너무 자주 일어난다면, 현재 진행중인 프로세스는 제대로 실행할 수 없는 상황이 될 것이다. 또한 IRQ가 disable된 상황에서의 preemption도 옳지 못한 상황이기에 이를 막는 것이다. need_resched이하의 부분은 preemption을 유발하는 부분이다. 현재 프로세스의 thread_info 구조체에 있는 preempt_count를 PREEMPT_ACTIVE(=0x04000000)로 두고, schedule() 함수를 호출한다. schedule() 함수내에서 스케줄링이 발생해서 다른 프로세스가 수행될 것이다. 이후에 현 프로세스가 수행될 기회를 얻는다면, schedule() 함수에서의 호출 이후의 instruction을 수행할 것이기에, preempt_count를 다시 0으로 설정하는 부분에서 수행이 계속될 것이다. barrier()는 컴파일러에 memory barrier를 삽입하라는 지시가 되며, 하드웨어에는 아무런 영향을 미치지 못한다. 다만 현재까지 CPU 레지스터내에서 사용중이고, 변경된 모든 변수를 해당 메모리에 다시 저장하도록 하는 역할을 한다. 이후에 필요한 것이 있다면 새로 읽어들일 것이다. 여기까지 마치면, 다시 현재 프로세스를 수행하기에 앞서, 현재 프로세스의 thread_info 구조체의 flags 필드에 TIF_NEED_RESCHED가 설정되어 있는가를 확인한다. 만약 있다면, need_resched로 다시 jump해서 수행할 것이다. 이 preempt_schedule() 함수도 역시 CONFIG_PREMPT가 설정된 상황에서만 유효한 의미를 가진다.

2.22.4. schedule() 함수의 분석

schedule() 함수는 Linux에서 스케줄링을 담당하는 함수이다. 이 함수가 호출되는 시점은 시스템이 현재 프로세스를 계속 수행할 수 없는 상황을 만났을 때가 되며, 이러한 예로서는 더 높은 우선순위를 가지는 프로세스가 수행 가능한 상태가 되었거나, 현재의 프로세스가 I/O를 위해서 wait하는 경우, 그리고, 현재 프로세스의 time slice를 다 사용한 경우로 나누어 볼 수 있다. 따라서, 이 함수가 호출될 경우에는 어떤 식으로든 다음에 실행할 프로세스를 결정해 주어야 하며, 해당 프로세스의 context를 복구하고, 현재 수행중이던 프로세스의 context를 save하는 일이 필요하다. 호출의 결과로 나중에 다시 현재 프로세스가 수행될 기회를 얻어개되면, schedule() 함수를 호출한 다음에 오는 instruction을 수행할 것이다.

```
/*
 * schedule() is the main scheduler function.
 */
asm linkage void schedule(void)
{
    task_t *prev, *next;
    runqueue_t *rq;
    prio_array_t *array;
    struct list_head *queue;
    int idx;

    /*
     * Test if we are atomic. Since do_exit() needs to call into
     * schedule() atomically, we ignore that path for now.
     * Otherwise, whine if we are scheduling when we should not be.
     */
    if (likely(current->state != TASK_ZOMBIE)) {
        if (unlikely(in_atomic())) {
            printk(KERN_ERR "bad: scheduling while atomic!\n");
            dump_stack();
        }
    }
}
```

코드 218. schedule() 함수의 정의

schedule() 함수가 호출되면, 가장 먼저 현재 프로세스의 상태가 TASK_ZOMBIE가 인가를 확인한다. likely()는 현재 버전에서는 아직 구현된 것이 없고, 단순히 parameter의 값을 돌려주는 일을 한다. 따라서, current 프로세스의 상태가 TASK_ZOMBIE인가 확인한다. TASK_ZOMBIE 상태는 프로세스는 이미

종료했지만, 종료 상황을 상위의 parent process에게 보고하지 않은 상태로 task_struct 구조체만 가지고 있는 상태이다. 만약 TASK_ZOMBIE의 상태인 프로세스에서 schedule()이라하는 함수가 호출되었다면, in_atomic()을 호출해서 preemption 할 수 있는 여부를 확인하게 된다. 만약 in_atomic()이 0이 아닌 값을 돌려준다면, 이미 preemption 할 수 없는 상황이라는 것을 알게되며, 이때는 dump_stack()을 호출해서 현재 프로세스의 커널 stack을 dump 한다.

```
#if CONFIG_PREEMPT
# define in_atomic()      ((preempt_count() & ~PREEMPT_ACTIVE) != kernel_locked())
...
#else
# define in_atomic()      (preempt_count() != 0)
...
#endif
```

코드 219. in_atomic() 매크로의 정의

in_atomic()이라는 매크로는 CONFIG_PREEMPT가 정의되지 않았을 경우에는 단순히 현재 프로세스의 preempt_count가 0인가를 확인하기 위해서 사용한다. preempt_count가 0이 아니라면, 1을 0이라면 0을 돌려줄 것이다. 만약 CONFIG_PREEMPT가 정의된 경우라면, preempt_count에서 PREEMPT_ACTIVE bit를 clear시킨 후, 이 값과 kernel_locked()의 값을 비교한다. 같다면 0을, 그렇지 않다면, 1을 돌려줄 것이다. kernel_locked()은 현재 수행중인 프로세스의 task_struct에 있는 lock_depth가 0보다 큰가를 알아 보기 위해서 사용하는 매크로이며, SMP이고, CONFIG_PREEMPT인 경우에 동작할 것이다. 그렇지 않다면, 항상 1을 돌려준다. 즉, 커널 모드에서 이미 진행중인 상황이라는 것을 알려준다.

여기서, 커널의 lock에 대해서 잠시 보고 넘어가도록 하자. 커널에 lock을 거는 상황은 전역적인 커널에 대한 접근을 금한다는 뜻이된다. 즉, 커널에 lock을 걸고 진입한 프로세스는 preemption이 되지 않고, 수행을 마칠 수 있다는 것을 보장한다는 의미이다. 즉, 수행을 마치고 복귀하는 시점이나, 중요한 데이터 구조에 대한 수정을 마치면, 다시 커널의 lock을 해제할 것이다. 관련된 함수나 매크로들은 ~/include/linux/smp_lock.h에 아래와 같이 정의되어 있다.

```
#include <linux/config.h>
#include <linux/sched.h>
#include <linux/spinlock.h>

#if CONFIG_SMP || CONFIG_PREEMPT
extern spinlock_t kernel_flag;

#define kernel_locked()          (current->lock_depth >= 0) /* 현재 프로세스가 lock을 설정한 depth */
#define get_kernel_lock()        spin_lock(&kernel_flag) /* 커널에 대한 전역 lock의 획득 */
#define put_kernel_lock()        spin_unlock(&kernel_flag) /* 커널에 대한 전역 lock의 해제 */
*/
 * Release global kernel lock.
 */
static inline void release_kernel_lock(struct task_struct *task)
{
    if (unlikely(task->lock_depth >= 0))
        put_kernel_lock();
}

/*
 * Re-acquire the kernel lock
 */
static inline void reacquire_kernel_lock(struct task_struct *task)
{
    if (unlikely(task->lock_depth >= 0))
        get_kernel_lock();
}
```

```

}

/*
 * Getting the big kernel lock.
 *
 * This cannot happen asynchronously,
 * so we only need to worry about other
 * CPU's.
 */
static inline void lock_kernel(void)
{
    int depth = current->lock_depth+1;
    if (likely(!depth))
        get_kernel_lock();
    current->lock_depth = depth;
}

static inline void unlock_kernel(void)
{
    if (unlikely(current->lock_depth < 0))
        BUG();
    if (likely(--current->lock_depth < 0))
        put_kernel_lock();
}
#else
#define lock_kernel()          do { } while(0)
#define unlock_kernel()        do { } while(0)
#define release_kernel_lock(task) do { } while(0)
#define reacquire_kernel_lock(task) do { } while(0)
#define kernel_locked()        1
#endif /* CONFIG_SMP || CONFIG_PREEMPT */

```

코드 220. Kernel의 lock에 대한 매크로 및 함수의 정의

release_kernel_lock() 함수는 현재 lock의 depth가 0 이상인 경우에 커널에 대한 전역 lock을 해제하는 역할을 한다. 반대 역할을 하는 것이 reacquire_kernel_lock()이라는 함수다. 이 함수는 lock depth를 확인한 후, 0 이상의 값을 가진다면 커널의 lock을 획득하기 위해서 get_kernel_lock()을 호출한다. 두 경우 모두 항상 lock의 depth는 0이상이 되어야 한다는 것을 보장한다.

전역적인 커널에 대한 lock을 획득하는 다른 방법으로는 lock의 depth에 영향을 주는 lock_kernel()이라는 함수가 있다. 이 함수는 depth가 0인 경우에 대해서만 lock을 설정하기 위해서 get_kernel_lock()을 호출한다. 즉, 이전에 이미 lock의 depth가 -1인 경우에 대해서만 lock을 설정하고, 그렇지 않다면, 단순히 lock의 depth만 증가시키는 일을 한다. 이와 반대되는 일을 하는 함수가 unlock_kernel()이다. 이 함수에는 0보다 작은 값을 가지는 lock depth에 대해서는 에러라는 것을 알리기 위해서 BUG()를 호출하고, 그렇지 않다면, 현재 lock depth를 감소시킨다. 이때 감소된 lock의 depth가 0보다 작다면, put_kernel_lock()을 호출해서 커널에 설정된 lock을 해제하는 역할을 한다. 위에서 나열한 모든 함수 및 매크로들은 kernel_locked()을 제외하곤 CONFIG_SMP나 CONFIG_PREEMPT가 정의되지 않았다면, 아무런 일을 하지 않을 것이다. kernel_locked()은 단순히 1만 돌려준다.

```

#if CONFIG_DEBUG_HIGHMEM
    check_highmem_ptes();
#endif
need_resched:
    preempt_disable();
    prev = current;
    rq = this_rq();

```

```

release_kernel_lock(prev);
prev->sleep_timestamp = jiffies;
spin_lock_irq(&rq->lock);

/*
 * if entering off of a kernel preemption go straight
 * to picking the next task.
 */
if (unlikely(preempt_count() & PREEMPT_ACTIVE))
    goto pick_next_task;

switch (prev->state) {
case TASK_INTERRUPTIBLE:
    if (unlikely(signal_pending(prev))) {
        prev->state = TASK_RUNNING;
        break;
    }
default:
    deactivate_task(prev, rq);
case TASK_RUNNING:
    ;
}
pick_next_task:

```

코드 221. schedule() 함수의 정의(계속)

여기까지 진행했다면, 스케줄링 요청이 일단은 제대로 되었다는 것을 확인한 것이므로, 본격적인 스케줄링을 하기 위한 준비 작업을 한다. preempt_disable()은 현재 프로세스의 preempt count를 증가시킬 것이며, check_highmem_ptes()는 CONFIG_DEBUG_HIGHMEM이 설정되었을 때 high memory 영역에 대한 접근을 확인하기 위한 부분이다. need_resched는 새로이 스케줄링을 해야할 경우에 제어가 넘어오는 label로서 사용된다. 현재 프로세스(current)를 prev에 놓고, 이 프로세스가 속한 run queue⁴⁸를 얻기 위해서 this_rq()를 호출한다. 복귀 값은 rq에 저장될 것이다. 만약 현재 프로세스가 커널에 대해서 lock을 설정한 것이 있다면, 이를 해제하기 위해서 release_kernel_lock()을 호출한다. 그리고, 현재 프로세스가 sleep하는 시간을 간직하기 위해서 sleep_timestamp에는 jiffies값을 기억시켜두도록 한다. 이젠 run queue에 대해서 연산을 할 것이므로, lock을 설정하기 위해서 spin_lock_irq()를 호출하도록 한다. 현재 프로세스의 preempt count값과 PREEMPT_ACTIVE를 AND시켜서 PREEMPT_ACTIVE가 설정되어 있는지 본다. 만약 설정되어 있다면, pick_next_task로 제어를 옮긴다. 그렇지 않다면, 현재 프로세스의 상태에 따라서 다음과 같은 일을 한다. 만약 현재 프로세스가 TASK_INTERRUPTIBLE이라면, 처리해야 할 signal이 있는가를 확인해서 있다면, 프로세스의 상태를 TASK_RUNNING으로 바꾼다. TASK_RUNNING인 경우에는 아무런 일을 해주지 않으며, default로는 deactivate_task()를 호출해서 프로세스의 포인터와 CPU에서 관리하는 run queue를 넘겨준다.

```

/*
 * deactivate_task - remove a task from the runqueue.
 */
static inline void deactivate_task(struct task_struct *p, runqueue_t *rq)
{
    rq->nr_running--;
    if (p->state == TASK_UNINTERRUPTIBLE)
        rq->nr_uninterruptible++;
    dequeue_task(p, p->array);
    p->array = NULL;
}

```

⁴⁸ Linux 커널 버전 2.5에서 CPU 별로 run queue를 가지는 구조로 변경되었다.

코드 222. deactivate_task() 함수의 정의

deactivate_task() 함수는 넘겨받은 run queue의 포인터에서 현재 running할 수 있는 프로세스의 개수를 줄이고(nr_running--) 프로세스의 상태가 TASK_UNINTERRUPTIBLE인 경우엔 nr_uninterruptible의 수를 증가 시켜준다. 이젠 dequeue_task()를 호출해서 프로세스를 run queue에서 빼어내도록 한다. 프로세스의 array 필드는 NULL을 가질 것이다.

```
/*
 * Adding/removing a task to/from a priority array:
 */
static inline void dequeue_task(struct task_struct *p, prio_array_t *array)
{
    array->nr_active--;
    list_del(&p->run_list);
    if (list_empty(array->queue + p->prio))
        __clear_bit(p->prio, array->bitmap);
}
```

코드 223. dequeue_task() 함수의 정의

dequeue_task() 함수는 프로세스를 run queue에서 제거하기 위해서 사용된다. 먼저, 우선 순위 별로 정렬된 프로세스를 가지는 배열에서 활성화(active)된 프로세스의 숫자를 감소시키고(array->nr_active--), run queue list와의 연결을 가지는 프로세스의 run_list필드를 이용해서 프로세스를 run queue에서 제거한다(list_del()). 만약 이렇게 제거된 프로세스가 속하는 우선 순위의 queue가 전체가 비게될 경우에는(list_empty()), 우선 순위의 queue에서 해당 우선 순위가 비었다는 것을 알리기 위해서 bitmap을 지운다. 즉, 이제 더이상 이 우선 순위에서는 실행할 프로세스가 없다는 것을 나타낸다.

여기서, 우린 몇가지의 자료구조가 새롭게 있음을 보았을 것이다. 먼저 run queue라는 자료구조가 있으며, 또한 우선순위 별로 프로세스를 관리하기 위한 priority array가 있다는 것을 알 수 있었다. 이를 조금더 자세히 보고 넘어가도록 하자.

```
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct runqueue {
    spinlock_t lock;                                /* Run queue에 대한 lock */
    unsigned long nr_running, nr_switches, expired_timestamp,
               nr_uninterruptible;                  /* 실행가능한 프로세스의 수,
                                                context switch가 일어난 회수,
                                                Expire된 time stamp들의 수,
                                                uninterruptible인 상태의 프로세스의 수 */
    task_t *curr, *idle;                            /* 현재 프로세스와 idle 프로세스의 포인터 */
    prio_array_t *active, *expired, arrays[2];      /* Active한 프로세스의 우선 순위 배열 포인터,
                                                Expired된 프로세스의 우선 순위 배열 포인터,
                                                우선 순위 배열 */
    int prev_nr_running[NR_CPUS];                   /* 이전에 실행중이던 프로세스의 CPU별 개수 */
    task_t *migration_thread;
    struct list_head migration_queue;              /* Migration(이주, 이전)된 프로세스의 포인터 */
                                                /* Migration되어 들어온 프로세스의 큐 */
} ____cacheline_aligned;
```

```

typedef struct runqueue runqueue_t;

static struct runqueue runqueues[NR_CPUS] __cacheline_aligned; /* CPU별 run queue의 정의 */

#define cpu_rq(cpu)          (runqueues + (cpu))      /* 특정 CPU의 run queue를 찾는다.*/
#define this_rq()            cpu_rq(smp_processor_id()) /* CPU ID를 가지고 run queue를 찾는다.*/
#define task_rq(p)           cpu_rq(task_cpu(p))       /* 특정 프로세스를 가지고 run queue를 찾는다.*/
#define cpu_curr(cpu)         (cpu_rq(cpu)->curr)      /* CPU에서 현재 실행중인 프로세스를
찾는다.*/
#define rt_task(p)           ((p)->prio < MAX_RT_PRIO) /* 현재 프로세스의 우선 순위를 비교한다.
*/

```

코드 224. runqueue 구조체의 정의

Run queue란 일반적으로 실행할 프로세스들이 들어있는 연결리스트로 구현된다. Linux에서는 각각의 CPU마다 이러한 run queue를 가지고 있다(runqueue[NR_CPUS]). 여기서 CPU특정 CPU에 시스템의 부하(load)가 집중되지 않도록 하기 위해서 load balancing이란 방법을 사용하기 위한 자료구조로서 migration_thread라는 task_t⁴⁹이란 타입에 대한 포인터를 가지는 필드가 있으며, migration_queue라는 큐를 가진다. 또한 각 CPU에서 현재 실행중인 프로세스의 개수를 파악하기 위한 prev_nr_running[]도 가진다.

```

#define BITMAP_SIZE (((MAX_PRIO+1+7)/8)+sizeof(long)-1)/sizeof(long)
struct prio_array {
    int nr_active;                      /* 큐에서 실행 가능한 프로세스의 수 */
    unsigned long bitmap[BITMAP_SIZE];   /* 우선 순위 별 프로세스의 여부를 알려주는 flag */
    struct list_head queue[MAX_PRIO];   /* 우선 순위 별 프로세스의 연결 리스트(list) */
};

```

코드 225. prio_array 구조체의 정의

prio_array 구조체는 우선순위(priority) 별로 프로세스를 관리하기 위해서 사용하는 큐(queue)이다. 이때 해당 우선순위에 대해서 실행준비가 된 프로세스가 있는가를 나타내기 위해서 bitmap이라는 필드를 사용한다. 만약 이 field의 특정 bit이 설정되었다면, 해당 우선순위의 큐에 실행준비가 된 프로세스의 연결 리스트가 있을 것이다. 여기서 우선 순위와 관련된 각종 상수는 아래와 같이 정의된다.

```

/*
 * Priority of a process goes from 0..MAX_PRIO-1, valid RT
 * priority is 0..MAX_RT_PRIO-1, and SCHED_NORMAL tasks are
 * in the range MAX_RT_PRIO..MAX_PRIO-1. Priority values
 * are inverted: lower p->prio value means higher priority.
 *
 * The MAX_RT_USER_PRIO value allows the actual maximum
 * RT priority to be separate from the value exported to
 * user-space. This allows kernel threads to set their
 * priority to a value higher than any user task. Note:
 * MAX_RT_PRIO must not be smaller than MAX_USER_RT_PRIO.
 */
#define MAX_USER_RT_PRIO      100
#define MAX_RT_PRIO           MAX_USER_RT_PRIO
#define MAX_PRIO               (MAX_RT_PRIO + 40)

```

코드 226. 최대 우선 순위들에 대한 상수 정의

MAX_USER_RT_PRIO는 최대 사용자 프로세스가 가질 수 있는 우선 순위 값을 표시한다. 현재는 100으로 설정되어 있다. MAX_RT_PRIO는 최대 real-time 프로세스가 가질 수 있는 우선 순위 값을

⁴⁹ task_t이란 task_struct 구조체를 나타낸다. 따라서, 프로세스를 가르킨다고 보면된다.

의미하며, MAX_USER_RT_PRIO와 같다. MAX_PRIO는 MAX_RT_PRIO에 40을 더한 값을 가지기에 140이 될 것이다. 중요한 점은 낮은 우선 순위 값을 가질 수록 실질적으로는 우선 순위가 더 높다는 점이다. 따라서, 프로세스의 우선 순위 값으로 가질 수 있는 값은 0에서 MAX_PRIO(=140)사이의 값이 되며, real-time 프로세스들이 얻을 수 있는 우선 순위는 0에서 MAX_RT_PRIO - 1(=99)의 값을 가질 것이다. 나머지 일반 사용자 프로세스들은 MAX_RT_PRIO(=100)에서부터 MAX_PRIO - 1 (=139)까지를 가질 것이다. MAX_RT_USER_PRIO는 위와 같이 정의할 경우, 일반 사용자 프로세스와 real-time 프로세스를 구분하는 경계가 되므로, 일반 사용자 프로세스보다는 항상 우선 순위가 높은 시스템 프로세스를 수행하기 위한 우선 순위 값으로 사용할 수 있을 것이다.

그럼, 이제 남는 것은 언제 프로세스가 이 우선 순위 배열과 관련을 맺는지를 보는 것이다. 이것은 wake_up_forked_process()라는 함수를 do_fork() 함수에서 호출하면서 이루어지게 된다. wake_up_forked_process()라는 함수를 잠시 보도록 하자.

```
/*
 * wake_up_forked_process - wake up a freshly forked process.
 *
 * This function will do some initial scheduler statistics housekeeping
 * that must be done for every newly created process.
 */
void wake_up_forked_process(task_t * p)
{
    runqueue_t *rq = this_rq_lock();

    p->state = TASK_RUNNING;
    if (!rt_task(p)) {
        /*
         * We decrease the sleep average of forking parents
         * and children as well, to keep max-interactive tasks
         * from forking tasks that are max-interactive.
         */
        current->sleep_avg = current->sleep_avg * PARENT_PENALTY / 100;
        p->sleep_avg = p->sleep_avg * CHILD_PENALTY / 100;
        p->prio = effective_prio(p);
    }
    set_task_cpu(p, smp_processor_id());
    activate_task(p, rq);

    rq_unlock(rq);
}
```

코드 227. wake_up_forked_process() 함수의 정의

wake_up_forked_process()는 새로 생성된(forked) 프로세스를 스케줄링이 될 수 있도록 만들어 주는 함수이다. 먼저 프로세스가 생성된 CPU의 run queue에 대한 lock을 얻은 후(this_rq_lock()), 프로세스의 상태를 TASK_RUNNING으로 바꾼다. rt_task()는 앞에서 보았듯이 MAX_RT_PRIO보다 작은 값인가를 확인하는 매크로이기에 이것이 0이라고 판단되면, real-time 프로세스가 아니라는 것을 의미한다. 이때는 현재 프로세스(fork())를 호출한 프로세스, 부모 프로세스의 sleep_avg를 PARENT_PENALTY(=100)와 곱해서 이를 다시 100으로 나눈 값으로 조정한다. 생성된 프로세스(child 프로세스)는 CHILD_PENALTY(=95)와 곱해서 100으로 나눈 값으로 sleep_avg가 조정되며, 우선 순위는 effective_prio()를 호출해서 얻은 값으로 두도록 한다. 만약 real-time 프로세스라고 생각된다면, 위와 같은 과정은 없을 것이다. set_task_cpu()는 생성된 프로세스가 실행될 CPU를 설정하기 위한 것이며, activate_task()에서 해당 프로세스가 활성화(active)된다. 마지막으로 앞에서 설정한 run queue에 대한 lock을 해제하기 위해서 rq_unlock()을 호출한다.

```
/*
```

```

* effective_prio - return the priority that is based on the static
* priority but is modified by bonuses/penalties.
*
* We scale the actual sleep average [0 .... MAX_SLEEP_AVG]
* into the -5 ... 0 ... +5 bonus/penalty range.
*
* We use 25% of the full 0...39 priority range so that:
*
* 1) nice +19 interactive tasks do not preempt nice 0 CPU hogs.
* 2) nice -20 CPU hogs do not get preempted by nice 0 tasks.
*
* Both properties are important to certain workloads.
*/
static inline int effective_prio(task_t *p)
{
    int bonus, prio;

    bonus = MAX_USER_PRIO*PRIO_BONUS_RATIO*p->sleep_avg/MAX_SLEEP_AVG/100 -
            MAX_USER_PRIO*PRIO_BONUS_RATIO/100/2;

    prio = p->static_prio - bonus;
    if (prio < MAX_RT_PRIO)
        prio = MAX_RT_PRIO;
    if (prio > MAX_PRIO-1)
        prio = MAX_PRIO-1;
    return prio;
}

```

코드 228. effective_prio() 함수의 정의

effective_prio() 함수는 프로세스의 우선 순위를 계산하기 위해서 호출되는 함수이다. 이때 계산되는 우선 순위는 다음과 같은 공식으로 계산된다.

$$\begin{aligned}
 & \frac{\text{MAX_USER_PRIO} \times \text{PRIO_BONUS_RATIO} \times p->\text{sleep_avg} - \text{MAX_USER_PRIO} \times \text{PRIO_BONUS_RATIO}}{\text{MAX_SLEEP_AVG} \times 100} \times 100 \times 2 \\
 &= \frac{\text{MAX_USER_PRIO} \times \text{PRIO_BONUS_RATIO}}{100} \times \left(\frac{p->\text{sleep_avg}}{\text{MAX_SLEEP_AVG}} - \frac{1}{2} \right)
 \end{aligned}$$

즉, 프로세스의 평균적인 sleep기간동안(sleep_avg)을 최대 sleep의 평균(MAX_SLEEP_AVG)의 값으로 나눈 값을 계산해서 이를 MAX_USER_PRIO와 PRIO_BONUS_RATIO를 곱해서 100분류로 나타내고, 이를 다시 MAX_USER_PRIO에 PRIO_BONUS_RATIO를 백분률로 만든 값의 50%를 빼서 우선 순위를 계산하는데 있어서 bonus 값으로 제공하게 된다는 것이다. 이것은 나중에 프로세스의 정적인 우선 순위(static_prio)에 음의 값으로 더해지기에 우선 순위를 높이는 영향을 미치게 된다. 만약 이렇게 계산된 우선 순위 값(prio)이 MAX_RT_PRIO보다 작다면, MAX_RT_PRIO를 주게되며, MAX_PRIO에 -1한 값보다 크다면, MAX_PRIO에 -1을 해서 조정된다. 즉, effective_prio()라는 함수가 호출되어 계산하는 우선 순위란 것은 일반적인 프로세스를 위한 값을 만드는데 사용되기에 실시간 프로세스가 가지는 우선 순위는 가지게 하지 못하게 만들어주는 것이다.

```

#define USER_PRIO(p) ((p)-MAX_RT_PRIO)
#define MAX_USER_PRIO (USER_PRIO(MAX_PRIO))
#define PRIO_BONUS_RATIO 25
#define MAX_SLEEP_AVG (2*HZ)

```

코드 229. 우선 순위 계산에 사용된 상수의 정의

MAX_SLEEP_AVG는 2에 HZ를 곱한 값을 사용한다. 여기서 재미있는 점은 2.4 버전에서 i386 CPU상의 커널에서는 HZ를 100으로 정의해서 사용하지만, 2.5 버전의 커널에서는 1000이란 값으로 사용한다는 것이다. 따라서, MAX_SLEEP_AVG는 2000이란 값을 가질 것이다. 또한 MAX_USER_PRIO는 USER_PRIO()라는 매크로를 사용해서 항상 0에서 39사이의 값을 가질 것이라는 것이다. 즉, MAX_RT가 1400이란 값을 가지므로, USER_PRIO()를 사용하면, MAX_USER_PRIO는 400이란 값을 가지게 될 것이다. 위의 공식을 보면, 한가지 값을 제외하고는 전부 상수라는 사실을 알 수 있을 것이다. 즉, 프로세스가 가지는 sleep_avg라는 값은 변동이 일어나게 되며, 이 값이 커질수록 bonus가 커지게되며, 결국 우선순위 값은 떨어지게 되어, 우선 순위 자체는 올라간다는 것이다. 따라서, 프로세스가 sleep한 평균적인 시간이 긴 프로세스가 우선 순위가 올라가게되는 효과가 발생한다. 즉, interactive한 프로세스 일수록 sleep하는 시간이 많아지기에 우선 순위가 상대적으로 높아지게 되는 현상이 생긴다.

```
/*
 * activate_task - move a task to the runqueue.
 *
 * Also update all the scheduling statistics stuff. (sleep average
 * calculation, priority modifiers, etc.)
 */
static inline void activate_task(task_t *p, runqueue_t *rq)
{
    unsigned long sleep_time = jiffies - p->sleep_timestamp;
    prio_array_t *array = rq->active;

    if (!rt_task(p) && sleep_time) {
        /*
         * This code gives a bonus to interactive tasks. We update
         * an 'average sleep time' value here, based on
         * sleep_timestamp. The more time a task spends sleeping,
         * the higher the average gets - and the higher the priority
         * boost gets as well.
         */
        p->sleep_avg += sleep_time;
        if (p->sleep_avg > MAX_SLEEP_AVG)
            p->sleep_avg = MAX_SLEEP_AVG;
        p->prio = effective_prio(p);
    }
    enqueue_task(p, array);
    rq->nr_running++;
}
```

코드 230. activate_task() 함수의 정의

activate_task() 함수는 wake_up_forked_process() 함수에서 호출된다는 것을 앞에서 보았다. 이 함수가 하는 일은 우선 순위를 새로 계산해서 프로세스를 run queue에 두는 일을 한다. 먼저 rt_task()를 사용해서 real-time 프로세스인가를 확인한다. real-time 프로세스가 아니고, sleep한 시간이 있다면(sleep_time), 프로세스의 sleep_avg에 현재 시간(jiffies)과 sleep한 시점(sleep_timestamp)와의 차이를 더하고, 이 값을 최대 sleep_avg값과 비교해서 더 적은 값으로 설정한다. 그리고나서, 우선 순위(prio)를 새로 계산하기 위해, 앞에서와 마찬가지로 effective_prio() 함수를 호출한다. Real-time 프로세스의 경우에는 아무런 영향을 미치지 못할 것이다. 마지막으로 run queue에 프로세스를 넣기 위해서 enqueue_task() 함수를 호출한다. run queue는 array가 가르키며, 넣게되는 프로세스는 p가 가르킨다. Run queue는 이제 실행 준비가 된 프로세스의 개수가 하나 증가할 것이다(nr_running++).

```
static inline void enqueue_task(struct task_struct *p, prio_array_t *array)
{
    list_add_tail(&p->run_list, array->queue + p->prio);
    __set_bit(p->prio, array->bitmap);
    array->nr_active++;
}
```

```
p->array = array;
}
```

코드 231. enqueue_task() 함수의 정의

enqueue_task() 함수는 run queue에 프로세스를 넣는 역할을 한다. 주의할 점은 이때 run queue에서 위치하는 곳은 프로세스의 우선 순위(p->prio)에 바탕한다는 점이다. list_add_tail()로 우선 순위에 해당하는 run queue의 위치의 제일 마지막에 프로세스를 연결한다. 즉, 큐의 꼬리(tail)에 프로세스가 연결될 것이다. 이전 실행준비가 된 프로세스가 run queue의 어떤 우선 순위에 있는가를 표시하기 위해서 __set_bit()을 호출하도록 한다. 넘겨주는 파라미터에 우선 순위값과 해당 bitmap에 대한 포인터가 있다. 우선 순위 배열에는 이제 active상태인 프로세스의 개수가 하나 증가할 것이며(array->nr_active++), 프로세스의 array필드는 자신이 들어있는 우선 순위 배열에 대한 포인터를 얻는다.

```
if (unlikely(!rq->nr_running)) {
#ifndef CONFIG_SMP
    load_balance(rq, 1);
    if (rq->nr_running)
        goto pick_next_task;
#endif
    next = rq->idle;
    rq->expired_timestamp = 0;
    goto switch_tasks;
}
array = rq->active;
if (unlikely(!array->nr_active)) {
    /*
     * Switch the active and expired arrays.
     */
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
}
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, task_t, run_list);
```

코드 232. schedule() 함수의 정의(계속)

schedule() 함수의 이야기로 다시 돌아가서, 이 부분까지 왔다면, 현재 프로세스에 대해서는 scheduling에 대한 처리를 다한 상황이다. 이전 수행할 프로세스를 찾는 일을 해야한다. 만약 수행할 프로세스가 하나도 없다면(unlikely(!rq->nr_running))이 참인 상황, SMP인 경우에는 다른 CPU에서 수행할 프로세스를 가져오기 위해서 load_balance()를 호출한 후, 실행할 프로세스의 개수를 가지는 nr_running이라는 run queue의 변수를 확인해서, 실행할 프로세스가 있다면, pick_next_task로 제어를 이동해서 다시 수행하도록 한다. 그렇지 않다면, 시스템에 전반적으로 실행할 프로세스가 없다는 말이되므로, 수행해야 할 프로세스를 가르키는 next에는 run queue가 가지는 IDLE 프로세스⁵⁰의 포인터를 얻게 된다. 이때 run queue의 timestamp가 만료된(expired) 값은 0으로 초기화되며, switch_tasks로 제어를 옮긴다.

이후의 과정은 run queue에 수행할 프로세스가 있다는 가정하에 run queue에서 우선 순위가 가장 높은 프로세스를 찾는 것이다. array는 run queue의 active된 프로세스를 가지는 queue를 가르키도록 만든다. 이 array의 nr_active가 0이라면, 역시 active한 프로세스가 없는 상황이므로, rq->active는 rq->expired로 바꾸고,

⁵⁰ 시스템의 IDLE 프로세스는 시스템의 초기화시에 생성되는 프로세스로 PID를 0을 가지는 프로세스이다. 이 프로세스는 scheduling에는 관여하지 않지만, 실행할 프로세스가 없을 경우에는 실행되어, 실행할 프로세스가 있을 때까지, 계속 scheduling() 함수를 호출하는 일을 수행할 것이다.

예전의 rq->active를 가지는 array로 rq->expired에 둔다. 즉, active와 expired priority array를 바꾸게 된다(swap). run queue의 expired_timestamp는 expired priority array가 새로 설정되므로 0이 된다. sched_find_first_bit()⁵¹ 함수는 run queue에서 첫번째로 우선순위가 높은 프로세스의 연결리스트가 있는 array의 index를 찾는 일을 한다. 이때 array가 가르키는 priority array의 bitmap필드가 참조되게 된다. 이렇게 얻은 인덱스(idx)를 array에 대한 index로 사용해서 run queue가 가지는 첫번째 우선 순위의 프로세스를 가져오게된다(list_entry()).

```
/*
 * Current runqueue is empty, or rebalance tick: if there is an
 * imbalance (current runqueue is too short) then pull from
 * busiest runqueue(s).
 *
 * We call this with the current runqueue locked,
 * irqs disabled.
 */
static void load_balance(runqueue_t *this_rq, int idle)
{
    int imbalance, idx, this_cpu = smp_processor_id();
    runqueue_t *busiest;
    prio_array_t *array;
    struct list_head *head, *curr;
    task_t *tmp;

    busiest = find_busiest_queue(this_rq, this_cpu, idle, &imbalance);
    if (!busiest)
        goto out;

    /*
     * We first consider expired tasks. Those will likely not be
     * executed in the near future, and they are most likely to
     * be cache-cold, thus switching CPUs has the least effect
     * on them.
     */
    if (busiest->expired->nr_active)
        array = busiest->expired;
    else
        array = busiest->active;

new_array:
    /* Start searching at priority 0: */
    idx = 0;
skip_bitmap:
    if (!idx)
        idx = sched_find_first_bit(array->bitmap);
    else
        idx = find_next_bit(array->bitmap, MAX_PRIO, idx);
    if (idx == MAX_PRIO) {
        if (array == busiest->expired) {
            array = busiest->active;
            goto new_array;
        }
        goto out_unlock;
    }
}
```

⁵¹ sched_find_first_bit() 함수는 ~/include/asm-i386/bitops.h에 assembly로 정의된 inline 함수이다. 총 140개의 bit을 검사해서 어떤 bit이 설정되어 있는가를 찾는다. 각각의 CPU별로 이 함수를 반드시 구현하도록 하고 있다.

```

head = array->queue + idx;
curr = head->prev;

skip_queue:
tmp = list_entry(curr, task_t, run_list);

/*
 * We do not migrate tasks that are:
 * 1) running (obviously), or
 * 2) cannot be migrated to this CPU due to cpus_allowed, or
 * 3) are cache-hot on their current CPU.
 */

#define CAN_MIGRATE_TASK(p,rq,this_cpu) \
((jiffies - (p)->sleep_timestamp > cache_decay_ticks) && \
 !task_running(rq, p) && \
 ((p)->cpus_allowed & (1UL << (this_cpu)))) \
\ \
\

curr = curr->prev;

if (!CAN_MIGRATE_TASK(tmp, busiest, this_cpu)) {
    if (curr != head)
        goto skip_queue;
    idx++;
    goto skip_bitmap;
}
pull_task(busiest, array, tmp, this_rq, this_cpu);
if (!idle && --imbalance) {
    if (curr != head)
        goto skip_queue;
    idx++;
    goto skip_bitmap;
}
out_unlock:
spin_unlock(&busiest->lock);
out:
;
}

```

코드 233. load_balance() 함수의 정의

```

switch_tasks:
prefetch(next);
clear_tsk_need_resched(prev);
RCU_qsctr(prev->thread_info->cpu)++;

if (likely(prev != next)) {
    rq->nr_switches++;
    rq->curr = next;
    prepare_arch_switch(rq, next);
    prev = context_switch(prev, next);
    barrier();
    rq = this_rq();
    finish_arch_switch(rq, prev);
} else
    spin_unlock_irq(&rq->lock);

```

```

    reacquire_kernel_lock(current);
    preempt_enable_no_resched();
    if (test_thread_flag(TIF_NEED_RESCHED))
        goto need_resched;
}

```

코드 234. schedule() 함수의 정의(계속)

prefetch()는 해당 데이터를 미리 CPU의 cache에 가져오라는 inline() 함수이다.⁵² clear_tsk_need_resched()는 프로세스의 thread_info 구조체의 flags 필드에 있는 TIF_NEED_RESCHED 지운다. RCU_qsctr()은 CPU별 RCU(Read Copy Update)에 관련된 자료구조를 갱신하기 위해서 호출된다.⁵³ 만약 현재 수행중인 프로세스와 다음에 수행할 프로세스가 같다면, run queue에 설정한 lock을 해제하고, 커널 lock을 해제한 후(reacquire_kernel_lock()), preempt_enable_no_resched()를 호출해서 preemption이 가능하도록 만든다. 이때 다시 한번 더 현재 프로세스의 thread_info 구조체에 있는 flags 필드에 TIF_NEED_RESCHED가 설정되어 있는지를 확인하고, 설정되어 있다면 need_resched로 제어를 옮겨서 다시 스케줄링을 할 것이다. 그렇지 않다면, schedule() 함수는 복귀하게 된다.

만약, 현재 schedule() 함수를 호출한 프로세스와 다음에 실행하도록 선택된 프로세스가 다르다면, if절 이하를 수행하게 된다. run queue의 context switch가 일어난 회수는 증가하게 되며(rq->nr_switch++), run queue의 현재 프로세스는 next(스케줄링에 의해서 선택된 프로세스)가 될 것이다(rq->curr = next). prepare_srch_switch()는 context switch를 하기 위한 architecture에 의존한 준비과정을 하는 것이며, context_switch()가 실제 context switch를 하는 부분이다. barrier()는 메모리의 update가 완전히 다 된다는 것을 보장할 것이며, this_rq()를 이용해서, 해당 run queue를 찾을 것이다. 마지막으로 finish_arch_switch()를 호출해서 context switch 과정을 완전히 마치게 된다.

```

/*
 * Default context-switch locking:
 */
#ifndef prepare_arch_switch
#define prepare_arch_switch(rq, next) do { } while(0)
#define finish_arch_switch(rq, next) spin_unlock_irq(&(rq)->lock)
#endif

```

코드 235. prepare/finish_arch_switch() 매크로의 정의

prepare/finish_arch_switch()는 SUN의 Sparc CPU를 사용하는 경우를 제외하면, 기본적으로(default) 별다른 일을 하지 않는 매크로로 정의된다. prepare_arch_switch()는 아무런 일도 하지 않지만, finish_arch_switch()는 run queue에 있는 lock을 해제하는 역할을 수행한다.

```

/*
 * context_switch - switch to the new MM and the new
 * thread's register state.
 */
static inline task_t * context_switch(task_t *prev, task_t *next)
{
    struct mm_struct *mm = next->mm;
    struct mm_struct *oldmm = prev->active_mm;

    if (unlikely(!mm)) {
        next->active_mm = oldmm;
        atomic_inc(&oldmm->mm_count);
    }
}

```

⁵² Intel Pentium이나 AMD의 Athlon 프로세스일 경우가 그렇다는 것이다. CPU마다 구현은 달라질 수 있다.

⁵³ RCU(Read Copy Update)와 관련된 곳으로 <http://lse.sourceforge.net/locking/rcupdate.html>을 참조하기 바란다.

```

        enter_lazy_tlb(oldmm, next, smp_processor_id());
    } else
        switch_mm(oldmm, mm, next, smp_processor_id());
    if (unlikely(!prev->mm)) {
        prev->active_mm = NULL;
        mmdrop(oldmm);
    }
    /* Here we just switch the register state and the stack. */
    switch_to(prev, next, prev);
    return prev;
}

```

코드 236. context_switch() 함수의 정의

context_switch() 함수는 가장 먼저 프로세스가 사용하는 주소공간(address space)을 교체하려는 작업을 진행한다. 만약 스케줄링된 프로세스(next)의 task_struct 필드에 있는 mm_struct를 가르키는 포인터(mm)가 NULL을 가진다면(즉, 이전에 수행되던 프로세스의 주소 공간을 공유하게 된다면), 이전의 프로세스가 가지고 있는 mm_struct의 포인터를 가지고, active_mm 필드를 설정한다. 이때, mm_struct 구조체의 reference count는 하나 증가할 것이다(atomic_inc()). enter_lazy_tlb()는 TLB(Translation Look-aside Buffer)를 refresh하는 방법을 설정하도록 만드는 부분이다. 그렇지 않다면, 주소 공간이 다른 프로세스가 스케줄링된 것이므로, switch_mm()을 호출해서, 주소공간을 교체(switch)한다. 만약 현재 실행중인 프로세스의 mm_struct를 가르키는 포인터가 NULL이라면, 즉, 현재 수행중이던 프로세스가 주소공간을 누군가와 공유하고 있다는 말이된다. 따라서, prev의 active_mm을 이젠 NULL로 두도록 한다. 즉, 주소공간을 공유하더라도, 더이상 실행되지 않는 상황이되면 active_mm이란 필드는 NULL로 설정하게 한다. mmdrop()은 mm_struct에 대한 reference count를 감소시키고, 더이상 사용되지 않으면 관련된 page table을 갱신하고, 구조체를 위해서 할당된 메모리를 해제하는 역할을 한다. switch_to() 매크로에서 실제로 CPU에 의존적인 context switch가 일어난다.

```

static inline void switch_mm(struct mm_struct *prev, struct mm_struct *next, struct task_struct *tsk, unsigned cpu)
{
    if (likely(prev != next)) {
        /* stop flush ipis for the previous mm */
        clear_bit(cpu, &prev->cpu_vm_mask);
#ifndef CONFIG_SMP
        cpu_tlbstate[cpu].state = TLBSTATE_OK;
        cpu_tlbstate[cpu].active_mm = next;
#endif
        set_bit(cpu, &next->cpu_vm_mask);
        /* Re-load page tables */
        load_cr3(next->pgd);
        /*
         * load the LDT, if the LDT is different:
         */
        if (unlikely(prev->context.ldt != next->context.ldt))
            load_LDT_nolock(&next->context, cpu);
    }
#ifndef CONFIG_SMP
    else {
        cpu_tlbstate[cpu].state = TLBSTATE_OK;
        if (cpu_tlbstate[cpu].active_mm != next)
            BUG();
        if (!test_and_set_bit(cpu, &next->cpu_vm_mask)) {
            /* We were in lazy tlb mode and leave_mm disabled
             * tlb flush IPI delivery. We must reload %cr3.
             */
            load_cr3(next->pgd);
        }
    }
#endif
}

```

```

        load_LDT_nolock(&next->context, cpu);
    }
#endif
}

```

코드 237. switch_mm() 함수의 정의

switch_mm() 함수는 스케줄링된 프로세스가 사용하는 주소공간(address space)이 이전에 수행중이던 프로세스의 주소공간과 다를 경우에, 스케줄링된 프로세스의 주소공간으로 바꿔주는 역할을 하는 함수이다. 먼저 교체할 두개의 주소공간이 같은 가를 확인하고(두개의 프로세스가 가르키고 있는 mm_struct가 다르다는 것을 확인한다.), 이전 프로세스 mm_struct 구조체에 있는 cpu_vm_mask 필드에, 현재 프로세스에 해당하는 bit field를 지우고(clear_bit()), SMP의 경우에는 CPU별 TLS 상태를 TLBSTATE_OK로 만들고 동시에 active_mm을 스케줄링된 프로세스의 mm_struct로 둔다. 이젠 현재 프로세스의 mm_struct에 있는 cpu_vm_mask에 현재 사용중인 CPU에 해당하는 bit를 설정하도록 한다(set_bit()). load_cr3()는 CR3 레지스터에 스케줄링된 프로세스의 PGD(Page Global Directory)를 넣는 일을 한다. 이것을 하고나면, 프로세스의 주소공간이 바뀌게(switch)된다. 스케줄링되었던 프로세스의 LDT가 이전에 수행중이던 프로세스의 LDT와 다르다면, load_LDT_nolock()은 프로세스를 위해서 LDT(Local Descriptor Table)을 하나 할당하고, 적재하는 일을 것이다.

나머지는 SMP인 경우에 해당하는 것이다. 즉, 스케줄링된 프로세스와 이전에 수행되던 프로세스의 주소 공간이 같다면, CPU별 TLB의 상태는 TLBSTATE_OK가 될 것이고, active_mm은 당연히 next와 같아야 할 것이다. 다르다면 에러가 된다(BUG!). 이전 스케줄링된 프로세스의 mm_struct에 있는 cpu_vm_mask필드에 CPU에 해당하는 bit을 설정한다. 만약 이전에 있던 값이 0이라면, 아직 CR3 레지스터가 적재가 되지 않은 것이므로, load_cr3()를 호출해서 PGD를 적재하도록 한다. load_LDT_nolock()함수는 LDT를 가져올 것이다. 여기까지 마치면, 주소공간에 대한 context switching은 마친것이 된다. 이후는 i386의 task management에 관련된 context switching을 할 차례가 되는 것이다.

```

#define switch_to(prev,next,last) do { \
    asm volatile("pushl %%esi\n\t" \
               "pushl %%edi\n\t" \
               "pushl %%ebp\n\t" \
               "movl %%esp,%0\n\t" /* save ESP */ \
               "movl %2,%esp\n\t" /* restore ESP */ \
               "movl $1f,%1\n\t" /* save EIP */ \
               "pushl %3\n\t" /* restore EIP */ \
               "jmp __switch_to\n\t" \
               "1:\n\t" \
               "popl %%ebp\n\t" \
               "popl %%edi\n\t" \
               "popl %%esi\n\t" \
               :"=m" (prev->thread.esp),"=m" (prev->thread.eip) \
               :"m" (next->thread.esp),"m" (next->thread.eip), \
                 "a" (prev), "d" (next)); \
} while (0)

```

코드 238. switch_to() 매크로의 정의

switch_to() 매크로는 ~/include/asm-i386/system.h에 정의되어 있다. i386 계열의 CPU상에서의 구현이므로, 다른 CPU에서는 달라질 수 있다. 먼저 현재 실행중인 프로세스의 레지스터들을 프로세스의 커널 stack에 저장하도록 한다. ESI, EDI, EBP을 저장하고, ESP는 프로세스의 thread 필드에 있는 ESP에 두고, 실행할 프로세스(next)의 thread 필드의 esp에서 저장된 ESP를 가져오도록 한다. EIP는 thread 구조체의 eip에 저장하는데, 이때는 다음번에 어디서 수행할 것인가를 결정하기 위해서 “1f”라는 곳에서 수행하도록 저장한다. 이전 ESP가 바뀐 상황이므로, next의 커널 stack에 thread 구조체가 가지는 eip를 저장하도록 한다. 이것은 __switch_to()가 호출된 후에 “ret”라는 assembly가 자동으로 수행되기에, ESP가 가르키는 곳에 다음 수행할 주소를 기억시켜두는 것이다. __switch_to()를 호출하는 순간까지는 아직

수행중인 프로세스의 context상이지만, 수행을 마치면 switch된 프로세스가 수행될 것이다. Context switch가 완전히 다 일어난 후에는 “1:”이라는 레이블(label)에서 스케줄링된 프로세스가 수행될 것이다. 저장된 EBP와 EDI, ESI를 차례로 복구한다.

```
/* ~/include/asm-i386/system.h 파일에서 */
extern void FASTCALL(__switch_to(struct task_struct *prev, struct task_struct *next));

/* ~/arch/i386/kernel/process.c 파일에서 */
void __switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
    struct thread_struct *prev = &prev_p->thread, *next = &next_p->thread;
    int cpu = smp_processor_id();
    struct tss_struct *tss = init_tss + cpu;

    /* never put a printk in __switch_to... printk() calls wake_up*() indirectly */
    unlazy_fpu(prev_p);
    /*
     * Reload esp0, LDT and the page table pointer:
     */
    tss->esp0 = next->esp0;
    /*
     * Load the per-thread Thread-Local Storage descriptor.
     */
    load_TLS(next, cpu);
    /*
     * Save away %fs and %gs. No need to save %es and %ds, as
     * those are always kernel segments while inside the kernel.
     */
    asm volatile("movl %%fs,%0":"=m" (*(int *)&prev->fs));
    asm volatile("movl %%gs,%0":"=m" (*(int *)&prev->gs));
    /*
     * Restore %fs and %gs if needed.
     */
    if (unlikely(prev->fs | prev->gs | next->fs | next->gs)) {
        loadsegment(fs, next->fs);
        loadsegment(gs, next->gs);
    }
}
```

코드 239. __switch_to() 함수의 정의

__switch_to() 함수는 FASTCALL이란 매크로를 사용해서 정의하고 있다. FASTCALL이란 매크로를 추적하면, ~/include/asm-i386/linkage.h에서 아래와 같은 것을 찾을 수 있을 것이다.

```
#define FASTCALL(x) x __attribute__((regparm(3)))
```

즉, compiler(GCC)에게 이 함수를 호출하기 위해서는 파라미터를 stack을 통해서 전달하지 말고, EAX, EDX, ECX 레지스터를 통해서 전달하라고 알려주는 것이다. 따라서, 앞에서 switch_to() 매크로를 보면, EAX에는 prev가 EDX에는 next가 있다는 것을 알 수 있기에, __switch_to() 함수의 prev_p와 next_p도 같은 것을 가르키게 될 것이다.

prev와 next는 prev_p와 next_p가 가르키는 task_struct 구조체의 thread 구조체 필드를 접근하기 위해서 사용한다. cpu는 현재 스케줄링을 수행하고 있는 CPU의 ID를 구한다(smp_processor_id()). 이것을 가지고, init_tss에 대한 index로 사용해서 tss_struct 구조체에 대한 포인터를 얻는다(tss). unlazy_fpu()는 만약 프로세스가 FPU를 사용하고 있었다면, 이를 저장(save)하도록 한다.

여기서 잠시, task_struct에 있는 thread_struct에 대해서 보기로 하자. 이 함수에서 계속 사용되는 thread_struct는 CPU별로 다르게 정의될 수 있으며, 사용자 모드에서의 context를 저장하기 위해서 사용한다. 정의는 ~/include/asm-i386/processor.h에 아래와 같이 되어 있다.

```
struct thread_struct {
    /* cached TLS descriptors */
    struct desc_struct tls_array[GDT_ENTRY_TLS_ENTRIES];
    unsigned long      esp0;
    unsigned long      eip;
    unsigned long      esp;
    unsigned long      fs;
    unsigned long      gs;
    /* Hardware debugging registers */
    unsigned long      debugreg[8]; /* %%db0-7 debug registers */
    /* fault info */
    unsigned long      cr2, trap_no, error_code;
    /* floating point info */
    union i387_union  i387;
    /* virtual 86 mode info */
    struct vm86_struct *vm86_info;
    unsigned long      screen_bitmap;
    unsigned long      v86flags, v86mask, saved_esp0;
    /* IO permissions */
    unsigned long      *ts_io_bitmap;
};
```

코드 240. thread_struct의 정의

thread_struct 구조체가 저장하고 있는 것은 위에서와 같이 GDT table에 대한 entry와 esp0, eip, esp, fs, gs, debug 레지스터들, fault 상황에 대한 정보를 가지는 필드들, 그리고, FPU 및 가상 86모드를 사용하는 프로세스를 위한 정보들, I/O permission을 나타내는 bit에 대한 포인터이다. 이것은 사용자 모드에서 사용했던 것들이거나, 혹은 사용자 모드에서 사용될 것들을 담고 있다는 것을 알 수 있을 것이다. Architecture에 의존적인 프로세스의 실행환경을 저장하기 위한 한 구조체로 여기기 바란다.

```
struct tss_struct {
    unsigned short      back_link,__blh;
    unsigned long       esp0;
    unsigned short      ss0,__ss0h;
    unsigned long       esp1;
    unsigned short      ss1,__ss1h;
    unsigned long       esp2;
    unsigned short      ss2,__ss2h;
    unsigned long       __cr3;
    unsigned long       eip;
    unsigned long       eflags;
    unsigned long       eax,ecx,edx,ebx;
    unsigned long       esp;
    unsigned long       ebp;
    unsigned long       esi;
    unsigned long       edi;
    unsigned short      es, __esh;
    unsigned short      cs, __csh;
    unsigned short      ss, __ssh;
    unsigned short      ds, __dsh;
    unsigned short      fs, __fsh;
    unsigned short      gs, __gsh;
    unsigned short      ldt, __ldth;
```

```

unsigned short    trace, bitmap;
unsigned long     io_bitmap[IO_BITMAP_SIZE+1];
/*
 * pads the TSS to be cacheline-aligned (size is 0x100)
 */
unsigned long __cacheline_filler[5];
};

```

코드 241. i386에서의 tss_struct의 정의

TSS(Task-State Segment)는 i386에서 프로세스(혹은 task)를 다시 원래대로 복구하기 위해서 필요한 자료구조(혹은 정보)를 저장하는 시스템 세그먼트이다. 각 CPU별로 이러한 TSS들은 init_task당 하나씩 주어져 있다. i386 계열의 CPU에서는 프로세스를 수행하기 위해서 반드시 이러한 TSS를 적어도 하나는 가지고 있어야 된다. 나중에 이것이 프로세스가 수행될 때는 Task Register(TR)로 들어가서 code, data, stack과 같은 것을 접근할 때 사용될 수 있을 것이다. 또한 각각의 특권 레벨별로도 해당 stack을 가질 수 있도록 i386 계열의 CPU는 허락하고 있다. 이와 더불어 사용하는 page들에 대한 정보를 가르키는 page directory를 포인트하기 위해서 CR3(Control Register 3)도 사용된다.⁵⁴ 어쨌든, 이러한 tss_struct 구조체는 정해진 형식이 있으며, 이를 소프트웨어적으로 구현하고 있는 모습을 보여준다.

다시 이전의 코드에 대한 설명으로 돌아가서, tss의 esp0에는 수행할 프로세스의 thread 구조체에 있는(next) esp0를 불러오고, load_TLS⁵⁵()를 호출해서 GDT(Global Descriptor Table)을 수행할 프로세스를 위해서 설정한다. 사용하던 FS 레지스터 및 GS 레지스터는 수행중이던 프로세스의 thread 구조체(prev)의 fs와 gs 필드에 저장되고, 수행할 프로세스의 thread 구조체에서 fs와 gs를 가지고와서 FS와 GS 레지스터를 채운다(loadsegment()). ES와 DS는 현재 커널 모드에서 진행중이라면, 항상 같은 값을 가질 것이므로 처리할 필요가 없다.

```

/*
 * Now maybe reload the debug registers
 */
if (unlikely(next->debugreg[7])) {
    loaddebug(next, 0);
    loaddebug(next, 1);
    loaddebug(next, 2);
    loaddebug(next, 3);
    /* no 4 and 5 */
    loaddebug(next, 6);
    loaddebug(next, 7);
}
if (unlikely(prev->ts_io_bitmap || next->ts_io_bitmap)) {
    if (next->ts_io_bitmap) {
        /*
         * 4 cachelines copy ... not good, but not that
         * bad either. Anyone got something better?
         * This only affects processes which use ioperm().
         * [Putting the TSSs into 4k-tlb mapped regions
         * and playing VM tricks to switch the IO bitmap
         * is not really acceptable.]
         */
        memcpy(tss->io_bitmap, next->ts_io_bitmap,
               IO_BITMAP_BYTES);
    }
}

```

⁵⁴ Intel에서 제공하는 System Programmer를 위한 Manual의 Task Management 부분을 보기 바란다.

⁵⁵ TLS(Thread Local Storage)란 i386 계열의 CPU에서 thread를 수행하기 위해서 필요한 코드나 데이터를 저장하는 segment register의 selector를 저장하는 GDT의 한 entry를 말한다. GDT에는 Kernel, APM, PNP 및 사용자 프로세스의 segment register에 대한 selector를 저장하는 entry가 있다. ~/arch/i386/kernel/head.S를 참조하기 바란다.

```

        tss->bitmap = IO_BITMAP_OFFSET;
    } else
    /*
     * a bitmap offset pointing outside of the TSS limit
     * causes a nicely controllable SIGSEGV if a process
     * tries to use a port IO instruction. The first
     * sys_ioperm() call sets up the bitmap properly.
     */
        tss->bitmap = INVALID_IO_BITMAP_OFFSET;
    }
}

```

코드 242. __switch_to() 함수의 정의

이전 debug 레지스터들을 load할 차례이다. 이것은 loaddebug()을 사용해서 처리한다. 마지막으로 I/O의 permission을 처리하기 위해서, 수행중이던 프로세스나 수행할 프로세스에 ts_io_bitmap이 있는가를 확인한다. 만약 있다면, 수행할 프로세스가 가진 I/O permission을 처리하기 위해서 tss의 io_bitmap 필드에 수행할 프로세스의 ts_io_bitmap을 복사한다. 만약 수행할 프로세스가 I/O permission을 가지는 bitmap을 지니고 있지 않다면, 당연히 tss의 io_bitmap에는 영향을 미치지 못한다. 전자의 경우에는 tss의 bitmap이 IO_BITMAP_OFFSET(tss에서 io_bitmap 필드의 offset)이 설정될 것이며, 후자의 경우에는 bitmap에 INVALID_IO_BITMAP_OFFSET(=0x8000)이 들어간다.

3. File System

파일 시스템은 파일을 디스크나 기타 미디어에 저장하는 방식과 그에 연관된 연산(operation) 및 자료구조를 포괄적으로 포함한다. 즉, 파일이 디스크에 저장되는 방식을 결정하며, 그것을 열고, 읽고, 쓰는 등등의 연산들과 이를 운영체제에서 유지하기 위한 자료구조까지도 가진다는 말이다. 이번 장에서는 리눅스에서 사용되는 파일 시스템에 대해서 자료구조 및 연관된 연산을 위주로 살펴볼 것이며, 또한 구체적인 예를 보게 될 것이다.

기본적인 리눅스의 파일 시스템은 ext2 파일 시스템이다. 하지만 이것 이외에도 리눅스는 많은 파일 시스템을 지원하고 있으며, 예로는 NTFS, VFAT, UFS, NFS 등등이 있다. 이것을 가능하게 하는 것이 바로 VFS(Virtual File System)이며, 이는 실제의 파일 시스템과 커널 사이에 가상의 파일 시스템층(layer)를 생성해서 커널에 대해서 일관된 인터페이스(interface)를 제공함과 더불어 파일 시스템의 개발자에게는 자신과 연관된 독특한 특성을 감추어주는 역할을 한다. 따라서, 커널은 하위의 파일 시스템이 어떤 것인가에 한정되지 않고, 일관되게 각각의 파일 시스템을 접근 할 수 있는 것이다.

3.1. VFS(Virtual File System)

가상의 파일 시스템이라는 말이다. 즉, 실제적으로 존재하지는 다만 연결고리의 역할을 하는 파일 시스템이라는 것을 암시한다. 커널은 사용자 프로세스의 요구를 받아서 있을지도모를 여러 파일 시스템에 대한 일관된 연산을 VFS를 통해서 할 수 있다. 하위의 파일 시스템은 커널에서 정의하고 있는 VFS 요구 사항만 맞춰준다면, 어떻게 구현되어도 상관없게 된다.

리눅스에서 파일 시스템으로 존재할 수 있는 것으로는 먼저 디스크 상에 존재하는 파일 시스템과 네트워크를 통해서 제공되는 서비스들에 의해서 접근할 수 있는 파일 시스템들, 그리고 마지막으로 특수한 파일(special file)들을 저장하는 파일 시스템으로 나눌 수 있다. 디스크상에 존재하는 파일 시스템으로는 각각의 하위 파일 시스템이 접근 가능한 형태로 커널에 의해서 표현되며, 이를 위해서 기본적으로 리눅스에 고유한 EXT2와 VFAT, CD-ROM(ISO9660 format), HPFS, UFS와 같은 것들이 있으며, 네트워크를 통해서 접근 가능한 파일 시스템으로는 NFS(Network File System), AFS(Andrew's File System), SMB(Samba File System), NCP(Novell NetWare Core Protocol) 파일 시스템 등이 있다.

특수 파일의 형태로 존재하는 파일 시스템으로는 리눅스의 각종 자원에 대한 종합적인 정보를 제공해주는 /proc 파일 시스템과 /dev 이하에 존재하는 각종 디바이스를 나타내는 파일들이 있다. 디바이스를 접근하기 위해서 사용되는 파일들이 이곳에 존재해서 사용자 프로그램에서는 이를 파일들에 대한 입출력 요구가 디바이스에 대한 연산으로 작용한다.

3.2. File System Objects

파일 시스템을 관리하기 위해서는 커널내에 프로세스를 관리하기 위한 자료구조와 마찬가지로 여러 가지 object들을 가지고 있어야 한다. 물론 이러한 object들은 서로 연관이 있으며, 특히 프로세스의 실행에 관련되어 있다. 모든 프로그램은 어떤 형식으로든 메인 메모리 속에 올라와야지만 실행이 되므로, 프로그램이 저장된 저장 매체로부터, 프로그램을 읽어서 실행하는 것은 당연히 파일 시스템에 대한 접근이 있어야만 가능하다.

[그림23]은 하나의 프로세스가 실행되면서 3개의 파일을 open한 상황을 보여준다. 실제로는 기본적으로 프로세스를 실행하게 될 때 3개의 파일 디스크립터(file descriptor)공간이 STDIN, STDOUT, STDERR에 할당이 되므로 그 이후에 여는(open)파일에 대한 것이라고 보면 될 것이다. 각각의 열려진 파일이 서로 다른 디렉토리에 저장되어 있다고 가정하여, 3개의 dentry object를 생성했다. 이러한 object들은 접근이 자주 일어나는 object들이므로, 캐쉬에 넣어서 관리하는 것이 편하다. 물론 여기서 말하는 캐쉬란 소프트웨어적으로 구현한 것을 말한다. 생성된 디렉토리 엔트리(Dentry)를 통해서 실제적으로 파일을 가리키는 Inode object에 대한 포인터를 얻어올 수 있으며, 다시 이를 이용해서 다시 파일로의 접근이 가능하다.

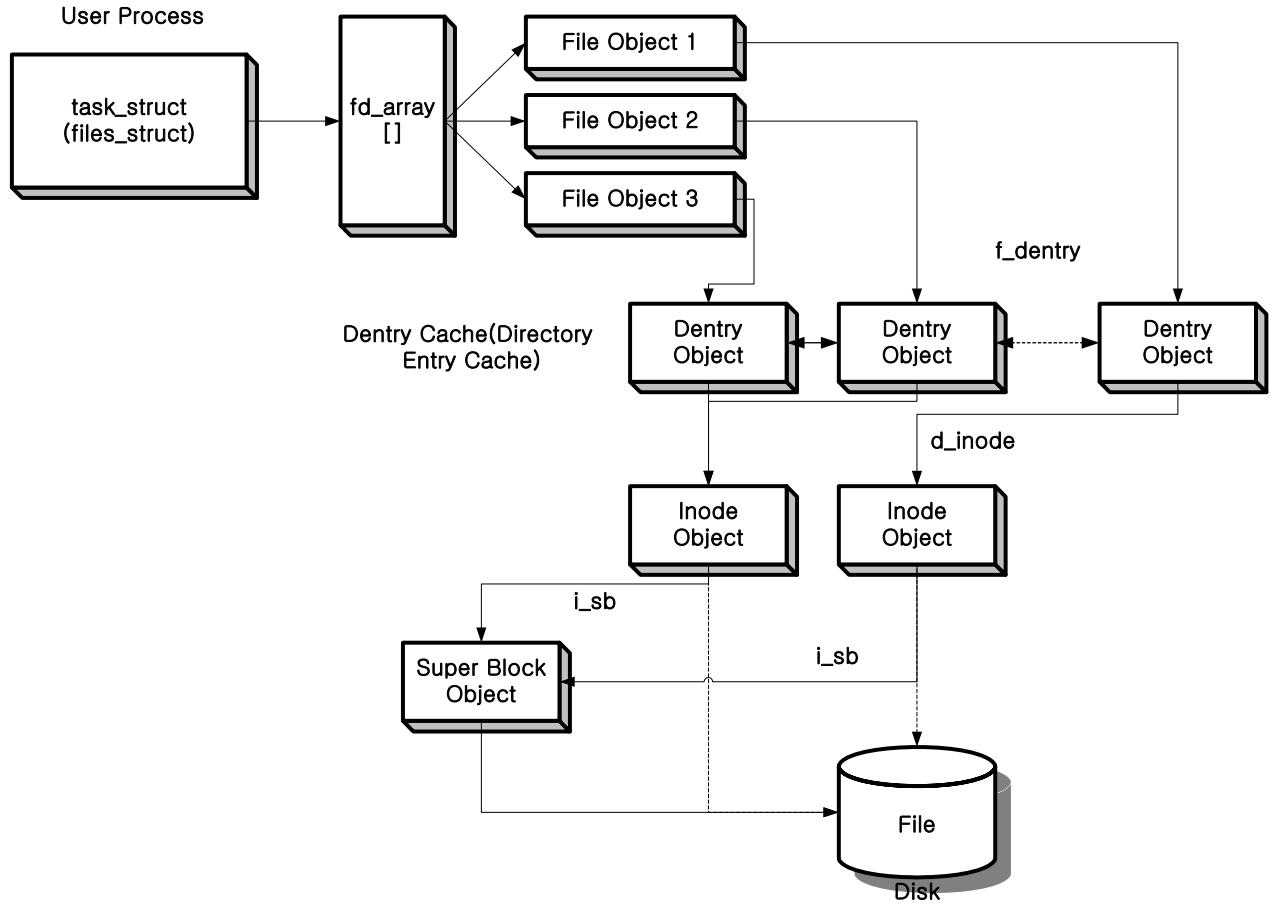


그림 29. File System을 구성하는 요소들.

위에서 대략적인 파일 시스템의 구조를 보았는데, 이제는 조금 더 자세히 들어가서 파일 시스템을 구성하는 각각의 구성요소에 대해서 더 자세히 알아보도록 하자.

3.2.1. File Object

이 object는 커널의 메모리 상에서만 존재하며, 프로세스와 열린 파일 간의 연결을 만들어주는 고리 역할을 한다. 파일 object는 파일 디스크립터 테이블에 연결되어 있으며, 프로세스가 파일에 대한 열기 연산을 할 때, 하나의 파일 object가 생성되어 프로세스의 파일 디스크립터 테이블에 들어가게 된다. 프로세스가 새로이 생성될 때, 파일 디스크립터가 상속되므로 자식 프로세스도 부모 프로세스와 같은 파일을 접근할 수 있다. Object에 대한 정의는 `~/include/linux/fs.h`에 있다.

```

struct file {
    struct list_head f_list;
    struct dentry *f_dentry;
    struct vfsmount *f_vfsmnt;
    struct file_operations *f_op;
    atomic_t f_count;
    unsigned int f_flags;
    mode_t f_mode;
    loff_t f_pos;
    unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
    struct fown_struct f_owner;
    uid_t f_uid, f_gid;
}

```

```

int          f_error;
unsigned long f_version;
/* needed for tty driver, and maybe others */
void         *private_data;
};

```

코드 243. File 구조체의 정의

주의할 점은 이 object에 대한 어떤 것도 디스크 상에 존재하지 않는다는 점이다. 즉, 이것은 커널이 열려진 파일에 대해서 연산을 하기 위해서 필요한 자료 구조체이다.

아래에 각각의 필드들에 대한 설명을 볼 수 있다.

Field	Description
struct list_head f_list	파일 object의 연결 리스트를 위한 자료구조
struct dentry *f_dentry	관련된 dentry object에 대한 포인터
struct vfsmount *f_vfsmnt	마운트 구조체를 위한 포인터
strcut file_operations *f_op	파일 연산에 대한 포인터(open,read,write,release,etc.)
atomic_t f_count	파일 object의 사용 계수(usage count)
unsigned int f_flags	파일을 열 때 사용한 flags
mode_t f_mode	프로세스의 access mode
loff_t f_pos	현재의 파일 offset (파일 포인터)
unsigned long f_reada, f_ramax, f_raend,f_ralen,f_rawin	파일에 대한 read ahead와 관련된 사항들을 가지는 데이터 구조로, read ahead flag과 read ahead될 최대 page의 수 마지막 read ahead후의 파일 포인터와 read ahead되는 byte의 수와 페이지를 나 타낸다.
struct fown_struct f_owner	시그널을 사용해서 파일에 대한 비동기(asynchronous)I/O를 하려고 할 때 사용하는 데이터
unsigned int f_uid,f_gid	파일 사용자의 user ID와 group ID
int f_error	네트워크를 통한 write연산시에 설정되는 error 코드를 저장
unsigned long f_version	파일의 현재 버전 번호, 이것은 사용시마다 하나씩 증가한다.
void *private_data	TTY 드라이버를 위한 데이터의 포인터

표 21. File 구조체의 필드 값

파일 object에 대한 이야기 중에서 read ahead라는 것은 현재 시점에서 파일의 사용이 미리 있을 것이다는 것을 염두에두고 파일에서 일정 내용을 사용되기 전에 미리 읽어오는 것이다. 이것은 시스템이 디스크에 대한 연산을 줄이는 효과가 있으며, 이렇게 함으로서 전체적인 I/O의 성능을 개선할 수 있다. 물론 이것은 전적으로 커널에서 행해야 하며, 사용자 프로세스는 모른다. 물론 이것을 하기위해선 커널의 메모리를 사용하게 되기에 비효율적으로 관리하면 오히려 역효과가 날 수 있다. 나중에 사용자 프로세스는 다시 파일에 대한 연산을 하려고 할 때 이곳에서 데이터를 가져올 수 있게 된다. 따라서,I/O가 일어나길 기다릴 필요없이 미리 읽어온 데이터를 가지고 연산을 계속 할 수 있게 되는 것이다.

파일 object의 자료 구조체중에서 나중에 디바이스 드라이버를 말할 때 등장하게 되겠지만, file operation구조체가 있다. 각각의 파일 시스템은 자신의 file operation구조체를 가질 수 있으며, 이 구조체에 자신만의 연산 구조체를 정의해서 자신의 파일 시스템에 맞는 고유한 연산을 할 있다. 이곳에 들어가는 연산 구조체는 inode object의 inode_operation의 default_file_ops구조체에서 가져오게 된다. 즉, inode를 커널이 디스크에서 메모리로 가져올 때 설정된다. 이곳에 들어갈 수 있는 연산으로는 llseek(), read(), write(), readdir(), poll(), ioctl(), mmap(), open(), flush(), release(), fsync(), fasync(), check_media_change(), revalidate(), lock()등이 있다. 이들에 대해서는 디바이스 드라이버를 살펴볼 때 자세히 보도록 한다.

3.2.2. Dentry Object

해당하는 파일에 대한 딕렉토리 정보를 가지는 object이다. 각각의 파일 시스템은 자신의 파일을 뮤어서 전체적인 구조를 저장하기 위한 구조를 가지고 있으며, 이 정보는 디스크의 일정 구역에 저장된다. Unix나 Windows NT 및 Linux에서는 이러한 정보를 딕렉토리라고 한다. 즉, 특정한 파일로서 딕렉토리를 다룬다. 딕렉토리의 내용은 각각의 파일을 나타내는 엔트리(entry)이다. 딕렉토리는 커널에 의해서 메모리 속으로 읽어드려지면 dentry라는 object로 변환되며, 코드는 ~/include/linux/dcache.h에 있으며, 아래와 같이 정의 된다.

```
struct dentry {
    atomic_t d_count;
    unsigned int d_flags;
    struct inode *d_inode; /* Where the name belongs to - NULL is negative */
    struct dentry *d_parent; /* parent directory */
    struct list_head d_vfsmnt;
    struct list_head d_hash; /* lookup hash list */
    struct list_head d_lru; /* d_count = 0 LRU list */
    struct list_head d_child; /* child of parent list */
    struct list_head d_subdirs; /* our children */
    struct list_head d_alias; /* inode alias list */
    struct qstr d_name;
    unsigned long d_time; /* used by d_revalidate */
    struct dentry_operations *d_op;
    struct super_block *d_sb; /* The root of the dentry tree */
    unsigned long d_reftime; /* last time referenced */
    void *d_fsdata; /* fs-specific data */
    unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
};
```

코드 244. Directory entry 구조체의 정의

각각의 필드가 하는 역할은 다음과 같다.

Field	Description
atomic_t d_count	Dentry Object의 사용계수
unsigned int d_flags	Dentry의 flag
struct inode *d_inode	파일 이름과 관련된 inode에 대한 포인터
struct dentry *d_parent	Parent directory에 대한 dentry 포인터
struct list_head d_vfsmnt	VFS mount에 대한 연결 구조체
struct list_head d_hash	Dentry를 찾기 위한 hash 연결 구조체
struct list_head d_lru	사용되지 않은 Dentry에 대한 연결 구조체
struct list_head d_child	Parent directory에 포함된 dentry object들에 대한 연결 구조체
struct list_head d_subdirs	하위(sub)directory에 있는 dentry에 대한 연결 구조체
struct list_head d_alias	관련된 inode들에 대한 연결 구조체
struct qstr d_name	파일의 이름
unsigned long d_time	d_revalidate 연산에 사용되는 필드로 어떤 시간이 흐른 뒤에 다시 유효한지를 검사하기 위한 것이다
struct dentry_operations *d_op	Dentry에 대한 연산을 정의한 구조체에 대한 포인터
struct super_block *d_sb	Super block object에 대한 포인터
unsigned long d_reftime	Dentry가 제거된 시점을 가리키는 시간
void *d_fsdata	파일 시스템에 의존적인 데이터 구조를 가르키는 포인터 (이 구조체에 파일 시스템에 고유한 데이터 구조를 저장해 둘 수

	있다.)
unsigned char d_iname[DNAME_INLINE_LEN]	짧은 파일 이름에 대해서 할당된 공간

표 22. Directory entry 구조체의 필드 값

Dentry는 프로세스가 파일 시스템에 접근해서 구조적인 정보(directory가 되겠다.) 읽어와서 생성하는 데이터 구조체이다. 실행중인 프로세스는 빈번한 접근이 일어나기에 이것을 캐쉬에 넣어서 보관하고 사용을 마쳤을 경우에는 사용이 끝났다는 것을 알리고, 다시 제 사용되기 위해서 연결리스트로 관리된다. 이러한 dentry구조체는 상태정보를 가지며, 아무런 유효한 정보를 가지고 있지 못할 때는 FREE, 커널에서는 사용하지 않지만 관련된 inode object에 대한 포인터를 가지고 있을 경우에는 UNUSED, 현재 커널에서 사용중이라면 INUSE, dentry와 관련된 inode object를 없지만 나중에 다시 이것을 사용할 가능성이 있을 경우를 대비해서 캐쉬내에 보관할 경우에는 NEGATIVE라는 상태가 주어진다. Dentry에 대한 연산(operation)으로는 d_revalidate(), d_hash(), d_compare(), d_delete(), d_compare(), d_release(), d_iput()이 있다. 이러한 연산에 대한 기본적인 연산은 미리 정의되어 있으며, 필요할 경우 이를 대치해서 특정한 일을 수행할 수 있다.

Dentry는 캐쉬 속에서 유지되기에 캐쉬의 효율적인 관리를 위해서 d_lru필드가 존재한다. 이는 LRU(Least Recently Used)라는 방법을 사용하기 때문에 붙여진 이름이며, 가장 최근에 사용된 것이 리스트의 가장 처음에 존재한다. 즉, 제일 마지막에 들어있는 것이 가장 오래전에 사용되었음을 나타낸다. 따라서, 만약 새로운 dentry object가 필요하다면 이 리스트의 마지막에서 가져와서 사용하면 될 것이다.

3.2.3. Inode Object

Inode object는 특정 파일에 대한 정보를 저장한다. 이곳에 저장된 정보는 디스크를 갖춘 경우 FCB(File Control Block)에서 읽어와서 생성되며, 하나의 파일에 대해서 하나의 inode object가 생성된다. 따라서, 특정 파일에 대한 연산을 돋기 위해서 inode ID라는 것을 가지고 파일을 나타낸다. Inode object는 아래와 같은 구조로 정의 된다. 코드는 ~/include/linux/fs.h에 있다.

```
struct inode {
    struct list_head i_hash;
    struct list_head i_list;
    struct list_head i_dentry;
    struct list_head i_dirty_buffers;
    unsigned long          i_ino;
    atomic_t               i_count;
    kdev_t                 i_dev;
    umode_t                i_mode;
    nlink_t                i_nlink;
    uid_t                  i_uid;
    gid_t                  i_gid;
    kdev_t                 i_rdev;
    loff_t                 i_size;
    time_t                 i_atime;
    time_t                 i_mtime;
    time_t                 i_ctime;
    unsigned long           i_blksize;
    unsigned long           i_blocks;
    unsigned long           i_version;
    struct semaphore        i_sem;
    struct semaphore        i_zombie;
    struct inode_operations *i_op;
    struct file_operations  *i_fop; /* former ->i_op->default_file_ops */
    struct super_block       *i_sb;
```

```

wait_queue_head_t      i_wait;
struct file_lock     *i_flock;
struct address_space   *i_mapping;
struct address_space   i_data;
struct dquot          *i_dquot[MAXQUOTAS];
struct pipe_inode_info *i_pipe;
struct block_device    *i_bdev;
unsigned long          i_dnotify_mask; /* Directory notify events */
struct dnotify_struct  *i_dnotify; /* for directory notifications */
unsigned long          i_state;
unsigned int           i_flags;
unsigned char          i_sock;
atomic_t               i_writecount;
unsigned int           i_attr_flags;
__u32                  i_generation;

union {
    struct minix_inode_info      minix_i;
    struct ext2_inode_info       ext2_i;
    struct hpfs_inode_info      hpfs_i;
    struct ntfs_inode_info      ntfs_i;
    struct msdos_inode_info     msdos_i;
    struct umsdos_inode_info    umsdos_i;
    struct iso_inode_info        isofs_i;
    struct nfs_inode_info        nfs_i;
    struct sysv_inode_info       sysv_i;
    struct affs_inode_info       affs_i;
    struct ufs_inode_info        ufs_i;
    struct efs_inode_info        efs_i;
    struct romfs_inode_info     romfs_i;
    struct shmem_inode_info     shmem_i;
    struct coda_inode_info       coda_i;
    struct smb_inode_info        smbfs_i;
    struct hfs_inode_info        hfs_i;
    struct adfs_inode_info       adfs_i;
    struct qnx4_inode_info       qnx4_i;
    struct bfs_inode_info        bfs_i;
    struct udf_inode_info        udf_i;
    struct ncp_inode_info        ncpfs_i;
    struct proc_inode_info       proc_i;
    struct socket                socket_i; /* BSD 소켓을 가진다. */
    struct usbdev_inode_info    usbdev_i;
    void                         *generic_ip;
} u;
};

```

코드 245.Inode 구조체의 정의

코드에서 보다시피 아주 많은 구조를 가지고 있으며, union으로 정의 된 부분은 특정한 파일 시스템에 대한 정보를 가지기 위해서 정의하고 있다. 즉, 공통적인 정보는 정의해서 가지며, 그렇지않고 파일 시스템에 의존적인 정보만을 구현에 알맞은 형태로 각각의 파일 시스템이 정의하도록 요구하고 있다. 각각의 필드들이 의미하는 것은 아래와 같다.(여기서는 각각의 파일 시스템에 고유한 것은 제외시킨다.)

Field	Description
struct list_head i_hash	Inode를 관리하기 위한 hash의 연결 리스트
struct list_head i_list	Inode들에 대한 연결 리스트
struct list_head i_dentry	Dentry 리스트에 대한 포인터
struct list_head i_dirty_buffers	Dirty한 inode들에 대한 연결 리스트
unsigned long i_ino	Inode의 번호
atomic_t i_count	Inode의 사용 계수
kdev_t i_dev	Inode의 디바이스 ID
umode_t i_mode	Inode가 가리키는 파일의 타입과 접근 권한
nlink_t i_nlink	Inode가 가지는 hard link의 수
uid_t i_uid	Inode의 사용자 ID
gid_t i_gid	Inode의 그룹 ID
kdev_t i_rdev	Inode가 가지는 실제 디바이스 ID
loff_t i_size	Inode가 가리키는 파일의 길이를 Byte로 표현
time_t i_atime, i_mtime, i_ctime	Inode가 가리키는 파일에 대한 마지막 접근 시간, 마지막 고침 시간, inode를 바꾼 시간
unsigned long i_blksize,i_blocks	블록 사이즈와, 파일 속에 있는 블록의 갯수
unsigned long i_version	각각의 사용시마다 증가하는 inode의 version number
struct semaphore i_sem, i_zombie	Inode에 대한 동기화를 위한 semaphore값
struct inode_operations *i_op	Inode연산 구조체에 대한 포인터
struct file_operations *i_fop	Inode와 관련된 파일 연산에 대한 포인터
struct super_block *i_sb	Inode가 언급하고 있는 파일 시스템의 super block object에 대한 포인터
wait_queue_head_t i_wait	Inode의 대기 큐(동기화-synchronization을 목적)
struct file_lock *i_flock	Inode가 가리키는 파일에 대한 lock을 탄내는 구조체에 대한 포인터
struct address_space *i_mapping,i_data	파일 mapping을 위한 구조체의 포인터와 데이터
struct dquot *i_dquot[MAXQUOTAS]	Inode의 disk quota를 위한 포인터
struct pipe_inode_info *i_pipe	Inode가 파이프(pipe)인 경우에 가지는 정보를 위한 포인터
struct block_device *i_bdev	Inode와 연관된 블록 디바이스를 가리키는 포인터
unsigned long i_dnotify_mask	Directory에 대한 notification을 위한 mask값
struct dnotify_struct *i_dnotify	Directory Notification에 사용할 구조체
unsigned long i_state	Inode의 현재 상태
unsigned int i_flags	파일 시스템의 파운트 flag
unsigned char i_sock	Inode가 socket을 나타낼 경우에는 TRUE값을 가진다.
atomic_t i_writecount	Write를 하고 있는 프로세스들에 대한 계수
unsigned int i_attr_flags	파일의 생성에 관련된 flag
u32 i_generation	RESERVED

표 23. Inode 구조체의 필드 값

모든 파일에 대한 연산을 하게 될 때 필요한 정보들은 inode object에 들어간다. 파일의 이름은 바꿀 수 있을지 모르나 해당하는 inode의 번호는 바뀌지 않는다. 따라서, 시스템 내에서 고유하다고 볼 수 있다. Inode는 시스템의 자원이며, 하나의 파일이 열리게 될 때마다 할당된다. 따라서, inode들은 사용상태에 있을 수 있고, 또한 사용되지 않는 상태로도 있을 수 있다. 또한 더 이상 유용한 정보를 가지고 있지 못하다고 할 경우 DIRTY상태로 있을 수도 있다. DIRTY상태에 있는 inode들은 디스크에 저장된 정보와 일관성을 유지하기 위해서 반드시 갱신(update)되어야 한다. Inode들은 번호로 해당하는 inode에 대한 접근을 가속하기 위해서 hash형태로 유지되고 있으며, inode는 고유한 inode연산을 가질 수 있다. 그리고, 해당하는 파일에 대한 연산을 유지하기 위한 필드도 정의하고 있다. 이것은 나중에 해당 파일에 대한 연산을 할 때 사용된다.

Inode에 대한 연산의 종류로는 create(), lookup(), link(), unlink(), symlink(), mkdir(), rmdir(), mknod(), rename(), readlink(), follow_link(), truncate(), permission(), revalidate(), setattr(), getattr()들이 있다. 이들 연산은 특정의 파일 시스템을 위해서는 이들중 일부만이 가능할 수 있으며, 나머지는 NULL로 정의될 수 있다.

이상에서 우리는 inode object에 대한 설명을 살펴보았는데, 이곳에서 나열된 필드이외에 나중에 예를 들어서 EXT2를 보게 될 때, union으로 정의된 ext2_inode_info를 다시한번 볼 수 있을 것이다.

3.2.4. Super Block Object

슈퍼블록 object는 마운트된 파일 시스템에 대한 정보를 가지는 object이다. 이것 역시 특정의 파일 시스템에 대한 control block의 정보를 가지고 mapping한다. 슈퍼블록 object에 대한 자료구조는 super_block으로 정의되어 있다. 전체 슈퍼블록들은 super_blocks라는 변수를 통해서 접근할 수 있다. 아래에 그 정의가 나와 있다. 코드는 ~/include/linux/fs.h에 있다.

```
extern struct list_head super_blocks;
#define sb_entry(list)    list_entry((list), struct super_block, s_list)
struct super_block {
    struct list_head s_list;           /* Keep this first */
    kdev_t             s_dev;
    unsigned long      s_blocksize;
    unsigned char      s_blocksize_bits;
    unsigned char      s_lock;
    unsigned char      s_dirt;
    struct file_system_type *s_type;
    struct super_operations *s_op;
    struct dquot_operations *dq_op;
    unsigned long      s_flags;
    unsigned long      s_magic;
    struct dentry      *s_root;
    wait_queue_head_t  s_wait;
    struct list_head   s_dirty; /* dirty inodes */
    struct list_head   s_files;

    struct block_device *s_bdev;
    struct list_head   s_mounts; /* vfsmount(s) of this one */
    struct quota_mount_options s_dquot; /* Diskquota specific options */
    union {
        struct minix_sb_info    minix_sb;
        struct ext2_sb_info     ext2_sb;
        struct hpfs_sb_info    hpfs_sb;
        struct ntfs_sb_info    ntfs_sb;
        struct msdos_sb_info   msdos_sb;
        struct isofs_sb_info   isofs_sb;
        struct nfs_sb_info     nfs_sb;
        struct sysv_sb_info    sysv_sb;
        struct affs_sb_info    affs_sb;
        struct ufs_sb_info     ufs_sb;
        struct efs_sb_info     efs_sb;
        struct shmem_sb_info   shmem_sb;
        struct romfs_sb_info   romfs_sb;
        struct smbfs_sb_info   smbfs_sb;
        struct hfs_sb_info     hfs_sb;
    };
}
```

```

        struct adfs_sb_info      adfs_sb;
        struct qnx4_sb_info      qnx4_sb;
        struct bfs_sb_info       bfs_sb;
        struct udf_sb_info       udf_sb;
        struct ncp_sb_info       ncpfs_sb;
        struct usbdev_sb_info    usbdevfs_sb;
        void                      *generic_sb_p;
    } u;
    struct semaphore s_vfs_rename_sem; /* Kludge */
    struct semaphore s_nfsd_free_path_sem;
};
```

코드 246. Super block 구조체의 정의

슈퍼블록 구조체가 지니는 각각의 필드에 대한 설명은 아래와 같다. 이곳에서도 특정 파일 시스템에 의존적인 데이터 구조는 살펴보지 않기로 한다.

Field	Description
struct list_head s_list	Super block의 리스트에 대한 포인터
kdev_t s_dev	Super block과 관련된 디바이스의 ID
unsigned long s_blocksize	블록의 사이즈를 byte로 나타낸 수
unsigned char s_blocksize_bits	블록의 사이즈를 나타내기 위한 bit의 수
unsigned char s_lock	Lock flag
unsigned char s_dirt	변경이 있었음을 나타내는 flag
struct file_system_type *s_type	Super block의 type
struct super_operations *s_op	Super block의 연산들에 대한 포인터
struct dquot_operations *dq_op	Disk quota를 위한 연산에 대한 포인터
unsigned long s_flags	마운트 flag
unsigned long s_magic	특정 파일 시스템을 나타내는 magic number
struct dentry *s_root	Mount directory에 대한 dentry object의 포인터
wait_queue_head_t s_wait	Mount wait queue
struct list_head s_dirty	변경된 inode들에 대한 연결 리스트
struct list_head s_files	Super block에서 열려진 파일들에 대한 file object들의 연결 리스트
struct block_device *s_bdev	블록 디바이스 구조체에 대한 포인터
struct list_head s_mounts	슈퍼블록들에 대한 mount의 연결 리스트
struct quota_mount_options s_dquot	디스크 쿼터를 위한 필드
struct semaphore s_vfs_rename_sem	Rename을 위한 세마포어
struct semaphore s_nfsd_free_path_sem	NFS로 마운트된 디렉토리에 대한 세마포어

표 24. Super block 구조체의 필드 값

파일 시스템을 마운트 한다는 말은, root가 되는 파일 시스템의 한 디렉토리에 다른 파일 시스템을 옮겨놓고 쓰겠다는 말이다. 즉, 그 디렉토리를 통해서 새로운 파일 시스템으로의 접근을 허용한다는 말이다. 가령 MS-DOS나 Windows에서 사용하는 파일 시스템을 하나의 디렉토리에 마운트 할 경우, 마치 마운트된 디렉토리로의 접근이 사용자 프로세스에게 하나의 디렉토리를 접근하는 것과 같이 보이도록 하는 것이다. 이렇게 해서 사용할 수 있는 것으로는 앞에서 언급한 파일 시스템 이외에 네트워크를 통해서 다른 시스템에 있는 파일 시스템도 디렉토리를 접근하는 것처럼 사용자 프로세스가 사용할 수 있다. 물론 이것은 시스템의 관리자가 개입되어야지만 가능하다.

슈퍼블록에 대한 연산은 super_operation이란 구조체로 정의가 되며, 이에 속하는 연산으로는 read_inode(), write_inode(), put_inode(), delete_inode(), put_super(), write_super(), statfs(),

`remount_fs()`, `clear_inode()`, `umount_begin()`이 있다. 이것도 마찬가지로 특정의 파일 시스템에 대해서 특정한 몇몇 연산만이 적용될 수 있으며, 나머지 정의되지 않은 연산들에 대해서는 NULL이 들어갈 수 있다.

이상에서 우리는 파일 시스템을 구성하는 각각의 object들이 어떤 관련을 가지고 연결되어 있는지를 살펴보았다. 이러한 각각의 object들은 커널이 관리하게되며, 파일의 생성과 삭제, 읽기와 쓰기등의 연산에 파라미터 값으로 정의된 함수로 넘겨진다. 사용자 프로세스의 입장에서는 커널의 내부에서 어떤 일이 행해지는 지는 관심이 없으며, 오로지 파일 디스크립터에만 역점을 둔다. 커널의 입장에서는 위에서 설명한 object들을 사용해서 사용자 프로세스의 요구를 만족시키는 일을 하게된다.

3.3. File System Initialization

파일 시스템의 초기화는 `~/init/main.c`에서 일어난다. 이 파일의 소스를 보면, `~/fs/filesystem.c`에 있는 `filesystem_setup()`함수를 호출하는 것으로 가능하다. 하지만 이것은 커널 2.4버전 이전에서 하던 방법이며, `~/init/main.c`에서 `filesystem_setup()`함수를 호출하기 전에 `do_initcalls()`함수를 호출한다. `do_initcalls()`함수는 `_initcall_start`와 `_initcall_end`사이에 등록된 각종 초기화 함수들을 하나씩 차례로 호출하는 일을 하는 함수로 이곳에서 대부분의 파일 시스템 초기화 함수들이 불려진다. `System.map` 파일을 살펴보면 알수 있듯이 `_initcall_start`와 `_initcall_end`사이에 파일 시스템의 초기화와 관련된 함수들이 들어있다. 다시 `_initcall_start`와 `_initcall_end`는 `~/arch/i386/vmlinux.lds`에 `ld` 유ти리티를 위한 `script`속에 정의되어 있다. 이곳에 각종 시스템 초기화 함수들이 들어간다.

지금까지는 일반적인 파일 시스템에 대한 설명을 했다. 다음 chapter에서는 실제적으로 구현된 EXT2 파일 시스템을 가지고, 특징 및 어떤 식으로 구현되어있는 가를 보도록 하겠다.

3.4. EXT2 File System

이곳에서는 물리적으로 어떻게 하드웨어에 파일이 저장되는지를 보도록 하겠다. 여기서 보게될 파일 시스템은 리눅스에서 가장 기본적인 EXT2 파일 시스템이다. 다른 파일 시스템은 다른 물리적인 구조를 가질 수 있으므로, 파일 시스템간에 호환성을 찾기는 힘들다. 실제 예로 MS-DOS같은 경우는 FAT(File Allocation Table)라는 것을 가지고 파일의 저장된 구조를 찾을 수 있게 되며, EXT2의 경우는 super block을 디스크상에 가지고 있어서, 이를 읽어야지 디스크 상에 저장된 파일을 접근할 수 있다.

초기의 리눅스 파일 시스템은 Minix에서 사용하던 파일 시스템이었으며, 이의 단점을 보완해서 새로이 개발된 것이 EXT이다. EXT2는 이와 같은 EXT의 새로운 버전이며, 리눅스만 가지는 파일 시스템이다. EXT2 파일 시스템은 일반적인 Unix에서 지원하는 파일 시스템을 이루는 구성 요소들⁵⁶을 지원하고 있으며, 아래와 같은 특성을 가지고 있다.

- 파일의 삭제시에 먼저 데이터를 완전히 임의의 데이터를 채운 후에 파일의 엔트리를 제거한다. 이것은 도스에서 파일의 첫글짜만 제거하고 데이터를 그대로 두는 것과는 차이가 난다. 따라서, 도스에서는 파일이 있는 디렉토리의 첫글자를 ‘?’로 둔 경우 지웠다고 표현하고, 나중에 이를 다시 복구하는 것이 가능하지만, 리눅스에서는 한번 지워진 파일에 대한 복구가 불가능하다.
- 동기화된 파일에 대한 쓰기가 가능하다. 즉, 파일에 가해진 변화는 동기적으로 실제 디스크 상의 파일에 같은 변화를 일으킬 수 있다.
- 파일에 대한 변동 및 지우기가 불가능 하도록 만들 수 있다.
- 파일에 대해서 덧붙이는 연산만이 가능하도록 만들 수 있다.
- 마운트 옵션을 통해서 새로운 파일에 관련된 그룹을 선택적으로 설정할 있다.
- 특정의 심볼릭 링크들은 전혀 데이터 블록을 사용하지 않을 수 있다. 즉, 디스크상의 inode에 파일의 이름을 담아서 쓸데없이 디스크의 블록을 낭비하지 않도록 한다.
- 각각의 파일 시스템의 상태는 기억되어, `fsck`와 같은 프로그램에서 파일 시스템을 검사하는 속도를 높여준다.

⁵⁶ 예를 들어서, 일반적인 파일과 디렉토리 및 특수 파일(이것에는 장치를 나타내는 파일도 포함된다.) 그리고, 심볼릭 링크가 있다.

- 마운트된 횟수와 최대 지연(delay)를 가지고 fsck 프로그램에서 파일 시스템을 언제 검사해야 할지를 결정해 줄 수 있다.
- 파일 시스템을 지탱하는 코드는 여러가 있는 상황에서는 읽기 전용으로 새로이 파일 시스템을 마운트하는 등의 행동을 통해서 시스템의 혼란을 막고, 데이터가 손상되지 않도록 해준다.

또한 이러한 것 이외에도 read ahead와 같은 것을 지원해서 시스템의 성능을 높여주며, 관련된 데이터 블록들을 연속적으로 저장할 수 있도록, 미리 여러개의 블록을 예약할 수 있는 기능도 가진다.

실제적으로 EXT2가 가지는 디스크 상의 구조는 아래의 [그림24]와 같다.

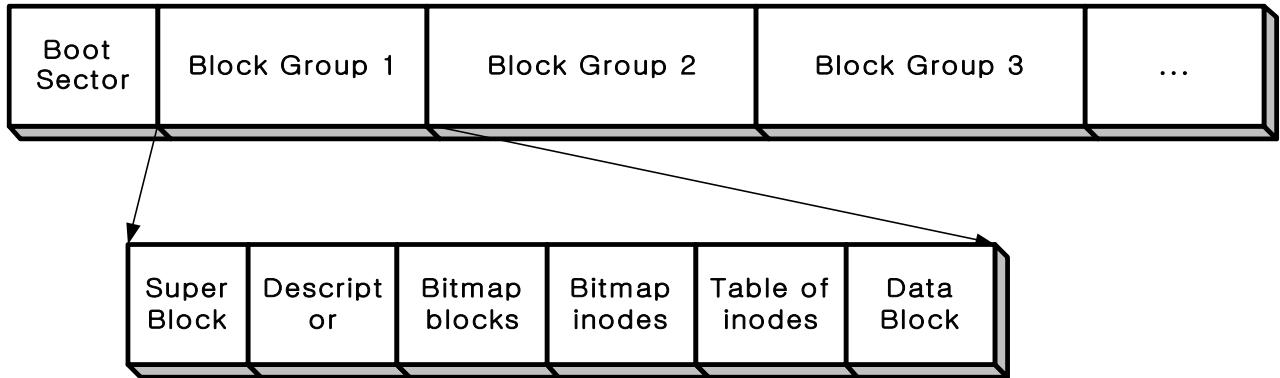


그림 30. Ext2 파일 시스템의 구조

즉, 실제로 디스크 상에서 EXT2 파일 시스템이 가지는 구조는 먼저 booting을 위한 boot sector 512 bytes와 여러개의 블록을 뭉쳐놓은 block group들로 구성된다. 각각의 블록 그룹이 가지는 구조는 먼저 super block의 복사본과 중요한 디스크 구성 정보를 가지는 블록에 대한 주소를 가지는 descriptor와 블록 및 inode의 사용 정보를 가지는 bitmap 블록, inode의 테이블, 데이터 블록으로 구성된다. 이중에서 super block과 descriptor는 전체 블록 그룹에 복사본을 가진다. 커널에서 사용하는 것은 단지 첫번째 블록 그룹에 대한 super block과 descriptor이다. fsck와 같은 프로그램을 사용해서 나머지 블록 그룹에 대한 동기화가 이루어진다.

3.4.1. EXT2의 Super 블록

Super 블록은 전체 블록들에 대한 정보를 가지는 블록이다. 즉, 파일 시스템의 제어 정보를 보관하고 있다. 디스크 상의 파일 시스템 시작부분에 위치하며, 각각의 블록 그룹에 그 복사본을 가진다. 만약 시작분에 가지고 있는 super 블록이 깨질 경우, 블록 그룹에서 다시 super 블록을 읽어서 복구할 수 있도록 만들어 준다. 슈퍼블록은 다음과 같은 정보를 가진다. 코드는 ~/include/.linux/ext2_fs.h의 ext2_super_block을 참조하기 바란다.

Field	Description
__u32 s_inodes_count	전체 inode의 수
__u32 s_blocks_count	전체 블록의 수
__u32 s_r_blocks_count	슈퍼 유저를 위해서 예약된 블록의 수(이것은 나중에 파일 시스템이 가득 차더라도 슈퍼 유저를 위해서 블록들을 남겨 연산을 할 수 있도록 할 목적으로 예약 되어 있다.)
__u32 s_free_blocks_count	전체 free 블록의 수
__u32 s_free_inodes_count	전체 free inode의 수
__u32 s_first_data_block	첫번째 데이터 블록의 번호
__u32 s_log_block_size	논리적인 블록의 크기
__s32 s_log_frag_size	논리적인 fragment의 크기
__u32 s_blocks_per_group	블록 그룹당 블록의 수

__u32 s_frags_per_group	블록 그룹당 fragment의 수
__u32 s_inodes_per_group	블록 그룹당 inode의 수
__u32 s_mtime	마지막으로 마운트된 시간
__u32 s_wtime	마지막으로 슈퍼 블록에 대한 write 연산이 행해진 시간
__u16 s_mnt_count	파일 시스템이 마운트된 회수
__s16 s_max_mnt_count	최대 마운트 회수
__u16 s_magic	파일 시스템을 나타내는 magic number(EXT2의 경우 0xEF53)
__u16 s_state	파일 시스템의 상태
__u16 s_errors	에러가 발생할 경우 취할 행동을 나타내는 flag
__u16 s_minor_rev_level	파일 시스템의 minor revision 번호
__u32 s_lastcheck	파일 시스템이 마지막으로 체크된 시간
__u32 s_checkinterval	파일 시스템의 체크 간격
__u32 s_creator_os	파일 시스템을 생성한 운영체제의 ID
__u32 s_rev_level	파일 시스템의 revision level
__u16 s_def_resuid	예약된 블록들에 대한 default 사용자 ID
__u16 s_def_resgid	예약된 블록들에 대한 default 그룹 ID
__u32 s_first_ino	첫번째 예약되지 않은 inode
__u16 s_inode_size	디스크 상의 inode의 크기
__u16 s_block_group_nr	이 슈퍼 블록에 있는 블록 그룹의 수
__u32 s_feature_compat	호환 feature의 집합(호환되는 feature들에 대해서 bit으로 표시)
__u32 s_feature_incompat	호환되지 않는 feature의 집합
__u32 s_feature_ro_compat	읽기 전용의 호환 feature의 집합
__u8 s_uuid[16]	파일 시스템의 ID(128bit)
char s_volume_name[16]	Volume name
char s_last_mounted[64]	마지막 마운트 포인트에 대한 디렉토리 이름
__u32 s_algorithm_usage_bitmap	압축을 위한 정보
__u8 s_prealloc_blocks	미리 할당된 블록의 수
__u8 s_prealloc_dir_blocks	디렉토리를 위해서 미리 할당된 블록의 수
__u16 s_padding1	1024 bytes를 맞추기 위한 padding
__u32 s_reserved[204];	블록의 끝까지 padding함

표 25. 디스크상의 Super Block의 구조

위에서 살펴본 것은 디스크 상에서 저장된 super 블록에 대한 정보이다. 이는 나중에 파일 시스템이 마운트 될 때, super 블록 object의 union으로 지정된 필드의 값으로 사용된다. 이전에 super block object를 다룰 때 미루었던 부분에 대해서 정리해 보면 아래와 같은 필드값이 있다. 코드는 ~/include/linux/ext2_fs_sb.h의 ext2_sb_info를 참고하기 바란다.

Field	Description
unsigned long s_flag_size	Fragment의 크기를 byte 단위로 표시
unsigned long s_flags_per_block	블록당 fragment의 수
unsigned long s_inodes_per_block	블록당 inode의 수
unsigned long s_frags_per_group	블록 그룹당 fragment의 수
unsigned long s_blocks_per_group	블록 그룹당 블록의 수
unsigned long s_inodes_per_group	블록 그룹당 inode의 수
unsigned long s_itb_per_group	블록 그룹당 inode table 블록의 수
unsigned long s_gdb_count	블록 그룹 디스크립터 블록의 수
unsigned long s_desc_per_block	블록당 그룹 디스크립터의 수
unsigned long s_groups_count	파일 시스템에 있는 블록 그룹의 수
struct buffer_head *s_sbh	슈퍼 블록을 가지고 있는 버퍼에 대한 포인터

struct ext2_super_block * s_es	버퍼내에 있는 슈퍼블록에 대한 포인터
struct buffer_head ** s_group_desc	그룹 디스크립터를 가지는 버퍼에 대한 포인터
unsigned short s_loaded_inode_bitmaps	현재 메모리에 들어있는 inode의 bimap
unsigned short s_loaded_block_bitmaps	현재 메모리에 들어있는 block의 bitmap
unsigned long s_inode_bitmap_number [EXT2_MAX_GROUP_LOADED]	inode bitmap의 캐쉬를 위한 필드
struct buffer_head * s_inode_bitmap [EXT2_MAX_GROUP_LOADED]	inode bitmap의 캐쉬를 위한 필드
unsigned long s_block_bitmap_number [EXT2_MAX_GROUP_LOADED]	block bitmap의 캐쉬를 위한 필드
struct buffer_head * s_block_bitmap [EXT2_MAX_GROUP_LOADED]	block bitmap의 캐쉬를 위한 필드
unsigned long s_mount_opt	마운트 옵션
uid_t s_resuid	예약된 블록에 대한 사용자 ID
gid_t s_resgid	예약된 블록에 대한 그룹 ID
unsigned short s_mount_state	파일 시스템의 마운트 상태
unsigned short s_pad	padding
int s_addr_per_block_bits	블록 bitmap에 대한 address
int s_desc_per_block_bits	블락 bitmap당 디스크립터의 수
int s_inode_size	inode의 크기
int s_first_ino	첫번째 inode의 번호

표 26. 메모리 상의 super block의 구조

이와 같은 super block object에 union구조체로 들어가는 정보는 당연히 메모리 상에 존재할 것이며, 나중에 슈퍼 블록에 대한 연산에 중요한 정보로 사용될 것이다.

[그림25]는 디스크 상의 super 블록과 메모리 상에 존재하는 super block object간의 관계를 보여준다. [그림25]에서 알 수 있듯이 super block object는 현재 마운트 하고 있는 파일 시스템에 대한 정보를 주게되며, 파일에서 대한 읽고, 쓰기, 및 생성등에 관련된 일련의 연산을 버퍼내에서 처리할 수 있다. 물론 이와 같은 버퍼는 결국에는 디스크 상의 정보와 동기화가 되어야 할 것이다. 버퍼에 대한 동기화는 ll_rw_block()함수를 호출하게 될 것이며, 결국 ll_rw_block()함수는 디바이스 드라이버를 접근한다.

한가지 유념해야 할 것은 항상 메모리와 디스크 간에 같은 정보를 가지고 있지는 않다는 점이다. 가장 최신의 정보는 메모리가 될 가능성이 많으며, 이는 어떤 식으로든 디스크 상의 정보를 업데이트 시켜주어야 한다는 것이다. 시스템은 이러한 연산을 뒤로 미루는 식으로 메모리 상에서 필요한 연산을 대부분 하려고 할 것이며, 또한 한번에 여러개의 블록을 읽어와서 메모리 버퍼에 저장하려고 할 것이다.

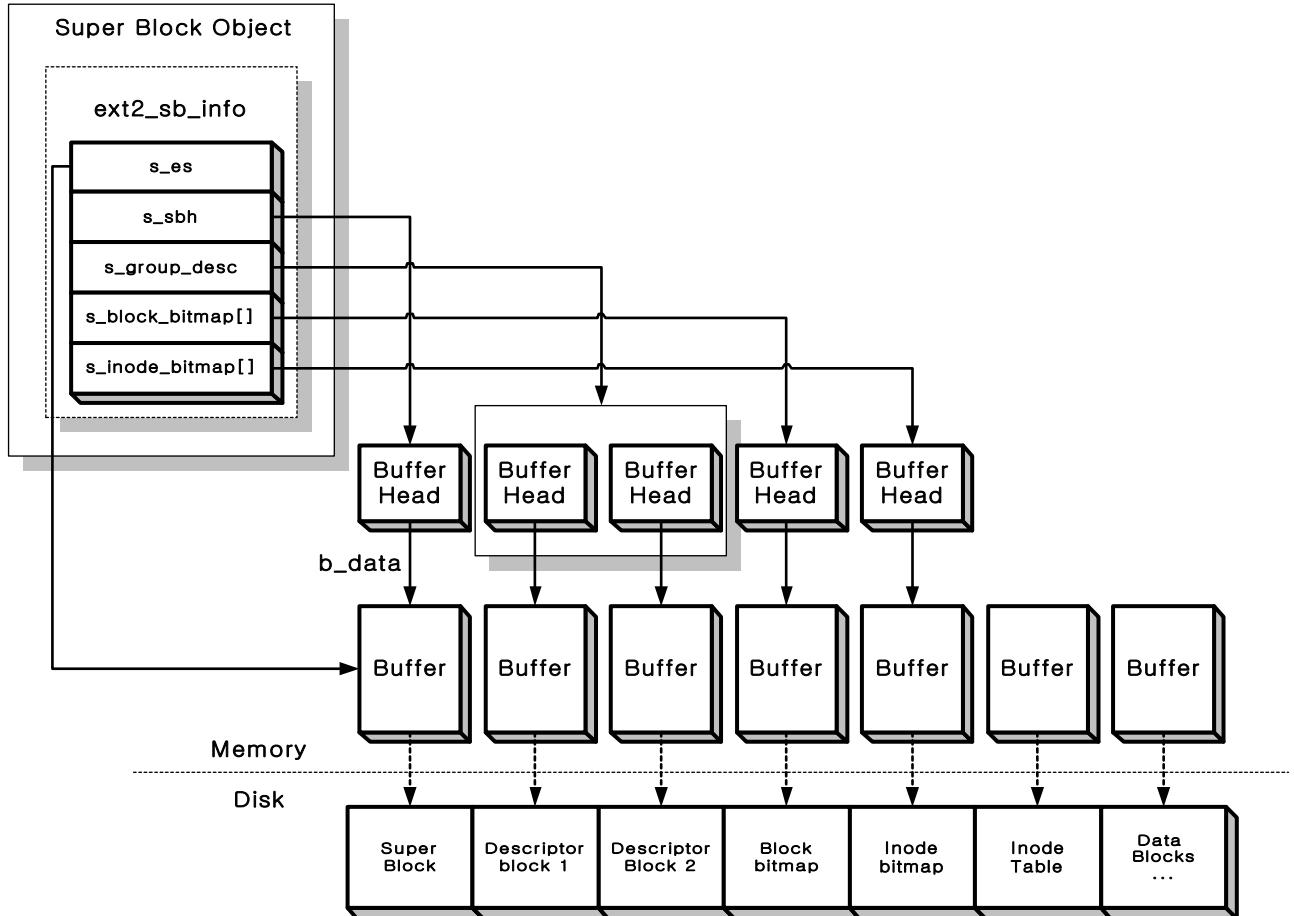


그림 31. EXT2 파일 시스템에서의 Super Block Object의 구조와 디스크 이미지의 관계

마찬가지로 inode object를 설명할 때 나열한 필드들 중에서 각 파일 시스템에 의존적인 정보가 있었는데, 이곳에도 `ext2_inode_info`라는 구조체가 정의되어 있다. 이것 역시 메모리 상에 당연히 존재하며, 아래와 같은 필드들이 존재한다. 코드는 `~/include/linux/ext2_fs_i.h`의 `ext2_inode_info`를 참고하기 바란다.

Field	Description
<code>__u32 i_data[15]</code>	데이터 블록을 가리키는 포인터들의 집합(직,간접,이중,삼중 간접)
<code>__u32 i_flags</code>	Inode의 flag
<code>__u32 i_faddr</code>	fragment의 주소
<code>__u8 i_frag_no</code>	fragment의 번호
<code>__u8 i_frag_size</code>	fragment의 사이즈
<code>__u16 i_osync</code>	디스크 상의 inode가 동기적(synchronous)하게 유지될 것인지를 나타냄
<code>__u32 i_file_acl</code>	파일의 access control list
<code>__u32 i_dir_acl</code>	디렉토리의 access control list
<code>__u32 i_dtime</code>	inode의 deletion 시간
<code>__u32 not_used_1</code>	사용되지 않는 필드로 자리만 차지함
<code>__u32 i_block_group</code>	inode가 속한 block 그룹에 대한 index
<code>__u32 i_next_alloc_block</code>	다음번에 할당될 블록의 번호
<code>__u32 i_next_alloc_goal</code>	다음번에 할당될
<code>__u32 i_prealloc_block</code>	블록을 미리 할당하기 위해서 사용됨

__u32 i_prealloc_count	블록을 미리 할당하기 위해서 사용됨
__u32 i_high_size	큰 파일 사이즈(64bit 크기의 파일 사이즈)를 지원하기 위한 필드(일반적으로 i_dir_acl)값을 가진다.
int i_new_inode:1	새로이 할당된 inode를 나타내는 flag

표 27. 메모리상의 ext2_inode_info구조체의 정의

ext2_inode_info구조체 역시 메모리 상에 존재하는 디스크 상의 inode구조라고 생각하면 될 것이다. inode object의 일부를 이루고 있으며, 파일 시스템에 의존적인 정보를 제공한다.

3.4.2. EXT2의 Descriptor 테이블 블록

각각의 블록 그룹들은 자신만의 블록을 이루는 그룹에 대한 디스크립터를 가지고 있다. 디스크립터 테이블 블록은 이러한 디스크립터들의 목록을 가진다. 그룹 디스크립터는 ext2_group_desc구조체로 ~/include/linux/ext2_fs.h에 정의되어 있으며, 아래와 같은 필드들을 가진다.

Field	Description
__u32 bg_block_bitmap	블록 bitmap의 블록 번호
__u32 bg_inode_bitmap	inode bitmap의 블록 번호
__u32 bg_inode_table	inode table의 블록 번호
__u16 bg_free_blocks_count	그룹에 속한 free 블록의 수
__u16 bg_free_inodes_count	그룹에 속한 free inode의 수
__u16 bg_used_dirs_count	그룹에서 디렉토리를 위해서 사용된 블록의 수
__u16 bg_pad	padding
__u32 bg_reserved[3]	RESERVED(24byte로 만들기 위한 padding)

표 28. 디스크 상의 Descriptor 구조체의 정의

위에서 보인 그룹 디스크립터 테이블의 필드들 중에서 bg_inode_table은 나중에 살펴볼 inode table에 대한 인덱스(index) 역할을 한다.

3.4.3. EXT2의 Bitmap 블록

블록이나 inode가 할당된 상태를 나타내는 bit의 연속적인 블록이다. 이것은 하나의 블록에 저장되므로 블록의 크기에 따라서 나타낼 수 있는 블록이나 혹은 inode의 상태 갯수가 정해진다. 즉, 0일 경우는 해당하는 블록이나 inode가 할당되지 않았음을 나타내며, 1일 경우에는 해당하는 블록이나 inode가 할당되어서 사용되고 있음을 가리킨다.

커널은 이와 같은 bitmap block을 메모리내로 읽어드려서 새로운 블록이나 inode를 할당하고자 할 때, 빈공간을 찾기위해서 사용할 수 있으며, 지워진 블록이나 inode를 명시하기 위해서도 사용한다. 물론 지우기나 쓰기연산은 메모리내에서 일어나게 되므로, 나중에 디스크에 직접적으로 쓰기 연산을 해서 동기화를 시켜주어야 할 것이다. 만약 시스템이 불안정해서 갑작스럽게 down되는 경우 메모리내에 있던 bitmap이 디스크와 일치하지 못하는 경우가 생길 수 있으며, 이는 데이터의 손실을 가져올 것이다.

3.4.4. EXT2의 Inode 테이블

Inode 테이블은 블록의 연속적인 집합으로 각각의 블록은 미리 정의된 수의 inode를 가진다. 첫번째 inode 테이블을 가지고 있는 블록의 주소는 앞에서 설명한 descriptor에 지정되어 있으며, 나중에 메모리에 유지되는 inode object를 위한 정보를 제공하는 역할을 한다.

Field	Description
__u16 i_mode	파일의 타입과 접근모드
__u16 i_uid	파일의 사용자 ID

<code>__u32 i_size</code>	파일의 크기를 byte 단위로 나타냄
<code>__u32 i_atime</code>	마지막으로 파일에 접근한 시간
<code>__u32 i_ctime</code>	파일이 생성된 시간
<code>__u32 i_mtime</code>	마지막으로 파일에 변동이 있었던 시간
<code>__u32 i_dtime</code>	파일이 지워진 시간
<code>__u16 i_gid</code>	파일의 그룹 ID
<code>__u16 i_links_count</code>	파일에 대한 hard link의 수
<code>__u32 i_blocks</code>	파리에 할당된 블록의 수
<code>__u32 i_flags</code>	파일의 flag
<code>osd1</code>	특정의 운영체제에 대한 정보(linux1, hurd1, masix1이라는 구조체가 union으로 묶여 있음)
<code>__u32 i_block[EXT2_N_BLOCKS]</code>	블록에 대한 포인터(즉, 실제로 할당된 블록들에 대한 포인터로 구성된다. 직접, 간접, 이중 간접, 삼중 간접으로 블록들에 대한 포인터를 가진다.)
<code>__u32 i_generation</code>	파일의 버전 정보(NFS를 위해서 사용됨)
<code>__u32 i_file_acl</code>	파일의 access control list
<code>__u32 i_dir_acl</code>	디렉토리에 대한 access control list
<code>__u32 i_faddr</code>	fragment의 주소
<code>osd2</code>	특정의 운영체제에 대한 정보(linux2, hurd2 ⁵⁷ , masix2라는 구조체가 union으로 묶여 있음)

표 29. 디스크 상의 inode table의 구조

관련된 메모리 상의 inode object의 union 필드를 채워주는 ext2_inode_info는 앞에서 이미 살펴보았다. 데이터가 inode과 연관되어 저장될 때는 i_block을 이용해서 저장되며 [그림26]과 같은 구조를 가진다. 이에 사용되는 상수값은 아래와 같다

<code>#define EXT2_NDIR_BLOCKS</code>	12
<code>#define EXT2_IND_BLOCK</code>	EXT2_NDIR_BLOCKS
<code>#define EXT2_DIND_BLOCK</code>	(EXT2_IND_BLOCK + 1)
<code>#define EXT2_TIND_BLOCK</code>	(EXT2_DIND_BLOCK + 1)
<code>#define EXT2_N_BLOCKS</code>	(EXT2_TIND_BLOCK + 1)

즉, 처음의 12개의 entry는 직접적으로 데이터 블록을 가리키게 되며, 나머지들은 12, 13, 14의 entry를 가지고, 간접, 이중 간접, 삼중 간접으로 데이터 블록의 포인터를 가지는 데이터 블록을 가리키도록 만든다. 전체 entry의 수는 15가 된다.

⁵⁷ GNU에서 자체적으로 개발하고 있는 운영체제이다. 리눅스는 GNU에서 지원하는 운영체제의 하나일 뿐이다.

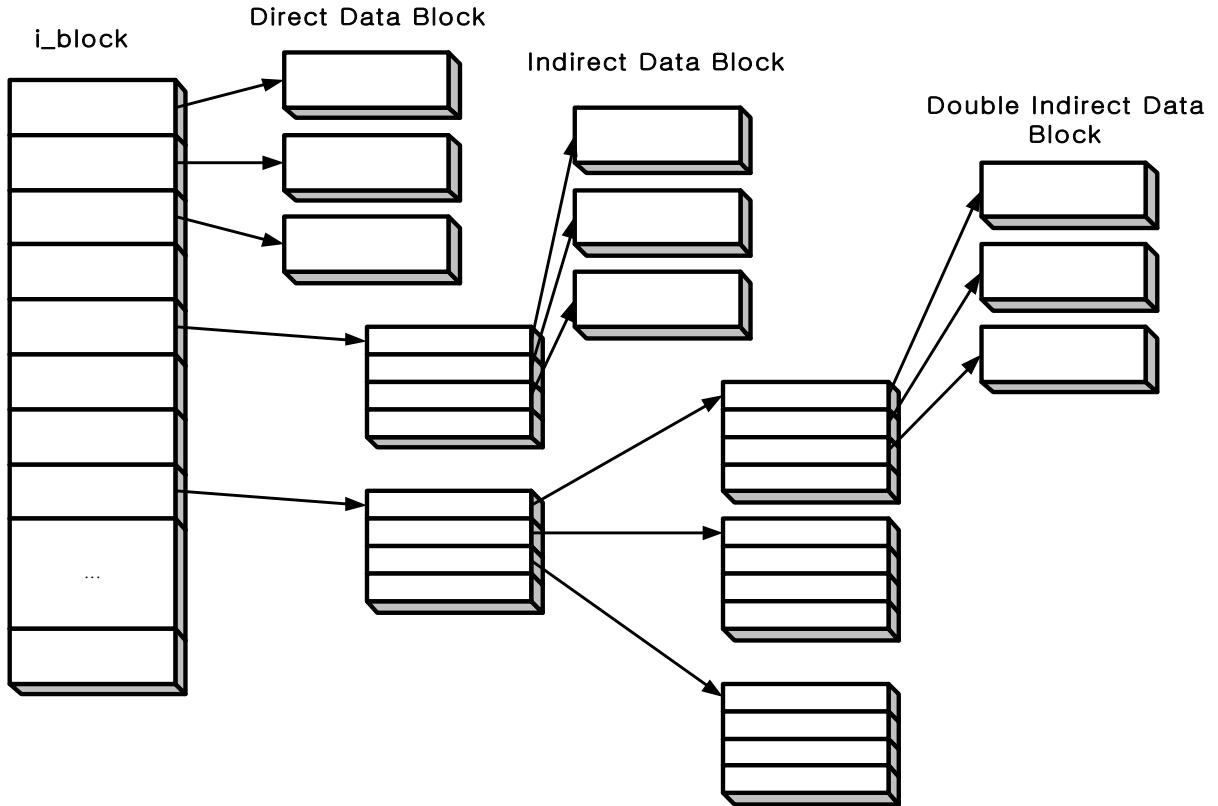


그림 32. i_block필드가 가리키는 데이터 블록

이와 같은 구조는 일반적인 유닉스를 구성하는 inode와 같다. 최대 파일이 가질 수 있는 크기는 일반적일 경우 i_size필드로 표기되기에 32bit값인 4GBytes까지 될 수 있다. 하지만, 최상위 bit이 사용되지 않기에 2GBytes로 한정된다. 물론 64bit machine을 위해서 I_dir_acl필드를 사용해서 파일의 크기를 64bit으로 표현할 수도 있으나, 32bit 구조의 시스템에서는 지원되지 않는다.

VFS와 같은 파일 시스템에서는 특정의 inode를 찾기 위해서 전체 시스템에서 고유한 inode번호를 사용하는데, 이 번호는 이상에서 설명했던 블록의 그룹번호와 그 그룹에 속한 inode 테이블의 index로 표시될 수 있을 것이다.

지금까지 리눅스의 파일 시스템의 구성과 실제적으로 사용되는 EXT2 파일 시스템에 대해서 알아보았다. 관련된 것으로 더 보아야 할것이 특수 파일이 디바이스를 나타내는 파일이다. 이는 나중에 디바이스 드라이버를 설명하면서 자세히 보도록 할 것이다.

4. Memory Management

메모리란 프로그램의 임시적인 이미지를 저장하기 위해서 사용하는 시스템의 자원이다. 따라서, 이것도 하나의 관리요소가 되며, 운영체제는 이를 효과적으로 관리해 주어야 한다. 이를 위한 자료구조 및 CPU에 의존적인 기능을 가지고 있으며, x68의 경우에는 세그먼테이션(segmentation)과 페이지를 동시에 사용해서 메모리를 관리하도록 하고 있다. 특히 하드웨어에 의존적인 것들이 많기에 개념적인 것만을 보아도 좋을 거라 생각한다.

4.1. Segmentation

주소란 명령어나 데이터가 메모리상에서 위치하는 공간을 가리킨다. 즉, 주소를 가지고 모든 수행할 명령어와 데이터를 찾아가게 되는 것이다. 주소에는 크게 3가지가 존재한다. 각각은 논리적인 주소(logical address), 선형 주소(linear address), 물리적인 주소(physical address)이다. 논리적인 주소는 프로세스가 보는 주소이며, 선형주소는 운영체제에서 보는 주소이며, 물리적인 주소는 실제 하드웨어상의 주소이다. 아래와 같이 정의된다.

- 논리적인 주소(Logical Address) – 세그먼트(segment)와 오프셋(offset)으로 구성되며, 세그먼트 경계로 얼마만큼 떨어져 있느냐(offset)를 나타내는 주소 지정 방식이다. x86과 구조에서는 세그먼트를 나타내는 레지스터와 오프셋을 나타내는 레지스터가 구분되어 사용된다.
- 선형주소(Lineare Address) – 세그먼트와 오프셋으로 나타내진 주소를 단일의 32bit 주소로 만든 것으로 4Gbytes까지 지정이 가능하다⁵⁸. 0x00000000에서 0xFFFFFFFF까지를 말한다. 선형주소는 전체 프로세스가 접근할 수 있는 영역을 말하며, 리눅스의 경우는 3Gbytes를 프로세스에서, 나머지 1Gbytes를 커널에서 사용한다.
- 물리적인 주소(Physical Address) – 시스템에 있는 전체 물리적인 메모리 공간을 말한다. 이 역시 32bit의 형태로 나타나며, 시스템의 주소 버스(Address Bus)에 전기적인 신호(signal)로 메모리의 특정 주소를 나타낸다. 이것은 현재 시스템에 설치된 메모리의 크기로 한정된다.

이상에서 이야기한 세가지를 요약하면 [그림27]과 같다. 즉, 논리적인 주소는 선형주소로 맵핑(mapping)되며, 다시 물리적인 주소 공간으로 맵핑된다.

⁵⁸ 물론 이것은 Intel의 32bit 구조상에서의 이야기다. 64bit주소를 만들어낼 때는 달리 사용될 수 있다.

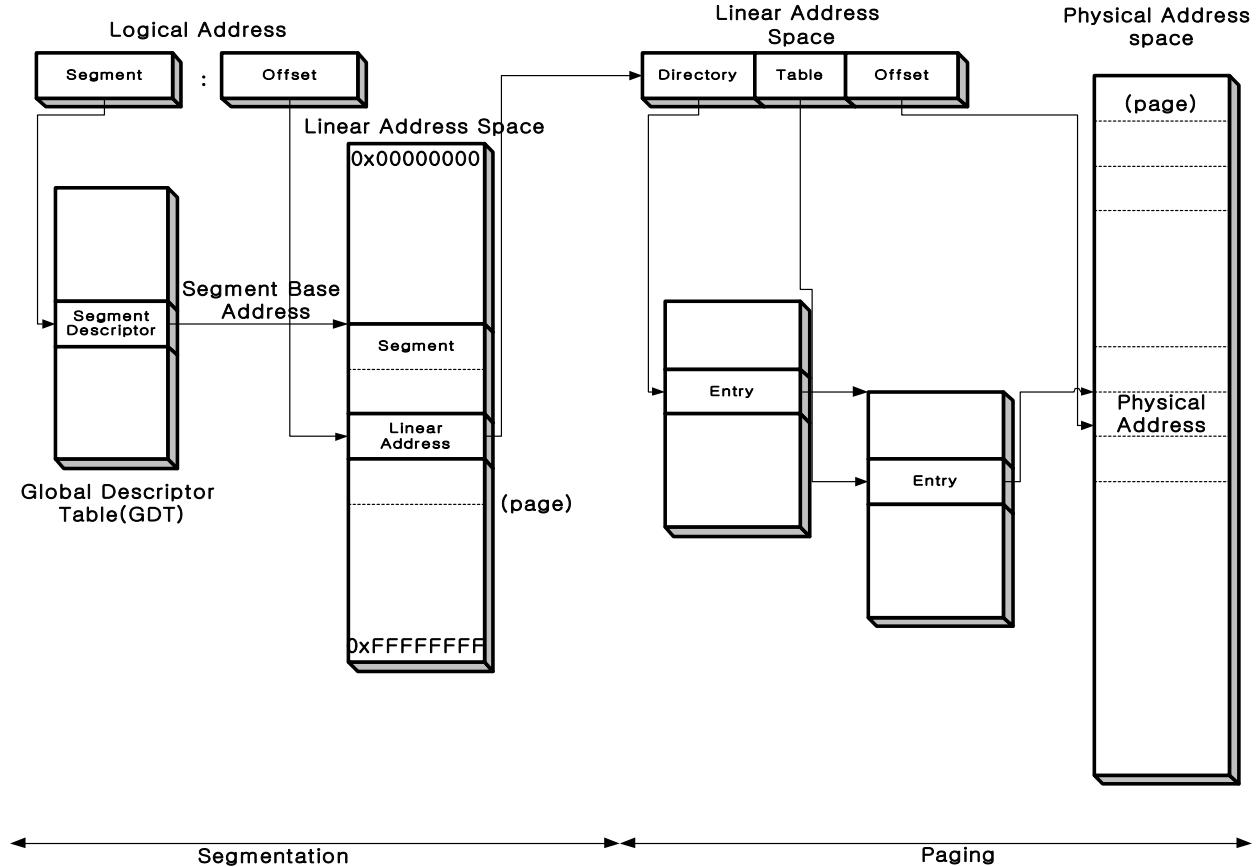


그림 33. Intel x86 시스템에서의 주소 변환

[그림27]에서 논리적인 주소는 세그먼트와 옵셋으로 이루어지며, 옵셋 부분은 세그먼트 디스크립터를 가지는 테이블에 대한 인덱스 역할을 한다. 이곳에서 찾은 주소와 다시 옵셋을 결합해서 하나의 선형주소를 만들고, 다시 이것은 페이징을 위해서 사용된다. 이전 과정까지가 세그먼테이션을 이용한 메모리 관리방법이며, 이후의 과정이 페이징 방법이다. 선형주소로의 변환이 끝나면, 이 주소는 페이지 디렉토리를 가지는 부분과 페이지의 테이블, 그리고, 다시 옵셋의 세부분으로 나누어진다. 먼저 디렉토리에서 해당부분의 페이지 테이블에 대한 entry를 찾고, 테이블내의 entry는 선형주소의 페이지 테이블 부분에서 선택한다. 이렇게 만들어진 주소는 물리적인 메모리내의 한 페이지를 가르키게 되며, 다시 이곳에서 옵셋부분을 더해서 하나의 물리적인 주소가 만들어진다.

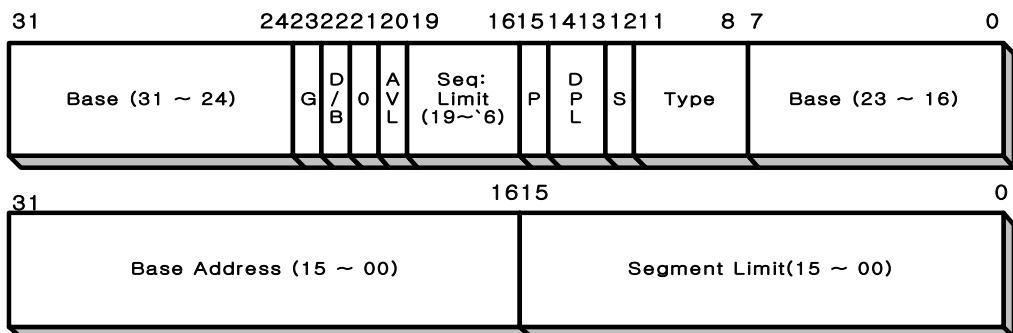
Intel에서 이와같은 2단계에 걸친 세그먼테이션과 페이징을 사용하는 이유는 논리적으로 연관이 있는 부분들로 프로그램을 나누어서 작성할 있도록 하기 위한 것인데, 세그먼테이션을 사용하면 각각의 프로세스에 다른 선형주소 공간을 할당할 수 있다. 반면에 페이징은 같은 선형 주소 공간을 각각 다른 물리적인 공간으로 맵핑을 할 수 있다.

일반적으로 페이지의 크기는 4 Kbytes를 가진다고 보면된다. 만약 페이지의 크기가 지나치게 커지면, 페이지내의 사용되지 않는 공간이 늘어날 가능성이 많으며, 지나치게 작아지면, 관리하는데 드는 커널내의 테이블이 커지게 된다. 따라서, 적당한 크기의 페이지 크기를 가지는 것이 중요한다. 또한 페이지 크기는 한번의 블록 디바이스에 대한 접근에서 얻을 수 있는 데이터 크기의 배가 되도록 해주는 것이 블록 디바이스에 대한 접근을 줄이며, 쓸모없이 버려지는 공간을 줄이는데 효과적이다. 대체로 블록 디바이스에 대한 접근은 512bytes 단위이기에 4 Kbytes는 여덟배가 된다.

Intel의 경우에 사용되는 세그먼테이션을 위한 레지스터에는 CS 레지스터, SS 레지스터, DS레지스터, FS 레지스터, ES 레지스터, GS레지스터가 있다. 이중에서 CS 레지스터는 프로그램의 코드 영역을 가리키는 세그먼트 레지스터이며, SS는 스택을 DS는 데이터 영역을 가리킨다. FS와 ES, GS는 보조적으로 사용되는

세그먼트 레지스터이다⁵⁹. 이중에서 CS레지스터는 다시 현재 실행되는 레벨을 표기하는 목적으로 사용하는데, 크게 실행의 레벨에는 리눅스에서 커널 레벨과 사용자 레벨⁶⁰가 있다. 각각에 대해서 0과 3값을 가지며, CS 레지스터의 하위 두 bit로 나타낸다.

GDT(Global Descriptor Table)은 대체로 시스템 내에 하나가 정의되어 사용되며, 각각의 프로세스는 또한 자신만의 LDT(Local Descriptor Table)을 가진다. 각각은 현재 사용되는 세그먼트 레지스터에 대한 설명/기술(description)을 프로세스에 제공하기 위해서 존재하며, 제공하는 정보는 세그먼트의 위치와 크기 및 상태와 제어가 있다. 리눅스의 경우 GDT를 부팅하는 과정에서 초기 설정을 하게되며, 설정은 GDTR 레지스터에 GDT의 주소를 넣어주는 *lgdt* 명령을 이용한다. LDT의 경우는 LDTR 레지스터에 *lldt* 명령을 이용한다. GDT에 들어가는 내용 세그먼트의 디스크립터이며, [그림28]와 같다.



AVL – 시스템 소프트웨어에 사용가능함

BASE – 세그먼트의 기본(base) 주소

D/B – 기본(default) 연산의 크기(0 for 16-bit segment, 1 for 32 bit segment)

DPL – Descriptor Privilege Level

G – Granularity

LIMIT – 세그먼트 한계(limit)

P – 세그먼트 존재(present)

S – 디스크립터의 타입(0 for system, 1 for code or data)

TYPE – 세그먼트 타입

그림 34. 세그먼트 디스크립터

각각의 디스크립터들은 8bytes의 크기를 가지며, 각각의 필드들에 대한 내용은 아래와 같다.

- AVL(Available and reserved bits) : 세그먼트 디스크립터의 두번째 double word(4bytes)의 20 bit를 차지하며, 시스템 소프트웨어가 사용할 수 있다. 21bits는 항상 0값을 가진다.
- BASE(Base address fields) : 4 Gbytes의 선형 주소공간 영역에서 세그먼트의 시작번지(0 byte)의 위치를 정의한다. 프로세서는 3개의 base주소를 합쳐서, 하나의 32bit값으로 만든다. 세그먼트 주소는 반드시 16byte경계로 맞춰져야 하기에 base주소도 이 경계를 맞춘다. 이렇게 정렬해 주는 것이 시스템의 성능을 높여준다.
- D/B(Default Operation size/Default stack pointer size and/or upper bound)플랙 : 이 플랙은 기본적인 연산의 크기를 나타내는것으로 정의하고 있는 세그먼트가 코드인지, 스택인지, 아니면 데이터 세그먼트인지에 따라서 달리 동작한다. 기본적으로 1일때는 32 bit 코드와 데이터 세그먼트를, 0일때는 16 bit 코드와 데이터 세그먼트로 설정된다.
- DPL(Descriptor Privilege Level) : 세그먼트의 접근 권한(privilege)을 나타낸다. 0에서 3의 값을 가질 수 있으나, 리눅스나 윈도우의 경우 0과 3을 커널 모드와 사용자 모드로 정의해서 사용하고 있다.

⁵⁹ CS(Code Segment)레지스터는 프로그램의 TEXT영역을 가지는 부분이며, DS(Data Segment)는 프로그램의 BSS(Block Started by Symbol)영역과 Heap영역을, SS(Stack Segment)는 스택영역을 가지고 있으며, 하위 방향으로 스택을 사용한다.

⁶⁰ 커널 모드와 사용자 모드.

- G(Granularity) 플랙 : 세그먼트 한계(limit) 필드에 대한 스케일링(scaling)을 결정한다. 0이라고 한다면, 세그먼트의 한계는 byte단위로 해석되며, 1로 되어 있다면 4 Kbytes단위로 해석된다. 이것은 나중에 offset부분을 세그먼트 한계와 비교하는 것에 영향을 미친다.
- LIMIT(Segment Limit Field) : 세그먼트의 크기를 명시한다. 두개의 세그먼트 한계(limit)필드를 합쳐서 20bit의 값을 프로세서가 생성하며, G 플랙의 설정에 따라서 두가지의 방식으로 동작한다. 첫번째가 G가 설정된 경우로서 세그먼트 크기는 4 Kbytes에서 4GBytes로 4 KBytes씩 증가하는 범위를 가질 수 있으며, 두번째가 G가 설정되지 않는 경우로 세그먼트 크기는 1 byte에서 1 Mbytes까지의 범위를 가질 수 있다.
- P(Segment Present) 플랙 : 세그먼트가 메모리에 존재하는지를 나타낸다. 설정된 경우에 세그먼트가 메모리에 존재하며, 설정되지 않았다면 메모리에 존재하지 않는다. 이 경우 프로세서는 그 세그먼트에 접근하려는 시도에 대해 “Segment Not Present” exception이 발생한다. 이것을 처리하는 것은 메모리를 관리하는 소프트웨어의 몫이다.
- S(Descriptor Type) 플랙 : 세그먼트가 시스템 세그먼트인지(0일 경우), 아니면 코드나 데이터를 나타내는 세그먼트 인지(1일 경우)를 표시한다.
- TYPE : 세그먼트나 gate의 type을 나타내며, 세그먼트에 가능한 접근 방법들 및 나아가는 (growth)방향을 명시한다. 이 필드의 해석은 디스크립터가 코드나 데이터를 위한 것인지, 아니면 시스템 디스크립터를 위한 것인지에 따라서 달라진다.

크게 시스템에는 3종류의 세그먼트 디스크립터가 존재하는데, 이것에는 데이터 세그먼트 디스크립터와 코드 세그먼트 디스크립터, 시스템 세그먼트 디스크립터가 있다.

아래에 보이는 것은 리눅스에서 부팅시에 설정하는 디스크립터 테이블의 정의다. 커널의 head.S를 참고하자.

```
...
ENTRY(gdt_table)
    .quad 0x0000000000000000 /* NULL descriptor */
    .quad 0x0000000000000000 /* not used */
    .quad 0x00cf9a00000ffff /* 0x10 kernel 4GB code at 0x00000000 */
    .quad 0x00cf9200000ffff /* 0x18 kernel 4GB data at 0x00000000 */
    .quad 0x00cffa000000ffff /* 0x23 user 4GB code at 0x00000000 */
    .quad 0x00cff2000000ffff /* 0x2b user 4GB data at 0x00000000 */
    .quad 0x0000000000000000 /* not used */
    .quad 0x0000000000000000 /* not used */

...
/* APM( Advanced Power Management) 세그먼트 디스크립터를 위한 부분이 아래에 연속된다.*/

```

코드 247. 리눅스의 GDT 정의

“.quad”는 8byte를 가지기에 뒤부분에 정의된 값이 하나의 디스크립터 테이블을 나타낸다. 제일 먼저 사용되지 않는 NULL 디스크립터⁶¹가 오며, 그 후에 커널에 사용하는 코드와 데이터를 위한 세그먼트 디스크립터, 그리고, 사용자 주소 공간에 대한 코드와 데이터 세그먼트 디스크립터가 나온다. 뒷쪽에 두 개의 디스크립터 영역도 사용하지 않는다.

커널과 사용자의 디스크립터를 보면 차이가 나는 부분은 9와 F이다. 커널 영역의 9는 “1001b”로 나타내며, 해당하는 디스크립터의 영역은 P, DPL, S부분이다. 즉, 세그먼트가 메모리에 존재하며, privilege level이 0이고, 시스템 코드나 데이터를 가진다고 보는 것이다. 사용자 영역의 해당부분은 F0이므로 “1111”로 나타내지며, 메모리에 존재하며, privilege level로 3을, 코드나 세그먼트를 가진다고 표시하고 있다. 커널이나 사용자 모두 G, D/B, 0, ALV필드는 같이 C(=1100b)로 설정하고 있다. 즉, Granularity가 4Kbytes단위임을, 32bit 세그먼트를 사용한다는 것을 나타낸다. 한계(limit)부분에 대한 설정은 모두

⁶¹ NULL 세그먼트 selector라고 합니다. 만약 이것을 가지고 있는 세그먼트 selector로 메모리를 접근하려고 하면, 예외상황(exception)이 발생한다. 이것은 사용되지 않는 세그먼트 레지스터를 초기화하는데 사용될 수 있다. CS나 SS를 이것으로 적재하려고 하면 general-protection exception이 발생한다.

0xFFFFF로 나타내서, 세그먼트 크기의 범위가 4 Kbytes에서 4Gbytes임을 나타낸다($G=1$ 이므로). 데이터와 코드 세그먼트 디스크립터는 각각 다시 A와 2라는 값의 차이가 있다. 이 부분은 TYPE에 해당하는 부분으로 각각 0x0A와 0x02라는 값을 정의되었다. 코드 세그먼트 디스크립터의 경우에는 “1XXXb”형식으로 TYPE이 나타내지며, 0x0A(=1010b)는 실행과 읽기(read)가 가능하다는 것을 표시한다. 데이터 세그먼트 디스크립터의 TYPE은 “0XXXb”형식으로 나타내지며, “0010b”는 읽기(read)와 쓰기(write)가 가능하다는 것을 나타낸다.⁶²

자, 이제는 조금더 자세히 세그먼테이션에 대한 이야기를 하도록 하자. 세그먼트 선택자(selector)는 16bit으로 이루어진 세그먼트 레지스터를 말하며, 다시 옵셋은 32bit으로 나타내진다. Intel x86 CPU에서는 세그먼트 selector가 선택 될 때마다, 해당하는 세그먼트 디스크립터가 CPU의 non-programmable 레지스터로 적재(load)되며, 이것을 이용해서 항상 세그먼트 디스크립터를 보지 않고도 빠르게 주소 연산을 실행할 수 있게 된다. 즉, 메모리를 보지 않고, 레지스터끼리의 연산만으로 한정되기 때문이다. 각각의 세그먼트 selector는 직접적으로 세그먼트를 가르키지 않고, 세그먼트를 정의하는 세그먼트 디스크립터를 가르킨다. 세그먼트 selector는 다음과 같은 필드들을 가진다.

- Index (bit3 ~ 15) : GDT나 IDT에 있는 8192개의 디스크립터 중에 하나를 가리키는 값이다. 프로세서는 이 값에 8을 곱해서 GDT(GDTR)나 LDT(LDTR)의 base에 더해서 해당 세그먼트를 찾는다.
- TI(Table Indicator : bit 2) 플랙 : 사용할 디스크립터 테이블을 명시한다. 0인 경우에는 GDT를, 1인 경우에는 LDT를 사용한다.
- RPL(Request Privilege Level: bit 0 ~ 1) : selector의 privilege 레벨을 명시한다. 0에서 3까지의 값을 가질 수 있으며, 0이 가장 높은 특권레벨(privilege level)을 나타내며, 3이 가장 낮다.

따라서, 세그먼트 디스크립터 테이블을 보기 위해서는 TI bit부분을 살펴서 GDTR과 LDTR중에서 어느것을 선택할 것인지를 결정하게되고, 이에 따라서 보게 되는 디스크립터를 이용해서 옵셋과의 합으로 선형주소(linear address)를 만들어낸다. [그림29]는 이 과정을 보여준다.

⁶² 현재 설명하고 있는 내용은 Intel의 System Programmer를 위한 manual에서 가져온 내용이다.

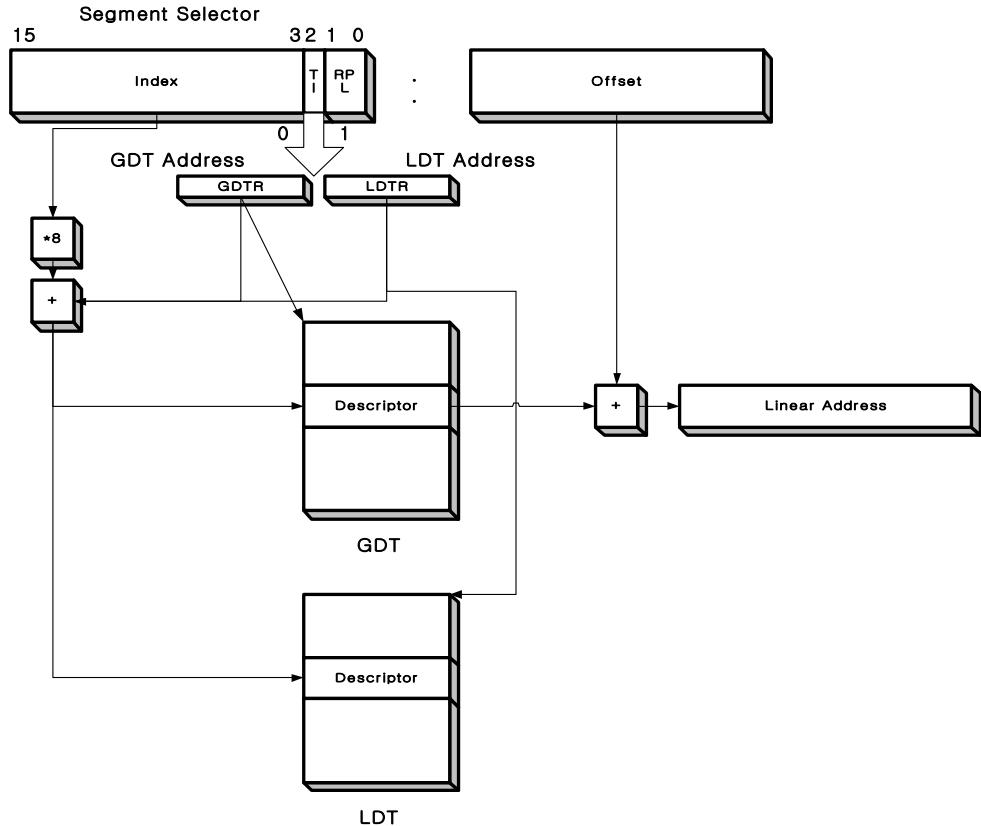


그림 35. 세그먼트 selector를 이용한 선형 주소의 생성

리눅스에서 사용되는 세그먼트에는 위에서 언급한 커널과 사용자를 위한 코드 및 데이터 세그먼트 이외에, TSS(Task State Segment)와 default LDT 세그먼트가 존재한다. 커널과 상요자의 코드 및 데이터 세그먼트를 가리키는 값은 `_KERNEL_CS`, `_KERNEL_DS`, `_USER_CS`, `_USER_DS`⁶³가 있다.

```
#define __KERNEL_CS      0x10
#define __KERNEL_DS     0x18

#define __USER_CS       0x23
#define __USER_DS      0x2B
```

코드 248. 커널과 사용자 세그먼트의 값

TSS 세그먼트의 디스크립터는 각각의 프로세스마다 정의되며, GDT에 저장된다. 세그먼트 디스크립터의 Base 필드는 각 CPU마다 정의된 `tss_struct` 구조체 부분의 주소를 가지며⁶⁴, G는 0으로 Limit는 0xEB로 설정되어 236 bytes의 길이를 가지도록 한다. TYPE 필드는 9나 11로 설정되며, DPL은 0이 설정되어 커널 모드에서만 접근할 수 있도록 한다.

Default LDT 세그먼트의 디스크립터는 모든 프로세스에 의해서 공유되며, `~/arch/i386/kernel/traps.c`에 `default_ldt` 변수로 정의되며, 아래와 같다.

```
/* ~/include/asm/desc.h에서 */
struct desc_struct {
    unsigned long a,b;
};
```

⁶³ `~/include/asm/segment.h`를 참조하라.

⁶⁴ 커널 버전 2.4.0 이전에는 `task_struct` 구조체에 `tss` 필드가 나오지만 버전 2.4.0은 다르다.

```
/* ~/arch/i386/kernel/traps.c에서 */
struct desc_struct default_ldt[] = { { 0, 0 }, { 0, 0 }, { 0, 0 }, { 0, 0 }, { 0, 0 } };
```

코드 249. default_ldt의 정의

Default LDT 세그먼트는 단순히 NULL 세그먼트 디스크립터로 구성되어 있으며, 각각의 프로세스가 자신만의 LDT 세그먼트 디스크립터를 가지고 이곳을 가르키도록 하고 있다.

앞에서 우리는 주로 사용되는 6개의 세그먼트 디스크립터에 대한 설명을 했는데, 이것 이외에도 ~/arch/i386/kernel/head.S에는 4개의 APM(Advanced Power Management)를 위한 세그먼트 디스크립터가 있다. 아래와 같이 정의된다.

```
.quad 0x0040920000000000 /* 0x40 APM set up for bad BIOS's */
.quad 0x00409a0000000000 /* 0x48 APM CS code */
.quad 0x00009a0000000000 /* 0x50 APM CS 16 code (16 bit) */
.quad 0x0040920000000000 /* 0x58 APM DS data */
.fill NR_CPUS*4,8,0      /* space for TSS's and LDT's */
```

코드 250. APM 및 TSS와 LDT를 위한 세그먼트 디스크립터 공간

이곳을 보면 NR_CPUS⁶⁵의 각각에 4개씩 8byte를 0으로 채우는 디스크립터 테이블이 존재한다. 이것은 각각의 CPU에 대한 TSS공간과 프로세스들의 LDT를 위한 공간으로 사용한다. 나머지는 APM의 수행을 위한 코드와 데이터 세그먼트 디스크립터이다.

프로세스가 생성되면, TSS와 LDT 디스크립터는 GDT에 더해지며, TSS를 GDT에 삽입하는 것은 ~/arch/i386/kernel/traps.c의 set_tss_desc() 함수와 set_ldt_desc() 함수가 담당한다. 아래와 같다.

```
/* ~/include/asm/desc.h에서 */
#define __FIRST_TSS_ENTRY 12
#define __FIRST_LDT_ENTRY (__FIRST_TSS_ENTRY+1)

#define __TSS(n) (((n)<<2) + __FIRST_TSS_ENTRY)
#define __LDT(n) (((n)<<2) + __FIRST_LDT_ENTRY)

/* 0이 하는 ~/arch/i386/kernel/trap.c에서 가져왔다.*/
#define _set_tssldt_desc(n,addr,limit,type) \
__asm__ __volatile__ ("movw %w3,0(%2)\n\t" \
"movw %%ax,2(%2)\n\t" \
"rorl $16,%%eax\n\t" \
"movb %%al,4(%2)\n\t" \
"movb %4,5(%2)\n\t" \
"movb $0,6(%2)\n\t" \
"movb %%ah,7(%2)\n\t" \
"rorl $16,%%eax" \
: "=m"(*(n)) : "a" (addr), "r"(n), "ir"(limit), "i"(type))

void set_tss_desc(unsigned int n, void *addr)
{
    _set_tssldt_desc(gdt_table+__TSS(n), (int)addr, 235, 0x89);
}

void set_ldt_desc(unsigned int n, void *addr, unsigned int size)
{
    _set_tssldt_desc(gdt_table+__LDT(n), (int)addr, ((size << 3)-1), 0x82);
```

⁶⁵ 현재 시스템이 가지고 있는 CPU의 갯수

{

코드 251. TSS와 LDT 세그먼트 디스크립터를 GDT에 추가하는 함수

즉, 현재 프로세스 번호의 4를 곱한 값과 사용가능한 첫번째 GDT 엔트리(entry)를 더해서 GDT 테이블의 엔트리(entry)를 찾고, 주소와 Limit, TYPE을 설정하는 일을 한다.

첫번째 TSS에 대한 초기화는 trap_init()에서 수행되며, 이곳에서 cpu_init()를 호출한다. cpu_init()는 ~/arch/i386/kernel/setup.c에 정의되어 있으며, 이곳에서 하는 일은 각각의 CPU 상태를 초기화 하는 일을 담당한다. 이곳에서 생성되는 것이 TSS는 프로세스 0번으로 init 태스크이다.

참고적으로 실제적인 세그먼트 레지스터⁶⁶의 구조는 크게 두 부분으로 나누어진다. 보이는 부분(visible part)과 감춰진 부분(hidden part)⁶⁷ 가 그것이다. 이중에서 보이는 부분만이 세그먼트 selector를 가지며, 나머지는 base 주소와 한계(limit), 접근 정보만을 가진다.

세그먼트를 사용하는 방법에는 크게 세가지 모델이 있으며, 각각 기본(basic) flat 모델, 보호(protected) flat 모델, 멀티 세그먼트 모델이 있다. 아래와 같이 정의된다.

1. 기본(basic) flat 모델 : 이 모델은 가장 간단한 방법으로 운영체제나 응용 프로그램 모두가 세그먼트화 되지 않고, 연속적인 주소 공간을 접근하는 방법이다. 이 모델을 구현하기 위해선 Intel에서는 최소 두개의 디스크립터가 있어야 하며, 하나는 코드를 하나는 데이터 세그먼트를 위한 것이다. 모든 세그먼트 레지스터는 기본 주소로 0을 가지며, 4 GBytes의 세그먼트 한계를 가진다.⁶⁸
2. 보호(protected) flat 모델 : 이 모델은 기본 flat 모델과 유사하나 실제적인 메모리 공간(physical address space)을 포함하기 위해서 세그먼트의 한계가 설정된다는 점이 다르다. 이 경우에는 앞에서와는 다르게 실제로 존재하지 않는 메모리에 대한 접근은 general-protection exception을 발생 시킨다. 따라서, 최소한의 하드웨어 보호 기능을 제공한다. 이 경우에는 페이지가 사용자와 커널 코드를 분리하기 위해서 4개의 세그먼트를 필요로하게 되며, 코드와 데이터 세그먼트의 각각에 하나씩 해당한다. 이 모델은 효과적으로 응용 프로그램과 운영체제를 보호하며, 또한 각각의 프로세스에 분리된 페이지 구조를 제공해서 프로세스간에도 보호기능을 제공하기 때문에, 많은 운영체제가 이 모델을 사용하고 있다.
3. 멀티 세그먼트 모델 : 이 모델은 모든 세그먼트의 기능을 다 쓰는 것으로 하드웨어에서 제공하는 모든 보호기능을 코드와 데이터, 그리고, 프로그램들과 프로세스들에 대해서 적용하는 것이다. 모든 세그먼트와, 시스템에서 실행중인 개개의 프로그램의 실행 환경에 대한 접근은 하드웨어에 의해서 제어된다.

이중에서 리눅스에서 사용하는 모델은 보호 flat 모델이다. 따라서, 코드 세그먼트를 나타내는 디스크립터와 나머지 세그먼트들을 공통적으로 나타내는 디스크립터를 가지고 있으며, 접근 제어와 한계(limit)가 실제 물리적인 주소 공간으로 한정된다. 앞에서 본 커널은 위한 코드와 데이터 세그먼트 디스크립터와 사용자를 위한 코드와 데이터 세그먼트의 정의를 이곳에서 유추할 수 있을 것이다. 더 이야기를 진행하기 전에, 위에서 잠시 소개된 특권 레벨(privilege level)에 대해서 좀더 정리하고 넘어가도록 하자.

4.2. Privilege Level

프로세서(Processor)의 세그먼트 보호(Segment Protection) 메커니즘⁶⁹은 4개의 특권 레벨(privilege level)을 인식하며, 각각은 0에서 3까지의 값으로 표현된다. [그림30]은 이러한 특권 레벨이 보호 링(Protection Ring⁷⁰)과 같이 해석될 수 있다는 것을 보여준다.

⁶⁶ CS, SS, DS, ES, FS, GS 세그먼트 레지스터를 말한다.

⁶⁷ Hidden Part는 때로 Descriptor Cache나 Shadow Register라고도 표현된다.

⁶⁸ 한계를 벗어난 메모리에 대한 접근도 GPE(General Protection Exception)을 발생시키지 않는다.

⁶⁹ Intel의 Intel Architecture Software Developer's Manual Vol3. System Programming을 참고하기 바란다.

⁷⁰ 각각의 특권 레벨이 중심에서 링(Ring)을 겹쳐놓은 형태로 설명되어, 특권 레벨의 링(Ring)이라는 말로 표현하는 것 같다.

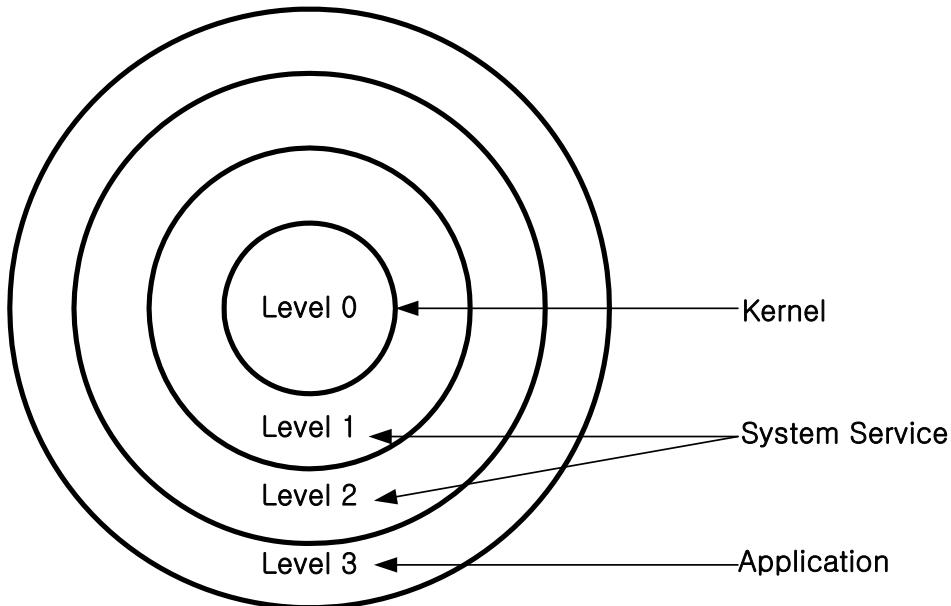


그림 36. 특권 레벨의 링

가장 중심에 있는 부분은 가장 특권 레벨이 높은 코드와 데이터 및 스택에 예약된다. 이것은 중요한(critical) 소프트웨어를 포함하는 세그먼트를 위해서 사용된다고 보면 될 것이다. 일반적으로는 운영체제의 커널 부분이 이곳에 해당한다. 나머지 링들은 단계별로 중요도가 낮은 소프트웨어들에 사용된다. 만약 4개의 링에서 2가지 만을 사용한다면, 0과 3이 될 것이다.

프로세스는 특권 레벨을 통제되고 있는 상황을 제외한, 낮은 특권 레벨에서 동작중인 프로그램이나 태스크(task)가 더 높은 특권 레벨을 가지는 세그먼트를 접근하는 것을 막을 목적으로 한다. 만약 프로세스가 특권 레벨을 깨는 행동을 감지하면, general-protection exception을 발생시킬 것이다. 이 오류는 운영체제에 보고되며, exception을 발생시킨 프로세스는 동작을 멈출 것이다.

코드 세그먼트와 데이터 세그먼트 사이에서 특권 레벨을 검사하기 위해서는 프로세스는 다음과 같은 세가지 종류(type)의 특권 레벨을 인식할 수 있어야 할 것이다.

- **Current Privilege Level(CPL)** : CPL은 현재 프로그램이나 태스크의 특권 레벨을 나타내는 값이다. CS나 SS 세그먼트 레지스터의 0과 1 bit에 저장되어, 0에서 3까지의 값으로 나타내진다. 일반적으로 CPL은 명령(instruction)을 가져오는 코드 세그먼트의 특권 레벨과 동일하다. 프로세스는 프로그램의 제어가 다른 특권 레벨을 가지는 코드 세그먼트로 옮겨질 때 CPL을 바꾸게 되며, conforming 코드 세그먼트⁷¹들에 대한 접근을 하게 될 때는 조금 다르게 취급된다. Conforming 코드 세그먼트들은 conforming 코드 세그먼트의 DPL보다 같거나 더 낮은 특권 레벨을 가진 어떠한 특권 레벨에서도 접근이 가능하다. 또한 CPL은 프로세서가 CPL과 다른 특권 레벨을 가지는 conforming 코드 세그먼트를 접근할 때 바뀌지 않는다.
- **Descriptor Privilege Level(DPL)** : DPL은 세그먼트와 gate의 특권 레벨로서, 세그먼트나 gate를 위한 세그먼트나 gate 디스크립터에 저장된다. 현재 실행중인 코드 세그먼트가 세그먼트나 gate를 접근하려는 시도를 하면, 세그먼트나 gate의 DPL이 세그먼트나 gate selector의 CPL 및

⁷¹ Conforming 코드 세그먼트란 세그먼트 디스크립터의 type 필드에 C(Conforming) bit이 설정된 코드 세그먼트를 말한다. 즉, 이 bit이 설정된 경우에는 더 높은 특권 레벨을 가진 세그먼트로의 실행 전환(transfer)가 현재의 특권 레벨을 가지고 계속 실행을 진행하는 것을 허락한다. 만약 C bit이 설정되지 않은 다른 특권 레벨을 가지는 non-conforming 세그먼트로 제어를 옮기려고 한다면, general-protection exception(#GP)를 발생시킬 것이다. 하지만, 이것도 call gate나 task gate를 통해서 #GP를 발생시키지 않고 진행하는 것이 가능하다.

RPL과 비교된다. DPL은 접근되는 세그먼트나 gate의 type에 따라서 달리 해석되며, 아래와 같이 요약할 수 있다.

- **Data Segment** : 프로그램이나 태스크가 세그먼트를 접근하는 것을 허락 받기 위해서 가질 수 있는 숫자적으로 가장 높은 특권 레벨을 가리킨다. 예를 들어서, 데이터 세그먼트의 DPL이 1이라면, CPL이 0이나 1인 값을 가지는 프로그램이나 태스크만 그 세그먼트를 접근할 수 있다.
- **Nonconforming Code Segment(without using a call gate)** : DPL은 프로그램이나 태스크가 세그먼트를 접근하기 위해서 가져야 하는 특권 레벨을 나타낸다. 예를 들어서, 만약 nonconforming 코드 세그먼트의 DPL이 0이라면, 오직 CPL이 0인 프로그램이나 태스크만이 그 세그먼트를 접근 할 수 있다.
- **Call Gate** : 여기에 있는 DPL은 현재 실행중인 프로그램이나 태스크가 특권 레벨로 가질 수 있으며, 계속적으로 call gate를 접근할 수 있는, 숫자적으로 가장 높은 특권 레벨을 나타낸다. 이것은 데이터 세그먼트와 같은 접근 규칙을 가진다.
- **Conforming Code Segment and Nonconforming Code Segment accessed through a call gate** : DPL은 프로그램이나 태스크가 세그먼트를 접근하기 위해서 가져야 하는 숫자적으로 가장 낮은 특권 레벨을 나타낸다. 예를 들어서, 만약 conforming 코드 세그먼트의 DPL이 2이고, 현재 실행중인 프로그램이나 태스크의 CPL이 0이나 1이라면 그 세그먼트를 접근할 수 없다.
- **TSS** : 이곳에 있는 DPL은 현재 실행중인 프로그램이나 태스크가 여전히 TSS를 접근하면서, 있을 수 있는 숫자적으로 가장 높은 특권 레벨을 나타낸다. 이것은 데이터 세그먼트에 적용되는 접근 방법과 같다.
- **Requested Privilege Level(RPL)** : RPL은 세그먼트 selector에 할당된 특권 레벨을 덮어쓰는(override) 값으로, 세그먼트 selector의 0과 1 bit에 들어간다. 프로세서는 CPL과 함께 RPL을 어떤 세그먼트에 대한 접근이 허락되는지를 결정하기 위해서 검사한다. 심지어 세그먼트를 접근하는 프로그램이나 태스크가, 세그먼트를 접근하기 위해서 충분한 특권을 가지고 있다고 하더라도, RPL이 충분한 특권 레벨을 가지지 못한다면, 접근은 거부(deny)될 것이다. 즉, 세그먼트 selector의 RPL이 숫자적으로 CPL보다 크다면, RPL은 CPL을 덮어쓰게 된다. CPL이 RPL보다 더 큰 값을 가지더라도 마찬가지로 RPL을 덮어쓰게 될 것이다. RPL은 프로그램이나 태스크 자체가 세그먼트에 대해서 접근 권한(privilege)을 가지지 않다면, 특권화된 코드(privilege code)가 응용프로그램을 위해서 세그먼트를 접근하지 못하도록 보장해 주는데 사용될 수 있다.

특권 레벨은 세그먼트 디스크립터의 세그먼트 selector가 세그먼트 레지스터로 적재(load)될 때 검사되며, 데이터를 접근하기 위해서 사용되는 검사는 코드 세그먼트 들간에 프로그램의 제어를 옮기기 위해서 사용되는 검사와는 다르다. 따라서, 두 가지의 검사는 각각 달리 처리되어야 할 것이다.

4.3. Paging

페이지 시스템이 하는 일은 이전의 [그림27]에서 보았듯이 선형 주소 공간을 물리적인 주소 공간으로 맵핑하는 것이다. 이것은 또한 선형 주소의 접근 타입에 대해서 요청된 접근 방식을 검사하는 부분이며, 만약 유효하지 않은 접근일 경우에는 페이지 fault exception이 발생한다. 이러한 exception이 발생할 때는 해당하는 핸들러로 제어를 옮기게 되며, 만약 현재 메모리에 있지는 않지만 유효한 주소에 대한 접근일 경우에는 그 페이지를 디스크로 부터 읽어서 메모리 영역을 채우고, 다시 페이지에 대한 기술 사항을 바꾼 후, 프로세스의 진행을 계속 할 수 있도록 만들어준다. 프로세스는 자신이 물리적인 메모리 상에 있는 페이지에 대해서 접근을 했는지, 아니면 있지 않은 페이지에 대한 접근을 했는지를 알 수 없다. 페이지 시스템에서 사용하는 어휘들에 대해서 보도록 하자. 가장 먼저 페이지(page)가 될 것이다. 페이지란 정해진 길이⁷²를 가지는 메모리 공간을 말한다. 페이지내에서의 선형주소는 물리적으로 연속적인 주소를 사용하게 되며, 커널은 페이지 단위로만 물리적인 주소와 접근 방식으로 표시한다. 또한 페이지는 물리적으로 연속된 공간을 의미하기도 하며, 그 속의 데이터도 페이지라고 표현하기도 한다. 따라서, 여기선 같은 의미로 사용하도록 하겠다.

⁷² 대략 4Kbytes에 해당한다.

페이지를 사용하기 위해서는 물리적인 메모리인 RAM을 일정한 크기로 나눈다. 이것을 **페이지 프레임(page frame)**이라고 하며, 하나의 페이지를 가진다고 본다. 즉, 4Kbytes가 될 것이다. 따라서, 만약 4Mbytes를 가지는 메모리를 시스템이 가지고 있다면, 1K개의 페이지 프레임을 가지게 된다. 페이지는 디스크 상에 있을 수도 있으며, 메인 메모리의 페이지 프레임을 차지할 수도 있다.

다시 페이지 프레임들에 대한 관리의 목적으로 페이지 테이블(page table)이 있다. 이것은 페이지 프레임들에 대한 목록을 가지는 메인 메모리 상의 데이터 구조이다. 따라서, 커널이 관리하는 영역이다. 선형주소로 만들어진 주소를 이곳에서 테이블에 대한 인덱스로 사용하며, 물리적인 메모리의 한 페이지 프레임을 가르킨다.

페이지 테이블에 대한 목록을 가지는 것이 페이지 딕렉토리이다. 즉, 페이지 테이블의 하나의 엔트리(entry)는 페이지 테이블을 가르키는 포인터가 된다. 이것은 페이지에서 선형주소의 물리적인 주소 맵핑에서 가장 첫번째로 처리되는 부분이며, 하나의 엔트리로 나타낼 수 있는 물리적인 메모리의 양은 페이지 테이블 엔트리의 갯수에 페이지 프레임의 크기를 곱한 값이 될 것이다. 따라서, 만약 10bit로 table을 표현한다면, 페이지 크기가 4Kbytes를 가질 때, 4Mbytes를 하나의 페이지 테이블 엔트리가 처리할 수 있다. 그리고, 페이지 테이블 역시 커널에서 관리하는 영역이기에 항상 메인 메모리에 존재하게 된다.

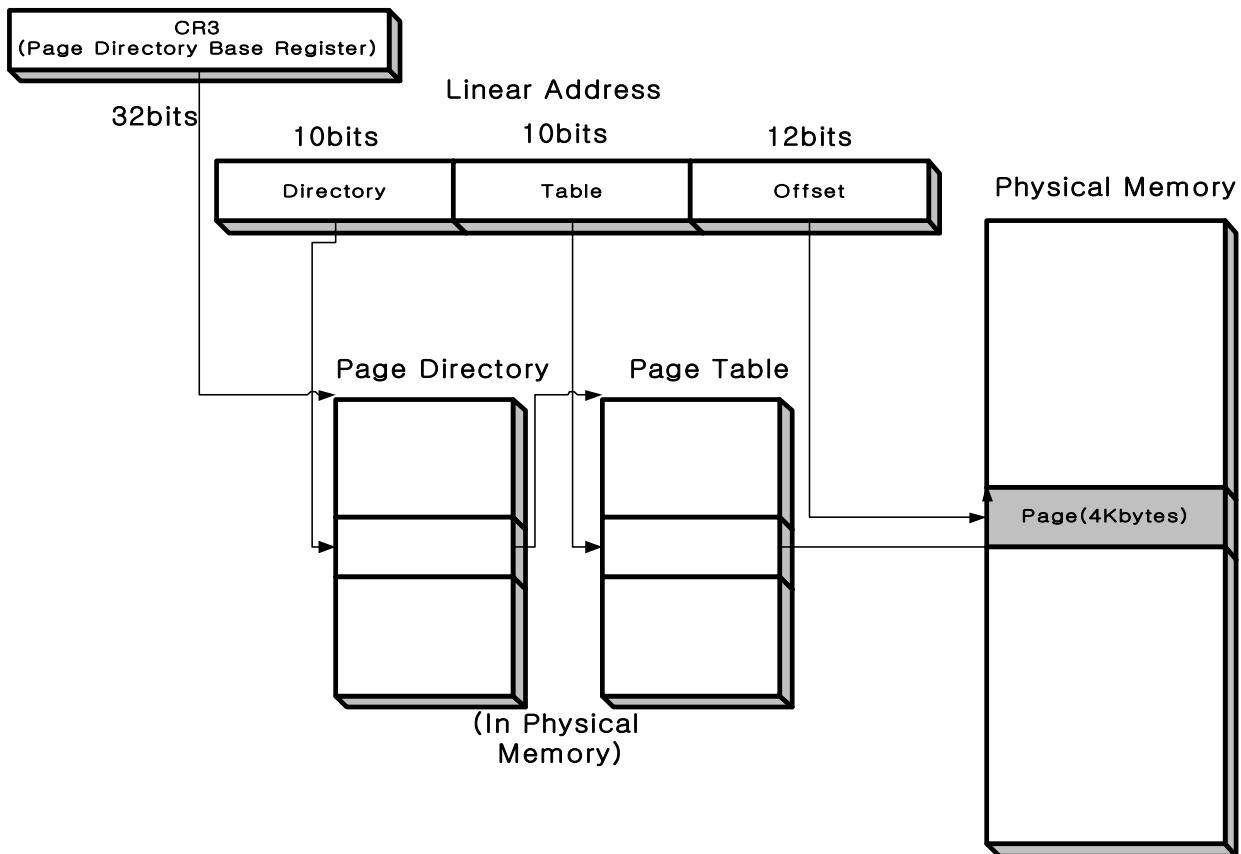


그림 37. Intel CPU의 페이지

전체적인 페이지는 [그림30]과 같이 나타낼 수 있다. 인텔의 경우에는 CR0 제어 레지스터의 PG 플랙을 설정함으로써, 페이지 기능을 사용할 수 있다. 따라서, 부팅시에 CR0의 PG플랙을 설정한다. 만약 PG 플랙이 설정되지 않았다면, 세그먼 테이션에서 생성된 선형주소가 곧바로 물리적인 주소를 가질 것이다.

또한 사용중인 페이지 딕렉토리의 물리적인 주소는 CR3 제어 레지스터에 보관된다. 따라서, 페이지 연산을 시작하기 전에 먼저 CR3의 값을 가지고 페이지 딕렉토리의 물리적인 주소를 찾을 수 있다⁷³.

Intel에는 세가지 페이지 사용방법이 있다. 각각 4Kbytes의 페이지 크기를 가지는 것과, 2Mbytes의 페이지 크기를 가지는 것, 4Mbytes의 페이지 크기를 가지는 것으로 나누어진다. 이를 각각을 사용하는 방법은 CR0 레지스터의 PG플랙을 설정하고, CR4레지스터의 PAE플랙과 PSE플랙, 그리고, 페이지 딕렉토리의 PS플랙 필드를 조합하는 방법이다. 아래의 표와 같다.⁷⁴

CR0(PG flag)	CR4(PAE flag)	CR4(PSE flag)	Page Directory(PS flag)	Page Size	Physical Addr. Size
0	X	X	X	-	Paging Disabled
1	0	0	X	4 Kbytes	32bits
1	0	1	0	4 Kbytes	32bits
1	0	1	1	4 Mbytes	32bits
1	1	X	0	4Kbytes	36bits
1	1	X	1	2Mbytes	36bits

표 30. 페이지의 종류

4Mbytes의 크기를 나타내는 페이지를 사용할 경우는 이중에서 CR0의 PG flag를 1로, CR4의 PAE는 0, PSE는 1, 페이지 딕렉토리의 PS가 1을 가지는 경우다. 이 경우 물리적인 주소는 32bits를 가진다. 이 경우에는 페이지 테이블은 사용되지 않으며, 페이지 딕렉토리와 옵셋만을 가지고, 물리적인 주소를 가르킨다. 즉, 옵셋 부분이 22 bit의 길이를 가지며, 페이지 딕렉토리는 10bit의 길이를 가진다.

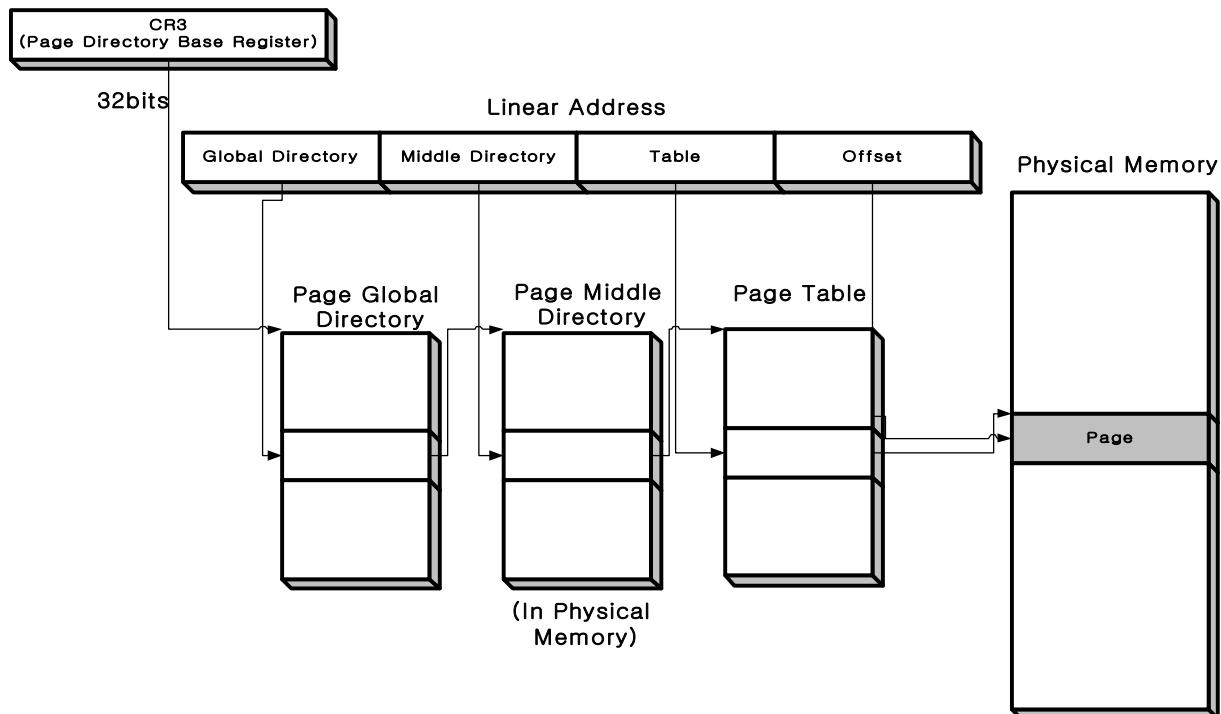


그림 38. 리눅스에서의 페이지

리눅스에서 페이지를 사용하는 방법은 Intel CPU만을 대상으로 하지 않는다. 즉, 리눅스에서는 위에서 제시한 페이지 딕렉토리를 다시 조금더 나누어서, 페이지 글로벌 딕렉토리(page global directory)와 페이지

⁷³ Intel CPU의 제어 레지스터는 총 4개가 있으며, CR0, CR1, CR2, CR3로 나타낸다. 이것은 나중에 부팅 과정을 설명하면서 보게될 것이다.

⁷⁴ PG 플랙은 Paging을, PAE는 Paging Address Extension, PSE는 Page Size Extension을 각각 나타낸다.

미들 디렉토리(page middle directory)가 있다고 생각한다. 하지만, Intel CPU에 이것을 적용할 때는 페이지 미들 디렉토리는 있지 않은 것으로 여긴다. 즉, 페이지 미들 디렉토리가 한개의 엔트리만을 가진다고 정의해서 페이지 글로벌 디렉토리의 적절한 엔트리로 맵핑 시킨다. 전체적인 구조는 [그림31]과 같다.

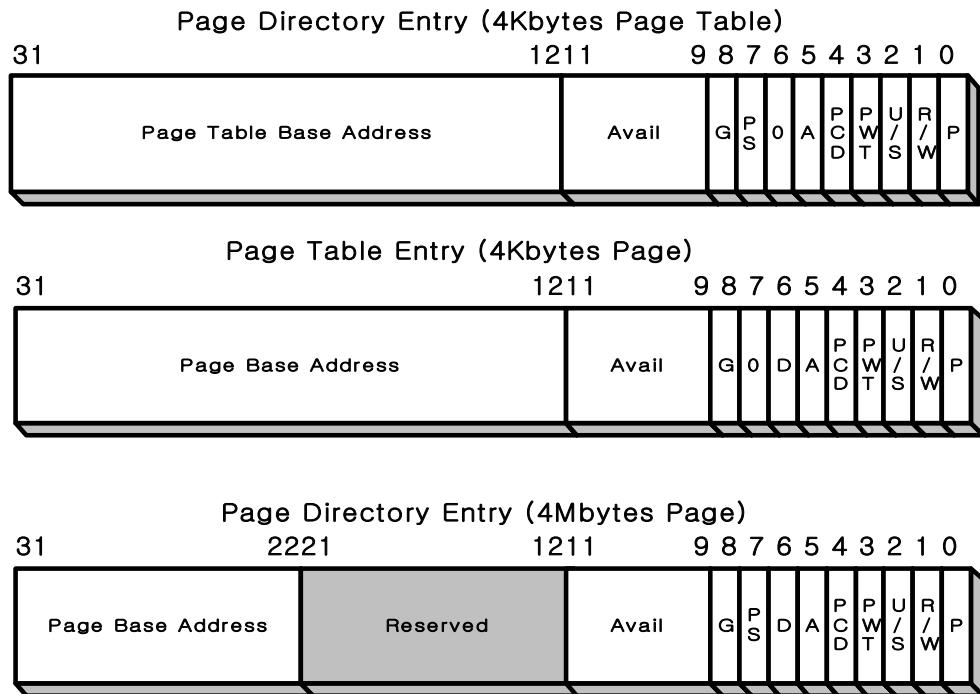


그림 39. 페이지 디렉토리와 테이블의 구조

페이지 디렉토리와 페이지 테이블의 엔트리는 [그림32]와 같다. 즉, 페이지 디렉토리와 페이지 테이블의 엔트리는 거의 같은 구성을 가지고 있으며, 4Mbytes의 페이지를 사용할 때는 페이지 테이블을 사용하지 않는다.

페이지 디렉토리와 테이블 필드들의 의미는 다음과 같다.

- **페이지 기본 주소(Page base address) :** 4Kbytes의 페이지에 대한 물리적인 주소를 나타낸다. 즉, 페이지 단위로 메모리를 나누었을 경우에 4Kbytes단위로 페이지들이 정렬되어 있다는 것을 알 수 있다. 물리적인 주소의 첫 20bit이 해당한다. 페이지 테이블의 기본 주소가 될 경우는 페이지 테이블이 4Kbytes단위로 정렬됨을 나타낸다. 4Mbytes단위의 페이지라면 물리적인 주소의 상위 10bit가 해당되며, 페이지는 4Mbytes단위로 정렬된다.
- **P(Present) 플랙 :** 현재 페이지나 페이지 테이블이 물리적인 메모리에 적재 되어 있음을 알린다. 만약 지워져 있다면(0), 현재 페이지에 찾고자 하는 페이지나 페이지 테이블이 존재하지 않으므로 페이지 폴트를 일으킨다. 만약 확장된 물리적인 주소 지정방식을 사용한다면 항상 1로 설정된다. 운영체제가 페이지 폴트를 처리하는 것은 아래와 같다.
 - 해당 페이지를 디스크로 부터 복사해서 물리적인 메모리로 가져온다.
 - 페이지의 주소를 페이지 테이블이나 페이지 디렉토리의 엔트리로 가져오고, P플랙을 설정한다. D와 A플랙도 이때 설정될 수 있다.
 - TLB(Translation Lookaside Buffer)⁷⁵ 의 현재 페이지 테이블에 대한 엔트리는 무효화(invalidate)된다.

⁷⁵ TLB는 페이지에 대한 연산 속도를 높이기 위한 것으로 고속의 선형주소 변환이 처리된다. `invlpg` 명령어로 TLB의 한 엔트리를 무효화 하며, 모든 TLB의 내용을 무효화하기 위해선 CR3레지스터의 새로운 내용을 쓰면된다. TLB는 페이지 테이블의 캐ши 역할을 하기 때문에, 페이지 테이블의 엔트리가 바뀐다면, 커널은 해당하는 TLB를 항상 무효화해 주어야 할 것이다. 이것을 담당하는 리눅스 함수는 `flush_tlb_page()`이다. 이곳에 페이지 테이블에 해당하는 주소를 넘겨준다. 이 함수는 다시

- 페이지 폴트는 처리가 되었으며, 이전 페이지 폴트를 일으킨 프로그램이나 작업(task)를 계속 진행한다.
- R/W(Read/Write) 플랙 : 페이지나 페이지의 그룹에 대해서 읽기와 쓰기의 권한을 나타낸다. 페이지 딕렉토리가 페이지 테이블을 가리키게 되면 그룹단위의 페이지가 이러한 권한을 적용받는다. 0이면 read-only를, 1인 경우에는 read/write가 가능하다.
- U/S(User/Supervisor) 플랙 : 페이지나 페이지의 그룹에 대해서 사용자나 관리자(supervisor)권한을 설정한다. 0이면 관리자를 위한 것이며, 1인경우에는 사용자권한으로 설정되었다는 것을 말해준다.
- PWT(Page-level Write-Through) 플랙 : write-through나 write-back 캐싱(caching) 방법⁷⁶을 개객의 페이지들에, 혹은 페이지 테이블들에 설정한다. 1인 경우 write-through 캐싱이 관련된 페이지나 페이지 테이블에 설정되며, 0인경우에는 write-back 캐싱이 설정된다. 만약 CR0의 캐쉬 불가(CD)가 설정되면, 프로세스는 이것을 무시한다.
- PCD(Page-level Cache Disable) 플랙 : 개개의 페이지들이나 페이지 테이블들에 대한 캐싱을 제어한다. 1로 설정되었다면, 관련된 페이지나 페이지 테이블에 대한 캐싱을 하지 않는다. 이것은 memory-mapped I/O port나 캐쉬되었을 경우에 성능상의 효용가치가 없을 경우에 캐싱을 하지 않도록 하는 방법이다. 만약 CR0레지스터의 CD가 설정된 경우에는 무시된다.
- A(Accessed) 플랙 : 페이지나 페이지 테이블이 접근(accessed)되었음을 나타낸다. 만약 페이지나 페이지 테이블이 물리적인 메모리로 적재되면, 메모리 관리를 받은 부분에서 이것을 0으로 둔다. 소프트웨어적으로만 이것을 지우는 것이 가능하다. 이 플랙과 D플랙이 메모리 관리 소프트웨어에서 물리적인 메모리로 페이지나 페이지 테이블을 가져오거나 쓸때 필요한 정보를 제공한다.
- D(Dirty) 플랙 : 1로 설정되었다면, 페이지는 쓰기(write)가 있었음을 나타낸다. 페이지가 초기에 물리적인 메모리로 적재되면 0으로 메모리 관리 소프트웨어가 설정한다. 페이지가 처음으로 쓰기 연산을 위해서 접근되면, 프로세스에서 이 플랙을 1로 설정한다. 한번 1로 설정하고나면 이것은 다시 프로세스가 0으로 설정하지 않는다. 다시 0으로 설정하는 것은 메모리 관리 소프트웨어의 몫이다.
- PS(Page Size) 플랙 : 페이지의 크기를 정한다. 이 플랙은 페이지 딕렉토리의 엔트리에만 사용되며, 0인 경우에는 4Kbytes의 페이지 크기를, 1인 경우에는 4Mbytes의 32bit주소지정 방식에서 사용한다. 만약 페이지 딕렉토리가 페이지 테이블을 가리킨다면, 페이지 테이블과 관련된 모든 페이지는 4Kbytes의 크기를 가질 것이다.
- G(Global) 플랙 : 1인경우에 global 페이지임을 나타낸다. CR4 레지스터의 PGE가 설정되었고, 이 플랙이 1이라면, 페이지를 위한 페이지 테이블이나 페이지 딕렉토리의 TLB내 엔트리는, CR3레지스터의 내용이 로드되거나 작업전환(task switch⁷⁷)가 있을 시에도 무효화(invalidate)되지 않는다. 이 플랙은 자주 사용되는 커널이나 다른 운영체제, 혹은 실행 코드에 대해서 사용될 수 있을 것이다. 단지 소프트웨어적으로만 이것을 설정(set)하거나 지울(clear) 수 있다. 만약 페이지 딕렉토리 엔트리가 페이지 테이블을 가리킨다면 이것은 무시되며, 페이지에 대한 Global 플랙은 페이지 테이블에 설정된다.
- Reserved/Avail(Available) : 시스템 소프트웨어에서 사용할 수 있거나 항상 0이란 값을 가져야하는 부분이다. CR4의 PSE와 PAE가 설정(set=1)되었다면, 프로세스는 예약된(reserved) 부분에 0이 설정되지 않았을 경우에 페이지 폴트(fault)를 만들어 낼 것이다.

`_flush_tlb_one()`을 호출하고, 결과적으로 `invlpg`를 사용한다. 모든 TLB의 내용을 무효화시키는 방법은 `_flush_tlb()`함수이다. 이것은 CR3레지스터에 현재의 값을 새로 써 넣는다. 이와 비슷한 것으로 모든 프로세스의 TLB를 비우는 `_flush_tlb_all()`함수도 존재한다. `~/include/asm/pgalloc.h`를 참조하라.

⁷⁶ Write-Through로 설정되면 프로세스의 캐쉬와 램이 항상 같은 내용을 가지게 되며, Write-Back이 선택되면, 캐쉬의 내용만이 write되고, 램은 바뀌지 않는다. 나중에 캐쉬 콘트롤러가 캐쉬에 있는 내용(cache line), CPU가 캐쉬를 비우거나(flush) 하드웨어의 FLUSH가 발생할 때(cache miss) 램에 쓰게 된다.

⁷⁷ 환경전환(Context Switch)를 말한다.

이상에서 우린 페이지 디렉토리와 페이지 테이블에 대해서 알아보았다. 만약 P(Present)가 설정되지 않은 페이지 디렉토리나 테이블에 대해서, 커널은 디스크 내에서 페이지의 위치를 나타내기 위해 임의로 사용할 수도 있다.

4.4. Reserved Memory Area

물리적인 메모리 공간은 많은 다른 목적으로 사용된다. 가령 예를 들어서 PC의 경우에 특정 메모리 번지에 대한 read나 write는 시스템에 붙어있는 장치에 대한 read/write로 인식된다. 또한 커널이 사용하는 부분의 메모리 공간은 일반적인 사용자 프로세스가 사용하는 메모리 공간과는 다른 성질을 띤다. 즉, 항상 커널은 메모리 속에 유지하고 있어야 하지만, 사용자 프로세스와 같은 중요치 않은 코드나 데이터는 디스크에 잠시 저장해서, 사용되는 시점에만 불러들여서 쓰면 될 것이다. 이렇게 해서 실제 물리적인 메모리 이상으로 많은 프로세스를 생성 및 실행할 수 있게 된다.

따라서, 커널이나 주변 기기들이 사용하는 물리적인 메모리를 따로 관리하거나 예약해 두는데, 다음과 같다.

- 커널의 코드와 데이터 구조는 예약된 페이지 프레임에 저장된다. 절대 동적으로 지정되지 않으며, 디스크에 스왑(swap)⁷⁸되지 않는다. 일반적으로 커널은 메모리의 0x00100000(1Mbytes) 영역에서부터 시작된다.
- 페이지 프레임 0은 BIOS에 의해서 사용된다. 이 부분은 POST(Power On Self Test)에서 감지된 시스템의 하드웨어 설정 정보를 저장할 목적으로 쓰인다.
- 0x000A0000에서 0x000FFFFF은 BIOS 함수들에 예약된 부분으로 시스템에 연결된 그래픽 카드들에 예약된 부분이다. 예를 들어서 VGA 그래픽 카드인 경우에는 0x000C0000에서부터 시작하는 메모리를 비디오의 메모리로 사용한다. 이 영역에 write연산을 수행하면 화면상에 바로 표시될 것이다.
- 또한 특정 하드웨어는 1Mbytes 이하의 메모리에서 특정부분을 예약해서 사용하기도 한다.

커널은 위와 같은 영역을 시스템의 초기화시에 설정한다. 즉, 이 부분에 대한 페이지 프레임을 가지는 페이지 테이블의 엔트리를 사용자 프로세스가 사용하지 못하도록 한다. [그림33]은 이와 같은 것을 보여준다. _text, _etext, _edata, _end는 커널을 컴파일 할때 ld(Loader)가 사용하는 ~/arch/i386/vmlinux.lds에 정의되어 있으며, 여기서는 물리적인 메모리의 첫 2Mbytes에 대한 페이지 프레임만 보여준다.

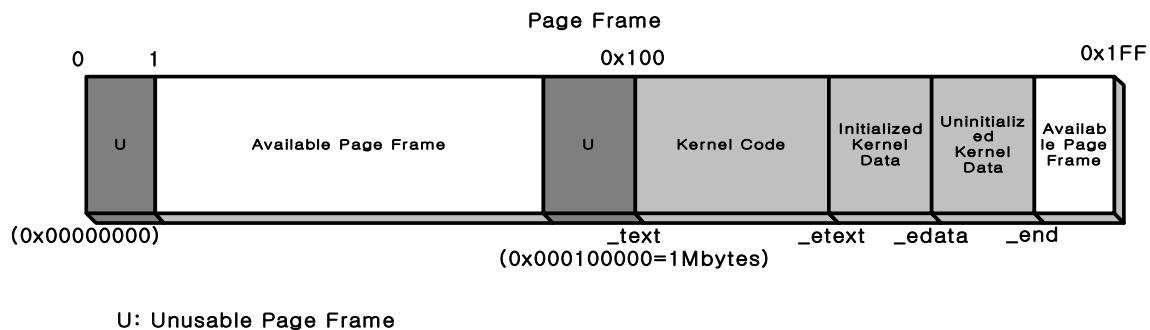


그림 40. 커널과 예약 영역을 위한 페이지 프레임들

사용자 프로세스는 선형주소의 0x00000000에서 0xC0000000 – 1 (=PAGE_OFFSET – 1)로 정의되며, 커널은 0xC0000000에서 0xFFFFFFFF 공간을 차지한다. 각각의 프로세스는 0xC0000000이하의 주소에 대해서 자신만의 페이지 디렉토리를 가지고 있으며, 0xC0000000이상의 주소공간에 대한 페이지 디렉토리는 모든 프로세스에 대해서 동일하다.

⁷⁸ 메모리의 내용을 디스크의 일부 공간을 스왑영역으로 지정해서 임시적으로 저장해 두는 것을 말한다.

4.5. 커널에서 사용하는 페이지 테이블

커널은 자신만의 메모리 공간을 부팅시에 확보한다. 즉, 커널 자신이 사용하게 될 페이지 테이블에 대한 초기화를 수행해야만 한다. 커널은 부팅시에 페이지 시스템을 사용하도록 설정되며, 페이지에 필요한 페이지 global directory와 페이지 테이블이 초기화 된다. 페이지 글로벌 딕토리는 swapper_pg_dir 변수를 사용해서 나타내지며, 첫 4Mbytes에 대한 페이지 테이블은 pg0 변수가 가르킨다. 각각은 아래와 같이 정의되어서 사용된다.(~/arch/i386/kernel/head.S 참조)

```
.org 0x1000
ENTRY(swapper_pg_dir)
    .long 0x00102007
    .long 0x00103007
    .fill BOOT_USER_PGD_PTRS-2,4,0
    /* default: 766 entries */
    .long 0x00102007
    .long 0x00103007
    /* default: 254 entries */
    .fill BOOT_KERNEL_PGD_PTRS-2,4,0
/*
 * The page tables are initialized to only 8MB here - the final page
 * tables are set up later depending on memory size.
 */
.org 0x2000
ENTRY(pg0)

.org 0x3000
ENTRY(pg1)
```

코드 252. 커널의 페이지 딕토리 정의

여기서 PAGE_OFFSET으로 사용되는 값은 0xC0000000이며, BOOT_USER_PGD_PTRS와 BOOT_KERNEL_PGD_PTRS는 각각 아래와 같이 정의된다.

```
#define TWOLEVEL_PGDIR_SHIFT 22
#define BOOT_USER_PGD_PTRS (_PAGE_OFFSET >> TWOLEVEL_PGDIR_SHIFT)
#define BOOT_KERNEL_PGD_PTRS (1024-BOOT_USER_PGD_PTRS)
```

즉, 페이지 딕토리와 페이지 테이블을 위해서 22bit을 사용하기 때문에, TWOLEVEL_PGDIR_SHIFT가 22라는 값을 가지며, BOOT_USER_PGD_PTRS는 __PAGE_OFFSET(PAGE_OFFSET=0xC0000000)값을 22bit 오른쪽으로 shift한 값을 가진다. 즉, 0x300(768)이란 값이 된다. 다시 BOOT_KERNEL_PGD_PTRS는 1024에서 0x300을 뺀 값을 가지게 되므로, $0x400 - 0x300 = 0x100$ 이 된다. 즉, 256이란 값을 가지게 된다. 위에서 swapper_pg_dir는 초기화로 USER를 위해서 768개 중에서 첫 2개를 제외하고, 766개에 대해서 4bytes 단위로 0으로 초기화를 하고, 나머지 커널을 위해, 256개 중에서 첫 2개를 제외하고, 254개에 대해서 다시 4bytes 단위로 0으로 초기화 시킨다. pg0는 0x2000에서, pg1은 0x3000에서 각각 시작한다.

swapper_pg_dir에서 첫 번째 두개의 엔트리에 대해서는 0x00102007과 0x00103007가 사용되었다. 따라서, 12bit에서부터 31bit까지는 0x00102와 0x00103이 사용되었다. 각각이 페이지 테이블의 첫 엔트리를 가리킨다. 나머지 0x007은 U/S와 R/W, P에 대한 설정으로 111b이기에 사용자 페이지 테이블을 가리키며, 읽기와 쓰기가 가능하고, 현재 메모리내에 존재한다는 것을 의미한다. 또한 pg0와 pg1만을 설정해서, 첫 2개의 4Mbytes 영역에 대한 초기화를 수행하고 있으며, 나머지 영역(8Mbytes 이상)에 대해서는 나중에 실제 메모리를 검사해서 초기화 시킨다.

이제는 시스템이 페이지를 하도록 제어 레지스터(control register)를 설정하는 일이 남게 된다. 이것은 booting에서 보게되겠지만, ~/arch/i386/kernel/head.S에서 아래와 같은 부분을 찾을 수 있을 것이다.

```

movl $swapper_pg_dir-__PAGE_OFFSET,%eax
movl %eax,%cr3      /* set the page table pointer.. */
movl %cr0,%eax
orl $0x80000000,%eax
movl %eax,%cr0      /* ..and set paging (PG) bit */
jmp 1f                /* flush the prefetch-queue */

1:

```

코드 253. 페이징 시스템의 초기화

swapper_pg_dir에서 __PAGE_OFFSET(0xC0000000)을 빼서 EAX레지스터에 보관한 다음, 커널에서 사용할 페이지 글로벌 디렉토리에 대한 베이스(base) 포인터 역할을 하는 CR3레지스터에 넣는다. 그리고나서 CR0에 있는 내용을 EAX 레지스터로 불러드려서, 0x80000000으로 OR시킨다. 이때 PG bit이 설정되어 페이징이 가능하게 된다. 그리고, 이 내용을 다시 CR0에 써서, 페이징의 시작을 CPU에 알린다. 이하에서는 1f(forward)로 jump해서 이전에 prefetch된 내용을 비우도록 만든다. 이것으로 페이징 시스템이 enable된다.

4.6. 페이지 디렉토리 및 페이지 테이블에 대한 초기화

더 이상 진행하기 전에 먼저 커널에서 페이징 시스템을 위해서 가지는 몇몇 구조체 들을 보기로 하자. 이것에는 페이지 글로벌 디렉토리와 페이지 미들 디렉토리, 그리고, 페이지 테이블이 있겠다.

```

...
#endif CONFIG_X86_PAE
typedef struct { unsigned long pte_low, pte_high; } pte_t;
typedef struct { unsigned long long pmd; } pmd_t;
typedef struct { unsigned long long pgd; } pgd_t;
#define pte_val(x)    ((x).pte_low | ((unsigned long long)(x).pte_high << 32))
#else
typedef struct { unsigned long pte_low; } pte_t;
typedef struct { unsigned long pmd; } pmd_t;
typedef struct { unsigned long pgd; } pgd_t;
#define pte_val(x)    ((x).pte_low)
#endif

...
typedef struct { unsigned long pgprot; } pgprot_t;
#define pmd_val(x)    ((x).pmd)
#define pgd_val(x)    ((x).pgd)
#define pgprot_val(x) ((x).pgprot)

#define __pte(x) ((pte_t) { (x) } )
#define __pmd(x) ((pmd_t) { (x) } )
#define __pgd(x) ((pgd_t) { (x) } )
#define __pgprot(x)   ((pgprot_t) { (x) } )
...

```

코드 254. ~/include/asm/page.h

CONFIG_X86_PAE가 설정되었다면, 페이지 테이블은 low와 high 각각에 32bit를 사용하는 것을 바꿔며, 그렇지 않을 경우에는 단순히 32bit만을 가진다. 페이지 글로벌 디렉토리와 페이지 미들 디렉토리도 각각 CONFIG_X86_PAE의 설정에 따라서, 64bit 과 32bit 크기를 가진다.

이전에서 하던 이야기로 돌아가서, 앞에서는 `swapper_pg_dir`에 대한 일반적인 초기화만을 보았다. 아직까지는 정확히 커널과 사용자 프로세스가 사용할 페이지 디렉토리에 대한 설정은 이루어 지지 않은 상태이며, 아래의 코드를 수행하는 것으로 초기화가 이루어진다. 이후부터는 커널과 사용자 프로세스가 다른 페이지 디렉토리에 대한 접근을하게 될 것이다.

페이지를 위한 페이지 디렉토리 및 페이지 테이블에 대한 초기화는 `~/init/main.c`의 `start_kernel()`함수에서 `setup_arch()`함수를 호출하면서 이루어 진다. `setup_arch()`함수는 `command_line`을 인수로 넘겨받는 함수이며, 정의는 `~/arch/i386/kernel/setup.c`에 있다. `setup_arch()`함수는 다시 `paging_init()`함수를 호출하게 되며, `paging_init()`함수는 `~/arch/i386/mm/init.c`에 있으며, 이곳에서 `pagetable_init()`함수를 호출하면서 페이지를 위한 페이지 디렉토리 및 페이지 테이블의 실제적인 설정이 이루어진다. `paging_init()`함수는 아래와 같이 정의되어 있다.

```
void __init paging_init(void)
{
    pagetable_init();
    __asm_( "movl %%ecx,%%cr3\n" ::"c"(__pa(swapper_pg_dir)));
#ifndef CONFIG_X86_PAE
    if (cpu_has_pae)
        set_in_cr4(X86_CR4_PAE);
#endif
    __flush_tlb_all();
#ifndef CONFIG_HIGHMEM
    kmap_init();
#endif
    {
        unsigned long zones_size[MAX_NR_ZONES] = {0, 0, 0};
        unsigned int max_dma, high, low;

        max_dma = virt_to_phys((char *)MAX_DMA_ADDRESS) >> PAGE_SHIFT;
        low = max_low_pfn;
        high = highend_pfn;
        if (low < max_dma)
            zones_size[ZONE_DMA] = low;
        else {
            zones_size[ZONE_DMA] = max_dma;
            zones_size[ZONE_NORMAL] = low - max_dma;
#ifndef CONFIG_HIGHMEM
            zones_size[ZONE_HIGHMEM] = high - low;
#endif
        }
        free_area_init(zones_size);
    }
    return;
}
```

코드 255. `paging_init()` 함수

`paging_init()`함수는 페이지 테이블에 대한 초기 설정을 한다. 주의할 점은 이미 `head.S`에 의해서 첫 8Mbytes 영역에 대해서는 매팅이 이루어져 있다는 것이다. 또한 커널의 주소 0번에 대한 페이지는 `unmap`시켜서 커널내에서 NULL에 대한 언급(reference)에 대해서 트랩(trap)을 걸 수 있다는 점이다. 복귀하기 전에 `free_area_init()`함수를 호출해서 free 페이지를 관리하기 위한 기본적인 커널 자료구조를 생성한다.

먼저 함수는 `pagetable_init()`함수를 호출해서 페이지 테이블을 초기화한다. 이것은 아래에서 볼 것이다. 이것을 마치면 `swapper_pg_dir`의 물리적인 주소를 CR3 제어 레지스터에 넣어서 커널을 위한 PGD의 기본 주소를 설정한다. 만약 `CONFIG_X86_PAE`설정을 가지고 있다면, CR4 제어 레지스터의 PAE bit을 설정한다. 이것을 마쳤다면, 이젠 TLB(Translation Lookaside Buffer)를 완전히 비우도록

한다(`__flush_tlb_all()`). 만약 `CONFIG_HIGHMEM` 설정이 있다면, `high memory`에 대한 초기화도 이루어진다(`kmap_init()`).

`zonee_size[]` 배열은 각각의 `zone0` 가지는 크기를 엔트리로 가진다. 먼저 최대 DMA 가능 주소(`MAX_DMA_ADDRESS = PAGE_OFFSET + 0x1000000 = 0xC1000000`)의 물리적인 주소를 구해서(`virt_to_phys()`) `max_dma`에 넣는다. 그리고, PFN(Page Frame Number)의 하한선과 상한선을 구하고, 만약 PFN의 하한선이 `max_dma`보다 작다면, 16Mbytes보다 작은 물리적인 주소를 가지고 있으므로 DMA가 가능한 주소를 낮춰주어야 한다(`zones_size[ZONE_DMA]=low`). 그렇지 않다면, 각각의 존에 대해서 앞에서 구한 값을 적절히 넣어주도록 한다. 이전 `zone`들에 대해서 크기만을 알게 되었으므로, 이 크기를 바탕으로 전체 free 메모리에 대한 초기화를 위해서 `free_area_init()` 함수를 호출한다. 넘겨주는 것은 각 `zone`의 크기를 가지는 배열(`zones_size`)이다. 이 배열에 들어가는 것은 PFN이라는 점을 눈여겨보도록 하자. 즉, 나중에 이 PFN 번호를 가지고 초기화를 수행한다.

```
static void __init pagetable_init (void)
{
    unsigned long vaddr, end;
    pgd_t *pgd, *pgd_base;
    int i, j, k;
    pmd_t *pmd;
    pte_t *pte;

    end = (unsigned long) __va(max_low_pfn*PAGE_SIZE);
    pgd_base = swapper_pg_dir;
#ifndef CONFIG_X86_PAE
    for (i = 0; i < PTRS_PER_PGD; i++) {
        pgd = pgd_base + i;
        __pgd_clear(pgd);
    }
#endif
    i = __pgd_offset(PAGE_OFFSET);
    pgd = pgd_base + i;
```

코드 256. `pagetable_init()` 함수

`pagetable_init()` 함수는 먼저 `max_low_pfn`에 `PAGE_SIZE(4096 bytes)`를 곱한 것의 가상 주소(Virtual Address)를 얻어온다(`__va()`). 또한 페이지 글로벌 딕렉토리의 기본 주소로 `swapper_pg_dir`의 범지를 가져온다. 만약 `CONFIG_X86_PAE`(Page Address Extension)이 설정되었다면, `swapper_pg_dir`에 들어있는 페이지 딕렉토리들에 대해서 비어있다고 만든다(`__pgd_clear()`).

그렇지 않다면, `PAGE_OFFSET(=0xC0000000)`에서 페이지 글로벌 딕렉토리의 인덱스 값을 가져온다(`__pgd_offset()`). 이 값과 `pgd_base`를 합해서 `pgd`(페이지 글로벌 딕렉토리) 값을 얻는다.

여기서 `PTRS_PER_PGD`는 2 level의 페이지와 3 level의 페이지에 대해서 다른 값을 가지는데, 각각 대해서 1024와 4를 가진다. 이것은 페이지 글로벌 딕렉토리당 몇개의 포인터를 가지는지를 말해주는 값으로 현재 `CONFIG_X86_PAE`가 설정되었다면, 3 level을 그릴지 않다면, 2 level을 사용한다.

```
for (; i < PTRS_PER_PGD; pgd++, i++) {
    vaddr = i*PGDIR_SIZE;
    if (end && (vaddr >= end))
        break;
#ifndef CONFIG_X86_PAE
    pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
    set_pgd(pgd, __pgd(__pa(pmd) + 0x1));
#else
    pmd = (pmd_t *)pgd;
```

```
#endif
    if (pmd != pmd_offset(pgd, 0))
        BUG();
    for (j = 0; j < PTRS_PER_PMD; pmd++, j++) {
        vaddr = i*PGDIR_SIZE + j*PMD_SIZE;
        if (end && (vaddr >= end))
            break;
        if (cpu_has_pse) {
            unsigned long __pe;

            set_in_cr4(X86_CR4_PSE);
            boot_cpu_data.wp_works_ok = 1;
            __pe = _KERNPG_TABLE + _PAGE_PSE + __pa(vaddr);
            /* Make it "global" too if supported */
            if (cpu_has_pge) {
                set_in_cr4(X86_CR4_PGE);
                __pe += _PAGE_GLOBAL;
            }
            set_pmd(pmd, __pmd(__pe));
            continue;
        }
        pte = (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);
        set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte)));
        if (pte != pte_offset(pmd, 0))
            BUG();
        for (k = 0; k < PTRS_PER_PTE; pte++, k++) {
            vaddr = i*PGDIR_SIZE + j*PMD_SIZE + k*PAGE_SIZE;
            if (end && (vaddr >= end))
                break;
            *pte = mk_pte_phys(__pa(vaddr), PAGE_KERNEL);
        }
    }
}
```

코드 257. pagetable_init()함수 – continued

이전 for loop를 돌면서 페이지 글로벌 딕렉토리(PGD)와 페이지 테이블에 대한 설정을 할 차례이다. 만약 CONFIG_X86_PAE가 설정되었다면, 페이지 미들 딕렉토리(PMD)를 할당해서(alloc_bootmem_low_pages()), 물리적인 주소(__pa())를 PGD가 가르키도록 만들고(set_pgd()), 그렇지 않다면, 2 level 페이징을 사용하게 되므로 PMD를 PGD와 동일하게 놓는다(pmd = (pmd_t *)pgd). 만약 PGD에 첫번째로 들어있는 PMD의 0번째 엔트리와 pmd가 같은 값을 가지지 않을 경우에는 BUG()를 호출한다.

이전 PMD를 초기화 할 차례이다. For loop를 돌면서 초기화 한다. 먼저 PGDIR_SIZE와 PMD_SIZE에 적절한 값을 곱해서 vaddr로 놓는다. 만약 end에 어떤 값이 있고, 이 값이 vaddr보다 작은 값을 가진다면 loop를 멈춘다. 그렇지 않다면 다음을 수행한다. 먼저 현재 CPU가 PSE(Page Size Extension)의 설정을 가지고 있다면, CR4 제어 레지스터의 PSE부분을 설정하고(set_in_cr4()), boot_cpu_data.wp_works_ok에 1을 설정한다.

만약 CPU가 PGE(Global) 설정을 가지고 있다면, 다시 CR4 제어 레지스터의 PGE를 설정하고, 최종적으로 PMD를 __pe값으로 설정한다. __pe에 들어가는 값은 아래와 같이 정의된다.

```
#define _PAGE_PSE      0x080 /* 4 MB (or 2MB) page, Pentium+, if present.. */
#define _KERNPG_TABLE (_PAGE_PRESENT | _PAGE_RW | _PAGE_ACCEDED | _PAGE_DIRTY)
#define _PAGE_GLOBAL    0x100 /* Global TLB entry PPro+ */
```

여기서 주의할 점은 PGD(혹은 PMD)에 대해서 U/S를 1로 설정하고 있지 않다는 점이다. 즉, supervisor 권한을 가지고 있을 경우에만 접근을 허가하도록 만들고 있다.

이제 남은 것은 페이지 테이블(PTE)에 대한 설정이다. 먼저 PTE에 대한 메모리를 할당받는다(`alloc_boot_mem_low_pages()`). 이것을 페이지 테이블에 대한 타입으로 캐스팅해서 `pte`를 설정한다. PMD의 엔트리를 역시 할당한 PTE를 가리키도록 한 후, 커널에서 사용하는 페이지로 PMD의 bit을 설정한다(`set_pmd()`). 마찬가지고 올바르게 할당이 이루어졌는지를 확인한 후, 다시 PTE에 대한 초기화를 위해서 `for loop`를 돈다.

이번에는 `vaddr`로 `PGDIR_SIZE`와 `PMD_SIZE`, `PAGE_SIZE`를 같이 적절한 수로 곱해서 초기화하고, 이것이 마지막 주소(`end`)보다 크거나 같지 않을 때까지 `loop`를 진행한다. `pte`가 가지게 될 물리적인 기본(base) 주소와 bit를 설정한다. 여기서 사용하는 `PAGE_KERNEL` 설정 bit는 아래와 같이 정의된다.

```
/* ~/include/asm/page.h에서 */
typedef struct { unsigned long pgprot; } pgprot_t;
#define __pgprot(x) ((pgprot_t) { (x) } )

/* ~/include/asm/pgtable.h에서 */
#define __PAGE_KERNEL W
(_PAGE_PRESENT | _PAGE_RW | _PAGE_DIRTY | _PAGE_ACCESSED)
...
#endif CONFIG_X86_PGE
#define MAKE_GLOBAL(x) __pgprot((x) | _PAGE_GLOBAL)
#else
#define MAKE_GLOBAL(x) W
W
W
W
if (cpu_has_pge) W
    __ret = __pgprot((x) | _PAGE_GLOBAL); W
else W
    __ret = __pgprot(x); W
    __ret; W
})
#endif
#define PAGE_KERNEL MAKE_GLOBAL(__PAGE_KERNEL)
```

즉, `_PAGE_PRESENT`, `_PAGE_RW`, `_PAGE_DIRTY`, `_PAGE_ACCESSED` bit를 기본으로 설정하고, 나머지 `_PAGE_GLOBAL`을 `cpu_has_pge`에 따라서 설정한다. 만약 `CONFIG_X86_PGE`가 설정되었다면 반드시 `_PAGE_GLOBAL`을 설정하도록 한다.

```
vaddr = __fix_to_virt(__end_of_fixed_addresses - 1) & PMD_MASK;
fixrange_init(vaddr, 0, pgd_base);
#if CONFIG_HIGHMEM
vaddr = PKMAP_BASE;
fixrange_init(vaddr, vaddr + PAGE_SIZE*LAST_PKMAP, pgd_base);
pgd = swapper_pg_dir + __pgd_offset(vaddr);
pmd = pmd_offset(pgd, vaddr);
pte = pte_offset(pmd, vaddr);
pkmap_page_table = pte;
#endif
#if CONFIG_X86_PAE
```

```

pgd_base[0] = pgd_base[USER_PTRS_PER_PGD];
#endif
}

```

코드 258. pagetable_init()함수 – continued

PGD와 PMD, PTE에 대한 초기화는 이미 이루졌다. 이전 고정되서 사용되는 페이지들에 대한 설정이 있다. 먼저 고정 주소의 마지막(_end_of_fixed_addresses)의 가상 주소를 PMD_MASK⁷⁹를 AND시켜서 가져온다(vaddr). fixrange_init()함수는 시작(start)과 마지막(end), 그리고, PGD의 기본 주소를 넘겨받아서, 고정주소에 대한 초기화를 담당하는 함수이다. 만약 CONFIG_HIGHMEM이 설정되었다면, PKMAP_BASE(=0xFE000000)이상의 고정 가상 메모리 영역에 대해서도 고정 주소의 PGD, PMD, PTE를 초기화한다⁸⁰.

마지막으로 이곳에서 자주 사용되는 _pa() 매크로와 _va() 매크로의 정의를 보도록 하자. ~/include/asm/page.h에 정의되어 있으며 아래와 같다.

```

#define __PAGE_OFFSET          (0xC0000000)
#define PAGE_OFFSET            ((unsigned long)__PAGE_OFFSET)
#define __pa(x)                 ((unsigned long)(x)-PAGE_OFFSET)
#define __va(x)                 ((void *)((unsigned long)(x)+ PAGE_OFFSET))

```

즉, 커널에서 사용하는 메모리는 3Gbytes이상의 영역, 즉, 0xC0000000(=__PAGE_OFFSET)에서 시작하는 가상 주소를 사용하므로, 어떤 가상 주소의 물리적인 주소를 구하고자 한다면, __PAGE_OFFSET를 가상 주소에서 뺀 값을 사용한다(__pa(x)). 따라서, 물리적인 주소를 가상 주소로 변환하고자 한다면, 다시 __PAGE_OFFSET를 더하면 될 것이다(__va(x)). 나중에 나오게 되는 디바이스 드라이버에서 사용하는 virt_to_phys()와 phys_to_virt()는 위에서 정의한 것을 이용해서 다시 아래와 같이 정의된다. ~/include/asm/io.h를 참고하라.

```

extern inline unsigned long virt_to_phys(volatile void * address)
{
    return __pa(address);
}

extern inline void * phys_to_virt(unsigned long address)
{
    return __va(address);
}

```

코드 259. 주소 변환 inline 함수들

즉, 간단히 _pa()와 _va() 매크로를 통하고 있음을 알 수 있다.

이상에서 우린 하드웨어 상에서 어떻게 페이징을 구현하는지와 이것을 가능하게 만드는 여러 커널의 데이터 구조에 대해서 알아보았다. 앞으로 볼 부분은 이것을 바탕으로 한 페이지 단위의 연산들 및 알고리즘이 될 것이다. 물론 페이지 보다 작은 메모리에 대해서는 하나의 페이지를 할당 받아서 여러개로 동일한 크기를 가지는 여러 블록들로 나누어서 사용하는 것을 보게 될 것이다. 어쨌든 물리적인 메모리의 할당 단위는 반드시 한 페이지라는 점을 염두에 두도록 하자.

⁷⁹ PGD부분만을 얻는다.

⁸⁰ 아직 이 부분에 대한 확신을 가지고 있지는 않다. 하지만, 64bit addressing을 사용할 경우, 64Gbytes 영역의 가상 메모리에 대한 접근이 가능하게 되며, 이때, 4Gbytes이상의 영역에서 페이지 테이블을 새로이 생성해서 쓰는 것 같다.

4.7. 페이지에 대한 초기화

이제부터 볼 것은 페이지 단위로 메모리를 할당하는 부분이다. 커널은 물리적인 메모리를 페이지 단위로 할당 및 해제를 할 수 있다. 이와 같은 연산은 이전에 보았던 내용중에 페이지 테이블에 대한 변화를 일으키게 될 것이다. 즉, 물리적인 메모리의 할당이 일어나면, 새로운 페이지 테이블의 내용을 추가해야 할 것이다.

먼저 커널은 물리적인 페이지를 관리할 목적으로 페이지 구조체(page struct)를 정의하고 있다. 아래와 같은 구조를 가진다(~include/linux/mm.h를 참고하라).

```
typedef struct page {
    struct list_head list;
    struct address_space *mapping;
    unsigned long index;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags; /* atomic flags, some possibly updated asynchronously */
    struct list_head lru;
    unsigned long age;
    wait_queue_head_t wait;
    struct page **pprev_hash;
    struct buffer_head * buffers;
    void *virtual; /* non-NULL if kmapped */
    struct zone_struct *zone;
} mem_map_t;
```

코드 260. page 구조체의 정의

자주 사용되는 것들을 구조체의 앞쪽에 배치 시켜서 CPU의 캐시를 고려하고 한다. 각각의 필드를 살펴보면 아래와 같다.

Field	Description
struct list_head list	페이지 구조체의 연결 리스트
struct address_space *mapping	메모리 매핑에 사용됨
unsigned long index	페이지 구조체의 인덱스
struct page *next_hash	페이지 구조체의 hash에서 다음을 가리킴
atomic_t count	페이지 구조체의 사용 카운터
unsigned long flags	페이지 구조체의 flag(아래에 다시 정리)
struct list_head lru	페이지 구조체의 LRU(Least Recently Used) 리스트
unsigned long age	메모리로 부터 지울 페이지를 선택하기 위해서 사용하는 카운터
wait_queue_head_t wait	페이지가 사용가능 할 때까지 기다리기 위한 대기 큐
struct page **prev_hash	페이지 구조체의 hash에서 앞에 있는 페이지 구조체를 가리킴
struct buffer_head *buffers	페이지에 관련된 버퍼 캐ши의 첫번째 버퍼 헤드에 대한 포인터
void *virtual	페이지 구조체의 가상 번지
struct zone_struct *zones	페이지가 어떤 메모리 부분을 가지는지 말해준다.(DMA, NORMAL, HIGHMEM)

표 31. 페이지 구조체의 필드 정의

페이지 구조체의 flag에 사용되는 값은 아래와 같은 의미를 가진다. 그리고, 전체 32bit중의 일부만 사용한다.

- PG_locked (0) : 페이지가 메모리에 lock되어 있다.

- PG_error (1) : 페이지가 메모리로 load될 때 에러가 발생했다.
- PG_referenced (2) : 페이지가 이미 접근(accessed)되었다.
- PG_uptodate (3) : 페이지의 내용이 최근(up to date) 것이다.
- PG_dirty (4) : 페이지의 내용에 변화(change)가 있었다.
- PG_decr_after (5) : 진행중인 input/output 연산의 끝에서 페이지의 reference count가 감소(decrement)되어야 한다.
- PG_active (6) : 페이지가 현재 활성중이다(active).
- PG_inactive_dirty (7) : 페이지가 현재 활성중이 아니며, dirty상태이다.
- PG_slab (8) : 페이지가 slab⁸¹에서 사용중이다.
- PG_swap_cache (9) : 페이지가 swap cache에서 사용중이다.
- PG_skip (10) : Sparc CPU에서 사용한다. 주소 공간의 일부를 생략(skip)한다.
- PG_inactive_clean (11) : 페이지가 inactive상태이며, 깨끗하다(clean).
- PG_highmem (12) : 페이지가 high memory영역에 있다.
- PG_arch_1 (30) : 페이지의 CPU에 의존적인 상태를 기록하는 bit이다.
- PG_reserved (31) : 커널 코드를 위해서 예약되었거나, 사용할 수 없는 페이지이다.

이상에서 정의한 것들 이외의 flag bit은 현재 사용되지 않고 있다. 커널 2.2.X에서 사용된 PG_DMA부분은 페이지 구조체의 zone으로 역할을 넘겨 주었다. 아래에 보여주는 것은 이와 같은 flag를 설정하거나 검사하는 매크로 들이다. 각각의 매크로는 이름이 의미하는 일을 해준다.

```
#define Page_Uptodate(page)           test_bit(PG_uptodate, &(page)->flags)
#define SetPageUptodate(page)         set_bit(PG_uptodate, &(page)->flags)
#define ClearPageUptodate(page)       clear_bit(PG_uptodate, &(page)->flags)
#define PageDirty(page)              test_bit(PG_dirty, &(page)->flags)
#define SetPageDirty(page)            set_bit(PG_dirty, &(page)->flags)
#define ClearPageDirty(page)          clear_bit(PG_dirty, &(page)->flags)
#define PageLocked(page)             test_bit(PG_locked, &(page)->flags)
#define LockPage(page)               set_bit(PG_locked, &(page)->flags)
#define TryLockPage(page)            test_and_set_bit(PG_locked, &(page)->flags)
```

코드 261 페이지 구조체의 flag의 설정 및 테스트 매크로 정의

PG_active와 PG_inactive_dirty 및 PG_inactive_clean은 각각이 LRU를 구현에 사용되며, 단계별로 최상위에 PG_active가 있으며, PG_inactive_dirty와 PG_inactive_clean이 순서대로 하위에 온다. 따라서, PG_inactive_clean이 LRU 알고리즘에서 가장 먼저 선택될 것이다. 이런 목적으로 각각에 대한 LRU를 위한 리스트가 유지 된다.

페이지는 크게 DMA가 가능한 영역에 있는 페이지와 그렇지 못한 영역에 있는 페이지, 그리고 high memory 영역에 있는 페이지로 나뉜다. 먼저 DMA가능한 영역은 ISA카드의 DMA를 지원하기 위한 부분에 해당하며, 그렇지 않은 일반 페이지에서는 ISA DMA가 불가능하다. 즉, 하위 16Mbyte 영역의 메모리가 DMA를 위해서 사용된다. 그리고, high memory영역은 4Gbytes 이상의 영역을 말한다.

이젠 각각의 자유(free) 메모리 영역에 대한 리스트를 커널이 만들어야 한다. 즉, 커널은 이 리스트를 기초로 해서 새로운 물리적인 메모리를 페이지 단위로 할당할 것이다. 전체 메모리에 대한 페이지 구조체의 연결 리스트는 **mem_map**가 가진다. 또한 자유 메모리 영역에 대한 초기화는 paging_init()함수에서 free_area_init()를 호출하면서 이루어진다. free_area_init()함수는 ~/mm/page_alloc.c에 정의되어 있으며, 다시 free_area_init_core()를 호출한다. 넘겨 받는 인수는 연속된(contiguous) 페이지에 대한 데이터와 mem_map, 그리고 각각의 zone⁸²에 대한 크기 정보가 있다.

⁸¹ Slab allocator라고 불리는 것으로 커널에서 메모리를 할당할 경우에 사용하는 방법들중의 하나이다.

⁸² DMA, NORMAL, HIGHMEM을 가리키는 zone들에 대한 크기 정보이다.

이곳에서 잠시 zone_struct구조체의 정의를 보도록 하자. free_area_init_core()함수에서 이 필드에 대한 초기화가 이루어 진다. 정의는 ~/include/linux/mmzone.h에 있다.

```

typedef struct free_area_struct {
    struct list_head    free_list; /* Free area 들의 list */
    unsigned int         *map;    /* Bitmap에 대한 포인터 */
} free_area_t;
struct pglist_data;           /* 페이지 리스트 데이터 */
typedef struct zone_struct {
    /* 자주 사용되는 필드들 */
    spinlock_t          lock;        /* zone_struct에 대한 lock */
    unsigned long        offset;     /* 옵셋 */
    unsigned long        free_pages; /* Free 페이지의 갯수 */
    unsigned long        inactive_clean_pages; /* Inactive clean 페이지의 갯수 */
    unsigned long        inactive_dirty_pages; /* Inactive dirty 페이지의 갯수 */
    unsigned long        pages_min, pages_low, pages_high; /* 페이지의 minimum, low, high 갯수
*/
    /* 다른 크기를 가지는 free area */
    struct list_head      inactive_clean_list; /* Inactive clean 페이지들의 리스트 */
    free_area_t           free_area[MAX_ORDER]; /* Free area 배열 */
    /* 잘 사용되지 않는 필드들 */
    char                 *name;       /* Zone의 이름 */
    unsigned long         size;        /* Zone의 크기 */
    /* 연속적이지 않은 메모리에 대한 지원을 위한 필드 */
    struct pglist_data    *zone_pgdat; /* 연속적이지 않은 페이지 리스트의 포인터 */
    unsigned long         zone_start_paddr; /* 연속적이지 않은 페이지의 물리적인 시작 주소 */
    unsigned long         zone_start_mapnr; /* 연속적이지 않은 페이지의 시작주소에 대한
맵핑의 수 */
    struct page           *zone_mem_map; /* mem_map에 대한 포인터 */
} zone_t;

```

코드 262. zone_struct구조체의 정의

free_area_init_core()함수는 전체 사용 가능한 메모리에 대한 관리를 초기화를 담당한다. 이곳에서 초기화가 이루어지는 커널의 자료구조는 아래의 [그림34]와 같다.

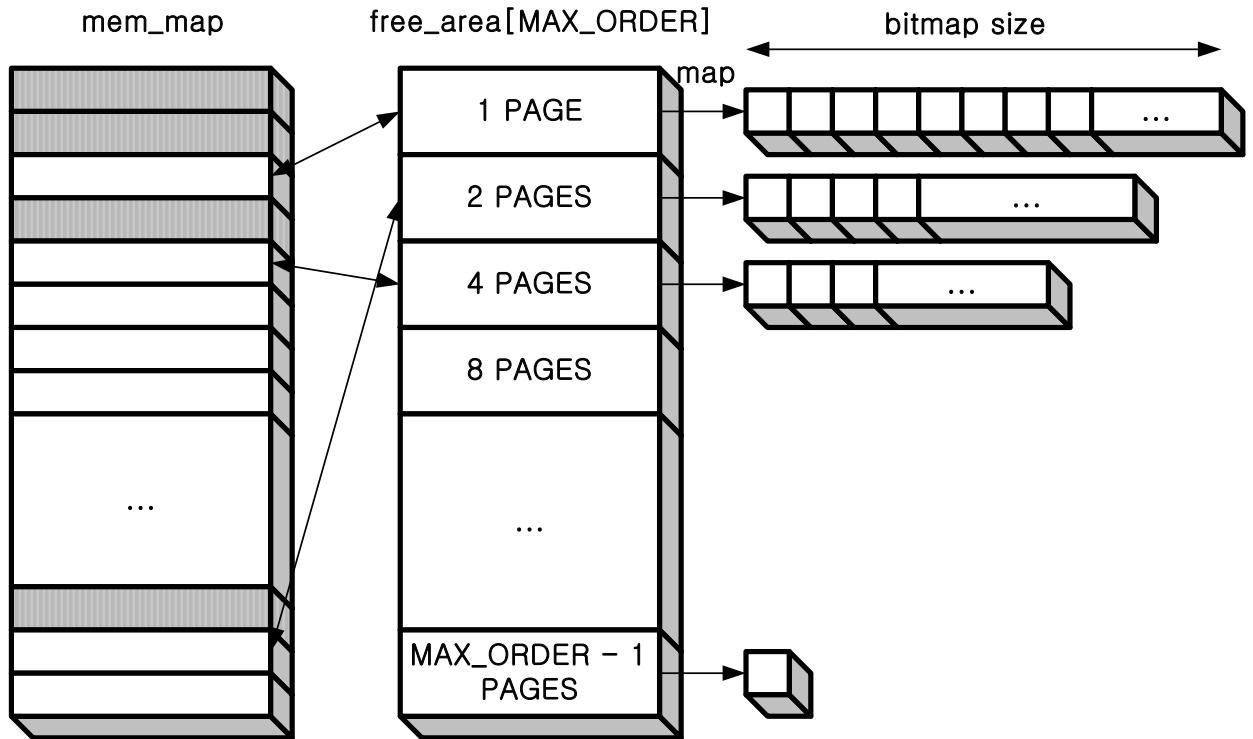


그림 41. 페이지 할당을 위한 커널 데이터 구조의 사용 예

[그림34]는 현재 메모리가 사용 중일 때의 커널 자료구조를 보여준다. MAX_ORDER는 10을 가지기에 free_area[] 배열은 10개의 엔트리를 가진다.

따라서, free_area_init_core()함수는 전체 사용 가능한 메모리를 mem_map구조체로 표현하기 위한 일을 해주는데, mem_map구조체를 보면 하나의 페이지를 나타내는 page구조체의 배열로 이루어진다. 다시 페이지 구조체의 한 필드는 zone_struct를 가지게 되며, 이곳에서 free_area_t으로 정의된 free_area 필드를 초기화 해주며, free_area는 2의 제곱승 갯수 만큼의 페이지를 가지는 연속된 페이지들에 리스트를 관리해 준다. 즉, free_area 배열의 인덱스 값이 바로 2의 제곱승 수⁸³가 된다.

이전 free_area_init_core()함수를 볼 준비가 되었다. 앞에서 paging_init()함수에서 free_area_init()함수를 zones_size[]배열을 인자로 넘겨주어서 호출하고, 다시 free_area_init()함수가 free_area_init_core() 함수를 불러준다. 넘겨주는 것은 contig_page_data와 mem_map의 주소, 그리고, paging_init()함수에서 생성된 zones_size[] 배열이다.

```
void __init free_area_init_core(int nid, pg_data_t *pgdat, struct page **gmap,
                                unsigned long *zones_size, unsigned long zone_start_paddr,
                                unsigned long *zholes_size, struct page *lmem_map)
{
    struct page *p;
    unsigned long i, j;
    unsigned long map_size;
    unsigned long totalpages, offset, realtotalpages;
    unsigned int cumulative = 0;
```

⁸³ $2^0, 2^1, 2^2 \dots$ 의 순으로 인덱스 값이 0이면 1개의 free 페이지를 가진다는 것을 나타내며, 인덱스 값이 1이면 2개의 연속된 free 페이지를 가진다는 것을 나타낸다. 따라서, 인덱스 2는 4개의 연속된 free 페이지가 될 것이다.

```

totalpages = 0;
for (i = 0; i < MAX_NR_ZONES; i++) {
    unsigned long size = zones_size[i];
    totalpages += size;
}
realtotalpages = totalpages;
if (zholes_size)
    for (i = 0; i < MAX_NR_ZONES; i++)
        realtotalpages -= zholes_size[i];
printf("On node %d totalpages: %lu\n", nid, realtotalpages);
memlist_init(&active_list);
memlist_init(&inactive_dirty_list);

```

코드 263. free_area_init_core() 함수

free_area_init_core() 함수가 하는 일은 메모리 zone⁸⁴의 데이터 구조를 만드는 일을 하는 것으로 크게 3가지의 일을 한다. 먼저 모든 페이지가 예약 되었다고 표시하고, 모든 메모리의 큐들을 비운다. 그리고, 나서 메모리의 구성 정보를 나타내는 bitmap⁸⁵을 지운다.

현재 넘겨받은 변수들은 nid는 0, pgdat는 config_page_data의 주소를, gmap는 mem_map의 주소를, zones_size는 zones_size[]의 주소를 가지며, 나머지 zone_start_paddr, zholes_size, lmem_map 등은 전부 0이나 NULL의 값을 가지고 있다.

먼저 전체 메모리의 사용 가능한 페이지의 갯수를 구한다(totalpages). 실제적인 전체 사용가능 메모리는 realtotalpages가 나타내도록 한다. zholes_size는 NULL을 가지므로 새로이 realtotalpages를 계산할 필요가 없으며, active_list 및 inactive_dirty_list를 초기화 한다(memlist_init()).

```

map_size = (totalpages + 1)*sizeof(struct page);
if (lmem_map == (struct page *)0) {
    lmem_map = (struct page *) alloc_bootmem_node(pgdat, map_size);
    lmem_map = (struct page *) (PAGE_OFFSET +
        MAP_ALIGN((unsigned long)lmem_map - PAGE_OFFSET));
}
*gmap = pgdat->node_mem_map = lmem_map;
pgdat->node_size = totalpages;
pgdat->node_start_paddr = zone_start_paddr;
pgdat->node_start_mapnr = (lmem_map - mem_map);
for (p = lmem_map; p < lmem_map + totalpages; p++) {
    set_page_count(p, 0);
    SetPageReserved(p);
    init_waitqueue_head(&p->wait);
    memlist_init(&p->list);
}

```

코드 264. free_area_init_core() 함수 – continued

이젠 실제 사용 가능한 메모리들의 메모리 맵(map) 구현을 위해서 필요한 공간을 구한다(map_size). lmem_map은 현재 0값을 가지고 있으므로, 전체 메모리의 맵 크기를 저장할 메모리를 할당받아 정렬한 다음 그 값을 lmem_map에 넣어준다. gmap은 현재 mem_map의 주소를 가지고 있으므로, 이것을 새로이 할당 받은 lmem_map으로 설정한다. 이젠 연속된 메모리에 대한 데이터를 가지는 contig_page_data에 이 정보를 기록한다. 즉, node의 mem_map주소와 node_size, node_start_paddr, node_start_mapnr 필드들을 업데이트 한다.

⁸⁴ 여기서 메모리 zone이라는 표현은 특별한 의미를 가지지 않는다. 다만 메모리의 한 구역 정도라고 보면 될 것이며, DMA, NORMAL, HIGHMEM으로 각각 나누어진다. 앞에서 말한 zone의 의미를 따른다.

⁸⁵ free_area 구조체의 bitmap이다.

이전 mem_map의 local한 값을 가지는 lmem_map를 따라가면서, 전체 map에 대한 설정을 한다. page 구조체가 이곳에서 사용된다. Count값을 0으로 두고, 페이지 reserve bit를 설정한 후, page 구조체의 wait 큐를 초기화하고(init_waitqueue_head()), page 구조체의 리스트를 초기화 한다.

```

offset = lmem_map - mem_map;
for (j = 0; j < MAX_NR_ZONES; j++) {
    zone_t *zone = pgdat->node_zones + j;
    unsigned long mask;
    unsigned long size, realsize;

    realsize = size = zones_size[j];
    if (zholes_size)
        realsize -= zholes_size[j];
    printk("zone(%lu): %lu pages.\n", j, size);
    zone->size = size;
    zone->name = zone_names[j];
    zone->lock = SPIN_LOCK_UNLOCKED;
    zone->zone_pgdat = pgdat;
    zone->free_pages = 0;
    zone->inactive_clean_pages = 0;
    zone->inactive_dirty_pages = 0;
    memlist_init(&zone->inactive_clean_list);
    if (!size)
        continue;
    zone->offset = offset;
    cumulative += size;
    mask = (realsize / zone_balance_ratio[j]);
    if (mask < zone_balance_min[j])
        mask = zone_balance_min[j];
    else if (mask > zone_balance_max[j])
        mask = zone_balance_max[j];
    zone->pages_min = mask;
    zone->pages_low = mask*2;
    zone->pages_high = mask*3;
}

```

코드 265. free_area_init_core()함수 – continued

아직은 lmem_map과 mem_map가 같은 값을 가지므로 offset은 0이다. 이전 for 루프를 돌면서 contig_page_data에 있는 자료 구조체를 초기화 한다⁸⁶. 먼저 지역적으로 쓸 zone에 대한 포인터를 하나 가져온다(zone). zones_size[] 배열에서 zone의 실제 크기를 가져와서 realsize와 size 지역 변수를 초기화 한다. 만약 zone내에 구멍(hole)⁸⁷이 있으면 zone의 크기에서 해당하는 구멍의 크기를 뺀 값으로 realsize를 초기화 한다. 이전 각각의 zone에 대한 초기화를 수행한다. 만약 size가 0을 가진다면, 해당 zone에 페이지가 없다는 뜻이되므로, loop를 다시 수행시킨다. 또한 zone의 offset값은 offset값을 가지도록 하며, cumulative에는 size값으로 더 해준다. zone_balance_ratio[]와 zone_balance_min[], zone_balance_max[], zone_names[]는 각각 아래와 같은 값을 가진다.

```

static char *zone_names[MAX_NR_ZONES] = { "DMA", "Normal", "HighMem" };
static int zone_balance_ratio[MAX_NR_ZONES] = { 32, 128, 128, };
static int zone_balance_min[MAX_NR_ZONES] = { 10, 10, 10, };
static int zone_balance_max[MAX_NR_ZONES] = { 255, 255, 255, };

```

이 값을 바탕으로 zone에서 유지할 페이지 갯수에 대한 하한 값(min), 낮은 값(low), 높은 값(high)에 대해서 각각 realsize/zone_balance_ratio[] 결과 및 2배와 3배 값으로 설정한다. 이러한 값들은 나중에

⁸⁶ pgdat가 contig_page_data구조체를 가지고 있다.

⁸⁷ BIOS등에 의해서 사용되는 영역이 있거나, 빈 공간이 존재하면 이를 hole이라고 한다.

페이지에 대한 할당 및 해제를 위한 water-mark 역할을 한다. 즉, 어떤 선 이상이나 혹은 이하로 free 페이지의 수가 떨어지면, 커널에서 적절한 연산을 수행할 것이다.

```

freepages.min += mask;
freepages.low += mask*2;
freepages.high += mask*3;
zone->zone_mem_map = mem_map + offset;
zone->zone_start_mapnr = offset;
zone->zone_start_paddr = zone_start_paddr;
for (i = 0; i < size; i++) {
    struct page *page = mem_map + offset + i;
    page->zone = zone;
    if (j != ZONE_HIGHMEM) {
        page->virtual = __va(zone_start_paddr);
        zone_start_paddr += PAGE_SIZE;
    }
}
offset += size;
mask = -1;
for (i = 0; i < MAX_ORDER; i++) {
    unsigned long bitmap_size;

    memlist_init(&zone->free_area[i].free_list);
    mask += mask; /* mask 값을 2배로 만든다.*/
    size = (size + ~mask) & mask; /* 2의 승수로 나누어 떨어지도록 만든다 */
    /* 하위의 값은 mask값으로 masking시킨다.*/
    bitmap_size = size >> i; /* loop의 index 제곱승으로 나눈다.*/
    bitmap_size = (bitmap_size + 7) >> 3; /* byte단위로 크기를 만든다.*/
    bitmap_size = LONG_ALIGN(bitmap_size); /* 4 bytes 단위로 다시 정렬한다.*/
    zone->free_area[i].map = /* bitmap을 할당한다.*/
        (unsigned int *) alloc_bootmem_node(pgdat, bitmap_size);
}
build_zonelists(pgdat);
}

```

코드 266. free_area_init_core()함수 – continued

이전 시스템내에 있는 전체 free 페이지들에 대한 정보를 주는 freepage⁸⁸를 갱신한다. 이전 zone의 zone_mem_map, zone_start_mapnr, zone_start_paddr 값을 초기화 시킨다. 이전 mem_map의 zone 필드를 갱신할 차례이다. 앞에서 초기화를 마친 zone의 주소 값을 넣도록 하며, high memory에 대한 zone이 아니라면, mem_map한 엔트리(page구조체)의 가상주소(page->virtual)에는 zone의 시작번지(zone_start_paddr)의 물리적인 주소를 가상주소로 변환(__va())해서 넣도록 한다. 나머지 zone들에 대한 것은 다시 PAGE_SIZE를 더한 만큼씩을 가상주소로 변환해서 넣어주면 될 것이다.

이전 offset을 size만큼 증가 시켜서 다음 번의 loop수행을 대비한다. zone에 대한 초기화는 거의 다 끝났으나 아직 zone이 가지는 free_area[]배열과 free_area[]의 map필드에 대한 초기화가 아직 이루어 지지 않았다. for loop를 돌면서 zone의 free_area[]배열에 대해서 설정해준다. 여기서 MAX_ORDER⁸⁹는 10이란 값을 가지므로, 전체 free_area[]배열은 10개의 엔트리를 가진다.

⁸⁸ 정의는 ~/mm/swap.c에 나와 있으며, 페이지의 할당 및 slab allocator에서 사용된다.

⁸⁹ 정의는 ~/include/linux/mmzone.h에 있다.

남은 것은 각 free_area[] 배열에 메모리 맵의 사용을 나타내기 위해서 bitmap을 할당하는 과정이다. bitmap의 하나의 bit는 두개의 연속된 페이지에 대해서 사용되는데, 인접한 페이지들이 모두 사용중이거나, 혹은 사용중이 아니라면 0을 가진다. 하나씩만 사용중이라면 1을 bit에 설정할 것이다.⁹⁰ 예를 들어 free_area[0]의 bitmap은 zone에 있는 2개의 연속된 페이지 프레임에 대해서 하나의 bit으로 나타난다. free_area[1]은 따라서 4개의 연속된 페이지 프레임에 대해 하나의 bit으로 상태를 표시할 것이다. build_zonelists()함수는 이곳에서 만들어진 정보로 zone 리스트를 갱신하는 역할을 한다. 최종적으로 이 함수를 마치게 되면, [그림34]와 같이 될 것이다.

~/init/main.c를 다시 보면 이전 메모리에 대해서 초기화 하는 함수가 2개가 더 있음을 볼 수 있다. 즉, kmem_cache_init()와 mem_init()함수이다. kmem_cache_init()함수는 slab allocator를 초기화 하는 함수이며, mem_init()는 바로 앞에서 할당받은 메모리에 대한 초기화를 수행한다. kmem_cache_init()함수는 나중에 slab allocator에서 보도록 하고, mem_init()부터 보자.

```
void __init mem_init(void)
{
    int codesize, reservedpages, datasize, initsize;
    int tmp;

    if (!mem_map)
        BUG();
#define CONFIG_HIGHMEM
    highmem_start_page = mem_map + highstart_pfn;
    max_mapnr = num_physpages = highend_pfn;
#else
    max_mapnr = num_physpages = max_low_pfn;
#endif
    high_memory = (void *) __va(max_low_pfn * PAGE_SIZE);
    /* clear the zero-page */
    memset(empty_zero_page, 0, PAGE_SIZE);
    /* this will put all low memory onto the freelists */
    totalram_pages += free_all_bootmem();
```

코드 267. mem_init()함수

codesize는 커널에서 사용할 코드 크기를 가진다. reservedpages는 BIOS나 커널, 하드웨어에 의해해서 예약된 영역을 가르키며, datasize는 커널의 데이터 크기를 가진다. initsize는 __init_begin과 __init_end사이의 크기를 가진다. 만약 mem_map이 정해지지 않았다면 에러가된다. 이전에 free_area_init()에서 이미 할당되었으니, 그냥 지나갈 것이다.

CONFIG_HIGHMEM이 설정되었다면, mem_map에 high memory의 시작 PFN을 더해서 highmem_start_page를 설정한다. 맵의 최대 크기를 가르키는 max_mapnr과 물리적인 페이지를 수를 나타내는 num_physpages는 high memory의 PFN값으로 설정한다. CONFIG_HIGHMEM설정이 없다면, max_mapnr과 num_physpages는 max_low_pfn이 될 것이다.

high_memory의 값을 max_low_pfn x PAGE_SIZE를 곱한 가상 주소로 가르키도록 한다. 또한 empty_zero_page는 0으로 PAGE_SIZE만큼을 초기화한다(memset()). empty_zero_page는 ~/arch/i386/kernel/head.S에서 0x4000 + 0xC0000000에 위치하도록 정해져 있다. 전체 사용 가능한 메모리는 부팅에서 필요한 메모리를 해제(free_all_bootmem())한 영역도 포함하도록 한다.

```
reservedpages = 0;
for (tmp = 0; tmp < max_low_pfn; tmp++)
    if (page_is_ram(tmp) && PageReserved(mem_map+tmp))
        reservedpages++;
#define CONFIG_HIGHMEM
```

⁹⁰ 하지만, 이곳에서 해준 것과 같은 연산을 작은 프로그램으로 짜서, 돌려보았는데, 결과는 예측을 벗어났다. 좀더 생각을 정리해야 할 것으로 나중에 다시 보도록 하겠다.

```

for (tmp = highstart_pfn; tmp < highend_pfn; tmp++) {
    struct page *page = mem_map + tmp;
    if (!page_is_ram(tmp)) {
        SetPageReserved(page);
        continue;
    }
    ClearPageReserved(page);
    set_bit(PG_hightmem, &page->flags);
    atomic_set(&page->count, 1);
    __free_page(page);
    totalhigh_pages++;
}
totalram_pages += totalhigh_pages;
#endif
codesize = (unsigned long) &_etext - (unsigned long) &_text;
datasize = (unsigned long) &_edata - (unsigned long) &_etext;
initsize = (unsigned long) &__init_end - (unsigned long) &__init_begin;
printk("Memory: %luk/%luk available (%dk kernel code, %dk reserved, %dk data, %dk init, %ldk
highmem)\n",
(unsigned long) nr_free_pages() << (PAGE_SHIFT-10),
max_mapnr << (PAGE_SHIFT-10),
codesize >> 10,
reservedpages << (PAGE_SHIFT-10),
datasize >> 10,
initsize >> 10,
(unsigned long) (totalhigh_pages << (PAGE_SHIFT-10))
);

```

코드 268. mem_init()함수 – continued

이전 예약된 영역의 페이지가 얼마나 차지 하는지 계산한다. 먼저 resesrvdpages에 0을 두고, max_low_pfn까지 페이지가 RAM에 있고(page_is_ram()), 예약되어 있는지를 확인(PageReserved())하면서 reservedpages를 증가 시켜준다.

만약 CONFIG_HIGHMEM이 설정되었다면, highstart_pfn에서 highend_pfn까지를 살펴본다. 만약 페이지가 RAM에 있지 않다면, 페이지를 예약하고(SetPageReserved()) 다음번의 for loop로 진행한다. 페이지가 RAM영역에 있다면, 페이지의 예약을 풀고(ClearPageReserved()), 페이지의 flags 필드에 PG_hightmem을 설정한 후, 사용 카운트를 1증가 시킨다(atomic_set()), 해당 페이지를 사용할 수 있도록 해제해주고(__free_page()), 전체 페이지(total_pages) 값을 증가 시킨다. loop를 마치고 나오면 total_pages값을 전체 RAM상에 있는 페이지 값에 더해준다(totalram_pages). 이전 codesize와 datasize, initsize를 초기화 한 후, 이 값을 프린트 한다(prink()).

```

#endif
if CONFIG_X86_PAE
    if (!cpu_has_pae)
        panic("cannot execute a PAE-enabled kernel on a PAE-less CPU!");
#endif
    if (boot_cpu_data.wp_works_ok < 0)
        test_wp_bit();
#ifndef CONFIG_SMP
    zap_low_mappings();
#endif
}

```

코드 269. mem_init()함수 – continued

만약 CONFIG_X86_PAE가 설정되었다면, CPU가 PAE를 지원하는지 확인한다. 만약 지원하지 않는 CPU라면 에러를 표시한다. `test_wp_bit()`⁹¹함수는 커널에서 WP(Write Protection)이 지원되는지를 확인하는 함수이다. 386에서는 지원되지 않으며, 특정 486에서도 지원되지 않는다. 586에서부터는 지원되고 있는 기능하다. `zap_low_mappings()` 함수는 하위 메모리 영역에 대한 해제를 수행한다.

4.8. 페이지의 할당과 해제

페이지를 할당하는 함수는 `~/mm/page_alloc.c`에 정의된 `_get_free_pages()`함수와 `get_zeroed_page()`함수이다. 나머지 할당 함수들은 이것을 이용해서 페이지 단위의 할당을 수행한다. 두 함수의 정의는 아래와 같다.

```
unsigned long __get_free_pages(int gfp_mask, unsigned long order)
{
    struct page * page;

    page = alloc_pages(gfp_mask, order);
    if (!page)
        return 0;
    return (unsigned long) page_address(page);
}

unsigned long get_zeroed_page(int gfp_mask)
{
    struct page * page;

    page = alloc_pages(gfp_mask, 0);
    if (page) {
        void *address = page_address(page);
        clear_page(address);
        return (unsigned long) address;
    }
    return 0;
}
```

코드 270. 페이지 할당 함수들

두 함수의 차이는 0으로 초기화된 페이지 영역을 할당 받는가 그렇지 못한가이며, 둘다 `alloc_pages()`함수를 호출한다. `get_zeroed_page()`함수의 경우에는 `clear_page()`⁹² 함수를 호출해서 페이지 영역에 있는 이전의 데이터를 0으로 지운다. 따라서, `alloc_pages()`함수가 페이지 할당의 핵심이다. `alloc_pages()`함수는 `~/include/linux/mm.h`에 정의되어 있으며, `order`가 유효한 값인지를 확인한 후, `_alloc_pages()`함수를 `gfp_mask`와 `order`값을 넘겨 주어서 호출한다. `Order`는 페이지의 수를 2의 승수 값으로 나타낸 것이며, `gfp_mask`로 들어올 수 있는 값은 아래와 같은 것이다.

GFP_MASK	값	설명
<code>__GFP_WAIT</code>	0x01	요구를 만족 시키기 전에 메모리를 해제(free)하기 위해서 커널이 페이지의 내용들을 버리는 것을 허락한다면 설정한다. 즉, 커널에서 적절한 조치를 취해주어 페이지를 할당 할 때까지 기다릴 수 있다는 것을 말한다.

⁹¹ Write Protection은 읽기는 가능하면 cache line에 읽어오게 되며, 만약 읽기가 miss가 되면, 자동적으로 cache line이 새로이 채워진다. 쓰기는 전체 시스템 버스로 전파되며, 버스 상의 모든 프로세스에 있는 cache line들이 무효화(invalidation) 된다. 이러한 캐ши 옵션은 P6 패밀리에서는 모두 사용가능하며, MTRR(Memory Type Range Register) 레지스터들을 프로그램해서 사용한다.

⁹² `~/include/asm/mm.h`에 정의되어 있으며, CPU가 3DNOW 기능을 지원하는 경우에는 그것을 이용하도록 하고, 그렇지 않으면, 그냥 `memset()`함수로 초기화 한다.

__GFP_HIGH	0x02	요구의 우선 순위를 명시한다. 커널에 메모리를 할당할 경우 사용된다.
__GFP_IO	0x04	해당하는 페이지 프레임들을 해제하기 위해서 커널이 페이지들을 디스크에 쓰는 것이 허락될 때 설정한다. 예를 들어서 swapping의 경우 커널내에서 프로세스가 블록킹이 될 수 있으며, 따라서, 커널의 중요한 데이터를 고치는 곳이나, 혹은 인터럽트의 처리중일때는 반드시 이 bit를 지워(clear)야 할 것이다.
__GFP_DMA	0x08	DMA를 수행할 수 있는 페이지를 요구한 경우 설정한다. 즉, ISA 디바이스들은 하위의 16Mbytes 영역에서만 시스템의 DMA channel을 이용해서 DMA를 수행할 수 있기 때문에, 이 bit를 설정해서 페이지를 요구한다.
__GFP_HIGHMEM	0x10 or 0x00	High memory에 대한 페이지를 요구할 때 사용한다.
GFP_BUFFER	__GFP_HIGH __GFP_WAIT	Buffer를 위한 메모리 할당을 요구할 때 사용된다. 위에서 정의한 __GFP_HIGH와 __GFP_WAIT를 설정한다.
GFP_ATOMIC	__GFP_HIGH	커널에서 사용할 메모리 영역의 할당을 사용한다. __GFP_HIGH와 동일한다.
GFP_USER	__GFP_WAIT __GFP_IO	사용자 영역을 위한 페이지 할당을 수행한다. __GFP_WAIT와 __GFP_IO가 설정되어, free 페이지가 적을 때 페이지를 디스크에 저장하고, 해당 페이지를 해제 하도록 기다릴 수 있다.
GFP_HIGHUSER	__GFP_WAIT __GFP_IO __GFP_HIGHMEM	GFP_USER와 같지만 상위 메모리 영역에 대한 할당을 기대한다.
GFP_KERNEL	__GFP_HIGH __GFP_WAIT __GFP_IO	커널에서 사용할 메모리를 요구하는 것으로 __GFP_HIGH와 __GFP_WAIT, __GFP_IO를 설정한다.
GFP_NFS	__GFP_HIGH __GFP_WAIT __GFP_IO	NFS(Network File System)을 위한 페이지를 할당 받는다.
GFP_KSWAPD	__GFP_IO	Swapping에서 사용할 페이지 메모리를 할당 받는다.
GFP_DMA	__GFP_DMA	DMA를 위해서 사용할 페이지 메모리를 할당 받는다.
GFP_HIGHMEM	__GFP_HIGHMEM	High memory에 있는 페이지 메모리를 할당 받는다.

표 32. 페이지 할당을 위한 mask값

위에서 정의한 GFP mask값은 페이지의 할당과 해제시 해당하는 메모리 영역을 명시하기 위해서 사용한다. 아래의 __alloc_pages()함수를 보도록 하자.

```
struct page * __alloc_pages(zonelist_t *zonelist, unsigned long order)
{
    zone_t **zone;
    int direct_reclaim = 0;
    unsigned int gfp_mask = zonelist->gfp_mask;
    struct page * page;

    memory_pressure++;
    if (order == 0 && (gfp_mask & __GFP_WAIT) &&
        !(current->flags & PF_MEMALLOC))
        direct_reclaim = 1;
    if (inactive_shortage() > inactive_target / 2 && free_shortage())
        wakeup_kswapd(0);
    else if (free_shortage() && nr_inactive_dirty_pages > free_shortage()
             && nr_inactive_dirty_pages >= freepages.high)
        wakeup_bdfflush(0);
```

코드 271. __alloc_pages()

zone은 해당 메모리 영역을 나타내는 포인터이다. gfp_mask는 해당 메모리 zone의 gfp_mask를 가진다. 메모리를 할당하는 연산이므로 메모리에 대해서 시스템이 얼마나 많은 페이지를 사용하고 있는지를 나타내는 memory_pressure⁹³변수를 증가 시킨다. 나중에 페이지가 해제되면 이 값은 감소할 것이다. 만약 order값이 0이고(1개의 페이지),gfp_mask가 __GFP_WAIT이며, 현재 프로세스의 flags에 PF_MEMALLOC이 설정되지 않았다면⁹⁴, direct_reclaim을 1로 설정한다.

만약 현재 inactive 한 페이지가 inactive_target/2보다 크고, free 페이지가 적다면(free_shortage()), kswapd 데몬 프로세스를 깨운다. 즉, swapping을 시작하도록 한다. 만약 free 페이지가 적고, 현재 inactive dirty 페이지 수가 현재 부족을 격고 있는 페이지의 수보다 많으며, freepages의 최대값보다 같거나 큰 값을 가진다면 bdfflush 데몬 프로세스를 깨워서 dirty 상태에 있는 페이지를 디스크에 쓰고, 시스템에 있는 free 페이지의 수를 늘린다.

```
try_again:
    zone = zonelist->zones;
    for (;;) {
        zone_t *z = *(zone++);
        if (!z)
            break;
        if (!z->size)
            BUG();
        if (z->free_pages >= z->pages_low) {
            page = rmqueue(z, order);
            if (page)
                return page;
        } else if (z->free_pages < z->pages_min &&
                   waitqueue_active(&kreclaimd_wait)) {
            wake_up_interruptible(&kreclaimd_wait);
        }
    }
```

코드 272. __alloc_pages()함수 – continued

먼저 free 메모리를 많이 가지고 있는 zone이 있는지 확인한다. zone이 있다면, zone의 크기를 확인하고, zone의 free 페이지가 zone의 low값(pages_low)보다 크거나 같은 값을 가진다면, 이곳에서 order에 맞는 하나의 페이지를 제거(rmqueue())해서 돌려주고, 만약 그렇지 않다면, 다시 zone의 free페이지 수가 free 페이지의 하한선(pages_min)보다 작으며, kreclaimed_wait큐에 대기하고 있는 프로세스가 있는지 확인한다. 있다면 깨워준다(wake_up_interruptible()).

```
page = __alloc_pages_limit(zonelist, order, PAGES_HIGH, direct_reclaim);
if (page)
    return page;
page = __alloc_pages_limit(zonelist, order, PAGES_LOW, direct_reclaim);
if (page)
    return page;
wakeup_kswapd(0);
if (gfp_mask & __GFP_WAIT) {
    __set_current_state(TASK_RUNNING);
    current->policy |= SCHED_YIELD;
    schedule();
}
page = __alloc_pages_limit(zonelist, order, PAGES_MIN, direct_reclaim);
if (page)
```

⁹³ ~/mm/swap.c에 정의되어 있다.

⁹⁴ 이미 메모리 할당 요구가 프로세스에 대해서 존재한다.

```
return page;
```

코드 273. __alloc_pages()함수 – continued

zone으로 부터 free 페이지와 inactive clean 페이지를 합한 크기가 water mark 역할을 하는 zone의 pages_min, pages_low, pages_high값보다 클 때, 페이지를 할당하는 __alloc_pages_limit()함수를 호출한다. 넘겨 주는 값으로 들어가는 direct_reclaim은 앞의 연산 결과에 따라 설정되며, 사용된 페이지를 reclaim하지 아니면, zone에서 새로운 페이지를 할당받을지를 결정한다. water mark값으로 사용될 것을 명시하는 것이 PAGE_MIN, PAGE_LOW, PAGE_HIGH이 하는 역할이다.

만약 위에서 할당의 여의치 않는다면, zone에 free 페이지가 작다는 말이된다. 따라서, kswapd데몬을 깨워서 일단 swapping을 실시한다. 만약 gfp_mask가 __GFP_WAIT로 명시되었다면, 현재 프로세스의 상태를 TASK_RUNNING으로 만들고, 프로세스의 스케줄링 정책(policy)을 SCED_YIELD로 설정한 후 reschedulig를 요구한다(schedule). 즉, kswapd 데몬 프로세스가 수행될 기회를 얻게 될 것이다.

마지막으로 다시 __alloc_pages_limit()함수를 PAGES_MIN을 인수로 전달해서 페이지를 할당 받도록 노력한다. 할당 받았다면 페이지를 돌려주고 그렇지 않다면 이하를 수행한다.

```
if (!(current->flags & PF_MEMALLOC)) {
    if (order > 0 && (gfp_mask & __GFP_WAIT)) {
        zone = zonelist->zones;
        /* First, clean some dirty pages. */
        current->flags |= PF_MEMALLOC;
        page_launder(gfp_mask, 1);
        current->flags &= ~PF_MEMALLOC;
        for (;;) {
            zone_t *z = *(zone++);
            if (!z)
                break;
            if (!z->size)
                continue;
            while (z->inactive_clean_pages) {
                struct page * page;
                /* Move one page to the free list. */
                page = reclaim_page(z);
                if (!page)
                    break;
                __free_page(page);
                /* Try if the allocation succeeds. */
                page = rmqueue(z, order);
                if (page)
                    return page;
            }
        }
    }
}
```

코드 274. __alloc_pages()함수 – continued

이전 현재 프로세스가 이미 메모리 할당을 요구 하지 않은 경우에 대한 처리를 해준다. 이미 zone에서 free 페이지를 할당하려는 노력을 했었다. 크게 다음과 같은 2가지의 이유에 때문에, 이러한 상황까지 오게된다. 즉, 첫번째로는 너무 큰 메모리를 할당하려고 하는 경우가 있을 수 있으며, 이것은 페이지 할당이 성공할 때까지 free 페이지들을 페이지의 free 리스트로 옮김으로 해결할 수 있다. 두번째로는 실제로 메모리의 사용이 많은 경우로 kwarpd 데몬이 free 페이지가 많이 생길까기 기다리도록 하고 있는, wait queue상에서 프로세스가 대기하도록 만든다.

Order가 0보다 크며, gfp_mask가 __GFP_WAIT로 명시된 경우에는, zone의 리스트에서 zone을 하나 구해온다(zonelist->zones). 먼저 프로세스가 메모리를 할당을 요구한다고 명시한다음(PF_MEMALLOC), dirty inactive 페이지를 지운다(page_launder()). 즉, inactive clean list로 페이지들을 보낸다. 이와 같은 연산이

끝나면, 이전 프로세스에 설정했던 PF_MEMALLOC flag를 해제하고, 무한 for loop를 돌면서 페이지를 할당받거나 zone이 없을 때까지 수행한다.

zone리스트에서 다시 다음 zone을 가져와서 유효한지 검사하고, 다시 크기가 0이 아닌지 확인한다. zone의 inactive clean 페이지들에 대해서 reclaim_page()함수를 호출해서 하나의 페이지를 inactive clean에서 새로이 할당받아, __free_page()함수로 넘겨준다. 그리고나서 다시 zone으로부터 페이지를 하나 제거(rmqueue())해서 이를 함수의 복귀 값으로 돌려준다.

```

if ((gfp_mask & (__GFP_WAIT|__GFP_IO)) == (__GFP_WAIT|__GFP_IO)) {
    wakeup_kswapd(1);
    memory_pressure++;
    if (!order)
        goto try_again;
} else if (gfp_mask & __GFP_WAIT) {
    try_to_free_pages(gfp_mask);
    memory_pressure++;
    if (!order)
        goto try_again;
}
zone = zonelist->zones;
for (;;) {
    zone_t *z = *(zone++);
    struct page * page = NULL;
    if (!z)
        break;
    if (!z->size)
        BUG();
    if (direct_reclaim) {
        page = reclaim_page(z);
        if (page)
            return page;
    }
    if (z->free_pages < z->pages_min / 4 &&
        !(current->flags & PF_MEMALLOC))
        continue;
    page = rmqueue(z, order);
    if (page)
        return page;
}
/* No luck.. */
printk(KERN_ERR "__alloc_pages: %lu-order allocation failed.\n", order);
return NULL;
}

```

코드 275. __alloc_pages()함수 – continued

gfp_mask값이 GFP_WAIT와 GFP_IO 둘다가 설정되었다면, kswapd를 깨우고, memory_presure를 증가시킨 후, order값을 확인해서 0이라면 try_again으로 제어를 옮겨서 새롭게 페이지 할당을 받도록 해준다. 만약 __GFP_WAIT만 설정되었다면, gfp_mask로 설정된 페이지들을 해제(free)하려고 한다(try_to_free_pages()). 역시 memory_pressure는 증가시켜주며, order값이 0이라면 try_again으로 이동해서 free페이지의 할당을 새로 시작한다.

새로이 zone리스트에서 다음 무한 for loop를 실행하고, zone의 free페이지수가 zone의 pages_min/4보다 작은 값을 가지며, 현재 프로세스의 플랙이 PF_MEMALLOC이 설정되지 않았다면, loop를 다시 돈다.

그렇지 않다면, 해당 zone을 찾은 것이 되며, 이곳에서 order에 해당하는 페이지를 가져온다. 전체 zone의 리스트에서 알맞은 page를 찾지 못한다면, 페이지 할당이 실패로 돌아가며, 복귀값은 NULL을 가진다.⁹⁵

좀 더 하위의 페이지 할당은 rmqueue()와 reclaim_page()함수가 맡고 있다. 각각에 대해서 보기로 하자. 먼저 rmqueue()함수는 아래와 같이 정의된다.

```
static FASTCALL(struct page * rmqueue(zone_t *zone, unsigned long order));
static struct page * rmqueue(zone_t *zone, unsigned long order)
{
    free_area_t * area = zone->free_area + order;
    unsigned long curr_order = order;
    struct list_head *head, *curr;
    unsigned long flags;
    struct page *page;

    spin_lock_irqsave(&zone->lock, flags);
    do {
        head = &area->free_list;
        curr = memlist_next(head);
        if (curr != head) {
            unsigned int index;

            page = memlist_entry(curr, struct page, list);
            if (BAD_RANGE(zone,page))
                BUG();
            memlist_del(curr);
            index = (page - mem_map) - zone->offset;
            MARK_USED(index, curr_order, area);
            zone->free_pages -= 1 << order;
            page = expand(zone, page, index, order, curr_order, area);
            spin_unlock_irqrestore(&zone->lock, flags);
            set_page_count(page, 1);
            if (BAD_RANGE(zone,page))
                BUG();
            DEBUG_ADD_PAGE
            return page;
        }
        curr_order++;
        area++;
    } while (curr_order < MAX_ORDER);
    spin_unlock_irqrestore(&zone->lock, flags);
    return NULL;
}
```

코드 276. rmqueue()함수

넘겨받은 zone에서 free_area[]에 order값을 더한 곳의 주소를 area에 넣는다. 현재의 order를 curr_order로 놓고, lock을 설정한(spin_lock_irqsave())에 do{} while loop를 수행한다. 이것은 커널에 critical 한 작업이므로 반드시 lock을 설정해야 한다.

먼저 free area의 리스트를 가져와서 head로 놓는다. 그리고, 다음에 있는 리스트의 엔트리를 curr로 놓는다. 현재 엔트리에서 페이지를 하나 가져온후 이것이 올바른 값을 가지는지 확인한다(BAD_RANGE()). 만약 올다면 curr을 리스트에서 지우고, index는 페이지 구조체에서 mem_map와

⁹⁵ 아직 이해가 완전하지 못해서 미진한 면이 없지 않으마, zone별로 페이지를 할당하려는 노력을 하고 있다는 것은 이해할 수 있을 것이다. 즉, 전체 메모리가 gfp_mask값에 따라 해당존에서 필요한 만큼의 페이지들을 할당하려고 한다.

zone의 offset값을 더한 값으로 두고, 페이지가 사용 중이라고 놓는다(MARK_USER()). 이전 zone의 free page들의 카운터를 order만큼 1을 좌측으로 shift한 값을 빼서, 사용에 들어간 페이지 수를 고려해준다. expand()함수는 order에서 curr_order까지 있는 free 페이지들의 리스트를 갱신 시켜준다. 즉, 상위에서 해당 페이지를 하나 사용하게 되므로 이하에 있는 free 페이지들은 할당된 페이지들에 대해서 사용여부를 가지는 bitmap이나 free 페이지의 list를 가지기 때문에 갱신 시켜주어야 할 것이다. 이전 lock을 해제하고, 페이지에 대한 사용카운터를 증가 시켜준 후, 할당한 페이지를 돌려준다. 만약 최대 order까지 검색을 해서 해당 페이지를 찾을 수 없다면 lock을 해제하고 NULL을 돌려준다.

```
struct page * reclaim_page(zone_t * zone)
{
    struct page * page = NULL;
    struct list_head * page_lru;
    int maxscan;

    spin_lock(&pagecache_lock);
    spin_lock(&pagemap_lru_lock);
    maxscan = zone->inactive_clean_pages;
```

코드 277. reclaim_page()함수

reclaim_page()함수는 inactive clean list에 존재하는 하나의 페이지를 재사용을 요구하는 함수이다. 넘겨받는 값은 재사용(reclaim)을 수행할 zone이 된다. Inactive clean 리스트에 있는 페이지들은 쉽게 재사용이 될 수 있는 페이지들이다. 먼저, 페이지를 얻는 연산에서 발생할 수 있는 dead lock⁹⁶을 피할수 있도록 페이지 캐쉬에 대한 lock과 페이지 맵의 lru를 위한 lock을 함께 얻는다. 그리고, 최대 페이지를 검색할 수 있는 값(maxscan)으로는 zone->inactive_clean_pages(zone의 inactive clean list에 있는 페이지들의 수)를 둔다.

```
while ((page_lru = zone->inactive_clean_list.prev) !=
       &zone->inactive_clean_list && maxscan--) {
    page = list_entry(page_lru, struct page, lru);
    if (!PageInactiveClean(page)) {
        printk("VM: reclaim_page, wrong page on list.\n");
        list_del(page_lru);
        page->zone->inactive_clean_pages--;
        continue;
    }
    /* Page is or was in use? Move it to the active list.*/
    if (PageTestandClearReferenced(page) || page->age > 0 ||
        (!page->buffers && page_count(page) > 1)) {
        del_page_from_inactive_clean_list(page);
        add_page_to_active_list(page);
        continue;
    }
    /* The page is dirty, or locked, move to inactive_dirty list.*/
    if (page->buffers || PageDirty(page) || TryLockPage(page)) {
        del_page_from_inactive_clean_list(page);
        add_page_to_inactive_dirty_list(page);
        continue;
    }
    /* OK, remove the page from the caches.*/
    if (PageSwapCache(page)) {
        __delete_from_swap_cache(page);
        goto found_page;
```

⁹⁶ 둘 이상의 프로세스가 서로 자원을 점유한 상태에서 상대방의 자원을 요구하는 상황으로 시스템이 더 이상 수행할 수 없는 상태로 빠지는 것을 말한다.

```

        }
        if (page->mapping) {
            __remove_inode_page(page);
            goto found_page;
        }
        /* We should never ever get here. */
        printk(KERN_ERR "VM: reclaim_page, found unknown page\n");
        list_del(page_lru);
        zone->inactive_clean_pages--;
        UnlockPage(page);
    }
}

```

코드 278. reclaim_page()함수 – continued

이전 while loop를 돌면서 할당할 페이지를 찾는 부분이다. page_lru값에 zone의 inactive_clean_list의 prev값을 넣고, 이 값이 zone의 inactive_clean_list와 같지 않고, maxscan값이 0이 아닐때까지 수행한다. page_lru에서 하나의 페이지를 가져와서 이것을 page로 놓는다. 만약 페이지가 inactive clean상태가 아니라면 에러가되며, 해당 페이지를 list에서 제거한다(list_del()). 페이지의 zone에 inactive_clean_pages는 감소할 것이며, 다음 loop로 진행한다.

언급된(referenced) 페이지이거나, 혹은 page 구조체의 age가 0보다 크거나, 페이지 버퍼에 널을 가지고 있지만 count값이 1보다 큰 경우의 페이지에 대해서는 inactive_clean_list에서 제거해서 active_page로 넣어준다. 페이지가 buffer에 유효한 데이터를 가지고 있거나, 페이지가 dirty상태이일때는 lock을 설정할 수 있는지 확인하고, inactive_clean_list에서 페이지를 제거해서 inactive_dirty_list로 넣어준다.

페이지 스왑 캐시에 해당 페이지를 가지고 있다면(PageSwapCache()), 스왑 캐시에서 페이지를 제거(__delete_from_swap_cache())하고, 이를 새로 사용할 페이지로 알려준다. 제어는 found_page로 이동할 것이다.

만약 페이지가 메모리 mapping에서 사용되고 있다면, inode에서 관련된 페이지를 제거하고(__remove_inode_page()), 해당 페이지를 사용한다. 제어는 다시 found_page로 이동한다.

만약 아직도 해당하는 페이지를 찾지 못했다면, page_lru가 잘 못된 것인으로 이것을 리스트에서 제거하고(list_del()), zone의 inactive_clean_pages의 수를 감소시킨다. 페이지에 설정되었던 lock은 풀도록 한다(UnlockPage()).

```

/* Reset page pointer, maybe we encountered an unfreeable page.*/
page = NULL;
goto out;
found_page:
    del_page_from_inactive_clean_list(page);
    UnlockPage(page);
    page->age = PAGE_AGE_START;
    if (page_count(page) != 1)
        printk("VM: reclaim_page, found page with count %d!\n",
              page_count(page));
out:
    spin_unlock(&pagemap_lru_lock);
    spin_unlock(&pagecache_lock);
    memory_pressure++;
    return page;
}

```

코드 279. reclaim_page()함수 – continued

found_page로 제어가 옮겨 왔다면, 일단 해당 페이지를 찾은 것이다. 따라서, 할당할 페이지를 inactive_clean_list에서 제거한 후, 페이지에 걸린 lock을 해제하고, 페이지의 age(나이)에는 PAGE_AGE_START를 기록한 후, 페이지 count를 확인한다. 1이 아니라면 오류가 있다는 말이된다. 이젠

앞에서 설정했던 lock들(pagemap_lru_lock, pagecache_lock)을 풀고, memory_pressure값을 증가시킨 후, 찾은 페이지를 돌려준다.

메모리 페이지의 해제는 ~/mm/page_alloc.c에 선언된 __free_pages()나 free_pages()함수를 사용한다. 아래를 보도록 하자.

```
void __free_pages(struct page *page, unsigned long order)
{
    if (!PageReserved(page) && put_page_testzero(page))
        __free_pages_ok(page, order);
}

void free_pages(unsigned long addr, unsigned long order)
{
    struct page *fpage;

#ifndef CONFIG_DISCONTIGMEM
    if (addr == 0) return;
#endif
    fpage = virt_to_page(addr);
    if (VALID_PAGE(fpage))
        __free_pages(fpage, order);
}
```

코드 280. 페이지의 해제 함수들

free_pages() 함수는 가상 주소와 order를 넘겨받아서, 이를 물리적인 페이지로 고친(virt_to_page()) 후, 페이지가 유효한지를 따져서, 다시 __free_pages() 함수를 호출한다. __free_pages()함수는 페이지가 예약영역에 속하지 않고, 해당 페이지의 사용 카운트(count)를 감소시켜서 0을 가지는지 확인한다. 만약 그런 페이지를 해제한다고 하면, __free_pages_ok()함수를 호출해서 페이지를 해제시킨다.

__free_pages_ok()함수는 아래와 같이 정의된다. 위에서 본 alloc_pages()함수와 함께, 리눅스에서 메모리를 관리하는 알고리즘인 Buddy의 핵심을 이룬다.

```
static void FASTCALL(__free_pages_ok (struct page *page, unsigned long order));
static void __free_pages_ok (struct page *page, unsigned long order)
{
    unsigned long index, page_idx, mask, flags;
    free_area_t *area;
    struct page *base;
    zone_t *zone;

    if (page->buffers)
        BUG();
    if (page->mapping)
        BUG();
    if (!VALID_PAGE(page))
        BUG();
    if (PageSwapCache(page))
        BUG();
    if (PageLocked(page))
        BUG();
    if (PageDecrAfter(page))
        BUG();
    if (PageActive(page))
        BUG();
    if (PageInactiveDirty(page))
```

```

BUG();
if (PageInactiveClean(page))
    BUG();

```

코드 281. __free_pages_ok()함수

__free_pages_ok()함수는 free하려는 페이지의 page구조체의 포인터와 order를 넘겨받는다. 먼저 해제하려는 페이지가 올바른 형태인지를 확인한다. 관련된 buffer가 없어야 하며, 메모리 mapping도 하지 않고 있어야 하며, page구조체가 유효한 값을 가져야한다. 또한 swap cache로 사용되고 있지 않아야 하며, Lock설정도 없어야 하고, PageDecrAfter도 설정되지 않아야 할 것이며, active하지 않아야 한다. Inactive dirty리스트에도 속해있지 않아야 하고, inactive dirty clean 리스트에 속하지 않아야 한다.

```

page->flags &= ~((1<<PG_referenced) | (1<<PG_dirty));
page->age = PAGE_AGE_START;
zone = page->zone;
mask = (~0UL) << order;
base = mem_map + zone->offset;
page_idx = page - base;
if (page_idx & ~mask)
    BUG();
index = page_idx >> (1 + order);
area = zone->free_area + order;
spin_lock_irqsave(&zone->lock, flags);
zone->free_pages -= mask;
while (mask + (1 << (MAX_ORDER-1))) {
    struct page *buddy1, *buddy2;

    if (area >= zone->free_area + MAX_ORDER)
        BUG();
    if (!test_and_change_bit(index, area->map))
        break;
    buddy1 = base + (page_idx ^ -mask);
    buddy2 = base + page_idx;
    if (BAD_RANGE(zone, buddy1))
        BUG();
    if (BAD_RANGE(zone, buddy2))
        BUG();
    memlist_del(&buddy1->list);
    mask <= 1;
    area++;
    index >= 1;
    page_idx &= mask;
}
memlist_add_head(&(base + page_idx)->list, &area->free_list);
spin_unlock_irqrestore(&zone->lock, flags);
if (memory_pressure > NR_CPUS)
    memory_pressure--;
}

```

코드 282. __free_pages_ok()함수 – continued

page구조체의 flags에서 PG_reference와 PG_dirty bit을 지운다. 그리고, page구조체의 age에 PAGE_AGE_START($=2^{97}$)값을 넣는다. 그리고 page 구조체의 zone필드를 페이지가 속한 zone으로 설정한다(zone). ~0를 Order값으로 원쪽으로 shift한 값을 mask로 설정하며, base는 mem_map에서 zone의 offset을 더한 값으로 설정한다. page_idx는 이 페이지에서 앞에서 구한 offset을 빼도록 한다. 즉, 해당

⁹⁷ ~/include/linux/swap.h에서 정의

페이지가 속한 mem_map을 구하는 것이다. 만약 page_idx값이 ~mask와 AND를 해서 어떤 값이 있다면 에러가 된다. 즉, 적어도 해제할 페이지의 크기 이상의 값은 가지는 page_idx값을 찾는다. 그리고, index값은 order에 1을 더해서 page_idx를 $2^{order+1}$ 로 나눈 값이 들어간다. 이젠 zone에서 free_area[]에서 order에 해당하는 bitmap을 찾고, 이것을 area로 나타낸다. zone구조체의 free_pages는 mask값을 빼주어 해당하는 만큼의 free_pages들이 늘었음을 표시한다⁹⁸.

While loop는 mask값에 MAX_ORDER -1로 1을 왼쪽으로 shift한 값을 더해서 0이 아닌 동안 수행한다. 즉, MAX_ORDER가 100이므로 9만큼을 1을 왼쪽으로 shift하게 한 값에, mask를 더한다. 따라서, mask 값에 512를 더한 값이 while loop의 수행을 결정할 것이다. 먼저 두개의 buddy를 정의한다. 각각이 buddy1과 buddy2이다. area값이 zone의 free_area에 MAX_ORDER보다 크다면 bitmap을 벗어나게 되므로 error가 된다. test_and_change_bit()은 해당 index에 해당하는 bitmap을 바꾸고 이전에 있던 값을 돌려준다. 만약 돌려준 값이 0이면 loop를 빠져나온다. 즉, bitmap이 1로 설정되어 있었다면, 0이 되고, 1이 복귀 값이 될 것이다. 0으로 되어 있었다면, 1로 바뀌고 복귀값이 0이 될 것이다. 1로 이전에 설정되어 있었다고 하면, 아래부분이 실행될 것이다.

buddy1에는 page_idx에 ~mask값을 EXCLUSIVE OR를 시켜주어서 base를 더하고, buddy2에는 base에 page_idx값을 더한다. buddy1과 buddy2가 zone에 있는 올바른 값인지를 확인하고(BAD_RANGE()), buddy1을 list에서 지운다(memlist_del()). 다시 mask에 2를 곱하고(mask<<1), area도 증가 시켜주며, index값은 2로 나눈다(index>>1). 다음번에 해당하는 page_idx에는 mask에 AND를 시킨 값이 된다. Loop를 계속 수행하면, bitmap에 1로 설정된 buddy들이 검색이 되며, 최종적으로 0으로 이미 설정된 bitmap에서 수행을 마칠 것이다. 이 상태에서 free가 되는 페이지가 새로이 추가될 곳이 어딘지를 찾게 되며, free_list의 head에 추가된다(memlist_add_head()). memory_pressure값이 NR_CPUS(시스템에 있는 총 CPU의 갯수)보다 많아지면 memory_pressure를 감소시켜준다.

페이지의 할당과 해제는 Buddy 시스템이라고 불리는 알고리즘에 기초하고 있다. 이 알고리즘은 두개의 인접한 페이지 덩어리⁹⁹를 합쳐서 하나의 큰 연속된 페이지로 만들어 연속된 페이지들에 대한 할당 요구에 대비하도록 하고 있다.

4.9. Buddy 알고리즘

리눅스의 메모리 할당 알고리즘은 Buddy¹⁰⁰ Algorithm에 기초하고 있다. 즉, 페이지 단위의 할당에 기본을 두고 있다. 각각의 free 메모리에 대상하는 페이지는 free_area의 bitmap으로 사용 여부를 나타내며, 인접한 2^{order} 갯수의 연속된 페이지는 서로 합쳐질 수 있으며, 다시 더 작은 페이지들을 할당하기 위해서 나누어 질 수 있다. 여기서 order에 들어갈 수 있는 수는 0에서 9에 해당한다. 따라서, 최대 2^9 개의 연속된 페이지가 할당 될 수 있다. 이들 각각의 연결된 페이지들은 다시 여러 zone으로 나누어지며, DMA 가능, 일반 메모리, High memory의 zone으로 되어있다. 각각의 zone은 자신이 가진 free 메모리에 대한 정보를 유지하고 있다.

Buddy 알고리즘을 사용하는 목적은 연속된 페이지 프레임들의 그룹을 할당하는 것에 두고 있다. 즉, 연속적으로 메모리를 할당하고, 나중에 이를 다시 합치는 방법을 고안한 것이다. 이와 같이 해서 얻을 수 있는 장점은 바로 외부 단편화(external fragmentation)를 줄인다는 점이다. 즉, 할당과 해제가 많이 일어나는 경우 이것을 적절히 처리하지 못하면, 충분한 메모리 공간이 있음에도 불구하고, 메모리

⁹⁸ mask는 음수이다. 즉, order이하로 들어갈 수 있는 페이지들의 수를 가지고 있다. 2로 나눈 몫을 가지고 있다는 뜻이다.

⁹⁹ 페이지 덩어리라고 표현한 것은 여러개의 페이지가 2의 제곱승 갯수 만큼씩 order로 표현되기 때문이다. 만약 order 가 1이라면 2개의 연속된 page가 될 것이며, 2라면 4개의 연속된 페이지가 될 것이다. 빈번한 페이지 단위의 연산이 일어나기 때문에, 페이지 덩어리들은 쉽게 여러개의 조각 페이지가 될 수 있으며, 나중에 더 큰 메모리의 할당을 위해서 연속된 페이지들은 합쳐져서 관리되어야 한다. 이것이 Buddy Algorithm의 핵심이다.

¹⁰⁰ 영어 단어상 buddy는 절친한 친구, 동료를 말한다. 메모리에서 인접한 페이지를 같이 고려해서 할당과 해제를 해주겠다는 의미로 받아들이면 될 것이다.

요구를 처리하지 못하게 되는 경우가 발생하는데, 이것은 페이지 단위의 내에서 일어나는 쓰이지 않는 메모리를 말하는 것이 아니라, 페이지 단위보다 큰 곳에서 생기는 문제이다.

한번 내부 단편화(internal fragmentation)은 할당받은 메모리를 다 사용하지 않는데서 생기는 문제로 페이지 단위의 할당에서 할당받은 메모리를 페이지 크기에 맞춰서 다 사용하지 않는 것을 말한다. 페이지 단위로 할당할 경우, 만약 페이지 크기가 4096bytes라고 할 때, 할당 요구가 4000bytes하고 한다면, 나머지 96bytes는 사용하지 않는 상태가 된다. 그리고, 해제시에도 역시 페이지 단위로의 해제만이 가능하므로, 페이지 내부의 일부분만을 해제하지는 못하게 되는데서 발생하는 문제이다.

Buddy 알고리즘을 구현하기 위한 방법은 이미 커널의 코드를 설명하는 곳에서 보았다. 이곳에서는 알고리즘상에서 어떻게 진행되는 것을 보여주도록 하겠다. 먼저 **buddy**라고 지칭하는 의미인데, 이는 같은 크기를 가지는 두개의 메모리 블록이 서로 물리적으로 연속된 주소 공간을 차지하고 있으며, 두 메모리 블록중 물리적인 메모리에서 선행하는 위치에 있는 블록의 첫번째 페이지 프레임이, 페이지 크기(PAGE_SIZE)에 블록의 두배크기를 곱한 위치에 있을 경우를 뜻한다. 예를 들어, 블록의 크기가 4개의 페이지로 나타내진다고 할 때, $2 \times 4 \times \text{PAGE_SIZE}$ 에 첫번째 블록의 첫 페이지 프레임이 존재했을 때를 말한다. 만약 이와 같은 두개의 메모리 블록이 존재한다면, 두개의 메모리 블록은 **buddy**이며, 사용되지 않을 때는 합쳐져서 8개의 페이지로 이루어진 하나의 메모리 블록을 생성할 것이다. 이와 같은 연산은 반복적으로 더 큰 메모리 블록에 대해서 계속 이루어지며, 전체 메모리 블록을 다시 다 합쳐 놀을 수도 있을 것이다.

할당하는 방법은 위에서 했던 방법을 하위로 진행시키는 것이다. 예를 들어, 10개의 페이지를 할당 요청을 처리한다고 가정하고, 모든 free 페이지 프레임은 1, 2, 4, 8, 16, 32, 64, 128, 256, 512¹⁰¹개의 연속된 페이지 프레임의 그룹으로 쪼개어서 관리된다고 가정하자. 10개의 페이지를 할당해야 하기에, 16개의 연속된 페이지 프레임에서 할당 받는다. 나머지 6개에 대해서는 다시 free 페이지 프레임을 관리해 주어야 하기에, 2개와 4개로 쪼개어서 각각 인덱스 값으로 1과 2를 가지는 free area 배열의 리스트에 삽입될 것이다. 따라서, buddy 알고리즘을 사용할 경우 최대로 살펴보게 되는 값은 먼저 해당하는 리스트를 찾기위해 10번을, 그리고, 최대 512개의 페이지 프레임을 반으로 나눈값에 1을 더한 값을 다시 다른 free 페이지 프레임을 만들기 위해서 나누어주는 과정이 9번을 반복해야 할 것이다.

Buddy 시스템을 관리하기 위해서 해당 페이지의 사용여부를 나타내는 bitmap이 필수다. 이것은 앞에서 살펴본 bitmap에 대한 이야기가 되며, 하나의 bit은 인접은 2개의 페이지 블록이 사용중인지 그렇지 않은지를 표현한다. 만약 두 페이지 블록중에 하나만 사용중인 경우에는 1을, 그렇지 않고 둘다 사용중이거나 둘다 사용되지 않을 경우에는 0을 가지도록 만든다. 새로운 연속된 페이지 블록이 할당되면, 페이지 블록들의 상태를 나타내는 bitmap은 갱신될 것이며, 해제시에 이 bitmap을 이용해서 행당하는 연속된 페이지 블록들의 상태 정보를 읽어서 서로 합쳐야 할지를 결정하게 된다.

¹⁰¹ 실제로 free area를 나타 배열은 10개의 엔트리를 가진다는 것을 이미 보았다. 엔트리의 index가 2^{index} 개의 연속된 페이지 프레임을 가지는 free area를 표현하고 있다. 이러한 index를 order라는 말로 앞에서 사용했었다.

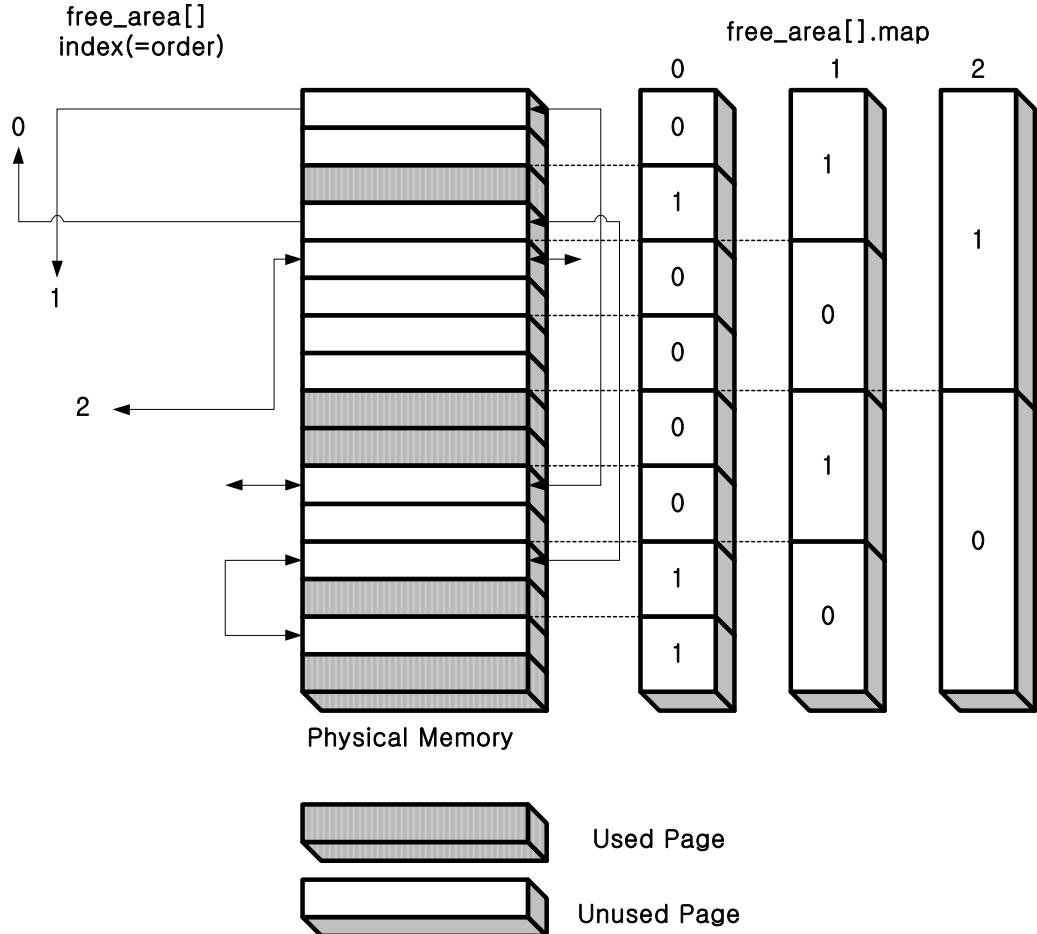


그림 42. Buddy Algorithm의 예

[그림35]는 buddy 알고리즘의 간단한 예를 보여준다. 만약 위에서 3번째에 있는 페이지 프레임이 해제가 된다면, 해당하는 `free_area[0]`에 있는 bitmap은 0으로 바뀔 것이며 4개의 연속된 페이지 프레임이 사용 가능하게 된다. 따라서, `free_area[1]`에 있는 bitmap의 첫번째 엔트리 1도 0으로 바뀌게 되며, 이것은 다시 8개의 연속된 페이지 프레임으로 합쳐질 수 있으므로, `free_area[2]`에 있는 1도 0으로 바뀔 것이다.

4.10. Slab Allocator 알고리즘

Slab¹⁰² 할당자(allocator)는 Buddy 알고리즘의 단점을 개선하기 위해서 커널 버전 2.2.X에서부터 새로이 생성된 것이다. 이것은 리눅스 만의 독특한 방법이라기보다는 솔라리스 2.4에서 있는 개념을 빌려온 것이다. 솔라리스에서 살펴보면 slab 할당자는 커널에서 관리하는 데이터 구조체에 대한 할당에서 사용하며, 이름에서 의미하듯이, 큰 크기를 가지는 메모리 영역(slab이라고 부른다.)을 커널 데이터 구조체를 위해서 작은 부분들로 쪼개는 역할을 한다. 할당이 일어나는 메모리의 풀(pool)들은, 동일한 연속적인 메모리 조각(segment)들에서 같은 크기를 가지는 객체(object)들이 할당 되도록 구성되어 있으며, 이렇게 함으로써 내부 단편화(internal fragmentation)을 크게 줄여주고 있다.

먼저 Buddy 할당 알고리즘이 가지는 약점을 보도록 하자. 아래와 같다.

- 할당 요구를 만족시키는데 있어서 속도가 느린다.
- 심각한 단편화를 발생 시킬 가능성이 있다.
- 할당된 메모리를 관리하는데, 많은 메모리를 사용하므로 메모리의 낭비가 심하다.

¹⁰² 영어 단어 상의 의미는 넓은 판을 이야기 한다. 즉, 메모리의 넓은 판의 형태로 보고, 특정한 부분에 대한 메모리는 특정한 판에 있는 영역에서 할당한다고 가정한다.

- 메모리 할당에 관련되어, 일관된 인터페이스를 제공하지 못한다. 따라서, 비슷한 코드들이 커널의 여러 곳에서 발견된다.

이에 반해 slab 할당자를 쓰게 되는 경우는, 아래와 같은 3가지가에 있어서 더 좋은 성능을 발휘한다.

- 페이지의 크기보다 작은 메모리를 할당하는 경우.
- 짹수개의 페이지 크기의 배로 메모리를 할당하는 경우. 이 경우 Buddy 알고리즘을 쓸 경우, 하나 이상의 페이지가 사용되지 않을 가능성이 있다.
- 자주 할당과 해제가 있는 경우. 이 경우 단편화가 심각한 영향을 미칠 가능성이 있다.

커널은 slab 할당자를 위해서 공간을 할당해주고 있으며, slab 할당자는 메모리를 일정 부분의 크기를 가진 부분들로 나누어서, 각각이 하나의 object형태로 관리되도록 한다. 관련된 object에 대한 함수도 역시 존재해서, 해당 object에 대해서 연산을 적용할 수 있다. 이 같은 함수에는 생성(construct)와 해제(destroy)가 있을 수 있을 것이다.

Slab 할당자에서 쓰는 용어에 대해서 간단히 보면, 먼저 object라는 것을 볼 수 있다. 이곳에서 사용하는 object란 하나의 메모리 할당 단위이다. 캐시(cache)¹⁰³는 그러한 object들의 풀(pool)이며, slab이란 캐시내에 있는 object들의 그룹을 말한다. 각각의 object는 태입을 가지고 있으며, 하나의 캐시를 가질 수 있다. 이러한 캐시는 하나나 그 이상의 slab으로부터 생성될 수 있다.

Slab 할당자는 내부 단편화 현상을 줄일 수 있다고 했는데, 이것이 가능한 것은 서로 다른 크기를 가지는 메모리 object들을 분리된 캐시에 저장해서 그룹화 시키기 때문이다. 각각의 object 캐시는 자신만의 object 크기와 특성을 가지고 있다. 즉, 비슷한 크기를 가지는 캐시들에 메모리 object들을 그룹화 시키기 때문에, 할당자는 각각의 캐시내에 있는 free 영역을 최소화 시킬 수 있다. 물론 slab내에 object들을 루는 것에 많은 주의를 요구하며, 캐시내에 있는 slab들은 연속된(contiguous) 페이지들의 그룹으로 구성된다.

메모리의 요구를 받으면 해당하는 object를 slab에서 할당하며, slab들은 캐시내에 보관된다. 또한 각각의 slab들은 해당하는 메모리 페이지들을 가지고 있다. 현재 시스템에서 가지고 있는 slab 할당자의 정보를 보고 싶다면, /proc/slabinfo를 열어서 보기 바란다¹⁰⁴.

이젠 slab 할당자를 사용하는 방법에 대해서 보기로 하자. 먼저 볼 것은 slab 할당자를 위한 데이터 구조이다. 이것을 이해하는 것이 항상 모든 것의 핵심이기 때문이다. 정의는 ~/mm/slab.c에 kmem_cache_s¹⁰⁵로 나와 있다. 동일한 태입의 정의가 kmem_cache_t이다.

Field	Description
struct list_head slabs	slab의 연결 리스트 구조
struct list_head firstnotfull	다 사용되지 않은 slab에 대한 연결 리스트 구조
unsigned int objsize	slab내에 있는 object의 크기
unsigned int flags	slab의 상태를 나타내는 flag
unsigned int num	slab당 object의 수
spinlock_t spinlock	slab에 대한 spin lock
#ifdef CONFIG_SMP unsigned int batchcount #endif	CONFIG_SMP가 설정된 경우에 사용되는 필드로, 한꺼번에 할당 가능한 slab의 갯수
unsigned int gfporder	slab당 페이지들의 order값(2의 승수에 해당한다.)
unsigned int gfpflags	GFP flag으로 앞에서 나온 GFP flag들이 사용된다.
size_t colour	캐시의 colour 범위

¹⁰³ 여기서 말하는 캐시는 하드웨어에 존재하는 캐시하고는 상관이 없다. 물론 하드웨어 캐시가 일부를 가질 수는 있으나, 어디까지나 소프트웨어적인 개념으로 데이터를 보관하는 역할을 하는 메모리 영역이다.

¹⁰⁴ 어떤 목적을 위해서 slab 할당자를 쓰는지는 확인할 수 있지만,

¹⁰⁵ Slab 할당자에서 쓰는 캐시에 대한 descriptor이다. 일반적인 cache를 나타낼 때도 사용된다.

unsigned int colour_off	colour의 offset
unsigned int colour_next	다음에 있는 캐쉬의 colour
kmem_cache_t slabp_cache	캐쉬내에 있는 slab의 포인터
unsigned int growing	캐쉬의 크기가 증가 상태에 있다.
unsigned int dflags	동적인 flag값
void(*ctor)(void*, kmem_cache_t *, unsigned long)	생성 함수에 대한 포인터
void(*dtor)(void*, kmem_cache_t *, unsigned long)	해제 함수에 대한 포인터
unsigned long failures	실패 이유
char name[CACHE_NAMELEN]	캐쉬의 이름을 나타냄
struct list_head next	캐쉬의 생성과 삭제를 위한 연결 리스트
#ifdef CONFIG_SMP cpucache_t *cpudata[NR_CPU] #endif	CPU당 캐쉬와 관련해서 유지해야하는 데이터
#if STATS unsigned long num_active unsigned long num_allocations unsigned long high_mark unsigned long grown unsigned long reaped unsigned long errors #endif CONFIG_SMP atomic_t allochit atomic_t allocmiss atomic_t freehit atomic_t freemiss #endif #endif	현재 캐쉬의 상태 정보를 알려주는 필드들이다. 먼저 active한 캐쉬의 갯수 할당된 캐쉬의 갯수 캐쉬의 high water mark 캐쉬의 크기가 늘어났다. 캐쉬가 획득(reap) 되었다. 캐쉬에 관해서 일어난 에러의 수 CONFIG_SMP정의가 있을 경우에 해당한다. 할당시에 hit이 일어난 횟수 할당시에 miss가 일어난 횟수 할당시에 free인 캐쉬에서 hit이 일어난 횟수 할당시에 free인 캐쉬에서 miss가 일어난 횟수

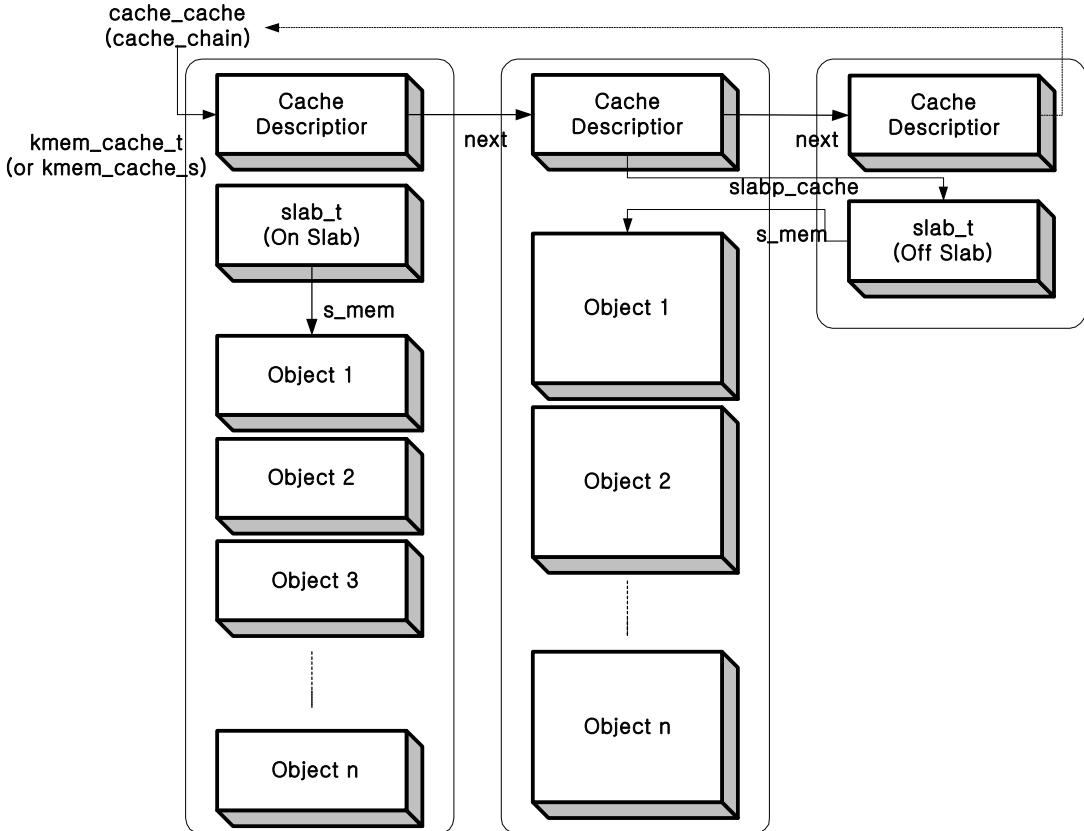
표 33. kmem_cache_s(kmem_cache_t)의 정의

Slab내에 있는 하나의 object를 다루기 위한 데이터 타입으로는 slab_t(slab_s)이 정의 되어 있다. 아래와 같다.

Field	Description
struct list_head list	slab_s타입을 가지는 구조체의 연결 리스트
unsigned long colouroff	slab의 colour offset
void *s_mem	slab을 위해 할당된 메모리의 시작 번지
unsigned int inuse	slab내에서 active한 object들의 수
kmem_bufclt_t free	slab내에서 free한 영역을 가리킨다.

표 34. slab_s(slab_t)의 정의

kmem_cache_s(kmem_cache_t)와 slab_s(slab_t)의 관계는 아래의 [그림36]을 참고하기 바란다. [그림36]에서 하나의 캐쉬 디스크립터(kmem_cache_s or kmem_cache_t)은 여러개의 slab 디스크립터(slab_s or slab_t)를 가질 수 있다.

그림 43. `kmem_cache_t`(=`kmem_cache_s`)와 `slab_t`(`slab_s`)의 관계

캐시 메모리에 대한 초기화는 `~/init/main.c`에서 `start_kernel()` 함수를 수행하는 도중에, `kmem_cache_init()` 함수를 호출하므로써 이루어진다. `kmem_cache_init()` 함수는 전체 캐시에 대한 자료구조를 가지는 전역 변수인 `cache_cache`에 대한 초기화를 담당한다. `cache_cache`는 아래와 같이 정의된다.

```
static kmem_cache_t cache_cache = {
    slabs:           LIST_HEAD_INIT(cache_cache.slabs),
    firstnotfull:   &cache_cache.slabs,
    objsize:         sizeof(kmem_cache_t),
    flags:          SLAB_NO_REAP,
    spinlock:       SPIN_LOCK_UNLOCKED,
    colour_off:     L1_CACHE_BYTES,
    name:           "kmem_cache",
};
```

코드 283. `cache_cache` 구조체의 정의

`cache_cache`는 자신이 관리하는 `slab`의 리스트를 초기화하고, 가장 먼저 나오는 `full`이 아닌 `slab`의 포인터를 `firstnotfull`에 두며, 할당하는 `objsize`에는 `kmem_cache_t`의 크기를 넣는다. `flags`의 값으로는 `SLAB_NO_REAP`을 선언해두고, `colour_off`에는 `L1_CACHE_BYTES`를, 캐시의 이름에는 “`kmem_cache`”로 넣는다. `/proc/slabinfo` 파일에서 가장 먼저 나오는 캐시이다. 캐시의 `flags` 값으로 올 수 있는 것으로는 아래와 같은 값들이 있다.

<pre>/* flags for kmem_cache_alloc() */ #define SLAB_BUFFER #define SLAB_ATOMIC #define SLAB_USER #define SLAB_KERNEL</pre>	<pre>GFP_BUFFER /* 페이지 할당에서 사용하게 되는 Flag과 동일하다. GFP_ATOMIC GFP_USER GFP_KERNEL</pre>
-----------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------

```
#define SLAB_NFS GFP_NFS
#define SLAB_DMA GFP_DMA

#define SLAB_LEVEL_MASK      (__GFP_WAIT|__GFP_HIGH|__GFP_IO)
#define SLAB_NO_GROW         0x00001000UL /* 캐쉬 크기를 키우지 말것 */

#define SLAB_DEBUG_FREE      0x00000100UL /* 해제 시에 체크 연산을 수행 */
#define SLAB_DEBUG_INITIAL   0x00000200UL /* 생성자를 호출하라. */
#define SLAB_RED_ZONE         0x00000400UL /* 캐쉬에서 RED zone에 있는 object이다. */
#define SLAB_POISON          0x00000800UL /* Posion object이다. */
#define SLAB_NO_REAP          0x00001000UL /* 캐쉬로 부터 reap되지 않는다. */
#define SLAB_HWCACHE_ALIGN    0x00002000UL /* 하드웨어 캐쉬라인테 object를 위치시켜라.*/
#define SLAB_CACHE_DMA        0x00004000UL /* GFP_DMA 메모리를 사용할 것 */

/* flags passed to a constructor func */
#define SLABCTOR_CONSTRUCTOR 0x001UL      /* 생성자가 있다. */
#define SLABCTOR_ATOMIC       0x002UL      /* 생성자는 sleep 할 수 없다. */
#define SLABCTOR_VERIFY       0x004UL      /* 생성자가 verification을 수행한다. */
```

코드 284. 캐쉬에 설정되는 flags

`kmem_cache_init()` 함수는 전역 구조체인 `cache_cache` 변수를 초기화 하는 것을 담당한다. 위에서 나온 `cache_cache`의 선언에서 이미 일부 필드들은 초기화 되었다.

```
#define cache_chain (cache_cache.next)
...
void __init kmem_cache_init(void)
{
    size_t left_over;

    init_MUTEX(&cache_chain_sem);
    INIT_LIST_HEAD(&cache_chain);
    kmem_cache_estimate(0, cache_cache.objsize, 0,
                        &left_over, &cache_cache.num);
    if (!cache_cache.num)
        BUG();
    cache_cache.colour = left_over/cache_cache.colour_off;
    cache_cache.colour_next = 0;
}
```

코드 285. `kmem_cache_init()` 함수

먼저 semaphore 구조체로 선언된 전역변수 `cache_chain_sem`를 초기화 한다. 또한 `cache_cache.next`를 가르키는 `cache_chain` 리스트를 초기화 하고, `kmem_cache_estimate()` 함수를 호출한다. 만약 `cache_cache`의 `num` 필드가 0이면 새로운 cache를 생성할 수 없으므로 에러가 된다. 마지막으로 `cache_cache` 변수의 `colour` 필드를 `kmem_cache_estimate()` 함수에서 계산된 `left_over`를 `cache_cache`의 `colour_off` 필드로 나눈 값으로 두고, `colour_next`를 0으로 설정한다.

```
#define SLAB_LIMIT 0xFFFFFFFF
#define CFLGS_OFF_SLAB 0x010000UL /* slab management in own cache */
#define CFLGS_OPTIMIZE 0x020000UL /* optimized slab lookup */
...
static void kmem_cache_estimate (unsigned long gfporder, size_t size,
                                int flags, size_t *left_over, unsigned int *num)
{
    int i;
```

```

size_t wastage = PAGE_SIZE<<gfporder;
size_t extra = 0;
size_t base = 0;

if (!(flags & CFLGS_OFF_SLAB)) {
    base = sizeof(slab_t);
    extra = sizeof(kmem_bufctl_t);
}
i = 0;
while (i*size + L1_CACHE_ALIGN(base+i*extra) <= wastage)
    i++;
if (i > 0)
    i--;
if (i > SLAB_LIMIT)
    i = SLAB_LIMIT;
*num = i;
wastage -= i*size;
wastage -= L1_CACHE_ALIGN(base+i*extra);
*left_over = wastage;
}

```

코드 286. kmem_cache_estimate() 함수

kmem_cache_estimate() 함수는 주어진 slab의 크기에 대해서 object의 갯수와 소모량(wastage), 남은 byte수를 계산하는 함수이다. 넘겨 받는 인수들을 보면, gfporder에는 0을, size에는 cache_cache.objsize를, flags에는 0을, left_over에는 left_over의 주소를, num에는 cache_cach.num의 주소를 kmem_cache_init()에서 받았다.

먼저 소모량을 나타내는 wastage에 gfporder인 0으로 PAGE_SIZE를 좌로 shift했으므로 일단 PAGE_SIZE가 들어간다. CFLGS_OFF_SLAB가 flags에 설정되지 않았으므로 slab_t크기의 기본(base)값과 kmem_bufctl_t크기의 extra값이 들어갈 것이다. 이젠 하나의 slab_t이 차지하는 크기와 여러개의 kmem_cache_t이 차지하는 크기가 wastage보다 작을 때까지 while loop를 수행해서 몇개의 kmem_cache_t이 들어갈 수 있는지 계산한다. 만약 이렇게 계산된 값이 0 이상이라면 1을 빼서 하나는 사용한다고 표시한다. 만약 이렇게 계산된 값이 SLAB_LIMIT(=0xFFFFFFF)을 넘어선다면, 이 값을 SLAB_LIMIT으로 둔다. 즉, 4Gbytes개 이내로 설정한다. 이 값을 *num으로 설정하고, 소모량(wastage)은 object의 크기에 들어갈 수 있는 object의 갯수의 곱으로 빼고, 다시 slab_t의 크기에 L1_CACHE_ALIGN()을 적용한 값을 빼준다. 즉, 아직 할당은 받았지만 kmem_cache_t과 slab_t이 사용하고 있지 않기 때문이다.

다시 ~/init/main.c의 start_kernel()에서 mem_init()를 호출한 이후에, kmem_cache_sizes_init() 함수의 호출이 있다. 이 함수는 내부적으로 사용하게될 캐쉬와 일반적인(general) 캐쉬에 대해서 초기화 한다. 아래와 같다.

```

#define BREAK_GFP_ORDER_HI 2
#define BREAK_GFP_ORDER_LO 1
static int slab_break_gfp_order = BREAK_GFP_ORDER_LO
...
static unsigned long offslab_limit;
...
/* Size description struct for general caches. */
typedef struct cache_sizes {
    size_t                  cs_size;
    kmem_cache_t            *cs_cachep;
    kmem_cache_t            *cs_dmacachep;
} cache_sizes_t;
static cache_sizes_t cache_sizes[] =
#endif PAGE_SIZE == 4096
{
    { 32, NULL, NULL },
}

```

```

#endif
{
    { 64, NULL, NULL},
    { 128, NULL, NULL},
    { 256, NULL, NULL},
    { 512, NULL, NULL},
    { 1024, NULL, NULL},
    { 2048, NULL, NULL},
    { 4096, NULL, NULL},
    { 8192, NULL, NULL},
    { 16384,      NULL, NULL},
    { 32768,      NULL, NULL},
    { 65536,      NULL, NULL},
    {131072, NULL, NULL},
    {     0, NULL, NULL}
};

...
void __init kmem_cache_sizes_init(void)
{
    cache_sizes_t *sizes = cache_sizes;
    char name[20];

    if (num_physpages > (32 << 20) >> PAGE_SHIFT)
        slab_break_gfp_order = BREAK_GFP_ORDER_HI;
    do {
        sprintf(name, "size-%Zd", sizes->cs_size);
        if (!(sizes->cs_cachep =
            kmem_cache_create(name, sizes->cs_size,
                0, SLAB_HWCACHE_ALIGN, NULL, NULL))) {
            BUG();
        }
        /* Inc off-slab bufctl limit until the ceiling is hit.*/
        if (!!(OFF_SLAB(sizes->cs_cachep))) {
            offslab_limit = sizes->cs_size-sizeof(slab_t);
            offslab_limit /= 2;
        }
        sprintf(name, "size-%Zd(DMA)", sizes->cs_size);
        sizes->cs_dmacachep = kmem_cache_create(name, sizes->cs_size, 0,
            SLAB_CACHE_DMA|SLAB_HWCACHE_ALIGN, NULL, NULL);
        if (!sizes->cs_dmacachep)
            BUG();
        sizes++;
    } while (sizes->cs_size);
}

```

코드 287. kmem_cache_sizes_init()함수

실제 메모리에 있는 물리적인 페이지 수(num_physpages)가 0x1000((32<<20)>>PAGE_SHIFT)보다 큰 경우에는 slab_break_gfp_order를 BREAK_GFP_ORDER_HI로 설정해서 상위의 메모리(16Mbytes이상의 메모리)에 대해서 더 큰 페이지 order¹⁰⁶ 만을 사용하도록 해준다. do {} while() loop를 돌면서 kmem_cache_create()함수를 호출해서 각각의 크기에 대해서 캐시를 할당받는다. 할당받는 캐시는 하드웨어 캐시에 쉽게 저장되는 형태로 될 수 있도록 SLAB_HWCACHE_ALIGN을 인자로 넘겨준다. offslab_limit는 slab당 object의 최대 갯수를 나타내며, slab_t을 내부에 유지하지 않는 캐시들에 해당한다. 이것은 kmem_cache_grow()함수에서 loop를 제어하는데 사용된다. OFF_SLAB()매크로는 현재 캐시에

¹⁰⁶ 여기서 order란 앞에서 이미 나왔지만, 다시한번 상기한다는 생각에서 페이지의 수가 2의 제곱승으로 나타낼 때 그 승수를 말한다.

CFLGS_OFF_SLAB flag이 설정되어 있는지 확인하는 것이며, 만약 설정되지 않았다면, offslab_limit을 현재 캐쉬의 크기에서 slab_t를 뺀 값으로 주고, 다시 이 값을 반으로 나눈다.

앞에서는 DMA를 사용하지 않는 캐쉬를 할당했고, 이젠 DMA가 가능한 캐쉬(size->cs_dmacachep)을 할당한다. kmem_cache_create()함수에 SLAB_CACHE_DMA까지 설정해서 인자로 넘겨준다. 할당받지 못한다면, 에러가 되며, 다음 크기의 캐쉬를 할당하기 위해서 size의 옵셋을 증가 시켜준다.

이렇게해서 할당되는 캐쉬들에 대한 정보는 아래와 같이 /proc/slabininfo에서 찾을 수 있을 것이다.

...					
size-131072(DMA)	0	0	131072	0	0
size-131072	0	0	131072	0	0
size-65536(DMA)	0	0	65536	0	0
size-65536	0	0	65536	0	0
size-32768(DMA)	0	0	32768	0	0
size-32768	0	1	32768	0	1
size-16384(DMA)	0	0	16384	0	0
size-16384	0	1	16384	0	1
size-8192(DMA)	0	0	8192	0	0
size-8192	0	2	8192	0	2
size-4096(DMA)	0	0	4096	0	0
size-4096	10	40	4096	10	40
size-2048(DMA)	0	0	2048	0	0
size-2048	38	46	2048	20	23
size-1024(DMA)	0	0	1024	0	0
size-1024	26	48	1024	7	12
size-512(DMA)	0	0	512	0	0
size-512	46	56	512	6	7
size-256(DMA)	0	0	256	0	0
size-256	19	60	256	2	4
size-128(DMA)	0	0	128	0	0
size-128	469	510	128	16	17
size-64(DMA)	0	0	64	0	0
size-64	222	649	64	4	11
size-32(DMA)	0	0	32	0	0
size-32	1979	3503	32	20	31
...					

예제 1. /proc/slabininfo의 내용 중 일부

즉, 크기가 32에서 131072 bytes를 가지는 캐쉬가 각각 DMA영역과 일반 메모리 영역에서 할당 받았음을 확인할 수 있다. 나중에 이와 같은 영역들은 커널에서 해당 크기에 맞도록 캐쉬 메모리를 할당할 때 사용될 것이다.

이전 실제적인 캐쉬의 생성이 일어나는 kmem_cache_create()함수를 보도록 하자. 이 함수가 넘겨 받는 인자는 /proc/slabininfo에서 사용할 이름, 이 캐쉬에서 생성하고자 하는 object들의 크기, 페이지내의 옵셋, slab에서 사용하는 플랙값, object의 생성자(constructor) 함수, object의 파괴자(destructor) 함수이다. 성공했을 경우에는 캐쉬에 대한 포인터를, 그렇지 않으면 NULL을 돌려준다. 인터럽트 내에서는 호출이 되지 않지만, 함수의 수행중에는 인터럽트가 발생할 수 있다. 아래와 같은 flag들이 정의되어서 사용된다.

Flag	Description
SLAB_POSITION	이미 알려진 패턴(a5a5a5)로 slab을 초기화 시켜서, 초기화 되지 않은 메모리에 대한 언급을 알기 위해서 사용한다.
SLAB_RED_ZONE	할당된 메모리주의에 “Red”를 집어넣어서 버퍼의 overrun을 감지한다.
SLAB_NO_REAP	현재 메모리 pressure를 겪고 있다면, 이 캐쉬에 대해서 reap하지 못하게 한다.

SLAB_HWCACHE_ALIGN	캐쉬내에 있는 object들이 하드웨어 캐쉬 라인 ¹⁰⁷ 에 정렬되도록 만든다.
--------------------	---------------------------------------------------------

표 35. kmem_cache_create()함수의 flag 파라미터 정의

```

kmem_cache_t *kmem_cache_create (const char *name, size_t size, size_t offset,
                                 unsigned long flags, void (*ctor)(void*, kmem_cache_t *, unsigned long),
                                 void (*dtor)(void*, kmem_cache_t *, unsigned long))
{
    const char *func_nm = KERN_ERR "kmem_create: ";
    size_t left_over, align, slab_size;
    kmem_cache_t *cachep = NULL;

    if ((!name) ||
        ((strlen(name) >= CACHE_NAMELEN - 1)) ||
        in_interrupt() ||
        (size < BYTES_PER_WORD) ||
        (size > (1<<MAX_OBJ_ORDER)*PAGE_SIZE) ||
        (dtor && !ctor) ||
        (offset < 0 || offset > size))
        BUG();
    if (flags & ~CREATE_MASK)
        BUG();
    cachep = (kmem_cache_t *) kmem_cache_alloc(&cache_cache, SLAB_KERNEL);
    if (!cachep)
        goto opps;
    memset(cachep, 0, sizeof(kmem_cache_t));

```

코드 288. kmem_cache_create()함수

먼저 확인할 것은 넘겨받은 파라미터 값들이 올바른 값을 가지고 있는지를 확인하는 일이다. 이것이 끝나면, 실제적인 캐쉬 디스크립터의 메모리를 할당받는 kmem_cache_alloc()함수를 호출한다. 넘겨주는 값은 cache_cache전역 변수와 SLAB_KERNEL 플랙 값이다. 만약 할당 받을 수 없다면 opps로 제어를 옮긴다. 할당받았다면 메모리를 0으로 초기화 시킨다(memset()).

```

if (size & (BYTES_PER_WORD-1)) {
    size += (BYTES_PER_WORD-1);
    size &= ~(BYTES_PER_WORD-1);
    printk("%sForcing size word alignment - %s\n", func_nm, name);
}
align = BYTES_PER_WORD;
if (flags & SLAB_HWCACHE_ALIGN)
    align = L1_CACHE_BYTES;
/* Determine if the slab management is 'on' or 'off' slab. */
if (size >= (PAGE_SIZE>>3))
    flags |= CFLGS_OFF_SLAB;
if (flags & SLAB_HWCACHE_ALIGN) {
    while (size < align/2)
        align /= 2;
    size = (size+align-1)&(~(align-1));
}

```

코드 289. kmem_cache_create()함수 – continued

size가 WORD단위로 정렬되어 있는지를 확인해서, 그렇지 않다면 WORD단위로 정렬시킨다. 정렬 타입(align)을 BYTES_PER_WORD로 설정하고, 만약 넘겨받은 flag에 SLAB_HWCACHE_ALIGN이

¹⁰⁷ Intel의 경우 16bytes를 하나의 캐쉬라인으로 사용하고 있다.

설정되어 있다면, L1_CACHE_BYTES로 정렬 탑입을 바꾼다. 또한 크기가 PAGE_SIZE를 8로 나눈 값보다 크거나 같은지를 확인해서 slab_t이 어디에 위치할 것인지를 결정한다. 크다면 flag에 CFLGS_OFF_SLAB를 OR시킨다. 만약 flag에 SLAB_HWCACHE_ALIGN이 설정되었다면, 캐쉬에 있는 object들이 정렬될 수 있도록 size를 재조정한다.

```

do {
    unsigned int break_flag = 0;
cal_wastage:
    kmem_cache_estimate(cachep->gfporder, size, flags,
                        &left_over, &cachep->num);
    if (break_flag)
        break;
    if (cachep->gfporder >= MAX_GFP_ORDER)
        break;
    if (!cachep->num)
        goto next;
    if (flags & CFLGS_OFF_SLAB && cachep->num > offslab_limit) {
        /* Oops, this num of objs will cause problems. */
        cachep->gfporder--;
        break_flag++;
        goto cal_wastage;
    }
    if (cachep->gfporder >= slab_break_gfp_order)
        break;
    if ((left_over*8) <= (PAGE_SIZE<<cachep->gfporder))
        break;
next:
    cachep->gfporder++;
} while (1);

```

코드 290. kmem_cache_create()함수 – continued

kmem_cache_estimate()함수를 호출해서 캐쉬에 들어갈 수 있는 object들의 수 및 남은 공간이 얼마나 되는지 측정한다. break_flag값을 확인해서 만약 설정되었다면, loop를 끝내고 그렇지 않다면 계속 진행한다. 캐쉬를 요구한 gfporder값이 MAX_GFP_ORDER보다 크거나 같다면 캐쉬를 할당할 수 없으므로 loop를 중단한다. 또한 캐쉬에 들어갈 수 있는 object의 갯수가 0이라면 next로 제어를 옮겨서 gfporder값을 증가 시켜서 더 큰 영역에서 캐쉬를 할당받을 수 있는지 확인한다.

만약 flags에 CFLGS_OFF_SLAB이 설정되어 있고, 캐쉬에 들어갈 수 있는 object의 수가 offslab_limit보다 크다면, gfporder값을 줄여서 더 작은 곳에서 캐쉬의 할당이 일어나도록 만들고, break_flag의 값을 증가 시킨후 다시 한번 캐쉬에 들어갈 수 있는 object의 수를 다시 계산하도록 한다(cal_wastage).

만약 캐쉬의 gfporder가 slab_break_gfp_order보다 크거나 같다면 loop를 빠져나오게 된다. 또한 남게될 캐쉬의 공간(left_over)에 8을 곱한 값이 PAGE_SIZE를 캐쉬의 gfporder로 좌측으로 SHIFT한 값보다 작거나 같다면 받아 들일만한 내부 단편화(internal fragmentation)로 생각해서 loop를 빠져나오게 된다. 다시 한번 loop를 시작하게 될때는 캐쉬의 gfporder가 하나 증가할 것이다.

```

if (!cachep->num) {
    printk("kmem_cache_create: couldn't create cache %s.\n", name);
    kmem_cache_free(&cache_cache, cachep);
    cachep = NULL;
    goto opps;
}
slab_size = L1_CACHE_ALIGN(cachep->num*sizeof(kmem_bufctl_t)+sizeof(slab_t));
if (flags & CFLGS_OFF_SLAB && left_over >= slab_size) {
    flags &= ~CFLGS_OFF_SLAB;
    left_over -= slab_size;
}

```

```

    }
    offset += (align-1);
    offset &= ~(align-1);
    if (!offset)
        offset = L1_CACHE_BYTES;
    cachep->colour_off = offset;
    cachep->colour = left_over/offset;
}

```

코드 291. kmem_cache_create()함수 – continued

위에서 계산된 캐쉬의 object수가 0인 값을 확인한다. 만약 0을 가진다면, 할당받은 캐쉬를 해제하고(kmem_cache_free()), 캐쉬를 가르키는 포인터에 NULL을 둔 후, 제어를 opps로 옮긴다. 이젠 slab의 크기를 구할 차례이다. 캐쉬에 있는 object들을 다 표현하고 있어야 하므로 캐쉬의 num필드에 kmem_bufctl_t을 곱한 크기에 다시 slab_t의 크기를 더해서 이를 L1_CACHE_ALIGN()으로 정렬시킨다. 최종적으로 slab_size가 그 값을 가진다.

만약 flags에 CFLGS_OFF_SLAB가 설정되었고, 남은 크기가 slab_size를 충분히 가질 수 있다면, slab을 내부에 두도록 하기 위해서 flags에서 CFLG_OFF_SLAB을 지우고, left_over에서 slab_size를 뺀다. offset은 정렬 단위의 배수가 되도록 만들고¹⁰⁸, .offset값이 0이면 L1_CACHE_BYTES로 설정한다. 이렇게 계산된 값으로 캐쉬의 colour_off을 설정하고, colour에는 left_over/offset으로 나눈 값으로 설정한다.

```

/* init remaining fields */
if (!cachep->gfporder && !(flags & CFLGS_OFF_SLAB))
    flags |= CFLGS_OPTIMIZE;
cachep->flags = flags;
cachep->gfpflags = 0;
if (flags & SLAB_CACHE_DMA)
    cachep->gfpflags |= GFP_DMA;
spin_lock_init(&cachep->spinlock);
cachep->objsize = size;
INIT_LIST_HEAD(&cachep->slabs);
cachep->firstnotfull = &cachep->slabs;
if (flags & CFLGS_OFF_SLAB)
    cachep->slabp_cache = kmem_find_general_cachep(slab_size,0);
cachep->ctor = ctor;
cachep->dtor = dtor;
/* Copy name over so we don't have problems with unloaded modules */
strcpy(cachep->name, name);
#endif CONFIG_SMP
if (g_cpcache_up)
    enable_cpcache(cachep);
#endif

```

코드 292. kmem_cache_create()함수 – continued

이젠 나머지 할당받은 캐쉬의 필드들을 설정할 차례이다. gfporder필드가 0이고(즉, 캐쉬를 위해서 하나의 페이지를 할당 했다면), flags필드에 CFLGS_OFF_SLAB가 설정되지 않았다면, flags에 CFLG_OPTIMIZE를 설정한다. 이 값으로 캐쉬의 flags의 필드를 설정한 후, 다시 gfpflags에는 0을 넣어준다. 만약 flags에 SLAB_CACHE_DMA(DMA가능 메모리 영역에서 캐쉬를 할다할 경우)가 설정되어 있다면, 할당받은 캐쉬의 gfpflags에 GFP_DMA를 OR한다.

할당받은 캐쉬의 spinlock을 초기화하고, objsize에 size넣으며, slabs을 가리키는 리스트를 초기화한다(INIT_LIST_HEAD()). 만약 flags에 CFLGS_OFF_SLAB가 선택된 경우에는 slab따로이 저장하는 것으로 해당하는 slab의 캐쉬를 찾기 위해서 kmem_find_general_cachep()를 호출한다. 넘겨주는 것은 slab의 크기와 gfpflags(=0)이다.

¹⁰⁸ 항상 정렬되어 있으면 접근 속도가 빨라진다. 이것은 하드웨어의 특성을 항상 커널이 고려해 주어야 한다는 것을 말한다.

캐쉬에 대한 생성자와 파괴자 함수를 초기화하고, 캐쉬의 이름을 준다(cachep->name). CONFIG_SMP가 설정된 경우에는 g_cpcache_up¹⁰⁹을 확인해서 enable_cpcache()¹¹⁰에 할당한 캐쉬를 인자로 주어서 호출한다.

```

/* Need the semaphore to access the chain.*/
down(&cache_chain_sem);
{
    struct list_head *p;

    list_for_each(p, &cache_chain) {
        kmem_cache_t *pc = list_entry(p, kmem_cache_t, next);

        /* The name field is constant - no lock needed.*/
        if (!strcmp(pc->name, name))
            BUG();
    }
    list_add(&cachep->next, &cache_chain);
    up(&cache_chain_sem);
}
 opps:
    return cachep;
}

```

코드 293. kmem_cache_create()함수 – continued

이젠 캐쉬가 이미 cache_chain에 존재하는지를 확인하는 일이다. 이미 있다면 예러가 될 것이다. 없다면, 이것을 cache_chain에 삽입하도록 한다. 이 연산중에는 반드시 cache_chain에 대한 세마포어 값을 획득해야 할 것이다. 즉, cache_chain¹¹¹에 리스트의 엔트리를 추가하고 검사하는 과정이 있기 때문이다. 참고로, /proc/slabininfo에 있는 나머지 캐쉬들은 시스템이 그때 그때 상황에서 kmem_cache_create()함수를 수행한 결과이다.

kmem_cache_create() 함수의 내부에서 kmem_cache_alloc()함수를 호출해서 하나의 캐쉬를 할당 받을 수 있으며, 해제를 맡고 있는 것은 kmem_cache_free()함수이다. kmem_cache_alloc()함수는 다시 __kmem_cache_alloc() 함수를 호출해서 할당 요구를 처리한다. 실제적인 메모리의 할당은 __kmem_cache_alloc() 함수에서 kmem_cache_grow()함수를 호출해서 처리하며, 다시 kmem_cache_grow() 함수는 kmem_getpages()함수를 호출한다. 이 kmem_getpages() 함수를 보면 앞에서 설명했던 Buddy 알고리즘과의 관계를 확인할 수 있다. 즉, __get_free_pages()함수를 캐쉬의 flags과 gfporder를 넘겨주어서 새로운 페이지들의 뮤음을 할당 받는다. slab은 이렇게 할당받은 캐쉬내에 있을 수도 있으며, slab만을 위해서 새로이 할당받은 메모리에 있을 수도 있다(CFLGS_OFF_SLAB).

캐쉬에서 하나의 object를 삭제하고자 한다면 kmem_cache_free()함수에 캐쉬의 포인터와 object의 포인터를 넘겨줘서 처리한다. 이 함수는 다시 __kmem_cache_free()를 같은 파라미터 값으로 호출하게 되며, 이 함수에서 kmem_cache_free_one() 함수를 불러서 하나의 object에 대한 삭제를 처리한다. 캐쉬 크기를 줄이거나(kmem_slab_shrink()), kmem_cache_reap() 함수가 호출되면 kmem_slab_destroy() 함수가 호출되게 되며, 이 함수에서 캐쉬를 위해서 할당받은 페이지를 해제하는 kmem_freepages() 함수를 호출한다. 다시 이 함수에서는 Buddy 알고리즘에서 페이지를 해제하는 free_pages() 함수에 시작 번지의 주소와 캐쉬의 gfporder를 넘겨 주어서 캐쉬를 위해서 할당받은 페이지를 해제하게 된다.

마지막으로 slab 할당자에서 짚고 넘어가야 할 부분은 kmalloc()함수와 kfree()함수이다. 이 함수들은 커널에서 메모리를 할당하기 위해서 많이 사용되는 함수이므로 이곳에서 꼭 보아야 할 것이다.

¹⁰⁹ 현재 general cache가 가동 중임을 나타내는 flag값이다.

¹¹⁰ CPU의 캐쉬에 할당 받은 캐쉬를 옮겨놓도록 한다.

¹¹¹ cache_chain은 이미 보았듯이 cache_cache의 next필드이다.

```

void * kmalloc (size_t size, int flags)
{
    cache_sizes_t *csizep = cache_sizes;

    for (; csizep->cs_size; csizep++) {
        if (size > csizep->cs_size)
            continue;
        return __kmem_cache_alloc(flags & GFP_DMA ?
                                  csizep->cs_dmacachep : csizep->cs_cachep, flags);
    }
    BUG(); // too big size
    return NULL;
}

```

코드 294. kmalloc()함수

kmalloc()함수가 넘겨받는 파라미터 값은 할당할 메모리의 양과 메모리 할당과 관련된 flags값이다. Flags의 값으로 들어갈 수 있는 것은 앞에서 보았던, GFP_BUFFER, GFP_ATOMIC, GFP_USER, GFP_KERNEL, GFP_NFS, GFP_KSWAPD이다. 마지막에 사용된 GFP_KSWAPD는 swapd 데몬을 위한 경우로 해당 상황이 없을 것이다.

kmalloc()함수는 먼저 cache_sizes[]배열에서 넘겨받은 size를 포함할 수 있는 크기를 가지는 캐쉬에 대한 인덱스(csizep->cs_cachep or csizep->cs_dmacachep)를 구한 후, 이를 __kmem_cache_alloc()함수를 호출해서 캐쉬 메모리를 할당 받는다.

```

void kfree (const void *objp)
{
    kmem_cache_t *c;
    unsigned long flags;

    if (!objp)
        return;
    local_irq_save(flags);
    CHECK_PAGE(virt_to_page(objp));
    c = GET_PAGE_CACHE(virt_to_page(objp));
    __kmem_cache_free(c, (void*)objp);
    local_irq_restore(flags);
}

```

코드 295. kfree()함수

kfree()함수가 넘겨 받는 파라미터는 kmalloc()함수에서 생성된 메모리에 대한 포인터이다. 먼저 넘겨받은 인자(objp)가 올바른지를 확인한 후, lock을 설정하고, 인자가 올바른 메모리 공간에 대한 언급인지 확인한다(CHECK_PAGE()). 인자가 가지는 페이지 캐쉬에 대한 포인터를 얻은 후에, __kmem_cache_free()함수에 앞에서 얻은 캐쉬와 인자를 넘겨서 할당받은 메모리를 해제한다.

이상에서 우린 커널내에서의 메모리 할당과 관련된 것을 간략하게 나마 살펴보았다. 가장 기본이 되는 것은 Buddy 알고리즘이며, 이를 이용해서 크기 별로 묶은 메모리를 할당하는 것은 Slab 할당자가 처리한다. 이제부터 보고자 하는 것은 프로세스와 관련된 메모리 할당이다.

4.11. 프로세스의 주소 공간 관리

프로세스는 실행되기 위해서는 디스크로부터 해당 프로그램을 메모리로 읽어들여야만 한다. 이때, 전체 프로그램을 다 읽어들이기보다는 프로그램의 일정 부분만을 메모리에 읽어들이는 편이 실행을 더 효과적으로 할 수 있다. 즉, 여러분의 메모리 공간을 다른 프로세스가 점유해서 사용할 수 있으므로, 더 많은 프로세스에게 실행 기회를 열어줄 수 있으며, 또한 전체 프로그램을 다 읽어들이지 않고, 수행을 시작하게 되므로 프로그램의 실행까지 걸리는 시간도 단축할 수 있다. 이것은 프로세스의 수행중인

상태를 고찰해서 나온 결과로, 프로그램의 일정 부분이 반복적으로 수행된다는 locality¹¹²에 기반하고 있다. 따라서, 프로그램의 실행 초기에는 다순히 메모리만을 예약해주고, 이후에 해당 프로그램에서 명령(instruction)을 가져와서 수행하게 될 때, 실제적인 메모리를 디스크로부터 프로그램을 읽어서 설정해준다. 역시 이때도 페이지 단위의 연산이 일어날 것이다. 수행이 다른 페이지에 대한 언급을 할 경우에는 다시 그 페이지를 디스크로 부터 읽어와서 메모리에 넣어주면 되기 때문에, 이를 demand paging¹¹³이라고 한다. 즉, 요구될 때만 할당해 주겠다는 말이다.

요구 페이징을 구현하는데 있어서는 두가지 고려할 점이 있다. 즉, 프로세스가 올바른 페이지에 대한 언급을 하고 있는지 확인할 방법이 필요하며, 만약 올바르지 않은 언급일 경우에는 커널이 프로세스가 올바르지 않은 페이지에 대한 접근을 한다고 알려주어야 하며, 올바른 페이지에 대한 언급일 경우에는 해당 페이지를 디스크로 부터 읽어서 메모리에 넣어주어야 할 것이다.

프로세스의 메모리 사용을 위한 데이터 구조는 task_struct에 있는 mm_struct 구조체로 나타낸다. 아래와 같다.

```
struct mm_struct {
    struct vm_area_struct * mmap;           /* list of VMAs */
    struct vm_area_struct * mmap_avl;        /* tree of VMAs */
    struct vm_area_struct * mmap_cache;      /* last find_vma result */
    pgd_t * pgd;
    atomic_t mm_users;          /* How many users with user space? */
    atomic_t mm_count;          /* How many references to "struct mm_struct" (users count as 1) */
    int map_count;              /* number of VMAs */
    struct semaphore mmap_sem;
    spinlock_t page_table_lock;
    struct list_head mmlist;       /* List of all active mm's */
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;
    unsigned long rss, total_vm, locked_vm;
    unsigned long def_flags;
    unsigned long cpu_vm_mask;
    unsigned long swap_cnt;         /* number of pages to swap on next pass */
    unsigned long swap_address;
    /* Architecture-specific MM context */
    mm_context_t context;
};
```

코드 296. mm_struct의 정의

mm_struct 구조체는 ~/include/linux/sched.h에 정의되어 있으며, 각각의 필드가 가지는 의미는 아래와 같다.

Field	Description
mmap	첫번째 vm_area_struct에 대한 포인터로 가상 메모리를 가르킨다.
mmap_avl	AVL ¹¹⁴ tree로 관리되는 가상 메모리들에 대한 포인터를 가진다. AVL tree의 root를 가진다.

¹¹² 지역성이란 뜻으로 프로그램의 일정부분이 반복적으로 수행될 때를 말한다. 즉, 특정한 부분을 나타내는 말로도 쓰인다.

¹¹³ 우리말로 표현하자면 요구 페이징이다. 조금 어감이 이상하기는 해도 가장 잘 그 의미를 표현하고 있다.

¹¹⁴ Adelchild-Velskii and Landis(AVL) tree로 이진 트리(binary tree)의 특수한 형태이다. 두개의 하위 node에 대한 포인터를 가지며, 이것을 right child와 left child로 나타낸다. right child들은 항상 node보다 작은 값은 index로 가지며, left child는 항상 node보다 큰 값을 index로 가진다. 항상 balanced된 형태로 유지되는데, 모든 node의 left child의 depth와 right child의 depth의 차이가 1이거나 -1 혹은 0이 되는 것을 balance되어

mmap_cache	마지막으로 가상 메모리에 대한 검색을 수행한 위치를 가진다.
pgd	Page Global Directory의 포인터를 가진다.
mm_users	사용자 주소 공간을 가지고 있는 mm_struct 구조체의 사용자가 얼마나 있는가를 나타낸다.
mm_count	mm_struct 구조체를 사용하고 있는 프로세스의 갯수
map_count	가상 주소의 갯수(vm_area_struct의 갯수)
mmap_sem	메모리 mapping에 사용되는 세마포어
page_table_lock	페이지 테이블에 대한 접근시 사용되는 lock
mmlist	모든 active한 mm_struct 구조체의 리스트
start_code, end_code, start_data, end_data	코드(프로그램)의 시작과 끝, 데이터의 시작과 끝을 나타낸다.
start_brk, brk, start_stack	brk()콜에 의해서 할당된 메모리 블록의 시작(start_brk)과 brk()콜에 의해서 할당된 메모리 블록의 끝(brk), 그리고, 스택의 시작(start_stack)을 나타낸다.
arg_start, arg_end, env_start, env_end	프로세스가 넘겨받는 argument의 시작과 끝(arg_start, arg_end)을 나타내며, 환경변수(environmental variable)의 시작과 끝(env_start, env_end)을 나타낸다.
rss, total_vm, locked_vm	프로세스를 위해서 현재 메모리에 있는 페이지의 카운트(rss)와 전체 주소 공간상에 있는 bytes의 수(total_vm), 그리고 lock된 메모리의 byte수(locked_vm)를 나타낸다.
def_flags	메모리 영역이 생성될 때 선택될 기본 상태(status)를 나타낸다.
cpu_vm_mask	CPU의 가상 메모리에 대한 mask값
swap_cnt	Swapping된 가상 메모리에 대한 count값
swap_address	Swapping된 가상 메모리의 address값
context	CPU에 의존적인 context값으로 segment을 가진다.

표 36. mm_struct의 필드 정의

context필드는 아키텍쳐에 의존적인 정보를 담는 것으로 i386의 경우에는 아래와 같이 정의되며, 프로세스의 local descriptor table에 대한 포인터를 저장한다.

```
typedef struct {
    void *segments;
} mm_context_t;
```

코드 297. mm_context_t의 정의

def_flags에 설정될 수 있는 값은 ~/include/asm/mman.h에 아래와 같이 정의되어 있다. 각각의 값이 하는 역할도 찾을 수 있을 것이다.

Flag	Value	Descriptions
MAP_SHARED	0x0001	공유되는 맵핑이다. 메모리에 대한 변화는 다른 프로세스들이 같은 메모리를 공유할 때, 공유된다.
MAP_PRIVATE	0x0002	메모리 맵핑이 개인(private)으로 국한된다. 즉, 해당 메모리 영역의 변화는 다른 프로세스에서는 알지 못한다.
MAP_TYPE	0x000F	맵핑 타입의 mask값
MAP_FIXED	0x0010	주소를 정확히 해석(interpret)하라.
MAP_ANONYMOUS	0x0020	Anonymous mapping인 경우로 파일에 대해서는 사용하지 않는다.
MAP_GROWSDOWN	0x0100	Stack처럼 하위 번址로 메모리 사용이 커나간다.
MAP_DENYWRITE	0x0800	Write가 금지되어 있다. 예전에 -ETXTBSY가 돌려질 것이다.

있다고 한다. 즉, 새로운 노드를 더하거나 뺄 때, 모든 node의 right child와 left child가 balance되도록 만들어 주어야 한다. 이렇게 했을 경우 최대 검색횟수는 $\log_2 N$ 이 될 것이다. 여기서 N은 node의 수를 말한다.

MAP_EXECUTABLE	0x1000	실행가능하다는 것을 나타낸다.
MAP_LOCKED	0x2000	맵핑된 메모리의 페이지들이 lock되었다.
MAP_NORESERVE	0x4000	예약에 대한 것을 검사하지 말라.

코드 298. mm_struct의 def_flags값

새로운 mm_struct구조체의 할당은 mm_alloc()이 맡고 있다. 새로운 프로세스를 생성하는 fork()시스템 콜에서 mm_alloc()함수를 사용하는 것을 찾을 수 있을 것이며, 아래와 같다.

```
#define allocate_mm()      (kmem_cache_alloc(mm_cachep, SLAB_KERNEL))
#define free_mm(mm)        (kmem_cache_free(mm_cachep, (mm)))

static struct mm_struct * mm_init(struct mm_struct * mm)
{
    atomic_set(&mm->mm_users, 1);
    atomic_set(&mm->mm_count, 1);
    init_MUTEX(&mm->mmap_sem);
    mm->page_table_lock = SPIN_LOCK_UNLOCKED;
    mm->pgd = pgd_alloc();
    if (mm->pgd)
        return mm;
    free_mm(mm);
    return NULL;
}

struct mm_struct * mm_alloc(void)
{
    struct mm_struct * mm;

    mm = allocate_mm();
    if (mm) {
        memset(mm, 0, sizeof(*mm));
        return mm_init(mm);
    }
    return NULL;
}
```

코드 299. mm_alloc()함수

mm_alloc()함수는 allocate_mm() 함수를 다시 호출하여, 할당 받은 mm_struct구조체를 0으로 다시 초기화하며, mm_init()함수를 호출해서 mm_struct에 대한 초기화 작업을 수행한다. allocate_mm()함수는 kmem_cache_alloc()으로 정의된 함수이며, slab allocator로부터 필요한 mm_struct구조체를 할당 받는다. 초기화하는 mm_init()함수를 보면, mm_struct의 mm_users와 mm_count를 1로 설정하고, mm_struct의 세마포어를 초기화(init_MUTEX())한다. mm_struct의 페이지 테이블에 대한 lock상태로는 SPIN_LOCK_UNLOCKED로 설정하고, 프로세스의 PGD에 대한 영역을 할당한다(pgd_alloc()). 이 PGD값은 프로세스의 전체 주소 공간에 대한 접근에서 사용하게 됨으로 프로세스의 context switch에서 CR3레지스터에 들어갈 값이 된다. 만약 PGD를 할당 받지 못할 경우에는 이미 할당된 mm_struct구조체를 해제하고(free_mm()) NULL로 복귀한다. 그렇지 않다면, 할당받은 mm_struct가 복귀 값이 될 것이다. free_mm()은 위에서 보면 kmem_cache_free()로 정의되어 있다.

```
/* ~/include/linux/sched.h에서 */
extern inline void FASTCALL(__mmdrop(struct mm_struct *));
static inline void mmdrop(struct mm_struct * mm)
{
    if (atomic_dec_and_test(&mm->mm_count))
        __mmdrop(mm);
```

```

}
/* ~/kernel/sched.c에서 */
inline void __mmdrop(struct mm_struct *mm)
{
    if (mm == &init_mm) BUG();
    pgd_free(mm->pgd);
    destroy_context(mm);
    free_mm(mm);
}

void mmput(struct mm_struct *mm)
{
    if (atomic_dec_and_lock(&mm->mm_users, &mmlist_lock)) {
        list_del(&mm->mmlist);
        spin_unlock(&mmlist_lock);
        exit_mmap(mm);
        mmdrop(mm);
    }
}

```

코드 300. mmput()함수

mm_alloc()의 반대 역할을 하는 함수는 mmput()함수이다. mmput()함수는 먼저 mm_users에 대한 decrement를 해서 이 값이 0이 되면, 해당 mm_struct구조체를 없앤다. 즉, mm_struct의 mmlist에서 제거시키고(list_del()), mmlist_lock에 설정된 lock을 해제한 후, exit_mmap()함수와 mmdrop()함수를 호출한다. mmdrop()함수는 다시 mm_count필드를 감소시킨 후, __mmdrop()을 호출하는데, __mmdrop()함수는 커널(init_mm)에 대한 호출인지를 확인하고 그렇지 않다면, 해당 페이지 디렉토리를 해제(pgd_free())하고, mm_struct의 CPU의존적인 내용(context)¹¹⁵을 없앤 후(destroy_context()), free_mm()을 호출해서 mm_struct구조체를 커널에서 삭제한다.

exit_mm()은 ~/kernel/exit.c에 아래와 같이 정의되어 있다. 프로세스의 종료시에 호출되기 때문이다. 위에서 mmdrop()함수를 호출하기 전에 불리는 것을 볼 수 있다.

```

/* ~/kernel/fork.c에서 */
void mm_release(void)
{
    struct task_struct *tsk = current;

    /* notify parent sleeping on vfork() */
    if (tsk->flags & PF_VFORK) {
        tsk->flags &= ~PF_VFORK;
        up(tsk->p_opptr->vfork_sem);
    }
}

/* ~/kernel/exit.c에서 */
static inline void __exit_mm(struct task_struct * tsk)
{
    struct mm_struct * mm = tsk->mm;

    mm_release();
    if (mm) {
        atomic_inc(&mm->mm_count);
        if (mm != tsk->active_mm) BUG();
        /* more a memory barrier than a real lock */
    }
}

```

¹¹⁵ 프로세스의 LDT(Local Descriptor Table)이 될 것이다.

```

        task_lock(tsk);
        tsk->mm = NULL;
        task_unlock(tsk);
        enter_lazy_tlb(mm, current, smp_processor_id());
        mmput(mm);
    }
}

void exit_mm(struct task_struct *tsk)
{
    __exit_mm(tsk);
}

```

코드 301. exit_mm()함수

exit_mm()함수는 __exit_mm()함수의 wrapper함수이다. __exit_mm()함수는 먼저 task의 mm_struct를 구해와서 이를 mm으로 놓는다. mm_release()를 호출해서 현재 프로세스(current)의 task_struct에 있는 flags필드에 PF_VFORK가 설정되었는지 확인한다. 만약 설정되었다면, PF_VFORK를 지우고, 현재 프로세스의 original parent 프로세스를 가르키는 p_opptr의 vfork_sem을 증가 시켜 vfork()로 잠들어있는 부모 프로세스를 깨우도록 한다. 즉, 현재 프로세스가 메모리를 해제하고 종료하려고 한다는 것을 부모 프로세스에 알리게 되는 것이다.

이제 프로세스의 mm구조체가 있을 경우에, 이를 해제하도록 한다. mm_struct의 count를 증가시키고(atomic_inc), 만약 현재의 active mm이 아니라면 에러가 된다. 프로세스의 mm_struct구조체 필드를 NULL로 만들기위해서는 먼저 프로세스에 lock을 설정한다(task_lock()). 그리고, mm을 NULL로 둔고, 다시 프로세스에 설정했던 lock을 해제한다(task_unlock()). enter_lazy_tlb()함수에는 mm_struct와 현재 프로세스, 그리고, CPU의 ID를 인자로 넘겨서 호출한다. enter_lazy_tlb()함수는 현재 CPU의 TLB(Translation Lookaside Buffer)관리를 위해서 사용하는 함수이다. 현재 CPU의 TLB 상태가 TLBSTATE_OK인 경우에 TLBSTATE_LAZY로 설정을 바꾼다. 이는 SMP 시스템을 고려하기 위한 함수이며, 나중에 TLB의 내용을 비우겠다는 말이다. 마지막으로 사용을 마친 mm_struct구조체는 mmput()함수를 호출해서 버리도록 한다.

프로세스는 자신이 사용하는 메모리에 대한 정보를 mm_struct로 나타낸다는 것을 앞에서 살펴보았다. 이전 mm_struct가 실제 사용하는 메모리 영역(region)을 어떻게 나타내는지를 볼 차례이다. 프로세스가 사용하는 각각의 메모리 영역은 vm_area_struct로 나타낸다. 각각의 메모리 영역을 나타내는 vm_area_struct는 연속적인 선형 주소 공간을 표시하며, ~/include/linux/mm.h에 아래와 같이 정의되어 있다.

```

struct vm_area_struct {
    struct mm_struct * vm_mm; /* VM area parameters */
    unsigned long vm_start;
    unsigned long vm_end;
    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next;
    pgprot_t vm_page_prot;
    unsigned long vm_flags;
    /* AVL tree of VM areas per task, sorted by address */
    short vm_avl_height;
    struct vm_area_struct * vm_avl_left;
    struct vm_area_struct * vm_avl_right;

    struct vm_area_struct *vm_next_share;
    struct vm_area_struct **vm_pprev_share;

    struct vm_operations_struct * vm_ops;
    unsigned long vm_pgoff;           /* offset in PAGE_SIZE units, *not* PAGE_CACHE_SIZE */
    struct file * vm_file;
    unsigned long vm_raend;
    void * vm_private_data;          /* was vm_pte (shared mem) */
};

```

};

코드 302. vm_area_struct 구조체의 정의

위에서 정의한 mm_struct의 mmap, mmap_avl, mmap_cache에 연결된 구조체로 각각의 필드는 다시 아래와 같이 정의된다.

Field	Description
vm_mm	관련된 주소 공간의 디스크립터에 대한 포인터(mm_struct를 가리킨다.)
vm_start	연속된 선형주소 공간의 시작을 나타낸다.
vm_end	연속된 선형주소 공간의 마지막을 나타낸다.
vm_next	프로세스와 관련된 선형주소 공간의 리스트에 있는 다음번 디스크립터를 나타낸다.
vm_page_prot	선형 주소 공간의 보호(protection)을 위해서 사용된다.
vm_flags	선형 주소 공간의 상태를 나타내는 값이다.
vm_avl_height	AVL tree를 관리하기 위해서 사용되는 부분으로 AVL tree의 weight값으로 사용된다.
vm_avl_left	AVL tree에서 왼쪽에 있는 child node를 나타낸다.
vm_avl_right	AVL tree에서 오른쪽에 있는 child node를 나타낸다.
vm_next_share	공유 메모리 공간에 대한 다음번 vm_area_struct를 가리킨다.
vm_pprev_share	공유 메모리 공간에 대한 이전번의 vm_area_struct를 가리킨다.
vm_ops	선형주소 공간에 대한 연산 벡터 ¹¹⁶ 를 나타낸다.
vm_pgoff	선형주소 공간상에서 PAGE_OFFSET 단위의 offset을 나타낸다.
vm_file	선형주소 공간을 차지하고 있는 파일에 대한 파일 object 구조체를 나타낸다.
vm_raend	파일에 대한 read ahead의 마지막 번지에 해당하는 offset을 나타낸다.
vm_private_data	vm_area_struct에서 사용하는 private를 저장할 목적으로 사용한다.

표 37. vm_area_struct의 필드 정의

vm_area_struct 구조체의 flags값이 가질 수 있는 것으로는 아래와 같은 것이 있다. 각각은 현재 vm_area_struct의 상태를 나타내는 값이다.

Flag	Value	Description
VM_READ	0x00000001	이 가상 메모리 공간에 대해서 READ 접근이 가능하다.
VM_WRITE	0x00000002	이 가상 메모리 공간에 대해서 WRITE 접근이 가능하다.
VM_EXEC	0x00000004	이 가상 메모리 공간에 대해서 EXEC 접근이 가능하다.
VM_SHARED	0x00000008	이 가상 메모리 공간이 여러 프로세스에 의해서 공유되고 있다.
VM_MAYREAD	0x00000010	이 가상 메모리 공간은 READ 접근을 위해서 변경될 수 있다.
VM_MAYWRITE	0x00000020	이 가상 메모리 공간은 WRITE 접근을 위해서 변경될 수 있다.
VM_MAYEXEC	0x00000040	이 가상 메모리 공간은 EXEC 접근을 위해서 변경될 수 있다.
VM_MAYSHARE	0x00000080	이 가상 메모리 공간은 공유를 위해서 변경될 수 있다.
VM_GROWSDOWN	0x00000100	이 가상 메모리 공간은 하위 방향으로 성장(grow) 한다.
VM_GROWSUP	0x00000200	이 가상 메모리 공간은 상위 방향으로 성장(grow) 한다.
VM_SHM	0x00000400	이 가상 메모리 공간은 공유 메모리의 한 조각(segment)에

¹¹⁶ 일반적으로 vector라는 말을 사용해서 연산들의 루트를 나타낸다. 즉, 구조체에서 사용할 연산들이 된다.

		해당한다. (System V IPC에서 사용하는 공유 메모리이다.)
VM_DENYWRITE	0x00000800	이 가상 메모리 공간에 맵핑된 파일에 대한 WRITE는 에러를 유발할 것이다. 즉, WRITE를 거부한다.
VM_EXECUTABLE	0x00001000	이 가상 메모리 공간은 실행 가능한 코드를 가진 부분이다.
VM_LOCKED	0x00002000	이 가상 메모리 공간에 대해서 LOCK이 설정 되었다.
VM_IO	0x00004000	메모리 mapped I/O와 같은 곳에서 사용된다.
VM_SEQ_READ	0x00008000	응용 프로그램이 데이터를 순차적(sequentially)으로 접근한다.
VM_RAND_READ	0x00010000	응용 프로그램이 데이터를 임의적(randomly)으로 접근한다.
VM_DONTCOPY	0x00020000	fork()와 같은 시스템 콜에서 이 부분에 해당하는 가상 주소 공간은 copy하지 못하도록 한다.
VM_DONTEXPAND	0x00040000	mremap()과 같은 함수로 가상 주소 공간에 대한 확장을 블허한다.
VM_RESERVED	0x00080000	swap out으로 부터 이 부분의 가상 주소 공간에 대하여 unmap을 하지 못하도록 만든다.
VM_STASK_FLAGS	0x00000177	프로그램의 스택 영역을 나타내는 flag값이다.
VM_READHINTMASK	VM_SEQ_READ VM_RAND_READ	가상 주소 공간에 대해서 read를 어떻게 할 것인가를 알려주는 hint역할을 하는 mask이다. 이것과 flag을 AND연산을 해서 사용한다.(실제로 flag에 설정되지는 않는다.)

표 38. vm_area_struct의 flags에 들어가는 값

다시 아래와 같은 매크로가 이 flags필드에 대해서 정의되어 있다. 각각의 매크로는 이름이 의미하는 뜻과 같은 역할을 해주고 있다.

```
#define VM_ClearReadHint(v)          (v)->vm_flags &= ~VM_READHINTMASK
#define VM_NormalReadHint(v)         (!((v)->vm_flags & VM_READHINTMASK))
#define VM_SequentialReadHint(v)     ((v)->vm_flags & VM_SEQ_READ)
#define VM_RandomReadHint(v)        ((v)->vm_flags & VM_RAND_READ)
```

또한, vm_area_struct에 대한 연산은 아래와 같은 것들이 존재한다. 이를 각각은 해당 메모리에 대한 연산으로 적용될 것이다.

```
struct vm_operations_struct {
    void (*open)(struct vm_area_struct * area);
    void (*close)(struct vm_area_struct * area);
    struct page * (*nopage)(struct vm_area_struct * area, unsigned long address, int write_access);
};
```

코드 303. vm_area_struct에 대한 연산 구조체 정의

연산은 열기(open)와 닫기(close), 그리고, nopage등이 존재한다. 이중에서 nopage연산은 해당 페이지가 메모리에 없을 경우에 호출되는 함수로, 페이지를 디스크로 부터 읽어오는 역할을 수행할 것이다.

~/mm이하의 디렉토리를 살펴보면 크게 다음과 같은 두개의 vm_operations_struct에 대한 정의가 나온다. 각각은 filemap.c와 shmem.c에 정의되어 있으며, 파일 맵핑에 사용되는 vm_operations_struct와 공유 메모리에 사용되는 vm_operations_struct이다.

```
/* filemap.c에서 */
static struct vm_operations_struct file_shared_mmap = {
    nopage:           filemap_nopage,
```

```

};

static struct vm_operations_struct file_private_mmap = {
    nopage:           filemap_nopage,
};

/* shmem.c에서 */
static struct vm_operations_struct shmem_private_vm_ops = {
    nopage: shmem_nopage,
};

static struct vm_operations_struct shmem_shared_vm_ops = {
    nopage: shmem_nopage,
};

```

코드 304. filemap.c와 shmem.c에 정의된 vm_operations_struct 구조체

각각에서 정의하고 있는 연산에는 단지 `nopage` 연산에 대한 것만이 있다. 메모리상에 파일을 맵핑해서 사용하고 있던 곳에서 `page fault`가 발생하면, `filemap_nopage()` 함수가 수행될 것이며, 공유메모리 상에서 `page fault`가 발생한다면, `shmem_nopage()` 함수가 수행될 것이다.

이상에서 우린 프로세스의 주소 공간을 표현하기 위한 커널 데이터 구조를 보았다. 이젠 이들간의 관계를 머리속에 유지하기 위해서 그림을 그려보자. [그림37]은 위에서 설명한 데이터 구조를 연관시켜 보여준다.

이러한 데이터 구조는 뒤에서 계속 설명하는 함수들이, 그때 그때 필요할 때마다 생성과 해제를 반복할 것으로 기억해 두는 것이 좋을 것이다.

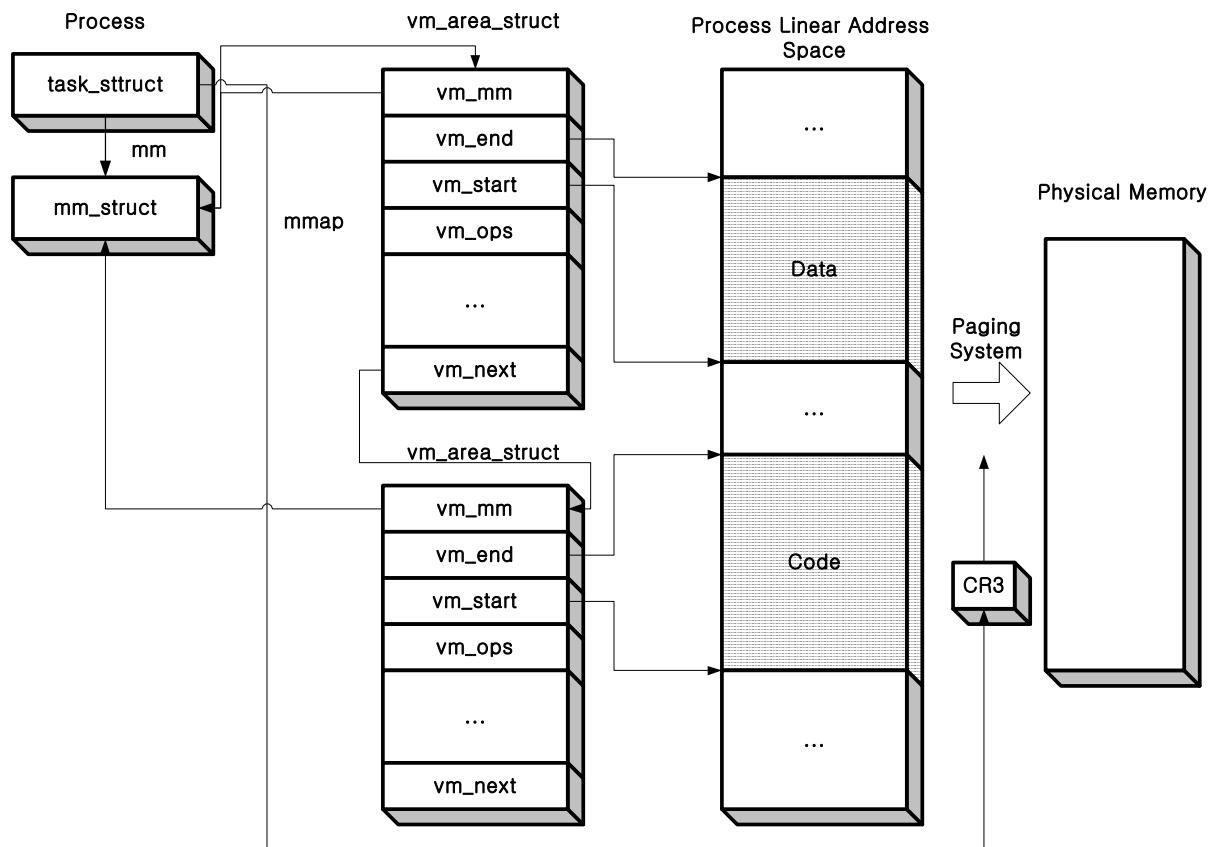


그림 44. 프로세스의 주소 공간

이전 실제적으로 프로세스에 대해서 메모리 할당과 메모리의 해제를 보도록 하자. 프로세스의 선형 주소 메모리 할당과 해제는 `do_mmap()` 함수와 `do_munmap()` 함수가 맡고 있다.

먼저 `do_mmap()` 함수는 현재 프로세스(`current`)의 선형 주소 공간상의 메모리 할당을 맡고 있다. 정의는 `~/include/linux/mm.h`에 아래같은 inline 함수로 되어 있다.

```
static inline unsigned long do_mmap(struct file *file, unsigned long addr,
                                    unsigned long len, unsigned long prot,
                                    unsigned long flag, unsigned long offset)
{
    unsigned long ret = -EINVAL;
    if ((offset + PAGE_ALIGN(len)) < offset)
        goto out;
    if (!(offset & ~PAGE_MASK))
        ret = do_mmap_pgoff(file, addr, len, prot, flag, offset >> PAGE_SHIFT);
out:
    return ret;
}
```

코드 305. `do_mmap()` 함수의 정의

`do_mmap()` 함수가 넘겨 받는 파라미터 값으로는 `file` 구조체와 주소, 할당하려는 공간의 크기를 가지는 `len`, 할당하는 공간에 대한 `protection`을 나타내는 `prot`, `flag`과 `offset`이 있다. 만약 `len`를 페이지 단위로 정렬했을 때, 이 값과 `offset`의 합이 원래의 `offset`과의 합보다 작은 경우에는 error로 `-EINVAL`을 돌려준다. 이제 `offset`에서 `PAGE_MASK`(=0xFFFFF000)을 NOT한 값(0x00000FFF)에 AND시켜서 어떤 값이 있다면¹¹⁷, `do_mmap_pgoff()` 함수를 넘겨받은 파라미터 값과 `offset`을 `PAGE_SHIFT`(=12) 오른쪽으로 shift한 값을 넘겨주어 호출한다. 그렇지 않다면, 다시 error로 `-EINVAL`을 복귀값으로 넘겨준다.

```
unsigned long do_mmap_pgoff(struct file * file, unsigned long addr, unsigned long len,
                            unsigned long prot, unsigned long flags, unsigned long pgoff)
{
    struct mm_struct * mm = current->mm;
    struct vm_area_struct * vma;
    int correct_wcount = 0;
    int error;

    if (file && (!file->f_op || !file->f_op->mmap))
        return -ENODEV;
    if ((len = PAGE_ALIGN(len)) == 0)
        return addr;
    if (len > TASK_SIZE || addr > TASK_SIZE-len)
        return -EINVAL;
    if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
        return -EINVAL;
    if (mm->map_count > MAX_MAP_COUNT)
        return -ENOMEM;
    /* mlock MCL_FUTURE? */
    if (mm->def_flags & VM_LOCKED) {
        unsigned long locked = mm->locked_vm << PAGE_SHIFT;
        locked += len;
        if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
            return -EAGAIN;
    }
}
```

¹¹⁷즉, 한 페이지보다 작은 메모리를 할당하려고 한다면.

코드 306. do_mmap_pgoff()함수

do_mmap_pgoff()함수가 넘겨받은 값은 앞에서 do_mmap()함수가 넘겨받는 값과 하나만 차이가 날뿐 동일하다. 차이가 나는 점은 pgoff이다. 이 값이 do_mmap()함수에서는 페이지의 갯수를 나타내는 값이 된다. 먼저 파일 연산자에 mmap()함수가 정의되어 있는지를 확인한다. 없다면 -ENODEV를 돌려준다. 할당받으려는 메모리 영역의 길이(len)을 페이지 단위로 정렬(PAGE_ALIGN()¹¹⁸)해서 0인지 확인하고, 만약 그렇다면 시작 address를 돌려준다. 만약 len이 TASK_SIZE(=0xC0000000)보다 크거나, addr이 TASK_SIZE - len값 보다 클 경우에는 -EINVAL을 돌려준다. 즉, 너무 큰 영역을 할당 받으려는 시도를 예외로 처리한다. 다시 pgoff와 len을 PAGE_SHIFT만큼 우측으로 SHIFT한 값을 더한 후, 이 값이 pgoff보다 작은지 계산한다. 만약 더 작다면 다시 -EINVAL을 돌려준다. 이것은 offset이 overflow를 일으키는지 확인하는 부분이다.

최대 맵 카운트(MAX_MAP_COUNT=65536) 보다 더 많은 맵핑을 가지고 있다면, -ENOMEM을 돌려주어 너무 많은 메모리 mapping을 하지 못하도록 만든다. 이전 현재 프로세스의 mm구조체에 설정된 def_flags가 VM_LOCKED를 설정하고 있는지 확인해서, 만약 설정이 되었다면, lock된 가상 메모리가 현재 프로세스가 가진 자원 사용 한계(rlim[])과 비교한다. 넘어선다면 -EAGAIN을 복귀값으로 돌려준다.

```

if (file != NULL) {
    switch (flags & MAP_TYPE) {
        case MAP_SHARED:
            if ((prot & PROT_WRITE) && !(file->f_mode & FMODE_WRITE))
                return -EACCES;
            if (IS_APPEND(file->f_dentry->d_inode) && (file->f_mode & FMODE_WRITE))
                return -EACCES;
            if (locks_verify_locked(file->f_dentry->d_inode))
                return -EAGAIN;
            /* fall through */
        case MAP_PRIVATE:
            if (!(file->f_mode & FMODE_READ))
                return -EACCES;
            break;
        default:
            return -EINVAL;
    }
}

```

코드 307. do_mmap_pgoff()함수 – continued

파일 object가 NULL이 아닐 경우에는 파일에 대한 메모리 맵핑을 처리한다. 넘겨받은 flags에 MAP_SHARED, MAP_PRIVATE가 설정되었나에 따라서, 해당하는 접근(access) 허가 모드를 살펴본다. 넘겨받은 prot이 PROT_WRITE가 설정되었고, 파일 object의 f_mode에 FMODE_WRITE가 설정되지 않았다면 허가 사항이 틀리므로 -EACCESS를 돌려준다. 다시 파일 object의 inode object를 찾아서 APPEND모드가 설정되어 있는지를 확인하고, 다시 파일 object의 f_mode에 FMODE_WRITE가 설정되어 있는지 확인한다. 이럴 경우에는 APPEND ONLY 파일에 write를 허가하는 것이기에 다시 -EACCESS를 돌려준다. 이전 inode object에 lock이 설정되어 있는지 확인한다. 있다면 -EAGAIN을 돌려준다. 아래 부분은 MAP_PRIVATE와 같은 실행을 보인다. 파일 object의 f_mode에 FMODE_READ가 설정되어 있지 않다면 다시 -EACCESS를 돌려준다. 즉, WRITE가 설정되어 있다면 당연히 READ도 설정되어야 할 것이다. 혹은 MAP_PRIVATE라고 할지라도 READ는 가능한다.

```

if (flags & MAP_FIXED) {
    if (addr & ~PAGE_MASK)
        return -EINVAL;
} else {
    addr = get_unmapped_area(addr, len);
}

```

¹¹⁸ #define PAGE_ALIGN(addr) (((addr) + PAGE_SIZE - 1) & PAGE_MASK) : PAGE_MASK(=0xFFFFF000)

```

    if (!addr)
        return -ENOMEM;
    }
    vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!vma)
        return -ENOMEM;

    vma->vm_mm = mm;
    vma->vm_start = addr;
    vma->vm_end = addr + len;
    vma->vm_flags = vm_flags(prot,flags) | mm->def_flags;

```

코드 308. do_mmap_pgoff()함수 – continued

flags에 MAP_FIXED가 설정된 경우에는 addr를 ~PAGE_MASK(0x00000FFF)로 AND해서 값이 있다면 -EINVAL을 돌려준다. 즉, 고정되었을 경우에는 addr이 페이지 단위로 맞아떨어져야 한다. 그렇지 않다면, get_unmapped_area()함수를 호출해서, 새로운 맵핑이 되지 않은 메모리 영역의 주소를 취한다. 이전 vm_area_struct를 하나 할당받기 위해서 kmem_cache_alloc()함수를 호출하며, 넘겨주는 값으로는 vm_area_struct를 생성하기 위한 캐시(vm_area_cachep)와 SLAB_KERNEL이다. 즉, 커널에서 사용할 데이터 구조를 위한 메모리 공간을 할당한다. 만약 할당할 수 없다면 -ENOMEM을 돌려준다. 할당받은 vm_area_struct를 초기화하자. vm_mm에는 mm을, vm_start에는 addr을, vm_end에는 addr + len을, vm_flags에는 vm_flags()를 호출해서 protection과 flags를 가지고 조합한 것을 vm_flags로 매핑해서 다시 mm의 def_flags와 OR시켜서 놀는다.

```

if (file) {
    VM_ClearReadHint(vma);
    vma->vm_raend = 0;
    if (file->f_mode & FMODE_READ)
        vma->vm_flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
    if (flags & MAP_SHARED) {
        vma->vm_flags |= VM_SHARED | VM_MAYSHARE;
        if (!(file->f_mode & FMODE_WRITE))
            vma->vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
    }
} else {
    vma->vm_flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
    if (flags & MAP_SHARED)
        vma->vm_flags |= VM_SHARED | VM_MAYSHARE;
}
vma->vm_page_prot = protection_map[vma->vm_flags & 0x0f];
vma->vm_ops = NULL;
vma->vm_pgoff = pgoff;
vma->vm_file = NULL;
vma->vm_private_data = NULL;

```

코드 309. do_mmap_pgoff()함수 – continued

관련된 파일 object가 있는 경우에는 if()이하의 절을 실행한다. vm_area_struct의 vm_flags에 설정되었을지도모를 VM_SEQ_READ와 VM_RAND_READ를 지우고(VM_ClearReadHint()), vm_raend에는 0을 설정한다. 즉, 아직 read ahead가 되지 않았다. 맵핑된 파일이 FMODE_READ가 설정되어 있다면, vm_flags에는 VM_MAYREAD와 VM_MAYWRITE, VM_MAYEXEC를 OR시키고, flags에 MAP_SHARED되어 있다면, 다시 vm_flags에 VM_SHARED와 VM_MAYSHARE를 설정해서 공유 영역에 대한 설정을 해준다. 만약 파일 object의 f_mode에 FMODE_WRITE가 설정되지 않았다면, 위에서 한 일종에서 VM_MAYWRITE와 VM_SHARED를 vm_flags에서 지운다. 매핑된 파일 object가 없다면, vm_flags에는 VM_MAYREAD와 VM_MAYWRITE, VM_MAYEXEC가 설정되고, 다시 MAP_SHARED에 따라서, vm_flags에 VM_SHARED와 VM_MAYSHARE가 설정된다.

이전 vm_page_prot 및 vm_ops, vm_pgoff, vm_file, vm_private_data에 해당하는 값으로 초기화 한다. protection_map[]은 X86하드웨어에서 제공하는 protection 메커니즘이 리눅스에서 쓰는 protection 메커니즘보다 못하기 때문에 둘간에 어떤 식으로 맵핑을 할지를 결정하는 배열이다. 정의는 ~/mm/map.c에 아래와 같이 나와있다.

```
pgprot_t protection_map[16] = {
    __P000, __P001, __P010, __P011, __P100, __P101, __P110, __P111,
    __S000, __S001, __S010, __S011, __S100, __S101, __S110, __S111
};
```

배열의 구성은 PROT_NONE, PROT_READ, PROT_WRITE, PROT_EXEC에 따라서, MAP_SHARED와 MAP_PRIVATE각각에 대해서 r/w/x(read/write/execute)를 할 수 있는지를 맵핑한다. 간단히 보면, MAP_SHARED인 경우 PROT_NONE은 r/w/x가 전부 불가이며, PROT_READ는 w가 불가(r/x는 가능), PROT_WRITE는 r/w/x가 전부 가능, PROT_EXEC는 r/x가 가능하다(w는 불가). MAP_PRIVATE의 경우에는 PROT_NONE이 r/w/x가 불가이며, PROT_READ는 r/x가 가능(w는 불가), PROT_WRITE는 r/x가 가능(w는 copy가 일어남), PROT_EXEC는 r/x가 가능(w는 불가)로 맵핑이 일어난다.

잠시 vm_flags에 대해서 사용되는 매크로 들의 정의를 보도록 하자. 정의는 ~/include/linux/mm.h에 아래와 같이 되어있다.

```
#define VM_READHINTMASK          (VM_SEQ_READ | VM_RAND_READ)
#define VM_ClearReadHint(v)        (v)->vm_flags &= ~VM_READHINTMASK
#define VM_NormalReadHint(v)       (!((v)->vm_flags & VM_READHINTMASK))
#define VM_SequentialReadHint(v)   ((v)->vm_flags & VM_SEQ_READ)
#define VM_RandomReadHint(v)       ((v)->vm_flags & VM_RAND_READ)
```

위에서 정의한 매크로의 이름에서 어떤 일을 확인할 수 있을 것이다. VM_ClearReadHint는 vm_flags에서 VM_SEQ_READ와 VM_RAND_READ를 지우며, VM_NormalReadHint()는 VM_SEQ_READ나 VM_RAND_READ가 vm_flags에 되지 않았는지를 검사하며, VM_SequentialReadHint는 VM_SEQ_READ를, VM_RandomReadHint()는 VM_RAND_READ가 설정되었는지를 검사한다.

```
error = -ENOMEM;
if (do_munmap(mm, addr, len))
    goto free_vma;
if ((mm->total_vm << PAGE_SHIFT) + len
    > current->rlim[RLIMIT_AS].rlim_cur)
    goto free_vma;
/* Private writable mapping? Check memory availability.. */
if ((vma->vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
    !(flags & MAP_NORESERVE) &&
    !vm_enough_memory(len >> PAGE_SHIFT))
    goto free_vma;
if (file) {
    if (vma->vm_flags & VM_DENYWRITE) {
        error = deny_write_access(file);
        if (error)
            goto free_vma;
        correct_wcount = 1;
    }
    vma->vm_file = file;
    get_file(file);
    error = file->f_op->mmap(file, vma);
    if (error)
        goto unmap_and_free_vma;
} else if (flags & MAP_SHARED) {
```

```

error = shmem_zero_setup(vma);
if (error)
    goto free_vma;
}

```

코드 310. do_mmap_pgoff() 함수 – continued

이전 예전에 이미 있을 수 있는 메모리 맵을 지운다(do_munmap()). 만약 잘못된 에러를 발생시킨다면, free_vma로 제어를 옮겨서 할당받은 vm_area_struct 구조체를 지운다. 맵핑하려는 메모리의 길이와 mm_struct 구조체에 있는 total_vm 필드를 페이지 크기와 곱(<<PAGE_SHIFT)해서 얻은 값이, 현재 프로세스가 가진 rlim[]보다 크다면 너무 많은 메모리를 매핑하려고 시도했기에 free_vma로 제어를 옮겨서 할당받았던 vm_area_struct 구조체를 다시 해제한다.

또한 vm_area_struct의 vm_flags가 VM_SHARED나 VM_WRITE와 AND시켜서, 이 값이 VM_WRITE와 같고, flags에 MAP_NORESERVE가 설정되지 않았으며, 충분한 가상 메모리가 존재하지 않는다면(vm_enough_memory()), 제어는 역시 free_vma로 옮긴다. 즉, 개인적인(private) writable 맵핑에 대해서 메모리가 남았는지를 확인하는 것이다.

만약 맵핑과 관련된 파일 object가 있다면, vm_flags에 VM_DENYWRITE가 설정되어 있다면 파일 object에 write를 거부하도록 만들고(deny_write_access()). deny_write_access() 함수의 호출중에 에러가 있었다면, 제어는 다시 free_vma로 옮긴다. 그리고, correct_wcount는 1로 설정한다. 즉, write count의 조정값이다. vm_file에는 파일 object(file)을 두고, 파일에 대한 획득(get_file())¹¹⁹을 지시한다. 만약 파일 object가 mmap() 연산을 정의하고 있다면 mmap() 연산을 수행한다. 수행중 에러가 발생한다면 제어는 unmap_and_free_vma로 옮긴다.

관련된 파일 object는 없지만 flags에 MAP_SHARED가 설정된 경우라면 할당 받은 vm_area_struct 구조체를 shmem_zero_setup()의 인자로 주고 호출해서 공유 메모리에 대한 초기화(0으로)를 수행한다. 역시 에러가 있다면, 제어는 free_vma로 옮길 것이다.

```

flags = vma->vm_flags;
addr = vma->vm_start;
insert_vm_struct(mm, vma);
if (correct_wcount)
    atomic_inc(&file->f_dentry->d_inode->i_writecount);

mm->total_vm += len >> PAGE_SHIFT;
if (flags & VM_LOCKED) {
    mm->locked_vm += len >> PAGE_SHIFT;
    make_pages_present(addr, addr + len);
}
return addr;
unmap_and_free_vma:
if (correct_wcount)
    atomic_inc(&file->f_dentry->d_inode->i_writecount);
vma->vm_file = NULL;
fput(file);
flush_cache_range(mm, vma->vm_start, vma->vm_end);
zap_page_range(mm, vma->vm_start, vma->vm_end - vma->vm_start);
flush_tlb_range(mm, vma->vm_start, vma->vm_end);
free_vma:
kmem_cache_free(vm_area_cachep, vma);
return error;
}

```

코드 311. do_mmap_pgoff() 함수 – continued

¹¹⁹ #define get_file(x) atomic_inc(&(x)->f_count)

이전 flags에 vm_area_struct 구조체에 설정했던 vm_flags를 주고, addr에는 vm_start를 준다. vm_area_struct 구조체의 초기화가 끝났으므로 insert_vm_struct() 함수를 호출해서 mm_struct의 vm_area_struct로 삽입한다. current_wcount가 설정되었다면, 관련된 파일 object의 inode object에서 iwrite_count 필드를 증가시킨다(atomic_inc()).

전체 맵핑된 크기(mm_struct의 total_vm)에는 이제 len>>PAGE_SHIFT가 더해질 것이며, flags에 VM_LOCKED가 설정되었다면, mm_struct의 locked_vm에도 len>>PAGE_SHIFT가 더해질 것이다. 또한 해당 페이지에는 present bit이 설정된다(make_pages_present()). 복귀값은 맵핑이 시작되는 번지이다(addr). unmap_and_free_vma에서는 관련된 파일 object에서 연산이 오류가 있는 경우이므로 iwritecount를 증가시키고, vm_area_struct의 file에는 NULL을, 관련된 파일 object는 버린다(fput()). 맵핑이 되어 있는 영역에 대한 cache는 없어지고(flush_cache_range()), 그 영역에 속하는 페이지도 버려진다(zap_page_range()). 마지막으로 가상 메모리 영역에 대한 TLB(Translation Lookaside Buffer)도 없어지게 되며(flush_tlb_range()). 할당 받았던 vm_area_struct는 커널의 메모리에서 제거된다(kmem_cache_free()). 복귀값은 에러 코드가 될 것이다.

이상에서 보았던 do_mmap() 함수의 역에 해당하는 것이 do_munmap() 함수이다. ~/mm/mmap.c에 아래와 같이 나온다.

```
int do_munmap(struct mm_struct *mm, unsigned long addr, size_t len)
{
    struct vm_area_struct *mpnt, *prev, **npp, *free, *extra;

    if ((addr & ~PAGE_MASK) || addr > TASK_SIZE || len > TASK_SIZE - addr)
        return -EINVAL;
    if ((len = PAGE_ALIGN(len)) == 0)
        return -EINVAL;
    mpnt = find_vma_prev(mm, addr, &prev);
    if (!mpnt)
        return 0;
    if (mpnt->vm_start >= addr + len)
        return 0;
    if ((mpnt->vm_start < addr && mpnt->vm_end > addr + len)
        && mm->map_count >= MAX_MAP_COUNT)
        return -ENOMEM;
    extra = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!extra)
        return -ENOMEM;
    npp = (prev ? &prev->vm_next : &mm->mmap);
    free = NULL;
    spin_lock(&mm->page_table_lock);
    for ( ; mpnt && mpnt->vm_start < addr + len; mpnt = *npp) {
        *npp = mpnt->vm_next;
        mpnt->vm_next = free;
        free = mpnt;
        if (mm->mmap_avl)
            avl_remove(mpnt, &mm->mmap_avl);
    }
    mm->mmap_cache = NULL; /* Kill the cache. */
    spin_unlock(&mm->page_table_lock);
}
```

코드 312. do_munmap() 함수

do_munmap() 함수가 넘겨받는 것은 프로세스의 mm_struct와 do_munmap()을 실행할 시작 주소(addr), 그리고 그 크기(len)이다. 만약 ~PAGE_MASK(=0x00000FFF)와 AND를 해서 어떤 값이 있거나(즉, 페이지 단위의 시작 주소가 아니거나), 혹은 addr이 TASK_SIZE(=0xC0000000)보다 크거나, len이 TASK_SIZE - addr보다 큰 값을 가진다면 -EINVAL을 돌려준다. len을 페이지 단위로 정렬(PAGE_ALIGN())해서 0이 나오면 다시 -EINVAL을 돌려준다.

`find_vma_prev()`는 주어진 주소에 대해서 이 주소보다 앞서 있는 메모리 영역의 디스크립터의 주소를 돌려준다. 즉, `mpnt`는 `addr`에 앞서서 이미 존재하는 `vm_area_struct`에 대한 포인터이다. 만약 이 값이 없다면, 0을 돌려준다. 그리고, 이렇게 찾은 `mpnt`의 `vm_start`가 `addr + len`보다 크다면, 잘못 찾은 것이 되므로 다시 0을 돌려준다. 또한 이렇게 찾은 `vm_area_struct`의 `vm_start`가 `addr`보다 작고, `vm_end`가 `addr + len`보다 크게 되며, `mm_struct`의 `map_count`가 `MAX_MAP_COUNT`보다 크다면, 이미 맵핑된 영역에 새로이 맵핑을 하려고 하지만, 이미 그 최대 맵핑 값을 초과하기에 `-ENOMEM`을 에러 값으로 돌려준다.

위에서 대부분의 검사(check)를 맞쳤다. 이젠 메모리 맵핑을 처리하기 위해서 `vm_area_struct`를 slab에서 하나 할당 받는다(`kmem_cache_alloc()`). 할당 받을 수 없다면, 역시 `-ENOMEM`을 에러 값으로 돌려줄 것이다.

`npp`는 `find_vma_prev()`에서 `prev`에 어떤 값을 받았을 경우에는 `prev->vm_next`의 주소를, 그렇지 않을 경우에는 `mm_struct`의 `mmap`의 주소를 받는다. 즉, `npp`는 `prev`가 있는 경우에는 있는 `vm_area_struct`의 주소를 가지는 곳을 가르키던가, 아니면 `mm_struct`의 첫번째 `vm_area_struct`의 `vm_next`필드를 가지는 `mmap`의 주소를 가질 것이다. 이젠 페이지 테이블에 대한 lock을 설정한다(`spin_lock()`).

`for()` loop는 메모리 맵핑된 리스트에서, `vm_area_struct` 구조체의 `vm_start`가 `addr + len`을 만족하는 `vm_area_struct`를 찾는다. 만약 해당하는 `vm_area_struct` 구조체를 찾았다면, 이것을 `free`로 놓고, 해당하는 `mm_struct`의 `mmap_avl0` 값을 가지는 경우에는 AVL Tree에서 `vm_area_struct`를 삭제한다(`avl_remove()`). `mm_struct`의 `mmap_cache`는 더 이상의 유효하지 않다는 것을 나타내기 위해서 `NULL`로 설정하고, 페이지 테이블에 대한 lock를 해제한다(`spin_unlock()`).

```

while ((mpnt = free) != NULL) {
    unsigned long st, end, size;
    struct file *file = NULL;

    free = free->vm_next;
    st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
    end = addr+len;
    end = end > mpnt->vm_end ? mpnt->vm_end : end;
    size = end - st;
    if (mpnt->vm_flags & VM_DENYWRITE &&
        (st != mpnt->vm_start || end != mpnt->vm_end) &&
        (file = mpnt->vm_file) != NULL) {
        atomic_dec(&file->f_dentry->d_inode->i_writecount);
    }
    remove_shared_vm_struct(mpnt);
    mm->map_count--;
    flush_cache_range(mm, st, end);
    zap_page_range(mm, st, size);
    flush_tlb_range(mm, st, end);
    extra = unmap_fixup(mm, mpnt, st, size, extra);
    if (file)
        atomic_inc(&file->f_dentry->d_inode->i_writecount);
}
if (extra)
    kmem_cache_free(vm_area_cachep, extra);
free_pgtables(mm, prev, addr, addr+len);
return 0;
}

```

코드 313. `do_munmap()` 함수 – continued

이제 해제(free)시킬 영역에 대한 `vm_area_struct`를 찾았다. `mpnt`에 `free`를 넣어서 `NULL`이 아닐 동안 `while()` loop를 실행한다. `free`에는 `vm_next`를 넣어서 다음번 loop를 대비하고, `st`는 시작주소를 가지는 해제할 영역의 시작 주소를 가지는 값이므로, `addr`가 `vm_start`보다 작을 때는 `vm_start`의 주소를 그래로 사용하고, 그렇지 않을 때는 `addr` 값을 지니도록 한다. `end`는 해제할 영역의 마지막 주소로서, `addr +`

len값으로 초기화 하고, end가 vm_end보다 클 경우에는 vm_end로 두고, 작을 경우에는 end값을 그대로 사용한다. 이젠 해제할 영역의 크기를 size로 나타내기 위해서 end에서 st를 뺀다.

만약 vm_flags에 VM_DENYWRITE가 설정되어 있으며, st와 end가 vm_start와 vm_end와 같지 않으며, vm_file에 NULL로 설정되지 않았을 경우에는 파일 object가 가지는 inode의 i_writecount를 하나 감소시킨다(atomic_dec()). 만약 해제하려는 가상 주소 공간이 공유되고 있을지도 모르므로, remove_shared_vm_struct()함수를 해제하려는 vm_area_struct를 넘겨서 호출한다. mm_struct에는 맵핑이 하나 감소되며(map_count--), mm_struct의 st와 end사이에 있는 캐쉬된 영역을 비운다(flush_cache_range()). zap_page_range()를 호출해서 해당 영역에 있는 사용자 페이지들을 삭제하고, flush_tlb_range()를 호출해서 TLB(Translation Lookaside Buffer)를 비운다. unmap_fixup()함수는 메모리 맵핑을 고치는(fix) 역할을 하는 함수이다. 최종적으로 돌려받는 값은 지우게 될 vm_area_struct 구조체이며, 이것이 파일과 관련된 맵핑인 경우에는 위에서 inode에 대한 iwrite_count를 감소 시켜주었으므로 다시 증가 시켜준다.

이전 vm_area_struct를 커널 메모리 영역에서 해제하고(kmem_cache_free()), 프로세스의 페이지 테이블을 고치는 일이다(free_ptables()). 이것을 마쳤다면, 제대로 연산을 모두 수행한 것이며, 복귀값으로는 0을 가진다.

```
asmlinkage long sys_munmap(unsigned long addr, size_t len)
{
    int ret;
    struct mm_struct *mm = current->mm;

    down(&mm->mmap_sem);
    ret = do_munmap(mm, addr, len);
    up(&mm->mmap_sem);
    return ret;
}
```

코드 314. sys_munmap() 시스템 콜

sys_munmap()은 do_munmap()에 대한 시스템 콜 인터페이스이다. 정의는 ~/mm/map.c에 나와 있다. 단순히 현재 프로세스의 mm_struct 구조체의 세마포어를 획득(down())한 다음 do_munmap()을 호출해서 처리한다. 연산을 마치면 다시 획득한 mm_struct 구조체의 세마포어를 놓고(up()), do_munmap() 함수의 호출 결과를 돌려준다.

sys_brk()시스템 콜은 다음과 같이 ~/mm/map.c에 정의되어 있다. sys_brk() 시스템 콜은 do_brk() 커널 함수에 대한 사용자 프로세스 인터페이스이다. 하는 일은 현재 프로세스의 heap 크기를 변경하는 일을 한다. 즉, 새로이 데이터 세그먼트에서 할당할 수 있는 메모리 양을 늘여주는 일이다.

```
asmlinkage unsigned long sys_brk(unsigned long brk)
{
    unsigned long rlim, retval;
    unsigned long newbrk, oldbrk;
    struct mm_struct *mm = current->mm;

    down(&mm->mmap_sem);
    if (brk < mm->end_code)
        goto out;
    newbrk = PAGE_ALIGN(brk);
    oldbrk = PAGE_ALIGN(mm->brk);
    if (oldbrk == newbrk)
        goto set_brk;
    if (brk <= mm->brk) {
        if (!do_munmap(mm, newbrk, oldbrk-newbrk))
            goto set_brk;
        goto out;
    }
}
```

```

rlim = current->rlim[RLIMIT_DATA].rlim_cur;
if (rlim < RLIM_INFINITY && brk - mm->start_data > rlim)
    goto out;
if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
    goto out;
if (!vm_enough_memory((newbrk-oldbrk) >> PAGE_SHIFT))
    goto out;
if (do_brk(oldbrk, newbrk-oldbrk) != oldbrk)
    goto out;
set_brk:
mm->brk = brk;
out:
retval = mm->brk;
up(&mm->mmap_sem);
return retval;
}

```

코드 315. sys_brk() 시스템 콜의 정의

먼저 현재 프로세스의 mm_struct의 세마포어(mmap_sem)를 획득한다(down()). 넘겨받은 brk값이 프로세스의 코드의 끝을 나타내는 부분(end_code)보다 작은 값을 가진다면, out으로 제어를 옮긴다. brk를 페이지 단위로 정렬한 후(PAGE_ALIGN()), 이를 newbrk에 넣는다. 원래있는 mm_struct의 brk는 페이지 단위로 정렬해서 oldbrk에 둔다. 만약 이 둘의 값이 같다면, set_brk로 제어를 옮긴다.

만약 이전에 있던 mm_struct의 brk보다 넘겨받은 brk가 작다면, 이는 메모리를 줄이는 요청이 되므로 do_munmap()함수에 mm_struct와 newbrk, 그리고, 줄이는 양(oldbrk-newbrk)를 넘겨주어 호출하고, 호출의 결과에 따라서, set_brk나 out으로 제어를 옮긴다.

현재 프로세스가 가진 자원 사용 한계(rlim[])를 확인해서 이 값을 rlim으로 두고, 새로 할당 받을려는 영역이 이 rlim을 넘어서는지 확인한다. 만약 넘어선다면 할당요구에 문제가 있으므로 제어를 out으로 옮긴다. 이미 존재하는 메모리 맵핑에 대해서 혹시 교차(intersection)되는 부분이 있는지 확인(find_vma_intersection())한다. 있다면 다시 에러가 되므로 제어를 out으로 옮긴다.

이제 남은 것은 충분한 메모리가 있는가이다(vm_enough_memory()). 없다면 다시 제어를 out으로 옮긴다. 메모리 할당과 관련된 모든 부분에 대한 검사가 완료가 되었으므로 이젠 실제적으로 do_brk()함수를 호출해서 메모리를 할당 받는 것이다(do_brk()). 문제가 있다면 다시 제어는 out으로 옮겨진다. set_brk에서는 mm_struct가 어디서부터 다시 할당을 제게 할지를 나타내는 brk필드를 넘겨받은 brk로 설정하고, out에서는 복귀 값을 넘겨받은 brk로 설정한 후, 획득했던 세마포어를 해제(up())하고 곧바로 돌아간다. 남은 것은 모두 do_brk()에서 어떤 처리가 일어나는지를 보는 것이다.

do_brk()함수는 아래와 같이 ~mm/map.c에 정의되어 있다. do_brk()함수는 do_mmap()함수의 축소판이다. 이 함수는 단순히 anonymous 맵¹²⁰들만을 다룬다.

```

unsigned long do_brk(unsigned long addr, unsigned long len)
{
    struct mm_struct * mm = current->mm;
    struct vm_area_struct * vma;
    unsigned long flags, retval;

    len = PAGE_ALIGN(len);
    if (!len)
        return addr;
    if (mm->def_flags & VM_LOCKED) {
        unsigned long locked = mm->locked_vm << PAGE_SHIFT;
        locked += len;
        if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)

```

¹²⁰ 할당된 메모리와 관련된 파일 object가 없는 경우를 anonymous mapping이라고 한다.

```

        return -EAGAIN;
    }
    retval = do_munmap(mm, addr, len);
    if (retval != 0)
        return retval;
    if ((mm->total_vm << PAGE_SHIFT) + len
        > current->rlim[RLIMIT_AS].rlim_cur)
        return -ENOMEM;
    if (mm->map_count > MAX_MAP_COUNT)
        return -ENOMEM;
    if (!vm_enough_memory(len >> PAGE_SHIFT))
        return -ENOMEM;
    flags = vm_flags (PROT_READ|PROT_WRITE|PROT_EXEC,
                      MAP_FIXED|MAP_PRIVATE) | mm->def_flags;
    flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
    if (addr) {
        struct vm_area_struct * vma = find_vma(mm, addr-1);
        if (vma && vma->vm_end == addr && !vma->vm_file &&
            vma->vm_flags == flags) {
            vma->vm_end = addr + len;
            goto out;
        }
    }
}

```

코드 316. do_brk() 함수

do_brk() 함수가 넘겨받는 파라미터 값은 할당될 메모리의 시작 번지와 메모리의 길이이다. 현재 프로세스의 mm_struct를 mm에 넣어서 사용한다. 길이는 페이지 단위로 정렬(PAGE_ALIGN())되며, 만약 이렇게 했을 경우 0을 길이가 가길 때는 단순히 시작 번지의 주소를 돌려준다. 즉, 메모리 할당이 이전에 이미 할당 받은 페이지를 사용하게 될 것이다. 만약 mm의 def_flags에 VM_LOCKED bit가 설정된 경우에는 현재 프로세스의 lock된 메모리 영역이 자원 한계를 넘어서는지 확인하고, 넘어선다면 -EAGAIN을 돌려준다.

이전 예전에 할당받은 메모리 영역을 지운다(do_munmap()). do_munmap()의 호출 결과가 0이 아닌 경우에는 에러가 있는 경우이므로 호출 결과를 복귀값으로 두고 돌아간다(return retval). 이전에 할당 받은 메모리 영역을 해제했으므로 다시 각종 검사(rlim[], map_count, vm_enough_memory())를 실행하고, mm의 def_flags에 PROT_READ, PROT_WRITE, PROT_EXEC, MAP_FIXED, MAP_PRIVATE를 OR시켜서 flags의 값으로 둔다. 여기서 vm_flags() 함수는 inline으로 정의된 것으로 PROT_XXX와 MAP_XXX를 VM_XXX로 변환하는 일을 한다. 다시 이렇게 생성된 flags에 VM_MAYREAD, VM_MAYWRITE, VM_MAYEXEC를 설정한다.

이전 메모리 영역을 확장이 가능한지를 확인하는 일이다. addr이 어떤 값을 가진다면, addr에 해당하는 영역의 vm_area_struct를 찾고(find_vma()), 찾은 vm_area_struct의 vm_end가 addr과 같고, 관련된 file object가 없으며, vm_flags 필드가 flags과 같은지 확인한다. 같다면 vm_area_struct의 vm_end 필드에 addr+len 값을 넣고 제어를 out으로 옮긴다. 즉, 이미 존재하는 vm_area_struct의 크기만을 변경하게 된다.

```

vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!vma)
    return -ENOMEM;
vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = flags;
vma->vm_page_prot = protection_map[flags & 0x0f];
vma->vm_ops = NULL;
vma->vm_pgoff = 0;
vma->vm_file = NULL;
vma->vm_private_data = NULL;

```

```

insert_vm_struct(mm, vma);
out:
    mm->total_vm += len >> PAGE_SHIFT;
    if (flags & VM_LOCKED) {
        mm->locked_vm += len >> PAGE_SHIFT;
        make_pages_present(addr, addr + len);
    }
    return addr;
}

```

코드 317. do_brk() 함수 – continued

이전 anonymous 맵핑에 대한 vm_area_struct 구조체를 slab에서 할당 받는다(kmem_cache_alloc()). 이는 커널이 관리하는 자료 구조체이다. 만약 할당받을 수 없다면, -ENOMEM을 돌려준다. 할당받은 vm_area_struct는 vma가 되며, 이 변수의 각 필드에 대한 초기화를 실시한다. vm_mm에는 프로세스의 mm_struct를, vm_start에는 addr을, vm_end에는 addr+len을, vm_flags에는 앞에서 만들어준 flags를, vm_page_prot에는 protection_map[flags & 0x0F]를, vm_ops와 vm_file, vm_private_data에는 NULL을, vm_pgoff에는 0을 각각 넣는다.

out이 하는 mm_struct 구조체의 필드를 업데이트 하는 부분이다. total_vm에는 len을 페이지 단위로 만들어서 더하고, VM_LOCKED이 flags에 설정되었다면, locked_vm에 len을 페이지 단위로 만들어서 넣는다. 이전 새로이 할당 받은 메모리 영역에 페이지가 존재한다고 만들어준다(make_pages_present()). 복귀값은 새로이 할당 받은 메모리 영역의 시작 번지이다.

4.12. 페이지 오류(fault)의 처리

페이지 풀트(fault)는 다음과 같은 경우에 발생한다. 즉, 프로세스가 자신의 주소 공간에는 있지만, 메모리에 존재하지는 않는 페이지에 대한 접근을 시도한 경우와, 올바르지 않은 주소 공간에 대한 접근을 했을 경우이다. 전자는 합당한 연산이므로 커널은 해당 페이지를 디스크로 부터 읽어와서 물리적인 페이지를 할당받아서 채워주게 되며, 후자의 경우에는 프로세스가 자신이 할당받은 공간이 아닌 영역에 접근을 하려고 했으므로, 프로세스는 정상적인 실행을 계속할 수 없다. 즉, 커널이나 다른 프로세스의 주소 공간을 접근했다는 말이된다.

페이지 풀트(fault)는 인터럽트로 처리가 된다. 즉, 페이지 풀트 오류가 발생하면, 해당하는 인터럽트 핸들러가 수행된다. 이를 처리하는 인터럽트 핸들러로는 do_page_fault()가 ~/arch/i386/mm/fault.c에 정의되어 있다.

do_page_fault() 함수가 넘겨받는 에러코드(error_code)에는 아래와 같은 의미가 있다.

- bit 0 : 0이라면 페이지가 발견되지 않았다는 말이며, 1은 protection 오류가 발생했다는 말이다.
- bit1 : 0은 읽기에서, 1은 쓰기에서 오류가 생겼다는 말이다.
- bit 2 : 0은 커널이 오류를 일으켰으며, 1은 사용자 프로세스가 일으켰다는 말이된다.

이것 이외에 넘겨받는 값으로는 레지스터 값이 있다(struct pt_regs *regs).

```

asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    struct task_struct *tsk;
    struct mm_struct *mm;
    struct vm_area_struct * vma;
    unsigned long address;
    unsigned long page;
    unsigned long fixup;
    int write;
    siginfo_t info;
}

```

```

/* get the address */
__asm__("movl %%cr2,%0":"=r" (address));
tsk = current;
if (address >= TASK_SIZE)
    goto vmalloc_fault;
mm = tsk->mm;
info.si_code = SEGV_MAPERR;
if (in_interrupt() || !mm)
    goto no_context;
down(&mm->mmap_sem);
vma = find_vma(mm, address);
if (!vma)
    goto bad_area;
if (vma->vm_start <= address)
    goto good_area;
if (!(vma->vm_flags & VM_GROWSDOWN))
    goto bad_area;
if (error_code & 4) {
    if (address + 32 < regs->esp)
        goto bad_area;
}
if (expand_stack(vma, address))
    goto bad_area;

```

코드 318. do_page_fault() 함수

먼저 페이지 풀트를 일으킨 주소를 구해오도록 하자. 이 주소는 CR2 레지스터에 들어있을 것이다. 따라서, 이값을 address에 넣는다. 현재 프로세스를 구해서 tsk로 두고, address가 TASK_SIZE보다 큰 곳에서 발생했다면, vmalloc_fault로 제어를 옮긴다. 즉, 커널 영역에서의 가상 메모리 할당에서 문제가 일어났다고 보는 것이다. 그렇지 않다면, tsk의 mm_struct를 구해와서 mm에 넣고, info.si_code에는 SEGV_MAPPER(Segment Violation Map Error)를 둔다. 만약 interrupt중에 발생했거나, mm이 NULL을 가진다면 프로세스의 메모리 context를 알지 못하므로 no_context로 제어를 옮긴다.

프로세스의 mm_struct에 있는 세마포어를 획득하고(down()), 주소를 가지고 해당하는 vm_area_struct를 찾는다(find_vma()). 만약 찾을 수 없다면, 잘못된 주소 영역에 대한 접근 오류이므로 bad_area로 제어를 옮긴다. vm_start가 address보가 작거나 같다면 올바른 주소에 대한 접근 오류이므로 good_area로 제어를 옮긴다. vm_flags에 VM_GROWSDOWN이 설정되지 않았다면, .address가 vm_start보다 작은 주소를 가진 경우에는 다시 잘못된 영역에 대한 접근이므로 bad_area로 제어가 옮겨진다.

error_code를 0x100(=4)으로 AND시켜서 커널이 오류를 일으켰는지 아니면 사용자 프로세스가 오류를 일으켰는지 확인한다. 사용자가 오류를 일으킨 경우에는 address + 32가 regs->esp보다 작은지 확인하게 되며, 작을 경우에는 bad_area로 제어를 옮긴다. 즉, VM_GROWSDOWN이 설정되어 있으므로 stack 영역으로 지정된 곳에서 페이지 오류가 생겼기에 address + 32(pusha와 같은 연산의 경우에는 stack에 대해서 post-decrement가 행해지기 때문이다.)와 비교하도록 한다. expand_stack()은 스택을 확장해보려는 함수이다. 오류가 있다면 다시 제어는 bad_area로 옮겨간다.

```

good_area:
    info.si_code = SEGV_ACCERR;
    write = 0;
    switch (error_code & 3) {
        default: /* 3: write, present */
        /* fall through */
        case 2:      /* write, not present */
        if (!(vma->vm_flags & VM_WRITE))
            goto bad_area;
        write++;
        break;
    case 1:      /* read, present */

```

```

        goto bad_area;
    case 0:      /* read, not present */
        if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
            goto bad_area;
    }
switch (handle_mm_fault(mm, vma, address, write)) {
case 1:
    tsk->min_flt++;
    break;
case 2:
    tsk->maj_flt++;
    break;
case 0:
    goto do_sigbus;
default:
    goto out_of_memory;
}
if (regs->eflags & VM_MASK) {
    unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
    if (bit < 32)
        tsk->thread.screen_bitmap |= 1 << bit;
}
up(&mm->mmap_sem);
return;

```

코드 319. do_page_fault()함수 – continued

good_area는 올바른 메모리 영역에 대한 페이지 오류를 처리한다. info.si_code에는 SEGV_ACCERR(Segment Violation Access Error)를 설정하고, write에 0을 넣고, error_code에 0x11(=3)을 AND시킨다. 즉, 위에서 보여준 error_code의 bit의 의미에 대한 처리를 수행한다. 0은 페이지도 없으며, 읽기에서 발생했으므로 vm_flags에 VM_READ와 VM_EXEC 설정이 있는지 확인한다. 없다면 bad_area로 제어를 옮긴다. 1은 읽기에서 페이지의 protection 오류를 일으켰으므로 bad_area로 제어를 바로 옮긴다. 2는 쓰기에서 페이지가 없는 경우가 되므로 vm_flag에 VM_WRITE가 있는지 확인하고, 없다면 bad_area로 제어를 옮긴다. default는 그냥 아래로 통관한다.

handle_mm_fault()함수는 프로세스의 해당 mm_struct와 vm_area_struct, 오류를 유발한 address 및 write에서 오류가 생겼는지를 나타내는 값을 파라미터로 넘겨받아서, 오류를 처리하는 함수이다. 복귀 값에 따라서, 이곳에서는 1일 경우에는 task_struct의 min_flt¹²¹를 증가시키고, 2일 경우에는 maj_flt를 증가시킨다. 0이라면 do_sigbus로 제어를 옮긴다. 즉, 프로세스에 SIGBUS 시스널을 보낸다. 기본적으로는 out_of_memory로 제어를 옮긴다.

만약, regs->eflags에 VM_MASK가 설정되었다면, address에서 0xA0000을 뺀 후, 이를 페이지 크기로 나눈다(>>PAGE_SHIFT). 만약 이렇게 한 값이 32보다 작다면 task_struct의 thread.screen_bitmap에 이 값으로 1을 왼쪽으로 SHIFT한 값과 OR시킨다. 이것은 즉, VM86 모드의 DOS screen 메모리의 가상 주소에서 오류가 생겼다는 말로서 해당 프로세스의 thread 구조체에 있는 screen_bit에 이를 알리기 위한 것이다. 이젠 획득했던 세마포어를 놓고(up()) 복귀한다.

```

bad_area:
    up(&mm->mmap_sem);
bad_area_nosemaphore:
    if (error_code & 4) {
        tsk->thread.cr2 = address;
        tsk->thread.error_code = error_code;

```

¹²¹ min_flt는 프로세스의 minor page fault의 수를 나타내며, 단지 새로운 페이지 프레임을 필요로 한 경우이다. maj_flt는 major page fault를 유발한 수를 나타내며, do_no_page()나 do_swap_page()함수를 호출해서 처리한 페이지 풀트를 말한다.

```

        tsk->thread.trap_no = 14;
        info.si_signo = SIGSEGV;
        info.si_errno = 0;
        /* info.si_code has been set above */
        info.si_addr = (void *)address;
        force_sig_info(SIGSEGV, &info, tsk);
        return;
    }
/*
 * Pentium F0 0F C7 C8 bug workaround.
 */
if (boot_cpu_data.f00f_bug) {
    unsigned long nr;

    nr = (address - idt) >> 3;
    if (nr == 6) {
        do_invalid_op(regs, 0);
        return;
    }
}

```

코드 320. do_page_fault()함수 – continued

bad_area는 잘못된 주소 공간에 대한 접근 오류이다. 먼저 획득했던 세마포어를 놓는다(up()). error_code에 0x1000이 설정되었다면, 사용자 모드에서 오류가 발생했다는 말이 되므로 task_struct의 thread 구조체에 있는 cr2에는 주소를 error_code필드는 error_code로 trap_no(trap 번호)필드에는 14를 넣는다. info.si_signo는 보내고자 하는 시그널로 SIGSEGV를 설정하고, info.si_errno에는 0을 넣는다. info.si_addr에는 오류를 일으킨 주소(address)를 넣도록 하고, force_sig_info()를 호출해서 해당 프로세스가 시그널을 처리하도록 만든 후 복귀한다. Pentium 프로세스의 오류에서 기인한 것이라면, nr에 address – idt값을 8로 나누어(>>3)이 값이 6을 가진다면 do_invalid_op()¹²²함수를 호출하고 복귀한다.

```

no_context:
    if ((fixup = search_exception_table(regs->eip)) != 0) {
        regs->eip = fixup;
        return;
    }
    bust_spinlocks();
    asm("movl %%cr3,%0":"=r" (page));
    page = ((unsigned long *) __va(page))[address >> 22];
    if (page & 1) {
        page &= PAGE_MASK;
        address &= 0x003ff000;
        page = ((unsigned long *) __va(page))[address >> PAGE_SHIFT];
        printk(KERN_ALERT "*pte = %08lx\n", page);
    }
    die("Oops", regs, error_code);
    do_exit(SIGKILL);

```

코드 321. do_page_fault()함수 – continued

no_context부분은 인터럽트의 진행중에 발생한 오류를 처리하는 부분이다. 이 부분의 오류를 처리할 준비가 이미 되어있다면 이를 구해와서(search_exception_table()), regs->eip에 넣은 후 복귀한다. bust_spinlock()은 console_lock과 timerlist_lock을 spin lock으로 초기화 하는 일을 한다. page에는 CR3 레지스터의 내용을 읽어와서 현재 페이지에 대한 PGD를 구한다. 이것은 context정보를 얻을 수 없기 때문이다. page의 가상 주소에서(__va()), address가 가르키는 값에서 페이지 디렉토리 offset 주소를 더해서

¹²² 프로세스 chip이 가진 오류를 처리해 주는 함수이다.

page 값으로 다시 설정한다. 만약 page의 제일 하위 bit이 설정되었다면, page에서 PAGE_MASK를 AND시키고, address는 0x003FF000과 AND시킨다. 이 값을 이용해서 page 값을 새로 앞에서와 같은 방법으로 갱신하면 PGD에 들어있는 PTE의 주소를 알게된다. PTE의 주소를 prink()를 써서 표시한 후, die()함수¹²³를 호출하고, 다시 do_exit()함수에 SIGKILL을 넣어서 호출한다.

```

out_of_memory:
    up(&mm->mmap_sem);
    printk("VM: killing process %s\n", tsk->comm);
    if (error_code & 4)
        do_exit(SIGKILL);
    goto no_context;
do_sigbus:
    up(&mm->mmap_sem);
    tsk->thread.cr2 = address;
    tsk->thread.error_code = error_code;
    tsk->thread.trap_no = 14;
    info.si_code = SIGBUS;
    info.si_errno = 0;
    info.si_code = BUS_ADRERR;
    info.si_addr = (void *)address;
    force_sig_info(SIGBUS, &info, tsk);
    /* Kernel mode? Handle exceptions or die */
    if (!(error_code & 4))
        goto no_context;
    return;

```

코드 322. do_page_fault()함수 – continued

out_of_memory는 할당된 메모리 영역을 넘어서는 곳에 대한 연산이 오류를 일으킨 것으로, 획득한 세마포어를 놓고(up()), 프로세스를 죽인다는 메시지를 프로세스와 관련된 실행 화일의 이름(comm)과 함께 표시한 후, 사용자 프로세스에서 발생한 오류라면 do_exit()함수에 SIGKILL을 넣어서 호출한다. 그렇지 않다면, no_context로 제어를 옮긴다.

do_sigbus는 획득한 세마포어를 놓고(up()), 프로세스의 thread 구조체에 있는 cr2 필드에 오류를 일으킨 주소(address)를 넣는다. error_code 필드는 역시 error_code로 설정하고, trap_no에는 14를 준다. info.si_code에는 SIGBUS를, info.si_errno에는 0을, info.si_code에는 BUS_ADRERR(Bus Address Error)를, info.si_addr에는 address를 넣고, force_sig_info() 함수를 호출해서 프로세스가 SIGBUS 시그널을 받도록 한다. 알려주는 시그널 정보는 info구조체에 들어가 있다. 만약 커널에서 오류를 일으켰다면(error_code & 0x100 = 0인 경우), no_context로 제어를 옮긴다.

```

vmalloc_fault:
{
    int offset = __pgd_offset(address);
    pgd_t *pgd, *pgd_k;
    pmd_t *pmd, *pmd_k;
    pgd = tsk->active_mm->pgd + offset;
    pgd_k = init_mm.pgd + offset;
    if (!pgd_present(*pgd)) {
        if (!pgd_present(*pgd_k))
            goto bad_area_nosemaphore;
        set_pgd(pgd, *pgd_k);
        return;
    }
    pmd = pmd_offset(pgd, address);
}

```

¹²³ ~/arch/i386/kernel/traps.c에 정의되어 있다. 단순히 레지스터의 내용을 화면에 표시하고, do_exit()함수에 SIGSEGV를 넘겨주어서 호출하는 일을 한다.

```

        pmd_k = pmd_offset(pgd_k, address);
        if (pmd_present(*pmd) || !pmd_present(*pmd_k))
            goto bad_area_nosemaphore;
        set_pmd(pmd, *pmd_k);
        return;
    }
}

```

코드 323. do_page_fault() 함수

vmalloc_fault는 커널에서 valloc을 실행하다가 생긴 오류를 처리하는 경우이다. 먼저 오류를 일으킨 주소를 가지고 PGD의 offset을 찾은 다음, 프로세스의 active_mm구조체 필드가 가르키는 pgd에 offset을 더해서 pgd를 구한다. init_mm은 init 태스크의 active_mm을 가지며, 이 구조체의 pgd에 offset을 더하면, 커널에서 사용하는 페이지 디렉토리의 한 entry를 구해오게 된다(pgd_k). pgd_present()를 호출해서 해당하는 PGD의 entry가 메모리에 존재하는지 확인하고, 없다면 다시 pgd_present()함수를 pgd_k를 넘겨주어서 호출해서 메모리에 존재하는지 확인한다. 만약 없다면 bad_area_nosemaphore로 제어를 옮긴다. set_pgd()함수는 커널의 PGD에 페이지 오류를 일으킨 pgd를 설정하고 복귀하게 된다. 만약 pgd가 존재한다면, pmd를 pgd와 address를 이용해서 구하고(pmd_offset()), pmd_k에는 pgd_k와 address를 이용해서 구한 값(pmd_offset())을 넣는다. pmd는 존재하지만, pmd_k가 메모리에 존재하지 않는다면, 다시 제어를 bad_area_nosemaphore로 옮긴다. 이전 pmd_k에 pmd를 설정(set_pmd())하고 복귀한다.

```

int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct * vma,
                     unsigned long address, int write_access)
{
    int ret = -1;
    pgd_t *pgd;
    pmd_t *pmd;

    pgd = pgd_offset(mm, address);
    pmd = pmd_alloc(pgd, address);
    if (pmd) {
        pte_t * pte = pte_alloc(pmd, address);
        if (pte)
            ret = handle_pte_fault(mm, vma, address, write_access, pte);
    }
    return ret;
}

```

코드 324. handle_mm_fault() 함수

handle_mm_fault()함수는 ~/mm/memory.c에 정의되어 있으며, 페이지 풀트를 처리하는 do_page_fault() 함수에서 호출된다. 넘겨받은 오류를 일으킨 주소에서 해당 페이지 디렉토리에 대한 index를 찾아와서 pgd에 놓는다. 이곳에서(pgd) 새로운 페이지 middle directory를 할당받아서(pmd_alloc()) 이를 pmd로 둔다. 남은 것은 페이지 테이블을 페이지 middle directory에서 하나 만들어서, 이를 처리하는 것이다. handle_pte_fault() 함수가 이를 맡고 있다. handle_pte_fault() 함수 역시 ~/mm/memory.c에 정의되어 있다.

```

static inline int handle_pte_fault(struct mm_struct *mm,
                                   struct vm_area_struct * vma, unsigned long address,
                                   int write_access, pte_t * pte)
{
    pte_t entry;

    spin_lock(&mm->page_table_lock);
    entry = *pte;
    if (!pte_present(entry)) {

```

```

spin_unlock(&mm->page_table_lock);
if (pte_none(entry))
    return do_no_page(mm, vma, address, write_access, pte);
return do_swap_page(mm, vma, address, pte, pte_to_swp_entry(entry), write_access);
}
if (write_access) {
    if (!pte_write(entry))
        return do_wp_page(mm, vma, address, pte, entry);
    entry = pte_mkdirty(entry);
}
entry = pte_mkyoung(entry);
establish_pte(vma, address, pte, entry);
spin_unlock(&mm->page_table_lock);
return 1;
}

```

코드 325. handle_pte_fault()함수

mm_struct의 페이지 테이블을 고치기 위한 lock을 먼저 구한다(spin_lock()). entry는 앞에서 handle_mm_fault()가 할당한 pte로 초기화 하고, pte에 present 필드가 설정되었는지 확인한다. 없다면, 이전 페이지 테이블의 lock을 버리고, pte의 존재 여부를 물어서(pte_none()¹²⁴) do_no_page()함수를 호출한다. do_no_page()는 해당 페이지를 읽어서 물리적인 메모리에 올리고, 해당 페이지 테이블의 내용을 바꿔준다. 만약 이미 존재한다면 swap영역에 있는 경우가 되며, do_swap_page()를 호출해서 해당 페이지를 swap영역에서 다시 불러 들이도록 한다.

이하는 pte가 현재 존재하지 않는 것이되며, write_access가 설정되어 있다면 pte_write()를 호출해서 write가 가능한지를 살펴보고 write가 가능하지 않다면, do_wp_page()를 호출해서 해당 메모리에 대한 write가 가능하도록 만든다. pte_mkdirty()는 페이지 테이블의 entry가 dirty하다고 만든다.

pte_mkyoung()는 페이지 테이블의 엔트리가 지금 막 생성되었다고 표시하며, establish_pte()는 vm_struct와 주소(address) 및 페이지 테이블의 엔트리와, 위에서 초기화한 entry를 넘겨받아서 페이지 테이블의 엔트리를 구성하는 일을 한다. 마지막으로 할일은 설정했던 lock을 해제하고, 복귀값으로 1을 넘겨준다.

우리는 지금까지 메모리 할당과 해제에 관련된 일련의 행동들을 살펴보았다. 하드웨어 레벨에서부터 시작해서 메모리 할당 알고리즘들과 프로세스의 메모리 관리를 위한 데이터 구조까지 풀어 보았으며, 관련된 함수들도 보았다. 메모리 할당의 최소 단위는 역시 페이지 단위가 되며, 이것은 프로세스의 주소 공간을 차지하게 된다. 올바른 주소 공간에 대한 접근은 만족될 것이며, 그렇지 못한 공간에 대한 접근은 페이지 오류를 발생시키게 될 것이다. 물론 이러한 것들은 프로세스의 개입이 따르지 않고, 순순히 커널에서 해당 오류를 찾아내어 적절한 처리를 해줄 것이다. 사용자 프로세스는 자신이 잘 못된 페이지에 대한 참조를 했는지를 모르게되며, 커널은 해당 페이지를 읽어와서 프로세스가 실행을 계속 할 수 있도록 만들어줄 것이다. 물론 오류가 프로세스의 주소 공간과 무관하다면 커널은 프로세스의 실행을 중지할 수 있다.

¹²⁴ ~/include/asm/pgtable-2level.h or ~/include/asm/pgtable-3level.h에 #define pte_none(x) (!x).pte_low 혹은 #define pte_none(x) (!x).pte_low && !(x).pte_high로 정의되어 있다.

5. Network

Network이란 computer와 computer를 연결시켜준다. Computer들은 혼자서 있을 때보다 연결되었을 때 창출하는 가치가 더 커진다. 즉, 정보를 공유할 수 있도록 하며, 다른 computer의 resource를 사용할 수 있다. 리눅스에서의 kernel내의 network architecture는 [그림38]과 같다.

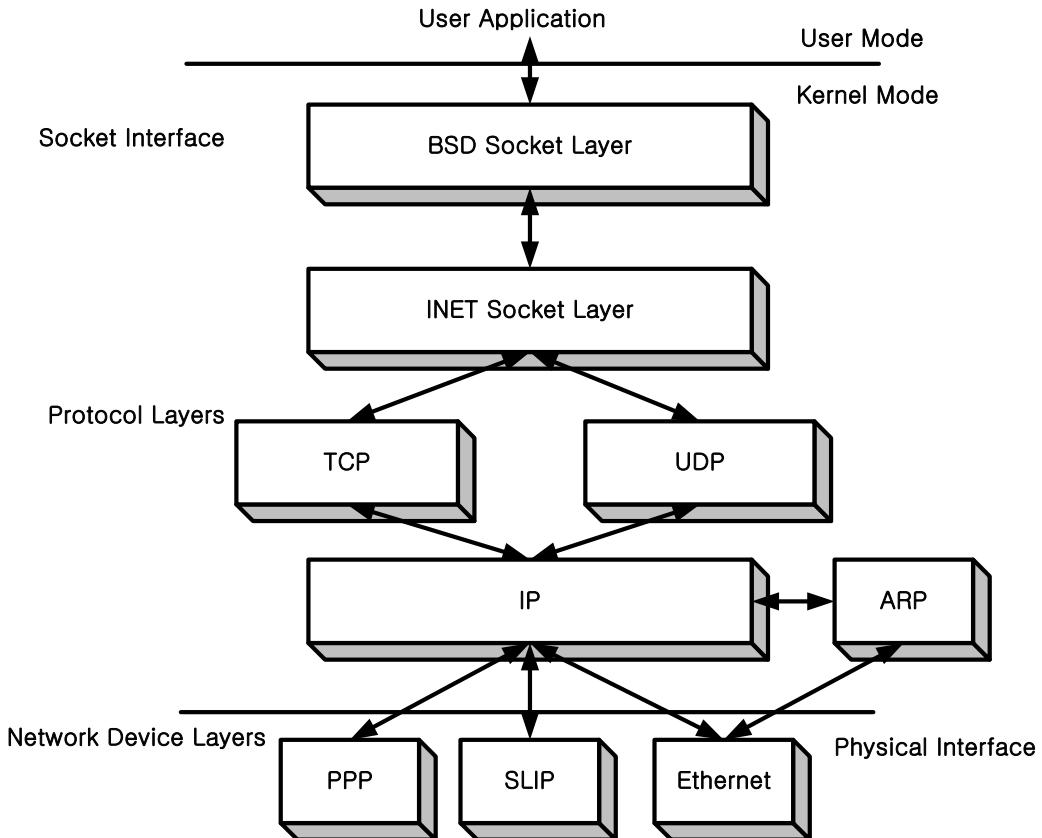


그림 45. Network architecture in Linux Kernel

이 구조를 살펴보면 기존의 BSD socket interface를 user application에 제공하기 위해서 BSD socket layer를 두고 있으며, 다시 이것을 internet protocol layer를 두어 protocol layer와 연관 짓고 있다. 또한, device driver layer와의 interface를 정해서 리눅스용 network device driver를 작성할 수 있게 만들어준다. 실제로 network device driver를 작성하는 사람은 상위로부터 넘겨받는 data에만 관심을 가질 뿐 상위에 어떤 protocol이 오는지는 상관하지 않는다. 이하에서는 리눅스 kernel이 network를 지원하기 위해서 어떤 구조를 가지는지를 살펴보도록 하자.

5.1. Network

잠시 기본적인 네트워크에 대한 이야기를 하도록 하겠다. 지금까지 가장 널리 많이 사용되고 있는 네트워크 프로토콜(protocol)로는 많은 책에서 언급한 TCP/IP를 들 수 있겠다. 이는 원래 군사적인 목적으로 만들어진 것이나, 인터넷이라는 용어가 보편화된 지금에서는 많은 사람들이 가장 친숙하게 사용하는 프로토콜이 되었다.

IP버전 4에서는 32bit를 가지고, 인터넷에 연결된 하나의 시스템을 지정하다.¹²⁵ 즉, 32만으로 전체 인터넷에 있는 시스템을 가르키게 되는 것이다. 따라서, 2^{32} 개의 시스템을 표시할 수 있는 것이다. 네트워크를 설명하면서, 왜 하필이면 주소를 먼저 말하게 되는 것일까? 이는 네트워크에 연결된

¹²⁵ IP버전 6에서는 128bit으로 주소를 지정한다.

시스템이 서로간에 의사를 전달하고자 하면, 반드시 상대방이 누구인가를 먼저 알아야 한다는 점이다. 또한 상대방과 어떤 언어, 즉 프로토콜로 대화를 할지도 결정해야 한다는 것이다. 이와 같은 과정이 없다면 네트워크는 불가능해진다.

다시 주소에 대한 이야기로 돌아가서, 기본적으로 주소는 8bit단위의 4개의 부분으로 나누어진다. XXX.XXX.XXX.XXX라는 형식으로 표시된다. 000.000.000.000에서 255.255.255.255까지의 주소가 지정 가능하다. 하지만, 특정 주소들은 다른 목적으로 사용되기 위해서 이미 지정되어 있기에, 그러한 주소 이외의 것을 선택해야 할 것이다. 참고로 IP공유란 하나의 인터넷주소를 여러대의 시스템이 공유하게 되는 것으로, 한대의 server 역할을 하는 시스템이 하위에 연결된 시스템들에 대한 인터넷 연결을 지원하는 것을 말한다. 이 경우에는 하위의 네트워크는 구성에 따라, 임의의 주소를 가질 수 있을 것이다.

IP주소는 크게 두부분으로 나누어진다. 한 부분은 네트워크 ID를 나타내고, 다른 한 부분은 그 네트워크에 속한 시스템의 주소를 나타내는 호스트 ID(host ID)이다. 이렇게 나누어보면, 주소의 종류에는 Class A, Class B, Class C, Class D, Class E, Class F 등이 있다. 이들은 각각 네트워크 ID와 호스트 ID를 위해서 얼마만큼의 bit를 사용할 것인가로 결정된다. [그림39]는 주소의 지정 방식을 보여준다.

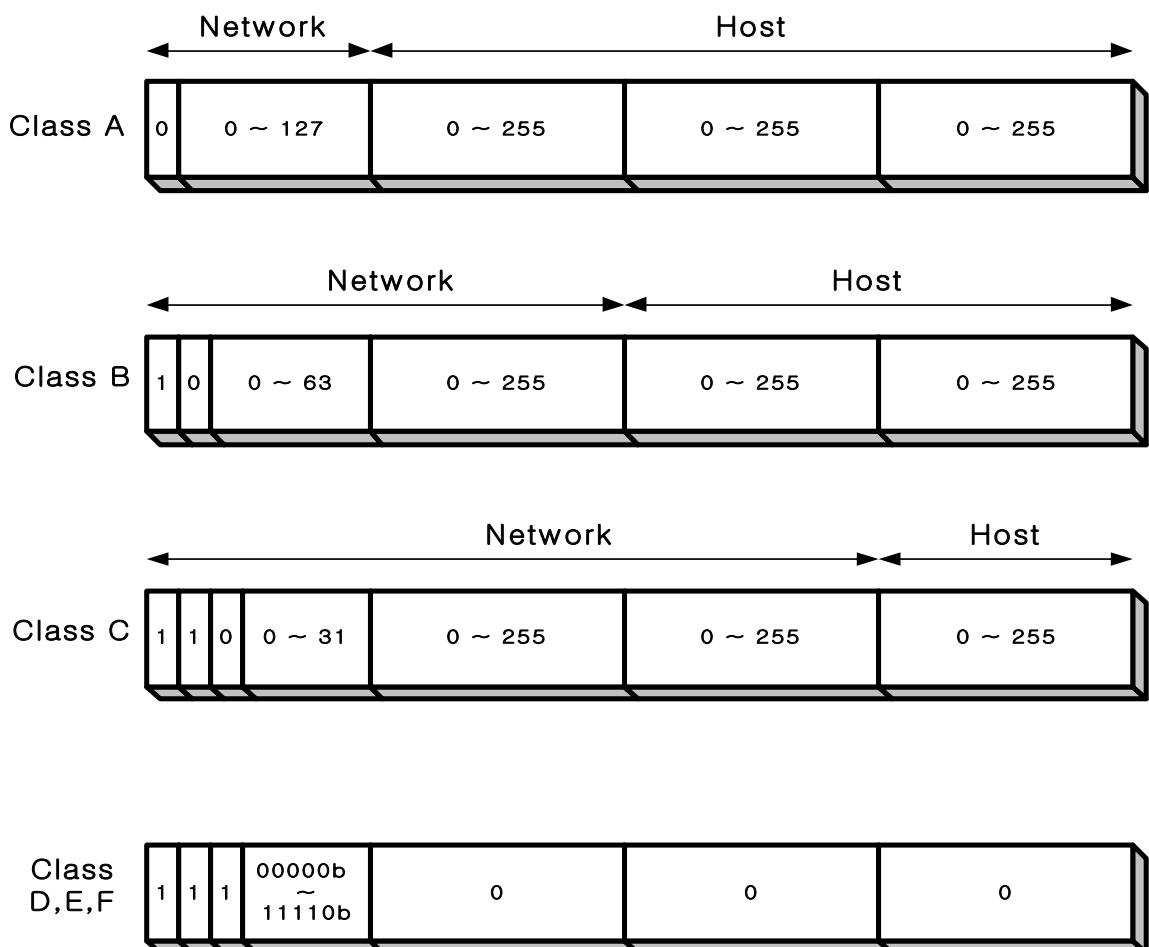


그림 46. IP Address Class

각각의 주소 계층(class)는 아래와 같다.

- Class A : Class A 주소는 네트워크 부분이 1.0.0.0에서 127.0.0.0으로 구성되며, 첫번째 1 byte에 network 주소가 지정된다. 나머지 24bit이 네트워크 내에서 호스트의 주소를 가르키게 된다.

- Class B : Class B 주소는 네트워크 부분이 128.0.0.0에서 191.255.0.0까지로 구성되며, 첫 2 bytes가 network를 나타내는 주소로 사용되며, 나머지 2 bytes를 가지고 네트워크 내의 호스트 주소를 가르킨다.
- Class C : Class C 주소는 네트워크 부분이 192.0.0.0에서 223.255.255.0까지로 구성되며, 첫 3 bytes를 네트워크의 주소로 사용하고, 나머지 1 byte를 호스트를 가르키는 주소로 사용한다.
- Class D, E, F: 이것에 해당하는 주소는 224.0.0.0에서 254.0.0.0에 해당하는 것으로 특수한 목적으로 사용된다. 예를 들어서 IP 패킷을 멀티캐스팅(Multicasting)¹²⁶ 하는 용도로 사용할 수 있다. 4번째 bit이 0인 경우에는 multicasting으로 Class D에 속하고 멀티캐스팅에 상요되며, 1로 설정되면 Class E와 F에 속한다.

여기서 한가지 짚고 넘어가야 할 것은 특정한 주소(예를 들어서 127.0.0.1)는 이미 사용 목적이 정해져 있다는 점이다. 127.0.0.1은 지역(local) 호스트¹²⁷의 주소를 가리킨다. 0.0.0.0의 경우에는 default router¹²⁸를 가르키며, 127.0.0.0은 loopback¹²⁹ 인터페이스를 가르킨다. 또한 네트워크 주소만을 가지고 호스트 부분의 bit를 전부 1로 설정한 경우는 네트워크에 속하는 모든 호스트를 가르키는 것으로 broadcast¹³⁰ 주소로 쓰인다.

주소가 결정되었다면, 이제는 상대방의 시스템만을 결정한 것이다. 즉, 접속할 시스템을 결정한 것이다. 이제는 접속할 시스템에서 어떤 사용자(혹은 프로그램, server 프로세스)와 통신할 것인가를 결정해야 할 것이다. 이것이 바로 포트(port)이다. 포트는 상대방 시스템에서 접속을 열어놓은 창구로 생각하면 될 것이다. 커널의 내부적으로 따지자면, 포트는 하나의 큐(queue) 역할을 한다. 즉, 이곳에 요청을 쌓아두게 되면, 나중에 server의 역할을 하는 프로세스가 이를 처리하고, 결과를 돌려줄 것이기 때문이다.

여기까지 했다면, 이제는 상대방 서버 프로세스까지가 결정된 것이다. 여기서 더 해주어야 하는 일은 하나의 서버가 있고 여러개의 클라이언트(client)가 있을 경우, 써버의 처리 방식이 달라질 수 있다는 것이다. 즉, 서버가 하나의 클라이언트에 대한 처리만을 하고, 다시 또 차례로 들어온 순서대로 처리할 것인가와 하나의 서버가 동시에 여러개의 클라이언트를 처리해 줄 것인가의 문제이다. 이를 위해서 소켓 디스크립터(socket descriptor)라는 것이 필요하다. 즉, 소켓 디스크립터를 할당해서 여러개의 동일한 서버 역할을 해주는 프로세스를 생성할 수 있으며, 각각이 고유한 소켓 디스크립터를 가지고, 클라이언트와 통신을 하게 되는 것이다.

소켓 디스크립터는 파일 디스크립터와 같다고 생각하면 될 것이다. 즉, BSD에서의 소켓의 구현은 파일 시스템을 구현하기 위한 것과 크게 다르지 않다. 따라서, 소켓에 대한 읽고 쓰기 연산은 파일에 대한 연산과 흡사하며, 부모와 자식 프로세스간에 상속관계가 성립한다. 즉, 새로이 생성된 프로세스는 부모가 사용하던 소켓 디스크립터를 상속받을 수 있다.

여기까지 이야기 해서, 주소를 지정하는 것을 마치기로 한다. 이제부터는 프로토콜에 대한 이야기를 하도록 하겠다. 먼저 설명해야 할 것이 연결을 가지는 통신(connection oriented)과 비연결성(connectionless)을 가지는 통신의 두 가지 방법에 대해서 보도록 하자. 연결성을 가지는 통신이란 항상 데이터가 올바르게 전달 되어는지를 확인하는 통신방법을 말하는 것이고, 그렇지 않은 경우를 비연결성 통신이라고 한다. TCP/IP에서는 이것을 크게 TCP와 UDP로 각각 구분한다. 즉, TCP는 데이터가 상대편에 확실히 전달되었다는 것을 보장해 주지만, UDP의 경우에는 그런 보장을 하지 않는다. 따라서, UDP의 경우에는 UDP 프로토콜의 상위에서 데이터의 무결성을 점검해 주어야 한다. 주로 TCP는 파일의

¹²⁶ Multicasting이란 네트워크에 맞물려있는 특정한 여러대의 컴퓨터에 한번에 동시에 패킷을 보내는 것이다.

¹²⁷ 현재 사용중인 호스트를 말한다.

¹²⁸ 패킷을 보내고자 하는 주소를 찾지 못할 경우에 특정한 설정이 없는 경우, 기본적으로 패킷을 전송하는 주소.

¹²⁹ 자신에게 되돌아오는 주소.

¹³⁰ 이것은 임의의 네트워크에 접속된 호스트로 패킷을 전송할 경우에 주소로 사용한다.(BOOTP, DHCP etc.)

전송과 같이 데이터가 정확히 전달되는 것을 보장하는 연산에 사용되며, UDP의 경우에는 스트림과 같이 짧은 데이터를 정확히 전달될 필요가 없을 때 사용된다.

TCP와 UDP이하에서는 IP가 존재한다. IP의 역할은 상위에서 넘겨받은 데이터 패킷을 전달하기 알맞은 형식으로 바꾸고, 이를 네트워크 디바이스 드라이버로 넘겨주는 역할을 한다. 넘겨받는 데이터의 내용은 상관하지 않으므로 TCP나 UDP 패킷 둘다를 처리할 수 있다. 넘겨주는 패킷은 IP 패킷이 될 것이다.

IP와 동일한 계층에 존재한다고 보는 것이 ARP/RARP이다. 이것은 하드웨어 주소를 IP주소로 바꾸어주는 역할을 하는 프로토콜이다. 패킷을 정확히 상대방에 전송하기 위해서는 상대의 하드웨어 주소를 알아야 하며, 이는 테이블 형태로 시스템에 저장된다. 시스템이 부팅하게 될 때, 자신이 속한 네트워크의 Routing Table을 만들고, 해당하는 IP주소에 대해서 하드웨어 주소를 mapping시켜둔다. 나중에 해당하는 IP주소로 데이터를 보내고자 할 때, 이곳에 등록된 하드웨어 주소를 이용하게 된다. 예를 들어서 가장 널리 쓰이고 있는 ethernet의 경우에는 6byte의 고유한 하드웨어 주소가 LAN카드에 지정되어 있다.¹³¹

이상에서 설명한 것들과 이하에서 설명하려고 하는 것에 대해서 간단히 그림을 그려본다면 아래와 같을 것이다.

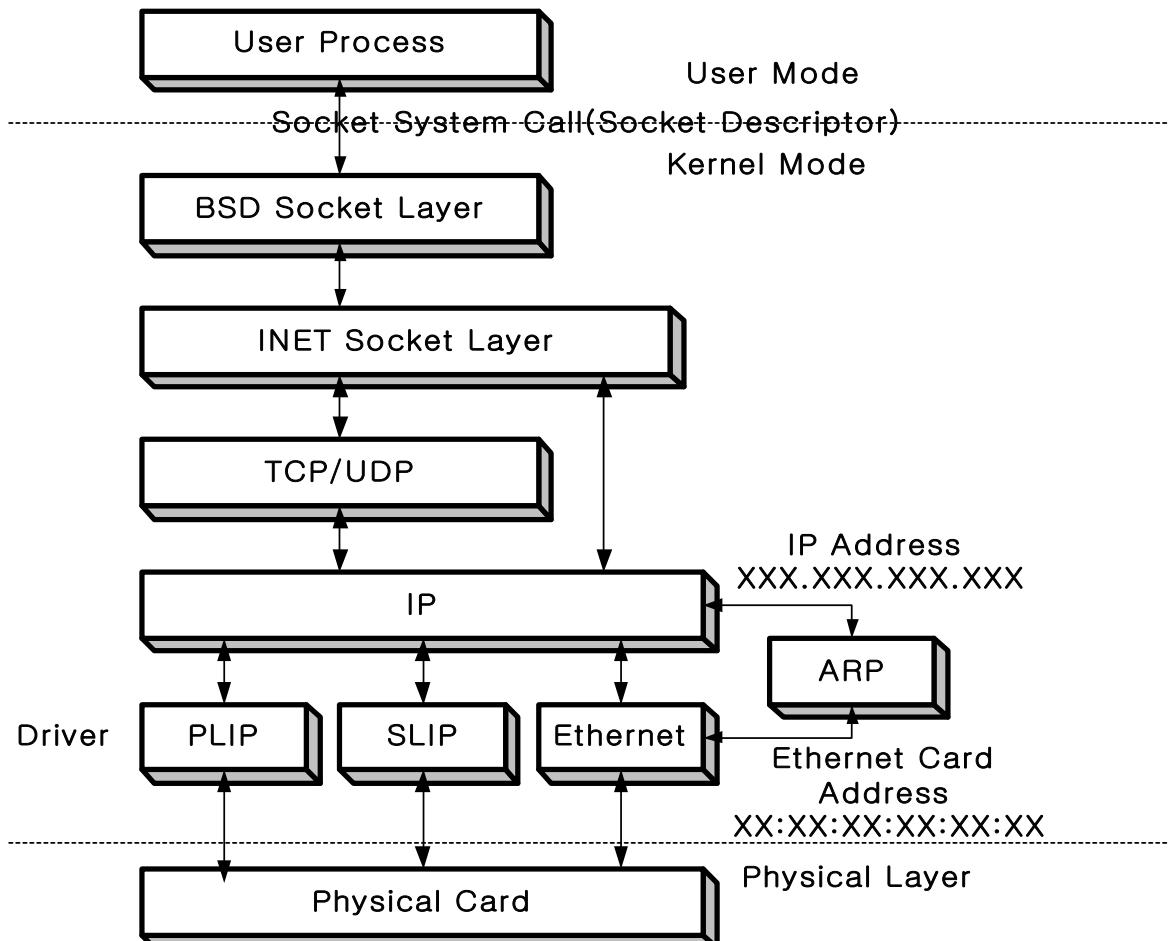


그림 47. 리눅스의 네트워크 구조

¹³¹ 나중에 네트워크 디바이스 드라이버를 살펴볼 때 알 수 있겠지만, EEPROM에 이 값을 저장하고 있는 네트워크 카드들이 많다. 이 주소는 항상 고정되어 사용되기도 하고, 혹은 프로그램으로 바꾸어 줄 수 있는 경우도 있기에 카드마다 다를 수 있다.

이하에서 볼 것은 시스템 콜과 각각의 layer가 하는 역할 및 관련된 함수들에 대해서 자세히 볼 것이다. 항상 위의 [그림40]을 잊지말고 기억하면서 보면 많은 도움이 될 것이다. 또한 논의하고자 하는 것은 internet address family로 한정하도록 한다. 이와같이 한정 짓는데는 다른 프로토콜에 대한 이해도 중요하지만, 가장 널리 사용되는 프로토콜에 대한 이해가 중요하리라는 생각에 바탕하고 있으며, 저자의 지식이 짧은 점도 일조를 했다. 일단 이정도로 간단히 네트워크의 구성에 대해서 마치도록하고 이젠 본격적으로 각 layer에 대해서 보도록 하자.

5.2. BSD Socket Layer

BSD 소켓 레이어는 시스템의 인터페이스를 지원하기 위해서 존재한다. 사용자 프로그램은 소켓 라이브러리를 통해서 시스템 콜을 하게되며, 이렇게 발생된 시스템콜은 BSD 소켓 레이어로 제어가 넘겨진다. 리눅스에서는 다음과 같은 네트워크에 대한 system call들을 정의하고 있다. 이것은 BSD계열의 socket 인터페이스와 똑같은 형태를 가지고 있으며, 기존에 나온 프로그램들에 대한 지원이라고 생각해도 좋을 것이다.

시스템 콜	Description
int socket(int addr_family, int type, int protocol)	소켓을 생성한다.
int bind(int s, struct sockaddr *address, int address_len)	이름을 소켓에 묶는다(bind).
int listen(int s, int backlog)	소켓에 대해서 연결을 기다린다.
int connect(int s, struct sockaddr *address, int address_len)	소켓에 대해서 연결을 요청한다.
int accept(int s, struct sockaddr *address, int *address_len)	소켓에 대한 연결을 받아 들인다.
int send(int s, char *msg, int len, int flags)	소켓에 대해서 데이터를 보낸다.
int sendto(int s, char *msg, int len, int flags, struct sockaddr *to, int tolen)	소켓에 대해서 데이터를 보낸다. 단지, 하나의 메시지를 보내고자 할 때 사용한다.
int recv(int s, char *buf, int len, int flags)	소켓에 대해서 데이터를 받는다.
int recvfrom(int s, char *buf, int len, int flags, struct sockaddr *from, int *fromlen)	소켓에 대해서 데이터를 받는다. 단지, 하나의 메시지만을 받고자 할 때 사용한다.
int getsockopt(int s, int level, int oname, char *ovalue, int *olen)	소켓의 현재 옵션을 가져온다.
int setsockopt(int s, int level, int oname, char *ovalue, int *olen)	소켓의 현재 옵션을 지정한다.

표 39. 소켓에 대한 시스템 콜

위에서 정의한 연산 이외에도 sendmsg(), recvmsg(), shutdown()이 더 있는데, 근본적으로 위에서 정의한 연산의 내용과 크게 다르지 않다. shutdown의 경우에는 생성된 socket을 없애주는 역할을 한다. 위와 같은 함수는 직접적으로 프로세스에서 사용할 수 있는 함수로서 정의되어 있지만, 시스템 콜로 들어가면, 단지 socketcall()만이 정의되어 있다. 나머지는 넘겨주는 파라미터 값에 따라 각각이 호출된다. 아래의 코드 socketcall()함수를 보여준다. 코드는 ~/net/socket.c를 참조하기 바란다.

```
asmlinkage long sys_socketcall(int call, unsigned long *args)
{
    unsigned long a[6];
    unsigned long a0,a1;
    int err;

    if(call<1||call>SYS_RECVMSG)
        return -EINVAL;

    /* copy_from_user should be SMP safe. */
    if (copy_from_user(a, args, nargs[call]))
        return -EFAULT;

    a0=a[0];
    a1=a[1];
```

```

switch(call)
{
    case SYS_SOCKET:
        err = sys_socket(a0,a1,a[2]);
        break;
    case SYS_BIND:
        err = sys_bind(a0,(struct sockaddr *)a1, a[2]);
        break;
    case SYS_CONNECT:
        err = sys_connect(a0, (struct sockaddr *)a1, a[2]);
        break;
    case SYS_LISTEN:
        err = sys_listen(a0,a1);
        break;
    case SYS_ACCEPT:
        err = sys_accept(a0,(struct sockaddr *)a1, (int *)a[2]);
        break;
    case SYS_GETSOCKNAME:
        err = sys_getsockname(a0,(struct sockaddr *)a1, (int *)a[2]);
        break;
    case SYS_GETPEERNAME:
        err = sys_getpeername(a0, (struct sockaddr *)a1, (int *)a[2]);
        break;
    case SYS_SOCKETPAIR:
        err = sys_socketpair(a0,a1, a[2], (int *)a[3]);
        break;
    case SYS_SEND:
        err = sys_send(a0, (void *)a1, a[2], a[3]);
        break;
    case SYS_SENDTO:
        err = sys_sendto(a0,(void *)a1, a[2], a[3],
                        (struct sockaddr *)a[4], a[5]);
        break;
    case SYS_RECV:
        err = sys_recv(a0, (void *)a1, a[2], a[3]);
        break;
    case SYS_RECVFROM:
        err = sys_recvfrom(a0, (void *)a1, a[2], a[3],
                          (struct sockaddr *)a[4], (int *)a[5]);
        break;
    case SYS_SHUTDOWN:
        err = sys_shutdown(a0,a1);
        break;
    case SYS_SETSOCKOPT:
        err = sys_setsockopt(a0, a1, a[2], (char *)a[3], a[4]);
        break;
    case SYS_GETSOCKOPT:
        err = sys_getsockopt(a0, a1, a[2], (char *)a[3], (int *)a[4]);
        break;
    case SYS_SENDSMSG:
        err = sys_sendmsg(a0, (struct msghdr *) a1, a[2]);
        break;
    case SYS_RECVMSG:
        err = sys_recvmsg(a0, (struct msghdr *) a1, a[2]);
        break;
    default:
        err = -EINVAL;
        break;
}

```

```

    }
    return err;
}

```

또한 리눅스에서는 소켓에 파일에 대한 연산을 직접 적용할 수 있도록 하기위해서 write, read등의 파일에 대한 연산을 가지고 있다. 이같은 연산을 하기위해서는 물론 이 경우에는 소켓이 이미 생성되어 있어야 하며, 연결 되어 있는 경우가 될 것이다.

5.3. Socket의 생성

소켓의 생성과 관련된 된 시스템 콜은 sys_socket()이다. 시스템 콜을 프로그램에서 하는 방법은 아래와 같다.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket( int domain, int type, int protocol );
```

넘겨주는 값은 domain에 해당하는 것으로는 PF_XXX로 시작하는 값을 가지며, 프로토콜의 패밀리를 나타낸다. 정의는 ~/include/linux/socket.h에 있다. Type에는 SOCK_XXX로 시작하는 값을 가지며, ~/include/asm/socket.h에 정의되어 있다. Type은 소켓의 타입을 결정한다. 마지막으로 protocol필드에는 RAW socket을 제외하고는 항상 0을 가지도록 한다.

```
asmlinkage long sys_socket(int family, int type, int protocol)
{
    int retval;
    struct socket *sock;

    retval = sock_create(family, type, protocol, &sock);
    if (retval < 0)
        goto out;
    retval = sock_map_fd(sock);
    if (retval < 0)
        goto out_release;
out:
    /* It may be already another descriptor 8) Not kernel problem. */
    return retval;
out_release:
    sock_release(sock);
    return retval;
}
```

코드 326. sys_socket() 함수

sys_socket함수는 하나의 BSD 소켓 구조체¹³²를 생성해서(sock_create()), 이것을 파일 디스크립터의 하나로 맵핑시킨다(sock_map_fd()). 수행 도중에 에러가 발생했다면, 이전에 할당 받은 BSD 소켓 구조체가 있다면 이를 해제하고 복귀한다. 에러가 없다면 매핑된 파일 디스크립터를 돌려줄 것이다.

```
int sock_create(int family, int type, int protocol, struct socket **res)
{
    int i;
```

¹³² 이 장의 뒷에서 이와 관련된 이야기를 할 때, 더 정확히 볼 것이다. 지금은 그러한 것이 있다는 것만 기억하도록 한다.

```

struct socket *sock;

if(family<0 || family>=NPROTO)
    return -EAFNOSUPPORT;
if (family == PF_INET && type == SOCK_PACKET) {
    static int warned;
    if (!warned) {
        warned = 1;
        printk(KERN_INFO "%s uses obsolete (PF_INET,SOCK_PACKET)Wn",
current->comm);
    }
    family = PF_PACKET;
}

```

코드 327. `sock_create()`함수

`sock_create()`함수는 BSD 소켓을 하나 생성하는 역할을 한다. 먼저 프로토콜 패밀리가 정해진 범위에 있는지를 확인하고, 만약 그렇지 않다면 `-EAFNOSUPPORT`를 돌려준다. 만약 `PF_INET`을 프로토콜 패밀리로 가지고, 원하는 소켓의 타입이 `SOCK_PACKET`이라면, 호환성을 고려해서 프로토콜 패밀리를 `PF_PACKET`으로 바꾼다.

```

#endif defined(CONFIG_KMOD) && defined(CONFIG_NET)
    if (net_families[family]==NULL)
    {
        char module_name[30];
        sprintf(module_name,"net-pf-%d",family);
        request_module(module_name);
    }
#endif
    net_family_read_lock();
    if (net_families[family] == NULL) {
        i = -EAFNOSUPPORT;
        goto out;
    }

```

코드 328. `sock_create()`함수 – continued

이전 해당하는 프로토콜 패밀리를 커널에서 지원하는지를 확인하는 일이다. 만약 커널에서 이미 해당 프로토콜 패밀리를 가지고 있지 않다면, 모듈의 이름으로 `net-pf-XXX`를 가지는 프로토콜 패밀리의 모듈을 적재하도록 요청한다(`request_module()`). 그리고 나서, 네트워크 패밀리에 대한 읽기 lock을 설정한 후, 해당 프로토콜 패밀리 모듈이 다시 적재되었는지 확인한다. 이때, 프로토콜 패밀리에 `NULL`을 가진다면 `-EAFNOSUPPORT`를 에러로 돌려준다.

```

if (!(sock = sock_alloc()))
{
    printk(KERN_WARNING "socket: no more socketsWn");
    i = -ENFILE;           /* Not exactly a match, but its the
                           closest posix thing */
    goto out;
}
sock->type = type;
if ((i = net_families[family]->create(sock, protocol)) < 0)
{

```

```

        sock_release(sock);
        goto out;
    }
    *res = sock;
out:
    net_family_read_unlock();
    return i;
}

```

코드 329. sock_create()함수 – continued

마지막으로 남은 것은 실제적인 BSD 소켓 구조체의 할당이다. 이것은 `sock_alloc()` 함수가 처리한다. `sock_alloc()` 함수가 0을 돌려준다면 더 이상 BSD 소켓을 생성할 수 없다는 말이되며 `out`으로 제어를 옮긴다. 대신으로 할당 받았다면, 이전 BSD 소켓 구조체의 `type` 필드에 넘겨받은 `type`을 넣고, 프로토콜 패밀리에서 제공하는 `create()` 함수를 호출한다. 넘겨주는 것은 할당받은 BSD 소켓 구조체와 넘겨받은 `protocol`이다. 만약 예러가 있다면 0보다 작은 값을 넘겨받을 것이며, 할당받은 BSD 소켓 구조체를 해제하고(`sock_release()`) `out`으로 제어를 옮긴다. 최종 결과는 생성된 BSD 소켓 구조체이며, 복귀전에 설정한 `lock`을 해제하고 예러코드와 함께 돌아간다.

```

struct socket *sock_alloc(void)
{
    struct inode * inode;
    struct socket * sock;

    inode = get_empty_inode();
    if (!inode)
        return NULL;
    inode->i_sb = sock_mnt->mnt_sb;
    sock = socki_lookup(inode);
    inode->i_mode = S_IFSOCK|S_IRWXUGO;
    inode->i_sock = 1;
    inode->i_uid = current->fsuid;
    inode->i_gid = current->fsgid;
    sock->inode = inode;
    init_waitqueue_head(&sock->wait);
    sock->fasync_list = NULL;
    sock->state = SS_UNCONNECTED;
    sock->flags = 0;
    sock->ops = NULL;
    sock->sk = NULL;
    sock->file = NULL;
    sockets_in_use[smp_processor_id()].counter++;
    return sock;
}

```

코드 330. sock_alloc()함수

`sock_alloc()` 함수는 `inode` object와 BSD 소켓을 연결하는 역할을 한다. 즉, 빈 `inode` object를 하나 찾아와서(`get_empty_inode()`), `i_sb`필드에 `sock_mnt->mnt_sb`를 연결하고, `inode`에 관련된 BSD 소켓을 찾은 후(`socki_lookup()`), `inode`의 각 필드들을 적절한 값으로 설정한다. 그리고, BSD 소켓의 `inode`필드를 이 `inode`를 연결하는데 사용하고, `wait`큐를 초기화한다(`init_waitqueue_head()`). 나머지는 BSD 소켓에 대한 초기화이다. 아직 연결되지 않은 상태이므로 `SS_UNCONNECTED`가 상태로 설정된다. BSD 소켓에 대한

사용 카운터를, CPU ID를 인덱스 값으로 해서 찾아서 증가 시켜준다. 돌려주는 값은 찾은 BSD 소켓이 된다.

```
void sock_release(struct socket *sock)
{
    if (sock->ops)
        sock->ops->release(sock);
    if (sock->fasync_list)
        printk(KERN_ERR "sock_release: fasync list not empty!\\n");
    sockets_in_use[smp_processor_id()].counter--;
    if (!sock->file) {
        iput(sock->inode);
        return;
    }
    sock->file=NULL;
}
```

코드 331. `sock_release()` 함수

`sock_release()` 함수는 단순히 이전에 할당받은 `inode`를 해제하기만 하면 된다. 먼저 이것을 해주기 전에 BSD 소켓 연산 구조체의 `release()` 함수를 호출해서 해제되기 위한 처리를 한 다음, 소켓의 `fasync_list`에 멤버가 있는지 확인한다. 멤버가 있다면, 예상 상황이다.

CPU ID를 인덱스로 한 BSD 소켓 사용 카운터를 감소시킨 후, 만약 소켓의 `file` 필드가 `NULL`을 가진다면 `inode object`를 버리고(`iput()`) 복귀한다. 그렇지 않다면, 일단 `file` 필드에 `NULL`을 넣는다. 나중에 다시 한번 호출이 있을 경우가 되면, 이미 `file` 필드가 `NULL`로 되어 있으므로 `inode object`가 지워질 것이다.

참고로 `inode object`를 보면 `union`으로 정의된 곳에 BSD 소켓을 위한 부분을 찾을 수 있을 것이다. 따라서, `inode object`만을 할당하더라도 BSD 소켓 구조체가 할당된다.

이제 남은 것은 프로토콜 패밀리의 `create()` 함수이다. 우리가 볼 것은 INET에 대한 것이기에 프로토콜 패밀리도 PF_INET이 해당한다. 아래와 같다. `net_families[]`는 `net_proto_family` 구조체¹³³의 배열로 정의되며, 프로토콜 패밀리의 이름과 생성 함수(`create()`), 인증과 암호화에 대한 필드로 구성된다. 정의는 `~/net/socket.c`를 참조하기 바란다. 새로운 프로토콜 패밀리의 등록은 `sock_register()`가 맡고 있으며, 해제는 `sock_unregister()` 함수¹³⁴가 담당한다. PF_INET 프로토콜 패밀리에 대한 초기화는 `~/net/ipv4/af_inet.c`에 있는 `inet_init()` 함수가 맡고 있다. 이곳에서 `sock_register()` 함수를 호출하는 것으로 하나의 프로토콜 패밀리를 등록한다. 넘겨 주는 값은 `inet_family_ops`의 주소이며, 정의는 `~/net/ipv4/af_inet.c`에 아래와 같이 나온다.

```
struct net_proto_family inet_family_ops = {
    PF_INET,
    inet_create
};
```

코드 332. `inet_family_ops`의 정의

즉, PF_INET으로 프로토콜 패밀리 이름을 등록하고, `inet_create()` 함수를 프로토콜 패밀리의 `create()` 함수로 서정했다. `inet_create()` 함수는 아래와 같다.

```
static int inet_create(struct socket *sock, int protocol)
```

¹³³ `~/include/linux/net.h`에 정의되어 있다.

¹³⁴ 이 함수들에 대한 정의는 `~/net/socket.c`에 나와 있다. 이것을 호출하는 부분을 커널에서 찾으면 어떤 프로토콜 패밀리들이 등록되는지 확인할 수 있을 것이다.

```
{
    struct sock *sk;
    struct proto *prot;

    sock->state = SS_UNCONNECTED;
    sk = sk_alloc(PF_INET, GFP_KERNEL, 1);
    if (sk == NULL)
        goto do_oom;
```

코드 333. inet_create() 함수

inet_create()함수는 INET 프로토콜 레이어에서 사용할 INET 소켓¹³⁵을 생성하는 일을 한다. INET 소켓의 구조체는 sock를 사용하여, 자세한 정의는 뒤에서 보도록 하겠다. 일단 현재 BSD 소켓은 연결 상태가 아니므로 SS_UNCONNECTED로 상태를 가진다. 그리고, 나서 해당 프로토콜 패밀리에 맞는 INET 소켓을 하나 할당 받는다(sk_alloc()). 넘겨주는 값은 PF_INET과 커널 메모리의 할당 우선 순위와 1을 준다. 만약 할당 받을 수 없다면 do_oom으로 제어를 옮긴다.

```
switch (sock->type) {
    case SOCK_STREAM:
        if (protocol && protocol != IPPROTO_TCP)
            goto free_and_noproto;
        protocol = IPPROTO_TCP;
        prot = &tcp_prot;
        sock->ops = &inet_stream_ops;
        break;
    case SOCK_SEQPACKET:
        goto free_and_badtype;
    case SOCK_DGRAM:
        if (protocol && protocol != IPPROTO_UDP)
            goto free_and_noproto;
        protocol = IPPROTO_UDP;
        sk->no_check = UDP_CSUM_DEFAULT;
        prot=&udp_prot;
        sock->ops = &inet_dgram_ops;
        break;
    case SOCK_RAW:
        if (!capable(CAP_NET_RAW))
            goto free_and_badperm;
        if (!protocol)
            goto free_and_noproto;
        prot = &raw_prot;
        sk->reuse = 1;
        sk->num = protocol;
        sock->ops = &inet_dgram_ops;
        if (protocol == IPPROTO_RAW)
            sk->protinfo.af_inet.hdrincl = 1;
        break;
    default:
        goto free_and_badtype;
}
```

¹³⁵ 이것은 BSD 소켓과는 또 다른 소켓이다. 나중에 이에 대해서도 자세히 보게 될 것이다.

코드 334. inet_create()함수 – continued

이전 소켓의 타입에 맞는 것들로 할당받은 INET 소켓을 초기화 하는 일이다. SOCK_STREAM과 SOCK_SEQPACKET, SOCK_DGRAM, SOCK_RAW등의 소켓 타입이 있으며, 해당하지 않는다면 free_and_badtype으로 제어를 옮긴다. SOCK_STREAM은 TCP를 사용한 연결이다. protocol을 IPPROTO_TCP로 설정하고, prot를 tcp_prot의 주소로 만든 다음 INET 소켓의 연산 구조체로 inet_stream_ops를 설정한다. SOCK_SEQPACKET에 대해서는 지원하고 있지 않으므로 그냥 free_and_badtype으로 제어를 옮긴다. SOCK_DGRAM은 UDP를 사용하는 경우로 protocol을 IPPROTO_UDP로 설정하고, UDP에서 사용하는 default checksum을 사용한다는 것으로 INET 소켓의 no_check를 설정한 다음, prot를 udp_prot의 주소로 만든다. INET 소켓의 연산 구조체는 당연히 inet_dgram_ops가 될 것이다. SOCK_RAW는 RAW 소켓을 사용하는 경우로, 현재 capability가 CAP_NET_RAW 지원하는지 확인한다. 지원하지 않는다면 free_and_badperm으로 제어를 옮긴다. 만약 protocol이 0값 가진다면 다시 제어를 free_and_noproto로 옮긴다. BSD 소켓의 reuse 필드를 1로 설정하고, num 필드를 protocol로 초기화 하며, 연산 구조체는 inet_dgram_ops로 둔다. 즉, 데이터 그램으로 전송을 하겠다는 의미가 된다. 만약 protocol이 IPPROTO_RAW로 되어 있다면, INET 소켓의 protoinfo.af_inet.hdrincl을 1로 설정한다.

```

if (ipv4_config.no_pmtu_disc)
    sk->protinfo.af_inet.pmtudisc = IP_PMTUDISC_DONT;
else
    sk->protinfo.af_inet.pmtudisc = IP_PMTUDISC_WANT;
sock_init_data(sock,sk);
sk->destruct = inet_sock_destruct;
sk->zapped = 0;
sk->family = PF_INET;
sk->protocol = protocol;
sk->prot = prot;
sk->backlog_rcv = prot->backlog_rcv;
sk->protinfo.af_inet.ttl=sysctl_ip_default_ttl;
sk->protinfo.af_inet.mc_loop=1;
sk->protinfo.af_inet.mc_ttl=1;
sk->protinfo.af_inet.mc_index=0;
sk->protinfo.af_inet.mc_list=NULL;
#endif INET_REF_CNT_DEBUG
    atomic_inc(&inet_sock_nr);
#endif

```

코드 335. inet_create()함수 – continued

이전 할당받은 INET 소켓의 일반적인 초기화를 할 차례이다. IPv4의 현재 설정이 경로 최대 전송 단위(Path Maximum Transfer Unit)를 찾도록 설정되었다면, IP_PMTUDISC_WANT를, 그렇지 않다면 IP_PMTUDISK_WANT를 INET 소켓의 protinfo.af_inet.pmtudisc에 넣어준다. sock_init_data()함수는 ~/net/core/sock.c에 정의되어 있으며, INET 소켓의 필드들 및 함수들과 큐에 대해 기본적인 초기화를 한다. 그 이하의 부분들 역시 INET 소켓에 대한 초기화이다. inet_sock_destruct() 함수는 INET 소켓을 없애는 역할을 하는 함수로 ~/net/ipv4/af_inet.c에 정의되어 있다.

```

if (sk->num) {
    sk->sport = htons(sk->num);
    sk->prot->hash(sk);
}
if (sk->prot->init) {
    int err = sk->prot->init(sk);
}

```

```

        if (err != 0) {
                inet_sock_release(sk);
                return(err);
        }
}
return(0);
free_and_badtype:
sk_free(sk);
return -ESOCKTNOSUPPORT;
free_and_badperm:
sk_free(sk);
return -EPERM;
free_and_noproto:
sk_free(sk);
return -EPROTONOSUPPORT;
do_oom:
return -ENOBUFS;
}

```

코드 336. `inet_create()` 함수 – continued

앞에서 RAW 소켓과 같이 INET 소켓에 번호(number)를 준 경우에, sport필드를 네트워크 번호(htons())로 바꾼 num필드로 설정하고, INET 소켓을 프로토콜 해쉬 체인에 추가한다¹³⁶. 만약 하위 프로토콜의 초기화 함수가 설정되어 있다면, 이것도 실행(`sk->prot->init()`) 시켜준다. 이곳에서 하위 프로토콜의 INET 소켓에 대한 초기화가 일어난다. 만약 에러가 있다면 할당받은 INET 소켓을 해제하고(`inet_sock_release()`) 에러값을 돌려주면서 복귀한다.

이상에서 에러가 발생하지 않았다면 최종적으로 0을 돌려주게 된다. 나머지는 함수의 수행중에 발생한 에러에 대한 처리를 담당하는 부분이다. 각각에 대해서 돌려주는 에러가 다르다는 정도만 알면 될 것이다.

¹³⁶ 이것은 나중에 해당하는 INET 소켓을 쉽게 찾을 수 있도록 만든다.

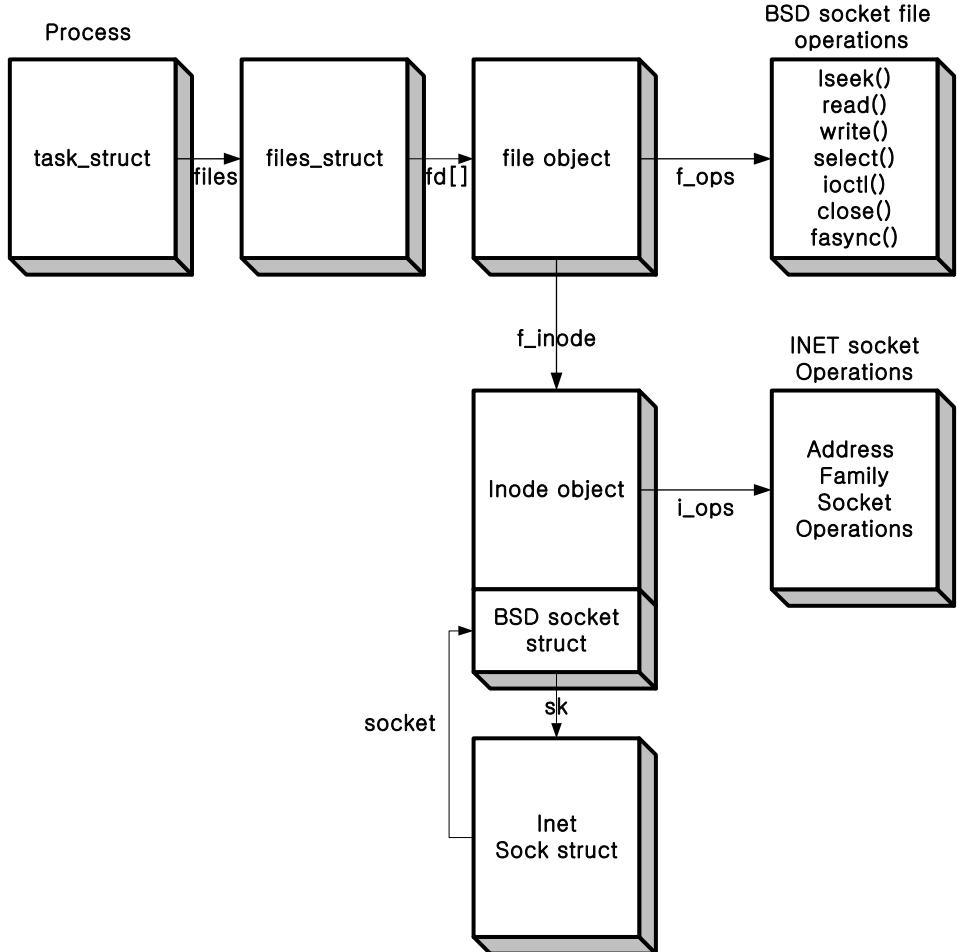


그림 48. 프로세스와 관련된 BSD/INET 소켓의 생성

이상에서 우리는 하나의 소켓을 생성하는 절차를 쭉 돌아보았다. 생성된 BSD 소켓을 프로세스와 연관시켜서 전체적으로 보려면 [그림41]을 참조하기 바란다. 즉, 하나의 프로세스는 파일 디스크립터 테이블을 가지고 있으며, 이것을 통해서 생성된 소켓을 접근한다. file object의 파일 연산은 BSD 소켓이 제공하는 연산을 사용하게 되며, 관련된 inode에는 BSD 소켓이 들어간다. 생성된 BSD 소켓에 대해서 이제 부터는 일반 파일에 대한 연산이 동일하게 적용될 수 있게 되는 것이다. 이젠 데이터를 실제로 어떤 식으로 전송하는지를 볼 차례이다.

5.4. Socket을 이용한 데이터 보내기

앞에서 본 소켓의 생성에 충분히 이해 했다면, 리눅스에서는 파일에 대한 연산을 그대로 네트워크에 대한 연산으로 적용할 수 있도록 하고 있음을 알게될 것이다. 가령 예를 들어서, 파일에 대한 write를 소켓에 대한 write연산과 동일하게 정의 하고 있다. 즉, 아래와 같은 형식의 write 시스템 콜을 정의 한다. write 시스템 콜은 sys_write()이며, ~/fs/read_write.c에서 볼 수 있다.

```

asmlinkage ssize_t sys_write(unsigned int fd, const char * buf, size_t count)
{
    ssize_t ret;
    struct file * file;

    ret = -EBADF;
    file = fget(fd);
    if (file) {
        if (file->f_mode & FMODE_WRITE) {
            if (file->f_op->write)
                ret = file->f_op->write(file, buf, count);
        }
        fput(file);
    }
}
  
```

```

        struct inode *inode = file->f_dentry->d_inode;
        ret = locks_verify_area(FLOCK_VERIFY_WRITE, inode, file,
                               file->f_pos, count);
        if (!ret) {
            ssize_t (*write)(struct file *, const char *, size_t, loff_t *);
            ret = -EINVAL;
            if (file->f_op && (write = file->f_op->write) != NULL)
                ret = write(file, buf, count, &file->f_pos);
        }
        if (ret > 0)
            inode_dir_notify(file->f_dentry->d_parent->d_inode,
                             DN MODIFY);
        fput(file);
    }
    return ret;
}

```

코드 337. sys_write()

sys_write() 시스템 콜은 VFS(Virtual File System)의 일부로서, 파일 디스크립터의 index를 받아서, 해당하는 파일 object에 대한 포인터를 얻고, 이를 다시 이용해서 해당하는 inode구조체를 얻는다. 그리고나서 해당하는 파일 object의 파일 연산자 write를 가지고와서 이를 수행하는 구조로 되어 있다. 즉, 파일 object를 소켓에 대해서도 정의해 주고, 이에 대한 디스크립터만 가지고 있다면, 프로세스가 소켓에 대한 write도 일반 파일에 대한 write와 같은 방식으로 할 수 있다는 것을 의미한다.

여기서의 파일 연산자의 write()에 해당하는 것이 바로 BSD의 sock_write()이다. 해당하는 코드는 ~/net/socket.c의 sock_write()이다.

```

static ssize_t sock_write(struct file *file, const char *ubuf,
                        size_t size, loff_t *ppos)
{
    struct socket *sock;
    struct msghdr msg;
    struct iovec iov;

    if (ppos != &file->f_pos)
        return -ESPIPE;
    if(size==0) /* Match SYS5 behaviour */
        return 0;

    sock = socki_lookup(file->f_dentry->d_inode);

    msg.msg_name=NULL;
    msg.msg_namelen=0;
    msg.msg_iov=&iov;
    msg.msg_iovlen=1;
    msg.msg_control=NULL;
    msg.msg_controllen=0;
    msg.msg_flags=!(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
    if (sock->type == SOCK_SEQPACKET)
        msg.msg_flags |= MSG_EOR;
    iov.iov_base=(void *)ubuf;
    iov.iov_len=size;

    return sock_sendmsg(sock, &msg, size);
}

```

코드 338. sock_write()

먼저 해당하는 파일 object에서 위치(position)을 확인한후, 옮바른 길이를 가지는 데이터의 전송인지를 확인한다. 해당하는 inode에서 socket 구조체를 찾은후, 메시지 구조체와 iovec구조체를 넘겨받은 파라미터 값으로 초기화 한다. 이와 같은 과정을 마쳤다면, sock_sendmsg()함수를 호출한다.

BSD 소켓 layer에서 사용하는 몇가지 데이터 구조를 정의하고 다음으로 넘어가도록 하자. 가장 먼저 접하게되는 것이 **socket** 구조체가 될 것이다. 아래와 같이 정의 된다.

```
struct socket
{
    socket_state          state;           /* 소켓의 상태 */
    unsigned long          flags;          /* 소켓이 가진 flags값 */
    struct proto_ops       *ops;           /* 소켓에 관련된 protocol의 연산자 */
    struct inode           *inode;          /* 소켓에 관련된 inode */
    struct fasync_struct  *fasync_list;   /* 비동기적인 wake up의 리스트 */
    struct file             *file;           /* 소켓에 관련된 file object에 대한 포인터 */
    struct sock             *sk;            /* sock 구조체 */
    wait_queue_head_t      wait;           /* 소켓에 대한 대기 큐 */
    short                  type;           /* 소켓의 타입 */
    unsigned char           passcred;        /* password credential */
};
```

코드 339. socket구조체

소켓의 상태로는 아래와 같은 값들이 올 수 있다.

```
typedef enum {
    SS_FREE = 0,                      /* 소켓이 현재 사용되지 않고 있다.*/
    SS_UNCONNECTED,                  /* 소켓이 연결되지 않은 상태이다.*/
    SS_CONNECTING,                   /* 소켓에 대한 연결이 진행중이다.*/
    SS_CONNECTED,                    /* 소켓이 연결상태에 있다.*/
    SS_DISCONNECTING                /* 소켓에 대해서 연결을 끊고 있는 중이다.*/
} socket_state;
```

또한 소켓의 타입에는 다시 아래와 같은 값들을 가질 수 있다.

```
#define SOCK_STREAM     1           /* Stream(Connection oriented) 소켓*/
#define SOCK_DGRAM      2           /* Datagram(Connectionless) 소켓 */
#define SOCK_RAW         3           /* Raw 소켓 */
#define SOCK_RDM         4           /* Reliably-Delivered Message 소켓*/
#define SOCK_SEQPACKET   5           /* 연속적인(sequential) 패킷 소켓*/
#define SOCK_PACKET      10          /* 디바이스 레벨에서 패킷을 가질수 있는
                                         리눅스만의 독특한 방법으로 RARP와 같은 것을 사용자
                                         레벨에서 쓸(write) 수 있도록 한다.*/
```

소켓 타입에 대해서, SOCK_STREAM은 TCP에, 그리고, SOCK_DGRAM은 UDP에 대당한다. 또한 SOCK_RAW는 IP 패킷을 주고 받기 위해서 사용한다.

sys_write() 시스템 콜에서 보았던 메시지(message)구조체에 대해서도 잠시 살펴보도록 하자. 이 메시지 구조체는 sock_sendmsg()함수에 대한 파라미터로 넘겨지는 자료구조로서 아래와 같이 정의 된다. ~/include/linux/socket.h를 참조하기 바란다.

```
struct msghdr {
```

```

void * msg_name; /* 소켓의 이름 */
int msg_namelen; /* 소켓 이름의 길이 */
struct iovec * msg_iov; /* 전달할 데이터 블록을 가리키는 포인터 */
__kernel_size_t msg iovlen; /* 블록의 수*/
void * msg_control; /* Per protocol magic (eg BSD file descriptor passing) */
__kernel_size_t msg_controllen; /* cmsg 리스트의 길이 */
unsigned msg_flags; /* 메시지의 플래그 값 */
};

```

코드 340.메시지 헤더의 정의

메시지에 넣을 데이터는 iovec구조체로 표현된다. 정의는 아래와 같다. ~/include/linux/uio.h를 참고하기 바란다.

```

struct iovec
{
    void *iov_base; /* BSD uses caddr_t (1003.1g requires void *) */
    __kernel_size_t iov_len; /* Must be size_t (1003.1g) */
};

```

코드 341.iovec구조체의 정의

데이터를 가리키는 iov_base가 있고, 데이터의 길이를 가르키는 부분이 온다. 만약 O_NONBLOCK으로 write 연산을 했을 경우에는 메시지의 전송이 완료되기를 기다리지 않고 바로 복귀하게 되며, 그렇지 않을 때는 프로세스는 블록킹된다(물론 바로 전송이 되지 않은 경우가 해당).

sock_write()함수는 sock_sendmsg()함수를 결과적으로 호출하게 되며, sock_sendmsg()함수는 아래와 같이 정의된다.

```

int sock_sendmsg(struct socket *sock, struct msghdr *msg, int size)
{
    int err;
    struct scm_cookie scm;

    err = scm_send(sock, msg, &scm);
    if (err >= 0) {
        err = sock->ops->sendmsg(sock, msg, size, &scm);
        scm_destroy(&scm);
    }
    return err;
}

```

코드 342.sock_sendmsg()함수의 정의

scm_send()¹³⁷함수는 소켓 레벨에서 제어(control)메시지에 대한 처리를 담당하며, 에러가 없을 경우에는 0 이상의 값을 돌려주게 되며, 결국 socket의 연산자인 sendmsg()함수를 호출해서 전송을 시작한다. 만약 처리도중에 에러가 있다면, 0보다 작은 값을 돌려준다.

socket에 대한 연산자 ops에는 프로토콜(protocol)연산자가 들어가게 되며, proto_ops구조체로 정의되며, 아래와 같다. ~/include/net.h를 보도록 하자.

```

struct proto_ops {
    int family;
    int (*release)(struct socket *sock);
    int (*bind) (struct socket *sock, struct sockaddr *umyaddr,

```

¹³⁷ scm에 대한 정의는 ~/include/net/scm.h와 ~/net/core/scm.c를 참고하기 바란다.

```

        int sockaddr_len);
int (*connect) (struct socket *sock, struct sockaddr *uservaddr,
                int sockaddr_len, int flags);
int (*socketpair) (struct socket *sock1, struct socket *sock2);
int (*accept) (struct socket *sock, struct socket *newsock,
               int flags);
int (*getname) (struct socket *sock, struct sockaddr *uaddr,
                int *usockaddr_len, int peer);
unsigned int (*poll) (struct file *file, struct socket *sock, struct poll_table_struct *wait);
int (*ioctl) (struct socket *sock, unsigned int cmd,
              unsigned long arg);
int (*listen) (struct socket *sock, int len);
int (*shutdown) (struct socket *sock, int flags);
int (*setsockopt) (struct socket *sock, int level, int optname,
                   char *optval, int optlen);
int (*getsockopt) (struct socket *sock, int level, int optname,
                   char *optval, int *optlen);
int (*sendmsg) (struct socket *sock, struct msghdr *m, int total_len, struct scm_cookie *scm);
int (*recvmsg) (struct socket *sock, struct msghdr *m, int total_len, int flags, struct scm_cookie *scm);
int (*mmap) (struct file *file, struct socket *sock, struct vm_area_struct * vma);
};

}

```

코드 343. proto_ops구조체의 정의

즉, proto_ops구조체는 사용하게 될 프로토콜이 제공하는 서비스를 정의해 놓은 것이다. 만약 우리가 TCP를 사용하게 된다면, TCP layer에서 위에서 정의한 함수들을 제공할 것이다. 이하에서는 각각의 프로토콜에 따라서 달리 반응하는 코드가 수행될 것이다.

지금까지 BSD socket layer를 설명해왔다. 조금 구분을 하자면 BSD socket layer는 사용자와의 socket interface의 연결을 지원하기 위한 layer라고 생각하면되고, INET socket layer는 내부적으로 internet address family를 위한 layer라고 생각하면 된다. 이제부터 보는 부분은 INET socket layer에 대한 이야기다.

5.5. INET Socket Layer

INET socket layer는 IP에 기반한 TCP나 UDP 프로토콜의 통신을 관리하는 역할을 한다. 먼저 INET socket layer에서 정의하는 proto_ops구조체를 보도록 하자. [~/net/ipv4/af_inet.c¹³⁸](#)에 정의되어 있다.

```

struct proto_ops inet_stream_ops = {
    family: PF_INET,
    release: inet_release,
    bind: inet_bind,
    connect: inet_stream_connect,
    socketpair: sock_no_socketpair,
    accept: inet_accept,
    getname: inet_getname,
    poll: tcp_poll,
    ioctl: inet_ioctl,
    listen: inet_listen,
    shutdown: inet_shutdown,
    setsockopt: inet_setsockopt,
    getsockopt: inet_getsockopt,
    sendmsg: inet_sendmsg,
    recvmsg: inet_recvmsg,
    mmap: sock_no_mmap
};

```

¹³⁸ 이곳에서는 IPv4에 대해서만 다루기로 한다. IPv6에 대한 것은 이것을 바탕으로해서 보면 될 것이다.

```
struct proto_ops inet_dgram_ops = {
    family: PF_INET,
    release: inet_release,
    bind: inet_bind,
    connect: inet_dgram_connect,
    socketpair: sock_no_socketpair,
    accept: sock_no_accept,
    getname: inet_getname,
    poll: datagram_poll,
    ioctl: inet_ioctl,
    listen: sock_no_listen,
    shutdown: inet_shutdown,
    setsockopt: inet_setsockopt,
    getsockopt: inet_getsockopt,
    sendmsg: inet_sendmsg,
    recvmsg: inet_recvmsg,
    mmap: sock_no_mmap,
};
```

코드 344. INET proto_ops의 정의

TCP와 UDP각각에 대해 INET socket layer의 인터페이스를 정의해 놓고 있다. 이렇게 정의된 interface는 이전에 보았던 BSD socket layer에서 해당하는 연산에 대해서 각각의 소켓 타입에 맞춰서 호출된다. 예를 들어서 sys_write()에서는 TCP를 사용할 경우 sock_sendmsg()함수는 sock->ops->sendmsg() 함수에서 inet_sendmsg()함수를 호출하게 될 것이다.

inet_sendmsg()함수를 보도록 하자. 코드는 역시 ~/net/ipv4/af_inet.c를 참고하기 바란다.

```
int inet_sendmsg(struct socket *sock, struct msghdr *msg, int size,
                 struct scm_cookie *scm)
{
    struct sock *sk = sock->sk;

    /* We may need to bind the socket. */
    if (sk->num==0 && inet_autobind(sk) != 0)
        return -EAGAIN;
    return sk->prot->sendmsg(sk, msg, size);
}
```

소켓이 이미 생성되어 있으므로 소켓의 sk 필드를 가지고 오면, 그곳에 관련된 프로토콜의 해당하는 sendmsg()함수를 불러올 수 있다. 따라서, inet_sendmsg()함수는 해당하는 프로토콜의 sendmsg()함수를 호출하면 된다.

이곳에서 잠시 INET에서 사용하는 **sock**구조체에 대해서 보기로 하자. 이하에서 설명할 때 참고로 하면 될 것이다.. ~/include/net/sock.h에 정의되어 있다

Field	Description
__u32 daddr	목적지의 IP address(4byte: XXX.XXX.XXX.XXX)
__u32 rcv_saddr	Bind된 지역(자신의) IP address(4byte:XXX.XXX.XXX.XXX)
__u16 dport	목적지의 port번호
unsigned short num	지역 port 번호
int bound_dev_if	0이 아닐 경우에 device에 대한 index로 사용
struct sock *next, **pprev, *bind_next, **bind_pprev	다양한 프로토콜들에 대한 sock구조체의 hash table을 유지
volatile unsigned char state	연결 상태

volatile unsigned char zapped	AX25와 IPX에서 연결되지 않은 것을 나타냄
__u16 sport	발신자의 port번호
unsigned short family	Address Family(예를 들어서 AF_INET: Internet address family)
unsigned char reuse	SO_REUSEADDR의 설정을 위해
unsigned char shutdown	소켓의 연결이 끊어짐
atomic_t refcnt	Reference Count값(현재 sock을 사용하는 카운트)
socket_lock_t lock	sock에 대한 lock
int rcvbuf	Receive buffer의 크기(byte단위)
wait_queue_head_t *sleep	sock의 wait queue
struct dst_entry *dst_cache	목적지의 cache
rwlock_t dst_lock	목적지에 대한 read write lock
atomic_t rmem_alloc	얼마나 많은 메모리를 read를 위해서 현재 소켓이 요청했나?
struct sk_buff_head receive_queue	들어오는 packet에 대한 queue
atomic_t wmem_alloc	얼마나 많은 메모리를 write를 위해서 현재 소켓이 요청했나?
struct sk_buff_head write_queue	나가는(전달되는) Packet에 대한 queue
atomic_t omem_alloc	얼마나 많은 메모리를 option을 위해서 현재 소켓이 요청했나?
int wmem_queued	불변 write queue의 크기
int forward_alloc	선(forward) 할당된 공간의 크기
__u32 saddr	보내는 source의 주소
unsigned int allocation	할당 모드
int sndbuf	전송 버퍼의 크기(bytes)
struct sock *prev	Hash 연결 리스트의 앞에 있는 sock구조체에 대한 포인터
volatile char dead, done, urginline, keepopen, linger, destroy, no_check, broadcast, bsdism	소켓에 대해서 설정되는 값들을 위한 변수들
unsigned char debug	Debugging
unsigned char rcvstamp	Receive time stamp
unsigned char userlocks	소켓 사용자 lock
int proc	Out of band 데이터를 받았을 때 signal을 전송받을 프로세스나 프로세스의 그룹
unsigned long lingertime	Linger option을 사용할 경우 기다리는 시간
int hashent	Hash table entry값
struct sock *pair	accept()함수에서 새로운 sock구조체를 만들때 생기는 새 sock 구조체에 대한 포인터
struct{ struct sk_buff *head; struct sk_buff *tail; } backlog	Backlog queue. sk_buff구조체의 연결 리스트
rwlock_t callback_lock	Call back함수를 위한 read/write lock
struct sk_buff_head error_queue	에러 sk_buff구조체의 큐
struct proto *prot	소켓과 관련된 프로토콜의 연산 벡터를 가리키는 포인터
#if defined() net_pinfo, tp_pinfo #endif	Protocol의 information 및 option들을 가지는 필드들
int err, err_soft	에러가 발생했음을 알림. C에서 errno와 같은 역할을 함
unsigned short ack_backlog, max_ack_backlog	Acknowledge된 backlog과 그것의 최대 값
__u32 priority	소켓의 우선순위
unsigned short type	BSD socket 구조체로 부터 넘겨받은 소켓의 type
unsigned char localroute	패킷이 지역적으로만 route되어야 함
unsigned char protocol	소켓에 관련된 protocol
struct ucred peercred	상대편 사용자의 사용자 신용정보(credential)

int rcvlowat	Receive low wait값
long rcvtimeo	Receive timeout
long sndtimeo	Send timeout
#ifdef CONFIG_FILTER struct filter *filter; #endif	소켓 필터의 명령어를 위한 구조체(어떤 패킷들을 걸러내거나 어떤 처리를 해줄 것인지를 정함)
union { } protinfo	각각의 address family에 필요한 정보를 저장
struct timer_list timer	소켓의 clean up timer들에 대한 리스트
struct timeval	소켓의 Timestamp, 각각의 패킷이 도착할 때마다 갱신됨
struct socket *socket	sock구조체와 관련된 BSD socket구조체에 대한 포인터, 이를 이용해서 IO signal을 보낸다.
void *user_data	RPC layer를 위한 데이터의 포인터
void (*state_change)(struct sock *sk)	소켓의 상태 변화시에 호출되는 함수에 대한 포인터
void (*data_ready)(struct sock *sk, int bytes)	데이터를 받았을 경우에 호출되는 함수에 대한 포인터
void (*write_space)(struct sock *sk)	Write연산을 위해서 사용할 수 있는 메모리가 있을 때 호출되는 함수에 대한 포인터
void (*err_report)(struct sock *sk)	에러가 발생했을 때 호출되는 함수에 대한 포인터
int (*backlog_rcv)(struct sock *sk, struct sk_buff *skb)	Backlog에 대한 receive연산에 호출되는 함수에 대한 포인터
void (*destruct)(struct sock *sk)	sock구조체를 없애기(destruction) 위해서 호출되는 함수에 대한 포인터

표 40.sock 구조체의 필드정의

sock구조체에 대한 이야기 중에서 sk_buff구조체에 대한 이야기가 나오는데, 이것은 나중에 Network device driver를 보게될 때도 다시 거론 될 것이다. sk_buff구조체는 소켓연결에서 사용하는 구조체로 사용자의 데이터와 각종 프로토콜의 정보를 담는데 쓰인다. 커널내에서 메모리간의 복사(copy) 횟수를 줄여주며, 각각의 프로토콜에서 자신만의 정보를 나타내기에 용이한 점이 있다.

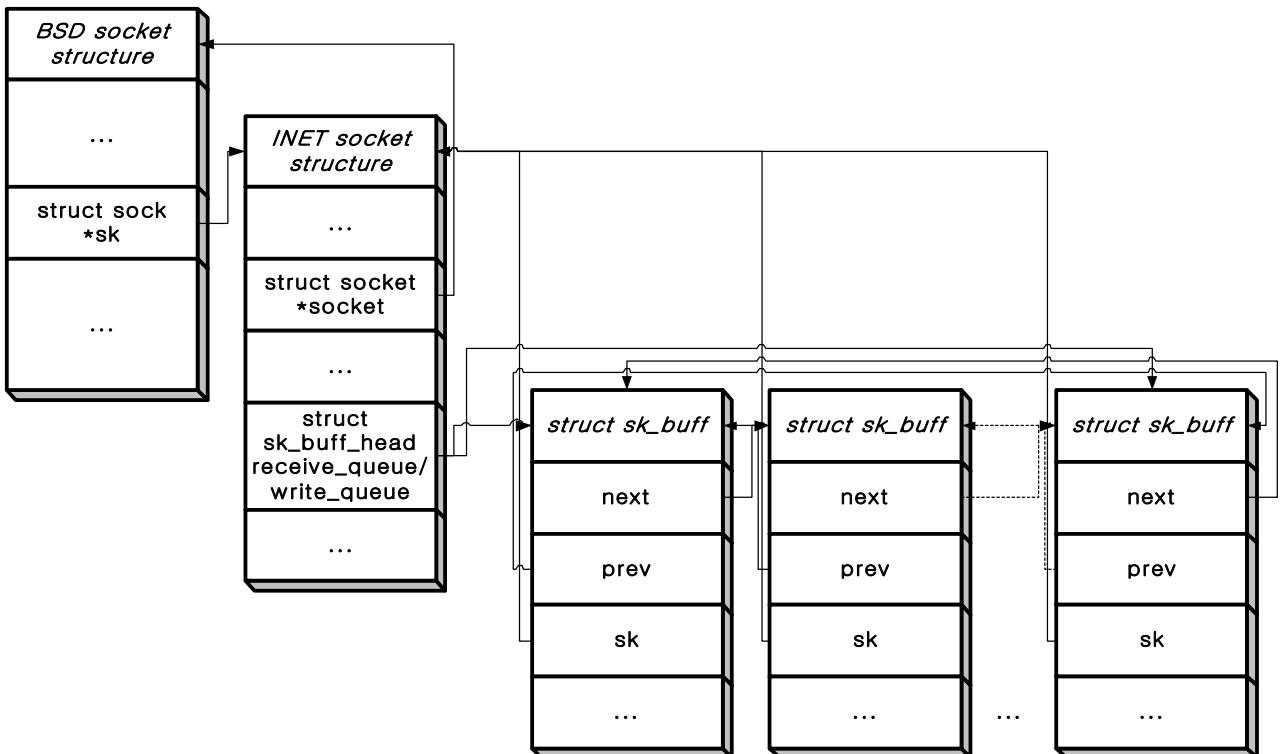
소켓 버퍼(socket buffer) 구조체인 sk_buff구조체는 아래와 같이 정의 된다. ~/include/linux/skbuff.h를 참고하기 바란다.

Field	Description
struct sk_buff *next, *prev	sk_buff list의 다음과 이전을 가리키는 포인터
struct sk_buff_head *list	현재 sk_buff가 속한 sk_buff_head에 대한 포인터
struct sock *sk	현재 sk_buff를 사용하는 INET sock에 대한 포인터
struct timeval stamp	sk_buff가 도착한 시간
struct net_device *dev	받거나 보내는 Network device의 구조체에 대한 포인터(Network device driver를 참고)
union{ } h	Transport layer(INET아래에 있는 layer)들의 header정보
union{ } nh	Network layer(Transport layer아래에 있는 layer)들의 header정보
union{ } mac	Mac layer(Medium Access Control: Link layer – Network layer아래에 있는 layer)의 header정보
struct dst_entry dst	목적지 주소의 entry에 대한 포인터
char cb[48]	Control buffer(모든 layer에서 사용 가능하다.)
unsigned int len	실제 데이터의 길이
unsigned int csum	Checksum
volatile char used	데이터가 사용자에게 넘어갔음을 나타냄
unsigned char cloned	sk_buff_head가 clone(복제)되었음을 나타냄
unsigned char pkt_type	패킷의 타입
unsigned char ip_summed	IP checksum으로 driver가 제공함

<code>__u32 priority</code>	패킷의 queueing 우선순위
<code>atomic_t users</code>	패킷의 사용자 카운트
<code>unsigned short protocol</code>	드라이버로부터 받은 패킷의 프로토콜
<code>unsigned short security</code>	패킷의 보안 레벨
<code>unsigned int truesize</code>	버퍼의 실제 크기
<code>unsigned char *head</code>	데이터 버퍼의 head를 가리키는 포인터
<code>unsigned char *data</code>	데이터 버퍼의 데이터 부분을 가리키는 포인터
<code>unsigned char *tail</code>	데이터 버퍼의 데이터 마지막 부분을 가리키는 포인터
<code>unsigned char *end</code>	데이터 버퍼의 마지막을 가리키는 포인터
<code>void (*destructor)(struct sk_buff)</code>	sk_buff 구조체를 없애는 함수에 대한 포인터
...	나머지 커널의 구성(configuration)에 따른 변수들

표 41. `sk_buff`구조체의 필드들.

이상에서 설명한 BSD socket의 구조체와 INET socket, `sk_buff` 구조체를 연결지어서 보면 아래의 [그림42]와 같다.

그림 49. BSD socket 및 INET sock, `sk_buff`의 상관 관계

[그림42]는 하나의 BSD socket에 하나의 INET sock구조체를, 그리고, INET sock구조체에 대해서 들어오거나 가갈 패킷을 가리키는 `sk_buff`구조체를 관련지어서 보여주고 있다. 받거나 보낼 패킷들은 전부 `sock`구조체에 연결되어 처리를 기다리게 된다. 소켓버퍼에 대한 자세한 이야기는 Network Device Driver를 참고하라. 이곳에선 `sk_buff`라는 단위로 데이터를 하위의 physical layer로 보내고, 다시 그곳에서 `sk_buff`의 단위로 데이터를 받아서, 해당하는 `sock`구조체에 연결지어준다는 것만 기억하기 바란다.

여기서 `sk_buff_head`구조체는 `sk_buff`들의 원형(circular) 큐를 관리하기 위해서 쓰이며, 아래와 같이 정의된다.

```
struct sk_buff_head {
```

```
/* These two members must be first. */
struct sk_buff *next;
struct sk_buff *prev;
__u32 qlen;
spinlock_t lock;
};
```

코드 345. `sk_buff_head` 구조체의 정의

즉, `sk_buff`를 가리키는 `next`와 `prev`필드, 현재 `queue`에 저장된 `sk_buff`가 얼마인지를 보여주는 `qlen`, 그리고, `sk_buff_head`에 대한 동기화(synchronization) 접근을 위한 `spinlock_t`의 `lock`으로 구성된다.

이전의 이야기로 다시 돌아가서 `sk->prot->sendmsg()`를 보도록 하자. 쓰기 연산이 아직 끝나지 않았다. 만약 하위의 프로토콜이 TCP라고 한다면, TCP 프로토콜에서 연산자로 가지는 `sendmsg()`함수가 호출 될 것이다. TCP 프로토콜에서 제공하는 연산자들은 아래와 같다. `~/include/net/tcp_ipv4.c`에 정의되어 있다.

```
struct proto tcp_prot = {
    name: "TCP",
    close: tcp_close,
    connect: tcp_v4_connect,
    disconnect: tcp_disconnect,
    accept: tcp_accept,
    ioctl: tcp_ioctl,
    init: tcp_v4_init_sock,
    destroy: tcp_v4_destroy_sock,
    shutdown: tcp_shutdown,
    setsockopt: tcp_setsockopt,
    getsockopt: tcp_getsockopt,
    sendmsg: tcp_sendmsg,
    recvmsg: tcp_recvmsg,
    backlog_rcv: tcp_v4_do_rcv,
    hash: tcp_v4_hash,
    unhash: tcp_unhash,
    get_port: tcp_v4_get_port,
};
```

코드 346. TCP Protocol에서 제공하는 함수의 정의

TCP프로토콜의 연산자인 `tcp_sendmsg()`함수가 `sendmsg`로 정의되어 있으므로, 이젠 드디어 TCP프로토콜의 `tcp_sendmsg()`함수를 볼 차례가 되었다. `~/include/net/tcp.c`에서 찾을 수 있다.

5.6. TCP and UDP Layer

TCP와 UDP layer에서는 사용자로부터 전달 받은 데이터를 실제의 `sk_buff`형태로 만들고, 이를 하위의 IP layer로 전달하는 역할을 한다. TCP와 UDP는 가장 널리 알려진 인터넷 프로토콜이며, 아래에서 살펴볼 IP 레이어와 같이 TCP/IP로 불린다. TCP와 UDP는 각각 사용하는 서비스의 질에서 차이가 있으며, 이것은 연결성을 지향하는가 아니면, 그렇지 않은가로 구분한다. 즉, 하위의 layer에는 이 layer에서 만들어진 데이터를 단순히 보내는 역할만 한다는 것이다.

`tcp_sendmsg()`함수를 보자. 이 함수의 주 목적은 사용자로부터 넘겨받은 데이터를 `sk_buff`형태로 만드는 것과 실제 전송을 시작하는 역할을 한다.

```
int tcp_sendmsg(struct sock *sk, struct msghdr *msg, int size)
{
    struct iovec *iov;
    struct tcp_opt *tp;
    struct sk_buff *skb;
```

```
int iovlen, flags;
int mss_now;
int err, copied;
long timeo;
```

코드 347. tcp_sendmsg().

먼저 tcp_sendmsg() 함수에서 사용할 지역 변수에 대한 정의다. iovlen은 msghdr 구조체로부터 초기화가 되며, tcp_opt는 sock 구조체의 union으로 정의된 부분에서 찾을 수 있다. 또한 새로이 할당 받은 sk_buff 구조체에 대한 포인터와 각종 flag들, timeout값들을 위한 변수가 온다.

```
err = 0;
tp = &(sk->tp_pinfo.af_tcp);
lock_sock(sk);
TCP_CHECK_TIMER(sk);
flags = msg->msg_flags;
timeo = sock_sndtimeo(sk, flags&MSG_DONTWAIT);
```

코드 348. tcp_sendmsg() – continued.

에러 값을 초기화하고, sock 구조체로 부터 TCP에 해당하는 프로토콜 information에 대한 포인터를 구한다. 또한 이제부터 sock에 대한 연산을 하기 위해서 sock 구조체에 대해서 lock을 취한다. sock 구조체에 대해서 timer가 있는지를 확인하고, 메시지 헤더로부터 flag를 얻어 소켓의 send timeout 값을 구한다.

```
/* Wait for a connection to finish. */
if ((1 << sk->state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT))
    if((err = wait_for_tcp_connect(sk, flags, &timeo)) != 0)
        goto out_unlock;
```

코드 349. tcp_sendmsg() – continued.

소켓의 connection이 이루어 질 때까지 기다린다. timeout 값을 주어서 얼마나 기다릴지를 결정한다.

```
/* This should be in poll */
clear_bit(SOCK_ASYNC_NOSPACE, &sk->socket->flags);
mss_now = tcp_current_mss(sk);
/* Ok commence sending. */
iovlen = msg->msg iovlen;
iov = msg->msg iov;
copied = 0;
```

코드 350. tcp_sendmsg() - continued.

BSD socket에 대해서 SOCK_ASYNC_NOSPACE bit를 지우고, 현재 TCP 메시지 segment의 크기를 구한다. 그리고나서 전송을 시작하는 단계이다. 먼저 보내고자 하는 데이터의 크기를 iovlen 값으로 둔다. 그리고, iov 값으로는 메시지 헤더의 msg iov 값으로 초기화하고, 복사 count 값을 0으로 초기화 한다.

```
while (--iovlen >= 0) {
    int seglen=iov->iov_len;
    unsigned char * from=iov->iov_base;
    iov++;
    while (seglen > 0) {
        int copy, tmp, queue_it;
        if (err)
            goto do_fault2;
        /* Stop on errors. */
        if (sk->err)
```

```

        goto do_sock_err;
/* Make sure that we are established. */
if (sk->shutdown & SEND_SHUTDOWN)
    goto do_shutdown;
/* Now we need to check if we have a half
 * built packet we can tack some data onto.
 */
skb = sk->write_queue.prev;
if (tp->send_head &&
    (mss_now - skb->len) > 0) {
    copy = skb->len;
    if (skb_tailroom(skb) > 0) {
        int last_byte_was_odd = (copy % 4);
        copy = mss_now - copy;
        if(copy > skb_tailroom(skb))
            copy = skb_tailroom(skb);
        if(copy > seglen)
            copy = seglen;
        if(last_byte_was_odd) {
            if(copy_from_user(skb_put(skb, copy),
                              from, copy))
                err = -EFAULT;
            skb->csum = csum_partial(skb->data,
                                      skb->len, 0);
        } else {
            skb->csum =
                csum_and_copy_from_user(
                    from, skb_put(skb, copy),
                    copy, skb->csum, &err);
        }
        tp->write_seq += copy;
        TCP_SKB_CB(skb)->end_seq += copy;
        from += copy;
        copied += copy;
        seglen -= copy;
        if (PSH_NEEDED ||
            after(tp->write_seq, tp->pushed_seq+(tp->max_window>>1))) {
            TCP_SKB_CB(skb)->flags |= TCPCB_FLAG_PSH;
            tp->pushed_seq = tp->write_seq;
        }
        if (flags&MSG_OOB) {
            tp->urg_mode = 1;
            tp->snd_up = tp->write_seq;
            TCP_SKB_CB(skb)->sacked |= TCPCB_URG;
        }
        continue;
    } else {
        TCP_SKB_CB(skb)->flags |= TCPCB_FLAG_PSH;
        tp->pushed_seq = tp->write_seq;
    }
}
copy = min(seglen, mss_now);
/* Determine how large of a buffer to allocate. */
tmp = MAX_TCP_HEADER + 15 + tp->mss_cache;
if (copy < mss_now && !(flags & MSG_OOB)) {
    queue_it = 1;
} else {
    queue_it = 0;
}

```

```

        }
        skb = NULL;
        if (tcp_memory_free(skb))
            skb = tcp_alloc_skb(sk, tmp, sk->allocation);
        if (skb == NULL) {
            /* If we didn't get any memory, we need to sleep. */
            set_bit(SOCK_ASYNC_NOSPACE, &sk->socket->flags);
            set_bit(SOCK_NOSPACE, &sk->socket->flags);
            __tcp_push_pending_frames(sk, tp, mss_now, 1);
            if (!timeo) {
                err = -EAGAIN;
                goto do_interrupted;
            }
            if (signal_pending(current)) {
                err = sock_intr_errno(timeo);
                goto do_interrupted;
            }
            timeo = wait_for_tcp_memory(sk, timeo);
            /* If SACK's were formed or PMTU events happened,
             * we must find out about it.
             */
            mss_now = tcp_current_mss(sk);
            continue;
        }
        seglen -= copy;
        /* Prepare control bits for TCP header creation engine. */
        if (PSH_NEEDED ||
            after(tp->write_seq+copy, tp->pushed_seq+(tp->max_window>>1))) {
            TCP_SKB_CB(skb)->flags = TCPCB_FLAG_ACK|TCPCB_FLAG_PSH;
            tp->pushed_seq = tp->write_seq + copy;
        } else {
            TCP_SKB_CB(skb)->flags = TCPCB_FLAG_ACK;
        }
        TCP_SKB_CB(skb)->sacked = 0;
        if (flags & MSG_OOB) {
            TCP_SKB_CB(skb)->sacked |= TCPCB_URG;
            tp->urg_mode = 1;
            tp->snd_up = tp->write_seq + copy;
        }
        /* TCP data bytes are SKB_PUT() on top, later
         * TCP+IP+DEV headers are SKB_PUSH()'d beneath.
         * Reserve header space and checksum the data.
         */
        skb_reserve(skb, MAX_TCP_HEADER);
        skb->csum = csum_and_copy_from_user(from,
                                              skb_put(skb, copy), copy, 0, &err);
        if (err)
            goto do_fault;
        from += copy;
        copied += copy;
        TCP_SKB_CB(skb)->seq = tp->write_seq;
        TCP_SKB_CB(skb)->end_seq = TCP_SKB_CB(skb)->seq + copy;
        /* This advances tp->write_seq for us. */
        tcp_send_skb(sk, skb, queue_it, mss_now);
    }
}

```

코드 351. tcp_sendmsg() – continued.

이부분은 while loop를 보낼 데이터가 있을때까지 반복적으로 시행하는 부분이다. 여기서 sk_buff구조체를 생성한다. 즉, 사용자로 부터 데이터를 넘겨받아서, sk_buff를 write queue에 할당한다음, sk_buff의 data를 나타내는 부분에 사용자 데이터를 복사한다. 만약 sk_buff를 할당할 만한 공간이 시스템에 없다면, 프로세스는 기다림 상태가 되며, 이미 전달하고 하는 패킷이 완성된 형태로 있지 않다면, 그것도 완성시켜준다. 또한 이곳에서는 checksum값을 사용자의 데이터를 복사하는 과정에서 생성하며(csum_and_copy_from_user()), TCP 패킷의 sequence 번호도 생성한다. 생성된 sk_buff는 tcp_send_skb()를 통해서 전송을 시작한다.

```

err = copied;
out:
    __tcp_push_pending_frames(sk, tp, mss_now, tp->nonagle);
out_unlock:
    TCP_CHECK_TIMER(sk);
    release_sock(sk);
    return err;
do_sock_err:
    if (copied)
        err = copied;
    else
        err = sock_error(sk);
    goto out;
do_shutdown:
    if (copied)
        err = copied;
    else {
        if (!(flags&MSG_NOSIGNAL))
            send_sig(SIGPIPE, current, 0);
        err = -EPIPE;
    }
    goto out;
do_interrupted:
    if (copied)
        err = copied;
    goto out_unlock;
do_fault:
    __kfree_skb(skb);
do_fault2:
    if (copied)
        err = copied;
    else
        err = -EFAULT;
    goto out;
}

```

코드 352. `tcp_sendmsg()` – continued.

이제 남은 것은 `tcp_sendmsg()`함수를 종료하는 것이다. 적절한 에러처리와 에러코드를 돌려주면 될 것이다. 여기서 제대로 보내고자 하는 데이터의 처리가 완료되었다면, `__tcp_push_pending_frames()`함수가 호출되어 생성중인 TCP패킷을 보관하도록 한다. 다시 보내는 시간을 기록하게 되며(`TCP_CHECK_TIMER()`), `sock`구조체에 대한 lock을 풀고나서, 에러코드를 돌려주며 복귀한다. 만약 소켓의 연결이 끊어졌고, 이 경우에 시그널을 받고자 원한다면, 현재의 프로세스(`current`)에 SIGPIPE 시그널을 보낸 후, -EPIPE를 에러코드로 넘겨준다.

이젠 소켓 버퍼의 전달에 대해서 보기 위해 `tcp_send_skb()`함수를 보도록 하자. 이 함수는 `~/net/ipv4/tcp_output.c`에 정의되어 있다.

```

void tcp_send_skb(struct sock *sk, struct sk_buff *skb, int force_queue, unsigned cur_mss)
{
    struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);

    /* Advance write_seq and place onto the write_queue. */
    tp->write_seq = TCP_SKB_CB(skb)->end_seq;
    __skb_queue_tail(&sk->write_queue, skb);
    tcp_charge_skb(sk, skb);
    if (!force_queue && tp->send_head == NULL && tcp_snd_test(tp, skb, cur_mss, tp->nonagle)) {
        /* Send it out now. */
        TCP_SKB_CB(skb)->when = tcp_time_stamp;
        if (tcp_transmit_skb(sk, skb_clone(skb, sk->allocation)) == 0) {
            tp->snd_nxt = TCP_SKB_CB(skb)->end_seq;
            tcp_minshall_update(tp, cur_mss, skb);
            if (tp->packets_out++ == 0)
                tcp_reset_xmit_timer(sk, TCP_TIME_RETRANS, tp->rto);
            return;
        }
    }
    /* Queue it, remembering where we must start sending. */
    if (tp->send_head == NULL)
        tp->send_head = skb;
}

```

코드 353. tcp_send_skb()함수의 정의

tcp_send_skb()함수는 주(main) 보내기 함수로서 보낼 sk_buff를 queueing할 것인지, 아니면 지금 보낼지를 결정한다. 먼저 write sequence를 증가시키고, 생성된 sk_buff를 sock구조체의 write_queue에 넣는다 (__skb_queue_tail()). tcp_charge_skb()함수는 단순히 sock구조체의 wmem_queued필드와 forward_alloc필드의 값을 update하는 역할을 한다. 이제는 보낼 것인지 아니면 queueing만 할 것인지를 결정하는 부분이다. 보내기로 했다면, 보내는 시점을 기록하고 (tcp_time_stamp=jiffies), tcp_transmit_skb()함수를 호출한다.

보내기가 잘 되었다면, 보낸 sequence를 업데이트하고, 작은(small) 패킷에 대해서 다음번에 어디서부터 전송을 시작할지를 기록하며 (tcp_minshall_update()), 보낸 패킷의 카운트를 증가 시킨다 (tp->packets_out++). 만약 보낸 패킷의 카운트 값이 0이 된다면, 전송타이머(transmission timer)를 재설정(reset)한다 (tcp_reset_xmit_timer()).

Queueing하기로 결정했다면, sock 구조체의 TCP를 위한 구조체내의 send_head를 살펴서 NULL 값을 갖는다면, 다음번 전송의 시작을 sk_buff를 가리키도록하고 종료한다.

실제적인 전송은 다시 tcp_transmit_skb()함수를 통한다. 아래와 같다. 역시 ~/net/ipv4/tcp_out.c를 참조하기 바란다. 이 함수가 하는 역할은 앞에서 queueing된 패킷(sk_buff)에 대한 실제적인 전송을 담당한다.

```

int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb)
{
    if(skb != NULL) {
        struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
        struct tcp_skb_cb *tcb = TCP_SKB_CB(skb);
        int tcp_header_size = tp->tcp_header_len;
        struct tcphdr *th;
        int sysctl_flags;
        int err;

#define SYSCTL_FLAG_TSTAMPS 0x1
#define SYSCTL_FLAG_WSCALE 0x2
#define SYSCTL_FLAG_SACK 0x4

        sysctl_flags = 0;
    }
}

```

코드 354. tcp_transmit_skb() 함수.

tcp_transmit_skb()함수에서 사용할 지역변수를 적절하게 설정하는 부분이다. #define으로 정의된 상수는 각각 Timestamp와 Window Scale, Select Acknowledgement를 정의하는 것이다.

```
if (tcb->flags & TCPCB_FLAG_SYN) {
    tcp_header_size = sizeof(struct tcphdr) + TCPOLEN_MSS;
    if(sysctl_tcp_timestamps) {
        tcp_header_size += TCPOLEN_TSTAMP_ALIGNED;
        sysctl_flags |= SYSCTL_FLAG_TSTAMPS;
    }
    if(sysctl_tcp_window_scaling) {
        tcp_header_size += TCPOLEN_WSCALE_ALIGNED;
        sysctl_flags |= SYSCTL_FLAG_WSCALE;
    }
    if(sysctl_tcp_sack) {
        sysctl_flags |= SYSCTL_FLAG_SACK;
        if(!(sysctl_flags & SYSCTL_FLAG_TSTAMPS))
            tcp_header_size += TCPOLEN_SACKPERM_ALIGNED;
    }
} else if (tp->eff_sacks) {
    /* A SACK is 2 pad bytes, a 2 byte header, plus
     * 2 32-bit sequence numbers for each SACK block.
     */
    tcp_header_size += (TCPOLEN_SACK_BASE_ALIGNED +
                        (tp->eff_sacks * TCPOLEN_SACK_PERBLOCK));
}
th = (struct tcphdr *) skb_push(skb, tcp_header_size);
skb->h.th = th;
skb_set_owner_w(skb, sk);
```

코드 355. tcp_transmit_skb() 함수 – continued.

sock 구조체에 들어가 있는 TCP control block의 flag설정에 따라서, TCP의 헤더(header)크기를 구하고, 이를 이용해서 sk_buff에 TCP 헤더가 차지할 공간을 마련해 둔다 (skb_push()). 그리고나서, sk_buff의 TCP를 위한 헤더정보를 나타내는 th필드를 초기화 한다. skb_set_owner_w()함수는 sk_buff가 현재 사용되고 있음과 누가 사용하는지(sock구조체), destructor함수 및 INET sock의 wmem_alloc필드를 업데이트 한다.

```
/* Build TCP header and checksum it.*/
th->source      = sk->sport;
th->dest        = sk->dport;
th->seq         = htonl(tcb->seq);
th->ack_seq     = htonl(tp->rcv_nxt);
*((__u16 *)th) + 6 = htons(((tcp_header_size >> 2) << 12) | tcb->flags);
if (tcb->flags & TCPCB_FLAG_SYN) {
    th->window     = htons(tp->rcv_wnd);
} else {
    th->window     = htons(tcp_select_window(sk));
}
th->check       = 0;
th->urg_ptr     = 0;
if (tp->urg_mode &&
    between(tp->snd_up, tcb->seq+1, tcb->seq+0xFFFF)) {
    th->urg_ptr     = htons(tp->snd_up-tcb->seq);
    th->urg          = 1;
}
if (tcb->flags & TCPCB_FLAG_SYN) {
```

```

        tcp_syn_build_options((__u32 *)(th + 1),
                              tcp_advertise_mss(sk),
                              (sysctl_flags & SYSCTL_FLAG_TSTAMPS),
                              (sysctl_flags & SYSCTL_FLAG_SACK),
                              (sysctl_flags & SYSCTL_FLAG_WSCALE),
                              tp->rcv_wscale,
                              tcb->when,
                              tp->ts_recent);
} else {
    tcp_build_and_update_options((__u32 *)(th + 1),
                                 tp, tcb->when);
    TCP_ECN_send(sk, tp, skb, tcp_header_size);
}
tp->af_specific->send_check(sk, th, skb->len, skb);
if (tcb->flags & TCPCB_FLAG_ACK)
    tcp_event_ack_sent(sk);
if (skb->len != tcp_header_size)
    tcp_event_data_sent(tp, skb);
TCP_INC_STATS(TcpOutSegs);

```

코드 356. `tcp_transmit_skb()` 함수 – continued

이전 TCP header와 TCP checksum을 생성하는 부분이다. 소스(source)와 목적지(destination)의 port번호 및 TCP sequence 번호와, ACK(acknowledge) sequence 번호를 설정한다. 또한 TCP의 window크기등의 정보들을 최기화 한다. 또한 TCP에 설정된 option들에 대한 해더정보도 이곳에서 설정된다. 마지막으로 데이터인지 ACK패킷인지를 확인해서 관련된 timer들을 설정하고, TCP의 통계정보를 update한다.

```

err = tp->af_specific->queue_xmit(skb);
if (err <= 0)
    return err;
tcp_enter_cwr(tp);
return err == NET_XMIT_CN ? 0 : err;
}
return -ENOBUFS;
#endif
#endif
#endif
}
```

코드 357.`tcp_transmit_skb()` 함수 – continued.

이전 하위의 전송 함수를 호출하는 일이다. 이것이 바로 `tp->af_specific->queue_xmit()` 함수이다. 먼저 `sock`구조체의 TCP를 위한 `tp_info`필드의 `af_specific`필드를 보자. 이것은 `tcp_func` 구조체로 정의되어 있으며, `tcp_func`구조체는 아래와 같다. (`~/include/net/tcp.h`를 참고하라.)

```

struct tcp_func {
    int (*queue_xmit)(struct sk_buff *skb);
    void (*send_check)(struct sock *sk,
                       struct tcphdr *th,
                       int len,
                       struct sk_buff *skb);

    int (*rebuild_header)(struct sock *sk);
    int (*conn_request)(struct sock *sk,
                        struct sk_buff *skb);

    struct sock *(*syn_recv_sock)(struct sock *sk,
                                 struct sk_buff *skb,
                                 struct open_request *req,
                                 struct dst_entry *dst);

    int (*hash_connecting)(struct sock *sk);
    int (*remember_stamp)(struct sock *sk);
    __u16 net_header_len;
    int (*setsockopt)(struct sock *sk,
                      int level,
                      int optname,
                      char *optval,
                      int optlen);

    int (*getsockopt)(struct sock *sk,
                      int level,
                      int optname,
                      char *optval,
                      int *optlen);

    void (*addr2sockaddr)(struct sock *sk,
                          struct sockaddr *);

    int sockaddr_len;
};

```

코드 358. **tcp_func**구조체의 정의

tcp_func구조체는 하위의 layer에서 제공해주어야 할 함수들에 대한 포인터들과 변수로 구성된다. 각각의 함수들은 TCP아래의 layer에서 정의하고 있으므로 IP부분을 살펴보아야 할 것이다. 현재 우리가 필요로 하는 함수는 `queue_xmit()`함수이다. **tcp_func**의 IPv4에 대한 정의는 아래와 같다. 정의는 `~/net/ipv4/tcp_ipv4.c`에 있다.

```

struct tcp_func ipv4_specific = {
    ip_queue_xmit,
    tcp_v4_send_check,
    tcp_v4_rebuild_header,
    tcp_v4_conn_request,
    tcp_v4_syn_recv_sock,
    tcp_v4_hash_connecting,
    tcp_v4_remember_stamp,
    sizeof(struct iphdr),
    ip_setsockopt,
    ip_getsockopt,
    v4_addr2sockaddr,
    sizeof(struct sockaddr_in)
};

```

코드 359. **TCP specific function**의 IPv4에 대한 정의

따라서, 우리가 관심을 가진 함수는 `ip_queue_xmit()`함수가 될 것이다. 이 함수는 IP layer에서 제공하고 있다. IP layer는 현재 IPv4와 IPv6로 나누어져 있다. 이것은 IP address의 부족이라는 이유 때문에

프로토콜에 대해서 변화를 주어야 할 필요가 생겨, 새로이 버전이 바뀜에 따라서 나타난 현상이다. 현재 IPv6는 아직 널리 유포되어서 사용되지는 않으며, 네트워크의 연동 테스트가 진행중이다. 여기서는 둘간의 차이점에 대해서는 논하지 않기로 한다.

5.7. IP Layer

IP 레이어의 역할은 상위에서 만들어진 데이터를 보내는 것과 하위에서 받은 데이터를 상위로 옮겨주는 역할을 한다. 이때 보내기 위한 데이터의 제조합이 일어나게되며, 물론 받은 데이터의 내용에 대해서는 신경쓰지 않으며, 다만 자신을 위한 헤더(header)나 옮바로 받았는지를 확인할 수 있는 CRC (Cyclic Redundancy Check) 정보 정도의 더하는 과정이 있게 된다.

ip_queue_xmit()함수를 보도록 하자. (~/net/ipv4/ip_output.c를 참조) 이 함수가 하는 역할은 넘겨받은 sk_buff를 디바이스에 맞게 다듬어서 보내는 역할을 한다. 따라서, 필요하다면 넘겨받은 데이터를 합치거나 나누는 일이 일어나게 되며, 데이터의 내용에는 관심을 두지 않는다.

```
int ip_queue_xmit(struct sk_buff *skb)
{
    struct sock *sk = skb->sk;
    struct ip_options *opt = sk->protinfo.af_inet.opt;
    struct rtable *rt;
    struct iphdr *iph;
```

코드 360. ip_queue_xmit()함수

ip_queue_xmit()함수가 넘겨받는 것은 sk_buff구조체 뿐이다. 앞에서 이미 TCP layer가 만든 sk_buff이다. 이곳에서는 sk_buff의 sock 구조체를 가르키는 필드를 찾아서 필요한 정보들을 얻을 수 있다. 즉, IP layer에서 필요한 option이 될 것이다. IP layer에서는 또한 어디로 보내는지에 대한 정보도 필요하다. IP 주소까지는 이미 알고 있으므로, IP 주소를 가지고 목적지에 해당하는 hardware주소를 찾아야 할 것이다. 이때 필요한 것이 바로 라우팅 테이블(routing table)이다(struct rtable).

```
/* Make sure we can route this packet. */
rt = (struct rtable *)__sk_dst_check(sk, 0);
if (rt == NULL) {
    u32 daddr;
    /* Use correct destination address if we have options. */
    daddr = sk->daddr;
    if(opt && opt->srr)
        daddr = opt->faddr;
    if (ip_route_output(&rt, daddr, sk->saddr,
                       RT_TOS(sk->protinfo.af_inet.tos) | RTO_CONN | sk->localroute,
                       sk->bound_dev_if))
        goto no_route;
    __sk_dst_set(sk, &rt->u.dst);
}
skb->dst = dst_clone(&rt->u.dst);
if (opt && opt->is_strictroute && rt->rt_dst != rt->rt_gateway)
    goto no_route;
```

코드 361. ip_queue_xmit()함수 – continued

라우팅 테이블에 대한 값을 구한다. 만약 이미 routing table에 목적지에 해당하는 값이 있다면 문제가 되지 않지만, 없다면 routing정보들 얻어야 할 것이다(__sk_dst_check()). 물론 routing table의 정보도 다시 업데이트 해야한다. 라우팅 테이블에 정보가 없다면(rt==NULL), 목적지의 주소(daddr)를 가져와서 ARP의

도움을 받도록 한다(ip_route_output()).¹³⁹ ARP(Address Resolution Protocol)은 IP 주소를 하드웨어 주소로 만드는 역할을 한다. 이젠 목적지의 주소도 결정되었다.

```

/* OK, we know where to send it, allocate and build IP header.*/
iph = (struct iphdr *) skb_push(skb, sizeof(struct iphdr) + (opt ? opt->optlen : 0));
*((__u16 *)iph) = htons((4 << 12) | (5 << 8) | (sk->protinfo.af_inet.tos & 0xff));
iph->tot_len = htons(skb->len);
iph->frag_off = 0;
iph->ttl = sk->protinfo.af_inet.ttl;
iph->protocol = sk->protocol;
iph->saddr = rt->rt_src;
iph->daddr = rt->rt_dst;
skb->nh.iph = iph;
/* Transport layer set skb->h.foo itself.*/
if(opt && opt->optlen) {
    iph->ihl += opt->optlen >> 2;
    ip_options_build(skb, opt, sk->daddr, rt, 0);
}
return NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev,
               ip_queue_xmit2);

no_route:
IP_INC_STATS(IpOutNoRoutes);
kfree_skb(skb);
return -EHOSTUNREACH;
}

```

코드 362. ip_queue_xmit() 함수 – continued

이제는 IP header를 정하는 일이 남았다. 물론 이것은 sk_buff의 데이터 부분에 들어가야 한다(sk_push()). 또한 sk_buff의 IP header를 가리키는 부분도 sk_buff의 IP header로 두도록 만든다(skb->nh.iph). 남은 것은 보내는 하위의 네트워크 디바이스 드라이버로 보내는 일만이 남았다. 보내고자 하는 데이터에 대한 모든 것은 sk_buff에 이미 들어가 있는 상태이다. sk_buff는 자신의 한 필드로 net_device구조체를 가진다. 이것은 sock구조체가 둑여진(bound)된 네트워크 디바이스의 index로부터 알 수 있다. 이제 이 네트워크 디바이스로 sk_buff를 보내면 되는 것이다(NF_HOOK()). 만약 경로(route)가 없다면, IP의 통계정보에 경로가 없는 패킷에 대한 연산이 있었음을 나타내고(IP_INC_STATS(IpOutNoRoutes), sk_buff구조체를 버리고(free_skb()) 복귀한다.

NF_HOOK()은 매크로이며, ~/linux/netfilter.h에 아래와 같이 정의되어 있다.

```

#ifndef CONFIG_NETFILTER
...
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn) \
(list_empty(&nf_hooks[(pf)][(hook)]) \
 ? (okfn)(skb) \
 : nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn)))
...
#endif
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn) (okfn)(skb)
#endif

```

코드 363. NF_HOOK() 매크로의 정의

¹³⁹ ip_route_output()함수는 나중에 나오는 디바이스 데이터 전송과 관련된 부분의 초기화를 맡고 있다.

지금 우리가 하고 있는 논의는 극히 단순화된 네트워크의 일반적인 흐름에 대한 것이므로 netfilter는 제외하기로 한다. 따라서, 현재는 CONFIG_NETFILTER¹⁴⁰ 가 정의(define)되지 않았기에, 단순히 제일 마지막에 있는 (okfn)(skb)가 호출된다. ip_queue_xmit()함수에서는 ip_queue_xmit2()함수가 될 것이다.

ip_queue_xmit2()함수의 정의는 아래와 같다. ~/net/ipv4/ip_output.c를 계속 참조하자.

```
static inline int ip_queue_xmit2(struct sk_buff *skb)
{
    struct sock *sk = skb->sk;
    struct rtable *rt = (struct rtable *)skb->dst;
    struct net_device *dev;
    struct iphdr *iph = skb->nh.iph;

    dev = rt->u.dst.dev;
```

코드 364. ip_queue_xmit2()함수

먼저 넘겨받은 sk_buff로 부터 필요한 정보를 가져온다. 이곳에서는 sock구조체와 net_device구조체, IP header정보 및 routing table이 필요하다.

```
/* This can happen when the transport layer has segments queued
 * with a cached route, and by the time we get here things are
 * re-routed to a device with a different MTU than the original
 * device. Sick, but we must cover it.
 */
if (skb_headroom(skb) < dev->hard_header_len && dev->hard_header) {
    struct sk_buff *skb2;
    skb2 = skb_realloc_headroom(skb, (dev->hard_header_len + 15) & ~15);
    kfree_skb(skb);
    if (skb2 == NULL)
        return -ENOMEM;
    if (sk)
        skb_set_owner_w(skb2, sk);
    skb = skb2;
    iph = skb->nh.iph;
}
if (skb->len > rt->u.dst.pmtu)
    goto fragment;
if (ip_dont_fragment(sk, &rt->u.dst))
    iph->frag_off |= __constant_htons(IP_DF);
ip_select_ident(iph, &rt->u.dst);
/* Add an IP checksum. */
ip_send_check(iph);
skb->priority = sk->priority;
return skb->dst->output(skb);
```

코드 365. ip_queue_xmit2()함수 – continued

만약 sk_buff가 가지는 여분의 header를 위한 공간이 하위의 디바이스가 쓰고자 하는 공간보다 작을 시에는 새로이 헤더를 위한 공간을 할당한다(skb_realloc_headroom()). 새로이 할당되었다면 필요한 초기화를 해주고, 보내고자 하는 경로를 살펴서 sk_buff가 가진 데이터의 길이보다 더 작은 패킷만을 한번에 보낼 수 있다면 패킷을 잘라주는 것이 필요하다. 이것을 나누기(segmentation)이라고 하며, 반대로 하는 것을 조립(assembly)/재조립(reassembly)이라고 한다.¹⁴¹

¹⁴⁰ 이것은 커널을 컴파일하게 될 때, 설정하는 옵션이다. 네트워크 부분에서 컴파일시에 netfilter를 고면 아무 효력이 없다. 단순히 routing table에 대한 조작과 받은 패킷에 대한 헤더의 조작이 관련되어 있다.

¹⁴¹ 합쳐서 SAR이라고도 한다.

보내는 경로의 MTU(Maximum Transfer Unit)이 작다면 자르는 나누기 과정으로 가고, 그렇지 않다면 IP checksum을 더한 후, sock구조체의 priority를 sk_buff의 우선순위로 준다음 디바이스의 output을 호출한다. 호출되는 것은 skb->dst->output(skb)이다.

fragment:

```
if (ip_dont_fragment(sk, &rt->u.dst)) {
    /* Reject packet ONLY if TCP might fragment
     * it itself, if were careful enough.
     */
    iph->frag_off |= __constant_htons(IP_DF);
    NETDEBUG(prtnt(KERN_DEBUG "sending pkt_too_big to self\n"));
    icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,
              htonl(rt->u.dst.pmtu));
    kfree_skb(skb);
    return -EMSGSIZE;
}
ip_select_ident(iph, &rt->u.dst);
return ip_fragment(skb, skb->dst->output);
}
```

코드 366. ip_queue_xmit2()함수 – continued

만약 나누어야 할 경우라면, fragment이하의 부분을 실행한다. sk_buff의 헤더 정보를 살펴서(IP header), 보내고자 하는 패킷이 더이상의 fragmentation을 하지 말것을 명시했다면, ICMP(Internet Control Message Protocol) 메시지를 보내고, 소켓 버퍼를 놓아준다. 그렇지 않다면, ip_fragment()함수로 sk_buff와 skb->dst->output()함수를 넘겨주고 호출한다.

skb->dst->output()함수를 보도록하자. IP layer에서 마지막으로 데이터를 보내기 위해서 호출되는 함수이다. 앞에서 ip_queue_xmit()함수에서 나온 ip_route_output()가 skb->dst->output()에 대한 함수 포인터를 연결한다. 먼저 ip_route_output()은 ~/include/net/route.h에 inline함수로 정의되어 있다. 이 함수는 routing table에 대한 key값을 생성한 후, 다시 ip_route_output_key()함수를 호출한다. ip_route_output_key()함수는 ~/net/ipv4/route.c에 정의되어 있으며, 이 함수는 routing table에 대한 key값으로 routing table을 검사해서 해당하는 목적지에 대한 routing정보를 가져온다. 옮바른 routing정보를 가지고 왔다면, 다시 ip_route_output_slow()함수를 routing table과 key값을 넘겨주면서 호출한다. ip_route_output_slow()함수는 경로(route)찾기의 주(main)함수로서, 이곳에서 skb->dst->output()함수의 포인터 값이 ip_output()함수로 결정된다.

ip_output()함수는 아래와 같다. ~/net/ipv4/ip_output.c를 참조하기 바란다.

```
int ip_output(struct sk_buff *skb)
{
#ifdef CONFIG_IP_ROUTE_NAT
    struct rtable *rt = (struct rtable*)skb->dst;
#endif
    IP_INC_STATS(IpOutRequests);
#ifdef CONFIG_IP_ROUTE_NAT
    if (rt->rt_flags&RTCF_NAT)
        ip_do_nat(skb);
#endif
    return ip_finish_output(skb);
}
```

코드 367. ip_output()함수

ip_output()함수는 CONFIG_IP_ROUTE_NAT¹⁴²가 설정되지 않았다면, IP layer의 output요구 횟수를 나타내는 정보를 증가시킨후, ip_finish_output()함수를 호출하는 역할을 한다. 그럼 다시 ip_finish_output()함수를 보도록 하자. 역시 ~/net/ip_output.c를 참조하자.

```
__inline__ int ip_finish_output(struct sk_buff *skb)
{
    struct net_device *dev = skb->dst->dev;
    skb->dev = dev;
    skb->protocol = __constant_htons(ETH_P_IP);
    return NF_HOOK(PF_INET, NF_IP_POST_ROUTING, skb, NULL, dev,
                  ip_finish_output2);
}
```

코드 368. ip_finish_output()함수

이것은 inline으로 정의된 함수로 netfilter를 위해서 정의된 함수이다. ip_finish_output2()함수가 호출됨을 알 수 있다. 계속해서 이것을 추적해보면, ip_finish_output2()함수는 아래와 같다.

```
static inline int ip_finish_output2(struct sk_buff *skb)
{
    struct dst_entry *dst = skb->dst;
    struct hh_cache *hh = dst->hh;

#ifndef CONFIG_NETFILTER_DEBUG
    nf_debug_ip_finish_output2(skb);
#endif /*CONFIG_NETFILTER_DEBUG*/
    if (hh) {
        read_lock_bh(&hh->hh_lock);
        memcpy(skb->data - 16, hh->hh_data, 16);
        read_unlock_bh(&hh->hh_lock);
        skb_push(skb, hh->hh_len);
        return hh->hh_output(skb);
    } else if (dst->neighbour)
        return dst->neighbour->output(skb);
    printk(KERN_DEBUG "khm\n");
    kfree_skb(skb);
    return -EINVAL;
}
```

코드 369. ip_output_finish2()함수

ip_output_finish2()함수는 실제적으로 하드웨어 주소(ethernet address)를 sk_buff구조체에 복사해서 넣고, hh->hh_output()함수를 호출하거나, dst->neighbour->output()함수를 호출한다. 여기서 hh_cache구조체는 하드웨어 헤더 캐시(hardware header cache)를 나타내는 것으로 이 값에 따라서 hh->hh_output()함수가 호출될지 아니면, dst->neighbour필드를 살펴서 dst->neighbour->output()함수를 호출할지를 결정한다. 그리고, hh->hh_output()함수나 dst->neighbour->output()함수 모두는 결국 neighbour구조체의 ops(operation)필드의 dev_queue_xmit()함수를 호출하게 된다.

dev_queue_xmit()함수는 ~/net/core/dev.c에 정의되어 있으며, 네트워크 디바이스 드라이버에서 export한 dev->hard_start_xmit()함수를 호출한다. 여기서 부터는 이전 디바이스 드라이버에 대한 직접적인 호출이 시작된다. 넘겨주는 것은 네트워크 디바이스를 가르키는 net_device구조체의 포인터와 상위에서 넘겨받은 sk_buff구조체에 대한 포인터이다.

¹⁴² NAT(Network Address Translation), IP 주소의 공유를 설정하는 것이다.

이상에서 우린 사용자 프로그램에서 데이터 통신을 하기 위해서 보내는 측면에 관련된 활동들을 살펴보았다. 이제부터 살펴볼 부분은 받는 측면에서 어떤 연산들이 일어나게 되는지를 확인하는 것이다. 디바이스 드라이버에서부터 올라가면서 보도록 하자.

먼저 네트워크 디바이스 드라이버가 패킷을 받게 되면, 인터럽트가 발생한다. 예를 들어서 Intel ethernet express pro 100의 경우, 아래와 같은 함수가 호출된다. ~/drivers/net/eopro100.c를 참고하자.

```
static void speedo_interrupt(int irq, void *dev_instance, struct pt_regs *regs)
{
    ...
    speedo_rx(dev);
    ...
}

static int speedo_rx(struct net_device *dev)
{
    ...
    netif_rx(skb);
    ...
}
```

코드 370. eopro100.c의 RX interrupt 및 처리

즉, speedo_interrupt()함수가 패킷의 도착시에 호출되며, 디바이스의 레지스터 상태를 확인해서 이것이 RX(receive)인 경우에 speedo_rx()함수를 호출한다. 도착한 패킷을 sk_buff의 형태로 만든 다음, 다시 speedo_rx()함수는 netif_rx()함수를 호출해서 네트워크의 패킷이 도착했음을 상위의 layer에 알려준다. netif_rx()함수를 보도록 하자. 여기까지 진행했다면, 이미 sk_buff구조체는 각각의 레이어에 필요한 정보들을 다 가지고 있다.¹⁴³

```
int netif_rx(struct sk_buff *skb)
{
    int this_cpu = smp_processor_id();
    struct softnet_data *queue;
    unsigned long flags;

    if (skb->stamp.tv_sec == 0)
        get_fast_time(&skb->stamp);
    queue = &softnet_data[this_cpu];
    local_irq_save(flags);
    netdev_rx_stat[this_cpu].total++;
```

코드 371. netif_rx() 함수

netif_rx()함수는 네트워크 디바이스 드라이버에서 호출되며, 상위의 프로토콜 레이어(예를 들어 IP나 ARP)에 패킷이 들어왔음을 알리는 일을 한다. 먼저 함수에서 사용하게될 지역변수들에 대한 초기화와 sk_buff의 timestamp값을 설정한다(get_fast_time()). 받은 패킷들을 저장할 각각의 CPU에 배정된 큐를 찾는다(softnet_data[this_cpu]). 이젠 인터럽트가 발생하지 못하도록 만들고(local_irq_save()), 네트워크를 통해서 받은 패킷의 갯수를 증가시킨다.

```
if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
    if (queue->input_pkt_queue.qlen) {
        if (queue->throttle)
            goto drop;
    }
enqueue:
```

¹⁴³ 받은 패킷을 그대로 옮겨주고, sk_buff구조체에 관련된 필드들을 명시한다.

```

        dev_hold(skb->dev);
        __skb_queue_tail(&queue->input_pkt_queue,skb);
        __cpu_raise_softirq(this_cpu, NET_RX_SOFTIRQ);
        local_irq_restore(flags);

#ifndef OFFLINE_SAMPLE
        get_sample_stats(this_cpu);
#endif
        return softnet_data[this_cpu].cng_level;
    }
    if (queue->throttle) {
        queue->throttle = 0;
#endif CONFIG_NET_HW_FLOWCONTROL
        if (atomic_dec_and_test(&netdev_dropping))
            netdev_wakeup();
#endif
    }
    goto enqueue;
}
if (queue->throttle == 0) {
    queue->throttle = 1;
    netdev_rx_stat[this_cpu].throttled++;
#endif CONFIG_NET_HW_FLOWCONTROL
    atomic_inc(&netdev_dropping);
#endif
}
drop:
    netdev_rx_stat[this_cpu].dropped++;
    local_irq_restore(flags);
    kfree_skb(skb);
    return NET_RX_DROP;
}

```

코드 372. netif_rx() 함수 - continued

이전 받은 패킷을 해당하는 CPU의 queue에 집어넣는 일이다. 최대 네트워크 디바이스의 backlog갯수보다 현재 가지고 있는 패킷 큐의 길이가 작다면, 큐에 집어넣도록 한다. 이때 혼잡제어(congestion control)를 하고 있다면, 패킷을 그냥 버린다.
 받은 패킷(sk_buff)을 해당하는 큐에 집어넣는 것은 __skb_queue_tail()이다. 그리고나서 해당하는 CPU에서 네트워크 패킷을 받았으니 처리하도록 소프트웨어 인터럽트를 발생시킨다(__cpu_raise_softirq()). 여기서 간단히 패킷의 queue구조체를 보도록 하자.

```

struct softnet_data
{
    int                      throttle;          /* Throttle값 : congestion control */
    int                      cng_level;         /* Backlog congestion의 level */
    int                      avg_blog;          /* 평균적인 backlog의 길이 */
    struct sk_buff_head *input_pkt_queue;      /* Input packet queue */
    struct net_device *output_queue;           /* Output device를 가르키는 포인터 */
    struct sk_buff *completion_queue;          /* 처리가 완료된 sk_buff의 큐 */
} __attribute__((__aligned__(SMP_CACHE_BYTES)));

```

코드 373. softnet_data type의 정의

__skb_queue_tail()함수는 받은 패킷을 input_pkt_queue에 집어넣게 되며, 만약 대기중인 패킷의 큐의 길이가 backlog의 최대 길이보다 크게 될 경우에는 큐의 throttle값을 설정한다.

이전 소프트웨어 인터럽트를 설정해 놓았기에 적절한 처리는 소프트웨어 인터럽트를 보아야 할 것이다. `__cpu_raise_softirq()`함수는 아래와 같이 정의된다. `~/include/interrupt.h`를 참조하라.

```
static inline void __cpu_raise_softirq(int cpu, int nr)
{
    softirq_active(cpu) |= (1<<nr);
}
```

코드 374. `__cpu_raise_softirq()`함수

위의 함수는 현재 CPU에 해당하는 소프트웨어 IRQ가 발생했다고 표시만하고 복귀한다. 나중에 소프트웨어 인터럽트를 처리하는 `do_softirq()`함수가 호출될 때, `NET_RX_SOFTIRQ`가 처리된다. 이때 `do_softirq()`함수는 `NET_RX_SOFTIRQ`와 관련된 `action`함수를 호출할 것이다. 패킷을 받을 때 호출되는 `action`함수는 `net_rx_action()`함수이다. `~/net/core/dev.c`를 참조하자.

```
static void net_rx_action(struct softirq_action *h)
{
    int this_cpu = smp_processor_id();
    struct softnet_data *queue = &softnet_data[this_cpu];
    unsigned long start_time = jiffies;
    int bugdet = netdev_max_backlog;
```

코드 375. `net_rx_action()`함수

현재의 CPU의 ID와 input queue에 대한 포인터(queue)를 구한다. 그리고나서 처리되는 시점(jiffies) 및 네트워크 디바이스의 최대 backlog의 크기를 가져온다.

```
br_read_lock(BR_NETPROTO_LOCK);
for (;;) {
    struct sk_buff *skb;
    struct net_device *rx_dev;
    local_irq_disable();

    skb = __skb_dequeue(&queue->input_pkt_queue);
    local_irq_enable();
    if (skb == NULL)
        break;
    skb_bond(skb);
    rx_dev = skb->dev;
#endif CONFIG_NET_FASTROUTE
    if (skb->pkt_type == PACKET_FASTROUTE) {
        netdev_rx_stat[this_cpu].fastroute_deferred_out++;
        dev_queue_xmit(skb);
        dev_put(rx_dev);
        continue;
    }
#endif
```

코드 376. `net_rx_action()`함수 – continued

일단 처리하기 위해서 lock을 설정한다(`br_read_lock()`). 그리고나서 무한 loop를 돌면서 `input_pkt_queue`로부터 하나씩 데이터 패킷을 가져와서 처리하게 된다. 패킷을 얻는 것은 `__skb_dequeue()`함수이다. 만약 더 이상 처리하고자 하는 패킷이 없다면, 당연히 loop의 수행을 마친다. 또한 `sk_buff`구조체로부터 패킷이 올라온 디바이스를 얻는다.

```
    skb->h.raw = skb->nh.raw = skb->data;
{
```

```

struct packet_type *ptype, *pt_prev;
unsigned short type = skb->protocol;

pt_prev = NULL;
for (ptype = ptype_all; ptype; ptype = ptype->next) {
    if (!ptype->dev || ptype->dev == skb->dev) {
        if (pt_prev) {
            if (!pt_prev->data) {
                deliver_to_old_ones(pt_prev, skb, 0);
            } else {
                atomic_inc(&skb->users);
                pt_prev->func(skb,
                                skb->dev,
                                pt_prev);
            }
        }
        pt_prev = ptype;
    }
}

```

코드 377. net_rx_action()함수 – continued

소켓 버퍼의 헤더를 카리키는 포인터를 sk_buff의 데이터 포인터로 초기화해 주고, 소켓 버퍼에서 프로토콜 정보를 가져온다(skb->protocol). 이 정보가 받은 패킷의 type을 결정해준다. 패킷의 타입에 따라서 핸들러를 찾기 위해서, packet_type이라는 구조체가 정의되어 있다. ~/include/linux/netdevice.h에 정의되어 있다.

```

struct packet_type
{
    unsigned short          type;      /* packet의 type */
    struct net_device       *dev;      /* 네트워크 디바이스 구조체 포인터 */
    int                   (*func)(struct sk_buff *, struct net_device *,
                                struct packet_type *); /* 패킷 타입에 따른 핸들러 함수 */
    void                  *data;     /* 패킷타입의 고유한 데이터들에 대한 포인터 */
    struct packet_type   *next;     /* 패킷타입의 리스트 */
};

```

코드 378. packet_type구조체의 정의

전체 packet type에 대한 리스트는 ptype_all로 유지한다. 만약 등록된 packet type에 대해서 디바이스가 NULL이거나, packet type의 디바이스 필드가 sk_buff의 디바이스 필드와 같다면, 다시 pt_prev가 NULL이 아닌지를 확인하고, packet type의 data필드를 살펴본다. NULL이라면, deliver_to_old_ones()가 호출되고, 그렇지 않다면 pt_prev->func()함수가 호출된다. 그리고, loop를 돌면서 모든 패킷 타입을 계속 살펴본다.

```

#ifndef CONFIG_NET_DIVERT
    if (skb->dev->divert && skb->dev->divert->divert)
        handle_diverter(skb);
#endif /* CONFIG_NET_DIVERT */
#ifndef CONFIG_BRIDGE
#define CONFIG_BRIDGE_MODULE
#endif
if (skb->dev->br_port != NULL &&
    br_handle_frame_hook != NULL) {
    handle_bridge(skb, pt_prev);
    dev_put(rx_dev);
    continue;
}
for (ptype=ptype_base[ntohs(type)&15];ptype;ptype=ptype->next) {

```

```

        if (ptype->type == type &&
            (!ptype->dev || ptype->dev == skb->dev)) {
            if (pt_prev) {
                if (!pt_prev->data)
                    deliver_to_old_ones(pt_prev, skb, 0);
                else {
                    atomic_inc(&skb->users);
                    pt_prev->func(skb,
                                   skb->dev,
                                   pt_prev);
                }
            }
            pt_prev = ptype;
        }
    }
    if (pt_prev) {
        if (!pt_prev->data)
            deliver_to_old_ones(pt_prev, skb, 1);
        else
            pt_prev->func(skb, skb->dev, pt_prev);
    } else
        kfree_skb(skb);
}

```

코드 379. net_rx_action() 함수 – continued

CONFIG_NET_DIVERT와 CONFIG_BRIDGE가 설정되었다면 해당하는 처리를 해주게되며, 다시 packet type에 따라 차례로 처리하기 위한 for loop를 실행하게 된다. 위에서 for loop를 실행하며 해준 것은 ethernet packet type이 ETH_P_ALL로 설정된 경우를 위한 것이고, 여기서 보이는 것은 그외의 packet type에 대한 처리이다.

```

dev_put(rx_dev);
if (bugdet-- < 0 || jiffies - start_time > 1)
    goto softnet_break;
#endif CONFIG_NET_HW_FLOWCONTROL
    if (queue->throttle && queue->input_pkt_queue.qlen < no_cong_thresh ) {
        if (atomic_dec_and_test(&netdev_dropping)) {
            queue->throttle = 0;
            netdev_wakeup();
            goto softnet_break;
        }
    }
#endif
}
br_read_unlock(BR_NETPROTO_LOCK);
local_irq_disable();
if (queue->throttle) {
    queue->throttle = 0;
#endif CONFIG_NET_HW_FLOWCONTROL
    if (atomic_dec_and_test(&netdev_dropping))
        netdev_wakeup();
}
local_irq_enable();
NET_PROFILE_LEAVE(softnet_process);
return;
softnet_break:

```

```

    br_read_unlock(BR_NETPROTO_LOCK);
    local_irq_disable();
    netdev_rx_stat[this_cpu].time_squeeze++;
    __cpu_raise_softirq(this_cpu, NET_RX_SOFTIRQ);
    local_irq_enable();
    NET_PROFILE_LEAVE(softnet_process);
    return;
}

```

코드 380. net_rx_action()함수 – continued

이전 패킷에 대한 처리보다는 이부분에서는 흐름 제어(flow control)을 담당한다. 너무 많은 패킷이 날아올 경우에는 네트워크의 혼잡이 커지므로 이것을 방지할 목적이다.

이상에서 대략적으로 네트워크와 관련된 bottom half에 대해서 살펴보았다. 결과적으로 이야기 한다면 결국 등록된 패킷타입에 해당하는 핸들러(handler)함수가 소켓버퍼에 대해서 호출된다. 우리가 지금 논의하고 있는 것은 TCP/IP로 한정되어 있기에, IP layer에 있는 패킷타입 핸들러가 호출될 것이다. IP 패킷 핸들러의 등록은 IP모듈을 초기화하면서 된다. 아래와 같다. ~/net/ipv4/ip_output.c를 참조하자.

```

/*
 *      IP protocol layer initialiser
 */
static struct packet_type ip_packet_type =
{
    __constant_htons(ETH_P_IP),
    NULL, /* All devices */
    ip_rcv,
    (void*)1,
    NULL,
};

/*
 *      IP registers the packet type and then calls the subprotocol initialisers
 */
void __init ip_init(void)
{
    dev_add_pack(&ip_packet_type);

    ip_rt_init();
    inet_initpeers();

#ifdef CONFIG_IP_MULTICAST
    proc_net_create("igmp", 0, ip_mc_procinfo);
#endif
}

```

코드 381. IP 프로토콜 layer의 초기화

즉, ~/net/core/dev.c에 정의된 dev_add_pack()함수를 호출하면서 등록한다. packet type으로는 ETH_P_IP가 핸들러 함수는 ip_rcv를 준다. 따라서, net_rx_action()에서는 ip_rcv()함수가 호출될 것이다.

ip_rcv()함수를 보도록 하자. 정의는 ~/net/ipv4/ip_input.c에 나와 있다.

```

int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)
{
    struct iphdr *iph = skb->nh.iph;

    if (skb->pkt_type == PACKET_OTHERHOST)
        goto drop;
}

```

```

IP_INC_STATS_BH(IpInReceives);
if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL)
    goto out;
if (skb->len < sizeof(struct iphdr) || skb->len < (iph->ihl<<2))
    goto inhdr_error;
if (iph->ihl < 5 || iph->version != 4 || ip_fast_csum((u8 *)iph, iph->ihl) != 0)
    goto inhdr_error;
{
    __u32 len = ntohs(iph->tot_len);
    if (skb->len < len || len < (iph->ihl<<2))
        goto inhdr_error;
    __skb_trim(skb, len);
}
return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
               ip_rcv_finish);

inhdr_error:
IP_INC_STATS_BH(IpInHdrErrors);
drop:
kfree_skb(skb);
out:
return NET_RX_DROP;
}

```

코드 382. ip_rcv()함수의 정의

이 함수에서 하는 역할은 받은 패킷의 옮바른지를 확인하는 것이다. 조건에는 먼저 최소한 IP header를 가지고 있는지와 IP version이 4인지, checksum은 옮바른지, 그리고 실제적인 길이를 가지는 데이터인지를 확인한다. 만약 받은 packet이 다른 호스트로 가는 패킷이라면 drop한다. 이 경우에는 hardware가 promiscuous 모드로 동작하는 경우가 될 것이다. 이때는 네트워크를 돌아다니는 모든 패킷이 다 들어온다. 그리고나서 통계적인 정보를 갱신(update)하게 되며, 소켓버퍼의 헤드를 새로 할당받아서 복사한다(skb_share_check()). 이때도 에러가 있다면, 해당하는 패킷을 처리하지 않고 복귀한다. 이전 패킷의 길이와 IP의 버전 번호, checksum을 처리하는 부분이다. 에러가 있다면, 에러통계를 내고, 패킷을 버리고 복귀한다. 마지막으로 다시 netfilter의 처리를 하게되는데, 역시 무시하기로 하고, ip_rcv_finish()함수가 바로 호출된다고 보자.

```

static inline int ip_rcv_finish(struct sk_buff *skb)
{
    struct net_device *dev = skb->dev;
    struct iphdr *iph = skb->nh.iph;

    if (skb->dst == NULL) {
        if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev))
            goto drop;
    }
#endif CONFIG_NET_CLS_ROUTE
    if (skb->dst->tclassid) {
        struct ip_rt_acct *st = ip_rt_acct + 256*smp_processor_id();
        u32 idx = skb->dst->tclassid;
        st[idx&0xFF].o_packets++;
        st[idx&0xFF].o_bytes+=skb->len;
        st[(idx>>16)&0xFF].i_packets++;
        st[(idx>>16)&0xFF].i_bytes+=skb->len;
    }
#endif

```

코드 383. ip_rcv_finish()함수

ip_rcv_finish()함수가 하는 일은 routing table을 확인하고, 그곳에 등록된 input함수를 다시 호출하는 일이다. 먼저 skb->dst가 NULL값을 가지는지 확인한다. NULL값을 가진다면, ip_route_input()함수를 호출해서 routing table을 갱신하고 multicasting에 대한 처리를 한다. 또한 CONFIG_NET_CLS_ROUTE가 설정되었다면, 관련된 accounting정보를 업데이트한다.

```

if (iph->ihl > 5) {
    struct ip_options *opt;
    skb = skb_cow(skb, skb_headroom(skb));
    if (skb == NULL)
        return NET_RX_DROP;
    iph = skb->nh.iph;
    skb->ip_summed = 0;
    if (ip_options_compile(NULL, skb))
        goto inhdr_error;
    opt = &(IPCB(skb)->opt);
    if (opt->srr) {
        struct in_device *in_dev = in_dev_get(dev);
        if (in_dev) {
            if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
                if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit())
                    printk(KERN_INFO "source route option %u.%u.%u.%u - 
> %u.%u.%u.%u\n",
NIPQUAD(iph->saddr), NIPQUAD(iph->daddr));
                in_dev_put(in_dev);
                goto drop;
            }
            in_dev_put(in_dev);
        }
        if (ip_options_rcv_srr(skb))
            goto drop;
    }
}
return skb->dst->input(skb);
inhdr_error:
IP_INC_STATS_BH(IpInHdrErrors);
drop:
kfree_skb(skb);
return NET_RX_DROP;
}

```

코드 384. ip_rcv_finish()함수의 정의

IP의 header길이가 5이상이라면, sk_buff에 대해서 IP head를 위한 적절한 공간이 있는지 확인하고(skb_cow()), 없다면 NET_RX_DROP를 돌려준다. 또한 IP header의 정보를 구하고, IP checksum을 sk_buff구조체에 0으로 둔다. 다음으로 하는 일은 IP의 option을 확인하고 새로운 IP header의 정보를 그 option에 맞게 구성하는 일이다(ip_options_compile()). 만약 IP의 option이 Source routing이라면, 해당하는 처리를 해주고, 마지막으로 skb->dst->input()함수를 호출한다. 예러가 발생한다면 해당하는 처리를 해준다음 NET_RX_DROP을 돌려주고 복귀한다.

skb->dst->input() 함수의 포인터는 Local일 경우에 ip_local_deliver()로 정해지며(ip_route_input()함수의 내부에서), 아래와 같다. ~/net/ipv4/ip_input.c를 참조하자.

```

int ip_local_deliver(struct sk_buff *skb)
{
    struct iphdr *iph = skb->nh.iph;
    if (iph->frag_off & htons(IP_MF|IP_OFFSET)) {

```

```

        skb = ip_defrag(skb);
        if (!skb)
            return 0;
    }
    return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,
                  ip_local_deliver_finish);
}

```

코드 385. ip_local_deliver()함수

이 함수의 역할은 fragment된 IP 패킷의 재조합(reassembly)하는 일이며(ip_defrag()), 끝나게되면 ip_local_deliver_finish() 함수를 호출한다. 이 함수 역시 ~/net/ipv4/ip_input.c에 나와 있다.

```

static inline int ip_local_deliver_finish(struct sk_buff *skb)
{
    struct iphdr *iph = skb->nh.iph;

#ifndef CONFIG_NETFILTER_DEBUG
    nf_debug_ip_local_deliver(skb);
#endif /*CONFIG_NETFILTER_DEBUG*/
    /* Point into the IP datagram, just past the header. */
    skb->h.raw = skb->nh.raw + iph->ihl*4;
}

```

코드 386. ip_local_deliver_finish()함수

넘겨받는 sk_buff로부터 IP header에 대한 포인터를 구하자. 이 포인터를 기준으로 해서, 해당하는 상위의 protocol을 찾을 것이다. 즉, IP의 datagram을 찾을 것이다.¹⁴⁴

```

{
    /* Note: See raw.c and net/raw.h, RAWV4_HTABLE_SIZE==MAX_INET_PROTOS */
    int hash = iph->protocol & (MAX_INET_PROTOS - 1);
    struct sock *raw_sk = raw_v4_htable[hash];
    struct inet_protocol *ipprot;
    int flag;

    if(raw_sk != NULL)
        raw_sk = raw_v4_input(skb, iph, hash);
    ipprot = (struct inet_protocol *) inet_protos[hash];
    flag = 0;
    if(ipprot != NULL) {
        if(raw_sk == NULL &&
           ipprot->next == NULL &&
           ipprot->protocol == iph->protocol) {
            int ret;
            /* Fast path... */
            ret = ipprot->handler(skb, (ntohs(iph->tot_len) -
                                         (iph->ihl * 4)));
            return ret;
        } else {
            flag = ip_run_ipprot(skb, iph, ipprot, (raw_sk != NULL));
        }
    }
}

```

코드 387. ip_local_deliver_finish()함수 – continued

¹⁴⁴ 즉, IP의 보내고자 하는 데이터가 된다. 이것은 TCP의 경우 자신의 헤더 첫부분이 될 것이다.

RAW socket 옵션이 있는지를 확인하고 해당하는 처리를 해준다(raw_v4_input()). 해쉬 인덱스를 계산해서 inet_protocol 구조체의 배열로부터 해당하는 프로토콜에 대한 값을 가져온다음 이것을 ipprot에 설정한다. 만약 ipprot가 NULL이 아니라면 해당하는 프로토콜 번호와 일치하는 핸들러를 찾아서 핸들러를 호출하게 된다. ip_run_ipprot() 함수 역시 해당하는 프로토콜을 찾아서 핸들러를 호출하는 역할을 한다.

```

if(raw_sk != NULL) {           /* Shift to last raw user */
    raw_rcv(raw_sk, skb);
    sock_put(raw_sk);
} else if (!flag) {           /* Free and report errors */
    icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PROT_UNREACH, 0);
    kfree_skb(skb);
}
}
return 0;
}

```

코드 388. ip_local_deliver_finish() 함수 – continued

역시 RAW socket에 대한 처리를 해주고, 만약 RAW socket도 아니고 flag(fragmentation)도 설정되지 않았다면 ICMP message를 만들어서 보내주고, 소켓 버퍼를 놀아준다(release).

여기서 잠시 inet_protocol 구조체를 보도록 하자. 이것은 IP에서 지원하는 상위의 프로토콜에 대한 정보를 가지고 초기화 된다. 다음과 같이 정의된다. ~/include/net/protocol.h를 참조하자.

```

struct inet_protocol
{
    int                      (*handler)(struct sk_buff *skb, unsigned short len);
    void                     (*err_handler)(struct sk_buff *skb, unsigned char *dp, int len);
    struct inet_protocol *next;
    unsigned char            protocol;
    unsigned char            copy:1;
    void                     *data;
    const char               *name;
};

```

코드 389. inet_protocol 구조체의 정의

inet_protocol 구조체는 상위의 프로토콜 핸들러 및 에러 핸들러, 프로토콜 번호, copy flag, 사용되는 데이터, 프로토콜의 이름으로 구성된다. TCP와 UDP에 대한 inet_protocol 구조체는 아래와 같이 정의된다.

```

static struct inet_protocol tcp_protocol =
{
    tcp_v4_rcv,                  /* TCP handler */
    tcp_v4_err,                  /* TCP error control */
    IPPROTO_PREVIOUS,
    IPPROTO_TCP,                /* protocol ID */
    0,                           /* copy */
    NULL,                        /* data */
    "TCP"                         /* name */
};

...
static struct inet_protocol udp_protocol =
{
    udp_rcv,                    /* UDP handler */
    udp_err,                    /* UDP error control */
    IPPROTO_PREVIOUS,
    IPPROTO_UDP,                /* protocol ID */
};

```

```

0,          /* copy */  

NULL,       /* data */  

"UDP"       /* name */  

};
```

코드 390. TCP와 UDP에 대한 inet_protocol구조체의 정의

이렇게 정의된 inet_protocol구조체들은 각각이 포인터로 서로 연결되어 있다. IP에서는 해당하는 프로토콜의 핸들러를 호출하는 역할만하고, 나머지는 해당 프로토콜이 담당한다. 우리가 현재 보고 있는 것은 TCP에 대한 것이므로, 이전 tcp_v4_rcv()핸들러 함수가 될 것이다. 이전 다시 TCP까지 올라오게 되었다.

5.8. TCP/UDP Layer에서 받기

TCP/UDP에서는 IP layer로부터 sk_buff형태로 버퍼를 넘겨받아서 프로토콜에 관련된 연산을 처리한 후, 다시 상위(INET layer)의 layer로 받은 데이터를 전송할 책임을 진다. tcp_v4_rcv()함수를 보자. 정의는 ~/net/ipv4/tcp_ipv4.c에 있다.

```

int tcp_v4_rcv(struct sk_buff *skb, unsigned short len)
{
    struct tcphdr *th;
    struct sock *sk;
    int ret;

    if (skb->pkt_type!=PACKET_HOST)
        goto discard_it;
    th = skb->h.th;
    /* Pull up the IP header. */
    __skb_pull(skb, skb->h.raw - skb->data);
    /* Count it even if it's bad */
    TCP_INC_STATS_BH(TcpInSegs);
```

코드 391. tcp_v4_rcv()함수

넘겨받은 패킷이 자신의 호스트(host)로 보내는지 확인하고, TCP header의 위치를 구한다. 이곳에서는 IP header에 대한 정보가 필요없으므로 버리게 된다(__skb_pull()). 그리고나서, account정보를 업데이트 한다.

```

if (th->doff < sizeof(struct tcphdr)/4 ||
    (skb->ip_summed != CHECKSUM_UNNECESSARY &&
     tcp_v4_checksum_init(skb) < 0))
    goto bad_packet;
TCP_SKB_CB(skb)->seq = ntohl(th->seq);
TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->fin +
    len - th->doff*4);
TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);
TCP_SKB_CB(skb)->when = 0;
TCP_SKB_CB(skb)->flags = skb->nh.iph->tos;
TCP_SKB_CB(skb)->sacked = 0;
skb->used = 0;
sk = __tcp_v4_lookup(skb->nh.iph->saddr, th->source,
                     skb->nh.iph->daddr, ntohs(th->dest), tcp_v4_iif(skb));
if (!sk)
    goto no_tcp_socket;
process:
    if(!ipsec_sk_policy(sk,skb))
        goto discard_and_reuse;
```

```

if (sk->state == TCP_TIME_WAIT)
    goto do_time_wait;
skb->dev = NULL;
bh_lock_sock(sk);
ret = 0;
if (!sk->lock.users) {
    if (!tcp_prequeue(sk, skb))
        ret = tcp_v4_do_rcv(sk, skb);
} else
    sk_add_backlog(sk, skb);
bh_unlock_sock(sk);
sock_put(sk);
return ret;

```

코드 392. tcp_v4_rcv() 함수 – continued

패킷이 옮겨온지를 확인하고, sk_buff구조체의 TCP를 위한 제어 블록(control block)을 sk_buff에 있는 내용으로 채운다. 그리고나서, 소스(source)와 목적지(destination)의 주소를 가지고, 해당하는 INET sock구조체가 연결(established) 상태에 있거나 혹은 연결대기(listen) 상태에 있는 것을 찾는다. 이미 연결되어 있어서고 가정하였기에 INET sock구조체는 이미 있을 것이다.

찾았다면, 이것을 가지고, 소켓의 보안(security)과 상태를 검사해서 적절한 행동을 취한다. INET sock구조체에 대한 lock을 걸고, 사용자가 있는지 확인한다. 없다면 tcp_prequeue()를 호출해서, 현재 사용자 프로세스로 바로 copy를 할 수 있는지를 확인한다. 그렇지 않다면 tcp_v4_do_rcv()함수를 호출하게되며, lock하고 있는 사용자가 있다면 backlog에 sk_buff를 넣어준다(sk_add_backlog()).

이전 INET socket에 대한 lock을 해제하고, INET socket에 대한 reference count를 감소시킨 후(sock_put()) 복귀한다.

```

no_tcp_socket:
    if (len < (th->doff<<2) || tcp_checksum_complete(skb)) {
bad_packet:
    TCP_INC_STATS_BH(TcpInErrs);
} else {
    tcp_v4_send_reset(skb);
}
discard_it:
/* Discard frame. */
kfree_skb(skb);
return 0;
discard_and_relse:
    sock_put(sk);
    goto discard_it;
do_time_wait:
    if (len < (th->doff<<2) || tcp_checksum_complete(skb)) {
        TCP_INC_STATS_BH(TcpInErrs);
        goto discard_and_relse;
    }
    switch(tcp_timewait_state_process((struct tcp_tw_bucket *)sk,
                                      skb, th, skb->len)) {
case TCP_TW_SYN:
{
    struct sock *sk2;
    sk2 = tcp_v4_lookup_listener(skb->nh.iph->daddr, ntohs(th->dest), tcp_v4_iif(skb));
    if (sk2 != NULL) {
        tcp_tw_deschedule((struct tcp_tw_bucket *)sk);
        tcp_tw_kill((struct tcp_tw_bucket *)sk);
        tcp_tw_put((struct tcp_tw_bucket *)sk);
        sk = sk2;
    }
}
}

```

```

        goto process;
    }
    /* Fall through to ACK */
}
case TCP_TW_ACK:
    tcp_v4_timewait_ack(sk, skb);
    break;
case TCP_TW_RST:
    goto no_tcp_socket;
case TCP_TW_SUCCESS:;
}
goto discard_it;
}

```

코드 393. tcp_v4_rcv()함수 – continued

에러를 처리하고, 할당된 sk_buff구조체를 해제하는 부분이다. do_time_wait로 제어가 넘어가게 되면, SYN 및 ACK와 RST에 대한 처리를 각각해주게 되며, 마지막으로 다시 sk_buff구조체를 해제한다.

tcp_v4_do_rcv()함수를 보도록하자. ~/net/ipv4/tcp_ipv4.c에 정의되어 있다. 이 함수는 INET socket의 상태에 따라서 달리 처리한다. 즉, 연결(established)되어 있는지 아니면 연결대기(listen)상태인지를 확인하게 된다.

```

int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
#ifdef CONFIG_FILTER
    struct sk_filter *filter = sk->filter;
    if (filter && sk_filter(skb, filter))
        goto discard;
#endif /* CONFIG_FILTER */
    IP_INC_STATS_BH(IpInDelivers);
    if (sk->state == TCP_ESTABLISHED) { /* Fast path */
        TCP_CHECK_TIMER(sk);
        if (tcp_rcv_established(sk, skb, skb->h.th, skb->len))
            goto reset;
        TCP_CHECK_TIMER(sk);
        return 0;
    }
    if (skb->len < (skb->h.th->doff<<2) || tcp_checksum_complete(skb))
        goto csum_err;
    if (sk->state == TCP_LISTEN) {
        struct sock *nsk = tcp_v4_hnd_req(sk, skb);
        if (!nsk)
            goto discard;
        if (nsk != sk) {
            if (tcp_child_process(sk, nsk, skb))
                goto reset;
            return 0;
        }
    }
    TCP_CHECK_TIMER(sk);
    if (tcp_rcv_state_process(sk, skb, skb->h.th, skb->len))
        goto reset;
    TCP_CHECK_TIMER(sk);
    return 0;
reset:
    tcp_v4_send_reset(skb);
}

```

코드 394. tcp_v4_do_rcv()함수

Netfilter 옵션이 정해져있다면 패킷에 대한 filtering을 하며, IP의 account정보를 업데이트한다(IpInDelivers). 만약 INET socket의 상태가 TCP_ESTABLISHED라면, INET socket의 TCP와 관련된 timer를 갱신하고, tcp_rcv_established()함수를 호출한 후 복귀한다. 패킷의 길이나 TCP checksum에 문제가 있다면 예러 처리로 넘어가게 되며, 만약 INET socket이 TCP_LISTEN상태에 있었다면, tcp_v4_hnd_req()함수를 호출해서 처리를 넘겨준다. tcp_v4_hnd_req()함수는 클라이언트(client)로부터 connect요구와 같은 것을 처리할 것이다. 이 경우 listen상태에 있는 INET socket을 찾아서 해당하는 INET socket구조체를 구할 것이다. 만약 구한 값이 없다면, sk_buff를 버리고 복귀하며, 구한 INET socket이 현재 처리 요구중인 INET socket과 다른 경우에는 tcp_child_process()를 호출한다. tcp_child_process()함수는 ESTABLISH나 TIME_WAIT상태가 아닌 child INET socket에 대한 처리를 하거나, 혹은 child INET socket의 backlog에 sk_buff를 더하는 일을 한다(tcp_rcv_state_process()를 child INET socket에 대해서 호출함). 모든 상황에 대한 처리를 다했을 경우에 대해, 지역(local) 호스트에 해당하는 INET socket이 없다면 상대방 호스트로 RESET메시지를 보낸다(tcp_v4_send_reset()).

```

discard:
    kfree_skb(skb);
    /* Be careful here. If this function gets more complicated and
     * gcc suffers from register pressure on the x86, sk (in %ebx)
     * might be destroyed here. This current version compiles correctly,
     * but you have been warned.
    */
    return 0;
csum_err:
    TCP_INC_STATS_BH(TcpInErrs);
    goto discard;
}

```

코드 395. tcp_v4_do_rcv()함수 – continued

sk_buff를 버리고 복귀하는 부분이다. Checksum error에 대해서 해당하는 예러 account정보를 업데이트 한다.

현재 우리가 보고 있는 경우는 소켓이 이미 연결 상태로 되어 있는 상황이므로, 진행을 위해서 tcp_rcv_established()함수가 선택된다. 이 함수의 정의는 ~/net/ipv4/tcp_input.c에 나와있다. 이 함수가 하는 역할은 Acknowledgement에 대한 처리와 상위의 프로토콜 layer에 패킷이 도착했음을 알리는 일을 한다. 이를 위해서 사용하는 함수로는 tcp_data()함수와 INET socket의 data_ready() 메소드를 호출하는 방법을 사용한다. 어떤 것을 사용할지는 패킷을 받아서 사용하게될 프로세스가 current process인지 아닌지에 기인한다.

만약 현재의 프로세스라면 INET socket의 data_ready()함수가 호출될 것이며, 그렇지 않다면, tcp_data()함수가 호출될 것이다. 현재 진행중인 프로세스가 아니라고 하는 것이 더 일반적인 경우가 되기에 tcp_data()함수를 주적하자. 정의는 ~/net/ipv4/tcp_input.c에 나와 있다.

```

static void tcp_data(struct sk_buff *skb, struct sock *sk, unsigned int len)
{
    struct tcphdr *th;
    struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);

    th = skb->h.th;
    skb_pull(skb, th->doff*4);
    skb_trim(skb, len - (th->doff*4));
    if (skb->len == 0 && !th->fin)
        goto drop;
    TCP_ECN_accept_cwr(tp, skb);
    if (atomic_read(&sk->rmem_alloc) > sk->rcvbuf ||
        !tcp_rmem_schedule(sk, skb)) {
        if (tcp_prune_queue(sk) < 0 || !tcp_rmem_schedule(sk, skb))

```

```

        goto drop;
    }
    tcp_data_queue(sk, skb);
#endif TCP_DEBUG
    if (before(tp->rcv_nxt, tp->copied_seq)) {
        printk(KERN_DEBUG "*** tcp.c:tcp_data bug acked < copied\n");
        tp->rcv_nxt = tp->copied_seq;
    }
#endif
    return;
drop:
    __kfree_skb(skb);
}

```

코드 396. **tcp_data()**함수

이 함수의 역할은 패킷 데이터에 대한 처리이다. 먼저 sk_buff에 들어있는 TCP header의 정보와 INET socket에 들어있는 TCP 옵션에 대한 정보를 구해온 후, 더이상 필요없는 헤더 정보를 빼어내고, 길이가 정확한지를 확인한다. 만약 receive를 위한 버퍼의 크기가 적당하지 못하다면, 받은 sk_buff를 버린다(drop). 적절하다고 생각되면, tcp_data_queue()함수를 호출해서 받은 sk_buff의 소유자를 INET socket으로 바꾸고, INET socket의 receive_queue의 끝에다가 sk_buff를 집어넣어주게되며, 다시 INET socket의 data_ready() 메쏘드를 호출한다.

5.9. INET Socket Layer에서 받기

앞에서 이미 INET socket layer의 data_ready() 메쏘드를 호출한다고 이야기 했다. 여기서부터는 이제 INET socket layer에서 제공하는 함수가 된다. INET socket layer의 data_ready()메쏘드는 사용되는 주소 패밀리(Address Family)마다 다른 형식을 취하고 있는데, 이곳에서는 INET에 대한 것으로 한정하기에 ~/net/core/sock.c파일의 sock_init_data()함수내에서 아래와 같이 정의된다.

sk->data_ready = sock_def_readable;

sock_def_readable()함수를 보면, 아래와 같다. 정의는 ~/net/core/sock.c에 나와 있다.

```

void sock_def_readable(struct sock *sk, int len)
{
    read_lock(&sk->callback_lock);
    if (sk->sleep && waitqueue_active(sk->sleep))
        wake_up_interruptible(sk->sleep);
    sk_wake_async(sk,1,POLL_IN);
    read_unlock(&sk->callback_lock);
}

```

코드 397. **sock_def_readable()**함수

이 함수가 하는 역할은 INET socket의 sleep queue에서 자고 있는 프로세스들을 깨우는 역할을 한다. 즉, 반기를 원하는 프로세스가 있다면, 그것들을 전부 깨워줄 것이다. 만약 프로세스가 read를 원한다면, BSD socket에 대해서 read()함수를 호출 했을 것이며, 이것은 다시 sys_read() 시스템 콜, sock_read()함수, inet_rcvmsg()함수, tcp_rcvmsg()함수의 순으로 호출하게 될 것이다. 버퍼가 비어있고, 프로세스가 블록킹 모드를 사용하고 있다면, INET socket의 sleep queue에서 잠들어 있게 된다. 따라서, 이젠 원하는 데이터가 도착했으므로 다시 깨어나게 될 것이며, 이후는 INET socket의 receive_queue에 들어있는 sk_buff데이터들을 처리하게 될 것이다.

sys_read()함수는 ~/fs/read_write.c를 참조하면 된다. 아래와 같다.

```
asmlinkage ssize_t sys_read(unsigned int fd, char * buf, size_t count)
```

```

{
    ssize_t ret;
    struct file * file;

    ret = -EBADF;
    file = fget(fd);
    if (file) {
        if (file->f_mode & FMODE_READ) {
            ret = locks_verify_area(FLOCK_VERIFY_READ, file->f_dentry->d_inode,
                                   file, file->f_pos, count);
            if (!ret) {
                ssize_t (*read)(struct file *, char *, size_t, loff_t *);
                ret = -EINVAL;
                if (file->f_op && (read = file->f_op->read) != NULL)
                    ret = read(file, buf, count, &file->f_pos);
            }
        }
        if (ret > 0)
            inode_dir_notify(file->f_dentry->d_parent->d_inode,
                             DN_ACCESS);
        fput(file);
    }
    return ret;
}

```

코드 398. sys_read()함수

sys_read()함수는 파일 디스크립터와 사용자 데이터 버퍼, 그리고, 버퍼의 크기를 나타내는 값을 넘겨받는다. 파일 디스크립터로 부터 해당하는 파일 object를 찾고, 이를 이용해서 파일 연산자의 read()함수를 부른다. 해당하는 파일 object를 찾을때 BSD socket에 해당하는 파일 연산자를 찾을 것이다. 따라서, read()함수는 sock_read()에 해당한다.

5.10. BSD socket layer에서 받기

sock_read()함수가 BSD socket layer에 해당하는 파일 object의 파일 연산자로 주어진다. 아래와 같다. 정의는 ~/net/socket.c에서 찾을 수 있다.

```

static ssize_t sock_read(struct file *file, char *ubuf,
                        size_t size, loff_t *ppos)
{
    struct socket *sock;
    struct iovec iov;
    struct msghdr msg;
    int flags;

    if (ppos != &file->f_pos)
        return -ESPIPE;
    if (size==0)           /* Match SYS5 behaviour */
        return 0;
    sock = socki_lookup(file->f_dentry->d_inode);
    msg.msg_name=NULL;
    msg.msg_nameLEN=0;
    msg.msg iov=&iov;
    msg.msg iovLEN=1;
    msg.msg control=NULL;
    msg.msg controlLEN=0;
    iov.iov_base=ubuf;

```

```

iov.iov_len=size;
flags = !(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
return sock_recvmsg(sock, &msg, size, flags);
}

```

코드 399. sock_read()함수

sock_read()함수는 파일 object의 포인터와 사용자 버퍼의 포인터, 버퍼의 크기와 현재의 연산 위치를 넘겨받는다. 이곳에서는 메시지 구조체를 생성하고, file object의 디렉토리 엔트리로 필드의 d_inode로부터 해당하는 BSD socket을 가져온다. 이를 이용해서 다시 BSD socket layer의 sock_recvmsg()함수를 호출하게 된다.

```

int sock_recvmsg(struct socket *sock, struct msghdr *msg, int size, int flags)
{
    struct scm_cookie scm;

    memset(&scm, 0, sizeof(scm));
    size = sock->ops->recvmsg(sock, msg, size, flags, &scm);
    if (size >= 0)
        scm_recv(sock, msg, &scm, flags);
    return size;
}

```

sock_recvmsg()함수는 다시 BSD socket의 프로토콜 연산 구조체인 recvmsg()함수를 호출하고, 넘겨받은 값이 0이상이 되면, scm_recv()를 호출해서 socket 제어를 처리한다. 복귀값으로는 받은 데이터의 크기가 될 것이다. 여기서는 INET 프로토콜에 대해서 논의하고 있으므로, 이전에 INET socket layer의 연산구조체를 말할때 나온 inet_recvmsg()함수가 호출될 것이다. ~/net/ipv4/af_inet.c를 보도록 하자.

```

int inet_recvmsg(struct socket *sock, struct msghdr *msg, int size,
                 int flags, struct scm_cookie *scm)
{
    struct sock *sk = sock->sk;
    int addr_len = 0;
    int err;

    err = sk->prot->recvmsg(sk, msg, size, flags&MSG_DONTWAIT,
                           flags&~MSG_DONTWAIT, &addr_len);
    if (err >= 0)
        msg->msg_namelen = addr_len;
    return err;
}

```

코드 400. inet_recvmsg()함수

inet_recvmsg()함수는 다시 BSD socket으로부터 INET socket구조체를 찾아내고, INET socket구조체로부터 해당하는 프로토콜의 recvmsg()함수를 호출한다. 넘겨주는 것은 recvmsg()를 호출한 결과값이다. INET socket의 prot필드는 하위의 프로토콜에서 제공하는 연산 구조체이다. TCP에 한정시킨다면, tcp_recvmsg()함수가 호출될 것이다. 정의는 ~/net/ipv4/tcp.c에 나와있다.

```

int tcp_recvmsg(struct sock *sk, struct msghdr *msg,
                int len, int nonblock, int flags, int *addr_len)
{
    struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
    int copied = 0;
    u32 peek_seq;
    u32 *seq;
    unsigned long used;

```

```

int err;
int target;           /* Read at least this many bytes */
long timeo;
struct task_struct *user_recv = NULL;

```

코드 401. tcp_recvmsg() 함수

tcp_recvmsg() 함수를 보자. tcp_recvmsg() 함수는 BSD socket 구조체와 메시지 헤더, 버퍼의 크기, non-blocking flag, 기타 flag과 주소의 길이를 넘겨받는다. BSD socket 구조체로 부터 TCP에 해당하는 옵션을 읽어오기 사용할 지역 변수를 할당한다.

```

lock_sock(sk);
TCP_CHECK_TIMER(sk);
err = -ENOTCONN;
if (sk->state == TCP_LISTEN)
    goto out;
timeo = sock_rcvtimeo(sk, nonblock);
/* Urgent data needs to be handled specially. */
if (flags & MSG_OOB)
    goto recv_urg;
seq = &tp->copied_seq;
if (flags & MSG_PEEK) {
    peek_seq = tp->copied_seq;
    seq = &peek_seq;
}
target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);

```

코드 402. tcp_recvmsg() 함수 – continued

함수의 주 부분으로 들어가기에 앞서 해주어야 할 부분들은 먼저 사용하는 BSD socket 구조체에 lock을 설정한다(lock_sock()). TCP의 타임머를 체크하고, BSD socket이 현재 LISTEN 상태라면 out으로 넘어간다. 받기 타임아웃에 사용할 값을 설정한 다음, 만약 넘겨받는 flag 값이 Out-Of-Band로 설정되어 있다면 recv_urg로 넘어간다. TCP의 옵션 필드에서 sequence 번호를 구하고, 다시 flag가 MSG_PEEK로 설정된 경우에는 TCP의 옵션 필드의 copied_seq로 부터 선택할(peek)할 sequence 번호를 가져온다. 그리고, 얼마만큼의 값을 읽을지를 설정한다(sock_rcvlowat())。

```

do {
    struct sk_buff *skb;
    u32 offset;
    /* Are we at urgent data? Stop if we have read anything. */
    if (copied && tp->urg_data && tp->urg_seq == *seq)
        break;
    /* We need to check signals first, to get correct SIGURG
     * handling. FIXME: Need to check this doesn't impact 1003.1g
     * and move it down to the bottom of the loop
     */
    if (signal_pending(current)) {
        if (copied)
            break;
        copied = timeo ? sock_intr_errno(timeo) : -EAGAIN;
        break;
    }
    /* Next get a buffer. */
    skb = skb_peek(&sk->receive_queue);

```

코드 403. tcp_recvmsg() 함수 – continued

여기서 부터는 loop를 돌면서 데이터를 읽어들이는 부분이다. 먼저 이곳에서 사용할 sk_buff구조체 변수를 선언하고, 현재의 읽기 offset이 어딘지를 나타내는 변수를 선언한다. 만약 급한 데이터(urgent data)를 가지고 있다면, 바로 loop를 끝낸다. 그리고, 현재 프로세스에 대해서 pending된 시그널이 있는지 확인하고, 만약 그런 시그널이 존재하고, 복사된 데이터가 있다면, error를 돌려주고 loop를 빠져나온다. 이제부터는 BSD socket에 queueing된 sk_buff에 대한 처리에 들어간다(skb_peek()). skb_peek()함수는 inline으로 정의 되어 있으며, 큐로부터 소켓 버퍼(sk_buff)를 하나씩 꺼내오는 역할을 한다.

```

do {
    if (!skb)
        break;
    /* Now that we have two receive queues this
     * shouldn't happen.
     */
    if (before(*seq, TCP_SKB_CB(skb)->seq)) {
        printk(KERN_INFO "recvmsg bug: copied %X seq %X\n",
               *seq, TCP_SKB_CB(skb)->seq);
        break;
    }
    offset = *seq - TCP_SKB_CB(skb)->seq;
    if (skb->h.th->syn)
        offset--;
    if (offset < skb->len)
        goto found_ok_skb;
    if (skb->h.th->fin)
        goto found_fin_ok;
    if (!(flags & MSG_PEEK))
        skb->used = 1;
    skb = skb->next;
} while (skb != (struct sk_buff *)&sk->receive_queue);

```

코드 404. tcp_recvmsg()함수 – continued

더 이상 가져올 sk_buff가 없다면 loop를 빠져나간다. 소켓버퍼(sk_buff)의 TCP header로부터 sequence넘버를 가져와서 offset을 계산한 다음, SYN를 나타낼 경우에는 offset을 1감소시키고, offset이 sk_buff의 길이보다 작을 경우에는 found_ok_skb로 넘어간다. 만약 TCP header에 FIN이 설정된 경우에는 다시 found_fin_ok로 넘어가게 되며, MSG_PEEK가 flag로 설정되지 않았다면, sk_buff를 사용되었다(used)고 놀는다. 이 loop는 BSD socket의 receive_queue를 다 소모할 때까지 계속된다.

```

/* Well, if we have backlog, try to process it now yet. */
if (copied >= target && sk->backlog.tail == NULL)
    break;
if (copied) {
    if (sk->err ||
        sk->state == TCP_CLOSE ||
        (sk->shutdown & RCV_SHUTDOWN) ||
        !timeo)
        break;
} else {
    if (sk->done)
        break;
    if (sk->err) {
        copied = sock_error(sk);
        break;
    }
    if (sk->shutdown & RCV_SHUTDOWN)
        break;
}

```

```

        if (sk->state == TCP_CLOSE) {
            if (!sk->done) {
                /* This occurs when user tries to read
                 * from never connected socket.
                 */
                copied = -ENOTCONN;
                break;
            }
            break;
        }
        if (!timeo) {
            copied = -EAGAIN;
            break;
        }
    }
cleanup_rbuf(sk, copied);

```

코드 405.tcp_recvmsg()함수 – continued

이미 복사된 것(받은 데이터의 복사)이 target을 넘어서거나, 다시 BSD socket의 backlog이 NULL일 경우에는 loop를 빠져나온다. 원가가 복사가 되었다면, 소켓의 상태를 확인하고 TCP_CLOSE상태를 나타내거나, 에러가 발생했을 경우, 그리고, shutdown되거나 timeout이 발생했다면, 다시 loop를 빠져나온다. 복사된 것이 없을 경우에는 BSD socket의 상태를 확인해서 에러나 혹은 앞에서와 비슷한 상황들에 대해서 처리를 해준다. cleanup_rbuf()함수는 sk_buff에를 해제(release)하고 필요할 경우 ACK메시지를 보내는 역할을 한다.

```

if (tp->ucopy.task == user_recv) {
    /* Install new reader */
    if (user_recv == NULL && !(flags&(MSG_TRUNC|MSG_PEEK))) {
        user_recv = current;
        tp->ucopy.task = user_recv;
        tp->ucopy.iov = msg->msg iov;
    }
    tp->ucopy.len = len;
    BUG_TRAP(tp->copied_seq == tp->rcv_nxt || (flags&(MSG_PEEK|MSG_TRUNC)));
    if (skb_queue_len(&tp->ucopy.prequeue))
        goto do_prequeue;
    /* __ Set realtime policy in scheduler __ */
}
if (copied >= target) {
    /* Do not sleep, just process backlog. */
    release_sock(sk);
    lock_sock(sk);
} else {
    timeo = tcp_data_wait(sk, timeo);
}

```

코드 406. tcp_recvmsg()함수 – continued

사용자 주소공간으로 복사하기를 원하는 프로세스가 user_recv(초기애 NULL로 설정)와 같다고, user_recv가 NULL이며, flag이 MSG_TRUNC와 MSG_PEEK이 설정되지 않은 경우에는 user_recv와 TCP옵션의 ucopy.task를 current프로세스로 만들며, 메시지 iovec를 이용해서 TCP옵션 필드의 ucopy.iov를 설정한다. 현재 복사된 크기를 len으로 설정하고, prequeue에 sk_buff가 들어있는지 확인한다.

만약 복사된 크기(copied)가 target보다 크거나 같다면, 소켓을 놓아준 다음 BSD socket의 lock을 대기중인 프로세스들을 깨우고(release_sock()) 다시 BSD socket에 lock을 걸려고 시도한다. 만약 복사된 양이 target양보다 작은 경우에는 tcp_data_wait()함수를 BSD socket에 대해서 timeout값을 넘겨주어 호출한다.

tcp_data_wait()함수는 BSD socket의 sleep필드에 현재 프로세스를 넣고, 현재 프로세스를 TASK_INTERRUPTIBLE로 만든다. 프로세스는 이 상태에서 잠들게 된다.

```

if (user_recv) {
    int chunk;
    /* __ Restore normal policy in scheduler __ */
    if ((chunk = len - tp->ucopy.len) != 0) {
        net_statistics[smp_processor_id()*2+1].TCPDirectCopyFromBacklog += chunk;
        len -= chunk;
        copied += chunk;
    }
    if (tp->recv_nxt == tp->copied_seq &&
        skb_queue_len(&tp->ucopy.prequeue)) {
do_pqueue:
        tcp_pqueue_process(sk);
        if ((chunk = len - tp->ucopy.len) != 0) {

            net_statistics[smp_processor_id()*2+1].TCPDirectCopyFromPrequeue += chunk;
            len -= chunk;
            copied += chunk;
        }
    }
    continue;
}

```

코드 407. tcp_recvmsg()함수 – continued

만약 user_recv필드가 NULL이 아니라면, CPU당 네트워크 통계정보를 업데이트 하고, 복사된 데이터의 양을 갱신한다. 이때 TCP의 다음번 받을 sequence번호가 복사된 sequence번호와 같고, prequeue된 값이 있을때 tcp_pqueue_process()를 호출해서 prequeue된 sk_buff를 처리한다. 물론 이와 관련된 CPU당 네트워크 통계정보와 복사 정보는 따라서 갱신된다. 이까지 진행했다면 다음번의 loop로 들어간다.

```

found_ok_skb:
    /* Ok so how much can we use? */
    used = skb->len - offset;
    if (len < used)
        used = len;
    /* Do we have urgent data here? */
    if (tp->urg_data) {
        u32 urg_offset = tp->urg_seq - *seq;
        if (urg_offset < used) {
            if (!urg_offset) {
                if (!sk->urginline) {
                    ++*seq;
                    offset++;
                    used--;
                }
            } else
                used = urg_offset;
        }
    }
}

```

코드 408. tcp_recvmsg()함수 – continued

소켓 버퍼의 길이가 offset보다 큰 소켓 버퍼를 찾은 경우다. offset을 소켓 버퍼에서 빼서 used필드로 넣은 다음, 이를 다시 남은 길이(len)과 비교한다. 남은 길이(len)이 작다면, 이를 used에 넣어준다. 만약

urgent데이터가 있다면, urgent sequence번호에서 현재 sequence번호를 뺀다음 urgent데이터의 offset으로 사용한다. 다시 이것이 used보다 작고 어떤 값을 지닌다면, used를 urgent 데이터의 offset을 used에 넣어준다. urgent offset이 없다면, BSD socket에 urgent데이터가 중간에 올 수 있는지를 확인한후, sequence번호와 offset을 증가시키고, used필드를 감소시킨다.

```

err = 0;
if (!(flags&MSG_TRUNC)) {
    err = memcpy_toiovec(msg->msg iov, ((unsigned char *)skb->h.th) + skb->h.th->doff*4
+ offset, used);
    if (err) {
        /* Exception. Bailout! */
        if (!copied)
            copied = -EFAULT;
        break;
    }
}
*seq += used;
copied += used;
len -= used;
if (after(tp->copied_seq, tp->urg_seq)) {
    tp->urg_data = 0;
    if (skb_queue_len(&tp->out_of_order_queue) == 0
#endif TCP_FORMAL_WINDOW
                                            && tcp_receive_window(tp)
#endif
    ) {
        tcp_fast_path_on(tp);
    }
}
if (used + offset < skb->len)
    continue;
/*      Process the FIN. We may also need to handle PSH
 *      here and make it break out of MSG_WAITALL.
 */
if (skb->h.th->fin)
    goto found_fin_ok;
if (flags & MSG_PEEK)
    continue;
skb->used = 1;
tcp_eat_skb(sk, skb);
continue;

```

코드 409. `tcp_recvmsg()`함수 – continued

`found_ok_skb`의 계속이다. `flag`에 `MSG_TRUNC`가 설정되지 않았다면, 메시지 헤더에 소켓버퍼의 `used` 만큼의 내용을 헤더를 제외하고 복사해 넣는다(`memcpy_toiovec()`). 만약 예러가 있다면 이것을 표시하고, `do {} while loop`를 끝낸다.

나머지는 `sequence`와 복사된 데이터의 양을 가르키는 정보의 갱신과 아직 더 남은 데이터가 있는지를 확인해서 `loop`를 더 돈다. 만약 `FIN`을 받았다면 `found_fin_ok`로 넘어가고, 소켓 버퍼는 사용되었다고 표시되며(`skb->used = 1`), `BSD socket`에서 사용된 `sk_buff`를 떼어낸다(`tcp_eat_skb()`). 모든 과정이 끝났다면 다시 `loop`를 돈다.

```

found_fin_ok:
    ++*seq;
    if (flags & MSG_PEEK)
        break;
    /* All is done. */

```

```

        skb->used = 1;
        break;
    } while (len > 0);
}

```

코드 410. tcp_recvmsg()함수 – continued

FIN0이 설정된 TCP헤더를 가지는 sk_buff를 받은 경우이다(found_fin_ok). sequence를 증가시키고, flag에 MSG_PEEK가 설정된 경우에는 loop를 빠져나온다. 그렇지 않다면, sk_buff를 사용되었다고 표시한 후 loop를 빠져나온다. loop는 요구한 데이터의 양(len)이 0보다 클때는 계속 순환한다.

```

if (user_recv) {
    if (skb_queue_len(&tp->ucopy.prequeue)) {
        int chunk;
        tp->ucopy.len = copied > 0 ? len : 0;
        tcp_pqueue_process(sk);
        if (copied > 0 && (chunk = len - tp->ucopy.len) != 0) {
            net_statistics[smp_processor_id()*2+1].TCPDirectCopyFromPrequeue += chunk;
            len -= chunk;
            copied += chunk;
        }
    }
    tp->ucopy.task = NULL;
    tp->ucopy.len = 0;
}
/* Clean up data we have read: This will do ACK frames. */
cleanup_rbuf(sk, copied);
TCP_CHECK_TIMER(sk);
release_sock(sk);
return copied;
out:
TCP_CHECK_TIMER(sk);
release_sock(sk);
return err;
recv_urg:
err = tcp_recv_urg(sk, timeo, msg, len, flags, addr_len);
goto out;
}

```

코드 411. tcp_recvmsg()함수 – continued

만약 user_recv(사용자 receive 프로세스)가 NULL이 아니라면 prequeue를 보고 이를 처리해주게 되며, 해당하는 네트워크 통계정보 및 복사된 양과 남은 요구 데이터의 양을 갱신한다. 그리고나서 TCP옵션 필드의 ucopy.task를 NULL로 만들고, ucopy.len을 0으로 다시 초기화 한다. 이와 같은 일이 끝나면 소켓 버퍼(sk_buff)를 해제하고(cleanup_rbuf), BSD socket의 TCP와 관련된 timer를 check하고, BSD socket에 대한 lock을 푼 후, 복사된 양을 돌려주고 복귀한다. Urgent데이터를 받은 경우에는 tcp_recv_urg()함수를 불러서 처리한다.

이상에서 우린 아주 단순화된 Linux에서의 network구현을 보았다. 이 과정을 전부 조합해서 하나의 그림으로 나타내자면 [그림43]과 같을 것이다.

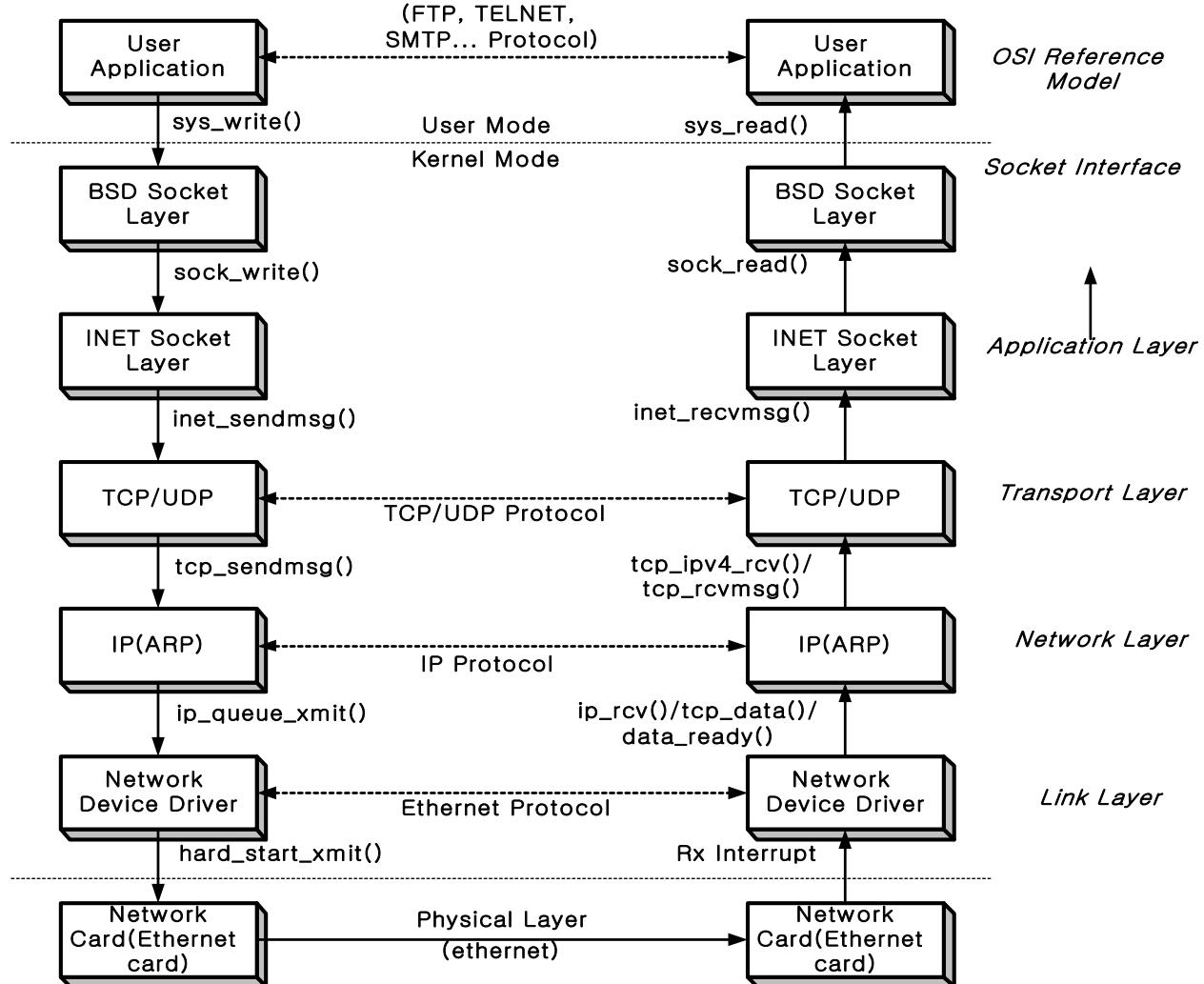


그림 50. Linux의 Network Architecture

5.11. ARP Layer

ARP(Address Resolution Protocol) 레이어는 하드웨어 주소를 논리적인 인터넷 주소(IP address)로 바꾸어주는 역할을 한다. 이것은 처음에 시스템이 네트워크에 맞물릴 경우 상대의 주소를 알지 못하므로, 먼저 현재 네트워크 상황(IP주소를 사용하는 시스템들의 주소를 알아보는 일)을 관찰해서 자신이 관리하는 테이블에 등록시켜둔다. 나중에 특정 IP주소로 데이터를 보내고자 할 때, 이 테이블에 등록된 하드웨어 주소를 근거로 받는 시스템에 대한 주소를 알게 된다. 따라서, 데이터는 보내지기 전에 이 테이블을 반드시 검색해서 적절한 주소를 찾아야 한다. 만약 테이블에서 등록된 주소를 찾지 못한다면 ARP 프로토콜을 사용해서 반드시 하드웨어의 주소를 알아본다.

하지만, 이러한 ARP를 상요할 필요가 없는 경우가 있다. 가령 PPP나 SLIP를 사용하는 경우, 상대방에 대한 연결은 단지 하나만이 존재하기에, 하드웨어의 주소를 알지 못하더라도 데이터를 보내는 것은 가능하다. Ethernet을 사용하는 경우에는 수많은 컴퓨터들이 네트워크에 물려있는 상황이기에 반드시 특정 컴퓨터의 하드웨어 주소를 알아야만 통신을 할 수 있는 것이다. IP 주소를 모르는 경우에는 broadcasting 방식으로 데이터를 주고 받는 것 만이 가능하다.

ARP레이어는 자신의 테이블(routing table)을 관리하기 위해서 사용자의 간섭을 요구하기도 한다. 즉, 기본적인 시스템의 설정은 사용자가 해 주어야 한다는 것이다. 나머지 상대방에 대한 주소는 ARP 프로토콜을 이용해서 상대방에게 물어보게 되며, 이때 물어보는 정보는 broadcasting의 형태로 날아가게

된다. 만약 다른 시스템에서 이렇게 날아온 정보를 받게 되면, 자신의 하드웨어 주소와 IP주소를 같이 실어서 날려준 시스템에 다시 보내주게 되며, 자신의 테이블도 받은 정보를 토대로 다시 갱신한다.¹⁴⁵

5.12. Network Device Driver Layer

네트워크 디바이스 드라이버 레이어는 나중에 디바이스 드라이버를 살펴보게 될 때 이야기를 할 것이지만 여기서는 간단히 그 구조를 살펴보고 마무리 하도록 한다. 네트워크 디바이스 드라이버는 상위에서 어떤 데이터를 받는지는 신경쓰지 않는다. 상위에 어떤 프로토콜이 오는지는 신경쓰지 않는다. 다만 자신이 원하는 것은 소켓버퍼(socket buffer)의 형태로 데이터가 자신에게 전달 된다는 사실과 다시 상위로 데이터를 전달하기 위해서 소켓버퍼를 만들어 주어야 한다는 사실이다.

모든 디바이스는 사용하기 위해서 먼저 열여야(open)한다. 또한 사용되기 이전에 디바이스를 초기화(initialization)하는 것도 필요하다. 또한 디바이스에 대한 쓰기(write)와 읽기(read)가 있을 수 있으며, 디바이스에 특정한 명령을 수행(ioctl)할 필요도 있을 것이다. 그리고, 현재 디바이스의 상태(statistics)을 알기를 원할 경우, 이를 지원해 줄 수 있어야 할 것이며, 마지막으로 사용을 마칠 때 닫기(close: release)를 행해야 한다. 따라서.. 이러한 연산들을 커널로부터 요구 받을 수 있으므로 커널에게 디바이스에서 이러한 연산을 지원하고 있으니, 불러서 사용하라고 알려주어야 할 것이다. 나중에 커널은 이러한 디바이스에서 제공하는 연산을 호출해서 사용자의 요구를 맞춰줄 것이다.

기본적으로 전송을 위한 커널과 네트워크 디바이스 드라이버 간의 인터페이스는 `hard_start_xmit()` 함수가 하고 있다. 또한 전송이 끝나거나, 새로운 패킷을 네트워크에서 받았을 경우에는 인터럽트가 발생하며, 새로운 패킷의 도착을 알리는 것은 `netif_rx()`가 맡고 있다. 따라서, 인터럽트를 할당 받았을 경우, 인터럽트가 발생하면 반드시 이것이 Tx를 위한 것인지, 아니면 Rx를 위한 것인지를 확인해야 할 것이다. Tx 인터럽트가 발생했다면, 이전에 Tx를 위해서 저장해 두었던 버퍼를 지우는 작업과 Tx의 상태 정보를 모으는 작업이 추가 되며, Rx의 경우에는 앞에 설명한 받은 패킷을 상위의 layer에 알리는 일과 Rx의 상태를 정보를 모으는 일이 추가 될 것이다. 새로운 패킷의 도착을 위해서 이곳에서 새 `skb_buff`를 할당하는 것도 잊지 말도록 하자.

자세한 것은 뒤에 나올 네트워크 디바이스 드라이버를 참고하기 바란다.

¹⁴⁵ ARP에 대한 것은 리눅스의 네트워크를 설명하는데 필수적인 요소지만 일단은 이 정도를 보는 것으로 만족하도록 하자.

6. Linux Device Driver Basic

이번 장에서는 특별히 Device Driver에 대해서 살펴보도록 하겠다. 이와 같이 특정한 한 chapter로 device driver를 다루는 것은 실제 대부분의 kernel 코드가 다양한 device들을 지원하기 위해서 작성 되었다고 생각되기 때문이다. 초기에는 특정의 몇몇 장치만을 제공했던 운영체제가 점차 그 효용성을 넓혀가게 되면, 많은 시장에 나온 장치들에 대한 지원이 빠질 수 없다. 리눅스 역시 이러한 점을 감안해서 2.4 version의 kernel에서는 2.2 version의 커널에서 보다 더 다양하고 많은 device들에 대한 지원을 하고 있다.

디바이스 드라이버는 크게 3개로 나누어 볼 수 있다. 대부분의 Unix의 운영체제의 경우 문자(character) 디바이스 드라이버와 블록(block) 디바이스 드라이버를 제공하고 있으며, 나머지 하나가 네트워크(network) 디바이스 드라이버이다. 때로 네트워크 디바이스 드라이버를 스트림(stream) 디바이스 드라이버로 부르기도 한다.¹⁴⁶ 디바이스 드라이버들은 커널에 정적으로 링크(link)되어서 사용되기도 하며, 때로는 동적으로 운영체제가 기동중인 상태에서 커널과 링크되거나 제거되어질 수도 있다. 이를 위해서 리눅스에서는 모듈(module) 기능을 제공하고 있으며, 리눅스 커널 2.4에서는 커널내에 모듈을 동적으로 링크하거나 제거하기 위한 기능을 가지고 있다. 모듈로서 제공되는 드라이버들은 완전히 모든 심벌(symbol)이 해당하는 주소를 가지지는 않으며, 나중에 커널에서 제공하는 도구를 이용해서 해결되지 않은 심벌들의 주소를 가지고 올 수 있게 된다. 따라서, 커널에서 제공하는 함수를 모듈에서도 사용할 수 있게 되는 것이다.

Device driver란 운영체제와 hardware 간의 정보 전달을 목적으로 만들어진다. 즉, 운영체제의 요구를 hardware에 전달 하는 역할을 한다. 이러한 device driver들은 운영체제와 밀접한 관계를 가지게 됨으로 운영체제의 내부적인 데이터 이동을 명확하게 이해하는 것을 필요로 한다. 또한 각각의 device가 시스템에 어떻게 연결되는지에 대해서도 명확히 이해해야 한다. 예를 들자면 device가 ISA slot에 연결되는지 아니면 PCI slot에 연결되는지를 알아야 한다. 이것은 나중에 device를 access하게 될 때 주소를 지정하는 것과 밀접한 관련이 있기에 이것의 이해는 필수적이다. 예를 들어서 지금 우리가 다루려고 하는 network 카드는 PCI slot에 끼워져서 사용되며, device의 기본 주소는 PCI specification에 따라서 정해지며, 그것을 기준으로 device의 register들의 번지가 정해진다.

Unix에서의 device는 file로서 access가 가능하며, open, read, write, close, ioctl과 같은 일반적인 interface를 가지고 있다. 같은 device driver에 의해서 control되는 모든 device들은 같은 major number가 주어지며, 또한 같은 major number를 가지는 다른 device들은 minor number로 구분이 된다. 즉, device들은 major number와 minor number의 pair로 구분이 가능하게 된다.

Linux에서는 다음과 같은 종류의 device driver들이 있다. 즉, character device, block device, scsi device, network device 등이 있다. 이 문서에서 지켜볼 것은 network device에 대해서 살펴보는 것으로 한정한다. 또한 device driver는 Linux kernel의 일부가 되기 때문에 모든 memory space에 대한 접근이 가능하며, 원하는 것은 모든 것을 할 수 있다. 예를 들어서 memory location의 어느 부분에도 write가 가능하며, 모니터나 hard disk에 치명적인 영향을 줄 수도 있다. 따라서, 작성하는 것에 있어서는 모든 경우에 대해서 대비하는 것이 필요하다. 또한 device driver는 kernel의 하위에서 실행되기에 pre-emptible하지가 않으므로 전체적인 system에 영향을 미칠 수 있다. 따라서 여러분의 작성하는 code는 자신이 하려고 하는 일을 가능한 한 빠른 시간에 마칠 수 있어야 한다.

대체적으로 소용량의 interactive한 데이터의 전송에는 character device driver에 해당하는 것이 사용되며, 대용량의 buffer 할당이 요구되는 것에는 block device driver가 사용된다고 보면 된다. scsi와 network device에 대해서는 이미 이름 자체에서 내포하는 것으로 알 수 있듯이 scsi와 network device driver를 의미한다.

이후에는 커널에서 수행되는 디바이스 드라이버들과 이를 지원하기 위한 모듈 프로그램 방법, 그리고 디바이스 드라이버를 이해하기 위해서 알아야 하는 기본적인 하드웨어 구조등에 대해서 살펴볼 것이다.

¹⁴⁶ SCO unix의 경우 stream driver라고 부른다.

6.1. Kernel structure에 대한 이해

Device driver에 대해서 살펴보기 전에 먼저 Linux 운영체제(Operating System)의 구조에 대해 간단히 살펴보도록 하자. 앞장에서 이미 이러한 구조를 충분히 보았겠지만, 이곳에서는 보고자 하는 부분은 각 운영체제의 구성요소의 아래에는 무엇이 있는가에 중점을 둔다. 먼저 운영체제는 다음과 같은 크게 네 부분으로 나누어진다.

- Memory management – memory 관리에 대한 부분이다. 이곳은 프로세스의 메모리 관리를 담당하는 부분으로 프로세스의 주소 공간을 만들어주는 역할을 하며, 디스크로부터 파일을 읽어오거나 쓰는 연산이 필요하다.
- Process management – 실행중인 프로그램의 memory내의 image에 대한 관리에 대한 부분이다. 프로세스 관리가 운영체제의 핵심이다. 모든 일은 이 프로세스에 중점을 두고 기술될 것이다. 즉, 운영체제는 사용자 프로그램이 잘 수행되도록 돋는 역할만을 담당하고 있다.
- File System – 저장 장치에 어떻게 파일을 저장하고 그것을 다시 가지고 올 것인가에 대한 부분이다. 파일 시스템은 파일의 저장과 읽어들이는 일을 담당하며, 이는 또한 프로세스가 일상적으로 실행하는데 필요한 자원들을 할당하고, 해제하는 일이 필요하기에 메모리 관리 및 버퍼의 관리와 반드시 따라야 한다.
- Network – network 관리에 대한 부분이다. 네트워크는 실제로 감춰진 부분으로 구현되고 있다. 일반 파일에 대한 연산을 네트워크에 대한 연산으로 바꾸어서 실제로는 네트워크에 대한 접근 연산이 되도록 만들고 있다.

이렇게 4개의 범주로 나누어진 운영체제는 사용자의 응용 프로그램이 수행되는 환경을 제공한다. 따라서, 사용자의 프로그램을 저장장치에 저장하거나 메모리로 읽어 들여 수행하고 결과를 화면이나 기타 출력장치에 표시하는 역할을 한다. 즉, 이러한 모든 일은 응용프로그램과 하위의 장치 사이에서 일어나며, 이를 운영체제에서 관리한다고 생각하면 된다.

Linux에서는 이러한 일을 해주는 부분이 kernel이라는 이름의 프로그램을 존재하며, [그림44]와 같은 구조를 가진다.

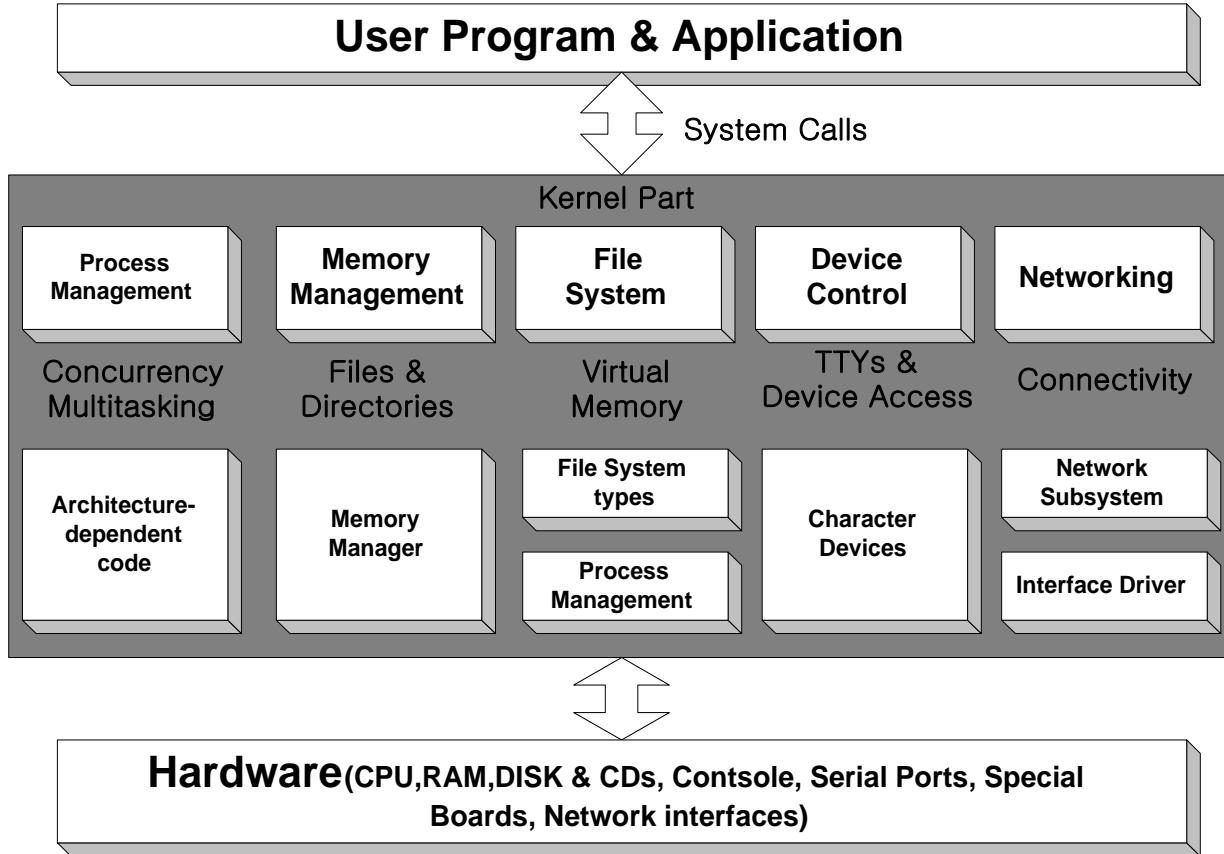


그림 51. Linux Kernel의 구조

Kernel은 빠른 속도로 사용자 프로그램에 대해서 응답을 해야 함으로 속도에 민감하게 반응하게 된다. 사용자의 프로그램은 kernel에 대한 접근을 system call의 형태로 하게 되면, 이러한 경우 kernel이 제어권을 넘겨 받게 된다. 사용자의 요구를 처리하기 위해서 kernel은 장치들에 대한 요구를 시작하게 되며, 장치들에 대한 요구는 device driver를 통해서 장치에 전달된다. 다시 결과는 device driver에 의해서 kernel로 전송된다. 결과적으로 kernel은 사용자 프로그램의 요구를 만족하게 되며, 사용자 프로그램으로 다시 제어권이 넘어가게 된다. 물론 이것은 간단한 사용자 프로그램과 kernel, device driver, 장치간의 동작에 대해서 간단히 보여준 것에 불과하며, 다른 몇 가지의 경우에 대해서도 kernel은 제어권을 가지게 될 수 있다.

6.2. 기본 하드웨어의 구조

이번 장에서는 기본적으로 디바이스 드라이버를 작성하기 위한 하드웨어의 기초 지식을 보도록 하겠다. 디바이스 드라이버를 작성하기 위해서는 가장 먼저 알아야 하는 것은 우리가 만들고자 하는 디바이스 드라이버로 구동되는 디바이스가 시스템에 어떻게 인터페이스를 하고 있는 가를 이해하는 일이다. Serial과 같은 것은 serial port를 통해서 연결될 것이며, Graphic card는 PCI나 혹은 AGP slot을 통해서 연결될 것이다. 즉, CPU와 데이터를 교환하기 위해서 어떤 통로를 사용하는지를 알아야만 한다는 말이된다.

6.2.1. Bus의 이해

Bus란 device와 CPU, memory 간의 정보전송 통로가 된다. 이러한 bus를 이용하게 될 때는 먼저 정보 전송의 목표가 되는 주소가 있어야 하는데 이 바로 여기서 device의 기본주소(base address)를 정해주게 된다. 따라서, 이것의 이해가 선행되어야 할 것이다. 다음과 같은 bus가 있으며, 우리가 최종적으로 이용하고자 하는 것은 PCI bus가 되겠다. [그림45]에서는 Bus 상에서의 device의 address가 어떻게 지정되는지를 보여주고 있다. 이와 같이 주소가 있는 장치에 대해서만 접근이 가능하다는 것을 명심하기 바란다.

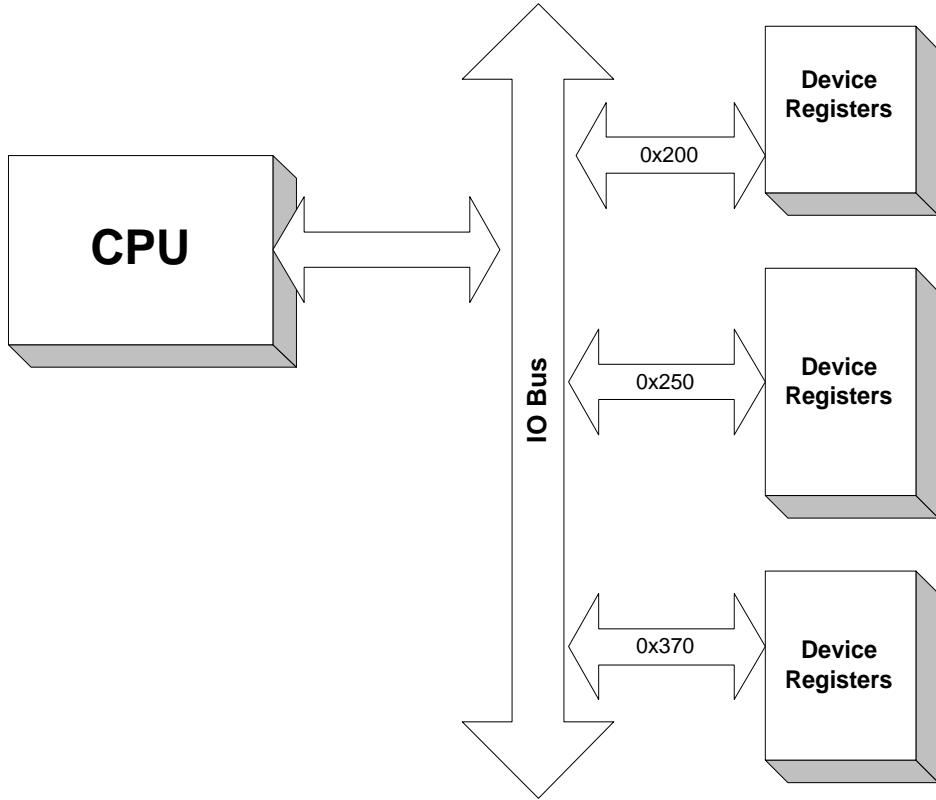


그림 52. I/O Address space

6.2.1.1. ISA Bus

ISA(Industry Standard Architecture) bus는 가장 오래된 bus로 가장 많이 사용되어 왔다. Original ISA bus는 24개의 address line에 대해서만 작동하기에 16MB의 메모리가 addressing될 수 있다. 16MB이상의 메모리를 가지는 시스템에서의 ISA bus이외에 메모리 전용의 32bit의 memory bus가 있다. [그림46]은 ISA bus의 address를 보여주고 있다.

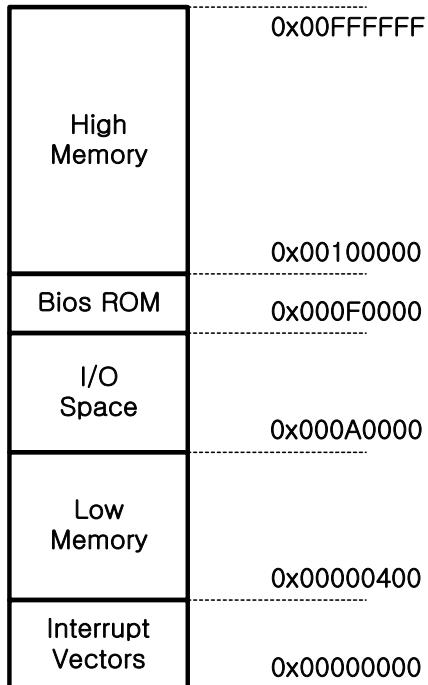


그림 53. ISA Bus상의 address

운영체제에서는 이러한 ISA 디바이스를 위한 DMA메모리 영역을 가지고 있으며, PC의 경우 크게 4개의 DMA channel을 하나의 디바이스 드라이버가 최대로 할당 받을 수 있다. 따라서, 희소한 자원을 사용하는 일이 되기에, 반드시 사용이 끝나면 할당받은 자원들을 운영체제에 반환하는 일이 필요하다.

6.2.1.2. Micro Channel Architecture(MCA) Bus

ISA bus보다 기술적으로는 월등했으나, 시장에서는 실패를 했다. 이것은 IBM에 의해서 ISA bus를 누르려는 목적으로 만들어졌으나 특허권의 문제로 시장에서는 성공적이지 못했다. 다음과 같은 특성들이 다른 bus architecture를 만드는데 많은 도움이 되었다.

- Self-identifying devices(자신을 스스로 인식하게 만드는 device)
- DIP switch보다는 software에 의해서 configuration(설정)을 할 수 있다.
- Level triggered interrupt이며, 여러 개의 device들이 같은 interrupt line을 공유할 수 있다.
- Bus master가 될 수 있다.
- Distributed DMA control

여기서 Bus master가 된다는 것은 DMA(Direct Memory Access)부분에서 살펴보게 되겠지만, 자신이 직접 bus에 데이터를 옮겨서 전송을 할 수 있다는 것을 의미한다. 즉, 다른 device에 의해서 data의 전송 명령을 받아야지 전송을 하는 것이 아니라 자신이 직접 data의 전송명령을 내릴 수 있다는 것이다. 또한 interrupt line이란 PC에서는 interrupt를 구분하기 위해서 사용하는 line의 수가 한정되어 있으므로 공유할 수 있다는 것은 더 많은 resource를 제공해 준다는 말과 같다.

6.2.1.3. EISA Bus

EISA(Extended Industry Standard Architecture) bus는 IBM MCA bus에 대항하는 업체들이 모여서 만들었다. 목표는 MCA의 여러 가지 특성들을 가지도록 만들자는 것이다. EISA는 ISA bus를 subset으로 가지기에 8bit PC-bus card나 16bit ISA bus card, 16bit 혹은 EISA-bus card를 끼울 수 있다. EISA bus는 완전한 32-bit address와 32-bit data 통로를 제공한다는 점을 명심하기 바란다. [그림47]은 EISA bus의 architecture를 보여준다.

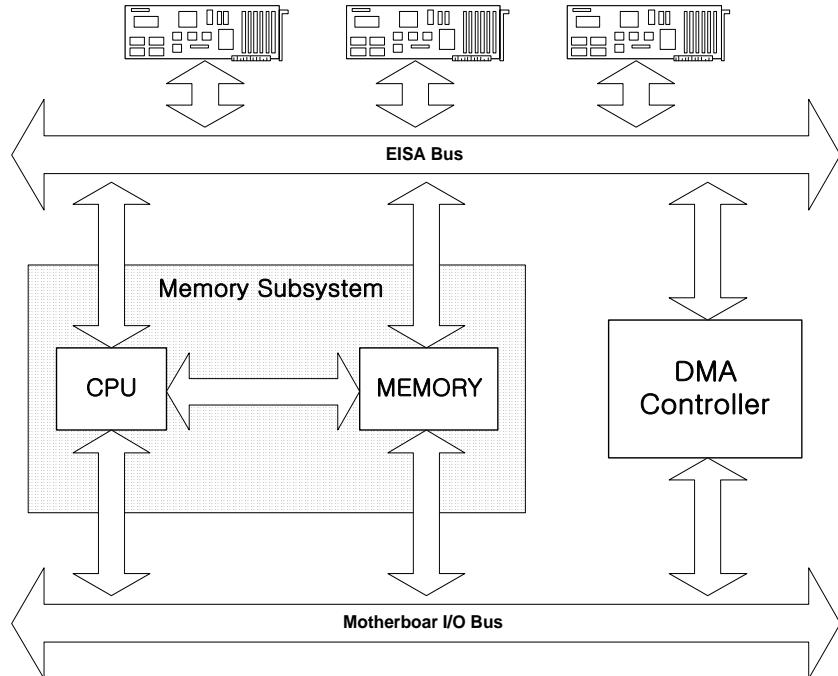


그림 54. EISA bus architecture

그리고, EISA Bus는 16개의 interrupt level을 지원한다. [표41]은 그것을 보여주고 있다. 표에서 보듯이 interrupt의 우선 순위 및 그것의 배치는 연속적이지 않다는 점이다. 이는 두개의 interrupt generator chip을 연결해서 사용하게 됨으로 해서 생기는 현상이다. 또한 interrupt를 발생시키는 방법에 두 가지가 있기에 같은 interrupt line에 두개나 그 이상의 EISA card를 사용할 수 있다.

Level	Priority	User For
IRQ0	1	8253 Timer Channel 0
IRQ1	2	Keyboard
IRQ2	-	Cascade from IRQ 9
IRQ3	11	COM2
IRQ4	12	COM1
IRQ5	13	LPT2 or available
IRQ6	14	Floppy disk
IRQ7	15	LPT1 or available
IRQ8	3	Real-time clock
IRQ9	4	Redirected to IRQ 2
IRQ10	5	Available
IRQ11	6	Available
IRQ12	7	Available
IRQ13	8	Math coprocessor
IRQ14	9	Hard disk controller
IRQ15	10	Available

표 42. EISA Bus IRQ Assignments

6.2.1.4. PCI Bus

ISA나 EISA, 그리고 MCA bus의 가장 심각한 한계는 총 bus의 bandwidth이다. 즉, 많은 데이터를 빠른 시간에 전송하지 못한다는 단점을 가지고 있다. PCI bus의 경우에는 데이터의 전송을 266MB/sec까지 낼 수 있다. PCI(Peripheral Component Interconnect) Bus의는 새로운 standard로 Alpha 및 Intel에서도

동작한다. PCI bus는 ISA나 EISA보다는 훨씬 복잡한 구조를 가지며, 다음과 같은 3개의 다른 bus system을 가진다.

- Processor/Memory bus
- PCI Bus
- Expansion bus(ISA 혹은 EISA)

[그림48]은 PCI Bus의 구조를 보여준다. PCI Bus의 performance의 가장 중요한 요소는 PCI Bridge가 buffer를 가지고 있다는 점과 sequential한 데이터의 전송요구를 한번의 burst로 합성해 준다는 점이다. 이것은 PCI Bus와 processor가 병렬(parallel)적으로 PCI 전송 동안에 동작할 수 있도록 해준다. 또한 PCI controller는 configuration data area에 대한 addressing을 허락하는데, 이 area에는 autoconfiguration data를 가진다. PCI Bridge는 ISA/EISA address와 PCI address, 그리고 PCI configuration address를 access할 수 있도록 해준다.. 그리고, PCI Bus는 multiple bus master를 지원하여, 한 device가 다른 device들에 대해서 processor의 개입 없이 직접 data를 전송할 수 있다.

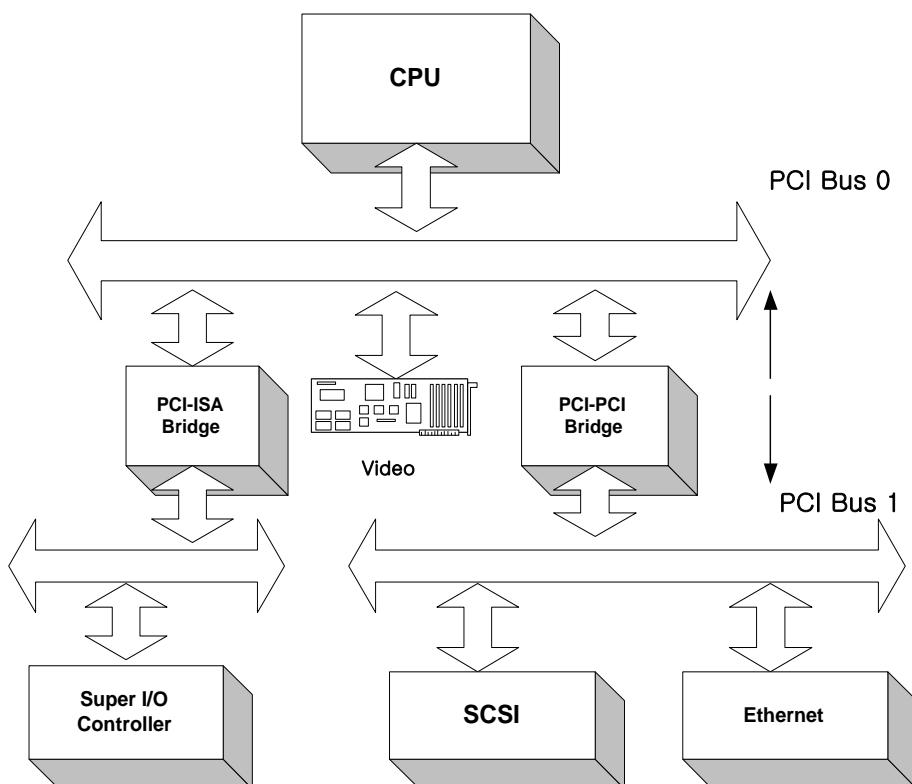


그림 55.PCI Bus Architecture.

다음으로 명심해서 보아야 할 부분은 PCI Configuration memory space이다. 이곳에는 우리가 원하는 PCI device specific한 정보를 담을 수 있다. 여기서 우리는 우리가 만드는 driver가 제대로 맞는 device를 찾을 수 있도록 하는 정보와 Device의 register의 base address과 같은 driver의 IO가 필요로 하는 address를 찾게 된다. 또한 그 외에도 여러 가지 정보가 있다. [표42]는 이러한 PCI device의 configuration memory space를 보여주고 있다.

Device ID	Vendor ID	0x00
Status	Command	0x04
Class Code		0x08

BIST	Hdr	LATENCY	CACHE	
	Base Address 0			0x0C
	Base Address 1			0x10
	Base Address 2			0x14
	Base Address 3			0x18
	Base Address 4			0x1C
	Base Address 5			0x20
	CardBus CIS Pointer			0x24
Subsystem ID	Subsys Vendor ID			0x28
	Expansion BIOS ROM Addr			0x2C
	Reserved			0x30
	Reserved			0x34
MaxLat	MinGnt	Int Pin	Int Line	0x38
				0x3C

표 43. PCI device의 configuration address space

이러한 정보 중에서 다음과 같은 것을 좀더 자세히 보도록 하자.

- Vendor Identification : PCI장치의 고유한 제작자를 나타내는 고유 번호이다. 이것은 window NT 같은 곳에서 device에 맞는 드라이버를 설정하는데 필요한 정보를 제공한다. 즉, 이것과 함께, 장치 식별자를 뒤어서 하나의 ID를 만들어 이것이 올바른 driver를 선택하는데 필요한 정보를 제공하게 되는 것이다.
- Device Identification : 장치 자체를 나타내는 고유 번호이다. 이것은 위에서 살펴본 것과 같은 형태로 사용된다. Linux에서는 device의 configuration을 읽어서 driver가 사용하고자 하는 device 가 있는지를 확인하는데 사용하게 된다.
- Status : 장치의 상태를 나타낸다.
- Class code : 장치의 유형을 구분한다.
- Base Address Register : PCI device의 register들에 대한 기본적인 주소를 표시한다. 즉, 이것과 offset을 더해서 사용하고자 하는 register의 주소를 알 수 있게 된다.
- Interrupt Pin : PCI card로부터 PCI Bus로 interrupt를 전송하는 역할을 한다.
- Interrupt Line : 인터럽트 handler가 PCI 장치로부터 온 interrupt를 Linux 운영체제에 있는 올바른 디바이스 드라이버의 인터럽트 처리 코드로 인터럽트를 전달할 수 있도록 한다.

이러한 PCI configuration space에 대한 접근은 운영체제가 정의하고 있는 BIOS function에 의해서 접근이 가능하다. 이 부분을 읽어드려서 device driver에서 사용하고자 하는 정보들을 얻어야 한다. 이것은 device driver의 load초기에 해주어야 할 일 중의 일부이다.

여기서 간단히 DMA(Direct Memory Access)를 짚고 넘어가도록 하자. DMA를 할 수 있는 PCI card는 bus master이다. 이것은 자기 자신만의 control register를 가지고 있으며, DMA같은 데이터의 전송을 책임진다. DMA transfer에는 반드시 두개의 PCI device가 있게 되며, 하나는 initiator가 되고 나머지 하나는 target이 된다. Initiator가 되는 PCI device는 bus master가 되는 것이다. Target이 되는 PCI device는 initiator에 의해서 address가 되며, initiator에 의해서 요구된 data의 주고 받기를 하게 된다.

PCI를 사용하는 디바이스가 요즘은 주를 이루기에, 리눅스에서 이것에 대해서 어떻게 구현하고 있는가를 따로이 나누어서 보도록 할 것이다.

6.2.1.5.PCI Bridge

PCI Bridge는 Bus와 Bus간을 연결하는 역할을 한다. 이러한 역할을 하는 것으로는 PCI-ISA Bridge, PCI-PCI Bridge가 있다. 이러한 bridge를 넘어갈 경우에는 device의 주소공간을 mapping하는 절차가 필요하게 된다.

6.2.1.6. PCI에서 사용하게될 기본 함수들

이상에서 우리는 우리가 작성하고자 하는 device driver가 사용하게 될 주소에 대해서 알게 되었다. 즉, 기본 주소(base address)가 결정되면, 그것을 기준으로 해서 device의 register에 대한 접근이 가능하게 됨을 의미한다. 방법은 기본주소 더하기 offset이 될 것이다. 이렇게 결정된 register에 대한 operation을 통해서 데이터의 전송을 control하게 되는 것이다. PCI Bus상에 놓인 device에 대한 function과 이것에 관련된 것들로는 다음과 같은 것들이 있다¹⁴⁷.

- #include <linux/config.h> – CONFIG_PCI : 0이 macro는 PCI와 관련된 code부분을 조건적으로 compile하게 될 때 사용한다.
- #include <linux/pci.h> – PCI register들에 대한 이름과 여러 개의 vendor와 deviceID를 나타내는 상수 값들이 정의 되어 있다. 이러한 값을 이용해서 PCI function을 사용하게 된다. 물론 새로이 만들려는 device의 경우에는 vendor와 deviceID가 없으므로 단순히 program내에서 정의해서 사용하면 될 것이다.
- #include <linux/bios32.h> – 모든 pcibios_ function들의 정의가 나와 있다. 하지만, 이것은 현재의 kernel version에서는 include해서 compile할 경우 warning message를 화면에 뿌리게 될 것이다. Kernel documentation을 읽어보면 새로운 PCI bios function을 쓰는 것을 권장하고 있다. 0이 header file에서 정의된 function에 대해서 살펴보도록 하자.
 - int pcibios_find_device(unsigned short vendor, unsigned short id, unsigned short index, unsigned char *bus, unsigned char *function);
 - int pcibios_find_class(unsigned int class_code, unsigned short index, unsigned char *bus, unsigned char *function);

이 같은 function은 성공했을 경우에는 pointer로 넘어간 변수에 값이 할당되어 돌아온다. 다음으로 살펴볼 것은 PCI device configuration register를 읽고 쓰는데 사용된다.

- pcibios_read_config_byte(unsigned char bus, unsigned char function, unsigned char where, unsigned char *ptr);
- pcibios_read_config_word(unsigned char bus, unsigned char function, unsigned char where, unsigned char *ptr);
- pcibios_read_config_dword(unsigned char bus, unsigned char function, unsigned char where, unsigned char *ptr);
- pcibios_write_config_byte(unsigned char bus, unsigned char function, unsigned char where, unsigned char val);
- pcibios_write_config_word(unsigned char bus, unsigned char function, unsigned char where, unsigned char val);
- pcibios_write_config_dword(unsigned char bus, unsigned char function, unsigned char where, unsigned char val);

여기서 한가지 유념할 것은 PCI bus는 little-endian이라는 사실이다. 따라서, multi-byte로 된 값을 접근할 경우에는 byte ordering에 신경을 써야 한다.

그럼, 위에서 보인 PCI Bios function을 이용해서 PCI slot에 있는 device를 검출하는 routine은 다음과 같이 작성될 수 있다. 이 코드는 ~/drivers/net/eopro100.c에서 PCI를 쓰는 card에 대한 검출에서 가져온 것이다. 이 코드는 커널 버전 2.2.X에 해당하는 것으로 단순히 참조만 하도록 하자.

```

if (!pcibios_present())
    return cards_found;
for (chip_idx = 0; pci_tbl[chip_idx].name; chip_idx++) {
    for (; pci_tbl[chip_idx].pci_index < 8; pci_tbl[chip_idx].pci_index++) {

```

¹⁴⁷ 현재 커널 버전 2.4.X에서는 이러한 방법을 사용하지 않고 있다. 디바이스 드라이버에서 PCI 디바이스의 table에 들어갈 항목을 정의해서 이를 pci_driver 구조체로 두고, 이를 pci_register_driver() 함수를 이용해서 등록하는 방법을 사용한다. 여기서 보여주고자 하는 것은 과거에는 어떤 식으로 PCI BIOS를 이용해서 드라이버를 검출했는가이다.

```

        unsigned char pci_bus, pci_device_fn, pci_latency;
        unsigned long pciaddr;
        long ioaddr;
        int irq;
        u16 pci_command, new_command;

        if (pcibios_find_device(pci_tbl[chip_idx].vendor_id,
                               pci_tbl[chip_idx].device_id,
                               pci_tbl[chip_idx].pci_index, &pci_bus,
                               &pci_device_fn))
            break;
    {
        pdev = pci_find_slot(pci_bus, pci_device_fn);
        pciaddr = pci_base_address(pdev, 1); /* Use [0] to mem-map
        */
# ifdef USE_IO
        pciaddr = pci_base_address(pdev, 0);
# endif
        irq = pdev->irq;
    }
}

```

코드 412. PCI 카드의 검출

가장 먼저 PCI BIOS 존재 여부를 묻는 pcibios_present()함수를 호출한다. 아무것도 없다면, 현재 검색된 카드의 수를 나타내는 card_found(=0)를 돌려준다. 만약 카드가 있다면 이하의 부분을 수행하게 되며, pci_tbl[]배열을 검색해서 해당하는 카드를 찾는다. pcibios_find_device()함수는 하드웨어 vendor ID(vendor_id), 디바이스의 id(device_id), 그리고 PCI 디바이스의 index값과 찾은 디바이스의 PCI BUS (pci_bus) 및 PCI 디바이스 기능(function)을 알려주는 필드를 파라미터로 받는다. 돌려주는 값은 해당 카드를 찾았을 경우에는 0이 아닌 값이 될 것이며, pci_bus와 pci_device_fn을 설정할 것이다.

이전 찾은 카드에 대해 slot번호를 구한다(pci_find_slot()). 넘겨주는 값은 앞에서 찾은 pci_bus와 pci_device_fn 값이다. 이것을 이용해서 pdev를 알게되며, 이때 설정된 인터럽트 번호도 알게된다. 이젠 pdev값으로 pci_base_address()를 호출해서 해당 디바이스가 사용하는 기본 주소(base address)를 구해온다. 이 값은 I/O mapped I/O를 사용하는지, 아니면 메모리 mapped I/O를 사용하는지에 따라서 pci_base_address()에 넘겨주는 값에 따라서 달라질 것이다. 해당 디바이스에 할당된 IRQ는 나중에 인터럽트 핸들러를 설치하고자 할 때 사용할 것이므로 보관하도록 하자(irq).

```

/* Remove I/O space marker in bit 0. */
if (pciaddr & 1) {
    ioaddr = pciaddr & ~3UL;
    if (check_region(ioaddr, 32))
        continue;
} else {
#endifdef __sparc__
    /* ioremap is hosed in 2.2.x on Sparc. */
    ioaddr = pciaddr & ~0xfUL;
#else
    if ((ioaddr = (long)ioremap(pciaddr & ~0xfUL, 0x1000)) == 0) {
        printk(KERN_INFO "Failed to map PCI address %#lx.\n",
               pciaddr);
        continue;
}

```

```

        }
#endif
}

```

코드 413. PCI 카드의 검출 – continued

얼어온 기본 주소에서 bit 0에 값이 있는지를 확인해서, LSB 2bit을 지우고, 이 값으로 ioaddr를 설정해서 올바른 영역을 나타내는지 확인한다(check_region()). 만약 bit 0에 아무 값도 가지지 않는다면, __sparc__정의가 있는 경우에는 구해온 주소에서 완전히 한 byte를 지워주고, __sparc__정의가 없다면 ioremap()함수에 pciaddr에서 한 byte를 지운값과 페이지 크기(0x1000)만큼을 넘겨서, 새로운 PCI 기본 주소(base address)에 대한 메모리 맵핑을 한다.

```

/* Get and check the bus-master and latency values. */
pcibios_read_config_word(pci_bus, pci_device_fn,
                         PCI_COMMAND, &pci_command);
new_command=           pci_command
PCI_COMMAND_MASTER|PCI_COMMAND_IO;
if (pci_command != new_command) {
    printk(KERN_INFO " The PCI BIOS has not enabled this"
          " device! Updating PCI command %4.4x->%4.4x.%Wn",
          pci_command, new_command);
    pcibios_write_config_word(pci_bus, pci_device_fn,
PCI_COMMAND, new_command);
}
pcibios_read_config_byte(pci_bus, pci_device_fn,
                         PCI_LATENCY_TIMER, &pci_latency);
if (pci_latency < 32) {
    printk(" PCI latency timer (CFLT) is unreasonably low at %."
          " Setting to 32 clocks.%Wn", pci_latency);
    pcibios_write_config_byte(pci_bus, pci_device_fn,
PCI_LATENCY_TIMER, 32);
} else if (speedo_debug > 1)
    printk(" PCI latency timer (CFLT) is %#x.%Wn", pci_latency);
/* 하드웨어에 의존적인 카드 찾기 */
}
}

```

코드 414. PCI 카드의 검출 – continued

이젠 기타 등등의 설정이 남았다. 직접적으로 PCI 디바이스의 configuration 메모리 영역에 대해서 읽기와 쓰기를 한다. 해주는 일은 BUS master와 latency 설정의 설정이다(pci_write_config_word()). 이것을 끝내면 PCI 디바이스를 사용하기 위한 기본적인 설정이 끝났다. 물론 이것은 디바이스에 의존적이며, BUS master로 동작하지 않는 경우에는 해줄 필요가 없다.

리눅스 커널 버전 2.4.X에서는 이러한 방법을 좀더 자동적으로 해주려는 노력이 엿보인다. 가령, pci_driver구조체를 정의해서 위에서 해주었던 일과 비슷한 일을 하는 것을 등록시키고, 커널에서 직접적으로 찾는 일을 해주도록 만들고 있다. 가령 예를 들어서 커널 버전 2.4.0의 ~/drivers/net/eepro100.c를 보면 아래와 같은 것을 찾을 수 있을 것이다.

```

static struct pci_device_id eepro100_pci_tbl[] __devinitdata = {
{ PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82557,
  PCI_ANY_ID, PCI_ANY_ID, },

```

```

{ PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82559ER,
    PCI_ANY_ID, PCI_ANY_ID, },
{ PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_ID1029,
    PCI_ANY_ID, PCI_ANY_ID, },
{ PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_ID1030,
    PCI_ANY_ID, PCI_ANY_ID, },
{ PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82820FW_4,
    PCI_ANY_ID, PCI_ANY_ID, },
{ 0, }
};

MODULE_DEVICE_TABLE(pci, eepro100_pci_tbl);
static struct pci_driver eepro100_driver = {
    name: "eepro100",
    id_table: eepro100_pci_tbl,
    probe: eepro100_init_one,
    remove: eepro100_remove_one,
#endif CONFIG_EEPROM100_PM
    suspend: eepro100_suspend,
    resume: eepro100_resume,
#endif
};

```

코드 415. eepro100.c에서 PCI 드라이버 등록하기

즉, 등록하려는 PCI 디바이스가 가지는 vendor ID와 device ID등을 정의한 pci_device_id를 만들고, 이를 pci_driver구조체의 id_table에 놓는다. 테이블에 들어갈 디바이스의 name은 eepro100을, 검침(probe) 함수는 eepro100_init_one, 제거(remove) 함수는 eepro100_remove_one을 둔다. 만약 전원 관리(power management)까지 하고 싶다면, suspend에 eepro100_suspend를 resume에는 eepro100_resume을 선언한다. 이렇게 하고 아래와 같이 호출한다¹⁴⁸.

```

#ifndef LINUX_VERSION_CODE < KERNEL_VERSION(2,3,48)
static int pci_module_init(struct pci_driver *pdev)
{
    int rc;

    rc = pci_register_driver(pdev);
    if (rc <= 0) {
        printk(KERN_INFO "%s: No cards found, driver not installed.\n",
               pdev->name);
        pci_unregister_driver(pdev);
        return -ENODEV;
    }
    return 0;
}
#endif
static int __init eepro100_init_module(void)
{
    if (debug >= 0 && speedo_debug != debug)
        printk(KERN_INFO "eepro100.c: Debug level is %d.\n", debug);
    if (debug >= 0)
        speedo_debug = debug;

    return pci_module_init(&eepro100_driver);
}

```

¹⁴⁸ 여기서 선언된 부분은 __devinitdata의 섹션을 차지하게 된다. 즉, 초기화시에 쓰일 데이터에 PCI ID들의 테이블 선언이 들어간다.

```
static void __exit eepro100_cleanup_module(void)
{
    pci_unregister_driver(&eepro100_driver);
}
module_init(eepro100_init_module);
module_exit(eepro100_cleanup_module);
```

코드 416. PCI 드라이버의 등록과 해지 예(eepro100.c)

모듈에 대해서는 아직 보지 않았지만, module_init()와 module_exit()가 모듈의 적재와 해지 함수를 표시한다. 적재는 eepro100_init_module()이, 해지는 eepro100_cleanup_module()이 해준다. 다시 eepro100_init_module()함수를 보면 pci_module_init()를 호출 하는 부분을 찾을 수 있을 것이다. 이 함수에서 PCI 디바이스 드라이버로 등록이 일어나게 되며, 해당하는 함수가 pci_register_driver()¹⁴⁹라는 것을 알수 있을 것이다. 따라서, 우린 pci_driver 구조체를 만드는 것에만 전념하면 된다. 나머지는 커널이 해줄 일이다.

따라서, 위에서 정의한 pci_driver 구조체의 probe()함수가 디바이스 드라이버가 사용하게 되는 디바이스를 찾는 함수가 될 것이다. eepro100_init_one() 함수가 해당될 것이다. 여기서 이 코드를 다 보기보다는 이 함수가 하는 일만 간단히 요약한다면, 먼저 PCI 디바이스를 위한 메모리 영역을 할당 받고, 이를 예약한다(request_region(), request_mem_region()), 이젠 PCI 드라이버가 이 영역을 사용한다고 나타내주고(pci_resource_start()), 만약 주소 공간의 remapping이 필요하다면 ioremap()함수를 호출한다. 전원 관리를 지원하는지 확인하고(pci_find_capability()), PCI 디바이스를 사용가능하게 만든다(pci_enable_device()). 마지막으로 함수가 복귀하기 전에 pci_set_master()을 호출해서 PCI master로 만든다.

보면 알수 있듯이 2.2.X버전의 커널보다 2.4.X버전의 커널은 디바이스 드라이버의 구현에서 좀더 디바이스 드라이버의 본연의 임부만을 강조하도록 바뀌었음을 확인할 수 있다.

6.2.2. I/O method

Device register들에 대한 접근 방법으로는 memory-mapped I/O와 I/O mapped I/O가 있다. Memory-mapped I/O라는 것은 memory의 일부공간을 device의 register공간으로 mapping하는 것을 의미하며, I/O mapped I/O란 특정의 port를 이용해서 device와 interface하는 경우이다. 우리의 경우에는 I/O port를 이용해서 device의 register들에 대한 access를 하게 되며, 이 경우에 사용하게 되는 inline function으로는 다음과 같은 것들이 있다.

- unsigned inb(unsigned port);
- void outb(unsigned char byte, unsigned port);
- unsigned inw(unsigned port);
- void outw(unsigned short word, unsigned port);
- unsigned inl(unsigned port);
- void outl(unsigned doubleword, unsigned port);

여기서 b는 byte단위의 연산에, w는 word단위, 그리고 l은 double word(32 bit)단위의 연산에 사용한다. 이러한 function들은 processor에 dependent하다는 것을 유념하고 사용하도록 하자. 즉, device driver가 사용되는 platform에 따라서 위의 function들은 제대로 작동하지 않을 수도 있다는 것을 고려해야 한다.

6.3. PCI(Peripheral Component Interface) Bus

PCI system은 ISA와 그 이전의 많은 bus system에서 장점만을 취합해서 만들어 졌으며, 고속의 데이터 전송 및 장치의 PNP(Plug and Play)기능을 지원하고 있다. ~/drivers/pci에 pci에서 pci와 관련된 code를 찾아볼 수 있다. PCI를 bus를 나타내는 구조체부터 살펴보기로 하자. ~/include/linux/pci.h를 참조하라.

¹⁴⁹ ~/drivers/pci/pci.c에 정의되어 있다.

```

struct pci_bus {
    struct list_head node;          /* 버스 리스트의 노드 */
    struct pci_bus *parent;         /* PCI 버스가 맞물린 parent버스 */
    struct list_head children;      /* 버스의 child 버스의 리스트 */
    struct list_head devices;       /* 이 버스에 맞물린 디바이스의 리스트 */
    struct pci_dev *self;           /* 부모에게 보여지는 PCI bridge 디바이스 */
    struct resource *resource[4];   /* 이 버스로 보내지게 되도록 설정된 주소공간 */
    struct pci_ops *ops;            /* 제어 접근 함수들에 대한 포인터 */
    void *sysdata; /* 특정 시스템에 대한 데이터 저장공간 */
    struct proc_dir_entry *proaddir; /* /proc파일 시스템에 대한 entry */
    unsigned char number;           /* 버스의 번호 */
    unsigned char primary;          /* 주(primary) bridge의 수 */
    unsigned char secondary;        /* 이차(secondary) bridge의 수 */
    unsigned char subordinate;      /* 최대 하위 버스의 수 */
    char name[48];                /* 디바이스의 이름 */
    unsigned short vendor;          /* 디바이스의 공급자(vendor)명 */
    unsigned short device;          /* 디바이스 ID */
    unsigned int serial;            /* 시리얼 번호 */
    unsigned char pnpver;           /* 플러그 & 플레이 버전 번호 */
    unsigned char productver;       /* 생산 버전 번호 */
    unsigned char checksum;         /* checksum이 있는지를 표시 0 - checksum passed */
    unsigned char pad1;             /* Padding */
};

```

코드 417. PCI bus구조체의 정의

PCI 버스 구조체의 정의중에서 `pci_ops`는 다시 아래와 같이 정의 된다. 이것은 특정 시스템에 의존적인 연산이다.

```

struct pci_ops {
    int (*read_byte)(struct pci_dev *, int where, u8 *val);
    int (*read_word)(struct pci_dev *, int where, u16 *val);
    int (*read_dword)(struct pci_dev *, int where, u32 *val);
    int (*write_byte)(struct pci_dev *, int where, u8 val);
    int (*write_word)(struct pci_dev *, int where, u16 val);
    int (*write_dword)(struct pci_dev *, int where, u32 val);
};

```

코드 418. PCI 연산구조체의 정의

이 함수들은 해당하는 PCI device로 부터 byte, word, double word를 읽거나 쓰기 위한 연산이다. 주로 configuration 주소 공간에 대한 read나 write연산에 사용한다.

PCI 버스상의 한 디바이스는 PCI device 구조체로 나타내지며, 아래와 같이 정의된다.

```

struct pci_dev {
    struct list_head global_list; /* 모든 PCI 디바이스에서의 리스트 */
    struct list_head bus_list;    /* 특정 PCI bus에서의 리스트 */
    struct pci_bus *bus;          /* 디바이스가 맞물린 버스의 포인터 */
    struct pci_bus *subordinate;  /* 이 디바이스가 bridge로서의 역할을 할때 하위의 버스*/
    void *sysdata; /* 특정 시스템을 위한 데이터 포인터 */
    struct proc_dir_entry *procent; /* /proc 파일 시스템의 entry */
    unsigned int devfn;           /* 기호화된(encoded) 디바이스나 function의 인덱스 값 */
    unsigned short vendor;         /* 제품 공급자 명 */
};

```

```

unsigned short device;          /* 디바이스 ID */
unsigned short subsystem_vendor; /* Subvendor 명 */
unsigned short subsystem_device; /* Subdevice ID */
unsigned int class;           /* 디바이스 클래스 : 3 bytes(base,sub,prog-if) */
u8      hdr_type;            /* PCI헤더 타입 */
u8      rom_base_reg;         /* Config 레지스터의 base주소 */
struct pci_driver *driver;    /* 드라이버를 가르키는 포인터 */
void     *driver_data;        /* 드라이버에서 사용할 데이터에 대한 포인터 */
dma_addr_t dma_mask;         /* 버스 주소에 대한 bit mask */

/* 다음의 공급자와 디바이스 ID에 대해서 호환성을 가진다.*/
unsigned short vendor_compatible[DEVICE_COUNT_COMPATIBLE];
unsigned short device_compatible[DEVICE_COUNT_COMPATIBLE];
unsigned int    irq;           /* 할당된 interrupt번호 */
struct resource resource[DEVICE_COUNT_RESOURCE]; /* I/O와 확장(expansion) ROM을 위한 메모리 공간*/
struct resource dma_resource[DEVICE_COUNT_DMA]; /* DMA(Direct Memory Access)를 위한 메모리 공간 */
struct resource irq_resource[DEVICE_COUNT_IRQ]; /* 디바이스의 Interrupt를 위한 메모리 공간 */
char      name[80];           /* 디바이스의 이름 */
char      slot_name[8];        /* 설치된 slot의 이름 */
int      active;              /* ISAPnP: 디바이스가 사용중이다.*/
int      ro;                  /* ISAPnP: 디바이스가 Read Only로 동작한다.*/
unsigned short regs;          /* ISAPnP: 제공되는 레지스터들 */
int (*prepare)(struct pci_dev *dev); /* ISAPnP를 위한 함수 포인터들 */
int (*activate)(struct pci_dev *dev);
int (*deactivate)(struct pci_dev *dev);
};

};
```

코드 419. PCI device 구조체의 정의

이전 PCI 버스와 디바이스의 구조체를 보았으니, 이를 사용하기 위한 드라이버 구조체를 볼 차례이다. PCI driver 구조체는 아래와 같이 정의된다.

```

struct pci_driver {
    struct list_head node;          /* PCI driver 리스트 */
    char *name;                    /* driver의 이름 */
    const struct pci_device_id *id_table; /* PCI 디바이스의 ID 테이블 */
    int (*probe)(struct pci_dev *dev, const struct pci_device_id *id); /* PCI 디바이스 probe 함수*/
    void (*remove)(struct pci_dev *dev); /* PCI 디바이스 제거 함수(NULL- not hot pluggable */
    void (*suspend)(struct pci_dev *dev); /* PCI 디바이스 suspend함수 : 잠시 멈춤 */
    void (*resume)(struct pci_dev *dev); /* PCI 디바이스 resume함수 : 계속 진행 */
};


```

코드 420. PCI driver 구조체의 정의

전체적인 PCI 구조는 [그림49]와 같다. PCI bus는 시스템 상에 여러개가 존재할 수 있으며, 각각은 bridge로 연결된다. 또한 bus상에는 여러 PCI 디바이스가 있을 수 있으며 각각은 해당하는 디바이스 드라이버를 가진고 있다.

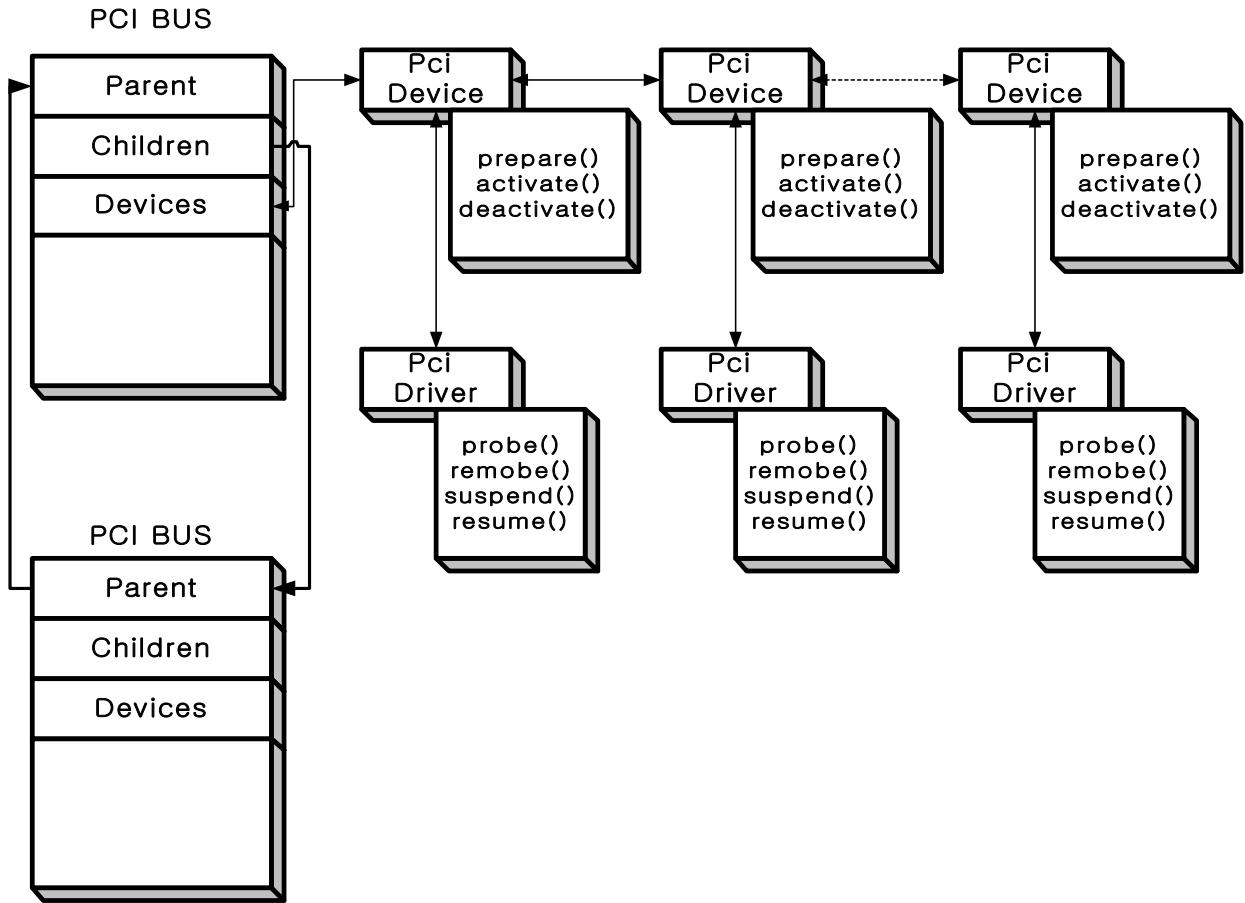


그림 56. PCI의 전체 구조

[그림49]에서 알 수 있듯이 pci_driver는 driver를 device와 관련 짓기 위해서 사용된다. Driver를 표시하는 name field와 PCI device의 configuration 공간에 들어가 있는 ID들에 대한 table¹⁵⁰, driver 자신이 제어할 수 있는 device를 찾는 probe routine에 대한 pointer와 remove, suspend, resume을 수행하는 routine에 대한 pointer들이 들어간다. 이러한 routine들은 나중에 PCI device에 대한 연산이 주어진 경우, kernel에서 해당하는 function을 불러주게 된다. 이중에서 probe routine을 살펴보면, 이는 새로운 PCI device 있는지를 검색하는데 사용되는데, 이 routine에서 해주어야 하는 일은 I/O를 위해서 사용될 resource에 대한 예약과 device가 사용할 memory공간에 대한 예약, 그리고, 할당된 IRQ를 알아내는 일이 있다. 또한 device가 Power management¹⁵¹을 지원하는지를 알아보아야 하며, PCI master¹⁵²로서 동작하는지도 확인해야 한다.

PCI에 대한 초기화는 ~/init/main.c의 start_kernel()에서 do_basic_setup()함수내에서 CONFIG_PCI가 설정된 경우에 pci_init()를 호출하면서 시작된다. 아래와 같다.

```

void __init pci_init(void)
{
    struct pci_dev *dev;
    pcibios_init();
}

```

¹⁵⁰ 이것은 driver가 제어할 수 있는 PCI device들에는 어떤 것들이 있나를 표시하게 된다.

¹⁵¹ Power Management 는 Power Management Network Device Class Reference Specification로 정해져 있으니 참고하기 바란다.

¹⁵² Device가 master로 동작한다는 말은 bus를 스스로가 구동할 수 있다는 말이다. 즉, data를 전송하기 위해서 device는 bus에 특정 주소를 가리키는 signal을 놓게 되고, bus의 cycle을 구동 시킬 수 있다.

```

pci_for_each_dev(dev) {
    pci_fixup_device(PCI_FIXUP_FINAL, dev);
}

#endif CONFIG_PM
pm_register(PM_PCI_DEV, 0, pci_pm_callback);
#endif
}

```

코드 421. pci_init()함수

여기서는 PCI bios에 대한 초기화와 각각의 PCI 디바이스에 대한 초기화가 이루어진다. 먼저 PCI bios에 대한 초기화부터 보도록 하자.

```

void __init pcibios_init(void)
{
    struct pci_ops *bios = NULL;
    struct pci_ops *dir = NULL;

#ifndef CONFIG_PCI BIOS
    if ((pci_probe & PCI_PROBE BIOS) && ((bios = pci_find_bios())))
    {
        pci_probe |= PCI BIOS_SORT;
        pci_bios_present = 1;
    }
#endif
#ifndef CONFIG_PCI DIRECT
    if (pci_probe & (PCI_PROBE_CONF1 | PCI_PROBE_CONF2))
        dir = pci_check_direct();
#endif
    if (dir)
        pci_root_ops = dir;
    else if (bios)
        pci_root_ops = bios;
    else {
        printk("PCI: No PCI bus detected\n");
        return;
    }
    printk("PCI: Probing PCI hardware\n");
    pci_root_bus = pci_scan_bus(0, pci_root_ops, NULL);
    pcibios_irq_init();
    pcibios_fixup_peer_bridges();
    pcibios_fixup_irqs();
    pcibios_resource_survey();
#endif CONFIG_PCI BIOS
    if ((pci_probe & PCI BIOS_SORT) && !(pci_probe & PCI_NO_SORT))
        pcibios_sort();
#endif
}

```

코드 422. PCI bios의 초기화(pcibios_init()함수)

pcibios_init()함수는 ~/arch/i386/pci-pc.c에 정의되어 있다. 이 함수가 하는 일은 pci_probe에 설정된 사항을 가지고, 해당하는 PCI버스에 대한 연산(pci_ops)를 초기화하고, PCI root bus(pci_root_bus)를 만드는 일이다. 만약 PCI가 디바이스에 대한 직접접근(direct access)가 가능하다면, dir연산으로 pci_root_ops를 초기화하고, PCI bios가 존재하고, 직접접근이 가능하지 않다면, bios로 pci_root_ops를 초기화 한다.

pci_scan_bus()함수는 모든 PCI버스에 대한 초기화를 담당하며, pcibios_irq_init()함수는 인터럽트에 대한 초기화를, pcibios_fixup_peer_bridges()함수는 나머지 PCI bridge들에 대한 검색과 초기화를 담당하며,

`pcibios_fixup_irqs()`함수는 현재 설정된 PCI 디바이스에 대한 인터럽트 번호를 재 설정하게되며, `pcibios_resource_survey()`함수는 커널 자원(resource)의 할당을 맡고 있다.

6.4. DMA(Direct Memory Access)

DMA(Direct Memory Access)란 말 그대로 device가 CPU의 개입 없이 직접적으로 memory에 접근을 하겠다는 것이다. DMA를 사용한 방법은 CPU의 개입 없이 작동하기에 대용량의 데이터를 전송하는데 많은 이점이 있다. 이것을 사용하는 device는 Bus의 master가 되어서 data의 전송을 시작한다. 다음과 같은 것에 대한 이해가 필요하다.

Frame buffer: memory의 일부로 DMA에서 사용하게 되는 memory이다. 이것은 chain으로 연결되어 DMA controller가 그 chain을 따라서 이동하며 memory상의 데이터를 device쪽으로 전송하거나, 혹은 device의 data를 memory로 전송하게 된다. 이러한 frame들은 memory에 연속(contiguous)하게 나와야 하는 device들도 있으며, 그렇지 않고 메모리에 산재해 있더라도 가능한 device가 있다. 이것은 device를 디자인한 사람의 몫이다. [그림50]은 DMA의 구조를 간단히 보여준다.

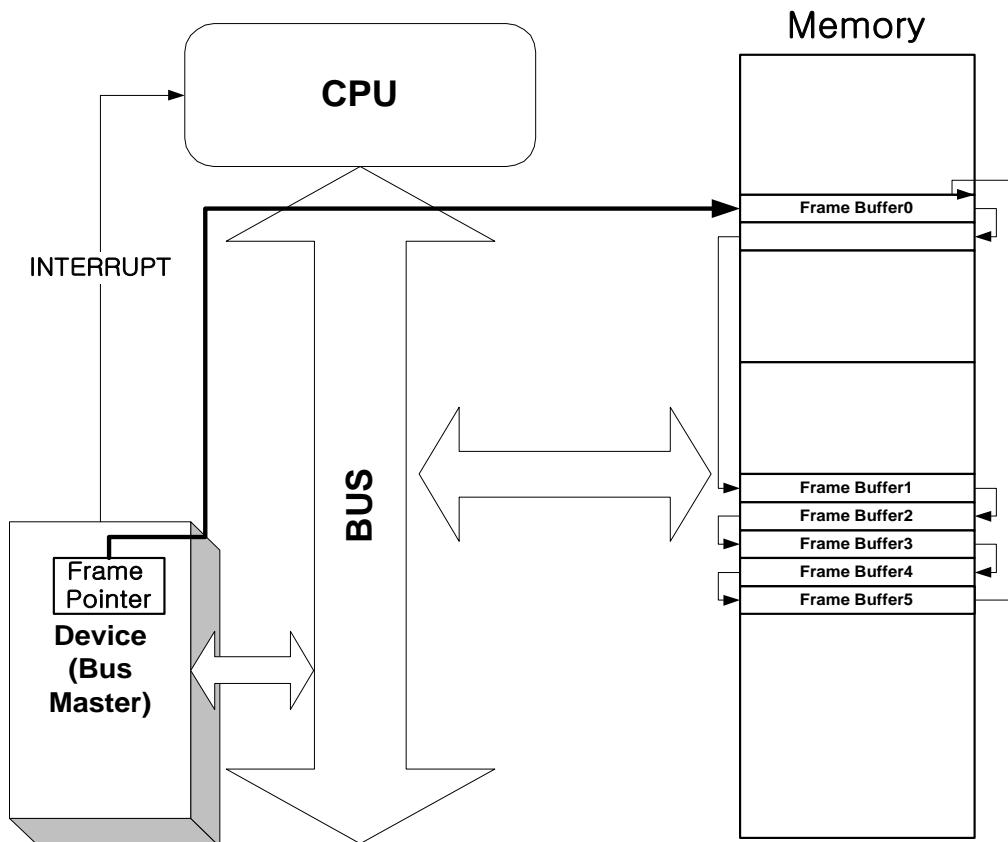


그림 57. DMA의 구조

[그림50]에서 보듯이 device의 한 register는 memory의 physical주소공간의 한 부분을 가르키게 되며, 전송하고자 하는 data들은 memory상에서 전송을 기다리게 된다. 이때, device의 전송을 담당한 register를 setting함으로써, Bus상 request를 보내게 되며, data의 전송은 CPU의 개입 없이 device와 memory간에 일어나게 된다. 데이터의 전송이 끝나게 되면, 다시 Interrupt가 발생해서 CPU에게 전송이 다 마쳤음을 알리게 되는 것이다.

DMA buffer를 묘사하는 field들이 있으며, DMA의 결과와 어디까지의 memory에, 이동시키고자 하는 data가 있는지를 DMA controller에게 알려주게 된다. Tx와 Rx각각에 이러한 정보를 나타내는 field가

있으며, 이것을 위해서 device의 manual에 반드시 지켜 주어야 할 규칙을 명시하고 있다.¹⁵³ 이러한 정보는 device driver에서 setting해주어야 하는 경우가 있으며, 자동으로 DMA가 일어났을 경우 setting이 되는 것들이 있다.

6.5. Kernel Module Programming

커널 모듈은 동적으로 커널과 결합해서 사용하는 커널 모드의 프로그램이다. 이 같은 모듈을 사용하는 것은 커널을 기동중에 동적으로 바꾸는 것을 가능하게 하며, 드라이버를 개발하는 일도 편리하게 한다. 하지만 잘 못된 코드를 사용하는 것은 커널에 심각한 오류를 발생시킬 수 있으므로 많은 주의를 요한다. 또한 필요시에만 모듈을 적재(load)하고, 필요가 없어지면 모듈을 내려(unload) 수 있기 때문에, 커널이 차지하는 메모리 공간을 줄이는 효과도 있다. 하지만 커널과 링크될 필요가 있기에, 초기에 실행되는데 시간이 걸린다.

커널에서 제공하는 모든 함수를 사용할 수 있으며(즉, 커널 모드로 동작한다.), 다른 모듈을 불러드릴 수도 있다. 모듈은 사용 카운터(counter)를 가지고 있다. 사용되면, 카운터가 증가하며, 사용을 마치면 카운터는 감소한다. 만약 모든 사용을 마친다면, 모듈은 자동적으로 내려(unload)질 것이며, 메모리의 낭비를 줄인다.

예전 버전(version 2.0.X)에서는 커널의 모듈을 올리고 내리는 helper가 사용자 주소공간을 사용하고 있었으나, 그 이후의 버전에서는 이러한 것이 모두 커널 내에 구현되어 있다. 모듈은 실행가능 파일로 파일 시스템에 저장되며, ELF형식으로 되어있다. 커널은 모듈이 가진 전역 심벌(symbol)들을 접근해서, 다른 모듈에서 사용할 수 있도록 해주어야하며, 위에서 설명한 사용 카운터를 관리하는 역할을 맡고 있다.

```
struct module_symbol
{
    unsigned long value;          /* 모듈내의 심벌의 값 */
    const char *name;             /* 모듈내의 심벌의 이름 */
};

struct module_ref
{
    struct module *dep; /* parent : 다른 모듈에 의해서 현재 모듈이 사용 때, 해당 모듈에 대한 연결 */
    struct module *ref; /* child: 현재 모듈이 다른 모듈을 언급할 때 사용하는 모듈에 대한 연결 */
    struct module_ref *next_ref; /* 다음번 연결 */
};
```

코드 423. module_symbol구조체와 module_ref구조체의 정의

모듈의 심벌과 모듈의 레퍼런스는 각각 모듈에서 정의하는 심벌들과 다른 모듈들에 대한 관계를 설정한다. 이것은 아래에 나오는 모듈 구조체에서 사용된다.

커널은 메모리로 적재되는 모듈들에 대해서 module구조체를 유지한다. 모듈 구조체 정의를 보도록 하자. 아래와 같다.

```
struct module
{
    unsigned long size_of_struct; /* 모듈 구조체의 크기 */
    struct module *next;           /* 다음 모듈구조체에 대한 포인터 */
    const char *name;              /* 모듈의 이름 */
    unsigned long size;            /* 모듈의 크기(모듈구조체의 크기는 size_of_struct가 나타냄.) */
    union
    {
```

¹⁵³ 이러한 것으로는 byte alignment와 같은 것이 있을 수 있다. 이것이 지켜지지 않으면, 올바른 DMA operation이 일어나지 못한다.

```

atomic_t usecount;
long pad;
} uc;                      /* Needs to keep its size - so says rth */
unsigned long flags;        /* 모듈의 플랙 */
unsigned nsyms;             /* Export된 심벌(symbol)의 수 */
unsigned ndeps;              /* 의존관계를 가지는 모듈의 수- 언급(reference)되는 모듈의 갯수 */
struct module_symbol *syms; /* Export되는 심벌의 테이블 */
struct module_ref *deps;    /* 언급되는 모듈의 리스트 */
struct module_ref *refs;    /* 언급하는 모듈의 리스트 */
int (*init)(void);          /* 초기화 함수의 포인터 */
void (*cleanup)(void);      /* Cleanup 함수의 포인터 */
const struct exception_table_entry *ex_table_start; /* 예외(exception) 테이블의 시작 */
const struct exception_table_entry *ex_table_end; /* 예외 테이블의 끝 */
#endif __alpha__
unsigned long gp;
#endif
/* 이하의 부분은 option으로 확장 부분이다.*/
const struct module_persist *persist_start;
const struct module_persist *persist_end;
int (*can_unload)(void);
int runsize;                /* 사용되지 않음 */
const char *kallsyms_start; /* 커널 비버깅(debugging)을 위한 심벌 */
const char *kallsyms_end;
const char *archdata_start; /* CPU에 의존적인 데이터 */
const char *archdata_end;
const char *kernel_data;    /* 커널이 내부적으로 사용할 목적으로 예약한 영역 */
};

}

```

코드 424. 모듈 구조체의 정의

모듈 구조체의 정의를 보면 `init()`함수에 대한 포인터와 `cleanup()`함수에 대한 포인터가 각각 정의되어 있음을 볼 수 있다. 이들 함수의 포인터는 모듈이 커널과 링크되는 동안에 모듈의 초기화와 모듈의 제거시에 필요한 연산을 수행하기 위한 함수의 포인터이다. 아래에서 보게될 `insmod`를 실행하게 될 때, `sys_init_module()`이 호출되며, 이 시스템 콜 내에서 `init()`함수가 수행될 것이다. `cleanup()`함수는 `sys_delete_module()`에서 보게 될 것이다.

모듈을 커널과 링크 시키기 위해서 사용하는 명령어는 `insmod`이다. 다시 커널에서 모듈을 제거하는 것은 `rmmmod`이다. 이들 명령어를 실행시키게 될 때 어떠한 일이 일어나는지를 살펴보도록 하자.

먼저 `insmod`를 호출하면, 시스템 콜인 `sys_create_module()`이 호출된다. `~/kernel/module.c`를 보도록 하자. 아래와 같다.

```

asmlinkage unsigned long sys_create_module(const char *name_user, size_t size)
{
    char *name;
    long namelen, error;
    struct module *mod;

    if (!capable(CAP_SYS_MODULE))
        return -EPERM;
    lock_kernel();
    if ((namelen = get_mod_name(name_user, &name)) < 0) {
        error = namelen;
        goto err0;
    }
    if (size < sizeof(struct module)+namelen) {

```

```

        error = -EINVAL;
        goto err1;
    }
    if (find_module(name) != NULL) {
        error = -EEXIST;
        goto err1;
    }
    if ((mod = (struct module *)module_map(size)) == NULL) {
        error = -ENOMEM;
        goto err1;
    }
    memset(mod, 0, sizeof(*mod));
    mod->size_of_struct = sizeof(*mod);
    mod->next = module_list;
    mod->name = (char *)(mod + 1);
    mod->size = size;
    memcpy((char*)(mod+1), name, namelen+1);
    put_mod_name(name);
    module_list = mod; /* link it in */
    error = (long) mod;
    goto err0;
err1:
    put_mod_name(name);
err0:
    unlock_kernel();
    return error;
}

```

코드 425. sys_create_module()시스템 콜

sys_create_module()이 넘겨받는 값은 모듈의 이름과 모듈이 차지할 메모리 공간의 크기이다. 가장 먼저 사용자가 CAP_SYS_MODULE의 capability를 가진지를 확인하고, 그렇지 않다면 에러(-EPERM)를 돌려준다. 커널에 대한 lock을 설정한 후, 사용자 공간으로 부터 모듈의 이름을 가져와서 새로 할당받은 메모리 영역에 복사해 넣는다(get_mod_name()). 올바른 크기로 가지는지를 확인한 다음, 혹시 해당하는 모듈이 이미 커널에 있는지를 확인한다(find_module()). 있다면 에러(-EEXIST)를 돌려준다. 이젠 모듈 크기만큼에 해당하는 메모리를 할당받고(module_map()), 만약 메모리가 없다면 에러(-ENOMEM)를 돌려준다.

할당받은 메모리를 0으로 초기화 시키고, 모듈 구조체의 크기와 다음 번 모듈에 대한 포인터(module_list) 및 모듈의 이름을 초기화 시키고, 다시 모듈의 크기를 초기화 시킨다. 이상의 과정이 끝나면, 모듈의 이름을 임시 저장을 위해서 할당받은 메모리를 반환하고(put_mod_name()), module_list 전역변수를 현재 적재된 모듈로 가리키게 한후 커널에 대한 lock을 해제하고 복귀한다. 여기서 module_list는 현재 커널에 적재된 모든 모듈의 module구조체 리스트를 가진다.

*rmmmod*명령은 결과적으로 delete_module()함수를 호출하고, 시스템 콜인 sys_delete_module()함수가 최종적으로 호출된다. 정의는 아래와 같다.

```

asmlinkage long sys_delete_module(const char *name_user)
{
    struct module *mod, *next;
    char *name;
    long error;
    int something_changed;

    if (!capable(CAP_SYS_MODULE))
        return -EPERM;
    lock_kernel();

```

```

if (name_user) {
    if ((error = get_mod_name(name_user, &name)) < 0)
        goto out;
    if (error == 0) {
        error = -EINVAL;
        put_mod_name(name);
        goto out;
    }
    error = -ENOENT;
    if ((mod = find_module(name)) == NULL) {
        put_mod_name(name);
        goto out;
    }
    put_mod_name(name);
    error = -EBUSY;
    if (mod->refs != NULL)
        goto out;
    spin_lock(&unload_lock);
    if (!__MOD_IN_USE(mod)) {
        mod->flags |= MOD_DELETED;
        spin_unlock(&unload_lock);
        free_module(mod, 0);
        error = 0;
    } else {
        spin_unlock(&unload_lock);
    }
    goto out;
}

```

코드 426. sys_delete_module() 시스템 콜

sys_delete_module() 시스템 콜은 궁극적으로 적재된 모듈을 커널에서 제거하는 일을 한다. 이 시스템 콜이 넘겨 받는 것은 모듈의 이름이다. 먼저 사용자가 CAP_SYS_MODULE capability가 있는지 확인하고, 그렇지 않다면 에러(-EPERM)를 돌려준다. 다시 커널에 대한 lock을 설정한 후, 사용자 공간에서 이름을 제공하고 있다면 아래와 같은 일을 한다.

모듈의 이름을 사용자 주소공간에서 가져온다(get_mod_name()). 이때 에러가 있다면 할당받은 공간을 해제하고(put_mod_name()), 에러값으로 -EINVAL을 돌려준다. 에러가 없다면, 이름을 가지고 해당하는 모듈을 찾는다(find_module()). 역시 에러가 있다면 put_mod_name()을 호출해서 모듈의 이름을 위해서 할당받은 공간을 해제하고 에러로는 -ENOENT를 돌려준다. 해당하는 모듈을 찾았다면, 이전 모듈이 언급(reference)하는 모듈이 있는지 확인한다. 만약 NULL이 아니라면 아직 사용하고 있는 모듈이 있다는 말이 되므로 여기서 즉시 복귀한다. 없다면 모듈의 사용 카운터를 확인한다(__MOD_IN_USE()). 사용 카운터가 0이라면 모듈이 제거되었다고 나타내고(MOD_DELETE), unload_lock을 해제한 후, 모듈 구조체를 위해 할당된 메모리를 제거한다. 만약 모듈 카운터가 0이 아니라면 아직 사용되고 있으므로 unload_lock에 설정된 lock만 해제하고 나간다(out).

```

/* Do automatic reaping */
restart:
    something_changed = 0;
    for (mod = module_list; mod != &kernel_module; mod = next) {
        next = mod->next;
        spin_lock(&unload_lock);
        if (mod->refs == NULL
            && (mod->flags & MOD_AUTOCLEAN)
            && (mod->flags & MOD_RUNNING)
            && !(mod->flags & MOD_DELETED))

```

```

    && (mod->flags & MOD_USED_ONCE)
    && !_MOD_IN_USE(mod)) {
        if ((mod->flags & MOD_VISITED)
            && !(mod->flags & MOD JUST_FREED)) {
            spin_unlock(&unload_lock);
            mod->flags &= ~MOD_VISITED;
        } else {
            mod->flags |= MOD_DELETED;
            spin_unlock(&unload_lock);
            free_module(mod, 1);
            something_changed = 1;
        }
    } else {
        spin_unlock(&unload_lock);
    }
}
if (something_changed)
    goto restart;
for (mod = module_list; mod != &kernel_module; mod = mod->next)
    mod->flags &= ~MOD JUST_FREED;
error = 0;
out:
unlock_kernel();
return error;
}

```

코드 427. sys_delete_module() 시스템 콜 - continued

모듈의 이름으로 주어진 값이 NULL인 경우에 해당하는 부분이다. module_list를 검사해서, 모듈이 언급하는 것이 없고(mod->refs==NULL), 모듈의 플랙이 MOD_AUTOCLAN, MOD_RUNNING, ~(MOD_DELETED), MOD_USED_ONCE가 설정되어 있으며, 사용 카운터가 0인 모듈들에 대해서, 방문되지 않았으며, 방금 해제(MOD JUST_FREED)되지 않은 경우에는 모듈 구조체의 flag를 MOD_DELETED를 설정한 후 메모리에서 제거하고(free_module()), 그렇지 않다면, unload_lock을 푼 후, 모듈의 플랙을 방문(MOD_VISITED)되지 않았다고 표시한다. 위와 같은 연산을 마쳤다면, 다시 module_list를 검색해서 모듈들을 전부 MOD JUST_FREED 플랙 설정을 지운다. 연산을 마치면 커널의 lock을 해제하고, 설정된 에러를 돌려주고 복귀한다. 이상에서 모듈의 flag부분이 가질 수 있는 값은 아래와 같다.

Flags	Description
MOD_UNINITIALIZED(0)	모듈이 초기화 되지 않았다.
MOD_RUNNING(1)	모듈이 사용되고 있다.
MOD_DELETED(2)	모듈이 제거되었다.
MOD_AUTOCLAN(4)	모듈에 대한 자동제거가 설정되었다.
MOD_VISITED(8)	모듈이 이미 방문되었다.
MOD_USED_ONCE(16)	모듈이 한번 사용되었다.
MOD JUST_FREED(32)	모듈이 방금 해제되었다.
MOD_INITIALIZING(64)	모듈이 초기화 되고 있다.

표 44. 모듈 구조체의 flag값

여기서 한가지 free_module()을 좀더 보도록 하자. 이곳에서 모듈에 대해서 정의된 cleanup()함수가 호출된다. 아래와 같다.

```

void
free_module(struct module *mod, int tag_freed)

```

```
{
    struct module_ref *dep;
    unsigned i;

    /* Let the module clean up. */
    if (mod->flags & MOD_RUNNING)
    {
        if(mod->cleanup)
            mod->cleanup();
        mod->flags &= ~MOD_RUNNING;
    }
    /* Remove the module from the dependency lists. */
    for (i = 0, dep = mod->deps; i < mod->ndeps; ++i, ++dep) {
        struct module_ref **pp;
        for (pp = &dep->dep->refs; *pp != dep; pp = &(*pp)->next_ref)
            continue;
        *pp = dep->next_ref;
        if (tag_freed && dep->dep->refs == NULL)
            dep->dep->flags |= MOD JUST_FREED;
    }
    /* And from the main module list. */
    if (mod == module_list) {
        module_list = mod->next;
    } else {
        struct module *p;
        for (p = module_list; p->next != mod; p = p->next)
            continue;
        p->next = mod->next;
    }
    /* And free the memory. */
    module_unmap(mod);
}
}
```

코드 428. free_module()함수

모듈이 활성(MOD_RUNNING)중이 아닌지를 확인한 후, 활성중이라면 모듈에 정의된 cleanup()함수를 호출하고, 모듈을 비활성(~MOD_RUNNING)으로 만든다. 그리고 난후 모듈의 의존성을 검사해서 의존하고 있는 모듈들로부터 현재의 모듈을 제거한다(MOD JUST_FREED). 만약 제거하려는 모듈이 module_list값과 일치한다면, module_list값을 다음번의 모듈을 가르쳐주도록 하고, 그렇지 않다면, 해당하는 모듈을 찾아서, 연결리스트에서 제거한다. 마지막으로 모듈 구조체의 메모리를 해제한다(module_unmap()).

모듈에 대한 초기화는 sys_create_module()시스템 콜을 호출한 후, 다시 sys_init_module()을 호출한다. 이 시스템 콜의 역할은 모듈에 대한 초기화를 맡고 있으며, 아래와 같이 정의된다.

```
asmlinkage long sys_init_module(const char *name_user, struct module *mod_user)
{
    struct module mod_tmp, *mod;
    char *name, *n_name, *name_tmp = NULL;
    long namelen, n_namelen, i, error;
    unsigned long mod_user_size;
    struct module_ref *dep;

    if (!capable(CAP_SYS_MODULE))
        return -EPERM;
    lock_kernel();
    if ((namelen = get_mod_name(name_user, &name)) < 0) {
```

```

        error = namelen;
        goto err0;
    }
    if ((mod = find_module(name)) == NULL) {
        error = -ENOENT;
        goto err1;
    }
    if ((error = get_user(mod_user_size, &mod->size_of_struct)) != 0)
        goto err1;
    if (mod_user_size < (unsigned long)&((struct module *)0L)->persist_start
        || mod_user_size > sizeof(struct module) + 16*sizeof(void*)) {
        printk(KERN_ERR "init_module: Invalid module header size.\n"
               KERN_ERR "A new version of the modutils is likely "
               "needed.\n");
        error = -EINVAL;
        goto err1;
    }
}

```

코드 429. sys_init_module() 시스템 콤

먼저 사용자의 capability를 확인한다. 그리고나서 커널에 lock을 설정하고(lock_kernel()). 사용자 주소공간에서 모듈의 이름을 가져온다. 이름에 해당하는 모듈을 찾고(find_module()), 다시 사용자 공간에서 모듈의 헤더부분을 가져온다(get_user). 가져온 모듈의 헤더와 할당받아서 초기화한 모듈의 값을 비교해서 크기가 올바른지 확인하게 되며, 만약 옳지 않다면 에러(-EINVAL)를 돌려준다.

```

/* Hold the current contents while we play with the user's idea
   of righteousness. */
mod_tmp = *mod;
name_tmp = kmalloc(strlen(mod->name) + 1, GFP_KERNEL);      /* Where's kstrdup()? */
if (name_tmp == NULL) {
    error = -ENOMEM;
    goto err1;
}
strcpy(name_tmp, mod->name);
error = copy_from_user(mod, mod_user, mod_user_size);
if (error) {
    error = -EFAULT;
    goto err2;
}
/* Sanity check the size of the module. */
error = -EINVAL;
if (mod->size > mod_tmp.size) {
    printk(KERN_ERR "init_module: Size of initialized module "
           "exceeds size of created module.\n");
    goto err2;
}

```

코드 430. sys_init_module() 시스템 콤 – continued

모듈이름을 복사할 영역을 설정하고, 모듈의 이름 및 모듈의 헤더 정보를 가져온다.(copy_from_user()). 이전 카피된 모듈의 크기와 실제 모듈의 크기에 대한 검사를 해서 에러(-EINVAL)가 있는지 확인한다. 여기서 발생할 수 있는 에러의 종류에는 -ENOMEM과 -EFAULT가 있으며, 해당 에러 처리부분으로 제어를 곧바로 이동한다.

```

...
/* 모듈에 헤더 및 각종 정보에 대한 정밀 검사 */
...

```

```

/* Ok, that's about all the sanity we can stomach; copy the rest. */
if (copy_from_user((char *)mod+mod->mod_user_size,
                   (char *)mod->mod_user_size,
                   mod->size-mod->mod_user_size)) {
    error = -EFAULT;
    goto err3;
}
if (module_arch_init(mod))
    goto err3;
/* On some machines it is necessary to do something here
   to make the I and D caches consistent. */
flush_icache_range((unsigned long)mod, (unsigned long)mod + mod->size);
mod->next = mod_tmp.next;
mod->refs = NULL;
/* Sanity check the module's dependents */
for (i = 0, dep = mod->deps; i < mod->ndeps; ++i, ++dep) {
    struct module *o, *d = dep->dep;
    /* Make sure the indicated dependencies are really modules. */
    if (d == mod) {
        printk(KERN_ERR "init_module: self-referential "
               "dependency in mod->deps.\n");
        goto err3;
    }
    /* Scan the current modules for this dependency */
    for (o = module_list; o != &kernel_module && o != d; o = o->next)
        ;
    if (o != d) {
        printk(KERN_ERR "init_module: found dependency that is "
               "(no longer?) a module.\n");
        goto err3;
    }
}
/* Update module references. */
for (i = 0, dep = mod->deps; i < mod->ndeps; ++i, ++dep) {
    struct module *d = dep->dep;
    dep->ref = mod;
    dep->next_ref = d->refs;
    d->refs = dep;
    /* Being referenced by a dependent module counts as a
       use as far as kmod is concerned. */
    d->flags |= MOD_USED_ONCE;
}

```

코드 431. sys_init_module() 시스템 콜 – continued

모듈의 각종 정보에 대한 검사를 한 후, 이젠 모듈을 실제적으로 사용자 공간에서 커널 공간으로 복사하는 일이다(copy_from_user()). 예러가 있다면 -EFAULT를 설정한 후 복귀한다. 모듈에 대한 커널 공간으로의 복사가 끝나면, CPU에 의존적인 초기화가 행해진다(module_arch_init()). 이젠 명령 캐쉬(instruction cache)에 대해서 제 설정하고(flush_icache_range()), 모듈을 module_list에 넣는다.

아직 모듈에 대해서 언급(reference)하고 있는 것이 없으므로 모듈의 refs는 NULL을 가리키도록 한다. 이젠 모듈이 의존하고 있는 다른 모듈들이 제대로 되어 있는지를 확인하는 일이다. 모듈의 deps필드에 언급되는 모듈들에 대한 정보를 가지고 있으므로 이 값이 옳은지를 확인한다. 자신에 대한 언급이 없어야 하며, kernel_module에 대한 언급도 가지지 않아야 한다. kernel_module은 커널에서 사용하는 모듈로서 이것은 dummy 모듈(가짜 모듈)의 기능을 갖는 것이다.

위의 과정이 끝났다면, 모듈의 레퍼런스 카운터를 고친다. 모듈의 deps리스트를 검사해서, 언급하고 있는 모듈들이 한번은 사용되었음을 나타낸다(MOD_USED_ONCE).

```

/* Free our temporary memory. */
put_mod_name(n_name);
put_mod_name(name);
/* Initialize the module. */
mod->flags |= MOD_INITIALIZING;
atomic_set(&mod->uc.usecount,1);
if (mod->init && (error = mod->init()) != 0) {
    atomic_set(&mod->uc.usecount,0);
    mod->flags &= ~MOD_INITIALIZING;
    if (error > 0) /* Buggy module */
        error = -EBUSY;
    goto err0;
}
atomic_dec(&mod->uc.usecount);
/* And set it running. */
mod->flags = (mod->flags | MOD_RUNNING) & ~MOD_INITIALIZING;
error = 0;
goto err0;
err3:
put_mod_name(n_name);
err2:
*mod = mod_tmp;
strcpy((char *)mod->name, name_tmp); /* We know there is room for this */
err1:
put_mod_name(name);
err0:
unlock_kernel();
kfree(name_tmp);
return error;
}

```

코드 432. sys_init_module() 시스템 콜 – continued

일시적인 사용목적으로 할당했던 모듈 이름에 대한 메모리에 대해서 해제하고(put_mod_name()), 모듈에 대한 초기화를 시작하기 MOD_INITIALIZING 플랙을 설정한다. 그리고나서 사용자 카운터를 증가시켜서 모듈이 현재 사용되고 있음을 표시한 후, 모듈의 init필들에 설정된 초기화 함수를 부른다. 에러가 발생한다면, 사용카운터를 0으로 만든후 모듈의 플랙에서 MOD_INITIALIZING를 지운후 에러를 돌려준다(-EBUSY). 에러가 없었다면, 다시 증가시켰던 사용카운터를 낮추고(atomic_dec()), 모듈의 플랙을 MOD_RUNNING상태로 바꾸고 MOD_INITIALIZING을 지운다.

마지막으로 커널에 대해서 설정했던 lock을 해제하고, 모듈의 이름을 위해서 할당해던 버퍼를 해제한후 에러코드와 함께 복귀한다.

모듈에 대한 초기화를 마쳤을 경우에 가지는 전체 모듈의 구조는 [그림51]과 같다. [그림51]에서 초기에 module_list로 초기화가 되는 것은 kernel_module이며, 이것이 기준이 되어, 다른 모듈들이 삽입된다.

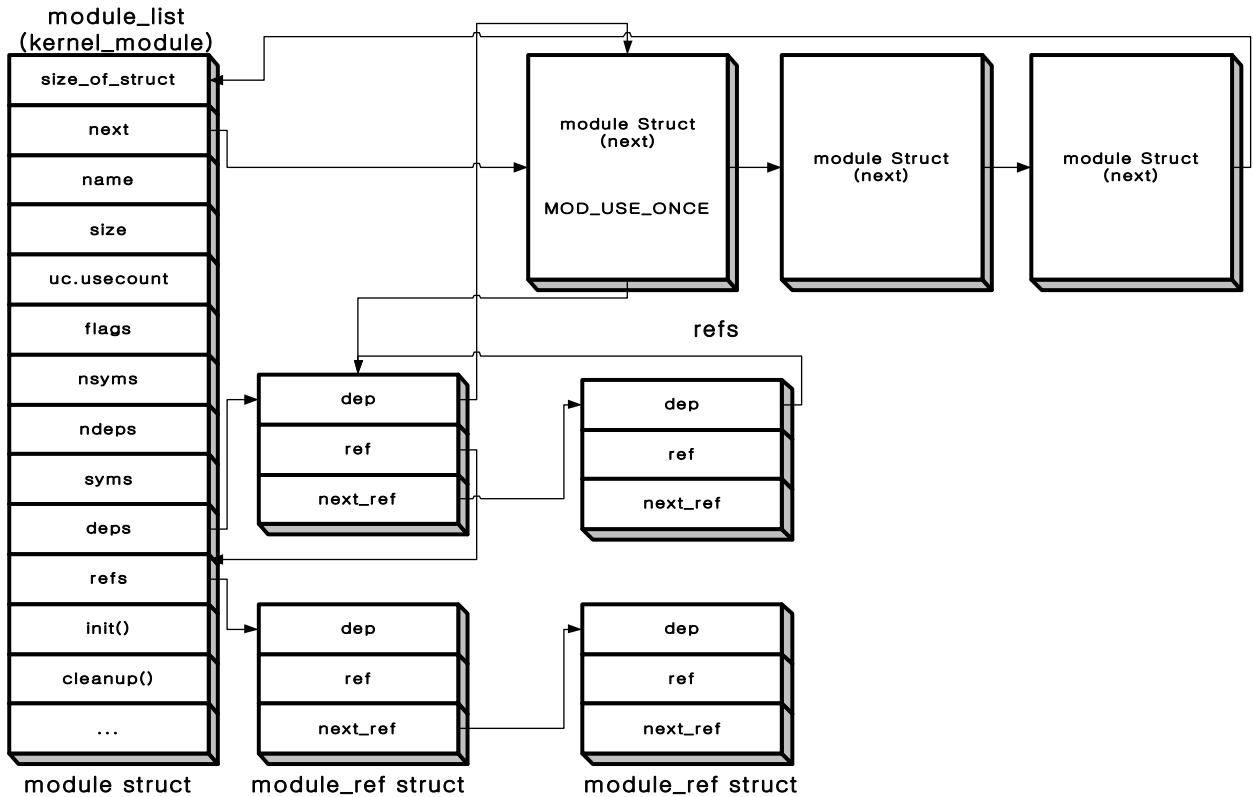


그림 58. 모듈의 전체 구조

모듈에 대한 질의는 `sys_qm_module()` 시스템 콜이 처리하며, 특정 모듈의 이름(`qm_modules()`)이나 의존성(`qm_deps()`), 언급하는 모듈(`qm_refs()`), 모듈의 심벌(`qm_symbols()`), 모듈의 정보(`qm_info()`) 등을 구할 때 사용한다. 이들 각각에 대해서 `QM_MODULES`, `QM_DEPS`, `QM_REFS`, `QM_SYMBOLS`, `QM_INFO`를 시스템 콜의 파라미터로 사용한다.

모듈에서 새로운 모듈을 요구하는 것은 `request_module()` 함수가 맡고 있다. 이것은 모듈이 다른 모듈에서 제공하는 함수를 사용하고자 할 때 있을 수 있는 일이다. 이렇게 해서 모듈 간의 reference 관계가 성립된다. `request_module()`은 다시 `exec_modprobe`라는 커널 쓰레드를 생성해서 사용자 모드의 helper를 이용해서 모듈을 불러온다. 정의는 `~/kernel/kmod.c`에 나와 있다.

6.6. Interrupt handler의 설치.

Kernel은 interrupt line들에 대한 registry를 관리한다. 따라서, module은 interrupt channel을 사용하기 위해서는 먼저 자신의 interrupt service routine을 interrupt line과 연결을 시켜야 하며, 이러한 것을 kernel은 알고 있어야 한다. 또한 device driver가 interrupt line을 더 이상 사용하지 않아도 된다면 kernel에 다시 해제 한다는 것을 알려 주어야 한다. 다음의 function이 이러한 역할을 하며, 각각의 parameter로 주어지는 값들에 대해서 자세히 살펴 보도록 하자.

1. `int request_irq(unsigned int irq, void (*handler)(int, void *, struct pt_regs *), unsigned long flags, const char *device, void *dev_id);`
 2. `void free_irq(unsigned int irq, void *dev_id);`
- `unsigned int irq` – interrupt number이다. 우리의 경우에는 이것은 PCI device들로부터 읽어 들인 값이다.
 - `void (*handler)(int, void *, struct pt_regs *)` – interrupt handler를 가리킨다. 즉, device로부터 interrupt가 발생했을 경우 그것을 받아서 처리하게 될 함수가 된다. 이러한 routine에서 먼저 해주어야 할 일은 interrupt를 발생시킨 원인을 찾는 것이며, 그에 해당한 처리를 해주는 것이다. 우리의 경우에는 transmission이나 receive가 될 수 있을 것이다.

- unsigned long flags – interrupt 관리와 연관이 있는 option의 bitmask이다. 다음과 같은 것들이 있다. – SA_INTERRUPT, SA_SHIRQ, SA_SAMPLE_RANDOM, 각각은 fast/slow interrupt handler, 공유(sharing)가 가능한지, random에 의해서 사용되는 지에 대한 것을 나타낸다. 우리의 경우에는 shared interrupt만을 setting해서 사용하고 있다.
- const char *device – interrupt의 소유자를 나타내는 문자열(string)이다.
- void *dev_id – interrupt sharing이 되고 있지 않다면 NULL로 setting하면 되지만, 만약 그렇다면 device를 나타내는 고유의 값을 넣어 주야 한다. 이 경우 device driver의 구조체 pointer를 넣어주면 된다.

이전 가장 기본적인 디바이스 드라이버를 볼 차례이다. 문자 디바이스 드라이버(character device driver)가 리눅스에서 그나마 가장 간단한 디바이스 드라이버이다.

6.7. IOCTL(I/O Control)

ioctl() 메소드는 일반적으로 정의된 것 이외에 사용자 프로세스에서 특정 디바이스 드라이버에 명령을 주어 디바이스에 의존적인 일을 수행할 수 있도록 한다. 물론 정해진 명령에 대해서는 해당되는 일을 수행해야 할 것이다. 디바이스 드라이버를 작성하며, ioctl 메소드에 대해서 어떤 식으로 반응할지를 미리 정해주어야 할 것이다.

ioctl()이 사용되는 방법은 아래와 같다. 즉, 파일에 대한 ioctl() 함수의 호출이다.

```
#include<sys/ioctl.h>

int ioctl(int fd, int request, ... )
```

이는 시스템 콜을 호출하는 것으로 첫번째 넘겨주는 파라미터는 열린 파일에 대한 파일 디스크립터이며, 두번째 넘겨주는 것은 파일 디스크립터에 대한 command,이고, 그 이하의 파라미터 값은 char*의 형태를 가진다. 따라서, 명령(command)에 따라 char* 이하의 부분이 해석을 달리 할 것이다.

이것을 처리하는 디바이스 드라이버 메소드는 아래와 같은 ioctl() 함수 부분이다.

```
int (*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);
```

여기서 넘겨 받는 파라미터는 먼저 해당 inode 객체에 대한 포인터와 파일 객체에 대한 포인터, 그리고 command를 나타내는 부분과 unsigned long으로 선언된 pointer이다. 즉, 커널에서 inode와 파일 객체에 대한 포인터를 찾아서, 해당하는 ioctl() 메소드를 호출해 준다. 시스템 콜 인터페이스는 아래와 같다. ~/fs/iotl.c를 참조하자.

```
asmlinkage long sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
    struct file * filp;
    unsigned int flag;
    int on, error = -EBADF;

    filp = fget(fd);
    if (!filp)
        goto out;
    error = 0;
    lock_kernel();
    switch (cmd) {
        case FIOCLEX:
            set_close_on_exec(fd, 1);
            break;
        case FIONCLEX:
```

```

        set_close_on_exec(fd, 0);
        break;
    case FIONBIO:
        if ((error = get_user(on, (int *)arg)) != 0)
            break;
        flag = O_NONBLOCK;
#endif __sparc__
/* SunOS compatibility item. */
if(O_NONBLOCK != O_NDELAY)
    flag |= O_NDELAY;
#endif
        if (on)
            filp->f_flags |= flag;
        else
            filp->f_flags &= ~flag;
        break;
    case FIOASYNC:
        if ((error = get_user(on, (int *)arg)) != 0)
            break;
        flag = on ? FASYNC : 0;
/* Did FASYNC state change ? */
if ((flag ^ filp->f_flags) & FASYNC) {
    if (filp->f_op && filp->f_op->fasync)
        error = filp->f_op->fasync(fd, filp, on);
    else error = -ENOTTY;
}
        if (error != 0)
            break;
        if (on)
            filp->f_flags |= FASYNC;
        else
            filp->f_flags &= ~FASYNC;
        break;
    default:
        error = -ENOTTY;
        if (S_ISREG(filp->f_dentry->d_inode->i_mode))
            error = file_ioctl(filp, cmd, arg);
        else if (filp->f_op && filp->f_op->ioctl)
            error = filp->f_op->ioctl(filp->f_dentry->d_inode, filp, cmd, arg);
}
unlock_kernel();
fput(filp);
out:
    return error;
}

```

코드 433. ioctl() 시스템 콜

가장 먼저 FD(File descriptor)를 통해서 파일 객체에 대한 포인터를 얻고, 커널에 대한 lock을 얻는다. 그리고, 이미 정의된 명령들(FIOCLEX, FIONCLEX, FIONBIO, FIOASYNC)에 대한 처리를 해준다. 만약 이곳에서 해당하는 command를 찾지 못한다면, default이하의 부분을 수행하게된다. 따라서, 디바이스 드라이버에서는 이곳에서부터 해당하는 연산을 파일 연산 구조체의 ioctl()에 정의해서 사용할 수 있게

된다. 넘겨주는 값은 파일 객체를 통해서 얻는 inode 객체와 파일 객체, 그리고, 넘겨받은 함수의 argument가 될 것이다.

FIOCLEX는 실행 종료 후 파일을 닫을 것임을, FIONCLEX는 실행 종료 후에도 파일에 대한 닫기를 하지 않는다는 것을 각각 set_close_on_exec() 함수를 통해서 나타낸다. FIONBIO는 I/O non blocking을, FIOASYNC는 비동기적인 I/O를 위해서 사용한다. 또한 get_user(X,Y)는 X에 Y값을 복사해서 가져온다. 즉, 사용자 모드에 있는 argument를 복사하는 역할을 한다. 파일 객체의 f_flag에 대한 설정과 fasync() 메소드의 호출이 있다.

이제 우리가 ioctl() 메소드를 사용하기 위해서 남은 것은 해당 command를 선언하는 일과 관련된 파라미터 값을 정의하는 일이 남게 된다. ioctl() 메소드를 호출하기 위한 command의 구조는 아래와 같다.

0 ~ 7 : instruction number

8 ~ 15 : driver identification

16 ~ 29 : parameter size

30 : write access

31 : read access

즉, 32bit으로 하나의 command를 나타내며, 명령은 8bits의 크기를 가지므로 총 256개를, 가질 수 있으며, 8bit으로 특정 디바이스 드라이버의 major 번호를 가질 수 있다. 또한 전달되는 파라미터의 크기는 10bits 만큼의 갯수를 가질 수 있으며, read인지 write인지를 나타내는 2bit가 따라온다. 같이 사용되는 macro는 ~/include/asm/ioctl.h에 아래와 같이 정의된다.

```
#define _IOC_NRBITS 8
#define _IOC_TYPEBITS 8
#define _IOC_SIZEBITS 14
#define _IOC_DIRBITS 2

#define _IOC_NRMASK ((1 << _IOC_NRBITS)-1)
#define _IOC_TYPEMASK ((1 << _IOC_TYPEBITS)-1)
#define _IOC_SIZEMASK((1 << _IOC_SIZEBITS)-1)
#define _IOC_DIRMASK ((1 << _IOC_DIRBITS)-1)

#define _IOC_NRSHIFT 0
#define _IOC_TYPESHIFT (_IOC_NRSHIFT+ _IOC_NRBITS)
#define _IOC_SIZESSHIFT (_IOC_TYPESHIFT+ _IOC_TYPEBITS)
#define _IOC_DIRSHIFT (_IOC_SIZESSHIFT+ _IOC_SIZEBITS)

#define _IOC_NONE 0U
#define _IOC_WRITE 1U
#define _IOC_READ 2U

#define _IOC(dir,type,nr,size) \
    (((dir) << _IOC_DIRSHIFT) | \
     ((type) << _IOC_TYPESHIFT) | \
     ((nr) << _IOC_NRSHIFT) | \
     ((size) << _IOC_SIZESSHIFT))

/* used to create numbers */
#define _IO(type,nr) _IOC(_IOC_NONE,(type),(nr),0)
#define _IOR(type,nr,size) _IOC(_IOC_READ,(type),(nr),sizeof(size))
#define _IOW(type,nr,size) _IOC(_IOC_WRITE,(type),(nr),sizeof(size))
#define _IOWR(type,nr,size) _IOC(_IOC_READ|_IOC_WRITE,(type),(nr),sizeof(size))

/* used to decode ioctl numbers.. */
```

```
#define _IOC_DIR(nr) (((nr) >> _IOC_DIRSHIFT) & _IOC_DIRMASK)
#define _IOC_TYPE(nr) (((nr) >> _IOC_TYPESHIFT) & _IOC_TYPEMASK)
#define _IOC_NR(nr) (((nr) >> _IOC_NRSHIFT) & _IOC_NRMASK)
#define _IOC_SIZE(nr) (((nr) >> _IOC_SIZESSHIFT) & _IOC_SIZEMASK)
```

코드 434. I/O control을 위한 macro들의 정의

메크로들에 대한 정의를 보면, type부분에 특정 디바이스 드라이버를 위해서 magic number와 같은 것을 정의할 필요가 있다. 이는 디바이스 드라이버를 작성하는 프로그램머가 자신의 디바이스 드라이버를 위한 것인지를 나타낼 수 있는 특정한 값을 넣어주면 된다. 8bit값을 필드가 가지므로 하나의 문자를 가져도 될 것이다. nr필드에 해당하는 값도 8bits를 가지므로 명령어를 256개까지 순차적으로 정할 수 있다. dir필드는 direction을 나타내는 것으로 read나 write를 가진다. 따라서, 우리가 사용하게 될 macro는 _IO(), _IOR(), _IOW(), _IOWR()이 될 것이다. 만약 넘겨받은 ioctl command를 디코딩(decoding)하고자 한다면, 이하에 있는 _IOC_DIR(), _IOC_TYPE(), _IOC_NR(), _IOC_SIZE()를 사용하면 될 것이다.

디바이스 드라이버에서의 ioctl()메쏘드의 구현은 switch문 형태의 구현이 될 것이다. 즉, 넘겨받는 command에 대한 switch문이 된다. Switch문의 default에 대한 처리는 크게 두가지로 나누어지며, 첫번째가 -EINVAL(invalid argument)이며, 두번째가 -ENOTTY가 된다. 이중에서 -ENOTTY는 posix 기준에서 가지는 에러를 나타내며, 반드시 따를 필요없다. 따라서, 둘중에서 어느것을 사용할지는 디바이스 드라이버 프로그램머가 결정해야 할 문제다. 기본적으로 커널의 ioctl()메쏘드가 취하는 것은 위의 코드에서 볼 수 있듯이 -ENOTTY이다.

이외에 또 다른 참고할 만한 것으로는 ~/Documentation/iotl_number.txt인데, 이곳에는 현재 시스템에서 어떤 것을 대상으로 ioctl()메쏘드를 선언해서 쓰고 있는지를 볼 수 있을 것이다. 새로운 ioctl() command를 정의하고자 한다면 이곳에서 쓰이고 있는 command인지를 확인하기 바란다.

6.8. 어떻게 compile할 것인가?

Kernel모듈로서 실행되는 것을 고려하기에 device driver를 compile하기 위해서는 Makefile을 만들었다고 가정한다면 __KERNEL__을 define하여 include하는 header file에서 어느 부분이 사용될지를 나타내준다. 또한 SMP machine을 고려할 것이라면 __SMP__까지도 define해 주어야 할 것이다. 또한 만약 device driver가 kernel image와 바로 link되지 않을 경우에는, MODULE을 <linux/module.h>를 include하기 전에 반드시 define되어야 한다.

또한 gcc로 compile하게 될 경우 inline에 대한 expansion을 위해서 -O flag를 compile option을 명시해 주어야 하며, error에 대한 고려를 위해서 -Wall(Warning all)을 해준다. 하지만, O2이상을 사용해서는 compile하여 실행하는데 있어서 위험이 따른다. 결과적으로 다음과 같이 될 것이다. 이것은 우리가 compile하고자 하는 device driver를 위한 Makefile의 예이다.

```
INCLUDEDIR = /usr/include
CFLAGS = -D__KERNEL__ -O -c -I$(INCLUDEDIR) -DKDEBUG -Wall
OBJS = object_file_name

all: $(OBJS)
ks8920:
    gcc $(CFLAGS) object_file_name.c
```

예제 2. Device Driver의 Compile

6.9. Debugging method.

가장 흔한 debugging방법은 printf함수를 이용하는 방법이란 것에 많은 프로그래머들은 동의를 할 것이다. Linux device driver에서도 이 같은 방법을 사용해서 할 수 있다. 즉, kernel printf 함수인 **printk**를 사용하는 것이다. 물론 이것 이외에도 다른 방법이 있지만 이것이 가장 간단한 방법으로 사용한다. 다음과 같이 사용하면 될 것이다.

```
printf(LOG_LEVEL "string and control character", variable);
```

즉, printf함수와 같이 사용하되 log level에 해당하는 상수를 사용해서 printf를 사용하면 된다. 이렇게 했을 경우 console상에 debugging message가 나타나게 된다. 만약 X console상에서 debugging message를 보고자 한다면, /etc/syslog.conf에 있는 kern.*에 해당하는 부분에 있는 comment를 제거해주고, /dev/console로 보이도록 해주어야 한다.

이외에 다른 여러가지 방법들이 있지만, 이곳에서는 다루지 않기로 한다. 자세한 디버깅 방법들에 대해서 보기를 원한다면, O'reilly에서 나온 “Linux Device Driver Programming”을 보기 바란다. 하지만, 아직 업데이트 되지 않은 내용들이 있으므로 주의해서 사용해야 할 것이다. 가장 좋은 것은 역시 커널에서 직접 어떻게 사용하는지를 코드를 보고 이해하는 것이다.

6.10. Character Device Driver

문자 디바이스 드라이버(character device driver)란 입출력에 버퍼(buffer)를 필요로 하지 않는 디바이스를 위한 드라이버이다. 이러한 디바이스 드라이버의 예로는 키보드나 마우스등을 포함한다. 이와는 달리 블록 디바이스 드라이버는 입출력 버퍼를 필요로 하며 디스크를 그 예로 들수 있을 것이다.

문자 디바이스 드라이버에 대한 접근은 파일 시스템에 있는 이름으로 가능하다. 이러한 이름이 들어있는 곳은 /dev 디렉토리이다. 이곳에서 ls -al명령을 수행할 경우 아래와 같은 것을 볼 수 있을 것이다.

```
#ls -al mem
crw-r----- 1 root      kmem      1,   1 May 6 1998  mem
```

즉, mem이라는 장치 이름은 문자 디바이스이며(c), root 소유이며, kmem이라는 그룹에 속해 있고, 접근 권한은 소유자가 read와 write를 그룹에 속한 사용자가 read만을 가지며, 다른 사용자는 아무런 권한을 가지지 못한다는 것을 보여준다. 여기서 한가지 중요한 점은 가운데 쯤에 나오는 major번호와 minor번호이다. 이것은 device에 고유하게 주어지는 값으로 major번호는 장치와 연관된 driver를 찾는데 사용되며, minor번호는 드라이버가 사용하는 값이다. 즉, 하나의 드라이버가 커널에서 minor번호를 받으면 그 minor번호에 따라 다른 연산을 해줄 수 있게 되는 것이다. 이것은 주로 하나의 드라이버가 여러 개의 장치를 제어하고자 할 때 사용된다. major번호는 0에서 255의 256개 값을 가질 수 있으며, 배열(array)형태의 테이블의 인덱스(index)값으로 사용된다. 이 배열에는 디바이스 드라이버 자료구조체를 가리키는 포인터(pointer)가 들어가게 된다.

문자 디바이스 드라이버들의 자료구조체가 저장되는 곳은 *chrdevs[]*이며, 아래의 자료구조를 가진다.

```
struct device_struct {
    const char * name;
    struct file_operations * fops;
};

...
static struct device_struct chrdevs[MAX_CHRDEV];
```

코드 435. *chrdevs[]*의 자료구조.

즉, 장치의 이름과 해당하는 파일 연산들에 대한 포인터로 구성된다. 따라서, major장치 번호를 통해서 *chrdevs[]*배열의 값을 가져오게되면, 그 장치에 해당하는 파일 연산¹⁵⁴을 할 수 있다는 것이다. 장치들에 행해지는 연산들에 대한 것은 이후에 알아 보기로 하고, 먼저 그러면 어떻게 문자 디바이스 드라이버를 등록하는지에 대한 것부터 알아보기로 하자.

¹⁵⁴ 파일 연산이라고 하면, open(), read(), write(), ioctl(), close()와 같은 것을 말한다. Linux와 같은 운영체제에서는 모든 장치를 파일로서 다룰 수 있도록 하고 있는데, 이것은 다른 운영체제와 같은 맥락을 가지고 있다.

문자 디바이스 드라이버의 등록은 `register_chrdev()` 함수를 사용한다. 아래의 코드는 `register_chrdev()` 함수를 보여준다. 코드는 ~/fs부분에서 확인할 수 있다.

```
int register_chrdev(unsigned int major, const char * name, struct file_operations *fops)
{
    if (major == 0) {
        write_lock(&chrdevs_lock);
        for (major = MAX_CHRDEV-1; major > 0; major--) {
            if (chrdevs[major].fops == NULL) {
                chrdevs[major].name = name;
                chrdevs[major].fops = fops;
                write_unlock(&chrdevs_lock);
                return major;
            }
        }
        write_unlock(&chrdevs_lock);
        return -EBUSY;
    }
    if (major >= MAX_CHRDEV)
        return -EINVAL;
    write_lock(&chrdevs_lock);
    if (chrdevs[major].fops && chrdevs[major].fops != fops) {
        write_unlock(&chrdevs_lock);
        return -EBUSY;
    }
    chrdevs[major].name = name;
    chrdevs[major].fops = fops;
    write_unlock(&chrdevs_lock);
    return 0;
}
```

코드 436. `register_chrdev()` 함수.

이 함수는 문자 디바이스 드라이버를 초기화 하는 부분에서 호출되며, 호출시 넘겨지는 `major`값을 가지고 장치를 등록해 준다. 0인 값이 `major`값으로 넘어온다면 `chrdevs[]`를 뒤쪽에서부터 빈자리를 찾아서 할당해 주며, 최대값(`MAX_CHRDEV:255`) 이상이라면 물론 에러로 처리한다. 만약 양수인 값을 `major`값으로 넘겨준다면, `chrdevs[]`에서 사용되지 않았다면 그곳에 파일 연산함수의 포인터를 두게 된다. 이렇게 등록된 문자 디바이스 드라이버들은 나중에 사용자 프로그램이 `/dev`이하의 파일을 접근하게 될 때 커널에는 `major`번호가 알려지게 되고, 그것을 이용해서 장치 드라이버를 찾은후 사용자 프로그램의 요구에 맞는 파일 연산을 불러서 수행해 주게 된다.

제거는 `unregister_chrdev()` 함수가 맡고 있다. 아래의 코드는 `unregister_chrdev()` 함수를 보여준다. 이 함수에서 하는 일은 이전에 `register_chrdev()`가 한 일을 반대로 하는 것이다. 장치의 이름과 파일 연산에 대한 포인터를 전부 `NULL`값으로 바꾼다.

```
int unregister_chrdev(unsigned int major, const char * name)
{
    if (major >= MAX_CHRDEV)
        return -EINVAL;
    write_lock(&chrdevs_lock);
    if (!chrdevs[major].fops || strcmp(chrdevs[major].name, name)) {
        write_unlock(&chrdevs_lock);
        return -EINVAL;
    }
```

```

chrdevs[major].name = NULL;
chrdevs[major].fops = NULL;
write_unlock(&chrdevs_lock);
return 0;
}

```

코드 437.unregister_chrdev()함수.

이상에서 우린 문자 디바이스 드라이버의 등록에 대해서 알아보았다. 이러한 등록은 주로 초기화 부분에서 하게 되며, 모듈을 사용할 때는 `init_module()`과 같은 모듈 초기화 함수에서 호출된다. `/dev`이하의 디렉토리에 장치를 나타내는 파일을 생성하는 것은 특별한 명령을 필요로 하며, 이는 `mknod`라는 명령을 사용한다. `mknod`를 사용하는 방법은 아래와 같다.

`mknod [OPTION]... NAME TYPE [MAJOR MINOR]`

여기서 OPTION으로는 `-m`, `--mode=MODE`(접근허가 모드), `--help`(도움말), `--version`(버전 정보)를 보여주며, NAME으로는 디바이스를 가리키는 이름을, TYPE에는 `b`(블록 디바이스), `c`(or `u`)(문자 디바이스 혹은 unbuffered 디바이스), `p`(FIFO)와 같은 값이 올 수 있다. 마지막으로 장치의 major와 minor번호를 준다. 가령 예을 들어서 위에서 나온 `/dev/mem`같은 문자 디바이스 파일을 만들고 싶다면,

`mknod -mode=0640 mem c 1 1`

과 같이 하면 될 것이다. 물론 이렇게 생성만 한다고 모든 일이 끝나는 것은 아니며, 직접 디바이스 드라이버 또한 접근할 수 있도록 만들어 주어야 할 것이다. 이것이 바로 디바이스 드라이버를 만드는 사람이 해줄 일이다. 나중에 이와 관련된 소스코드를 하나 분석하게 될 때 다시 한번 더 보게 될 것이다.

major번호와 minor번호와 관련된 함수나 매크로, 혹은 인라인(in-line)함수의 정의는 아래와 같다. major 및 minor함수는 32bit으로 되어 있으며, 하위 16bit의 중에서 다시 상위의 8bit는 major번호를 하위의 8bit는 minor번호를 나타낸다.

```

#define MINORBITS      8
#define MINORMASK     ((1U << MINORBITS) - 1)
typedef unsigned short kdev_t;
#define MAJOR(dev)      ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)      ((unsigned int) ((dev) & MINORMASK))
#define HASHDEV(dev)    ((unsigned int) (dev))
#define NODEV          0
#define MKDEV(ma,mi)    (((ma) << MINORBITS) | (mi))

...
static inline unsigned int kdev_t_to_nr(kdev_t dev) {
    return (MAJOR(dev)<<8) | MINOR(dev);
}
static inline kdev_t to_kdev_t(int dev)
{
    int major, minor;
#endif
    major = (dev >> 16);
    if (!major) {

```

```

        major = (dev >> 8);
        minor = (dev & 0xff);
    } else
        minor = (dev & 0xffff);
#endif
        return MKDEV(major, minor);
}

```

코드 438.kdev_t.h

위에서 나열한 매크로 중에서 `MAJOR()`, `MINOR()`, `MKDEV()`는 자주 쓰이기에 기억해 두는 것이 좋다. 이들 각각은 디바이스를 나타내는 번호에서 `major`번호와 `minor`번호를 구할 때와 두 번호를 합치는 경우에 사용한다.

다시 이전으로 약간 돌아가서 이제는 문자 디바이스 드라이버와 관련된 파일 연산에 대해서 알아보도록 하자. 실제적으로 디바이스 드라이버를 작성하는 사람은 이곳에서 자신의 일이 전부 결정된다고 할 수 있다. 즉, 커널에서 진입점으로 사용하는 이러한 함수들을 작성하는 것이 디바이스 드라이버를 개발하는 사람이 해야할 일이다. 여러 다른 운영체제 역시 이러한 것을 명시하고 있으며, 어떤 것은 디바이스 드라이버 개발 툴(tool)을 제공하기도 한다. 하드웨어의 특성을 이해 했다면, 이러한 진입 함수와 자료구조를 만드는 일만 남은 것이다.

```

struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
};

```

코드 439. file_operations구조체.

파일 연산 구조체는 문자 디바이스 드라이버 대한 모든 연산들을 포함한다. 즉, 여기에 정의된 함수를 통해서 커널이 접근하게 된다. 이제 각 필드들에 들어가는 함수들에 대한 것을 살펴보도록

하자.(이곳에서는 간단히 하는 역할만을 보게 될 것이며, 자세한 것은 직접 프로그램 코드를 보면서 이해하도록 하자.)

- struct module *owner – 어떤 모듈로 올라간 디바이스 드라이버와 연관을 가지는지를 나타내는 포인터.
- lseek()함수 포인터 – 현재의 읽기 쓰기 포인터를 옮기고자 할 때 사용된다. 새로운 위치가 리턴 값으로 돌려진다.
- read/write()함수 포인터 – 데이터를 장치에서 읽어드리거나 쓰고자 할 때 사용된다. 성공적으로 읽혀지거나 쓰여진 byte의 수를 리턴 값으로 돌려준다.
- readdir()함수 포인터 – 디렉토리에 대해서만 사용되며, 디바이스에 대해선 NULL값을 가진다.
- select()함수 포인터 – 디바이스가 읽기나 쓰기, 혹은 예외 상황을 일으켰는지를 묻고자 할 때 사용된다. 돌려주는 값은 어떤 조건이 현재 만족되었지를 나타낸다.
- ioctl()함수 포인터 – 특정 디바이스에 의존적인(혹은 독특한) 명령을 수행하고자 할 때 사용된다. 되돌려주는 값은 성공했다는 것을 나타내는 양수가 될거나 실패일 경우는 음수값이 나오게 된다.
- mmap()함수 포인터 – 디바이스의 메모리 일부를 현재 프로세스의 메모리 영역으로 매핑(mapping)하고자 할 때 사용한다. 가령 예를 들어서 그래픽을 하기 위해서 디바이스의 프래임 버퍼를 사용자 프로세스의 주소 영역으로 매핑해서 직접 응용프로그램에서 접근할 수 있도록 하는 경우가 있을 수 있다.
- open()/release()함수 포인터 – 디바이스에 대한 열기/닫기를 할 때 사용된다. 열기는 가장 먼저 호출되는 함수이다. (물론 모듈을 올리는 경우는 init_module()이 호출되면서 각종 초기화 동작이 일어나야 된다.)
- fsync()함수 포인터 – 디바이스에 대한 모든 연산의 결과를 지연하지 않고 즉시 일어나도록 한다.
- fasync()함수 포인터 – FASYNC 플랙의 변화를 디바이스에 알리기 위해서 사용한다. BSD호환을 지원하기 위한 것이다.
- lock()함수 포인터 – 디바이스에 lock을 걸고자 할 때 사용된다.
- readv()/writev()함수 포인터 – BSD형식의 read()/write()를 지원하기 위한것이다.

물론 위에서 열거한 모든 함수를 디바이스 드라이버가 가질 필요는 없다. 즉, 자신에게 필요한 함수만을 정의해서 파일 연산 구조체에 넣고, 이것을 문자 디바이스 드라이버를 등록(register_chrdev())함수를 불러서 등록해 주면 되는 것이다. 없는 함수들에 대해서는 커널이 에러로 처리해 준다.

위에서 나열하지는 않았지만 만약 다른고자 하는 디바이스가 인터럽트를 발생시킨다면 이와 관련된 처리도 해주어야 한다. 이것은 인터럽트 핸들러를 디바이스 드라이버가 설치함으로써 해결된다. 잠시 이와 관련된 것들을 보기로 하자. 먼저 인터럽트 핸들러를 등록하고 해제 하는 방법이다. 관련된 함수들은 ~/arch/i386/kernel/irq.c에서 뽑아 낸 것이다.

```
int request_irq(unsigned int irq,
               void (*handler)(int, void *, struct pt_regs *),
               unsigned long irqflags,
               const char * devname,
               void *dev_id)
{
    int retval;
    struct irqaction * action;
    ...
    if (irq >= NR_IRQS)
        return -EINVAL;
```

```

if (!handler)
    return -EINVAL;

action = (struct irqaction *)
    kmalloc(sizeof(struct irqaction), GFP_KERNEL);
if (!action)
    return -ENOMEM;

action->handler = handler;
action->flags = irqflags;
action->mask = 0;
action->name = devname;
action->next = NULL;
action->dev_id = dev_id;

retval = setup_irq(irq, action);
if (retval)
    kfree(action);
return retval;
}

```

코드 440. irq.c(1)

먼저 설치하고자 하는 인터럽트 핸들러를 인수로 넘겨준다. 이것은 나중에 인터럽트가 발생했을 때 불려지는 함수의 이름이 되겠다(물론 주소 형태로 넘어가게 되겠지만). 그리고, 인터럽트를 어떻게 사용할 것인가를 나타내는 irqflags과 디바이스의 이름, 디바이스의 ID가 같이 인수로 전달된다.

그리고, 디바이스 드라이버에 할당된 인터럽트를 해제하는 방법이다. 핸들러를 제거하는 것으로 가능하다.

```

void free_irq(unsigned int irq, void *dev_id)
{
    irq_desc_t *desc;
    struct irqaction **p;
    unsigned long flags;

    if (irq >= NR_IRQS)
        return;
    desc = irq_desc + irq;
    spin_lock_irqsave(&desc->lock, flags);
    p = &desc->action;
    for (;;) {
        struct irqaction * action = *p;
        if (action) {
            struct irqaction **pp = p;

```

```

        p = &action->next;
        if (action->dev_id != dev_id)
            continue;
        /* Found it - now remove it from the list of entries */
        *pp = action->next;
        if (!desc->action) {
            desc->status |= IRQ_DISABLED;
            desc->handler->shutdown(irq);
        }
        spin_unlock_irqrestore(&desc->lock,flags);
#endif CONFIG_SMP
        /* Wait to make sure it's not being used on another CPU */
        while (desc->status & IRQ_INPROGRESS)
            barrier();
#endif
        kfree(action);
        return;
    }
}

```

코드 441.irq.c(2)

인터럽트 번호와 디바이스 아이디를 인수로 넘겨 받아서, 인터럽트 디스크립터 값을 구한후에 그곳에 저장된 핸들러들을 차례로 디바이스 아이디값을 가지는지 찾아본다. 만약 같은 디바이스 아이디 값을 가지는 핸들러가 발견되면, 인터럽트 디스크립터의 상태값을 IRQ_DISABLED로 만들고 해당하는 핸들러를 제거해 준다. SMP(Symmetry Multi-Processor)에서는 다른 프로세스에서 인터럽트가 진행중인 경우도 있으므로 이에 해당하는 처리가 끝나기를 잠시 기다릴 필요가 있다.

지금까지 대략적으로 문자 장치 디바이스 드라이버를 작성하기 위해서 필요한 것들을 알아보았다. 물론 이것 이외에도 실제적으로 더 많은 장치에 대한 정보를 알아야 하겠지만, 이것은 실제 코드를 보면서 익히는 것이 효율적일 것이다. 이하에서는 실제적으로 어떻게 문자 디바이스 드라이버를 만드는지를 보도록 하겠다.

6.10.1. 문자 디바이스 드라이버의 예

소스 코드의 예제로 볼 것은 ~/drivers/char/softdog.c이다. 이것은 소프트웨어적으로 watchdog을 구현하는 예이다. watchdog이란 집을 지키는 개를 말하는 것으로 컴퓨터에서는 시스템의 상황을 감시해서 이상이 발생할 경우 이를 해결하는 역할을 해준다.

```

static struct miscdevice softdog_miscdev=
{
    WATCHDOG_MINOR,
    "watchdog",
    &softdog_fops
}

```

```

};

...
static int __init watchdog_init(void)
{
    int ret;

    ret = misc_register(&softdog_mscdev);
    if (ret)
        return ret;
    printk("Software Watchdog Timer: 0.05, timer margin: %d sec\n", soft_margin);
    return 0;
}

static void __exit watchdog_exit(void)
{
    misc_deregister(&softdog_mscdev);
}

module_init(watchdog_init);
module_exit(watchdog_exit);

```

코드 442. softdog.c(1)

모듈로서 컴파일되어 올라갈 때 가장 먼저 실행되는 것은 module_init()로 선언된 __init watchdog_init(void)이다. 즉, insmod를 실행해서 모듈을 올릴 때 실행된다. 이곳에서 하는 일은 가장 먼저 문자 디바이스 드라이버로 등록하는 부분이다. miscdevice구조체는 아래와 같이 정의 된다.

```

struct miscdevice
{
    int minor;
    const char *name;
    struct file_operations *fops;
    struct miscdevice * next, * prev;
    devfs_handle_t devfs_handle;
};

```

즉, 사용하고자 하는 디바이스의 minor번호와 이름, 그리고, 파일 연산구조체에 대한 포인터와 내부적으로 사용된 포인터 및 devfs_handle_t이 들어간다. softdog.c에서 관심을 가지는 부분은 앞의 세개의 필드이다. misc_register()함수는 devfs_register()함수를 내부적으로 호출하며, devfs_register()함수는 다시 register_chrdev()함수를 호출해서 문자 디바이스 드라이버로 등록하는 역할을 해준다. 이를 위해서 넘겨주는 인수값이 바로 miscdevice 자료구조이다. 다시 등록된 모듈을 나중에 rmmod와 같은 명령으로 커널과의 링크를 해제할 경우 호출되는 것이 module_exit()로 선언된 __exit watchdog_exit()이다. 이 함수에서 해주는 일은 단순히 문자 디바이스 드라이버를 해제하는 misc_deregister()함수를 호출하는 것이다. misc_deregister()함수는 이전과 마찬가지로 devfs_deregister()함수를 호출하고, 다시 unregister_chrdev()함수가 최종적으로 호출되어 문자 디바이스 드라이버를 해제해 준다.

```

static struct file_operations softdog_fops=
{
    owner:          THIS_MODULE,
    write:          softdog_write,
    ioctl:          softdog_ioctl,
    open:           softdog_open,
    release:        softdog_release,
};

```

코드 443.softdog.c(2)

이전 파일 연산 구조체를 볼 차례이다. 위의 코드에서 owner, write, ioctl, open, release에 대한 것만을 선언했다. 이처럼 문자 디바이스 드라이버가 커널에 제공하고자 하는 함수만을 정의해 넣으면 되는 것이다. 물론 이 함수들을 구현하는 것은 드라이버 작성자가 해 주어야 할 일이며 계속 설명해 나가도록 하겠다.

THIS_MODULE이라는 macro는 현재의 모듈의 포인터를 돌려준다. 나머지 각각의 필드들이 하는 역할은 이전에 설명한 것을 참조하기 바란다. 이젠 각각의 함수들을 볼 차례다.

```
static int timer_alive;
...
static int softdog_open(struct inode *inode, struct file *file)
{
    if(timer_alive)
        return -EBUSY;
#ifndef CONFIG_WATCHDOG_NOWAYOUT
    MOD_INC_USE_COUNT;
#endif
    /*
     *      Activate timer
     */
    mod_timer(&watchdog_ticktock, jiffies+(soft_margin*HZ));
    timer_alive=1;
    return 0;
}
```

코드 444.softdog.c(3)

소프트웨어 watchdog은 타이머와 관련되어 일정 시간 간격으로 계속적으로 시스템을 감시한다. 따라서, 만약 타이머가 이미 가동중이라면 에러가 될 것이다. 모듈에 대한 카운트 값을 증가 시켜서, 모듈이 현재 사용중이라는 것을 나타낸 다음, watchdog에서 타임 아웃이 될 때 실행될 함수를 타이머 리스트에 등록시켜준다. 이것은 mod_timer()함수를 불러서 처리하는데, 이전에 설치된 타이머가 있다면 이를 삭제하고 새로운 타이머를 설치한다. 그리고, 마지막으로 timer_alive값을 설정해서 현재 watchdog 타이머가 살아있다는 것을 알려준다.

```
static int softdog_release(struct inode *inode, struct file *file)
{
    lock_kernel();
#ifndef CONFIG_WATCHDOG_NOWAYOUT
    del_timer(&watchdog_ticktock);
#endif
    timer_alive=0;
    unlock_kernel();
    return 0;
}
```

코드 445.softdog.c(4)

watchdog을 해제하는 것은 softdog_release()함수이다. 설정된 타이머를 제거하고, 현재 watchdog이 설정되지 않았다는 것을 표시한다.

```
static ssize_t softdog_write(struct file *file, const char *data, size_t len, loff_t *ppos)
{
    /* Can't seek (pwrite) on this device */
    if (ppos != &file->f_pos)
        return -ESPIPE;
    if(len) {
        mod_timer(&watchdog_ticktock, jiffies+(soft_margin*HZ));
```

```

        return 1;
    }
    return 0;
}

```

코드 446.softdog.c(5)

watchdog에 대한 쓰기(write)는 softdog_write()가 맡고 있다. 이곳에선 타이머를 새롭게 업데이트(update)하는 일을 한다. 실제로 데이터를 쓰는 것이 아니라 단순히 타이머만 새롭게 경신한다.

```

static int softdog_ioctl(struct inode *inode, struct file *file,
                        unsigned int cmd, unsigned long arg)
{
    static struct watchdog_info ident=
    {
        0,
        0,
        "Software Watchdog"
    };
    switch(cmd)
    {
        default:
            return -ENOIOCTLCMD;
        case WDIOC_GETSUPPORT:
            if(copy_to_user((struct watchdog_info *)arg, &ident, sizeof(ident)))
                return -EFAULT;
            return 0;
        case WDIOC_GETSTATUS:
        case WDIOC_GETBOOTSTATUS:
            return put_user(0,(int *)arg);
        case WDIOC_KEEPALIVE:
            mod_timer(&watchdog_ticktock, jiffies+(soft_margin*HZ));
            return 0;
    }
}

```

코드 447.softdog.c(6)

softdog_ioctl() 함수는 사용자 프로그램에서 watchdog을 콘트롤 할 경우에 사용한다. 이곳에서는 사용자 프로그램에서 전송된 데이터를 watchdog_info데이터 구조체에 저장하거나, 혹은 데이터를 사용자 프로그램으로 전송할 목적으로 사용한다. 사용자 프로세스 주소공간과 커널 공간사이의 데이터 전송을 위해서 프로그램에서는 *copy_to_user()*함수와 *put_user()*함수를 사용한다. 이 함수들은 *~/include/asm/uaccess.h*에 정의되어 있다. 이 함수들은 디바이스 드라이버에서 사용자 프로그램으로 데이터를 전송하는 역할을 한다.

```

static void watchdog_fire(unsigned long);

static struct timer_list watchdog_ticktock = {
    function: watchdog_fire,
};

...
static void watchdog_fire(unsigned long data)
{
#ifndef ONLY_TESTING
    printk(KERN_CRIT "SOFTDOG: Would Reboot.\n");
#else
    printk(KERN_CRIT "SOFTDOG: Initiating system reboot.\n");
}

```

```

machine_restart(NULL);
printf("WATCHDOG: Reboot didn't ?????\n");
#endif
}

```

코드 448.softdog.c(7)

이 함수는 watchdog 타이머가 경과했을 경우에 호출되는 함수이다. 현재는 에러 메시지를 표시한 다음 시스템을 재 부팅시킨다. machine_restart()함수는 ~/arch/i386/kernel/process.c를 참조하기 바란다.

이상에서 문자 디바이스 드라이버에 대해서 살펴보았다. 블록 모드 디바이스 드라이버로 넘어가기 전에 ioctl()함수를 사용하는 것에 대해서 조금더 자세히 보도록 하자.

6.11. Block Device Driver

블록 디바이스 드라이버는 버퍼를 가지고 대용량의 데이터를 주고 받는 디바이스를 위한 드라이버이다. 따라서, 이전의 문자 디바이스 드라이버와는 다른 구조를 가지고 있다. 물론 /dev 디렉토리 이하에 등록되는 방법에는 한가지를 제외하고 차이가 없다. 즉, ls -al /dev를 했을 때, 제일 앞에 'b'라고 나오는 것들이 블록 디바이스 드라이버를 위한 디바이스 노드(node)가 되는 것이다. 블록 디바이스 드라이버가 가질 수 있는 최대의 major번호는 ~/include/linux/major.h에 *MAX_BLKDEV*라는 상수로 255가 정의되어 있다. 블록 디바이스를 위한 커널의 데이터 구조는 *blkdevs[]*가 있으며, 이것의 정의는 다음과 같다.

```

static struct {
    const char *name;
    struct block_device_operations *bdops;
} blkdevs[MAX_BLKDEV];

```

커널은 블록 디바이스 드라이버의 major번호를 가지고 구조체내에서의 인덱스 값으로 사용한다. 또한 구조체 내에서 해당 드라이버가 제공하는 함수들도 찾게 된다.

블록 디바이스 드라이버를 등록하는 방법은 *register_blkdev()*함수가 있으며, 해제는 *unregister_blkdev()*함수가 맡고 있다. 아래는 두 함수의 코드이다. 둘다 ~/fs/block_dev.c에서 가져왔다.

```

int register_blkdev(unsigned int major, const char * name, struct block_device_operations *bdops)
{
    if (major == 0) {
        for (major = MAX_BLKDEV-1; major > 0; major--) {
            if (blkdevs[major].bdops == NULL) {
                blkdevs[major].name = name;
                blkdevs[major].bdops = bdops;
                return major;
            }
        }
        return -EBUSY;
    }
    if (major >= MAX_BLKDEV)
        return -EINVAL;
    if (blkdevs[major].bdops && blkdevs[major].bdops != bdops)
        return -EBUSY;
    blkdevs[major].name = name;
}

```

```

blkdevs[major].bdops = bdops;
return 0;
}

```

코드 449. block device driver의 등록

major 번호가 주어진 경우에는 이것을 사용해서 빙자리가 있는지를 확인하게 되지만, 그렇지 않은 경우는 가장 큰 번호에서부터 빙자리를 검사해서 블록 디바이스 드라이버를 설치할 자리를 찾게 된다. 빙자리를 찾은 경우에는 그곳에 블록 디바이스의 이름과 드라이버에서 제공하는 블록 디바이스 연산에 대한 포인터를 넣고 복귀하게 된다.

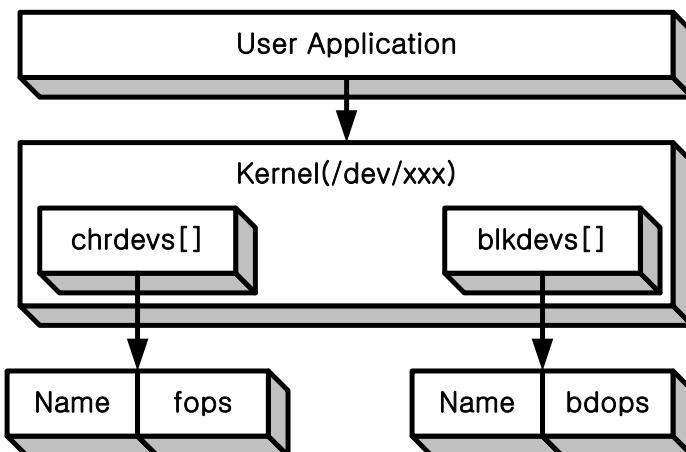
```

int unregister_blkdev(unsigned int major, const char * name)
{
    if (major >= MAX_BLKDEV)
        return -EINVAL;
    if (!blkdevs[major].bdops)
        return -EINVAL;
    if (strcmp(blkdevs[major].name, name))
        return -EINVAL;
    blkdevs[major].name = NULL;
    blkdevs[major].bdops = NULL;
    return 0;
}

```

코드 450. block device driver의 해제

해제는 등록의 역으로 행한다. 즉, 주어진 major 번호를 가지고 해제할 드라이버의 위치를 찾게 되며, 찾은 경우 그 위치에 NULL을 넣어주면 된다. 물론 디바이스 드라이버가 사용하는 자원들은 이미 해제가 되어야 할 것이다. 그렇지 않다면, 커널의 메모리내에 드라이버가 사용하던 메모리가 해제되지 않고 남아 있을 것이다. 이는 자원의 낭비가 된다.

**그림 59. 문자 디바이스 드라이버와 블록 디바이스 드라이버의 구성.**

앞에서 보았던 문자 디바이스 드라이버의 설치 및 해제와 거의 같지만 여기선 *bdops*라는 것이 차이가 난다. 즉, 블록 디바이스 드라이버에 행해지는 연산이 문자 디바이스 드라이버와는 다르다는 것이다. 차이만 제외한다면 [그림52]와 같은 구조가 사용되는 것을 알 수 있을 것이다. 물론 블록 디바이스 드라이버와 커널간에는 파일 시스템 및 버퍼 관리가 들어가며, 이곳에서는 생략했다. 이와 관련된 것을 보고자 한다면 파일 시스템을 보기 바란다.

그럼 이제는 *bdops*에 올 수 있는 블록 디바이스 드라이버의 연산에 대해서 알아 보기로 하자. 코드는 ~/include/linux/fs.h에서 가져왔다. 참고로 2.2.X버전의 커널에서는 문자 디바이스 드라이버와 블록

디바이스 드라이버의 연산에 대해 같은 자료구조를 사용하고 있으나, 2.4.0의 커널 버전에서는 다른 구조를 사용해서 나타내고 있다.

```
struct block_device_operations {
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
    int (*check_media_change) (kdev_t);
    int (*revalidate) (kdev_t);
};
```

다음은 각 연산의 역할에 대한 설명이다.

- open()함수 포인터 – 블록 디바이스에 대한 열기(open)연산을 처리한다. 모든 장치는 접근 되기전에 열어야 하므로 이곳에서 초기화에 필요한 연산을 수행할 수 있다.
- release()함수 포인터 – 블록 디바이스에 대한 닫기(close)연산을 처리한다. open()함수에서 행해진 것을 원래대로 복원시키는 연산을 해야 할 것이다.
- ioctl()함수 포인터 – 블록 디바이스에 대한 I/O Control을 수행한다. 디바이스에 특정한 연산을 정의해서 사용할 수 있다.
- check_media_change()함수 포인터 – 이 함수의 포인터는 블록 디바이스 드라이버에 고유한 것으로 제거 가능한 미디어(media)와 같이 사용된다. 예로서는 플로피(floppy)가 있을 수 있으며, 마지막으로 연산이 있은후에 물리적인 변화가 있었는지를 알기위해 사용된다.
- revalidate()함수 포인터 – 이 함수 포인터 역시 블록 디바이스 드라이버에 고유한 것으로, 버퍼(buffer) 캐시(cache)의 관리를 위해서 사용된다.

블록 디바이스 드라이버와 다른 커널 구성요소간의 관계는 [그림53]과 같다. 즉, ll_rw_block()에서 블록 디바이스 드라이버에 대한 접근이 있다. 사용자 응용 프로그램이 블록 디바이스를 나타내는 파일을 통해서 접근하게 되면, VFS(Virtual File System)에 의해서 block_read()/block_write()나 bread()/breada()가 호출이 되며, 이는 다시 버퍼 캐시(buffer cache)에 이미 내용이 들어 있는지를 확인하게 되며, 만약 없다면, ll_rw_block()을 호출한다. ll_rw_block()함수는 요청(request)를 만든후 전략루틴(strategy routine)을 스케줄링 한 다음에 복귀하게 되고, 최종적으로 전략루틴이 실제적인 요청을 처리하게 된다.

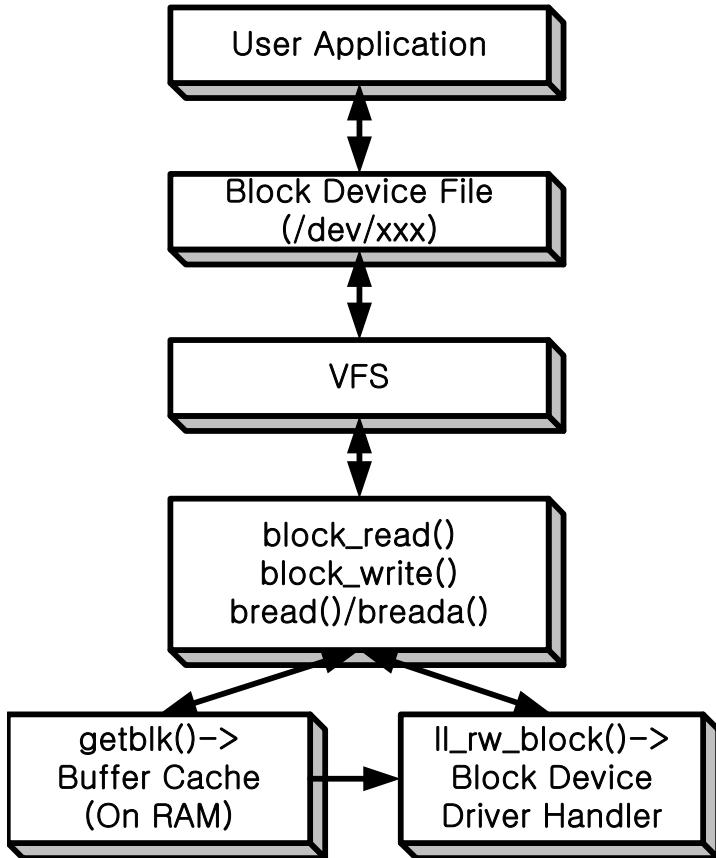


그림 60. 블록 디바이스에 대한 접근.

[그림53]에서 버퍼에 읽고자 하는 내용이 있는 가를 확인하는 것이 getblk() 함수이며, 없을 경우에 다시 ll_rw_block()을 호출해서 블록 디바이스 드라이버에 접근하게 된다.

실제로 블록 디바이스에 대한 기본(default)연산으로는 ~/fs/block_dev.c에 default_blk_fops 구조체가 정의되어 있다. 이것은 블록 디바이스 드라이버가 따로 설정하지 않는다면 기본 연산으로 이것을 사용할 것이다.

```

struct file_operations def_blk_fops = {
    open:          blkdev_open,
    release:       blkdev_close,
    llseek:        block_llseek,
    read:          block_read,
    write:         block_write,
    fsync:         block_fsync,
    ioctl:         blkdev_ioctl,
};
  
```

코드 451. 기본(default) 블록 디바이스 연산들.

이와 같은 함수들은 문자 디바이스 드라이버에 대해서 살펴볼 때와 같은 방식으로 사용되며 다음과 같다.

- ✓ open() – 블록 디바이스에 대한 열기를 행하는 함수이다. 모든 장치는 사용되기 전에 먼저 있는지를 확인하는 절차와 열기 연산이 수행되어야 한다.
- ✓ release() – 블록 디바이스를 해제하는 함수이다. 모든 블록 디바이스에 대한 연산이 끝났다면 이것을 호출해서 블록 디바이스를 해제할 수 있다.
- ✓ llseek() – 블록 디바이스에 대해서 현재의 읽기와 쓰기 위치를 변경하려고 할 때 호출되는 함수이다.

- ✓ read() – 블록 디바이스에 대한 읽기 연산을 수행한다. 예를 들어, 디스크와 같은 블록 디바이스에서 데이터를 읽고자 할 때 호출된다.
- ✓ write() – 블록 디바이스에 대한 쓰기 연산을 수행하고자 할 때 수행된다. 예를 들어, 디스크로 부터 데이터를 읽고자 할 때 호출된다.
- ✓ fsync() – 메모리내에 존재하는 메모리 블록을 디스크와 같은 블록 디바이스에 다시 쓰고자 할 때 호출된다. 즉, 현재 메모리내의 내용과 디스크 상의 내용이 같은 것을 가지고 있음을 보장해준다.
- ✓ ioctl() – 블록 디바이스에 대한 제어연산을 행하고자 할 때 호출된다. 특정한 명령을 수행하도록 만들어줄 수 있으며, 제어 정보를 얻고자 할 때도 사용될 수 있다.

블록 디바이스 드라이버 작성자는 위와 같은 연산에 대해서 커널이 제공하는 일반적인 함수를 기본으로 사용할 수 있으며, 혹은 자신만이 가진 방법으로 대체 할 수 있다. 이제는 실제적인 블록 디바이스의 연산이 일어나는 순서를 보기로 하자.

블록 디바이스에 대한 접근이 ll_rw_block()을 호출하게 될 경우, 이 함수는 다시 generic_make_request() 함수를 호출하게 되며, generic_make_request() 함수는 요구 큐(request queue)¹⁵⁵와 관련된 make_request() 함수를 호출한다. 이것은 버퍼의 초기화에서 __make_request() 함수를 가리키며¹⁵⁶, __make_request() 함수에서 요구 큐(request queue)의 plug_device_fn() 함수를 호출해서 tq_disk 태스크 큐에 요구 큐의 plug_tq를 삽입한 후 복귀한다. 나중에 tq_disk가 스케줄링 될 때 plug_tq가 실행 기회를 얻게 되며, 이곳에서는 generic_unplug_device() 함수가 큐에 대해서 호출된다. generic_unplug_device() 함수는 단순히 현재 큐를 살펴서 남은 요구가 있다면 다시 큐의 request_fn() 함수를 호출하는 역할을 한다. request_fn() 함수는 디바이스의 전략(strategy) 루틴이다. 전략 루틴은 요청 큐에 있는 현재의 요청을 처리하며, 블록 디바이스의 컨트롤러(controller)를 데이터의 전송이 끝났을 때 인터럽트를 발생 시키도록 설정한 후 종료한다. 나중에 인터럽트가 발생할 때 다시 bottom half를 활성화 시켜두게 되며, bottom half에서 요청 큐에서 처리한 요청을 제거한 후 다시 전략 루틴이 다음번 요청을 처리하도록 만든다.

모든 블록 디바이스에 대한 연산은 버퍼단위로 일어나게 되며, 이것을 위해서 커널에선 버버 헤드를 정의하고 있다. 버퍼 헤드에 대한 데이터 구조는 buffer_head이다.

```
struct buffer_head {
    /* First cache line: */
    struct buffer_head *b_next; /* Hash queue list */
    unsigned long b_blocknr; /* block number */
    unsigned short b_size; /* block size */
    unsigned short b_list; /* List that this buffer appears */
    kdev_t b_dev; /* device (B_FREE = free) */

    atomic_t b_count; /* users using this block */
    kdev_t b_rdev; /* Real device */
    unsigned long b_state; /* buffer state bitmap (see above) */
    unsigned long b_flushtime; /* Time when (dirty) buffer should be written */

    struct buffer_head *b_next_free; /* lru/free list linkage */
    struct buffer_head *b_prev_free; /* doubly linked list of buffers */
    struct buffer_head *b_this_page; /* circular list of buffers in one page */
    struct buffer_head *b_reqnext; /* request queue */

    struct buffer_head **b_pprev; /* doubly linked list of hash-queue */
    char * b_data; /* pointer to data block (512 byte) */
    struct page *b_page; /* the page this bh is mapped to */
    void (*b_end_io)(struct buffer_head *bh, int uptodate); /* I/O completion */
}
```

¹⁵⁵ 뒷에서 다시 살펴보겠지만, 블록 디바이스에 대한 요청이 큐의 형태로 만들어져서 관리되는 것을 의미한다.

¹⁵⁶ 다른 make_request() 함수로 blk_queue_make_request() 함수를 사용해서 바꿀 수 있다.

```

void *b_private;           /* reserved for b_end_io */

unsigned long b_rsector;    /* Real buffer location on disk */
wait_queue_head_t b_wait;

struct inode *            b_inode;
struct list_head          b_inode_buffers; /* doubly linked list of inode dirty buffers */
};

}

```

코드 452. 버퍼 헤드의 자료구조

버퍼 헤드 구조체의 각각의 필드는 아래와 같은 의미를 가지고 있다.

Field	Description
struct buffer_head *b_next	버퍼헤드의 연결 포인터
unsigned long b_blocknr	관련된 논리적인 블록의 번호
unsigned short b_size	블록의 크기
unsigned short b_list	현재의 버퍼가 속한 LRU 리스트
kdev_t b_dev	가사의 디바이스 ID
atomic_t b_count	현재의 블록을 사용 계수
kdev_t b_rdev	실제 블록 디바이스의 ID
unsigned long b_state	버퍼의 상태를 나타내는 플랙
unsigned long b_flushtime	버퍼를 flush하는 시간(비우는 시간)
struct buffer_head *b_next_free	LRU나 Free 리스트에 들어있는 다음 버퍼헤드
struct buffer_head *b_prev_free	LRU나 Free 리스트에 들어있는 이전의 버퍼헤드
struct buffer_head *b_this_page	페이지당 할당된 버퍼의 리스트
struct buffer_head *b_reqnext	Request 큐에서 연결된 버퍼헤드에 대한 포인터
struct buffer_head **b_pprev	Hash 큐상에서의 이전의 버퍼헤드
char *b_data	버퍼에 대한 포인터(데이터가 들어가있는 곳에 대한 포인터)
struct page *b_page	현재의 버퍼 헤드가 mapping 된 page에 대한 포인터
void (*b_end_io)(struct buffer_head *bh, int uptodate)	I/O를 끝마치는 routine에 대한 포인터
void *b_private	b_end_io routine에 reserved된 부분
unsigned long b_rsector	디스크 상에서의 실제적인 버퍼의 위치
wait_queue_head_t b_wait	버퍼헤드의 wait 큐
struct inode *b_inode	블록과 관련된 inode에 대한 포인터
struct list_head b_inode_buffers	inode dirty 버퍼들에 대한 연결 리스트

표 45. 버퍼 헤드의 필드값

버퍼 헤드가 가질 수 있는 상태(b_state)에는 다음과 같은 값이 있다.

- BH_Uptodate – 현재 버퍼는 유효한 데이터를 가지고 있다.
- BH_Dirty – 버퍼가 dirty하다. 즉, 현재의 버퍼는 디스크에 있는 데이터와는 다르다는 의미다. 따라서 나중에 write연산이 필요하다.
- BH_Lock – 현재 버퍼가 lock되어 있다. 즉, 디스크와 관련된 연산이 진행 중임을 나타낸다.
- BH_Req – 해당하는 블록이 이미 요청(request)되었으며, 유효한 데이터를 가지고 있을 때를 나타낸다. 만약 설정되지 않았다면, 유효하지 않다(invalidate).
- BH_Mapped – 버퍼가 현재 디스크에 mapping되어서 사용되고 있다.
- BH_New – 버퍼가 새로운 것이며, 아직 쓰기(write)되지 않았다.
- BH_Protected – 버퍼가 protected되고 있다는 것을 표시한다. 이러한 버퍼들은 free되지 않는다.

버퍼 헤드의 자료구조에서 `b_dev`과 `b_rdev`는 각각 블록을 포함하는 가상의 디바이스 ID와 실제의 디바이스를 나타내며, 이는 RAID(Redundant Array of Independent Disk)를 구현하기 위한 것이다. 이러한 디바이스는 각각의 디스크에 대해서 동시에 접근해서 원하는 데이터를 빠른 시간안에 구해올 수 있다는 장점을 가지기 때문이다. 흔히 멀티미디어 데이터를 여러개의 디스크에 분산 시켜서 저장해두고 동시에 접근해서 필요한 데이터를 읽어들일 경우에 유리하다.

블록 디바이스는 한번에 하나의 블록 만을 전송할 수 있다. 여기에 필요한 연산은 먼저 원하는 블록을 찾는 과정과 디스크의 헤드를 옮겨놓는 과정등의 여러가지 기계적인 동작을 요하게 된다. 따라서, 블록 단위로만 연산을 하는 것은 전체적인 시스템의 성능을 저하시킬 여지가 있다. 따라서, 커널은 이와 같은 블록 단위의 연산을 한번에 하나만 처리하는 것이 아니라, 여러개의 블록을 묶어서 한번에 처리하도록 하고 있다. 이것은 커널이 물리적인 동작이 일어나는 회수를 최소로 줄여서 성능을 극대화 하는 것이다.

블록 디바이스에 대한 모든 연산은 블록 디바이스 요청(block device request)를 생성한다. 즉, 커널의 어떤 부분이든 블록 디바이스에 대해서 연산을 요청하려고 하면, 블록 디바이스 요청 구조를 만들어서 블록 디바이스에 넘겨주게 된다. 이와같은 블록 디바이스 요청 구조에는 요청된 블록과 요청이 읽기인지 쓰기인지를 나타내는 값이 들어있게 되며, 커널은 요청 구조를 가지고, 이미 요청중이지만 끝나지 않은 것과 비교해서, 현재의 요청이 이전의 요청에 대해서 조금만 확장을 가해서 만족될 수 있다면, 이전의 요청을 변동시킨다. 이것은 디스크와 같은 블록 디바이스 장치에 대해서 물리적인 움직임을 최소화하기 위한 방법이다. 즉, 연관된 블록의 연산을 묶어서 하나로 만들어 주는 것이다.

블록 디바이스 요청은 나중에 전략 루틴(strategy routine)에 의해서 요청큐에서 하나씩 처리가된다. 전략 루틴을 제공하는 것은 블록 디바이스 드라이버 작성자가 해야할 일이며 나중에 보도록 하겠다. 각각의 블록 디바이스 드라이버는 자신이 관리하는 요청 큐(request queue)를 가지게 되며, 각각의 물리적인 블록 디바이스에 해당하는 요청큐를 하나는 가지고 있어야 한다. 요청 큐에 들어가게되는 블록 디바이스의 요청 구조는 디스크의 효율을 높이는 순으로 정렬되어 블록 디바이스를 효율적으로 사용할 수 있도록 한다. 가령 예를 들어서 디스크의 실린더 수가 증가하는 방향으로 배열을 관리해서 디스크의 헤드가 실린더의 수를 증가시키면서 요청을 처리하도록 만들어준다. 따라서, 디스크 헤드의 움직임이 최소가 된다.

앞에서 설명했듯이, 블록 디바이스에 대한 요구는 요청(request)이라는 자료구조로 형성되며, 이것을 요청 기술자(request descriptor)라고 한다. 요청 기술자는 아래와 같은 필드들을 가지고 있다.

```
struct request {
    struct list_head queue;
    int elevator_sequence;
    struct list_head table;
    struct list_head *free_list;

    volatile int rq_status;          /* should split this into a few status bits */

#define RQ_INACTIVE                (-1)           /* 현재 ACTIVE하지 않다. */
#define RQ_ACTIVE                   1              /* Active한 상태이다. */
#define RQ_SCSI_BUSY                0xffff         /* SCSI에 대한 request가 진행중이다. */
#define RQ_SCSI_DONE                0xfffe         /* SCSI에 대한 request를 끝마쳤다. */
#define RQ_SCSI_DISCONNECTING 0xffe0      /* SCSI가 disconnect되었다. */

    kdev_t rq_dev;
    int cmd;                      /* READ or WRITE */
    int errors;
    unsigned long sector;
    unsigned long nr_sectors;
    unsigned long hard_sector, hard_nr_sectors;
    unsigned int nr_segments;
    unsigned int nr_hw_segments;
```

```

unsigned long current_nr_sectors;
void * special;
char * buffer;
struct semaphore * sem;
struct buffer_head * bh;
struct buffer_head * bhtail;
request_queue_t *q;
elevator_t *e;
};

```

코드 453. 블록 디바이스에 대한 요청(request)구조체.

각각의 필드들이 하는 역할은 아래와 같다.

Field	Descriptor
struct list_head queue	요청들의 연결 리스트
int elevator_sequence	elevator algorithm을 이용하기 위한 sequence 번호
struct list_head table	요청 구조체를 관리하는 테이블의 리스트
struct list_head *free_list	사용되지 않는 요청 큐의 list
volatile int rq_status	요청의 상태
kdev_t rq_dev	요청과 관련된 디바이스의 ID
int cmd	요청된 연산의 종류(read/write)
int errors	에러 코드
unsigned long sector	첫번째 sector번호
unsigned long nr_sectors	요청한 섹터 수
unsigned long hard_sector, hard_nr_sectors	하드웨어의 sector번호와 섹터의 갯수
unsigned int nr_segments	세그먼트의 수
unsigned int nr_hw_segments	하드웨어 세그먼트의 수
unsigned long current_nr_sectors	현재 블록의 섹터 수
void *special	특수 목적으로 사용됨
char *buffer	I/O를 위한 메모리 버퍼
struct semaphore *sem	요청자료 구조를 위한 세마포어
struct buffer_head *bh	요청과 관련된 첫번째 버퍼 헤드의 포인터
struct buffer_head *bhtail	요청과 관련된 마지막 버퍼 헤드의 포인터
request_queue_t *q	관련된 요청 큐를 가리키는 포인터
elevator_t *e	elevator algorithm을 위한 포인터

표 46. 요청 기술자의 필드값

요청 기술자의 필드들 중에서 버퍼 헤드와 관련된 자료구조를 살펴보면, 먼저 각각의 버퍼 헤드는 b_rqnext는 요청큐에서 다음 번에 위치하는 버퍼헤드에 대한 포인터를 가지며, 요청 기술자의 bh와 bhtail은 이러한 버퍼헤드들의 리스트의 앞과 끝을 가리킨다. 따라서, 하나의 요청 기술자는 여러개의 버퍼와 관련을 맺고 있으며, 한번의 블록 디바이스에 대한 연산을 구성하게 된다.

커널은 블록 디바이스에 대한 모든 요청에 대해서 사용할 수 있는 요청 기술자의 수를 정의하고 있으며, 하나의 큐당 가질 수 있는 요청의 갯수는 QUEUE_NR_REQUEST로 256이란 수를 가지고 있다.

블록 디바이스는 버퍼 캐시(buffer cache)에 대한 연산을 위해서, blk_dev_struct구조체와 이의 배열인 blk_dev 구조체를 정의하고 있다. 이와 같은 버퍼 캐시에 대한 설정을 나타내는 자료구조는 디바이스 드라이버가 자신의 것으로 함수의 포인터를 설정해 주어야 한다.

```

struct blk_dev_struct {
    request_queue_t           request_queue;

```

```

queue_proc          *queue;
void               *data;
};

```

먼저 blk_dev_struct 구조체를 보도록 하자. 이는 ~/include/linux/blkdev.h에 정의되어 있다. 먼저 request_queue_t이 오고, queue_proc과 data를 가리키는 포인터가 필드(field)를 정의하는 타입으로 나온다. 각각의 정의는 아래와 같다.

```

typedef struct request_queue request_queue_t;
...
typedef request_queue_t * (queue_proc) (kdev_t dev);
...
struct request_queue
{
    struct list_head      request_freelist[2];
    struct list_head      queue_head;
    elevator_t            elevator;

    request_fn_proc       * request_fn;
    merge_request_fn     * back_merge_fn;
    merge_request_fn     * front_merge_fn;
    merge_requests_fn   * merge_requests_fn;
    make_request_fn      * make_request_fn;
    plug_device_fn        * plug_device_fn;
    void                 * queuedata;

    struct tq_struct      plug_tq;
    char                  plugged;
    char                  head_active;
    spinlock_t             request_lock;
    wait_queue_head_t     wait_for_request;
};

```

필드는 크게 3가지로 나누어진다. 먼저 요청(request) 큐를 관리하기 위한 request_freelist[], queue_head, elevator와, 요청함수 및 merge요청에 대한 처리함수 및 요청을 만들기 위한 함수와 데이터 포인터, 나머지는 동기화를 위한 필드들이다. 이것을 사용해서 디바이스 드라이버에 대한 요청이 큐로 만들어진다. queue_proc은 디바이스 아이디를 입력으로 받아서 요청큐를 둘려주는 함수들에 대한 포인터를 가진다.

여기서 elevator algorithm을 위한 필드로 elevator가 나온다. elevator algorithm은 요청(request)을 순서(sequence)에 따라서 정렬하기 위한 방법으로, 새로운 buffer에 대한 요구가 이전에 있던 요청의 buffer에 정렬되거나 merge될 수 있는지를 확인해서, 요청을 삽입하거나, 새로운 buffer에 대한 요구를 이전의 buffer에 대한 요구와 합치는데 사용한다.

실제 블록 모드 디바이스에 대한 구현은 이제 방금 앞에서 다룬 request들을 어떻게 처리해 줄지를 결정하는 것이 된다. 실제적인 데이터는 버퍼 cache에 들어가게 되며, 이것은 request구조체를 통해서 접근하게 된다.

6.11.1. blk.h 파일

blk.h 파일에는 블록 모드 디바이스 드라이버를 사용하기 위한 많은 값들이 존재한다. 중요한 것은 각 값들에 대해서 major 번호가 index 역할을 해서 접근한다는 점이며, 아래와 같은 것들이 존재한다.

```

static inline void blkdev_dequeue_request(struct request * req)
{
    if (req->e) {

```

```

        req->e->dequeue_fn(req);
        req->e = NULL;
    }
    list_del(&req->queue);
}

...
static inline void end_request(int uptodate) {
    struct request *req = CURRENT;

    if (end_that_request_first(req, uptodate, DEVICE_NAME))
        return;

#ifndef DEVICE_NO_RANDOM
    add_blkdev_randomness(MAJOR(req->rq_dev));
#endif
    DEVICE_OFF(req->rq_dev);
    blkdev_dequeue_request(req);
    end_that_request_last(req);
}

```

코드 454. blk.h파일

blk.h파일에서는 각종 블록 모드 디바이스의 major번호에 대한 선언과 디바이스 드라이버의 request_fn()함수에 대한 선언 및 interrupt handler에 대한 선언들을 찾을 수 있다. 또한 timeout 값과 디바이스에 대한 ON/OFF도 볼 수 있을 것이다. 각각은 아래와 같이 정의된다.

- MAJOR_NR : 이것 이하에서 접근하게될 blk_dev[], blksize_size[]에 대한 index를 얻을 수 있도록 하는 것이다. 따라서, 블록 모드 디바이스 드라이버의 major번호를 가지도록 정의한다.
- DEVICE_NAME : 생성하는 블록 모드 디바이스 드라이버의 이름을 가지도록 정의한다. end_request에서 블록 모드 디바이스의 이름을 printk()함수에서 나타낼 때 사용되고 있다.
- DEVICE_NR(kdev_t device) : 블록 모드 디바이스 드라이버에서 minor번호나 혹은 partition번호와 같은 번호를 얻고자 할때 사용한다. 동일한 physical 디바이스를 사용하는 partition들에 대해서는 같은 수를 돌려주어야 하기 때문에, disk 번호를 돌려주는데도 사용될 수 있다. 또한 request_fn()함수에서 사용하는 CURRENT_DEV를 선언하는데도 사용할 수 있다. 따라서, 현재의 request가 어떤 디바이스 상에서 진행중인지를 알게된다.
- DEVICE_INTR : SET_INTR()이나 CLEAR_INTR()가 같이 사용될 수 있으며, 인터럽트에 대한 bottom half 핸들러를 나타내는 함수의 포인터를 가진다.
- TIMEOUT_VALUE, DEVICE_TIMEOUT : jiffies값으로 timeout값을 설정할 때 사용된다. DEVICE_TIMEOUT은 현재는 사용되고 있지 않다. Timeout값은 지나치게 오래동안 연산을 수행하면 error라고 생각해서 해당하는 timeout handler를 호출하게 될 때 사용할 수 있다.
- DEVICE_NO_RANDOM : end_request()함수에서 사용되며, 시스템에 entropy를 주기 위해서 쓴다. /dev/random과 같이 사용되며, 만약 random 디바이스가 entropy에 큰 영향을 주지 못한다면 이것을 정의해야 할 것이다.
- DEVICE_OFF(kdev_t device) : 마찬가지로 end_request()함수에서 호출되며, 디바이스를 OFF시키는 함수를 가진다. 예를 들어서 floppy같은 경우에는 floppy_off()함수가 호출된다.
- DEVICE_ON(kdev_t device) : 사용되지 않는다.
- DEVICE_REQUEST : 블록 모드 디바이스 드라이버의 request를 처리하는 함수에 대한 포인터를 가진다. 따라서, 해당하는 블록 모드 디바이스를 만들었다면 request를 처리하는 함수로 설정해야 할 것이다.

이것 이외에 위에서 정의한 블록 모드 디바이스를 위한 request를 떼어내는 blkdev_dequeue_request()함수와 블록 디바이스 드라이버에서 하나의 request를 수행한 후에 호출하는

`end_request()`함수가 정의되어 있다. `blk_dequeue_request()`함수는 단순히 request구조체의 elevator algorithm에 정의된 `dequeue_fn()`함수를 호출하는 역할을 하며, `end_request()`함수는 하나의 request를 종료하는 역할을 하고, 다음번 request를 스케줄링 하는 역할을 한다. 따라서, 아직 처리해야 할 request가 있다면, 이것은 나중에 다시 기회가 올 경우에 처리가 될 것이다.

위에서 정의한 매크로의 간단한 사용의 예는 아래와 같다. 예에서 사용한 것은 `~/drivers/block/loop.c`이다.

ex) `loop.c`에서의 macro 정의들

```
#define MAJOR_NR LOOP_MAJOR
#define DEVICE_NAME "loop"
#define DEVICE_REQUEST do_lo_request
#define DEVICE_NR(device) (MINOR(device))
#define DEVICE_ON(device)
#define DEVICE_OFF(device)
#define DEVICE_NO_RANDOM
#define TIMEOUT_VALUE (6 * HZ)
#include <linux/blk.h>
```

즉, `MAJOR_NR`에는 `loop`의 major번호를 `DEVICE_NAME`에는 “loop”를, `DEVICE_REQUEST`는 `do_lo_request`를, `DEVICE_NR()`은 `loop`의 minor번호를 주도록 하고 있다. 또한 `TIMEOUT_VALUE`로는 $6 \times HZ$ 를 주고 있으며, `DEVICE_ON`과 `DEVICE_OFF`에 대해서는 아무것도 정의하지 않고 있다. `DEVICE_ON`과 `DEVICE_OFF`는 정의되지 않을 경우에는 `do{}while(0)`로 선언되어 아무런 일도 하지 않는다. 여기서 주의 해야 할 것은 반드시 이런 매크로들에 대한 정의가 `<linux/blk.h>`의 앞에 나와야 한다는 점이다.

6.11.2. 블록 모드 디바이스 드라이버에서 사용하는 전역 변수들

블록 모드 디바이스를 구현하기 위해서는 위에서 정의한 `blk.h`에 나오는 내용 이외에도, major번호로 접근되는 여러 전역 변수들이 존재한다. 아래와 같은 것들이 있다. 이것들에 대한 정의는 `fs.h`와 `blkdev.h`에서 찾을 수 있으나, `~/drivers/block/ll_rw_blk.c`에서 실제적인 정의가 나온다.

- `blk_size[MAX_BLKDEV]` : `blk_size[major]`로 접근이 가능하며, 모든 블록 모드 디바이스의 크기를 1024 bytes 크기 단위로 나타낸다.
- `blksize_size[MAX_BLKDEV]` : Integer를 가지는 배열에 대한 포인터의 배열로 `blksize_size[major][minor]`로 접근이 가능하다. 모든 블록 디바이스의 크기를 가진다. 즉, 각각의 디바이스에서 사용하는 블록의 크기를 byte단위로 표시한다. 만약 `blksize_size[]`가 NULL값을 가진다면 `BLOCK_SIZE157`를 기본으로 가진다고 본다.
- `hardsect_size[MAX_BLKDEV]` : `hardsect_size[major][minor]`로 접근이 가능하며, 실제 디바이스의 hardware sector크기를 가진다. 기본 값으로 512 bytes를 가정한다.
- `read_ahead[MAX_BLKDEV]` : 디스크에서 얼마나 많은 sector를 read ahead할 수 있는지를 정한다. 즉, 현재 사용되지는 않지만, 사용되리라는 가정하에서 먼저 읽어온 sector의 크기를 정한다. Read ahead방법은 시스템의 성능을 높이는 한 방법이 될 수 있으며, 느린 디바이스를 다룰 경우에는 큰 값을 주는 것이 좋다. 왜냐하면, 한번에 많이 읽어와서 쓰기 때문에 그만큼 디바이스에 대한 접근 횟수를 줄일 수 있기 때문이다. 하지만 이는 또 읽어온 내용을 저장할 buffer cache를 많이 필요하기 때문에 너무 큰 수를 사용하면 시스템 내의 buffer cache를 너무 많이 사용할 가능성이 있다. 따라서, 적정한 크기를 read ahead해야 할 것이다. 모든 minor 번호의 디바이스에 대해서 major당 하나의 read ahead값이 적용된다.
- `max_readahead[MAX_BLKDEV]` : 최대 read ahead할 수 있는 크기를 정한다.
- `max_sectors[MAX_BLKDEV]` : request당 최대 sector들의 수를 정한다.

¹⁵⁷ 1024내지 4096 bytes를 가진다. 블록의 크기는 항상 2의 제곱승 만큼의 크기를 가지도록 정의하는데, 이는 나중에 옵셋을 블록 번호로 변환할 때, shift연산을 사용하기 때문이다. 즉, 한번의 shift는 2로 나누는 것과 같기 때문이다.

이와 같은 값들은 전부 major번호를 가지고 접근되며, 정해주지 않을 경우에는 기본 값이 들어간다. 또한 디바이스 드라이버의 경우 minor번호를 가지고 접근하고자 하는 특정 디바이스를 가리킬 경우에는 major와 minor번호에 대응하는 값을 설정해 주어야 할 것이다. 이것에 대한 실제 사용예는 아래와 같다.

```
hardsect_size[MAJOR_NR] = rd_hardsec; /* Size of the RAM disk blocks */
blksize_size[MAJOR_NR] = rd_blocksizes;           /* Avoid set_blocksize() check */
blk_size[MAJOR_NR] = rd_kbsize;                  /* Size of the RAM disk in kB */
```

위의 예는 ram disk에서 초기화 하는 부분에서 발췌한 것이다. 위에서 정의한 배열들 중에서 `read_ahead[]`는 초기화 하지 않았다. 이와 같은 초기화는 반드시 블록 모드 디바이스에 대한 `open`이전에 해주어야 할 것이다.

이상에서 간단히 블록 디바이스의 구조와 자료구조 및 블록 디바이스 드라이버가 제공해야하는 함수들에 대해서 알아보았으며, 블록 디바이스 드라이버에서 사용하는 각종 전역 변수들에 대해서 알아보았다. 이제는 블록 디바이스 드라이버의 간단한 예를 통해서 실제로 어떻게 구현되고 있는지를 알아볼 차례이다.

6.11.3. 블록 디바이스 드라이버의 예

간단히 블록 모드 디바이스가 어떻게 구현되는지를 보기 위해선 앞장에서 설명했던 블록 모드 디바이스의 등록과, 연산, 그리고, 요청의 처리라는 관점으로 보는 것이 좋다. 예로 둘 디바이스 드라이버는 하드웨어의 특성을 될 수 있으며 따르지 않는 것을 선택하도록 역점을 맞추었으며, 해당하는 것으로는 `~/drivers/block/rd.c`가 있을 수 있다. 즉, ram disk를 구현하는 것이다.

먼저 ram disk에 대한 초기화를 보도록 하자. 아래와 같다.

```
#define MAJOR_NR RAMDISK_MAJOR      /* 1 번으로 ~/include/linux/major.h에 정의 */
...
/* This is the registration and initialization section of the RAM disk driver */
int __init rd_init (void)
{
    int i;

    if (rd_blocksize > PAGE_SIZE || rd_blocksize < 512 ||
        (rd_blocksize & (rd_blocksize-1)))
    {
        printk("RAMDISK: wrong blocksize %d, reverting to defaults\n",
               rd_blocksize);
        rd_blocksize = BLOCK_SIZE;
    }
    if (register_blkdev(MAJOR_NR, "ramdisk", &fd_fops)) {
        printk("RAMDISK: Could not get major %d", MAJOR_NR);
        return -EIO;
    }
    blk_queue_make_request(BLK_DEFAULT_QUEUE(MAJOR_NR), &rd_make_request);
    for (i = 0; i < NUM_RAMDISKS; i++) {
        /* rd_size is given in kB */
        rd_length[i] = rd_size << 10;
        rd_hardsec[i] = rd_blocksize;
        rd_blocksizes[i] = rd_blocksize;
        rd_kbsize[i] = rd_size;
    }
}
```

```

devfs_handle = devfs_mk_dir (NULL, "rd", NULL);
devfs_register_series (devfs_handle, "%u", NUM_RAMDISKS,
                      DEVFS_FL_DEFAULT, MAJOR_NR, 0,
                      S_IFBLK | S_IRUSR | S_IWUSR,
                      &fd_fops, NULL);
for (i = 0; i < NUM_RAMDISKS; i++)
    register_disk(NULL, MKDEV(MAJOR_NR,i), 1, &fd_fops, rd_size<<1);
#endif CONFIG_BLK_DEV_INITRD
/* We ought to separate initrd operations here */
register_disk(NULL, MKDEV(MAJOR_NR,INITRD_MINOR), 1, &fd_fops, rd_size<<1);
#endif
hardsect_size[MAJOR_NR] = rd_hardsec;           /* Size of the RAM disk blocks */
blksize_size[MAJOR_NR] = rd_blocksizes;         /* Avoid set_blocksize() check */
blk_size[MAJOR_NR] = rd_kbsize;                 /* Size of the RAM disk in kB */
/* rd_size is given in kB */
printk("RAMDISK driver initialized:
        \"%d RAM disks of %dK size %d blocksize\n",
        NUM_RAMDISKS, rd_size, rd_blocksizes);
return 0;
}
#endif MODULE
module_init(rd_init);
module_exit(rd_cleanup);
#endif

```

코드 455. rd_init()함수 - RAM disk에 대한 초기화

rd_init()함수는 RAM disk에 대한 초기화를 담당하고 있다. 모듈로 로드될 때 가장 먼저 수행되는 함수이다. 먼저 올바른 블록 크기를 가지는지 확인한다. 만약 올바른 블록 크기를 가지지 않는다면, BLOCK_SIZE로 초기화 시킨다. register_blkdev()함수를 MAJOR_NR로 major 번호를 두고 호출해서 블록 모드 디바이스로 등록 시킨다. 이때 넘겨주는 파일 연산 구조체는 fd_fops이며, 아래와 같이 정의된다.

```

static struct block_device_operations fd_fops = {
    open:          rd_open,
    release:      rd_release,
    ioctl:         rd_ioctl,
};

```

코드 456. RAM disk의 파일 연산 구조체 정의

파일 연산 구조체의 각각에 대한 정의는 아래에서 살펴볼 것이다. 다시 blk_queue_make_request()함수를 호출해서 request에 대한 queue와 request를 만드는 함수를 등록 시킨다. 이때 사용하는 값은 major번호를 가지는 BLK_DEFAULT_QUEUE() 매크로와 rd_make_request()를 사용한다. blk_queue_make_request()함수는 기본으로 제공하는 request를 만드는 함수 대신에 디바이스 드라이버가 제공하는 request 생성(make)함수를 등록 시키는 방법이다. 넘겨주는 값은 request queue에 대한 포인터와 make_request()를 담당하는 함수이다. 또한 BLK_DEFAULT_QUEUE는 아래와 같이 정의된다.¹⁵⁸

```
#define BLK_DEFAULT_QUEUE(_MAJOR)  &blk_dev[_MAJOR].request_queue
```

¹⁵⁸ ~/include/linux/blkdev.h를 참조하라.

따라서, 블록 디바이스의 major 번호를 index로 가지는 request queue를 가리킨다. 따라서, blk_queue_make_request()함수는 등록된 블록 디바이스의 request queue의 make_request_fn()필드를 rd_make_request()로 초기화 시킨다.

이것을 마쳤다면 이젠 각각의 RAM disk에 블록의 길이와 크기, hardware sector의 크기를 초기화 시키는 부분이다. 해당하는 부분은 rd_length[], rd_hardsec[], rd_blocksizes[], rd_kbsize[]이다. 이하의 devfs_XXX는 device file system에 대한 등록을 처리하는 부분이다. 먼저 directory entry를 만들고, 이하에 각각의 RAM disk에 대해서 하나의 device file을 등록한다. 이것에 대한 것은 여기서는 설명하지 않도록 한다. 알고 싶다면, ~/fs/devfs 이하를 참조하기 바란다. register_disk()¹⁵⁹ 함수는 하나의 디스크로 등록 시켜주는 함수이다. 첫번째 함수의 인자가 NULL인 경우에는 곧바로 return한다. 따라서, RAM disk에 대해서는 아무런 일도 일어나지 않는다.

이하의 부분은 RAM disk에서 사용할 각종 전역 변수에 대한 초기화이다. 이에 대한 것은 앞에서 이미 블록 모드 디바이스 드라이버에 대해서 볼 때 이야기 했었다. hardsect_size[MAJOR_NR]는 rd_hardsec으로, blksize_size[MAJOR_NR]는 rd_blocksizes로, blk_size[MAJOR_NR]는 rd_kbsize로 초기화 한다.

RAM disk의 cleanup에 해당하는 것은 rd_cleanup()함수이다. 아래와 같이 정의된다.

```
#ifdef MODULE
/* Before freeing the module, invalidate all of the protected buffers! */
static void __exit rd_cleanup (void)
{
    int i;

    for (i = 0 ; i < NUM_RAMDISKS; i+ + ) {
        if (rd_inode[i]) {
            /* withdraw invalidate_buffers() and prune_icache() immunity */
            atomic_dec(&rd_inode[i]->i_bdev->bd_openers);
            /* remove stale pointer to module address space */
            rd_inode[i]->i_bdev->bd_op = NULL;
            iput(rd_inode[i]);
        }
        destroy_buffers(MKDEV(MAJOR_NR, i));
    }
    devfs_unregister (devfs_handle);
    unregister_blkdev( MAJOR_NR, "ramdisk" );
    hardsect_size[MAJOR_NR] = NULL;
    blksize_size[MAJOR_NR] = NULL;
    blk_size[MAJOR_NR] = NULL;
}
#endif
```

코드 457. rd_cleanup()함수 – RAM disk에 대한 종료 함수

이 함수는 이전에 rd_init()에서 행했던 일을 원래 상태로 복구하는 역할을 맡고 있다. 간단히 보면 먼저 RAM disk와 관련된 inode들에 대한 open 카운터의 감소와 inode의 block디바이스에 대한 연산에 대한 벡터를 NULL로 놓고, 해당 inode를 버린다(iput). 또한 destroy_buffers()를 호출해서 해당 major의 버퍼를 없앤다.

devfs_unregister()함수를 호출해서 device file system에 등록된 entry를 해제하고, 다시 unregister_blkdev()를 호출해서 블록 디바이스를 unregister한다. 마지막으로 hardsect_size[], blksize_size[], blk_size[]를 NULL로 만든다.

¹⁵⁹ ~/fs/partitions/check.c를 참조할 것

위의 코드들에서 다루지 않는 것은 파일 연산 구조체와 inode에 등록되는 블록 디바이스를 위한 파일 연산 구조체, 그리고, `make_request()`함수 및 `request`를 처리하는 `request_fn()`함수가 남았다. 하나 하나씩 정리해 보도록 하자.

먼저 `fd_fops`파일 연산 구조체부터 보기로 하자. 해당하는 것은 `rd_open()`, `rd_release()`, `rd_ioctl()`이다. `rd_open()`은 아래와 같이 정의되어 있다.

```
static struct inode *rd_inode[NUM_RAMDISKS]; /* Protected device inodes */
...
static int rd_open(struct inode * inode, struct file * filp)
{
#ifndef CONFIG_BLK_DEV_INITRD
    if (DEVICE_NR(inode->i_rdev) == INITRD_MINOR) {
        if (!initrd_start) return -ENODEV;
        initrd_users++;
        filp->f_op = &initrd_fops;
        return 0;
    }
#endif
    if (DEVICE_NR(inode->i_rdev) >= NUM_RAMDISKS)
        return -ENXIO;
    /*
     * Immunize device against invalidate_buffers() and prune_icache().
     */
    if (rd_inode[DEVICE_NR(inode->i_rdev)] == NULL) {
        if (!inode->i_bdev) return -ENXIO;
        if ((rd_inode[DEVICE_NR(inode->i_rdev)] = igrab(inode)) != NULL)
            atomic_inc(&rd_inode[DEVICE_NR(inode->i_rdev)]->i_bdev-
>bd_openers);
    }
    MOD_INC_USE_COUNT;
    return 0;
}
```

코드 458. `rd_open()`함수

`rd_open()`함수는 디바이스에 대한 `open()` 파일 연산을 처리하는 함수이다. 만약 `CONFIG_BLK_DEV_INITRD`(Initial RAM disk)로 설정이 되었다면, 먼저 해당하는 `inode`에서 디바이스의 minor번호를 가져와서(`DEVICE_NR()`), 이것이 `INITRD_MINOR`와 같은지를 확인한다. 같다면 `initrd_start`가 설정되었는지를 확인한 후, 아직 시작(`start`)되지 않았다면 `-ENODEV`를 돌려주고, 만약 설정되었다면, `initrd_users`를 증가시킨후 해당하는 파일 연산에 대해서 `initrd_fops`를 설정하고 복귀한다.

`Inode`의 minor device의 필드값을 확인하고, 이 값이 RAM 디스크의 수보다 크다면 `-ENXIO`를 돌려주고 복귀하며, 그렇지 않을 경우에는 이하의 부분을 수행한다. 만약 RAM disk에서 사용하는 `inode`들에 대한 배열을 가지는 `rd_inode[]`에서 minor번호에 해당하는 값이 `NULL`값을 가질 경우 다음과 같이 수행한다. 먼저 `inode`가 가리키는 `i_bdev`에 `NULL`을 가질 경우 해당하는 블록 디바이스가 없는 경우로 `-ENXIO`를 돌려주고 복귀한다. 또한 `rd_inode[]`에서 minor 번호에 해당하는 값에 `inode`를 하나를 가져다(`igrab()`) 두고 이 값이 `NULL`인지를 비교한다. 만약 `NULL`이 아니라면, minor번호에 해당하는 `rd_inode`의 `i_bdev->bd_openers`의 값을 하나 증가 시킨다. 현재 이 `inode`를 사용하는 블록 디바이스가 하나 추가 되었다는 뜻이다.

이전 복귀전에 모듈에 대한 사용카운터를 증가시킨 후 복귀코드로 0으로 주고 돌아간다(`return`).

```
static int rd_release(struct inode * inode, struct file * filp)
{
```

```

    MOD_DEC_USE_COUNT;
    return 0;
}

```

코드 459. rd_release()함수

rd_release()함수는 단순히 모듈의 사용 카운터만 감소시킨다. rd_open()에서 사용되었던 rd_inode[]에 대한 것은 모듈의 cleanup 함수에서 처리하기 때문이다. 또한 initial RAM disk 역시 각각의 해당 파일 연산자가 있으므로 이곳에서 다를 만한 것이 되지 못한다.

```

static int rd_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
    unsigned int minor;

    if (!inode || !inode->i_rdev)
        return -EINVAL;
    minor = MINOR(inode->i_rdev);
    switch (cmd) {
        case BLKFLSBUF:
            if (!capable(CAP_SYS_ADMIN))
                return -EACCES;
            /* special: we want to release the ramdisk memory,
               it's not like with the other blockdevices where
               this ioctl only flushes away the buffer cache. */
            if ((atomic_read(&inode->i_bdev->bd_openers) > 2))
                return -EBUSY;
            destroy_buffers(inode->i_rdev);
            rd_blocksizes[minor] = 0;
            break;
        case BLKGETSIZE: /* Return device size */
            if (!arg) return -EINVAL;
            return put_user(rd_kbsize[minor] << 1, (long *) arg);
        case BLKROSET:
        case BLKROGET:
        case BLKSSZGET:
            return blk_ioctl(inode->i_rdev, cmd, arg);
        default:
            return -EINVAL;
    };
    return 0;
}

```

코드 460. rd_ioctl()함수

rd_ioctl()함수가 하는 일은 상위의 ioctl()함수가 호출한 일을 처리하는 일이다. 먼저 inode와 inode의 i_rdev가 NULL이 아닌지를 확인한 후, NULL이라면 -EINVAL을 돌려준다. inode를 이용해서 해당하는 블록 모드 디바이스의 minor번호를 가져온 다음(MINOR()), 명령(command)로 주어진 일을 처리한다. default값으로는 -EINVAL을 돌려준다. 처리하는 명령에는 BLKFLSBUF와 BLKGETSIZE, BLKROSET, BLKROGET, BLKSSZGET등이 있으며, BLKFLSBUF는 버퍼 캐쉬에 대한 flush연산을 수행하며, BLKGETSIZE는 block의 크기를 돌려준다. 나머지는 blk_ioctl()함수를 호출해서 처리하도록 한다. 먼저 BLKFLSBUF는 CAP_SYS_ADMIN capability를 현재의 프로세스가 가졌는지를 확인한 다음, 그렇지 못할 경우에는 -EACCESS를 돌려준다. 현재 inode의 i_bdev->bd_openers가 2보다 큰 값을 가진 경우에는 이미

다른 곳에서 사용중이므로 -EBUSY를 돌려주고, 그렇지 않을 경우에는 inode의 i_rdev에서 사용중인 buffer cache들을 버린다(destroy_buff()). 이것을 마치면, rd_blocksizes[]의 minor번호에 해당하는 필드를 0으로 초기화 시킨 후 복귀한다. BLKGETSIZE는 현재 block의 크기를 원하므로 rd_kbsize[]의 minor번호에 해당하는 값을 사용자 프로세스의 arg필드로 복사한 후 복귀한다.

blk_ioctl()함수는 블록 모드 디바이스의 common ioctl() 메쏘드에 해당하는 함수로 이곳에서는 자신만의 해당하는 사항을 처리한 후, 나머지 것들에 대해서 기본적으로 정의된 블록 모드 디바이스의 ioctl()메쏘드를 호출할 목적으로 사용하고 있다. 관련된 것을 보고 싶다면, ~/drivers/block/blkpg.c를 참조하라.

```
static int rd_make_request(request_queue_t * q, int rw, struct buffer_head *sbh)
{
    unsigned int minor;
    unsigned long offset, len;
    struct buffer_head *rbh;
    char *bdata;

    minor = MINOR(sbh->b_rdev);
    if (minor >= NUM_RAMDISKS)
        goto fail;
    offset = sbh->b_rsector << 9;
    len = sbh->b_size;
    if ((offset + len) > rd_length[minor])
        goto fail;
    if (rw==READA)
        rw=READ;
    if ((rw != READ) && (rw != WRITE)) {
        printk(KERN_INFO "RAMDISK: bad command: %dWn", rw);
        goto fail;
    }
    rbh = getblk(sbh->b_rdev, sbh->b_rsector/(sbh->b_size>>9), sbh->b_size);
    bdata = bh_kmap(sbh);
    if (rw == READ) {
        if (sbh != rbh)
            memcpy(bdata, rbh->b_data, rbh->b_size);
    } else
        if (sbh != rbh)
            memcpy(rbh->b_data, bdata, rbh->b_size);
    bh_kunmap(sbh);
    mark_buffer_protected(rbh);
    brelse(rbh);
    sbh->b_end_io(sbh,1);
    return 0;
fail:
    sbh->b_end_io(sbh,0);
    return 0;
}
```

코드 461. rd_make_request()함수

RAM disk을 구현하는 rd.c에서는 자신만의 make_request함수를 구현하고 있다. 이것이 바로 rd_make_request함수이다. 이것은 request들이 queue를 이루는 request queue구조체의 make_request_fn()

함수의 포인터를 자신이 구현하는 `make_request_fn()`으로 가르키도록 초기화했기에 가능하다. 커널은 블록 디바이스에 대한 요청을 만들 때 이 함수를 참조할 것이다. 코드를 보도록 하자.

먼저 넘겨받은 버퍼헤드의 `b_rdev`에서 해당하는 `minor` 번호를 가져온다. 만약 이것이 RAM disk의 수보다 크게 되면 곧장 `fail`로 제어를 옮긴다. 그리고 나서 어느 `offset` 값을 구하기 위해서 버퍼헤드에 들어 있는 `sector` 번호에 512 곱한다(`sbh->b_rsector << 9`). 또한 길이 값을 `sbh->b_size`로 초기화하고(`len`), `offset`과 `len`을 합쳐서 이것이 해당하는 `minor` 번호의 `rd_length[]`보다 큰지를 확인한 후, 만약 더 크다면 `fail`로 제어를 옮긴다. 넘겨받은 요청의 `type`이 `read ahead(READA)`라면, 다시 요청의 `type`을 `READ`로 놓는다. 요청의 `type`이 `READ`도 아니고, `WRITE`도 아니라면 잘 못된 요청을 한 것으로 보고 제어를 다시 `fail`로 옮긴다. 이전 직접적인 버퍼 헤드에 대한 처리가 남았다. 먼저 버퍼 캐쉬에 이미 해당하는 버퍼가 있는지를 확인하는 `getblk()` 함수를 호출한다. `getblk()` 함수는 버퍼 캐쉬를 뒤져서 만약 해당하는 버퍼가 있다면 이것의 버퍼 헤드를 돌려줄 것이며, 없다면 새로운 버퍼를 `free list`에서 할당받은 후, 초기화해서 버퍼 헤드를 돌려줄 것이다.

`bh_kmap()`과 `bh_kunmap()` 함수¹⁶⁰는 버퍼 헤드의 `data`를 가리키는 필드를 `mapping`하고 `unmapping`하는 함수이다. 따라서, `bh_kmap()` 함수를 호출해서 해당 페이지에서 `data`를 가리키는 부분에 대한 `offset`을 더해서 실제로 버퍼의 데이터가 기록될 위치에 대한 포인터를 반환한다. 이전 요청이 `READ`나 `WRITE`냐에 따라서 영역의 데이터를 버퍼로 옮기던가 아니면 넘겨받은 데이터를 버퍼 캐쉬에 복사하는 일만이 남았다(`memcpy()`). 이것을 마치면, `mark_buffer_protected()`를 호출해서 버퍼 캐쉬에 대해서 `protect(보호)`를 설정한다. 이것을 마치면 `brelse()`를 호출해서 버퍼가 `DIRTY`인지 확인한 후, 만약 `DIRTY` 상태라면 버퍼 헤드의 `b_flushtime` 필드를 설정해서 `flush` 될 시간을 설정한다.

함수를 복귀하기 전에 `b_end_io()` 함수를 버퍼헤드와 1 혹은 0을 인수로 사용해서 호출한다. 이것은 `b_end_io()` 함수가 I/O completion 함수를 가지기 때문이며, 1이나 0은 데이터의 전송이 성공인지 실패인지를 나타내는 값으로 사용된다. `b_end_io()`가 호출되면, 버퍼헤드의 `BH_Uptodate` 필드가 성공인지 실패인지를 나타내는 값으로 설정되고, 다시 `BH_Lock`를 풀어서 `b_wait`에서 잠들어 있는 프로세스들을 깨운다.

이상에서 RAM disk를 만드는데 핵심적인 역할을 하는 함수들에 대한 것을 살펴보았다. RAM disk에서는 `request`에 대한 처리를 buffer cache에서 처리하기에 `request_fn()`에 대한 것은 보이지 않는다. 또한 `check_media_change()`와 `revalidate()` 함수는 RAM disk에 대해서는 해당 사항이 없으므로 정의되지 않았다. 이제 RAM disk에서 남은 것은 `initial RAM disk`의 구현이 될 것이다. 즉, 앞에서 보았던 코드 중에서 `rd_open()` 함수에서 파일 연산자에 대해서 `initial RAM disk`가 구현되었다면, `initrd_fops` 파일 연산 구조체가 사용될 것이다. 아래와 같은 정의를 가진다.

```
#ifdef CONFIG_BLK_DEV_INITRD
...
static struct file_operations initrd_fops = {
    read:           initrd_read,
    release:       initrd_release,
};
#endif
```

코드 462. `initrd_fops` 파일 연산 구조체의 정의

즉, `initrd_read()`와 `initrd_release()`가 파일 연산 구조체로 정해져 있으며, 아래와 같다.

```
#ifdef CONFIG_BLK_DEV_INITRD
unsigned long initrd_start, initrd_end;
...
static ssize_t initrd_read(struct file *file, char *buf,
                           size_t count, loff_t *ppos)
{
```

¹⁶⁰ [~/include/linux/highmem.h](#)를 참조하기 바란다.

```

int left;

    left = initrd_end - initrd_start - *ppos;
    if (count > left) count = left;
    if (count == 0) return 0;
    copy_to_user(buf, (char *)initrd_start + *ppos, count);
    *ppos += count;
    return count;
}

...
static int initrd_users;
...

static int initrd_release(struct inode *inode, struct file *file)
{
    extern void free_initrd_mem(unsigned long, unsigned long);

    lock_kernel();
    if (!--initrd_users) {
        blkdev_put(inode->i_bdev, BDEV_FILE);
        iput(inode);
        free_initrd_mem(initrd_start, initrd_end);
        initrd_start = 0;
    }
    unlock_kernel();
    return 0;
}
#endif

```

코드 463. `initrd_read()`함수와 `initrd_release()`함수

`initrd_read()`함수는 initial RAM disk에 들어있는 데이터를 읽는 역할을 하는 함수로, `copy_to_user()`함수를 사용해서 사용자모드 프로세스의 주소영역에 데이터를 카피하고 있다. 돌려주는 값은 얼마나 많은 데이터를 복사했는지를 알려주는 `count`값이다. 만약 더 이상 남은 공간이 없을 경우에는, 즉, `initrd_end`와 `initrd_start`가 같은 값을 가질 경우에는 0을 돌려준다.

`initrd_release()`함수는 먼저 커널에 `lock`을 설정한 후(`lock_kernel()`), `initrd_users`(initial RAM disk의 사용자 카운터)를 감소시키고 만약 0값을 가진다면, 해당하는 `inode`에 관련된 버퍼들에 대해서 `sync`¹⁶¹를 한 후, 블록 모드 디바이스의 `release` 메쏘드가 있을 경우에는 이를 실행한다. 그리고나서, 이젠 더 이상 `inode`를 접근할 일이 없으므로 해당하는 `inode` 객체를 버리고(`iput`), `free_initrd_mem()`을 `initrd_start`와 `initrd_end`를 인자로 넘겨서 호출한다. `initrd_start`는 0으로 설정해서 현재 initial RAM disk가 사용되지 않는다는 것을 나타낸다. 마지막으로 복귀전에 커널에 대한 `lock`을 해제하고 0을 돌려준다. 여기서 `free_initrd_mem()`함수는 `~/arch/i386/mm/init.c`에 정의된 함수로 해당하는 메모리 영역의 페이지를 해제하는 역할을 한다.

나머지 initial RAM disk내의 함수 및 변수들은 시스템이 초기화 시에 호출되는 것들로, 커널에서는 `.text.init section`에 위치시켜서 초기화에 필요한 동작을 수행한 후, 더이상 필요없게 되면 차지하고 있던 메모리 영역을 해제한다.

이상에서 우린 간단한 RAM disk의 구현에 대해서 살펴보았다. 구현의 세부 사항은 좀더 코드를 참조할 필요하가 있지만, 일단 우리가 만들고자 한 블록 모드 디바이스의 기본적인 구조를 이해하는데는 많은 도움이 되었으리라 생각한다. 한가지 덧붙일 것은 모듈로서 구현되는 것이 항상 디바이스 드라이버에만 한정되지는 않는다는 점이다. 즉, 파일 시스템이나 기타 커널의 한 구성요소(component)도 모듈로서

¹⁶¹ 메모리에 있는 내용과 실제적으로 physical 디스크에 있는 내용이 일치하도록 만든다.

구현이 될 수 있다. 따라서, 모듈이라는 것은 하나의 디바이스 드라이버 구현 방법에 불과하며, 커널의 다른 구현에서도 찾을 수 있다. 모듈과 디바이스 드라이버를 혼동하지 말도록 하자.

이젠 디바이스 드라이버의 또 다른 형태인 Network 디바이스 드라이버에 대해서 보기로 하자. 지금까지의 이야기와는 전혀 상관이 없다고 봐도 된다. 다만 한가지 명심할 점은 비슷한 커널과 디바이스 드라이버의 인터페이스가 설정되어야 하며, 또한 네트워크 디바이스 드라이버만의 데이터 구조체를 가지고 있다는 점이다.

6.12. Network Device Driver

리눅스에서의 network device driver는 data의 수신과 전송을 담당한다. 실제적인 network device를 control하며, 상위 protocol layer에서 내려오는 data를 하위의 physical layer에 전달해 준다. 또한, 하위의 physical layer에서 올라오는 data를 protocol layer에서 인식할 수 있도록 적절한 연산을 하는 기능도 들어가 있다.

Ethernet device의 경우에는 device의 검출 시에 `~/drivers/net/net_init.c`에 있는 `init_etherdev()`를 호출하게 되고, 다시 이 함수는 `init_netdev()` 함수를 호출하게 되어 network device로 등록된다. 등록은 `register_netdev()` 함수가 맡고 있다. 함수를 호출할 때, 미리 device driver에서 고유하게 사용할 data 구조를 parameter값으로 넘겨줌으로써, 나중에 device driver routine이 호출될 때, 이 구조체에 담긴 정보를 토대로 연산을 수행하게 된다. 이러한 구조체의 값으로 사용될 수 있는 것으로는 Buffer들에 대한 description과 device driver의 고유한 정보가 있겠다.

자 그럼 이젠 kernel과 device driver 간의 interface에 대해서 살펴보도록 하자. Kernel은 명확히 정의된 interface를 통해서 device에 접근을 시도하게 되며, 따라서 device driver에서는 이러한 kernel이 알고 있는 interface를 제공하여야만 한다. 우리가 만들려고 하는 network device driver에 대한 interface로는 다음과 같은 것들이 있다. 이것을 기초로 해서 device driver의 기본 구성이 만들어 진다.

- Open – network device에 대해서 open을 하게 된다. 이곳에서 대부분의 hardware initialization 이 일어나게 된다.
- hard_start_xmit – data의 전송요구가 있을 경우에 호출되며, data를 bus상에 놓기 위한 기본적인 일을 행한다.
- stop – device에 대한 더 이상의 요구가 없을 때 kernel에 의해 호출된다. Device를 close하는 시점이 될 수 있을 것이다. Close를 하기 전에 사용하고 있던 resource에 대한 것들을 release한다.
- get_stats – device에 대한 statistics 정보를 구하기 위해서 kernel에 의해 호출된다. Network device driver는 당연히 이것과 관련된 정보를 수집해 주어야 할 것이다.
- set_multicast_list – multicast address를 setting하기 위해서 kernel에 의해 호출되며, 우리가 만들고자 하는 device에는 이것을 위한 register를 가지고 있다.¹⁶² 즉, device driver에서 이곳에 multicast address를 보관하거나 새로이 update하는 연산을 해주게 된다.
- do_ioctl – ioctl에 대한 요구를 처리하는 부분으로 우리가 만들고자 하는 network device driver에서는 별다른 일을 해주지 않는다. 다만 function entry는 채워주어야 한다.

¹⁶² CAM(Contents Addressable Memory)라는 것을 가지고 multicast address에 대한 요구를 처리한다.

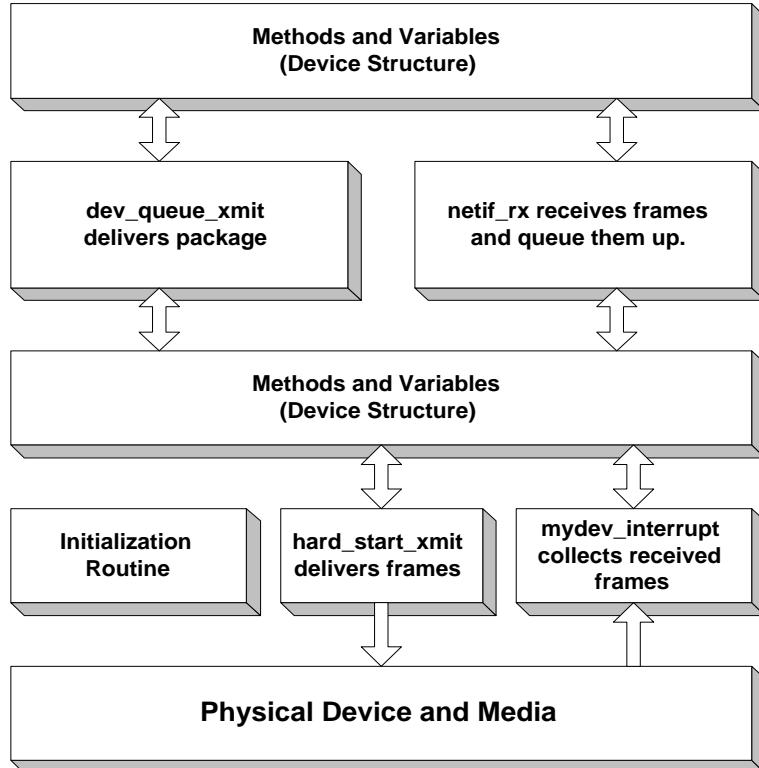


그림 61.Network Device Driver의 기본 구조

[그림54]는 network device driver의 기본구조를 보여준다. 여기서 중요한 것은 이러한 device driver에 대한 interface routine들은 전부 device structure에 이것을 가르키는 pointer를 저장해서 kernel에 알려주는 것으로 kernel은 새로운 network device가 있다는 것을 알게 되며, 나중에 kernel은 network device driver에 대한 요구가 있을 때, device structure를 찾아서 해당하는 function을 access하게 된다.

6.12.1. Ethernet에 대한 이해

Ethernet이란 1970년대에 Xerox에서 개발한 근거리의 packet교환 network을 말하는 것으로, IEEE 802.3 표준으로 정해져 있다. 현재는 가장 인기 있는 LAN 상에서 사용하는 기술이며, 10Mbps나 100Mbps로 data를 전송한다. CSMA/CD(Carries Sense Multiple Access/Collision Detection)방식으로 작동하는데, 이것은 전송미디어에 다른 host에서 packet을 보내고 있을 경우 충돌(collision)이 있음을 보내고자 하는 host측에서 알게 되며, 얼마간의 간격을 두고 다시 보내려는 시도를 하게 된다는 것이다. 따라서, 전송속도가 100Mbps라고 하더라도 traffic이 많을 경우에는 이러한 속도가 나오지 못한다.

보통이 LAN환경에서는 Ethernet card를 이용해서 network를 구성한다. Ethernet에서는 주소를 지정하는 방법으로 6byte의 card에 고유한 번호를 사용한다. 이렇게 할당된 address는 IP address와 함께, network상에서 system을 찾을 수 있도록 한다. [그림55]는 Ethernet packet의 구성을 보여준다. 또한 TCP/IP와 ethernet에서의 encapsulation에 대해서도 보여준다. 즉, TCP/IP로부터 넘겨받은 packet data를 다시 Ethernet packet으로 만들어서 network에 날려 보내게 된다.

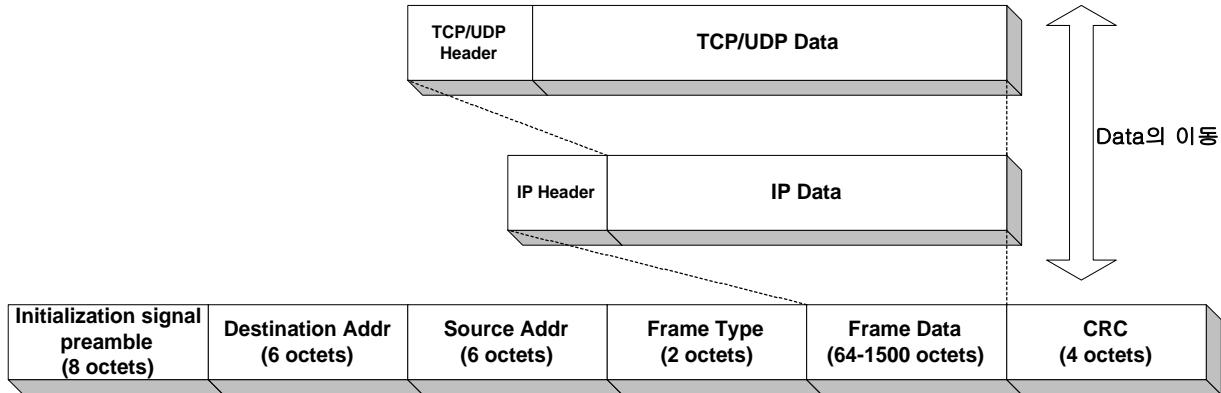


그림 62. Ethernet Frame Structure

[그림55]를 보면, 보내고자 하는 상위의 protocol data packet들은 frame data부분에 들어가서 전송되며, 이것은 physical media의 성질에 의해서 상위의 protocol에는 상관없이 data를 보내고 받기 위한 것이다.

Ethernet address는 network card vendor에 고유하게 할당되어 사용되며, ARP/RARP protocol에 의해서 IP address와 mapping되어 사용된다. 이러한 ethernet address중 특정의 목적으로 사용되는 address가 있는데 이러한 것으로는 multicast address와 broadcast address 등이 있다.

Multicast address는 주어진 multicast address를 허락한 network내의 몇몇 컴퓨터들에게만 전달되는 것이며, 이러한 multicast에 참여한 host들을 통틀어서, multicast group이라고 한다. Multicast group에 참여하기 위해서는 먼저 자신의 host에게 network interface에서 multicast address를 허락하라는 명령을 보내야 하며, Ethernet card에서는 이것을 자신에게 보내지는 모든 packet의 destination address와 비교하여, 자신에게 전송되는 packet인가를 찾게 되는 것이다. Broadcast address는 모든 network내의 host들이 다 packet을 받아보도록 하는 주소로서 사용되며, 이것은 network의 traffic을 증가시키게 할 수 있다.

6.12.2. Protocol이란?

Protocol이란 서로간에 합의한 통신 방법이다. 이러한 것 중에서 가장 많이 사용되는 것으로는 TCP/IP가 있으며, [그림56]에서와 같이 TCP/IP는 ethernet위에 올라가게 된다. Kernel은 이러한 TCP/IP protocol을 구현하고 있다. 물론 TCP/IP가 반드시 ethernet위에만 올라가야 한다는 것은 아니다. TCP는 연결 지향적인 service를 제공하며, 이와는 다른 UDP는 비연결적인 service를 제공하기 위해서 존재한다. IP는 이러한 두개의 protocol계층에서 넘겨받은 data가 상대편으로 정확히 전달되는 것을 보장하는 역할을하게 된다. Protocol의 실제적인 구현은 process와 queue로 되어 있는 경우가 많다. 즉, 날아오는 packet이나 보내고자 하는 packet이 있을 경우, 이것은 하나의 실행중인 process에 대한 input queue로 들어가게 되며, 이곳에서 process의 처리를 기다려, 하위나 상위의 protocol data unit으로 queue로 전달 되게 된다.

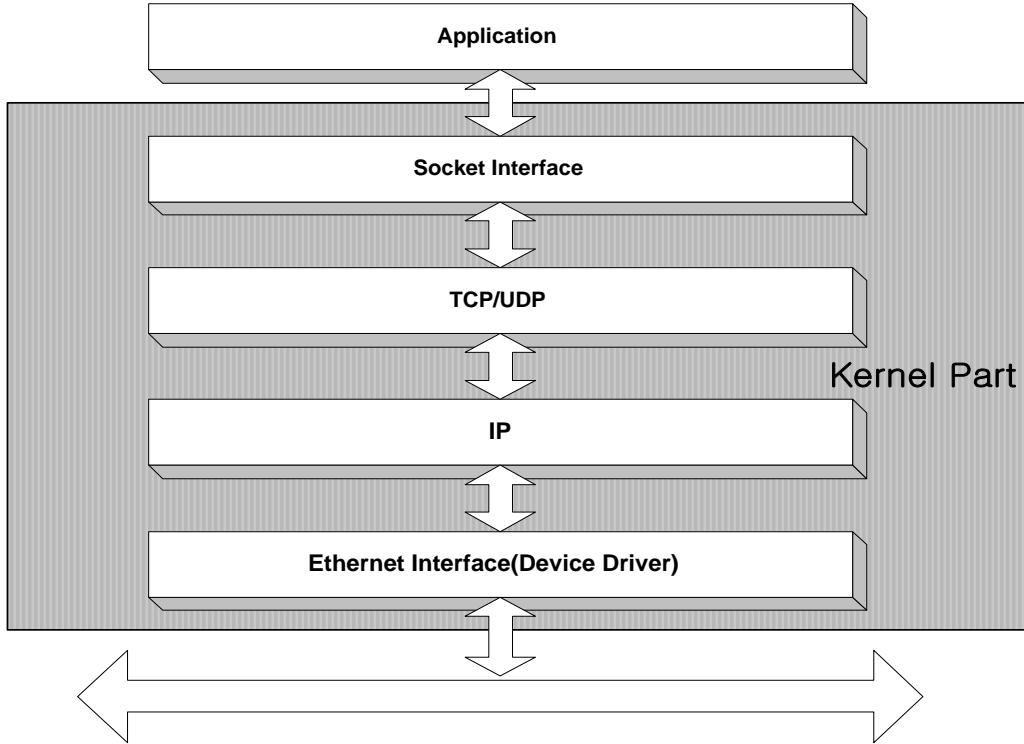


그림 63. Protocol Stack 및 interface

[그림56]은 TCP/IP상에서의 protocol stack과 application interface를 보여주고 있다. TCP/IP의 상위에는 application과의 interface를 제공하기 위해서 socket을 정의하고 있으며, 이것은 BSD UNIX상에서 먼저 구현되었으며, 다른 system에서도 채택되어 현재는 일반적인 이야기가 되었다. 이것에 대해서는 조금 후에 좀더 살펴보도록 한다.

6.12.3. Socket이란? 그리고, socket buffer란?

Socket이란 API로서 TCP/IP로의 연결통로를 제공한다고 생각하면 된다. 즉, socket을 통해서 application은 data를 network를 통해서 목적지로 전송할 수 있다. 이것은 주로 library로 제공되며, program을 compile하게 될 때 link된다. 여기서는 socket programming에 대한 것은 생략하도록 했다. 이것은 너무나 많은 책들에서 다루고 있기에 그것들을 참고하기 바란다.

Socket에서 사용하는 data들은 socket buffer라는 형태로 만들어져서 linux에서 제공하는 protocol stack의 처리를 받게 되며, 최종적으로 network device driver에 도달하게 된다. 따라서, network device driver writer는 socket buffer에 대한 것만 생각하면 된다. 나중에 data를 상위의 protocol stack으로 전송하게 될 때도 사용하게 되는 것이 바로 이러한 socket buffer이다.

먼저 socket buffer에 대한 field중 다음과 같은 것에 대해서 알아보자.

- struct device *dev – socket buffer가 전달되거나 보내진 device를 가르킨다.
- __u32 saddr, __u32 daddr, __u32 raddr – IP protocol에 의해서 사용되며, source, destination, router의 주소를 가르킨다.
- unsigned char *head, unsigned char *data, unsigned char *tail, unsigned char *end – network를 통해서 전달되는 packet의 data를 가르키는 부분이다.
- unsigned long len – data 자체의 길이를 나타내는 것으로 skb->tail – skb->head를 가진다.
- unsigned char ip_summed – TCP/IP에 의해서 checksum을 하는데 사용된다. 이것은 receive한 packet에 대해서 device driver에 의해서 set된다.
- unsigned char pkt_type – PACKET_HOST, PACKET_BROADCAST, PACKET_MULTICAST, 혹은 PACKET_OTHERHOST등의 값으로 driver에 의해서 set된다. 즉, 받은 packet이 host로 오는 것인지, broadcast된 것인지, 혹은 multicast 된 것인지를 나타낸다. 하지만, ethernet driver의

경우에는 `eth_type_trans`가 이 역할을 해주므로 따로 나타내어줄 필요는 없다.

- Union { unsigned char *raw; [...] } mac – packet을 receive할 때 set되며 packet을 처리하는 데 사용된다. 위에서 언급한 것과 마찬가지로 `eth_type_trans`가 처리를 해주므로 ethernet에서는 다를 필요가 없다.

이것 이외에도 여러 개의 field가 socket buffer에 있지만 관심을 가질 필요는 없다. 이것은 우리가 다른 어 주지 않아도 상관없다.

Socket buffer에서 사용하게 되는 function으로는 다음과 같은 것이 있다.

- `struct sk_buff *alloc_skb(unsigned int len, int priority);` – socket buffer를 할당한다.
- `struct sk_buff *dev_alloc_skb(unsigned int len);` – socket bufer를 할당하며, priority로 GFP_ATOMIC을 주고, head와 tail사이에 16bytes를 남겨둔다.
- `void kfree_skb(struct sk_buff *skb, int rw);` – 할당된 socket buffer를 free한다.
- `void dev_kfree_skb(struct sk_buff *skb, int rw);` – 할당된 socket buffer를 free하며, buffer의 locking을 정확히 처리한다. Driver에서는 `dev_kfree_skb()`를 사용해서 socket buffer를 free해주어야 한다.
- `unsigned char *skb_put(struct sk_buff *skb, int len);` – buffer의 끝에 data를 넣고, tail과 len field를 고친다.
- `unsigned char *skb_push(struct sk_buff *skb, int len);` – socket buffer의 앞에서부터 data를 채워넣는다.
- `int skb_tailroom(struct sk_buff *skb);` – socket buffer에 남은 data를 위한 space의 크기를 return한다.
- `int skb_headroom(struct sk_buff *skb);` – data앞에 남은 부분의 사용 가능한 space의 크기를 return한다.
- `void skb_reserve(struct sk_buff *skb, int len);` – socket buffer에 데이터를 쓰기 전에 headroom에 space를 마련한다.
- `unsigned char * skb_pull(struct sk_buff *skb, int len);` – packet의 head로부터 data를 빼어낸다.

위와 같은 함수들을 사용해서 kernel로부터 넘겨받은 socket buffer에 대한 연산을 하며, 또한 받은 packet에 대해서 kernel로 넘겨주기 전에 socket buffer의 형태로 만들어 준다. 즉, kernel에서는 보내고자 하는 data를 socket buffer의 형태로 만들어주며, 넘겨 받는 것은 socket buffer라는 것을 가정할 수 있게 된다.¹⁶³[그림57]은 socket buffer가 어떻게 구현되어 있는지를 보여주고 있다.

¹⁶³ SCO unix에서의 network driver는 stream이라는 것으로 이와 같은 일을 해주고 있다.

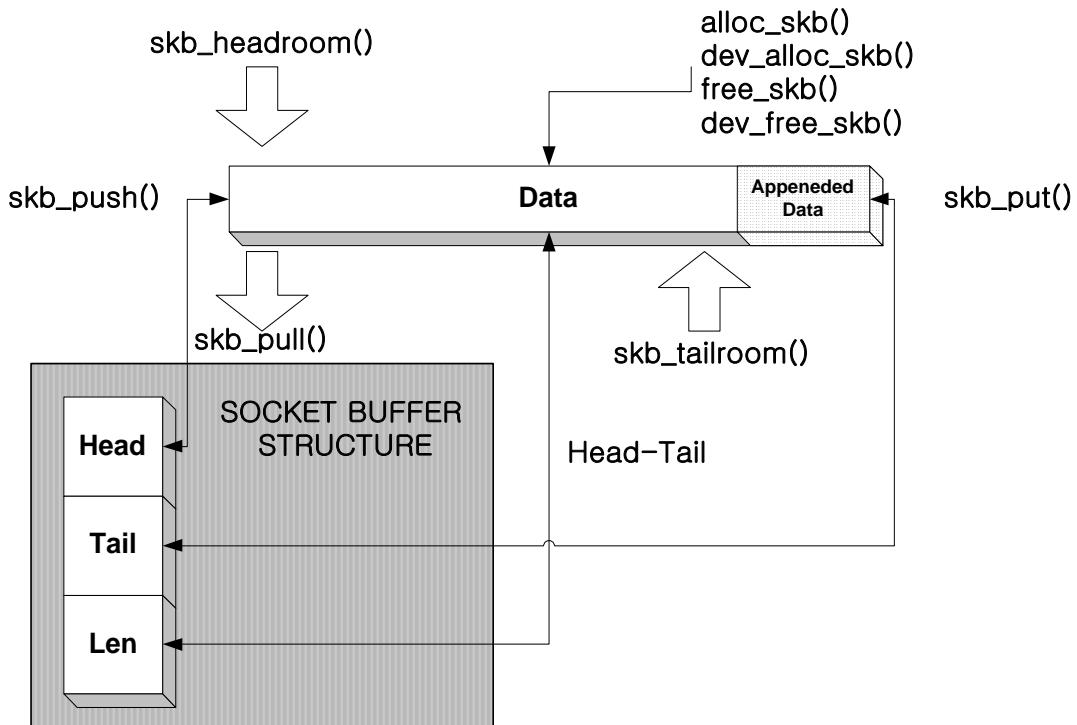


그림 64.Socket Buffer Structure

[그림57]에서 보듯이 socket buffer는 head와 tail에 data를 삽입할 수 있는데, packet의 header를 붙이거나 tail에 CRC(Cycle Redundancy Check)¹⁶⁴와 같은 checksum을 붙이게 될 때 사용하기 편리하다. 참고로 이것은 SCO UNIX의 경우 stream driver를 구현하는 것과 유사하다.

6.12.4. 전체적인 구조.

네트워크 디바이스 드라이버의 전체적인 프로그램 구조는 아래와 같다. 이 구조를 대부분이 사용하고 있으므로 이것을 위주로 해서 검증해 보면 될 것이다.

1. Module의 load – device driver를 module로 loading한다.
2. Device의 검출 – PCI device의 검출하며, network device driver를 kernel에 등록한다.
3. Device driver의 초기화 – DMA frame buffer의 초기화하고 multicasting address 저장 공간에 대해서 초기화를 한다.
4. Device의 open – Device를 사용하기 위해서 open을 한다. Device driver의 초기화 부분을 이곳에서 호출한다.
5. Device의 ioctl – IOCTL에 대한 것을 정해해서 둔다. (우리가 만들고자 하는 driver에서는 특별히 사용하고 있지 않지만 entry로서 가진다.)
6. Device의 close – Device의 사용을 끝내고, IRQ번호의 release와 DMA를 위해서 할당한 buffer의 free를 한다.
7. Data transmission(데이터의 전송) – DMA buffer상에 보내고자 하는 data를 옮겨놓고, DMA control을 setting한다.
8. Interrupt의 처리 – Transmission의 처리, receive의 처리를 담당하는 것으로 data의 전송이 끝이 났거나 혹은 새로운 데이터가 들어왔을 때에 발생하는 interrupt를 처리하는 것을 목적으로 한다.
9. Multicast address의 처리 – 이것은 우리가 만들고자 하는 device에 고유한 것으로 multicast

¹⁶⁴ CRC란 packet의 data가 정확한지를 점검하기 위한 목적으로 전달되며, 주로 polynomial로 나눈 나머지 값을 setting한다.

- address를 관리하는 register에 대한 처리를 담당한다.
10. Device의 software reset. – hardware적으로 이상이 있거나 혹은 device의 제어를 목적으로 software적으로 reset을 해준다. 이것은 특히 hardware가 돌이킬 수 없는 상황에 빠질 수도 있으므로 대체적으로 구현을 하는 것이 필요하다.
 11. Module의 unload – 사용이 끝난 module을 메모리에서 제거한다.

즉, 위에서 정의한 것들에 대해서 entry를 가지고 있어야 하며, 이러한 entry는 kernel에 의해서 사용하게 되므로 interface가 정의된 방법을 따라야 한다. 물론 위에서 말하는 것들에 대해서 전부가 필요한 것은 아니다. Network device driver에서는 다음과 같은 것을 정의 하도록 하고 있다.

6.12.5. Network의 Setting

드라이버를 설치한 후에는 Network 상황에 따라 적절한 setting이 필요한데, 여기에는 IP address의 할당 및 network mask, broadcast address를 주어야 하며, routing table에 등록 시켜야 한다. 이와 같은 일을 하기 위해서는 ifconfig과 route라는 명령어를 사용한다. 반드시 root권한을 가진 사용자만 해주어야 한다. 다음과 같은 방법으로 해보자.

```
(ex) ifconfig eth0 165.213.175.46
      ifconfig eth0 broadcast 165.213.175.63
      ifconfig eth0 netmask 255.255.255.192
      route add -net 165.213.175.0 netmask 255.255.255.192 dev eth0
      route add default gw 165.213.175.1
```

예제 3. Network의 setting

위의 예에서 eth0는 device에 대한 interface name이 되며, Ethernet card가 둘 이상이 존재할 경우에는 eth0, eth1, eth2, .. 순으로 운영체제가 Ethernet device에 이름을 할당한다. 따라서, 명령어를 실행하기에 앞서, 자신이 설치한 device의 이름을 확인하는 것이 필요하다. 이 경우, **ifconfig** 명령을 사용해서 현재 설치된 network device들의 이름을 확인할 수 있을 것이다.

위의 예제는 참고로서 사용하기 바라며, 자신의 network상황에 대해서 잘 이해한 후에 하기 바란다. Driver module을 제거하기 전에 반드시 다음과 같은 명령어로 더 이상 이 network interface를 사용하지 않는다는 것을 알려 주어야 한다. 주의하기 바란다.

ex)#ifconfig eth0 down

6.12.6. 네트워크 디바이스 구조체

네트워크 디바이스 드라이버를 작성하기 위해서는 먼저, 네트워크 디바이스만을 위한 net_device구조체에 대해서 이해해야 한다. ~/include/linux/netdevice.h를 보도록 하자.

Field	Description
char name[IFNAMSIZ]	네트워크 디바이스의 인터페이스 이름
unsigned long rmem_end	공유된 받기를 위한 메모리의 끝 주소
unsigned long rmem_start	공유된 받기를 위한 메모리의 시작 주소
unsigned long mem_end	공유된 메모리의 끝 주소
unsigned long mem_start	공유된 메모리의 시작 주소
unsigned long base_addr	디바이스의 I/O를 위한 주소
unsigned int irq	디바이스에 할당된 IRQ(인터럽트) 번호
unsigned char if_port	인터페이스를 위한 포트 타입(BNC, AUI, TP, etc.)
unsigned char dma	DMA 채널 번호(DMA master로 동작하는 PCI 카드는 해당 없음)
unsigned long state	디바이스의 현재 상태

struct net_device *next	네트워크 디바이스들의 연결 리스트
int (*init)(struct net_device *dev)	네트워크 디바이스의 초기화 함수에 대한 포인터(한번만 호출됨 - register_netdevice() 함수에서 호출함)
struct net_device *next_sched	네트워크 디바이스 구조체들의 다음 스케줄링 연결 포인터
int ifindex	인터페이스의 index
int iflink	인터페이스의 링크
struct net_device_stats *(*get_stats)(struct net_device *dev)	디바이스의 statistics(통계정보) 알려주는 함수에 대한 포인터.
struct iw_statistics *(*get_wireless_stats)(struct net_device *dev)	디바이스의 wireless statistics 알려주는 함수에 대한 포인터
unsigned long trans_start	마지막으로 Tx를 한 시점으로 jiffies값을 가진다.
unsigned long last_rx	마지막으로 Rx를 한 시점을 나타낸다.
unsigned short flags	현재 인터페이스의 flag이다.
unsigned short gflags	인터페이스의 flag이다. flags와는 사용이 다르다.
unsigned short mtu	최대 전송 단위를 byte 단위로 나타냄
unsigned short type	인터페이스 하드웨어 타입
unsigned short hard_header_len	하드웨어 헤더의 길이
void *priv	디바이스 드라이버의 private data에 대한 포인터(file object의 priv와 같은 역할을 할 수 있다.)
struct net_device *master	이 디바이스가 속해있는 그룹의 master 디바이스에 대한 포인터
unsigned char broadcast[MAX_ADDR_LEN]	하드웨어 broadcast 주소
unsigned char pad	dev_addr이 8bytes의 단위로 정렬되도록 만든다.
unsigned char dev_addr[MAX_ADDR_LEN]	하드웨어 주소
unsigned char addr_len	하드웨어 주소의 길이
struct dev_mc_list *mc_list	Multicast MAC 주소에 대한 포인터
int mc_count	현재 가지고 있는 multicast의 수를 유지한다.
int promiscuity	Promiscous mode를 표시(네트워크 상의 모든 패킷을 다 받는다.)
int allmulti	모든 multicast되는 패킷을 다 받는다.
int watchdog_timeo	Watch dog timer의 timeout 설정
struct timer_list	Watchdog_timer의 리스트 연결
void *atalk_ptr	Apple Talk link (이하의 4개 필드는 protocol에 의존한다.)
void *ip_ptr	IPv4에 특수한 데이터의 포인터
void *dn_ptr	DECnet에 특수한 데이터의 포인터
void *ipv6_ptr	IPv6에 특수한 데이터의 포인터
void *ec_ptr	Econet에 특수한 데이터의 포인터
struct Qdisc *qdisc	네트워크 디바이스 구조체와 관련된 Qdisc구조체 ¹⁶⁵ 의 포인터
struct Qdisc *qdisc_sleeping	현재 sleeping상태에 있는 Qdisc구조체
struct Qdisc *qdisc_list	Qdisk의 리스트에 대한 포인터
struct Qdisc *qdisc_ingress	Qdisk의 진입점에 대한 포인터
unsigned long tx_queue_len	Tx를 위한 패킷의 queue에 최대로 허용된 frame들의 수
spinlock_t xmit_lock	hard_start_xmit() 함수에서 사용할 lock
int xmit_lock_owner	hard_start_xmit() 함수를 호출하고 있는 CPU의 ID
spinlock_t queue_lock	디바이스의 queue lock

¹⁶⁵ Qdisc구조체는 패킷의 보내기와 받기를 위한 패킷의 큐를 말하는 것으로 해당 디바이스 드라이버와 연결되어 있다. 패킷을 보내거나 받기를 원하면 이곳에 패킷을 두고, 스캐줄링을 요청하면 해당 디바이스 드라이버나 상위의 프로토콜이 처리해 줄 것이다.

atomic_t refcnt	디바이스에 대한 reference count
int deadbeaf	디바이스가 현재 등록되지는 않았으나, 사용자가 있음을 말해주는 flag이다.
int features	네트워크 디바이스의 특성(feature)을 표시한다.
void (*uninit)(struct net_device *dev)	네트워크로 부터 detach된 후에 불려지는 함수로 init()함수와 반대되는 일을 한다. (unregister_netdevice()에서 호출함)
void (*destructor)(struct net_device *dev)	마지막 사용자의 reference가 사라지고난 후에 호출된다.
int (*open)(struct net_device *dev)	디바이스의 open()함수
int (*stop)(struct net_device *dev)	디바이스의 stop()함수
int (*hard_start_xmit)(struct sk_buff *skb, struct net_device *dev)	디바이스에 데이터의 전송을 위해서 커널이 호출하는 함수
int (*hard_header)(struct sk_buff *skb, struct net_device *dev, unsigned short type, void *daddr, void *saddr, unsigned len)	하드웨어 주소를 만들기 위해서 커널이 호출하는 함수(기본적으로 지원되는 함수이외에 하드웨어 헤더를 만들고자 할 때 사용됨)
int (*rebuild_header)(struct sk_buff *skb)	역시 하드웨어 헤더를 생성하는 함수로 사용됨
void (*set_multicast_list)(struct net_device *dev)	멀티 캐스팅이 지원되는 경우에 멀티 캐스트 리스트를 생성하기 위해서 호출됨
int (*set_mac_address)(struct net_device *dev, void *addr)	하드웨어의 MAC 주소를 변경하고자 할 때 호출됨. 참고로 Ethernet의 경우는 고정된 하드웨어 주소를 사용하는 경우가 많음
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd)	디바이스 드라이버에 대한 ioctl()메쏘드 함수
int (*set_config)(struct net_device *dev, struct ifmap *map)	하드웨어 설정을 바꾸기 위해서 호출됨
int (*hard_header_cache)(struct neighbour *neigh)	하드웨어에서 유지하는 이웃에 있는 하드웨어 주소를 관리하는 함수에 대한 포인터
void (*header_cache_update)(struct hh_cache *bh, struct net_device *dev, unsigned char *haddr)	하드웨어에서 유지하는 이웃에 있는 하드웨어 주소를 업데이트 해주는 함수에 대한 포인터
int (*change_mtu)(struct net_device *dev, int new_mtu)	경로(Path)에서 지원하는 MTU(Maximum Transfer Unit)를 변경하는 함수
void (*tx_timeout)(struct net_device *dev)	Tx timeout이 될 경우에 호출되는 함수에 대한 포인터
int (*hard_header_parse)(struct sk_buff *skb, unsigned char *haddr)	하드웨어 주소를 분석(parse)하기 위해서 사용되는 함수에 대한 포인터
int (*neigh_setup)(struct net_device *dev, struct neigh_parms *)	이웃(neighbour)에 대한 정보를 구성하기 위해서 호출되는 함수에 대한 포인터
int (*accept_fastpath)(struct net_device *, struct dst_entry *)	하드웨어에서 지원하는 목적지 엔트리(destination entry)를 찾기 위해서 호출되는 함수에 대한 포인터
struct module *owner	모듈의 사용자를 표시함
struct net_bridge_port *br_port	네트워크 bridge의 port를 나타냄
rwlock_t fastpath_lock	Fast path에서 사용되는 lock
struct dst_entry *fastpath[NETDEV_FASTROUTE_HMASK + 1]	Fast path에서 사용되는 목적지 엔트리의 배열에 대한 포인터
struct divert_blk *divert	Ethernet frame을 다른 곳으로 보내기 위해서 사용되는 divert_blk구조체에 대한 포인터로서 TCP및 UDP에 대한 목적지(destination)와 시작지(source)에 대한 port번호와 프로토콜 번호를 가지는 필드로 구성되어 있다.

표 47. net_device구조체의 정의

이곳에서 정의된 모든 필드들을 다 사용하는 것은 아니다. 디바이스가 지원하고자 하는 기능을 위해서 필요한 필드만을 설정해 주면 될 것이다.

`net_device` 구조체의 `feature` 필드를 차지하는 값은 아래와 같이 정의된다.

Feature	Value	Description
<code>NETIF_F_SG</code>	1	Scatter/gather I/O ¹⁶⁶ 를 지원하는 네트워크 디바이스이다.
<code>NETIF_F_IP_CSUM</code>	2	IPv4상에서 TCP/UPD만이 checksum이 가능하다.
<code>NETIF_F_NO_CSUM</code>	4	Checksum이 필요하지 않다.
<code>NETIF_F_HW_CSUM</code>	8	모든 패킷에 대해서 하드웨어적인 checksum이 가능하다.
<code>NETIF_F_DYNALLOC</code>	16	스스로가 할당과 해지가 가능한 디바이스이다.
<code>NETIF_F_HIGHDMA</code>	32	High memory에 대한 DMA가 가능하다.
<code>NETIF_F_FRAGLIST</code>	1	<code>NETIF_F_SG</code> 와 동일함

표 48. `net_device` 구조체의 `feature` 필드의 값

또한 통계정보를 가지는 `net_device_stats` 구조체는 아래와 같은 필드로 구성된다.

Field	Description
<code>unsigned long rx_packets</code>	받은 패킷의 수
<code>unsigned long tx_packets</code>	전달한 패킷의 수
<code>unsigned long rx_bytes</code>	받은 패킷의 byte수
<code>unsigned long tx_bytes</code>	전달한 패킷의 bytes수
<code>unsigned long rx_errors</code>	받은 패킷의 에러 수
<code>unsigned long tx_errors</code>	전달한 패킷의 에러 수
<code>unsigned long rx_dropped</code>	받은 패킷에서 drop된 수
<code>unsigned long tx_dropped</code>	전달한 패킷에서 drop된 수
<code>unsigned long multicast</code>	멀티 캐스트 된 패킷의 수
<code>unsigned long collisions</code>	패킷에서 충돌(collision)이 발생한 수
<code>unsigned long rx_length_errors</code>	받은 패킷의 에러중 길이에 에러가 있는 수
<code>unsigned long rx_over_errors</code>	받은 패킷이 receive buffer에서 overflow를 발생한 수
<code>unsigned long rx_crc_errors</code>	받은 패킷에서 CRC(Cyclic Redundancy Error)가 발생한 수
<code>unsigned long rx_frame_errors</code>	받은 frame이 align이 안된 에러가 발생한 수
<code>unsigned long rx_fifo_errors</code>	받은 패킷이 FIFO ¹⁶⁷ overrun을 일으킨 수
<code>unsigned long rx_missed_errors</code>	받지 못한 패킷의 수
<code>unsigned long tx_aborted_errors</code>	보내는 부분에서 abort된 패킷의 수
<code>unsigned long tx_carrier_errors</code>	보내는 부분에서 carrier ¹⁶⁸ 가 없어서 발생한 error의 수
<code>unsigned long tx_heartbeat_errors</code>	보내는 부분에서 생긴 heart beat ¹⁶⁹ 에러의 수
<code>unsigned long tx_window_errors</code>	보내는 곳에서 window상에서 에러가 발생한 수
<code>unsigned long rx_compressed</code>	받은 패킷이 압축된 수
<code>unsigned long tx_compressed</code>	보내는 패킷이 압축된 수

표 49. `net_device_stats` 구조체의 정의

¹⁶⁶ DMA와 같은 연산에서 하드웨어가 연속되지 않은 메모리 영역에 대해서도 DMA가 가능한 경우를 나타낸다.

¹⁶⁷ 디바이스 내의 FIFO(buffer)에서 에러를 일으킨 경우이다.

¹⁶⁸ 예를 들어서 네트워크 카드는 존재하지만, 네트워크 케이블이 빠진 경우가 있을 수 있다.

¹⁶⁹ Heart beat는 영어 단어상 심장 박동을 의미하며, 네트워크 카드가 제대로 주기적인 동작을 나타내고 있지 못하다는 것을 말해준다.

net_device_stats 구조체는 패킷을 받거나 보낼 때, 발생되는 에러 정보 및 통계정보를 가지는 구조체로 인터럽트의 발생시에 관련된 레지스터를 살펴서, 어떤 에러가 발생했는지를 검출해야지만 고쳐줄 수 있는 부분이다.

통계정보는 Error 혹은 디버깅(debugging) 정보를 상위의 application에 전달하는 것을 목적으로 사용된다. 이러한 정보는 디바이스 드라이버의 구조체¹⁷⁰에 들어가 있기에, 네트워크 디바이스의 함수 호출에서 파라미터로 넘겨지는 net_device 구조체를 이용해서 접근이 가능하다. 이러한 정보를 응용 프로그램에서 가지고 오는 방법으로는 ioctl() 메소드를 이용해서 디바이스 드라이버에 직접적으로 요청하게 되며, 디바이스 드라이버는 이 요구에 대해서 적절하게 응답하도록 프로그램하면 된다. 물론 내부적인 명령(command)에 대한 code를 미리 정의해 두어야 할 것이다.

만약 현재 네트워크 디바이스 드라이버를 디버깅한다고 가정한다면 이와 같은 정보를 이용해서 현재 network device driver가 제대로 작동하고 있는지를 가르쳐줄 목적으로도 사용할 수 있을 것이다. 이와 같은 정보는 ifconfig -a 명령어를 통해서 볼 수 있다¹⁷¹.

6.12.7. 간단한 Network Device Driver의 구현 예

이번 장에서 볼 것은 간단한 네트워크 디바이스 드라이버의 구현이다. 하드웨어적인 것은 전부다 생략하기로 하자. 이것은 실제로 사용하는 chip에 대해서 자세한 이해를 필요로 한다. 따라서, 우리가 선택할 수 있는 디바이스 드라이버 예는 ~/drivers/net/dummy.c와 ~/drivers/net/loop.c가 된다. Dummy.c는 dummy 네트워크 디바이스 드라이버 역할을 하는 드라이버이다. 즉, 아무런 동작도 하지 않는 단순히 네트워크 디바이스 드라이버로 존재만 하는 코드이다. Loop.c는 loopback 인터페이스를 구현하기 위해서 사용되는 네트워크 디바이스 드라이버이다. 이 디바이스 드라이버로 데이터를 보내면, 즉시 자신에게 데이터가 도착한 것처럼 보이게 될 것이다. 먼저 dummy.c부터 보기로 하자.

```
static struct net_device dev_dummy;
static int __init dummy_init_module(void)
{
    int err;

    dev_dummy.init = dummy_init; /* init()함수의 등록 */
    SET_MODULE_OWNER(&dev_dummy);

    /* Find a name for this unit */
    err=dev_alloc_name(&dev_dummy,"dummy%d");
    if(err<0)
        return err;
    if(register_netdev(&dev_dummy) != 0)
        return -EIO;
    return 0;
}

static void __exit dummy_cleanup_module(void)
{
    unregister_netdev(&dev_dummy);
    kfree(dev_dummy.priv);

    memset(&dev_dummy, 0, sizeof(dev_dummy));
    dev_dummy.init = dummy_init;
}
```

¹⁷⁰ 실제로 device driver는 program이므로 device driver에서 사용하는 정적인 변수에 대해서는 이곳에 선언하여 사용한다. 프로그램의 source에서 더 부연적으로 설명하도록 한다.

¹⁷¹ 참고로 SCO Unix용의 device driver를 작성하게 될 경우에는 IEEE 802.3 standard에서 제공하여야 할 필요 사항으로 Device driver kit document에 명시되어 있다. 따라서, 적어도 이 정도의 정보는 응용 프로그램에 알려주어야 할 것이다.

```
module_init(dummy_init_module);
module_exit(dummy_cleanup_module);
```

코드 464. 드라이버의 적재와 해제

모듈로 적재되는 것과 해제되는 부분이다. 이미 앞에서 보았을 것이다. `dummy_init_module()` 함수에서 `net_device`로 선언된 `dev_dummy.init`에 `dummy_init` 함수를 넣고, 모듈의 사용자가 누군지를 표시한다(`SET_MODULE_OWNER()`)¹⁷². 이젠 이 모듈에 대한 이름을 할당 받도록 한다(`dev_alloc_name()`). `dev_alloc_name()` 함수는 `~/net/core/dev.c`에 정의된 함수로 네트워크 디바이스의 연결 리스트를 검사해서 같은 이름이 있는지를 확인하는 함수이다. 없다면, 넘겨받은 `net_device` 구조체에 이름을 넣는다. 이것을 마치고 나면, 이젠 네트워크 디바이스로 등록하는 `register_netdev()` 함수를 호출한다. 모듈의 해제에서는 네트워크 디바이스로 등록된 것을 해제하는 역할을 하는 `unregister_netdev()` 함수를 호출한다. 그리고, `dev_dummy.priv`에 `dummy` 네트워크 디바이스로 할당된 메모리가 있다면 이것을 해제한다(`kfree()`). 마지막으로 `dummy` 네트워크 디바이스 구조체를 0으로 설정하고(`memset()`), 다시 초기화 함수를 `dummy_init()`로 설정한 다음 복귀한다.

네트워크 디바이스 드라이버의 등록과 해제는 `register_netdev()`와 `unregister_netdev()` 함수가 맡고 있다. 이 함수들의 정의는 `~/drivers/net/net_init.c`에 나와있으며 아래와 같다.

```
static struct net_device dev_dummy;
int register_netdev(struct net_device *dev)
{
    int err;

    rtnl_lock();
    if (strchr(dev->name, '%'))
    {
        err = -EBUSY;
        if(dev_alloc_name(dev, dev->name)<0)
            goto out;
    }
    if (dev->name[0]==0 || dev->name[0]==' ')
    {
        err = -EBUSY;
        if(dev_alloc_name(dev, "eth%d")<0)
            goto out;
    }
    err = -EIO;
    if (register_netdevice(dev))
        goto out;
    err = 0;
out:
    rtnl_unlock();
    return err;
}
void unregister_netdev(struct net_device *dev)
{
    rtnl_lock();
    unregister_netdevice(dev);
```

¹⁷² 아래와 같이 정의되어 있다.

```
#ifdef CONFIG_MODULES
#define SET_MODULE_OWNER(some_struct) do { (some_struct)->owner = THIS_MODULE; } while (0)
#else
#define SET_MODULE_OWNER(some_struct) do { } while (0)
#endif
```

```
rtnl_unlock();
}
```

코드 465. 네트워크 디바이스 드라이버의 등록과 해제

`rtnl_lock()`은 routing table과 netlink에서 사용되는 세마포어를 획득하는 매크로이다. `strchr()`함수는 넘겨받은 두 파라미터에서 두번째 파라미터로 시작하는 첫번째 파라미터의 문자부분을 찾는다. 만약 있다면, `dev_alloc_name()`을 호출해서 새로운 네트워크 디바이스의 이름을 할당받는다. 만약 네트워크 디바이스의 이름이 0이나 공백(space)로 시작한다면 `eth%d`로 시작하는 이름을 할당하려고 할 것이다. %d에는 번호가 들어갈 것이다. 마지막으로 `register_netdevice()`함수를 호출해서 네트워크 디바이스 이름을 등록할 것이다. 해제하는 것은 `unregister_netdev()`함수이며, 단순히 등록을 해제하는 `unregister_netdevice()` 함수를 호출하는 일을 한다.

`register_netdevice()`함수는 네트워크 디바이스 구조체의 queue 및 xmit에 대한 lock을 초기화하고, 네트워크 디바이스 드라이버의 init()함수를 호출해주며, `rebuild_header()`함수가 NULL인 경우에는 이를 `default_rebuild_header()`함수로 초기화 시켜주는 등의 일을 한다. `unregister_netdevice()`함수는 네트워크 디바이스의 close()함수를 호출해주며, 네트워크 디바이스들의 연결 리스트에서 디바이스를 나타내는 구조체를 제거하고, 디바이스 드라이버가 uninit()함수를 가진 경우에는 이 함수도 호출해 준다. 마지막으로 해당 네트워크 디바이스 구조체를 버리는 `dev_put()`함수를 호출한다¹⁷³.

dummy 네트워크 디바이스 드라이버의 등록시에 init()함수로 호출되는 것은, 모듈의 적재시에 등록된 `dummy_init()` 함수이며, 아래와 같다.

```
static int __init dummy_init(struct net_device *dev)
{
    /* Initialize the device structure. */
    dev->hard_start_xmit      = dummy_xmit;
    dev->priv = kmalloc(sizeof(struct net_device_stats), GFP_KERNEL);
    if (dev->priv == NULL)
        return -ENOMEM;
    memset(dev->priv, 0, sizeof(struct net_device_stats));
    dev->get_stats      = dummy_get_stats;
    dev->set_multicast_list = set_multicast_list;
    /* Fill in the fields of the device structure with ethernet-generic values. */
    ether_setup(dev);
    dev->tx_queue_len = 0;
    dev->flags |= IFF_NOARP;
    dev->flags &= ~IFF_MULTICAST;
#ifndef CONFIG_NET_FASTROUTE
    dev->accept_fastpath = dummy_accept_fastpath;
#endif
    return 0;
}
```

코드 466. `dummy_init()` 함수

이 함수가 해주는 일은 `net_device` 구조체를 초기화 시켜주는 일을 한다. 먼저 `hard_start_xmit()`을 `dummy_xmit()`을 가르키도록 하고, `priv` 필드에는 네트워크 디바이스의 통계정보를 가질 공간을 할당한다. 만약 할당받은 메모리가 NULL이라면 `-ENOMEM`을 에러값으로 돌려준다. 할당받은 메모리 공간은 0으로 초기화하고(`memset()`), 통계정보를 구하는 `get_stats`에는 `dummy_get_stats()`함수를, multicast 주소를 설정하는 `set_multicast_list`에는 `set_multicast_list()`함수를 넣어준다.

이외에 기본적인 `net_device` 구조체의 초기화를 위해서 `ether_setup()`을 호출해주게 되는데, 이곳은 조금 후에 다시 보도록 하자. Tx queue의 길이(`tx_queue_len`)은 0을 주고, 디바이스는 ARP를 사용하지 않는다는 것을 나타내기 위해서 `IFF_NOARP`를 flag에 설정한다. 또한 디바이스가 multicasting도 지원하지

¹⁷³ [~/net/core/dev.c](#)를 참조하기 바란다.

않으므로 IFF_MULTICAST를 flag에서 지운다. 만약 CONFIG_NET_FASTROUTE가 커널 컴파일에서 설정되었다면, accept_fastpath에는 dummy_accept_fastpath()를 넣는다.

net_device구조체의 flags필드에 들어갈 수 있는 값으로는 아래와 같은 것들이 있으며, 정의는 ~/include/linux/if.h에 나와 있다.

Flag	Value	Description
IFF_UP	0x0001	현재 네트워크 인터페이스가 UP상태이다(동작중).
IFF_BROADCAST	0x0002	Broadcasting을 지원한다.
IFF_DEBUG	0x0004	Debugging을 On시킨다.
IFF_LOOPBACK	0x0008	Loopback 인터페이스 이다.
IFF_POINTTOPOINT	0x0010	Point to point연결을 지원한다.
IFF_NOTAILERS	0x0020	Trailer의 사용을 금한다.
IFF_RUNNING	0x0040	현재 네트워크 디바이스에 자원이 할당되어 있다.
IFF_NOARP	0x0080	ARP protocol을 사용하지 않는다.
IFF_PROMISC	0x0100	모든 네트워크 상의 패킷을 받겠다.
IFF_ALLMULTI	0x0200	모든 multicasting되는 패킷을 받겠다.
IFF_MASTER	0x0400	부하 분산자(load balancer)의 master로 동작한다.
IFF_SLAVE	0x0800	부하 분산자의 slave로 동작한다.
IFF_MULTICAST	0x1000	Multicasting기능을 지원한다.
IFF_VOLATILE	->	IFF_LOOPBACK IFF_POINTPOINT IFF_BROADCAST IFF_MASTER IFF_SLAVE IFF_RUNNING
IFF_PORTSEL	0x2000	전송 media를 선택할 수 있다.
IFF_AUTOMEDIA	0x4000	전송 media를 자동으로 선택할 수 있는 기능이 있다.
IFF_DYNAMIC	0x8000	주소를 바꾸는 기능을 가진 dial-up 장치이다.

표 50. net_device구조체의 flag값

이러한 값들은 네트워크 디바이스의 현재 동작을 조절하는 목적으로도 사용되기 때문에 이 값들에 맞춰서 네트워크 디바이스 드라이버는 적절한 일을 해주어야 할 것이다. 만약 상위에서 IFF_PROMISC를 설정을 요청했다면, 디바이스 드라이버는 디바이스가 promiscuous모드를 지원하는지를 확인해서, 지원된다면 이러한 기능을 켜주어야(On) 할 것이다.

```
/* fake multicast ability */
static void set_multicast_list(struct net_device *dev)
{
}

#ifndef CONFIG_NET_FASTROUTE
static int dummy_accept_fastpath(struct net_device *dev, struct dst_entry *dst)
{
    return -1;
}
#endif
...
static int dummy_xmit(struct sk_buff *skb, struct net_device *dev)
{
    struct net_device_stats *stats;

    stats = (struct net_device_stats *)dev->priv;
    stats->tx_packets++;
    stats->tx_bytes+=skb->len;

    dev_kfree_skb(skb);
}
```

```

    return 0;
}

static struct net_device_stats *dummy_get_stats(struct net_device *dev)
{
    return dev->priv;
}

```

코드 467. dummy 네트워크 디바이스의 함수들

여기서 보이는 함수들은 전부 dummy 네트워크 디바이스 드라이버가 커널에 export하는 함수들이다. `set_multicast_list()`와 `dummy_accept_fastpath()`는 아무런 동작도 일으키지 않으며, `dummy_get_stats()`함수는 단순히 할당받았던 `net_device`구조체의 `priv`필드만을 돌려준다.

`dummy_xmit()`함수는 패킷의 전송을 위해서 호출되는 함수로 넘겨받는 것은 `sk_buff`와 해당 `net_device`구조체이다. 먼저 `net_device`구조체의 `priv`필드에서 통계정보에 대한 구조체를 가져오고, 전달한 패킷의 수를 나타내는 `tx_packets`을 증가 시켜주며, 전달한 패킷의 byte수를 나타내는 `tx_bytes`에 `sk_buff`의 길이 만큼을 더해준다. 패킷은 실제적인 전송이 일어나지는 않으므로 이곳에서 패킷을 버리는 `dev_kfree_skb()`함수를 호출한다.

이것으로 하나의 간단한 네트워크 디바이스 드라이버의 구현이 끝난다. 하지만, 실제적인 네트워크 디바이스 드라이버의 구현은 더 복잡하며, 인터럽트의 처리와 DMA의 구현등이 더 추가될 것이다.

이전 앞에서 잠시 논의하다가 만 `ether_setup()`함수를 보기로 하자. 이 함수는 기본적인 `net_device`구조체의 필드들에 대한 초기화를 수행하는 함수이다. 정의는 `~/drivers/net/net_init.c`에서 찾을 수 있다.

```

void ether_setup(struct net_device *dev)
{
    dev->change_mtu = eth_change_mtu;
    dev->hard_header = eth_header;
    dev->rebuild_header = eth_rebuild_header;
    dev->set_mac_address = eth_mac_addr;
    dev->hard_header_cache = eth_header_cache;
    dev->header_cache_update= eth_header_cache_update;
    dev->hard_header_parse = eth_header_parse;
    dev->type = ARPHRD_ETHER;
    dev->hard_header_len = ETH_HLEN;
    dev->mtu = 1500; /* eth_mtu */
    dev->addr_len = ETH_ALEN;
    dev->tx_queue_len = 100; /* Ethernet wants good queues */
    memset(dev->broadcast,0xFF,ETH_ALEN);
    /* New-style flags. */
    dev->flags = IFF_BROADCAST|IFF_MULTICAST;
    dev_init_buffers(dev);
}

```

코드 468. ether_setup()함수

`ether_setup()`함수는 Ethernet을 위한 `net_device`구조체를 기본적인 함수들과 상수값으로 채우는 일을 한다. 이곳에서 결정되는 필드들은 `change_mtu`, `hard_header`, `rebuild_header`, `set_mac_address`, `hard_header_cache`, `heard_cache_update`, `hard_header_parse`, `type`, `hard_header_len`(=ETH_HLEN¹⁷⁴:14bytes), `mtu`(=1500bytes:ethernet 패킷의 최대 길이), `addr_len`(=ETH_ALEN:6bytes), `tx_queue_len`(=100개), `flags`(=IFF_BROADCAST | IFF_MULTICAST)가 있다.

이것을 다 마쳤다면, 이제 `dev_init_buffers()`함수를 호출해서 디바이스에 관련된 buffer를 초기화 한다. 현재는 아직 `dev_init_buffers()`함수에서 아무런 동작도 해주지 않는다.

¹⁷⁴ `~/include/linux/if_ether.h`에 정의되어 있다.

지금까지 우리가 본것은 아무런 역할도 해주지 않는 가상의(dummy) 네트워크 디바이스 드라이버를 보았다. 이제부터 볼 것은 *ifconfig -a* 명령에서 *lo*라고 나오는 loopback 네트워크 인터페이스를 만들기 위한 디바이스 드라이버를 보도록 하자. Loopback 네트워크 인터페이스는 IP주소로 127.0.0.1이라는 특수한 주소를 할당받아서 사용하고 있다. 코드는 *~/drivers/net/loopback.c*에 해당한다. 단순히 디바이스 드라이버의 상위 layer에서 받은 패킷을, 다시 상위의 프로토콜 layer로 돌려주는 역할을 한다.

```
#define LOOPBACK_OVERHEAD (128 + MAX_HEADER + 16 + 16)
...
/* Initialize the rest of the LOOPBACK device. */
int __init loopback_init(struct net_device *dev)
{
    dev->mtu      = PAGE_SIZE - LOOPBACK_OVERHEAD;
    dev->hard_start_xmit = loopback_xmit;
    dev->hard_header = eth_header;
    dev->hard_header_cache = eth_header_cache;
    dev->header_cache_update= eth_header_cache_update;
    dev->hard_header_len   = ETH_HLEN;           /* 14 */
    dev->addr_len        = ETH_ALEN;          /* 6 */
    dev->tx_queue_len = 0;
    dev->type            = ARPHRD_LOOPBACK; /* 0x0001 */
    dev->rebuild_header = eth_rebuild_header;
    dev->flags           = IFF_LOOPBACK;
    dev->priv = kmalloc(sizeof(struct net_device_stats), GFP_KERNEL);
    if (dev->priv == NULL)
        return -ENOMEM;
    memset(dev->priv, 0, sizeof(struct net_device_stats));
    dev->get_stats = get_stats;
    if (num_physpages >= ((128*1024*1024)>>PAGE_SHIFT))
        dev->mtu = 4096*4 - LOOPBACK_OVERHEAD;
    dev_init_buffers(dev);           /* ~/include/linux/netdevice.h에 정의 됨, 아무일도 안함 */
    return(0);
};
```

코드 469. *loopback_init()* 함수

모든 네트워크 디바이스의 구조체(net_device구조체)는 *dev_base*의 연결된다. 이 변수는 loopback 디바이스를 나타내는 네트워크 디바이스 구조체로 초기화가 되며, 아래와 같이 *~/net/Space.c*에 정의되어 있다.

```
struct net_device loopback_dev =
    {"lo", 0, 0, 0, 0, 0, 0, 0, 0, 0, NEXT_DEV, loopback_init};
struct net_device *dev_base = &loopback_dev;
```

코드 470. *dev_base*의 정의

따라서, loopback디바이스는 *lo*를 인터페이스의 이름으로 가지며, *loopback_init()*가 초기화 함수로 디바이스의 등록에서 호출된다. *NEXT_DEV*는 *Space.c*에서 정의된 디바이스를 연결하기 위한 매크로이다. 이렇게 등록된 디바이스는 *~/drivers/block/genhd.c*에서 *device_init()*함수에서 *CONFIG_NET*으로 설정된 경우 *net_dev_init()*함수를 호출해서 초기화된다. *device_init()*함수를 불러주는 것은 *~/fs/partitions/check.c*에 정의된 *partition_setup()*함수이며, *__initcall()* 매크로 선언에 의해서, *~/init/main.c*에 정의된 *do_initcalls()*함수에서 *__initcall_start*와 *__initcall_end*사이에 있는 함수들을 호출하는 부분에서 실행 기회를 얻게 된다.

*loopback_init()*함수에서는 *net_device*구조체의 필드들을 초기화하는 역할을 하고 있다. 이곳에서 초기화되는 정보로는 *mtu*(=PAGE_SIZE - LOOPBACK_OVERHEAD), *hard_start_xmit*(*loopback_xmit*), *hard_header*(=eth_header), *hard_header_cache*(=eth_header_cache), *header_cache_update*(=eth_header_cache_update),

hard_header_len(=ETH_HLEN), addr_len(=ETH_ALEN), tx_queue_len(=0), type(=ARPHRD_LOOPBACK:0x0001), rebuild_header(=eth_rebuild_header), flags(=IFF_LOOPBACK), priv(=net_device_stats)가 있으며, get_stats는 get_stat()함수로 초기화 된다. 할당받은 priv필드의 메모리에 대해서는 0으로 초기화 되며, 현재 메모리의 물리적인 페이지의 수가 128 x 1024 x 1024(=128Mbytes)를 페이지 크기로 나눌 수 보다 크다면, mtu를 4096 x 4에서 LOOPBACK_OVERHEAD를 뺀 수로 초기화 한다. 이는 적어도 4페이지를 동시에 보내도록 하는 부분이다. 역시 dev_init_buffers()는 아무런 역할도 하지 않는 함수이다.

```
static int loopback_xmit(struct sk_buff *skb, struct net_device *dev)
{
    struct net_device_stats *stats = (struct net_device_stats *)dev->priv;

    /* 패킷을 보내는 부분이다.*/
    if(atomic_read(&skb->users) != 1)
    {
        struct sk_buff *skb2=skb;
        skb=skb_clone(skb, GFP_ATOMIC);           /* Clone the buffer */
        if(skb==NULL) {
            kfree_skb(skb2);
            return 0;
        }
        kfree_skb(skb2);
    }
    else
        skb_orphan(skb);

    /* 이하의 부분은 패킷을 받는 것을 처리한다.*/
    skb->protocol=eth_type_trans(skb,dev);
    skb->dev=dev;
#ifndef LOOPBACK_MUST_CHECKSUM
    skb->ip_summed = CHECKSUM_UNNECESSARY;
#endif
    netif_rx(skb);
    stats->rx_bytes+=skb->len;
    stats->tx_bytes+=skb->len;
    stats->rx_packets++;
    stats->tx_packets++;
    return(0);
}

static struct net_device_stats *get_stats(struct net_device *dev)
{
    return (struct net_device_stats *)dev->priv;
}
```

코드 471. loopback_xmit()와 get_stats()함수

loopback_xmit()함수는 초기화에서 hard_start_xmit필드로 정의되었기에 커널에서 패킷을 전달하기 위해서 호출될 것이다. 하는 일은 단순히 상위에서 받은 패킷을 다시 상위로 올려주는 역할을 한다. 마치 다른 곳에서 패킷을 받은 것과 같은 일을 해 줄 것이다.

먼저 네트워크 디바이스 드라이버 구조체에서 통계정보를 가지는 곳에 대한 주소를 얻는다(dev->priv). 소켓버퍼(sk_buff) 사용자 카운터를 읽어서(atomic_read()) 1이 아니라면, 새로운 소켓 버퍼를 하나 할당 받고, 동일한 소켓 버퍼를 하나 만든다. 이전에 있던 소켓 버퍼는 kfree_skb()함수를 호출해서 해제한다. skb_orphan()함수는 소켓 버퍼의 owner가 가진 destructor함수가 존재할 경우 이를 호출해주며, 소켓 버퍼의 owner를 없애주는 역할을 하는 함수이다.

소켓 버퍼의 protocol은 eth_type_trans()함수를 호출해서 초기화 한다. eth_type_trans()함수는 ~/net/ethernet/eth.c에 정의되어 있으며, 패킷의 타입을 결정해 주며, 프로토콜의 ID를 넘겨준다. 따라서,

이 함수의 복귀값으로 소켓 버퍼의 protocol필드를 채우고, dev 필드에는 현재의 네트워크 디바이스 구조체를 표시한다. LOOPBACK_MUST_CHECKSUM이 설정된 경우에는 소켓 버퍼의 ip_summed에 CHECKSUM_UNNECESSARY로 주어서 IP layer에서 checksum을 하지 않도록 만든다.

netif_rx()¹⁷⁵ 함수는 상위의 laery에 패킷이 도착했음을 알리는 함수로서, 상위에 있는 layer가 패킷을 처리할 수 있도록 받은 패킷을 큐잉(queueing)한다. 나머지는 통계정보를 업데이트 하는 부분이다. rx_bytes와 tx_bytes, rx_packets와 tx_packets등의 정보가 갱신된다.

get_stats()함수는 디바이스 드라이버가 관리하는 통계정보를 알려주는 역할을 한다. 단순히 net_device구조체의 priv에 할당한 net_device_stats구조체를 돌려주기만 하면 된다.

물론 이곳에서 보여주는 것은 하드웨어에 독립적인 부분들에 대한 코드만을 간단히 다루었을 뿐이다. 더 자세한 구현을 보기를 원하는 사람들을 위해서 Appendix에 커널 버전 2.0에서 작가가 구현한 디바이스 드라이버의 코드와 설명을 보기 바란다.

6.12.8. 특정 운영 체제에서의 구현

대부분의 운영 체제는 드라이버와의 인터페이스를 정의하고 있으며, 네트워크 디바이스 역시 예외가 아니다. 즉, 중요한 핵심부분에 대한 이해만 있으면, DDK(device driver development kit) 툴(tool)을 이용해서 인터페이스에 맞게 구현하면 된다는 말이다. 이것은 하나의 드라이버를 개발한 상태에서 새로운 운영 체제로 포팅(porting)하게 되는 경우에 해당한다.

가령 예를 들어, Windows NT의 경우에는 NDIS(Network Driver Interface Specification)에 따라 작성되며, 하드웨어에 의존적인 부분만을 새로이 코딩(coding)해 주면 된다. 또한 SCO UNIX의 경우에도 마찬가지로 드라이버 문서들에서 명시된 방법들과 드라이버와 커널 인터페이스만을 만족하도록 해주면 포팅은 쉬운 과정이 된다. 물론 그에 따라, 익히고 보아야 할 문서들의 분량은 많다. 이것은 단순한 시간의 문제가 될 뿐이다. 참고로 SCO Unix에서는 네트워크 디바이스 드라이버에 대해서 Stream Driver라는 말을 사용하고 있으며, www.sco.com에서 디바이스 드라이버에 대한 개발 툴과 예제 드라이버를 찾을 수 있을 것이다.

¹⁷⁵ ~/net/core/dev.c에 정의되어 있다.

7. Real-Time Clock(RTC)의 분석

Real-Time Clock(RTC)는 Linux에서 기본적으로 제공하는 system clock¹⁷⁶ 보다 더 정밀한(hight resolution) 시간(clock) tick을 얻기 위해서 사용한다. 즉, 일반적인 process가 이용할 수 있는 범위의 clock은 Linux에서는 10ms 이상이라는 점을 간과해서는 안된다. 예를 들어, 100 usec 단위로 동작하는 process를 만든다는 것은 Linux에서 제공하는 timer 메커니즘으로는 불가능하다는 것이다. 만약 system clock의 속도를 더 높여주면 좋을 것이라고 생각한다면, 이는 전체적으로 시스템의 성능을 저해할 가능성을 가지고 있기에 좋은 방법이 아니다. 즉, 더 좋은 정밀도(resolution)을 가지는 system clock을 사용했을 때 timer interrupt를 처리하는 overhead도 상대적으로 증가하기에 오히려 전반적인 시스템의 성능에 악영향을 미친다. 이럴 때 사용할 수 있는 것이 바로 RTC이다.

이는 시스템의 clock tick interrupt를 사용하지 않으며, RTC 자신이 가지고 있는 interrupt를 사용해서 작업을 진행할 수 있기에 시스템의 전체적인 performance에 주는 영향이 적다. 물론 이것을 사용한다고 해서 일반적인 Linux 커널에서는 얻을 수 있는 clock tick의 정밀도(resolution)에는 한계가 있다.¹⁷⁷ 현재로서는 대략 이렇게 얻을 수 있는 한계가 1.2 milliseconds 정도라고 한다.¹⁷⁸ 즉, 이 이하의 단위에서 외부의 event에 반응하고자 하는 process는 현재의 커널에서는 불가능하다는 것이다. 따라서, 우리가 초점을 맞출 수 있는 것은 RTC를 이용하더라도 1.2 milliseconds 이상의 latency를 최소한 가지는 프로세스여야 할 것이다.

7.1. RTC 하드웨어의 분석

PC에서 사용하는 RTC로는 대표적인 것으로 Motorola에서 나온 MC1468180이라는 Real-time clock과 CMOS RAM을 결합한 것이 있다. 즉, 시간을 저장하기 위한 CMOS RAM과 RTC가 결합된 형태를 가진다. 또한 여기에 추가적으로 시스템이 power off가 된 상황에서도 CMOS에서 저장된 시간정보를 기억하기 위해서 battery가 필요하다. 아래의 [그림 65]은 RTC의 대략적인 block diagram을 보여준다.

¹⁷⁶ 일반적으로 사용되는 clock tick은 jiffies값을 증가시켜주는 10ms(milliseconds) clock이다. 이것을 사용한다면, 더 좋은 정밀도(resolution)의 시간에 반응하는 process를 만들기는 어렵다.

¹⁷⁷ 즉, Low Latency Kernel Patch나 Preemptive Kernel Patch를 사용하지 않았을 경우이다.

¹⁷⁸ 이것은 Low Latency Kernel Patch와 Preemptive Kernel Patch를 적용했을 경우에, interrupt가 발생해서, 관련된 process가 처리를 시작하는 시점까지 걸리는 시간을 의미한다. Low Latency Kernel Patch만을 적용했을 경우에는 최대 interrupt latency가 1.3 milliseconds라고 하며, Preemptive Kernel Patch만을 적용했을 경우에는 최대 45.2 milliseconds가 걸렸다고 한다.

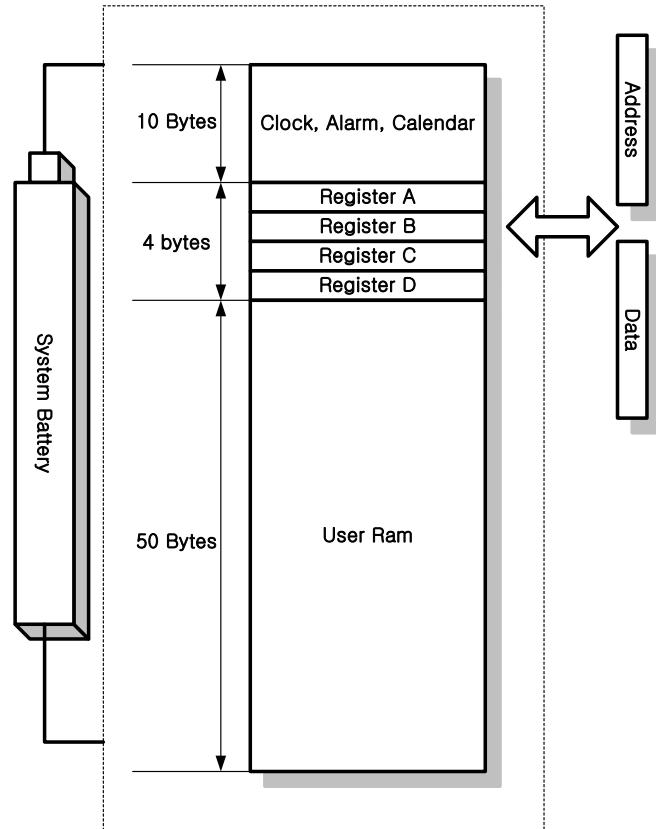


그림 65. RTC Hardware Module의 구성

그림과 같이 clock, alarm, calendar를 나타내기 위해서 10 bytes를 사용하고 RTC를 제어(control)하기 위해서 4개의 1 byte짜리 레지스터 A, B, C, D를 가지고 있다. 또한 사용자 정의 영역으로 사용할 수 있는 50 bytes를 가진다. 이들 값들에 대해서는 address와 data 레지스터를 이용해서 소프트웨어적으로 접근할 수 있다.

최상위에 있는 clock, alarm, calendar부분의 10 bytes는 BCD(Binary Coded Decimal)로 되어 있거나, 혹은 binary값으로 저장되는 부분이다. 각 byte 영역이 하는 역할과 A,B,C,D register의 주소는 다음의 그림과 같이 보여진다.

Seconds	00
Seconds Alarm	01
Minutes	02
Minutes Alarm	03
Hours	04
Hours Alarm	05
Day of Week	06
Date of Month	07
Month	08
Year	09
Register A	0A
Register B	0B
Register C	0C
Register D	0D

그림 66. RTC의 Address Map

각 10 bytes 영역에 저장되는 값은 수치가되는 값이 가질 수 있는 값의 한계 이내에 들어온다. 예를 들어, seconds라고하면, 0에서 59까지의 값을 가질 수 있을 것이며, 달(month)에서 특정한 날(date)을 가르키기 위해서는 1에서 31까지가 될 수 있을 것이다. Binary code로 나타내면, 31은 0x1F라는 값을 가지지만, BCD로 표시하면 “31”이라는 값을 4 bit단위로 가진다.

나머지 50 bytes의 RAM 공간은 MC146818에서 특별한 목적을 가지고 사용되는 공간이 아니다. 따라서, 이 부분은 사용자 프로그램(혹은 Kernel)에서 사용할 수 있는 공간이다. 시스템이 전원이 차단된 상황에서는 이 영역은 battery에 의해서 backup이 유지되므로, 사용자가 데이터를 기억해 놓을 수 있는 장치로도 사용가능한 부분이다. 나중에 MC146818의 기능이 확장될 경우에, 이 영역을 사용할 가능성도 있다.

RTC가 사용하는 interrupt는 어떻게 발생하는지 보도록 하자. 인터럽트의 source가 될 수 있는 것은 크게 3가지고 있다. 첫번째로 interrupt의 원인이 될 수 있는 것은 alarm이다. Alarm은 초당 1번에서부터 하루에 한번까지 가능하다. 두번째로는 주기적인(periodic) interrupt로서 0.5초에 한번에서부터 30.517microseconds에 한번까지 가능하다. 마지막으로 interrupt의 source가 될 수 있는 것은 update-ended interrupt라는 RTC의 시간값을 update하는 것이 끝났다는 것을 알려주는 interrupt이다. 따라서, 이것은 update-cycle과 동일한 값으로 발생될 것이다. 이러한 interrupt의 source가 될 수 있는 것은 register B라고 표시된 제어 register를 설정하는 것으로 가능하다.

그럼 이제는 RTC 제어에 중요한 역할을 수행하는 Register A, B, C, D에 대해서 알아보기로 하자. 각각의 레지스터의 각 bit이 어떤 역할을 하는지 이해해야만, 나중에 디바이스 드라이버를 분석하는데 어려움이 없을 것이다.

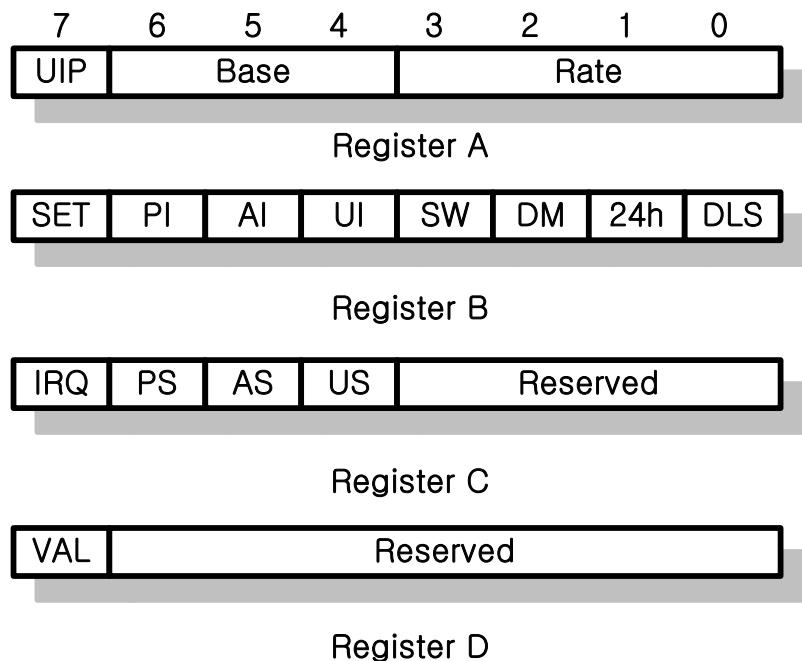


그림 67. RTC의 register A, B, C, D의 포맷(format)

각각의 레지스터의 field가 하는 역할은 아래와 같은 표로 정리해 볼 수 있다. 여기서, 레지스터 A의 경우에는 base와 rate를 나타내는 3 bit, 4bit의 필드를 하나로 묶어서 보였다는 점을 명심하기 바란다. 각각의 bit를 설정함에 따라서, 해당하는 base 값과 rate값은 조정될 것이다.

Register	Field	Description
Register A	Rate	15가지의 값을 표현하며, 주기적인 인터럽트나 clock의 SQW(Square Wave)의 형을 결정하는데 사용된다. ([표 52]을 참조)

	Base	RTC의 input으로 들어오는 crystal oscillator의 신호를 나누어(divide)주는 divisor를 제어한다. 이것으로 4.194304 MHz, 1.048576 MHz, 32.768 KHz의 신호를 설정할 수 있다. 000인 경우에는 첫번째것을, 001인 경우에는 두번째 것을 010인 경우에는 세번째 것을 RTC의 input신호로 사용한다. 110이나 111 및 기타 다른 값은 사용되지 않거나 test용으로 사용된다.
Register B	DLS	Day Light Saving Enable bit로 4월의 마지막 일요일 이후에 오전 1시 59분 59초 이후, 오전 3시로 설정되도록 만들어주기 위한 것이다.
	24/12	24시간 형태 혹은 12시간 형태로 시간을 표시하는 것을 제어한다.
	DM	BCD혹은 Binary모드로 시간과 calendar 데이터를 유지하는지를 결정한다.
	SW	Register A에서 설정한 square wave signal을 정해준 주기에 따라 발생시키도록 만들어준다.
	UI	Update-Ended Interrupt를 발생시키도록 한다.
	AI	Alarm Interrupt를 발생시키도록 한다.
	PI	주기적인(Periodic) Interrupt를 발생시키도록 한다.
	SET	시간이나 calendar의 update가 일어나도록(SET=0) 한다.
Register C	US	Update-Ended Interrupt flag으로 매번의 update후에 설정된다.
	AS	Alarm Interrupt Flag으로 현재 시간이 alarm시간과 일치할 경우에 설정된다.
	PS	Periodic Interrupt Flag으로 주기적인 인터럽트가 발생했을 경우에 설정된다.
	IRQ	Interrupt Request Flag으로 Periodic Interrupt나 Alarm Interrupt, Update-Ended Interrupt가 발생했을 경우에 설정된다.
Register D	VAL	Valid RAM and Time(VRT) bit으로 RAM의 내용에 대한 상태(condition)을 알려준다. RTC의 power-sense pin이 Low상태라면 0을 가진다. 이것은 RAM과 시간값이 유효하다는 것을 알리기 위한 목적으로 프로그램에 의해서 사용된다.

표 51. RTC 레지스터 A, B, C, D의 필드 정의

여기서 한가지 짚고 넘어가야 할 것이 있다. 즉, Register A의 Rate 필드에 있는 bit의 값에 따라서, 발생하는 주기적인 interrupt및 square wave의 주파수(frequency)이다. 아래의 표를 보도록 하자.

Register A				4.194304 or 1.048576 MHz		32.768 KHz	
RS3	RS2	RS1	RS0	Periodic Interrupt Rate	Square Wave Output Frequency	Periodic Interrupt Rate	Square Wave Output Frequency
0	0	0	0	None	None	None	None
0	0	0	1	30.517us	32.768kHz	3.90625ms	256Hz
0	0	1	0	61.035us	16.384kHz	7.8125ms	128Hz
0	0	1	1	122.070us	8.192kHz	122.070us	8.192kHz
0	1	0	0	244.141us	4.096kHz	244.141us	4.096kHz
0	1	0	1	488.281us	2.048kHz	488.281us	2.048kHz
0	1	1	0	976.562us	1.024kHz	976.562us	1.024kHz
0	1	1	1	1.953125ms	512Hz	1.953125ms	512Hz
1	0	0	0	3.90625ms	256Hz	3.90625ms	256Hz
1	0	0	1	7.8125ms	128Hz	7.8125ms	128Hz
1	0	1	0	15.625ms	64Hz	15.625ms	64Hz
1	0	1	1	31.25ms	32Hz	31.25ms	32Hz
1	1	0	0	62.5ms	16Hz	62.5ms	16Hz
1	1	0	1	125ms	8Hz	125ms	8Hz
1	1	1	0	250ms	4Hz	250ms	4Hz
1	1	1	1	500ms	2Hz	500ms	2Hz

표 52. RTC Register A의 Rate Bit 설정에 따른 Periodic Interrupt의 발생 비율(rate)와 SQ Output의 주파수

나중에 우리가 RTC를 이용해서 프로그램할 때 Register A에 있는 값을 이용해서 주기적인 인터럽트를 발생시키도록 할 때 사용할 수 있는 값이 될 수 있을 것이다. 이전 RTC에 대해서 대략이나마 이해했으리라 생각한다. 이제 본격적으로 RTC 디바이스 드라이버의 구현을 분석하도록 하자.

7.2. RTC의 초기화와 해제

RTC는 module로서 커널에 동적으로 link되는 것을 허용한다. 즉, 시스템의 핵심 구성요소가 아니라는 점이다. 만약 시스템에 중추적인 역할을 수행하는 subsystem이라면, 반드시 커널과 함께 loading이 되어야 할 것이다. 참고적으로 module의 정보부분을 차지하게되는 MODULE_AUTHOR() 및 MODULE_LICENSE() 매크로도 같이 보도록 하자.

```
module_init(rtc_init);
module_exit(rtc_exit);
...
MODULE_AUTHOR("Paul Gortmaker");
MODULE_LICENSE("GPL");
```

코드 472. module_init()/module_exit()

즉, 모듈의 개발자는 “Paul Gortmaker”가 되며, 이 모듈의 License가 GPL(GNU Public License)를 가진다는 것이다. 이것은 모듈의 컴파일하고 나서 생성되는 binary image에 포함되어, 나중에 모듈이 loading될 때 사용되는 정보들이다.

모듈로서 loading될 때나 혹은 kernel과 static하게 link되었을 경우에 초기화를 담당하는 rtc_init() 함수부터 살펴보기로 하자. 아래와 같이 ~/linux/drivers/char/rtc.c에 정의되어 있다.

```
...
#ifndef __sparc__
#include <asm/ebus.h>
#endif __sparc_v9__
#include <asm/isa.h>
#endif

static unsigned long rtc_port;
static int rtc_irq = PCI_IRQ_NONE;
#endif

static int rtc_has_irq = 1;
...
static int __init rtc_init(void)
{
#if defined(__alpha__) || defined(__mips__)
    unsigned int year, ctrl;
    unsigned long uip_watchdog;
    char *guess = NULL;
#endif
#ifndef __sparc__
    struct linux_ebus *ebus;
    struct linux_ebus_device *edev;
#endif __sparc_v9__
    struct isa_bridge *isa_br;
    struct isa_device *isa_dev;
#endif
#endif
```

```

#ifndef __sparc__
    for_each_ebus(ebus) {
        for_each_ebusdev(edev, ebus) {
            if(strcmp(edev->prom_name, "rtc") == 0) {
                rtc_port = edev->resource[0].start;
                rtc_irq = edev->irqs[0];
                goto found;
            }
        }
    }
#endif __sparc_v9__
    for_each_isa(isa_br) {
        for_each_isadev(isa_dev, isa_br) {
            if(strcmp(isa_dev->prom_name, "rtc") == 0) {
                rtc_port = isa_dev->resource.start;
                rtc_irq = isa_dev->irq;
                goto found;
            }
        }
    }
#endif
    printk(KERN_ERR "rtc_init: no PC rtc found\n");
    return -EIO;
found:
    if (rtc_irq == PCI_IRQ_NONE) {
        rtc_has_irq = 0;
        goto no_irq;
    }
}

```

코드 473. rtc_init() 함수의 정의

rtc_init() 함수의 첫부분에서는 먼저 시스템에 RTC가 있는지를 확인하는 절차를 지닌다. 즉, 각각의 architecture에 대해서 RTC가 연결될 수 있는 bus에 booting과정에서 찾은 디바이스 중에서 RTC가 있는지를 찾는다. Sparc의 경우에는 EBUS라는 것으로 Intel기반의 주변기기들을 불일수 있는 bus architecture를 가지고 있는데, for_each_ebus()라는 것을 통해서 각 EBUS에 대해서 RTC가 있는지를 찾는다. 이때 사용하는 resource(주로 I/O를 할 수 있는 영역)의 시작번지를 rtc_port로 두고, 사용하는 interrupt번호를 rtc_irq로 둔다. Sparc 9 architecture에서는 ISA bus상에서도 이와 같은 것이 있는지를 확인한다(for_each_isa(){ for_each_isadev(){}}). 없다면, RTC 디바이스를 찾을 수 없다는 에러 메시지를 출력하고 -EIO를 돌려준다. 있다고 하더라도 만약 PCI_IRQ_NONE으로 정의된 것과 같은 rtc_irq번호를 가진다면, RTC가 IRQ를 가지지 않는다고 표시한 후(rtc_has_irq = 0), no_irq로 jump한다. RTC를 찾았다면, ISA 디바이스로서 가지고 있는 RTC의 자원¹⁷⁹을 가져와서 이를 RTC가 사용하는 port 번호와 IRQ번호를 둔다.

```

/*
 * XXX Interrupt pin #7 in Espresso is shared between RTC and
 * PCI Slot 2 INTA# (and some INTx# in Slot 1). SA_INTERRUPT here
 * is asking for trouble with add-on boards. Change to SA_SHIRQ.
 */
if (request_irq(rtc_irq, rtc_interrupt, SA_INTERRUPT, "rtc", (void *)&rtc_port)) {
    /*
     * Standard way for sparc to print irq's is to use
     * __irq_itoa(). I think for EBus it's ok to use %d.
    */
}

```

¹⁷⁹ 일반적으로 디바이스가 사용하는 자원은 시스템의 특정 address 공간에 mapping된 번지와 IRQ가 될 수 있을 것이다. 즉, 모든 디바이스들은 어떤 식으로든 CPU에서 addressing할 수 있는 영역에 있어야 한다. 그렇지 않다면, CPU에서 제어할 방법이 없다.

```

        */
        printk(KERN_ERR "rtc: cannot register IRQ %d\n", rtc_irq);
        return -EIO;
    }

no_irq:
#else
    if (check_region (RTC_PORT (0), RTC_IO_EXTENT))
    {
        printk(KERN_ERR "rtc: I/O port %d is not free.\n", RTC_PORT (0));
        return -EIO;
    }
#endif RTC_IRQ
    if(request_irq(RTC_IRQ, rtc_interrupt, SA_INTERRUPT, "rtc", NULL))
    {
        /* Yeah right, seeing as irq 8 doesn't even hit the bus. */
        printk(KERN_ERR "rtc: IRQ %d is not free.\n", RTC_IRQ);
        return -EIO;
    }
#endif
    request_region(RTC_PORT(0), RTC_IO_EXTENT, "rtc");
#endif /* __sparc__ vs. others */

```

코드 474. rtc_init() 함수의 정의(계속)

request_irq()는 RTC가 사용하는 interrupt에 handler를 설치하도록 한다. 역시 아직까지는 sparc에서 사용하는 RTC에 대한 것이다. rtc_irq가 RTC가 사용하는 IRQ의 번호이며, rtc_interrupt는 handler를 SA_INTERRUPT는 RTC가 인터럽트를 공유하지 않는다는 것과 이 인터럽트를 처리하는 동안에는 다른 인터럽트들은 disable된다는 것을 보여주며, IRQ를 사용하는 기기가 RTC라는 것을 “rtc”라는 문자열을 사용해서 알려준다. 이 정보는 나중에 /proc/interrupts에 나오게 될 것이다. 또한 나중에 RTC의 ISR이 호출될 때 dev_id 포인터 필드에 넘겨질 값으로 rtc_port의 포인터를 사용한다. 예러가 있었다면, 예러 메시지를 출력하고 -EIO를 돌려준다.

no_irq에서부터는 IRQ가 없는 경우와 앞에서 IRQ를 설정한 경우에 대해서 모두 사용되는 부분이다. 이때 “#else”에서부터는 sparc architecture와는 관련되지 않은 부분으로, 먼저 RTC가 사용하는 IO space를 검사하도록 한다(check_region()). 만약 이미 사용하는 디바이스가 있다면, 오류 메시지를 출력하고 -EIO를 돌려준다. RTC_PORT()와 RTC_IO_EXTENT는 각각 아래와 같이 정의되어 있다.

```

/* ~/linux/drivers/char/rtc.c에서 가져왔음 */
#define RTC_IO_EXTENT 0x10      /* RTC에서 10 bytes의 time과 calendar를 가지는 공간이다. */

/* ~/linux/include/asm-i386/mc146818rtc.h에서 가져왔음 */
/*
 * Machine dependent access functions for RTC registers.
 */
#ifndef _ASM_MC146818RTC_H
#define _ASM_MC146818RTC_H
#include <asm/io.h>
#ifndef RTC_PORT
#define RTC_PORT(x)      (0x70 + (x))      /* RTC를 제어하기 위해서 레지스터의 base address가 0x700이다. */
#endif
#define RTC_ALWAYS_BCD     1      /* RTC operates in binary mode */
#endif
/*
 * The yet supported machines all access the RTC index register via
 * an ISA port access but the way to access the date register differs ...
 */
#define CMOS_READ(addr) ({ \

```

```

outb_p((addr),RTC_PORT(0)); \
inb_p(RTC_PORT(1)); \
})
#define CMOS_WRITE(val, addr) ({ \
outb_p((addr),RTC_PORT(0)); \
outb_p((val),RTC_PORT(1)); \
})
#define RTC_IRQ 8
#endif /* _ASM_MC146818RTC_H */

```

코드 475. RTC Port에 대한 읽고 쓰기 및 I/O port 영역의 정의

즉, RTC_IO_EXTENT는 대략 0x10으로 16개의 port(byte 단위로)를 사용할 수 있을 만큼의 공간을 차지하고, PC의 경우에는 기본적으로 RTC가 사용하는 port 번호로 0x70을 기본 입출력 주소(base I/O address)로 사용한다. CMOS_READ()와 CMOS_WRITE()는 각각 RTC가 사용하는 register에 입출력을 하기 위해서 사용하는 매크로로 읽기 위해서서는(CMOS_READ()), 읽을 주소를 RTC_PORT(0)에 쓰고, RTC_PORT(1)에서 데이터를 읽도록 하고 있으며, RTC에 데이터를 쓰기 위해서는 RTC_PORT(0)에 쓰려고 하는 주소를 넣고, 다시 RTC_PORT(1)에 데이터를 전달한다. 즉, RTC_PORT(0)(= 0x70 + 0)은 RTC 주소(address) 레지스터 가르키며, RTC의 데이터 레지스터는 RTC_PORT(1)(= 0x71)이 가르킨다. RTC_ALWAYS_BCD는 RTC가 항상 Binary Coded Decimal format으로 데이터를 가지도록 설정하며, RTC_IRQ는 8번 인터럽트를 사용한다고 본다. 따라서, RTC를 사용할 경우에는 /proc/interrupts에 8번에 다음과 같은 것을 찾을 수 있을 것이다.

IRQ번호	CPU번호	Interrupt Controller	사용하는 디바이스
-------	-------	----------------------	-----------

...			
8:	1	XT-PIC	rtc
...			

코드 476. /proc/interrupts에서의 RTC 관련 항목

RTC_IRQ가 정의되어 있다면, 당연히 request_irq()를 호출해서 ISR을 설정할 것이며, 이것이 성공하면 RTC가 사용하는 resource(I/O address space)를 할당받기 위해서 request_region()을 호출할 것이다. 이것이 성공적이라면 아래와 같은 것을 /proc/ioports에서 볼 수 있을 것이다.

시작번지	끝번지	: 사용하는 디바이스
------	-----	-------------

...		
0070	- 007F	: rtc
...		

코드 477. /proc/ioports에서의 RTC 관련 항목

물론 이것은 일반적인 Intel 기반의 PC 환경에서 본 것이다. Sparc과 같은 곳에서는 달라질 수 있다. 여기까지 문제없이 진행했다면, 우린 RTC가 사용하는 interrupt에 대한 handler를 설정했으면, 또한 우리의 RTC 디바이스 드라이버가 사용하는 I/O 영역 자원도 획득한 것이다.

```

misc_register(&rtc_dev);
create_proc_read_entry ("driver/rtc", 0, 0, rtc_read_proc, NULL);

```

코드 478. rtc_init() 함수의 정의(계속)

이젠 RTC를 misc_register()를 호출해서 miscellaneous(기타등등의) 디바이스 드라이버로 등록한다. 이 함수에서는 RTC를 miscellaneous 디바이스 종에서 minor 번호 135를 가지는 장치로 설정한다. 그리고

한가지 유념할 것은 RTC가 가지는 major번호는 MISC디바이스가 가지는 10번이라는 점이다.¹⁸⁰ 따라서, /dev/rtc는 major 10번, minor 135를 가지는 MISC 디바이스이다.

```
static struct file_operations rtc_fops = {
    owner:          THIS_MODULE,
    llseek:         no_llseek,
    read:          rtc_read,
#ifndef RTC_IRQ
    poll:          rtc_poll,
#endif
    ioctl:         rtc_ioctl,
    open:          rtc_open,
    release:       rtc_release,
    fasync:        rtc_fasync,
};

static struct miscdevice rtc_dev=
{
    RTC_MINOR,
    "rtc",
    &rtc_fops
};
```

코드 479. RTC의 file operation 구조체 및 miscdevice 구조체의 정의

rtc_dev구조체는 miscdevice 구조체로 정의된다. RTC_MINOR는 135라는 값을 가지며, 해당 디바이스의 이름은 “rtc”로 주어지며, 이 디바이스가 사용하는 file operation구조체는 rtc_fops를 사용한다. rtc_fops 구조체는 나중에 RTC를 접근하는 연산자로 사용될 것이므로 기억하도록 하자. 나중에 RTC를 사용하는 application에서는 major번호로 10번을 minor번호로 135번을 사용해서 RTC에 접근 가능하다(/dev/rtc를 open해서 사용하도록 한다.).

```
struct miscdevice
{
    int minor;                  /* MISC 디바이스로 등록된 디바이스의 minor번호 */
    const char *name;           /* MISC 디바이스로 등록된 디바이스의 이름 */
    struct file_operations *fops; /* 파일 연산자 vector(혹은 구조체)에 대한 포인터 */
    struct miscdevice *next, *prev; /* miscdevice구조체의 연결 리스트 */
    devfs_handle_t devfs_handle; /* device file system을 사용할 경우에 등록된 핸들의 포인터 */
};
```

코드 480. miscdevice 구조체의 정의

MISC 디바이스는 각종 일일이 class로 구분하기 힘든 디바이스를 한곳에 둑어서 관리할 수 있는 방법을 제공한다. 즉, RTC와 같이 character mode device로 등록해서 사용해도 상관없지만, 너무 많은 디바이스들이 실제로 존재하므로, 현재 Linux에서 사용하는 major번호가 한정된 까닭에 이와 같이 한꺼번에 둑어서 사용하면 더 효율적으로 major번호를 사용할 수 있기 때문이다. 각각의 major번호에 총 256개의 디바이스가 올 수 있기에 MISC 디바이스도 총 256개까지가 이론적으로 가능하다.

```
#if defined(__alpha__) || defined(__mips__)
    rtc_freq = HZ;
    /* Each operating system on an Alpha uses its own epoch.
       Let's try to guess which one we are using now. */
    uip_watchdog = jiffies;
```

¹⁸⁰ ~/linux/include/linux/major.h에 각종 디바이스가 가지는 major번호에 대한 정의를 찾을 수 있을 것이다. 여기에 MISC_MAJOR로 10번을 사용하고 있음을 알 수 있을 것이다.

```

if (rtc_is_updating() != 0)
    while (jiffies - uip_watchdog < 2*HZ/100) {
        barrier();
        cpu_relax();
    }
    spin_lock_irq(&rtc_lock);
    year = CMOS_READ RTC_YEAR;
    ctrl = CMOS_READ RTC_CONTROL;
    spin_unlock_irq(&rtc_lock);
    if (!(ctrl & RTC_DM_BINARY) || RTC_ALWAYS_BCD)
        BCD_TO_BIN(year); /* This should never happen... */
    if (year < 20) {
        epoch = 2000;
        guess = "SRM (post-2000)";
    } else if (year >= 20 && year < 48) {
        epoch = 1980;
        guess = "ARC console";
    } else if (year >= 48 && year < 72) {
        epoch = 1952;
        guess = "Digital UNIX";
    #if defined(__mips__)
    } else if (year >= 72 && year < 74) {
        epoch = 2000;
        guess = "Digital DECstation";
    #else
    } else if (year >= 70) {
        epoch = 1900;
        guess = "Standard PC (1900)";
    #endif
    }
    if (guess)
        printk(KERN_INFO "rtc: %s epoch (%lu) detected\n", guess, epoch);
#endif

```

코드 481. rtc_init() 함수의 정의(계속)

Alpha나 MIPS architecture의 경우에 사용한다. 여기서는 각각의 architecture에 맞게 현재 년도를 알기 위해서 사용하는 함수를 보여준다. CMOS_READ()에서 RTC_YEAR과 RTC_CONTROL을 사용했는데 각각은 아래와 같은 값으로 정의되어 있다. 이때 CMOS를 읽고 쓰는 동작에서 interrupt가 발생해서 잘못된 값을 읽을 가능성이 있기에 spin_lock_irq()를 호출해서 IRQlock을 설정하고, 나중에 읽기가 끝난 후에 다시 spin_unlock_irq()를 호출해서 lock을 풀어준다.

rtc_is_updating()은 RTC의 정보가 update하는 중인지를 알아보기 위한 것이다. 만약 update가 진행중이라면, 2초($= 2 \times Hz / 100$)간 기다리도록 한다.

```

/* ~/include/asm-i386/processor.h에서 */
/* REP NOP (PAUSE) is a good thing to insert into busy-wait loops. */
static inline void rep_nop(void)
{
    __asm__ __volatile__("rep;nop");
}
#define cpu_relax() rep_nop()

/* ~/include/linux/kernel.h에서 */
/* Optimization barrier */
/* The "volatile" is due to gcc bugs */
#define barrier() __asm__ __volatile__ ("": : :"memory")

```

코드 482. cpu_relax()와 barrier()의 정의

barrier()는 GCC의 “volatile”이 가지고 있는 optimization에 대한 오류를 정정하기 위해서 있는 것으로 모든 연산이 주어진 순서대로 끝나서, 메모리의 update가 완료가 될 때까지 기다리도록 만든다. cpu_relax()는 busy wait(대기하면서 어떤 상황이 발생하도록 기다린다.)하도록 만들어준다. 즉, 앞에서 보여준 코드는 while loop를 돌면서 2초간 실제적으로는 아무런 일도 하지 않고, 기다리는 과정이다.

제어(control) 정보를 읽어서, 읽은 연도의 값이 BCD를 나타내는지 아니면, binary 값을 가지는지를 확인한다. 이때 BCD값을 가진다면, 이를 다시 binary값으로 변환하기 위해서 BCD_TO_BIN()을 호출한다. 나머지는 각각의 시스템에 대한 읽혀진 연도를 변환하는 과정이다.

```
#ifndef BCD_TO_BIN
#define BCD_TO_BIN(val) (((val)&15) + ((val)>>4)*10)
#endif

#ifndef BIN_TO_BCD
#define BIN_TO_BCD(val) (((val)/10)<<4) + (val)%10
#endif
```

코드 483. BCD와 Binary 모드간의 변형(conversion)

BCD_TO_BIN()은 BCD로 된 값을 binary값으로 변환하는 일을 하며, BIN_TO_BCD는 이를 역으로 수행한다. 이때 변환된 값이 1 byte(= 8 bit)이라는 점에 유의하도록 하자.

```
*****
* register summary
*****  

#define RTC_SECONDS          0
#define RTC_SECONDS_ALARM    1
#define RTC_MINUTES          2
#define RTC_MINUTES_ALARM    3
#define RTC_HOURS            4
#define RTC_HOURS_ALARM      5
/* RTC_*_alarm is always true if 2 MSBs are set */
#define RTC_ALARM_DONT_CARE  0xC0  

#define RTC_DAY_OF_WEEK       6
#define RTC_DAY_OF_MONTH      7
#define RTC_MONTH             8
#define RTC_YEAR              9  

/* control registers - Moto names
 */
#define RTC_REG_A             10
#define RTC_REG_B             11
#define RTC_REG_C             12
#define RTC_REG_D             13
```

코드 484. MC146818 RTC의 register summary

RTC가 가지고 있는 레지스터들의 물리적인 주소는 이미 앞의 그림들에서 보았다. 따라서, 해당 주소 값을 표시하는 상수 값들이 이곳에 정의되어 있다. CMOS_READ()와 CMOS_WRITE()를 통해서 이와 같은 주소를 접근하게 된다. 여기서 레지스터 A, B, C, D는 앞에서 설명했듯이 RTC 제어(control)의 목적으로 사용하게 된다.

```
#if RTC_IRQ
```

```

if (rtc_has_irq == 0)
    goto no_irq2;
init_timer(&rtc_irq_timer);
rtc_irq_timer.function = rtc_dropped_irq;
spin_lock_irq(&rtc_lock);
/* Initialize periodic freq. to CMOS reset default, which is 1024Hz */
CMOS_WRITE(((CMOS_READ(RTC_FREQ_SELECT) & 0xF0) | 0x06), RTC_FREQ_SELECT);
spin_unlock_irq(&rtc_lock);
rtc_freq = 1024;
no_irq2:
#endif
    printk(KERN_INFO "Real Time Clock Driver v" RTC_VERSION "\n");
    return 0;
}

```

코드 485. rtc_init() 함수의 정의(계속)

만약 RTC_IRQ가 정의되어 있다면, '#if ~ endif'를 수행한다. rtc_has_irq가 0인 값을 가진다면, 바로 no_irq2로 제어를 옮기게 되며, 그렇지 않다면, rtc_irq_timer를 설정한다(init_timer). Timer에서 수행할 함수는 rtc_dropped_irq()가 될 것이다. 다시 RTC에 출력을 하기 위해서 spin_lock_irq()와 spin_unlock_irq()로 끌어준다. CMOS_WRITE()는 여기서 Register A에 제어정보를 기록하기 위해서 사용했다. RTC의 주기를 1024(>rtc_freq)로 맞춰준다. 나머지는 RTC의 정보를 보여주고 0을 복귀 값으로 알려준다. 아래에 보여주는 것은 제어 목적으로 사용하는 레지스터 A, B, C, D의 field에 대한 정의이다.

```

/****************************************************************************
 * register details
 ****
#define RTC_FREQ_SELECT      RTC_REG_A

/* update-in-progress - set to "1" 244 microsecs before RTC goes off the bus,
 * reset after update (may take 1.984ms @ 32768Hz RefClock) is complete,
 * totalling to a max high interval of 2.228 ms.
 */
#define RTC_UIP             0x80          /* Update in Progress */
#define RTC_DIV_CTL         0x70          /* Divider Control */
    /* divider control: refclock values 4.194 / 1.049 MHz / 32.768 kHz */
#define RTC_REF_CLK_4MHZ   0x00          /* Reference Clock */
#define RTC_REF_CLK_1MHZ   0x10
#define RTC_REF_CLK_32KHZ  0x20
    /* 2 values for divider stage reset, others for "testing purposes only" */
#define RTC_DIV_RESET1     0x60          /* Divider Reset */
#define RTC_DIV_RESET2     0x70
    /* Periodic intr. / Square wave rate select. 0=none, 1=32.8kHz,... 15=2Hz */
#define RTC_RATE_SELECT    0x0F          /* RTC Rate Select */

/***
#define RTC_CONTROL        RTC_REG_B
#define RTC_SET             0x80          /* disable updates for clock setting */
#define RTC_PIE             0x40          /* periodic interrupt enable */
#define RTC_AIE             0x20          /* alarm interrupt enable */
#define RTC_UIE             0x10          /* update-finished interrupt enable */
#define RTC_SQWE            0x08          /* enable square-wave output */
#define RTC_DM_BINARY       0x04          /* all time/date values are BCD if clear */
#define RTC_24H              0x02          /* 24 hour mode - else hours bit 7 means pm */
#define RTC_DST_EN           0x01/* auto switch DST - works f. USA only */

****/

```

```
#define RTC_INTR_FLAGS RTC_REG_C
/* caution - cleared by read */
#define RTC_IRQF 0x80          /* any of the following 3 is active */
#define RTC_PF 0x40            /* Periodic Interrupt Flag */
#define RTC_AF 0x20            /* Alarm Interrupt Flag */
#define RTC_UF 0x10            /* Update Interrupt Flag */

/*****************/
#define RTC_VALID    RTC_REG_D
#define RTC_VRT 0x80          /* valid RAM and time */
/*****************/
```

코드 486. RTC 레지스터의 상세(Specification)

따라서, 위에서 CMOS_WRITE()를 통해서 하는 일은, 먼저 CMOS_READ를 통해서 레지스터 A에 있는 Base 값을 읽고, 이 값에 0x06을 OR시켜서, 다시 레지스터 A에(RTC_FREQ_SELECT)에 write하는 것이다. 이것은 [표 52]에서 정의한 Register A의 값 중에서 1024Hz를 발생시키도록 만들어 준다.

```
static void __exit rtc_exit (void)
{
    remove_proc_entry ("driver/rtc", NULL);
    misc_deregister(&rtc_dev);
#ifndef __sparc__
    if (rtc_has_irq)
        free_irq (rtc_irq, &rtc_port);
#else
    release_region (RTC_PORT (0), RTC_IO_EXTENT);
#endif RTC_IRQ
    if (rtc_has_irq)
        free_irq (RTC_IRQ, NULL);
#endif /* __sparc__ */
}
```

코드 487. rtc_exit() 함수의 정의

rtc_exit() 함수는 rtc_init() 함수의 역할을 반대로 수행한다. 먼저 앞에서 등록했던 proc 파일 시스템의 entry를 삭제하고(remove_proc_entry()), misc_deregister()를 호출해서 miscellaneous device등록을 해제한다. sparc architecture의 경우에는 RTC가 irq를 가지고 있는 경우에 대해서 free_irq()를 호출해서 ISR등록을 풀어준다(release). Sparc이 아닌 architecture에서는 release_region()을 호출해서 RTC가 사용하는 I/O 영역(region)에 대해서 점유한 자원을 해제한다. 또한 RTC_IRQ가 정의된 경우에 대해서 rtc_has_irq 플래그가 설정되어 있다면, free_irq()를 호출해서 ISR등록을 해제한다. 여기까지 해서 사용한 시스템의 자원에 대한 것은 모두 해제된다.

```
#if RTC_IRQ
static void rtc_dropped_irq(unsigned long data)
{
    unsigned long freq;

    spin_lock_irq (&rtc_lock);
    /* Just in case someone disabled the timer from behind our back... */
    if (rtc_status & RTC_TIMER_ON)
        mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);
    rtc_irq_data += ((rtc_freq/HZ)<<8);
    rtc_irq_data &= ~0xff;
    rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);      /* restart */
    freq = rtc_freq;
```

```

spin_unlock_irq(&rtc_lock);
printk(KERN_WARNING "rtc: lost some interrupts at %ldHz.\n", freq);
/* Now we have new data */
wake_up_interruptible(&rtc_wait);
kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);
}
#endif

```

코드 488. rtc_dropped_irq() 함수의 정의

rtc_dropped_irq() 함수는 RTC가 설정한 timer에 의해서 커널이 호출하는 함수이다. 이 함수는 RTC의 interrupt주기(period)가 너무 짧아질 경우, 지나치게 많은 interrupt를 발생시키게 되는데, 이때는 시스템이 interrupt를 다 처리하지 못하고, 그냥 RTC에 남아 있게 된다. 이렇게 됨으로써 interrupt handler에서 interrupt status를 읽어내지 못해서 발생하는 interrupt의 blocking 현상(RTC가 interrupt를 생성하지 못한다.)을 방지하기 위해서 커널이 주기적으로 interrupt status flag를 읽어나가게하는 역할을 한다. RTC에 대한 접근을 하기전에 spin lock을 설정한다(spin_lock_irq()). RTC의 현재 상태를 알려주는 rtc_status에 timer가 설정되어 있다면(RTC_TIMER_ON), 이 timer를 다음 발생시간을 변경한다. 이때 사용하는 시간은 현재 시간(jiffies)에 HZ(= 100)/rtc_freq와 2 x HZ/100(= 대략 2초)을 더한 시간을 주도록 한다. 즉, 현재 시간으로부터 2초이후에 RTC interrupt가 발생하는 rtc_freq의 시간에 timer가 수행되도록 만들어준다. rtc_irq_data에 rtc_freq/HZ를 8 bit 좌측으로 shift(x 256)을 한 값을 더하고, rtc_irq_data의 하위 8 bit을 clear한 후, 다시 RTC Register C의 interrupt들의 상황을 나타낸는 flag값을 읽어서 OR시킨다. 이것으로 interrupt flag들을 읽었기에 RTC Register C의 특성상, Register C의 각 bit field들은 전부 clear된다. 즉, 새로운 interrupt가 발생할 수 있도록 만들어준 것이다. freq에는 rtc_freq를 넣고, RTC Register들에 대한 접근이 끝났으므로, spin_unlock_irq()를 호출한다. printk()는 RTC가 특정 rate에서 발생하는 interrupt를 잃어버렸다는 정보를 나타내게 되며, rtc_wait queue에서 RTC를 읽기위해, 잠들어 있을지도 모르는 프로세스를 깨운다(wake_up_interruptible()). kill_fasync()는 RTC에 대해서 비동기(asynchronous) 입출력을 설정한 프로세스들에게, RTC에 대한 비동기 입출력이 끝났음을 알려주기 위한 것이다. 이때 전달되는 signal은 SIGIO이다. 각각의 프로세스들은 이 signal에 맞는 동작을 취할 수 있는 handler를 가지고 있으면 될 것이다.

```

/*
 * Bits in rtc_status. (6 bits of room for future expansion)
 */

#define RTC_IS_OPEN          0x01    /* means /dev/rtc is in use      */
#define RTC_TIMER_ON          0x02    /* missed irq timer active     */

/*
 * rtc_status is never changed by rtc_interrupt, and ioctl/open/close is
 * protected by the big kernel lock. However, ioctl can still disable the timer
 * in rtc_status and then with del_timer after the interrupt has read
 * rtc_status but before mod_timer is called, which would then reenable the
 * timer (but you would need to have an awful timing before you'd trip on it)
 */
static unsigned long rtc_status = 0;    /* bitmapped status byte.      */
static unsigned long rtc_freq = 0;       /* Current periodic IRQ rate */
static unsigned long rtc_irq_data = 0;   /* our output to the world    */

```

코드 489. RTC의 status, frequency, interrupt data 변수의 정의

현재 RTC의 상태를 표현하는 rtc_status에는 2개의 값 만을 사용한다. 즉, RTC가 open되어 사용되고 있나는 뜻인 RTC_IS_OPEN과 timer가 설정되어 있다는 뜻으로 사용하는 RTC_TIMER_ON이 그것이다. rtc_freq와 rtc_irq_data등도 RTC의 현재 상태를 표시하기 위해서 사용하는 전역 변수들이다.

```
/*
```

```
* Returns true if a clock update is in progress
*/
/* FIXME shouldn't this be above rtc_init to make it fully inlined? */
static inline unsigned char rtc_is_updating(void)
{
    unsigned char uip;

    spin_lock_irq(&rtc_lock);
    uip = (CMOS_READ RTC_FREQ_SELECT) & RTC_UIP;
    spin_unlock_irq(&rtc_lock);
    return uip;
}
```

코드 490. rtc_is_updating() 함수의 정의

rtc_is_updating() 함수는 Register A를 읽어서, Update가 진행주인지를 알려주는 함수이다¹⁸¹. Register A를 읽어서, RTC_UIP(=0x80)와 AND한 후 복귀 코드로 돌려준다.

7.3. RTC의 Open과 Release

```
/*
 *
 * We enforce only one user at a time here with the open/close.
 * Also clear the previous interrupt data on an open, and clean
 * up things on a close.
 */
/* We use rtc_lock to protect against concurrent opens. So the BKL is not
 * needed here. Or anywhere else in this driver. */
static int rtc_open(struct inode *inode, struct file *file)
{
    spin_lock_irq (&rtc_lock);

    if(rtc_status & RTC_IS_OPEN)
        goto out_busy;
    rtc_status |= RTC_IS_OPEN;
    rtc_irq_data = 0;
    spin_unlock_irq (&rtc_lock);
    return 0;
out_busy:
    spin_unlock_irq (&rtc_lock);
    return -EBUSY;
}
```

코드 491. rtc_open() 함수의 정의

rtc_open() 함수는 RTC를 사용하기 위해서 프로세스가 가장 먼저 수행해야하는 함수이다. 여기서 중요한 점은 한번에 한 프로세스만 이 함수와 rtc_release() 함수를 사용하도록 허락한다는 점이다. 만약 RTC_IS_OPEN이 rtc_status에 이미 설정되어 있다면, RTC가 이미 open되어 다른 사용자가 쓰고 있다는 뜻이므로, out_busy로 바로 제어를 옮긴다. -EBUSY가 복귀 코드로 주어질 것이다. 그렇지 않다면, rtc_status에는 RTC_IS_OPEN이 설정되고, rtc_irq_data는 0으로 초기화된다. 복귀 코드는 0이 될 것이다.

```
static int rtc_release(struct inode *inode, struct file *file)
{
#if RTC_IRQ
    unsigned char tmp;
```

¹⁸¹ Update가 일어나거나 일어날 것이라면 1을 가지고, update가 진행중이지 않거나, 최소 244 us동안은 update가 일어나지 않을 것이면 0이라는 값을 가질 것이다.

```

if (rtc_has_irq == 0)
    goto no_irq;
/*
 * Turn off all interrupts once the device is no longer
 * in use, and clear the data.
 */
spin_lock_irq(&rtc_lock);
tmp = CMOS_READ(RTC_CONTROL);
tmp &= ~RTC_PIE;
tmp &= ~RTC_AIE;
tmp &= ~RTC_UIE;
CMOS_WRITE(tmp, RTC_CONTROL);
CMOS_READ(RTC_INTR_FLAGS);
if (rtc_status & RTC_TIMER_ON) {
    rtc_status &= ~RTC_TIMER_ON;
    del_timer(&rtc_irq_timer);
}
spin_unlock_irq(&rtc_lock);
if (file->f_flags & FASYNC) {
    rtc_fasync (-1, file, 0);
}
no_irq:
#endif
spin_lock_irq (&rtc_lock);
rtc_irq_data = 0;
spin_unlock_irq (&rtc_lock);
/* No need for locking -- nobody else can do anything until this rmw is
 * committed, and no timer is running.*/
rtc_status &= ~RTC_IS_OPEN;
return 0;
}

```

코드 492. rtc_release() 함수의 정의

rtc_release() 함수는 RTC가 interrupt를 가지는 경우에 대해서 해주어야 할일이 많다. 만약 RTC가 interrupt를 발생시키지 않는다면, RTC를 사용하는 중에 update가 된 rtc_irq_data를 reset(0으로 만든다.)하는 일과 rtc_status에서 RTC_IS_OPEN을 clear하는 일만 수행한다.

만약 RTC가 interrupt를 사용한다면, rtc_has_irq값을 보고 0과 같다면, no_irq로 제어를 옮겨서 앞에서 해준 일을 처리하게되며, RTC Register B(=RTC_CONTROL)을 읽어서(CMOS_READ()), 해당 interrupt를 전부 disable시킨 후, 다시 Register B에 써주게된다(CMOS_WRITE()). 이때 RTC의 interrupt status로 있어서 interrupt flag를 clear시킨다¹⁸². 또한 timer가 설정되어 있다면(RTC_TIMER_ON), 이 timer를 제거하기 위해서 del_timer()를 호출하고, rtc_status를 update한다. 또한 RTC를 open한 프로세스가 file flag에 FASYNC를 설정했다면, rtc_fasync()¹⁸³를 호출한다. 이때 호출에 넘겨지는 파라미터로는 첫번째가 -1, 두번째가 넘겨받은 file이 되며, 세번째가 0이다. 나중에 다시 이야기 하겠지만, rtc_fasync() 함수는 file operation에 들어가는 함수로, 역할은 비동기적인 입출력을 하기위해서 호출되는 함수이다. 즉, 전달된 file이라는 구조체에 f_flags에 FASYNC가 설정된 경우에 커널에서 비동기 입출력을 하기 위해서 호출된다. 비동기적인 입출력이 완료가 되었을 때는 커널에서 사용자 프로세스에 SIGIO라는 signal을 보내게된다. Signal을 보내기 위해서는 앞에서 보았듯이 kill_fasync()함수를 사용해야 한다. 여기서 fd(file descriptor)에 -1을주고, on에 0을 주어서 호출한 것은 asynchronous notification을 기다리고 있는 queue에서 제거하고자 할 때 사용하는 방법이다. 나머지는 interrupt를 사용하지 않는 경우와 동일하다.

¹⁸² 앞에서 이미 보았듯이 Register C는 interrupt flag를 설정하게 되는데 사용되며, 읽는 것으로 clear되는 레지스터이다.

¹⁸³ rtc_fasync() 함수는 다음과 같이 정의되어 있다. “static int rtc_fasync(int fd, struct file *filp, int on);”

7.4. RTC의 Read

RTC에서 사용자 프로세스가 읽어가는 데이터는 RTC에서 발생한 인터럽트 정보이다. 따라서, `rtc_irq_data`를 일어갈 것이다.

```
static ssize_t rtc_read(struct file *file, char *buf, size_t count, loff_t *ppos)
{
#if !RTC_IRQ
    return -EIO;
#else
    DECLARE_WAITQUEUE(wait, current);
    unsigned long data;
    ssize_t retval;

    if (rtc_has_irq == 0)
        return -EIO;
    if (count < sizeof(unsigned long))
        return -EINVAL;
    add_wait_queue(&rtc_wait, &wait);
    current->state = TASK_INTERRUPTIBLE;
    do {
        /* First make it right. Then make it fast. Putting this whole
         * block within the parentheses of a while would be too
         * confusing. And no, xchg() is not the answer. */
        spin_lock_irq(&rtc_lock);
        data = rtc_irq_data;
        rtc_irq_data = 0;
        spin_unlock_irq(&rtc_lock);
        if (data != 0)
            break;
        if (file->f_flags & O_NONBLOCK) {
            retval = -EAGAIN;
            goto out;
        }
        if (signal_pending(current)) {
            retval = -ERESTARTSYS;
            goto out;
        }
        schedule();
    } while (1);
    retval = put_user(data, (unsigned long *)buf);
    if (!retval)
        retval = sizeof(unsigned long);
out:
    current->state = TASK_RUNNING;
    remove_wait_queue(&rtc_wait, &wait);
    return retval;
#endif
}
```

코드 493. `rtc_read()` 함수의 정의

RTC의 interrupt 정보를 일거가기에 `RTC_IRQ`가 정의되지 않았다면, 단순히 `-EIO`(Error I/O)를 돌려준다. 또한 RTC가 인터럽트를 획득했다는 것을 알려주는 `rtc_has_irq`가 0인 경우에도 `-EIO`를 돌려준다. 그렇지 않다면, `count`값이 4 bytes(`=sizeof(unsigned long)`)보다 작은지 확인해서 더 작다면 `-EINVAL`을 돌려준다. 즉, RTC의 interrupt 정보를 가지는 `rtc_irq_data`의 크기보다 작은가를 확인하는 것이다. 이젠 RTC에서 interrupt관련 데이터를 읽어가기 위해서, 현재 프로세스를 wait queue에 넣기위한 일을 한다. 즉,

대기상태로 두고, 나중에 인터럽트가 발생했을 때 데이터를 읽어가기 위한 것이다. `add_wait-queue()`는 대기 queue에 프로세스를 두는 일을 한다. 프로세스의 상태는 이제 `TASK_INTERRUPTIBLE`이 될 것이다. 아직은 `schedule()`이라는 함수를 호출하지 않았기에, 현재 실행중인 프로세스가 계속 진행한다. 따라서, `do{} while()` loop를 진입하게 된다. 전역 데이터를 읽기 위해 spin lock을 설정하고, `rtc_irq_data`의 값을 얻어온다. 이때 만약 이렇게 얻어온 데이터가 0이 아니라면, 즉 RTC의 interrupt가 있었다고 한다면, loop를 빠져나오고, 그렇지 않다면, RTC와 관련된 file object의 `f_flags`에 `O_NONBLOCK`이 설정되어 있는지를 확인한다. 만약 설정되어 있었다면 현재 프로세스를 blocking 상태로 만들지 말라는 말이므로, `-EAGAIN`(Error Again : 다시 한번 시도하라)를 복귀 값을 두고 `out`으로 제어를 옮긴다. 이젠 혹시 현재 프로세스에 대해서 pending된 signal이 있는지를 본다. 즉, 앞에서 프로세스의 상태를 `TASK_INTERRUPTIBLE`로 두고 wait queue에 들어가기 때문에 signal의 pending 여부를 확인하는 것이다. 만약 pending된 signal이 있다면, 복귀 값으로 `-ERESTARTSYS`(Error Restart System Call)를 설정하고, `out`으로 제어를 옮긴다. loop를 다시 돌기전에 이젠 다른 프로세스에게 CPU를 넘겨주기 위해서 `schedule()`함수를 호출한다. loop에서 빠져나왔다면, 전달할 데이터를 가진다는 이야기다. 이때는 `put_user()`를 호출해서 사용자 영역의 buffer에 읽은 RTC의 interrupt 데이터를 복사한다. 만약 `retval`의 값이 0을 가진다면, 즉, 별다른 오류가 앞에서 발생하지 않았다면 복사한 데이터의 크기를 복귀 값에 넣어준다(`sizeof(unsigned long)`). `out`에서 부터는 프로세스를 원래의 상태로 바꾸는 일을 한다. 즉, `TASK_RUNNING`로 두고, RTC의 wait queue로부터 현재 프로세스를 제거하기 위해 `remove_wait_queue()`를 호출한다. 복귀 값은 앞에서 설정한 값이 넘겨질 것이다.

7.5. RTC의 Poll

RTC에서는 poll에 대해서 interrupt가 정의된 경우에만 사용하도록 한정하고 있다. 즉, RTC가 interrupt를 사용할 수 있도록 정의하는 `RTC_IRQ`가 컴파일 시에 정의(definition)가 되어 있어야 할 것이다.

```
#if RTC_IRQ
/* Called without the kernel lock - fine */
static unsigned int rtc_poll(struct file *file, poll_table *wait)
{
    unsigned long l;

    if (rtc_has_irq == 0)
        return 0;
    poll_wait(file, &rtc_wait, wait);
    spin_lock_irq (&rtc_lock);
    l = rtc_irq_data;
    spin_unlock_irq (&rtc_lock);
    if (l != 0)
        return POLLIN | POLLRDNORM;
    return 0;
}
#endif
```

코드 494. `rtc_poll()` 함수의 정의

기본적으로 poll은 응용프로그램에서 디바이스 드라이버에 대해, 입출력이 가능한가를 알아보기 위해서 호출된다. Linux의 일반적인 poll 함수 구현방법인 `poll_wait()`를 사용하게 되며, 만약 RTC에서 읽어갈 데이터가 있다면, `POLLIN`과 `POLLRDNORM`을 OR시킨 값을 돌려준다. 응용프로그램은 RTC를 polling하다가 준비가 된 데이터가 있다면 읽어갈 것이다. `rtc_has_irq`가 0이라면, 곧바로 0을 돌려준다. 그렇지 않다면, `poll_wait()`를 호출해서 RTC의 poll wait를 위한 구조를 등록하게되며, spin lock을 통해서 interrupt를 `rtc_irq_data` 값을 읽어낸다. 만약 이렇게 읽은 `rtc_irq_data`가 있다면, 사용자 프로세스에게 읽을 준비가 되어 있다고 알려주고, 그렇지 않다면 0을 돌려주도록 한다.

7.6. RTC의 Fasync

```
static int rtc_fasync (int fd, struct file *filp, int on)
```

```
{
    return fasync_helper(fd, filp, on, &rtc_async_queue);
}
```

코드 495. rtc_fasync() 함수의 정의

이 함수는 프로세스가 비동기(asynchronous) 입출력을 요구할 경우에 호출되는 함수이다. 즉, 입출력이 완료가 되는 시점까지 기다리겠다는 것이 아니라(동기적: synchronous), 입출력 요구를 하고, 입출력이 끝나면 알려달라는 뜻으로 사용한다. 즉, 사용자 프로세스는 입출력을 요구하고 바로 복귀하게 되며, 커널이 입출력이 종료되는 시점에 이것을 시그널(signal)을 통해서 알려주게 된다. 따라서, 프로세스에서는 SIGIO라는 시그널을 받게되면, 이 핸들러내에서 입출력과 관련된 버퍼관리와 같은 작업을 해주면 될 것이다. 이러한 메커니즘을 가지기 위해서는 입출력의 완료를 알려주기 위해서 관리하는 List가 필요하게 되며, 여기서 보듯이 rtc_async_queue라는 변수를 사용한다. rtc_async_queue는 fasync_struct라는 구조체로 정의된 변수이다. 여기서 잠시 fasync_struct 구조체를 보도록 하자.

```
struct fasync_struct {
    int      magic;           /* Magic Number */
    int      fa_fd;          /* File Descriptor Number */
    struct  fasync_struct   *fa_next; /* Singly linked list */
    struct  file            *fa_file;  /* File Object pointer */
};
```

코드 496. fasync_struct 구조체의 정의

즉, fasync_struct는 일종의 연결리스트를 생성하는 하나의 entry로서 사용되는 구조체이며, 입출력과 관련된 파일에 대한 정보를 유지하기 위해서 사용된다.

```
int fasync_helper(int fd, struct file * filp, int on, struct fasync_struct **fapp)
{
    struct fasync_struct *fa, **fp;
    struct fasync_struct *new = NULL;
    int result = 0;

    if (on) {
        new = kmem_cache_alloc(fasync_cache, SLAB_KERNEL);
        if (!new)
            return -ENOMEM;
    }
    write_lock_irq(&fasync_lock);
    for (fp = fapp; (fa = *fp) != NULL; fp = &fa->fa_next) {
        if (fa->fa_file == filp) {
            if (on) {
                fa->fa_fd = fd;
                kmem_cache_free(fasync_cache, new);
            } else {
                *fp = fa->fa_next;
                kmem_cache_free(fasync_cache, fa);
                result = 1;
            }
            goto out;
        }
    }
    if (on) {
        new->magic = FASYNC_MAGIC;
        new->fa_file = filp;
        new->fa_fd = fd;
        new->fa_next = *fapp;
    }
}
```

```

        *fapp = new;
        result = 1;
    }
out:
    write_unlock_irq(&fasync_lock);
    return result;
}

```

코드 497. fasync_helper() 함수의 정의

Linux에서는 파일에 대한 비동기 입출력을 위해서 사용할 함수로 fasync_helper()라는 것을 제공한다. 기본적으로 비동기 입출력에 대한 fasync_struct의 등록을 하는 일을 수행하게 되며, 여기에 등록된 자료구조들은 나중에 kill_fasync()함수를 이용해서 입출력이 끝났을 때 시그널을 받을 수 있도록 만들어준다. fasync_helper() 함수가 넘겨받은 파라미터 값은 파일 디스크립터 번호(fd), file object에 대한 포인터(filp)와 비동기 입출력 리스트를 가르키는 포인터이다. 이와 더불어 리스트에서 삭제 혹은 삽입할 것을 나타내는 flag값으로 on이라는 것을 사용한다. on이 설정된 경우에는 새로운 fasync_struct를 할당받기 위해서 kmeme_cache_alloc()이라는 것을 사용해서 slab allocator로부터 메모리를 요구한다. 할당할 수 없다면, 당연히 -ENOMEM을 돌려줄 것이다. 이전에 이미 등록된 것이 있는가를 확인하고, on의 값에 따라서, 삽입 혹은 삭제를 하기위한 준비를 한다. 리스트로 관리되는 fasync_struct 구조체들을 하나씩 추적하며, 넘겨받은 인자의 파일 object 포인터와 등록된 파일 object포인터가 같은 것을 찾는다. 만약 같은 것이 있다면, on이 설정(set)되었다면, file descriptor의 값을 갱신하고, 할당했던 fasync_struct 구조체를 해제한다. 만약 on이 설정되지 않았다면(clear), 해당 entry를 삭제하라는 것으로 해석해서, 등록된 fasync_struct 구조체를 제거한다. 결과(result)는 이때 1로 설정된다. 여기까지 진행했다면, out으로 제어를 옮긴다.

만약 같은 entry를 찾지 못한다면, 이전 on의 값이 설정된 경우에는 새롭게 할당된 fasync_struct의 각 필드를 초기화한다. FASYNC_MAGIC(=0x4601)을 magic number로 주고, file object 포인터와 file descriptor 및 연결리스트 추가등을 해준다. 이때도 결과 값(result)은 1로 주어진다. 즉, 주어진 연산을 무리없이 다 처리했기 때문이다. 나올때는 fasync_struct의 연결리스트에 설정했던 write lock을 해제하고, result를 돌려준다.

```

void __kill_fasync(struct fasync_struct *fa, int sig, int band)
{
    while (fa) {
        struct fown_struct * fown;
        if (fa->magic != FASYNC_MAGIC) {
            printk(KERN_ERR "kill_fasync: bad magic number in "
                  "fasync_struct!\n");
            return;
        }
        fown = &fa->fa_file->f_owner;
        /* Don't send SIGURG to processes which have not set a
           queued signum: SIGURG has its own default signalling
           mechanism. */
        if (fown->pid && !(sig == SIGURG && fown->signum == 0))
            send_sigio(fown, fa->fa_fd, band);
        fa = fa->fa_next;
    }
}

void kill_fasync(struct fasync_struct **fp, int sig, int band)
{
    read_lock(&fasync_lock);
    __kill_fasync(*fp, sig, band);
    read_unlock(&fasync_lock);
}

```

코드 498. kill_fasync() 함수의 정의

`kill_fasync()` 함수는 asynchronous I/O를 대기하는 프로세스들에 시그널을 보내기 위해서 사용하는 함수이다. 해당 `fasync_struct`의 연결리스트에 대한 포인터와 보낼 시그널의 번호 및 out-of-band 데이터와 같은 알림을 하기 위한 band¹⁸⁴를 넘겨받는다. `fasync_helper()`와는 달리 여기서는 연산의 앞뒤 부분에 `read lock`을 설정하고 있음에 주의하도록 하자. 실제적인 처리는 `_kill_fasync()` 함수를 호출해서 한다.

`_kill_fasync()` 함수는 `kill_async()` 함수와 동일한 값을 가진다.

`_kill_fasync()` 함수는 `fasync_struct` 구조체의 리스트를 돌면서, 등록된 `file object`의 소유자(`f_owner`)에게 시그널을 전달하는 일(`send_sigio()` : Send Signal I/O)을 수행한다. 이때 모든 등록된 `fasync_struct` 구조체의 `file object`에 대해서 같은 일을 반복적으로 수행한다는 점을 기억하라. 즉, 관련 파일에 대해서 관심이 있는 모든 프로세스들이 비동기 I/O가 끝났다는 것을 할게되는 것이다. 만약 등록된 `fasync_struct`의 magic number가 틀리다면 에러 메시지를 출력하고 바로 복귀 한다. 그렇지 않다면, `file object`로부터 비동기 I/O를 사용하는 프로세스에 정보를 가지는 `fown_struct`의 포인터를 얻게 되며, 이 포인터로부터 PID(Process ID)를 얻어서, 시그널을 해당 프로세스에게 보내게 된다.

```
struct fown_struct {
    int pid;          /* pid or -pgrp where SIGIO should be sent */
    uid_t uid, euid; /* uid/euid of process setting the owner */
    int signum;      /* posix.1b rt signal to be delivered on IO */
};
```

코드 499. fown_struct 구조체의 정의

`fown_struct`은 `file object`에 연결되는 자료구조로서, 비동기 입출력을 사용하고자 할 때 필요한 자료를 담고 있다. 사용자 프로세스의 ID(process ID)와 사용자 프로세스의 사용자 ID(uid), Effective 사용자 ID(euid) 및 IO시에 전달될 POSIX 1B의 Real-time Extension의 I/O 시그널을 담는 필드로 구성되어 있다. 따라서, 이 필드들을 `_kill_fasync()` 함수에서 이용한다.

앞의 이야기를 계속진행하면, 보내려는 `signal`은 SIGURG이고 `signum` 필드¹⁸⁵가 0을 가진다면, 시그널을 보내지지 않을 것이다. 즉, 보내고자 하는 `signal`이 SIGURG이거나, 혹은 `signum` 필드가 0을 가지지 않느다면, `send_sigio()`를 호출할 것이다. `send_signal()` 함수에는 `signum` 필드가 0을 가지면, default로 SIGIO를 보낸다. 그렇지 않다면, `signum`에 설정된 `signal`¹⁸⁶을 보낼 것이다.

7.7. RTC의 Interrupt Handler

RTC에서 발생하는 `interrupt`의 종류에 대해서는 앞에서 이미 살펴보았다. 즉, alarm, periodic, update timer `interrupt`등이 있었다. 이젠 이러한 `interrupt`를 어떻게 처리하는 가를 보도록 하자.

```
#if RTC_IRQ
static void rtc_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    /*
     *      Can be an alarm interrupt, update complete interrupt,
     *      or a periodic interrupt. We store the status in the
     *      low byte and the number of interrupts received since
     *      the last read in the remainder of rtc_irq_data.
    }
```

¹⁸⁴ 대부분의 경우, 주로 POLL_IN만을 사용해서 받아들일 데이터가 있음을 나타낸다. Network과 관련된 코드에서는 이 부분을 이용해서 URGENT데이터(out-of-band data)를 처리(SIGURG 시그널)할 때 사용할 수 있다.

¹⁸⁵ 이 필드는 `fcntl()`과 같은 시스템 콜을 통해서 사용자 프로세스에서 SIGIO를 포함해서, 비동기 I/O가 끝났을 때 받고자 하는 `signal`을 알려주기 위해서 설정할 수 있는 필드이다. `fcntl()`의 F_GETSIG나 F_SETSIG를 통해서 설정할 수 있을 것이다.

¹⁸⁶ 혹은 어떤 종류의 out-of-band 데이터를 나타내는 값이 될 수 있을 것이다.

```

*/
spin_lock (&rtc_lock);
rtc_irq_data += 0x100;
rtc_irq_data &= ~0xff;
rtc_irq_data |= (CMOS_READ RTC_INTR_FLAGS) & 0xF0;
if (rtc_status & RTC_TIMER_ON)
    mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);
spin_unlock (&rtc_lock);
/* Now do the rest of the actions */
wake_up_interruptible(&rtc_wait);
kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);
}
#endif

```

코드 500. rtc_interrupt() 함수의 정의

rtc_interrupt() 함수는 인터럽트의 핸들러로 수행된다. RTC가 사용하는 timer 및 RTC의 interrupt를 얻기 위해서 spin lock을 통해서 접근을 통제한다. 따라서, 한번에 프로세서(CPU)에서만 RTC 인터럽트 핸들러에서 일어나는 일을 처리해 줄 수 있을 것이다. 물론 여기있는 rtc_interrupt는 RTC_IRQ가 정의된 경우에만 사용가능하다.

rtc_irq_data 변수는 RTC에서 발생하는 인터럽트를 기록하기 위한 것이다. 이때 rtc_irq_data에 0x100을 더해서 인터럽트가 일어난 것이 몇번째인가를 대략적으로 판단할 수 있는 근거를 마련하도록 한다. 그리고나서, 읽을 interrupt 관련 레지스터(RTC_INTR_FLAGS : Register C)를 위한 필드로 나머지 하위 8 bit을 지운다. 이젠 RTC_INTR_FLAGS 레지스터를 읽어서 이 값을 rtc_irq_data와 OR시켜서 보관한다. 만약 RTC_TIMER_ON이 rtc_status에 설정되어 있다면, 놓친 인터럽트를 service하기 위한 것이기에, 다음번에 수행되도록 하기 위해서 mod_timer()를 사용해서 timer가 다음번에 활성화 될 시간을 변경(modify)한다. 만약 RTC에서 어떤 event가 있을 때까지 기다리고 있는 프로세스들이 있다면, interrupt가 발생했다는 것을 알려주기 위해서 wake_up_interruptible()를 호출한다. 마지막으로 RTC에 대해서 비동기(asynchronous) I/O를 원하는 프로세스들에 signal을 보내기 위해서 kill_fasync()를 호출한다. 역시 rtc_async_queue를 비동기 I/O의 signal을 전달하기 위해서 사용하고 있다. 한가지 마지막으로 짚고 넘어가고 싶은 것은, 중요한 것은 RTC의 인터럽트 핸들러는 매우 자주 수행될 가능성이 있다는 것이다. 따라서, 시스템의 timer interrupt와 RTC의 인터럽트가 동시에 SMP machine에서 각각 다른 CPU에서 서비스를 받을 수도 있다. 이때 생기는 문제로 인해서 spin lock을 사용해 RTC의 전역 데이터를 보호하고 있다¹⁸⁷.

7.8. RTC의 I/O Control

RTC를 사용하는데 있어서, 주된 연산(혹은 function)은 I/O Control을 통해서 이루어진다. 즉, 사용자 프로그램에서 RTC가 제공하는 기능을 완전히 이용하고, RTC를 제어하기 위해서는 I/O Control을 사용한다는 것이다.

그럼 먼저 RTC의 I/O Control로는 어떤 것들이 있으며, 이들 각각이 어떤 역할을 하는지를 알아보기로 하자. 다음과 같이 정리해 볼 수 있다.

#define RTC_AIE_ON	_IO('p', 0x01)	/* Alarm interrupt를 활성화(enable) 시킨다. */
#define RTC_AIE_OFF	_IO('p', 0x02)	/* Alarm interrupt를 비활성화(disable) 시킨다. */
#define RTC_UIE_ON	_IO('p', 0x03)	/* Update interrupt를 활성화 시킨다. */
#define RTC_UIE_OFF	_IO('p', 0x04)	/* Update interrupt를 비활성화 시킨다. */
#define RTC_PIE_ON	_IO('p', 0x05)	/* Periodic interrupt를 활성화 시킨다. */
#define RTC_PIE_OFF	_IO('p', 0x06)	/* Periodic interrupt를 비활성화 시킨다. */

¹⁸⁷ RTC의 source code에서는 set_rtc_mmss()라는 함수 때문에 이러한 마찰(conflict)이 발생한다고 되어 있다.

```
#define RTC_WIE_ON      _IO('p', 0x0f)      /* Watchdog interrupt를 활성화 시킨다.*/
#define RTC_WIE_OFF     _IO('p', 0x10)      /* Watchdog interrupt를 비활성화 시킨다.*188

#define RTC_ALM_SET     _IOW('p', 0x07, struct rtc_time)    /* Set alarm time */
#define RTC_ALM_READ     _IOR('p', 0x08, struct rtc_time)    /* Read alarm time */
#define RTC_RD_TIME      _IOR('p', 0x09, struct rtc_time)    /* Read RTC time */
#define RTC_SET_TIME     _IOW('p', 0x0a, struct rtc_time)    /* Set RTC time */
#define RTC_IRQP_READ    _IOR('p', 0x0b, unsigned long)      /* Read IRQ rate */
#define RTC_IRQP_SET     _IOW('p', 0x0c, unsigned long)      /* Set IRQ rate */
#define RTC_EPOCH_READ   _IOR('p', 0x0d, unsigned long)      /* Read epoch */
#define RTC_EPOCH_SET    _IOW('p', 0x0e, unsigned long)      /* Set epoch */

#define RTC_WKALM_SET_IOW('p', 0x0f, struct rtc_wkalrm) /* Set wakeup alarm*/
#define RTC_WKALM_RD _IOR('p', 0x10, struct rtc_wkalrm) /* Get wakeup alarm*/189
```

코드 501. RTC의 I/O Control Command Code

위의 RTC의 I/O Control command code를 그럼 어떻게 사용하고 있는지를 실제 코드를 보면서 이야기하도록 하겠다.

```
static int rtc_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
    struct rtc_time wtime;
    #if RTC_IRQ
        if (rtc_has_irq == 0) {
            switch (cmd) {
                case RTC_AIE_OFF:
                case RTC_AIE_ON:
                case RTC_PIE_OFF:
                case RTC_PIE_ON:
                case RTC_UIE_OFF:
                case RTC_UIE_ON:
                case RTC_IRQP_READ:
                case RTC_IRQP_SET:
                    return -EINVAL;
            };
        }
    #endif
    switch (cmd) {
        #if RTC_IRQ
            case RTC_AIE_OFF:      /* Mask alarm int. enab. bit */
            {
                mask_rtc_irq_bit(RTC_AIE);
                return 0;
            }
            case RTC_AIE_ON:       /* Allow alarm interrupts. */
            {
                set_rtc_irq_bit(RTC_AIE);
                return 0;
            }
            case RTC_PIE_OFF:      /* Mask periodic int. enab. bit */
            {
        
```

¹⁸⁸ 현재 Watchdog interrupt에 대한 것이 I/O Control command code에 나오지만 실제로 구현되지 않은 것으로 생각된다.

¹⁸⁹ Wakeup alarm 역시 i386계열의 CPU에서는 아직 사용되고 있지 않다. 역시 RTC의 구현 코드에서는 찾을 수 없다.

```

mask_rtc_irq_bit(RTC_PIE);
if (rtc_status & RTC_TIMER_ON) {
    spin_lock_irq (&rtc_lock);
    rtc_status &= ~RTC_TIMER_ON;
    del_timer(&rtc_irq_timer);
    spin_unlock_irq (&rtc_lock);
}
return 0;
}

```

코드 502. rtc_ioctl() 함수의 정의

RTC_IRQ가 compile option으로 주어졌지만, rtc_has_irq가 0인 값을 가진다면, 모든 Interrupt와 관련된 command에 대해서 -EINVAL이라는 값을 줄 것이다. 즉, command를 실행할 방법이 없다는 것이다. 앞에서 보듯이 모든 command들은 switch()문 이후가 I/O Control에서 각각의 command가 주어졌을 때 그에 맞는 처리를 하고 있는 부분이다. 역시 RTC_IRQ가 compile option으로 주어져 있는 경우에 대해서만 interrupt와 관련된 command를 처리하도록 한다. Register B에 대한 bit를 설정하고 지우기 위해서는 set_rtc_irq_bit()과 mask_rtc_irq_bit이라는 함수를 사용한다. Periodic Interrupt를 OFF(cmd에 RTC_PIE_OFF)시키기 위해서는 먼저 해당 bit를 clear하고(mask_rtc_irq_bit()), rtc_status에 RTC_TIMER_ON이 설정되어 있는지를 본다. 즉, 높이게되는 periodic interrupt를 대신할 목적인 timer이기 때문에 periodic interrupt가 OFF되는 상황에서는 제거되어야 할 것이다(del_timer()). rtc_status에서는 RTC_TIMER_ON bit이 clear된다.

```

#ifndef RTC_IRQ
static void mask_rtc_irq_bit(unsigned char bit)
{
    unsigned char val;

    spin_lock_irq(&rtc_lock);
    val = CMOS_READ(RTC_CONTROL);
    val &= ~bit;
    CMOS_WRITE(val, RTC_CONTROL);
    CMOS_READ(RTC_INTR_FLAGS);
    rtc_irq_data = 0;
    spin_unlock_irq(&rtc_lock);
}

static void set_rtc_irq_bit(unsigned char bit)
{
    unsigned char val;

    spin_lock_irq(&rtc_lock);
    val = CMOS_READ(RTC_CONTROL);
    val |= bit;
    CMOS_WRITE(val, RTC_CONTROL);
    CMOS_READ(RTC_INTR_FLAGS);
    rtc_irq_data = 0;
    spin_unlock_irq(&rtc_lock);
}
#endif

```

코드 503. mask_rtc_irq_bit() 함수와 set_rtc_irq_bit() 함수의 정의

mask_rtc_irq_bit() 함수와 set_rtc_irq_bit() 함수는 RTC_CONTROL(Register B)를 접근해서 특정 bit를 clear 혹은 set하는 역할을 수행한다. 이렇게 하드웨어의 레지스터를 접근해서 특정 bit를 설정하는 방법은 코드에서 보듯이, read-modify-write하는 순으로 반드시 진행되어야 할 것이다. 마지막에 있는 인터럽트 flag 레지스터를 읽는 것은, 인터럽트 flag 레지스터를 지우는 일을 하는 것이다. 즉, 새로이 설정한

interrupt control bit의 내용으로 인터럽트가 발생하도록 설정하고, 초기화 해준 것이다. `rtc_irq_data`는 초기화하기 위해서 0을 써준다. 이 모든 instruction들은 spin lock을 띄여서 한번에 한 프로세스만 사용할 수 있도록 하고 있다. `mask_rtc_irq_bit()` 함수는 특정 bit를 clear하는 일을 하며, `set_rtc_irq_bit()` 함수는 특정 bit를 설정하는 역할을 수행한다.

```
case RTC_PIE_ON:          /* Allow periodic ints */  
{  
    /*  
     * We don't really want Joe User enabling more  
     * than 64Hz of interrupts on a multi-user machine.  
     */  
    if ((rtc_freq > 64) && (!capable(CAP_SYS_RESOURCE)))  
        return -EACCES;  
    if (!(rtc_status & RTC_TIMER_ON)) {  
        spin_lock_irq(&rtc_lock);  
        rtc_irq_timer.expires = jiffies + HZ/rtc_freq + 2*HZ/100;  
        add_timer(&rtc_irq_timer);  
        rtc_status |= RTC_TIMER_ON;  
        spin_unlock_irq(&rtc_lock);  
    }  
    set_rtc_irq_bit(RTC_PIE);  
    return 0;  
}
```

코드 504. `rtc_ioctl()` 함수의 정의(계속)

`RTC_PIE_ON`도 역시 interrupt와 관련된 부분이다. `rtc_freq`가 64보다 클때는, `CAP_SYS_RESOURCE`(System Resource를 관리할 수 있는 권한(capability)이 있음을 나타낸다.)이 있는지를 확인한다. 만약 권한이 없다면 시스템의 부하를 증가시키는 요인이 될 수 있기 때문에 `-EACCESS`(Error Access: 접근 오류)를 돌려준다. 만약 `rtc_status`에 `RTC_TIMER_ON`이 설정되지 않았다면, 즉, timer가 설정이 되지 않았다는 뜻이되기에 새로운 RTC를 위한 timer를 생성해서 추가해준다(`add_tiemr()`). `rtc_status`에는 이제 `RTC_TIMER_ON`이 설정되어 있을 것이다. 마지막으로 periodic interrupt를 ON 시키기 위해서 `set_rtc_irq_bit()`를 호출한다. `RTC_PIE`가 해당 bit가 될 것이다.

```
case RTC_UIE_OFF:          /* Mask ints from RTC updates. */  
{  
    mask_rtc_irq_bit(RTC_UIE);  
    return 0;  
}  
case RTC_UIE_ON:          /* Allow ints for RTC updates. */  
{  
    set_rtc_irq_bit(RTC_UIE);  
    return 0;  
}  
#endif  
case RTC_ALM_READ:         /* Read the present alarm time */  
{  
    /*  
     * This returns a struct rtc_time. Reading >= 0xc0  
     * means "don't care" or "match all". Only the tm_hour,  
     * tm_min, and tm_sec values are filled in.  
     */  
    get_rtc_alm_time(&wtime);  
    break;  
}
```

코드 505. `rtc_ioctl()` 함수의 정의(계속)

RTC_UIE_OFF와 RTC_UIE_ON은 update interrupt를 OFF/ON시키는 일을 한다. RTC_ALM_READ는 RTC에 설정된 alarm 시간을 읽어오는 일을 한다.

```
static void get_rtc_alm_time(struct rtc_time *alm_tm)
{
    unsigned char ctrl;

    /*
     * Only the values that we read from the RTC are set. That
     * means only tm_hour, tm_min, and tm_sec.
     */
    spin_lock_irq(&rtc_lock);
    alm_tm->tm_sec = CMOS_READ(RTC_SECONDS_ALARM);
    alm_tm->tm_min = CMOS_READ(RTC_MINUTES_ALARM);
    alm_tm->tm_hour = CMOS_READ(RTC_HOURS_ALARM);
    ctrl = CMOS_READ(RTC_CONTROL);
    spin_unlock_irq(&rtc_lock);
    if (!(ctrl & RTC_DM_BINARY) || RTC_ALWAYS_BCD)
    {
        BCD_TO_BIN(alm_tm->tm_sec);
        BCD_TO_BIN(alm_tm->tm_min);
        BCD_TO_BIN(alm_tm->tm_hour);
    }
}
```

코드 506. get_rtc_alm_time() 함수의 정의

get_rtc_alm_time()이라는 함수가 RTC에 저장된 alarm 시간을 읽어오는 일을 처리할 것이다. 이 함수는 seconds, minutes, hours 각각에 대해서 alarm 시간을 저장하는 부분의 RTC의 byte들을 읽어서 이를 alm_tm이라는 변수가 가르키는 rtc_time 구조체에 저장한다. 이때 RTC에 저장되어 있는 데이터의 format(CMOS_READ(RTC_CONTROL))에 따라서, BCD 코드를 binary 코드로 변환하는 일을 하기 위해서 BCD_TO_BIN() 사용한다.

```
struct rtc_time {
    int tm_sec;           /* Second */
    int tm_min;           /* Minute */
    int tm_hour;          /* Hour */
    int tm_mday;          /* Date of Month */
    int tm_mon;           /* Month */
    int tm_year;          /* Year */
    int tm_wday;          /* Day of Week */
    int tm_yday;          /* 사용하지 않는다. */
    int tm_isdst;         /* 사용하지 않는다 */
};
```

코드 507. rtc_time 구조체의 정의

rtc_time 구조체의 각 필드를 보기로 보면, 이 구조체를 통해서 사용자 프로그램과 RTC 디바이스가 주고 받는 데이터가 어떤 것인가를 대략적으로 알 수 있다. 즉, 사용자 프로그램은 RTC에 저장된 10 byte 영역의 time, calendar, alarm 정보를 읽어가기 위해서(혹은 다시 지정하기 위해서) rtc_time 구조체를 사용한다.

```
case RTC_ALM_SET:      /* Store a time into the alarm */
{
    unsigned char hrs, min, sec;
    struct rtc_time alm_tm;
```

```

        if (copy_from_user(&alm_tm, (struct rtc_time*)arg, sizeof(struct rtc_time)))
            return -EFAULT;
        hrs = alm_tm.tm_hour;
        min = alm_tm.tm_min;
        sec = alm_tm.tm_sec;
        if (hrs >= 24)
            hrs = 0xff;
        if (min >= 60)
            min = 0xff;
        if (sec >= 60)
            sec = 0xff;
        spin_lock_irq(&rtc_lock);
        if (!(CMOS_READ RTC_CONTROL) & RTC_DM_BINARY) || RTC_ALWAYS_BCD)
        {
            BIN_TO_BCD(sec);
            BIN_TO_BCD(min);
            BIN_TO_BCD(hrs);
        }
        CMOS_WRITE(hrs, RTC_HOURS_ALARM);
        CMOS_WRITE(min, RTC_MINUTES_ALARM);
        CMOS_WRITE(sec, RTC_SECONDS_ALARM);
        spin_unlock_irq(&rtc_lock);
        return 0;
    }
}

```

코드 508. rtc_ioctl() 함수의 정의(계속)

RTC_ALM_SET은 RTC에 alarm 시간을 지정하기 위해서 사용한다. 사용자 프로그램은 이를 위해서 rtc_time 구조체를 만들어서, 이를 전달해 줄 것이라고 예상한다. 따라서, copy_from_user()를 이용해서 이 정보를 alm_tm에 저장한다. 이젠 이 정보를 보고, RTC의 Register B에 write하는 일이 될 것이다. 이때, 사용자가 전달한 값이 각 필드가 가져야하는 값보다 큰 경우에는 0xFF를 사용해서 “don’t care”¹⁹⁰라는 것을 나타내도록 한다. 저장할 때도 읽을 때와 마찬가지로 BCD로 저장해야 할 경우에는 BIN_TO_BCD()를 사용해서 BCD format으로 변환해서 저장하게 되며, 그렇지 않다면 바로 binary mode format으로 RTC의 calendar, time, alarm의 영역에 저장하게 된다.

```

case RTC_RD_TIME: /* Read the time/date from RTC */
{
    get_rtc_time(&wtime);
    break;
}

```

코드 509. rtc_ioctl() 함수의 정의(계속)

RTC_RD_TIME은 RTC에 저장된 시간과 날짜를 읽어오기 위해서 사용한다. get_rtc_time()이라는 함수를 사용하며, 넘겨주는 것은 rtc_time 구조체를 가르키는 포인터 값이다.

```

static void get_rtc_time(struct rtc_time *rtc_tm)
{
    unsigned long uip_watchdog = jiffies;
    unsigned char ctrl;
#define CONFIG_DECSTATION
    unsigned int real_year;
#endif
}

```

¹⁹⁰ Source code상에서는 “match all”이라는 것으로 표현되어 있다. 실제로는 0x0C에서 0xFF까지의 어떤 값도 전부 “don’t care”라는 것을 명시하는 값으로 사용할 수 있다. 만약 이러한 값이 설정되어 있다면, 매시간, 매분, 매초마다 alarm interrupt가 발생하게 된다.

```

if (rtc_is_updating() != 0)
    while (jiffies - uip_watchdog < 2*HZ/100) {
        barrier();
        cpu_relax();
    }
spin_lock_irq(&rtc_lock);
rtc_tm->tm_sec = CMOS_READ(RTC_SECONDS);
rtc_tm->tm_min = CMOS_READ(RTC_MINUTES);
rtc_tm->tm_hour = CMOS_READ(RTC_HOURS);
rtc_tm->tm_mday = CMOS_READ(RTC_DAY_OF_MONTH);
rtc_tm->tm_mon = CMOS_READ(RTC_MONTH);
rtc_tm->tm_year = CMOS_READ(RTC_YEAR);

```

코드 510. get_rtc_time() 함수의 정의

get_rtc_time() 함수는 먼저 RTC가 현재 내부에 저장된 데이터가 update중인지를 확인하면서, 대기하도록 한다. 대략 이렇게 대기하는 시간은 2×10 ms($jiffies$ 값이 대략 10ms를 이야기 한다.)정도가 된다. 즉, RTC가 update중이라면, 이정도의 시간을 기다린 후에 완전히 update가 끝난 시점에 RTC에 저장된 시간을 읽는다. 읽어들이는 값은 RTC_SECONDS, RTC_MINUTES, RTC_HOURS, RTC_DAY_OF_MONTH, RTC_MONTH, RTC_YEAR이다. 이때, RTC_DAY_OF_WEEK는 RTC가 초기에 0이 아닌 값으로 설정할 경우에만 update가 되기 때문에 이를 생략했다.

```

#ifndef CONFIG_DECSTATION
    real_year = CMOS_READ(RTC_DEC_YEAR);
#endif
ctrl = CMOS_READ(RTC_CONTROL);
spin_unlock_irq(&rtc_lock);
if (!(ctrl & RTC_DM_BINARY) || RTC_ALWAYS_BCD)
{
    BCD_TO_BIN(rtc_tm->tm_sec);
    BCD_TO_BIN(rtc_tm->tm_min);
    BCD_TO_BIN(rtc_tm->tm_hour);
    BCD_TO_BIN(rtc_tm->tm_mday);
    BCD_TO_BIN(rtc_tm->tm_mon);
    BCD_TO_BIN(rtc_tm->tm_year);
}
#endif CONFIG_DECSTATION
    rtc_tm->tm_year += real_year - 72;
#endif
    if ((rtc_tm->tm_year += (epoch - 1900)) <= 69)
        rtc_tm->tm_year += 100;
    rtc_tm->tm_mon--;
}

```

코드 511. get_rtc_time() 함수의 정의

CONFIG_DECSTATION이라는 것이 compile 옵션으로 주어진 경우에는 RTC_DEC_YEAR를 RTC로부터 읽어서 real_year에 저장한다. 이렇게 읽은 연도(year)값은 DEC에서 사용하는 year값으로 변환하기 위해서, 나중에 72이라는 값을 빼서 RTC에 저장된 year값과 더하도록 한다. 이전 이렇게 읽은 값을 저장된 format에 따라, binary format으로 변환해야 할 경우에는 BCD_TO_BIN()을 호출해서 바꾼다. 변환된 데이터들을 이전 현재 시간으로 다시 조금 조정하는 과정을 통과해야 한다. 이때 만약 읽은 연도(year)의 데이터에, epoch(default 값으로 1900을 가지지만, RTC 내부에서 변경된다.)에서 1900을 뺀 값을 더한다. 이렇게 생성된 값이 69보다 작다면, 다시 100을 연도(year)에 더하도록 한다. 이것은 각각의 시스템에서 사용하는 연도 표기가 달라지기 때문에 생기는 문제를 보정하기 위해서 필요한 것이다. 마지막으로 읽어온 달을 나타내는 tm_mon을 1 감소시켜준다.

```

case RTC_SET_TIME:      /* Set the RTC */
{
    struct rtc_time rtc_tm;
    unsigned char mon, day, hrs, min, sec, leap_yr;
    unsigned char save_control, save_freq_select;
    unsigned int yrs;
#endif CONFIG_DECSTATION
    unsigned int real_yrs;
#endif
    if (!capable(CAP_SYS_TIME))
        return -EACCES;
    if (copy_from_user(&rtc_tm, (struct rtc_time*)arg,
                      sizeof(struct rtc_time)))
        return -EFAULT;
    yrs = rtc_tm.tm_year + 1900;
    mon = rtc_tm.tm_mon + 1; /* tm_mon starts at zero */
    day = rtc_tm.tm_mday;
    hrs = rtc_tm.tm_hour;
    min = rtc_tm.tm_min;
    sec = rtc_tm.tm_sec;
    if (yrs < 1970)
        return -EINVAL;
    leap_yr = ((!(yrs % 4) && (yrs % 100)) || !(yrs % 400));
    if ((mon > 12) || (day == 0))
        return -EINVAL;
    if (day > (days_in_mo[mon] + ((mon == 2) && leap_yr)))
        return -EINVAL;
    if ((hrs >= 24) || (min >= 60) || (sec >= 60))
        return -EINVAL;
    if ((yrs -= epoch) > 255) /* They are unsigned */
        return -EINVAL;
}

```

코드 512. rtc_ioctl() 함수의 정의(계속)

RTC_SET_TIME은 RTC에 저장된 시간을 변경하기 위해서 사용한다. 먼저 사용자 프로그램이 CAP_SYS_TIME이라는 capability를 가지고 있는지를 확인한다. 이러한 권한(capability)을 가지지 못한다면, -EACCESS(Error Access: 접근 오류)를 돌려준다. 이전 사용자가 설정하고자 하는 시간을 rtc_time이라는 변수로 읽어오기 위해서 copy_from_user()를 사용한다. 오류가 발생한다면, -EFAULT(Error Fault)를 돌려준다. 사용자 데이터의 year는 1900년을 기준으로 해서 값을 가지고 있다고 가정하기에, 연도에 1900이라는 값을 더한다. mon 역시 1을 더한 값으로 설정한다. 나머지는 사용자가 전달한 값을 그대로 사용하도록 한다. 하지만, 만약 연도(year)값이 1970이하라는 값을 가진다면, 오류로 보고 -EINVAL(Error Invalid)를 돌려준다. 이것은 일반적으로 컴퓨터에서 설정하는 연도 값은 1970년 이후를 가정하기 때문이다. 또한 윤년을 계산에 넣기 위해서 생성된 연도가 4로 나누어 떨어지고, 또한 100으로도 나누어 떨어지지 않는 경우이거나, 혹은 400으로 나누어 떨어지는 경우에는 윤년이라고 본다. 이렇게 구한 값으로, 2월의 특정한 날이 28일이 되는지 29일이 되는지를 결정하는데 사용한다. 만약 이렇게 해서 구한 날이 사용자가 명시한 날(day)보다 작다면, 당연히 -EINVAL(Error Invalid)를 돌려줄 것이다. 이전 시간과 분, 초가 적절한 값을 가지는지를 본다. 역시 오류가 있었다면, -EINVAL(Error Invalid)를 돌려줄 것이다. 연도는 다시 epoch라는 값을 뺀 값으로 설정하고, 이렇게 설정된 값이 8 bit(= 255)이 가질 수 있는 값보다 크다면, -EINVAL(Error Invalid)를 돌려줄 것이다.

```

spin_lock_irq(&rtc_lock);
#endif CONFIG_DECSTATION
real_yrs = yrs;
yrs = 72;
/*
 * We want to keep the year set to 73 until March

```

```

        * for non-leap years, so that Feb, 29th is handled
        * correctly.
        */
if (!leap_yr && mon < 3) {
    real_yrs--;
    yrs = 73;
}
#endif
/* These limits and adjustments are independant of
 * whether the chip is in binary mode or not.
 */
if (yrs > 169) {
    spin_unlock_irq(&rtc_lock);
    return -EINVAL;
}
if (yrs >= 100)
    yrs -= 100;
if (!(CMOS_READ RTC_CONTROL) & RTC_DM_BINARY)
    || RTC_ALWAYS_BCD) {
    BIN_TO_BCD(sec);
    BIN_TO_BCD(min);
    BIN_TO_BCD(hrs);
    BIN_TO_BCD(day);
    BIN_TO_BCD(mon);
    BIN_TO_BCD(yrs);
}

```

코드 513. rtc_ioctl() 함수의 정의(계속)

RTC의 하드웨어에 직접적으로 연산을 해야하므로, spin lock을 사용해서 critical section에 속하는 코드를 보호한다. 만약 CONFIG_DECSTATION이라는 커널 옵션이 주어진 경우에는 real_yrs에는 앞에서 계산된 연도를 넣고, yrs에는 72로 초기화 시킨다. 그리고, 윤년이 아니고, 달이 1월이나 2월 달이라면, real_yrs에서 일을 감소시키고, yrs에는 73을 넣는다. 이것은 DEC의 기계상에서 연도를 표기하는데 있어서, 달리 사용하기 때문에 부분이다.

이전 앞에서 주어진 yrs의 값이 169보다 큰지를 보도록 한다. 이것은 2139년까지의 연도보다 더 큰 값을 RTC가 가져선 안되기 때문이다. 이때는 spin lock을 해제하고, -EINVAL(Error Invalid)를 돌려주도록 한다. 만약 yrs가 100보다 크거나 같은 값을 가진다면, -100을 해서 100이하의 값만을 가지고도록 만들어준다. 나머지는 이렇게 계산된 값을 RTC에 저장하는 format에 맞게 변환하는 과정이다.

```

save_control = CMOS_READ(RTC_CONTROL);
CMOS_WRITE((save_control|RTC_SET), RTC_CONTROL);
save_freq_select = CMOS_READ(RTC_FREQ_SELECT);
CMOS_WRITE((save_freq_select|RTC_DIV_RESET2), RTC_FREQ_SELECT);
#endif CONFIG_DECSTATION
    CMOS_WRITE(real_yrs, RTC_DEC_YEAR);
#endif
    CMOS_WRITE(yrs, RTC_YEAR);
    CMOS_WRITE(mon, RTC_MONTH);
    CMOS_WRITE(day, RTC_DAY_OF_MONTH);
    CMOS_WRITE(hrs, RTC_HOURS);
    CMOS_WRITE(min, RTC_MINUTES);
    CMOS_WRITE(sec, RTC_SECONDS);
    CMOS_WRITE(save_control, RTC_CONTROL);
    CMOS_WRITE(save_freq_select, RTC_FREQ_SELECT);
    spin_unlock_irq(&rtc_lock);
    return 0;
}

```

코드 514. rtc_ioctl() 함수의 정의(계속)

원래에 정해져 있는 RTC Control 레지스터(Register B)를 읽어서 save_control에 저장하도록 한다. 그리고나서, 이 값과 RTC_SET을 OR시켜서, clock에 의해서 update가 일어나지 않도록, 다시 RTC Control 레지스터에 쓴다. 이젠 RTC의 frequency 레지스터(Register A)를 읽어서 이것을 다시 save_freq_select에 저장하도록 한다. 이렇게 저장된 frequency값과 RTC_DIV_RESET2를 OR 시켜서, 다시 frequency 레지스터에 쓴다. 이것으로 RTC에 대한 clock input을 나누어주는(divider)를 reset시켜주는 일을 수행한다. CONFIG_DECSTATION으로 컴파일 옵션이 설정된 경우에는 RTC_DEC_YEAR의 byte부분에 앞에서 구한 real_yrs를 쓰도록 할 것이다. 나머지는 각각의 값을 해당 위치에 쓰는 일을 한다. 다시 RTC의 update가 일어나게 하기위해서 앞에서 저장해 두었던 save_control과 save_freq_select값을 Register B와 Register A에 각각 다시 쓰도록 한다. 이것으로 RTC에 대한 시간 설정이 마무리된다. 복귀전에 앞에서 설정했던 spin lock을 해제하도록 한다.

```
#if RTC_IRQ
    case RTC_IRQP_READ: /* Read the periodic IRQ rate. */
    {
        return put_user(rtc_freq, (unsigned long *)arg);
    }
    case RTC_IRQP_SET: /* Set periodic IRQ rate. */
    {
        int tmp = 0;
        unsigned char val;
        /*
         * The max we can do is 8192Hz.
         */
        if ((arg < 2) || (arg > 8192))
            return -EINVAL;
        /*
         * We don't really want Joe User generating more
         * than 64Hz of interrupts on a multi-user machine.
         */
        if ((arg > 64) && (!capable(CAP_SYS_RESOURCE)))
            return -EACCES;
        while (arg > (1<<tmp))
            tmp++;
        /*
         * Check that the input was really a power of 2.
         */
        if (arg != (1<<tmp))
            return -EINVAL;
        spin_lock_irq(&rtc_lock);
        rtc_freq = arg;
        val = CMOS_READ RTC_FREQ_SELECT) & 0xf0;
        val |= (16 - tmp);
        CMOS_WRITE(val, RTC_FREQ_SELECT);
        spin_unlock_irq(&rtc_lock);
        return 0;
    }
#endif
```

코드 515. rtc_ioctl() 함수의 정의(계속)

RTC_IRQP_READ는 periodic interrupt가 발생하는 frequency를 읽어가기 위해서 호출한다. 이 값은 rtc_freq에 있으므로, 이것을 put_user()를 통해서 사용자 프로그램의 버퍼로 복사하도록 한다.

RTC_IRQP_SET은 앞에서 읽어가는 periodic interrupt의 발생빈도(frequency)를 설정하기 위한 것이다. 먼저 이 값이 현재 RTC 드라이버가 지원하는 최대 frequency보다 큰 값(>8192Hz)을 가지거나, 혹은 최소(=

2Hz)보다 작은 값을 가지는 가를 조사한다. 이때는 -EINVAL(Error Invalid)를 돌려주도록 한다. 또한 64 Hz보다 큰 경우에 대한 RTC의 frequency를 설정을 요구할 때는 CAP_SYS_RESOURCE의 권한이 있는가를 보도록 한다(capable()). 권한이 없다면 -EACCESS(Error Access)를 돌려준다. 즉, 시스템의 load가 많이 걸리는 것은 특정 사용자에게만 권한이 있도록 만들어준다. 이전 전달받은 periodic interrupt를 설정하기 위해서 이 값을 변환 하도록 한다. 넘겨받은 값이 2의 정확한 배수가 되는가도 이 과정에서 검사된다. 이 조건을 만족하지 않는 값을 사용자가 요구했다면, -EINVAL(Error Invalid)를 돌려준다. 이전 Register A(RTC_FREQ_SELECT 레지스터)에 설정된 값을 읽어들인다. 이때 RS0, RS1, RS2, RS3의 상위 4 bit만을 읽어서 val에 저장하고, 16에서 전달받은 값의 2의 승수(order)를 뺀 후, OR시켜서 Register A에 도로 저장한다. 예를 들어 8192인 경우에는 이 과정을 통해서 3(0x3)이라는 값을 가지게 되므로, [표 52]에서 보듯이 RS0와 RS1만이 설정될 것이다. 2인 경우에는 이러한 과정을 통해서 15(0xF)라는 값을 가지게 될 것이다.

```

case RTC_EPOCH_READ: /* Read the epoch. */
{
    return put_user(epoch, (unsigned long *)arg);
}
case RTC_EPOCH_SET: /* Set the epoch. */
{
    /*
     * There were no RTC clocks before 1900.
     */
    if (arg < 1900)
        return -EINVAL;
    if (!capable(CAP_SYS_TIME))
        return -EACCES;
    epoch = arg;
    return 0;
}
default:
    return -EINVAL;
}
return copy_to_user((void *)arg, &wtime, sizeof wtime) ? -EFAULT : 0;
}

```

코드 516. rtc_ioctl() 함수의 정의(계속)

RTC_EPOCH_READ는 현재 기준이 되는 시대(epoch)를 읽어내가는 함수이다. 현재 설정된 시대는 epoch라는 변수에 있으므로, 여기에 저장된 값을 put_user()를 이용해서 사용자 프로그램으로 복사한다. RTC_EPOCH_SET은 반대로 epoch이라는 변수를 설정하는 역할을 한다. 전달받은 값이 1900보다 작다면, RTC가 지원하지 않는 연대를 나타내므로 -EINVAL(Error Invalid)를 돌려준다. 그렇지 않다면, 사용자 프로그램이 CAP_SYS_TIME의 권한을 가졌는지 확인한다. 그렇지 않다면, -EACCESS(Error Access)를 돌려주도록 한다. 결과적으로 epoch 변수는 사용자가 준 값으로 새로이 update된다. 이외의 command code에 대해서는 -EINVAL(Error Invalid)를 돌려주도록 하며, 각각의 command code를 처리하면서 return하지 않은 경우에 대해서는 copy_to_user()를 이용해서 사용자 영역의 buffer에 rtc_time 구조체를 복사해주고, 오류시에는 -EFAULT(Error Fault)를, 그렇지 않다면 0을 복귀 코드로 돌려준다.

7.9. RTC의 Proc File System 인터페이스

```

static int rtc_read_proc(char *page, char **start, off_t off, int count, int *eof, void *data)
{
    int len = rtc_proc_output(page);
    if (len <= off+count) *eof = 1;
    *start = page + off;
    len -= off;
    if (len>count) len = count;
    if (len<0) len = 0;
}

```

```
    return len;
}
```

코드 517. rtc_read_proc() 함수의 정의

rtc_read_proc() 함수는 PROC 파일 시스템에 대한 인터페이스를 담당한다. Read만 정의되어 있기 때문에 사용자 프로그램이 /proc 디렉토리에서 자료를 읽어가는 것만이 가능할 것이다. 실제 PROC 파일 시스템에 대한 인터페이스의 데이터 취합은 rtc_proc_output() 함수를 호출해서 처리하고, 나머지는 파일과 관련된 I/O를 처리하는 것처럼 emulation을 하는 과정이다. 즉, RTC를 사용하는 프로세스가 현재 파일에 대한 offset(off)을 주고, 이를 프로세스가 읽어가는 데이터의 크기(count)와 더해서 이 값이 rtc_proc_output() 함수의 복귀값보다 크다면 EOF(End of File)을 ON 시켜준다. 다음번 읽을 위치(*start)는 현재 위치(page에 offset(off))을 더한 값으로 갱신하고, 길이(len)는 offset(off)를 뺀값으로 갱신한다. 만약 이 값이 읽어가는 값의 길이(count)보다 크다면, 넘겨받은 읽어야 할 길이로 새로 설정한다. 만약 읽은 데이터의 크기(len)가 0보다 작다면, 0을 사용하도록 한다. 복귀 값은 이러한 과정을 통해서 생성된 읽은 데이터의 크기를 나타내는 len이 될 것이다.

```
static int rtc_proc_output (char *buf)
{
#define YN(bit) ((ctrl & bit) ? "yes" : "no")
#define NY(bit) ((ctrl & bit) ? "no" : "yes")
    char *p;
    struct rtc_time tm;
    unsigned char batt, ctrl;
    unsigned long freq;

    spin_lock_irq(&rtc_lock);
    batt = CMOS_READ(RTC_VALID) & RTC_VRT;
    ctrl = CMOS_READ(RTC_CONTROL);
    freq = rtc_freq;
    spin_unlock_irq(&rtc_lock);
    p = buf;
    get_rtc_time(&tm);
    /*
     * There is no way to tell if the user has the RTC set for local
     * time or for Universal Standard Time (GMT). Probably local though.
     */
    p += sprintf(p,
                 "rtc_time\t: %02d:%02d:%02d\n"
                 "rtc_date\t: %04d-%02d-%02d\n"
                 "rtc_epoch\t: %04lu\n",
                 tm.tm_hour, tm.tm_min, tm.tm_sec,
                 tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday, epoch);
    get_rtc_alm_time(&tm);
}
```

코드 518. rtc_proc_output() 함수의 정의

rtc_proc_output() 함수는 직접적으로 PROC 파일 시스템에 대한 read를 대응하기 위해서 사용한다. 즉, 사용자 프로그램의 데이터 버퍼에 RTC가 가지고 있는 정보를 복사하는 수행한다. 매크로로 정의된 YN()과 NY()는 RTC의 Register B에 특정 bit이 설정되어 있는가를 확인하는데 사용한다. 먼저 RTC_VALID(Register D)를 읽어서, RTC가 유효한(valid) 값을 저장하고 있는지를 확인한다. 이렇게 읽은 값을 batt에 저장하고, Register B를 읽어서 ctrl에 저장한다. p변수는 사용자 프로세스의 데이터 버퍼 영역을 가르키게 되며, get_rtc_time() 함수는 RTC에 저장된 현재 시간에 대한 정보를 읽어온다. 이제 이렇게 읽은 값을 format에 맞게 사용자 프로세스의 데이터 buffer에 copy하는 부분이다. 시간에 대한 정보중 alarm 시간에 대한 것은 get_rtc_alm_time() 함수를 통해서 읽어오도록 한다.

```
p += sprintf(p, "alarm\t:t: ");
```

```

if (tm.tm_hour <= 24)
    p += sprintf(p, "%02d:", tm.tm_hour);
else
    p += sprintf(p, "***:");
if (tm.tm_min <= 59)
    p += sprintf(p, "%02d:", tm.tm_min);
else
    p += sprintf(p, "***:");
if (tm.tm_sec <= 59)
    p += sprintf(p, "%02d\n", tm.tm_sec);
else
    p += sprintf(p, "***\n");
p += sprintf(p,
    "DST_enable\t: %s\n"
    "BCD\t\t: %s\n"
    "24hr\t\t: %s\n"
    "square_wave\t: %s\n"
    "alarm_IRQ\t: %s\n"
    "update_IRQ\t: %s\n"
    "periodic_IRQ\t: %s\n"
    "periodic_freq\t: %ld\n"
    "batt_status\t: %s",
    YN(RTC_DST_EN),
    NY(RTC_DM_BINARY),
    YN(RTC_24H),
    YN(RTC_SQWE),
    YN(RTC_AIE),
    YN(RTC_UIE),
    YN(RTC_PIE),
    freq,
    batt ? "okay" : "dead"));
return p - buf;
#endif YN
#ifndef NY
}

```

코드 519. rtc_proc_output() 함수의 정의(계속)

이전 alarm 시간을 사용자 프로그램의 버퍼에 쓸 차례이다. 시간값들이 올바른 형태의 값으로 저장되어 있는가를 확인하는 과정과 이를 버퍼에 copy하는 과정이다. 나머지는 Register B(RTC Control Register)에 특정 bit이 설정되어 있는가를 확인하고, 이를 역시 사용자 프로그램의 데이터 버퍼에 쓴다. 위와 같은 과정을 통해서 생성된 데이터의 크기가 복귀 값으로 주어질 것이다.

이상에서 우린 RTC가 어떤 구성을 가지며, Linux에서는 어떻게 RTC에 대한 디바이스 드라이버를 구현하는가를 보았다. 중간에 I/O를 다루는 과정에서는 비동기적인 I/O가 어떻게 구현되는가를 잠시 보았으며, 또한 PROC 파일 시스템을 통해서 RTC의 어떤 정보들을 알 수 있는가를 알게되었다. 이제부터는 본격적으로 RTC를 이용하는 프로그램을 작성하는 방법을 보도록 하겠다.

7.10. RTC를 이용하는 예제 프로그램

여기서 보는 RTC의 예제 프로그램은 ~/Documentation/rtc.txt에 있는 것이다. 간단한 예제로 RTC를 어떻게 사용하는지를 충분히 보여준다고 생각해서 잠시 보도록 하겠다.

```

/*
 *      Real Time Clock Driver Test/Example Program
 *
 *      Compile with:

```

```

/*
 *      gcc -s -Wall -Wstrict-prototypes rtctest.c -o rtctest
 *
 *      Copyright (C) 1996, Paul Gortmaker.
 *
 *      Released under the GNU General Public License, version 2,
 *      included herein by reference.
 *
 */

#include <stdio.h>
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

int main(void) {

    int i, fd, retval, irqcount = 0;
    unsigned long tmp, data;
    struct rtc_time rtc_tm;

    fd = open ("/dev/rtc", O_RDONLY);

    if (fd == -1) {
        perror("/dev/rtc");
        exit(errno);
    }

    fprintf(stderr, "\n\t\tRTC Driver Test Example.\n\n");

    /* Turn on update interrupts (one per second) */
    retval = ioctl(fd, RTC_UIE_ON, 0);
    if (retval == -1) {
        perror("ioctl");
        exit(errno);
    }
}

```

코드 520. RTC를 이용하는 예제 프로그램

RTC를 사용하기 위해서는 모든 디바이스에 접근하는 프로그램과 마찬가지로 디바이스를 여는(open) 과정이 필요하다. RTC에 대한 디바이스 노드(node)는 “/dev/rtc”이며, 파일에 대한 write 연산을 RTC가 정의하고 있지 않으므로, O_RDONLY(Read Only 모드로 파일을 open하라.)모드로 파일을 연다. 물론 read/write가 가능하도록 할 수 있으나, write시에 file operation이 정해져 있지 않으므로 오류가 발생할 것이다. “/dev/rtc”를 열지 못한다면, 에러값으로 fd에 -1이 설정될 것이기에 이때는 perror()를 호출해서 에러가 발생했다는 것을 알려준다. perror()를 호출하는 것으로 해당 error 코드가 설정될 것이므로, 이 errno를 exit 코드 값으로 사용하면 된다. RTC에 테스트의 목적으로 담고 있는 프로그램이기에 RTC의 핵심에 속하는 rtc_ioctl()함수를 호출하기 위해서 ioctl()를 사용한다. 당연히 프로그램에서는 sys/ioctl.h와 linux/rtc.h를 include해야 할 것이다. RTC_UIE_ON은 이미 앞에서 RTC에 대한 Update Interrupt Enable이라는 뜻으로 사용한다는 것을 보았다. 에러가 있었다면, 해당 에러를 구하고 에러 코드를 exit 코드 값으로 사용한다.

```

fprintf(stderr, "Counting 5 update (1/sec) interrupts from reading /dev/rtc:");
fflush(stderr);
for (i=1; i<6; i++) {

```

```

/* This read will block */
retval = read(fd, &data, sizeof(unsigned long));
if (retval == -1) {
    perror("read");
    exit(errno);
}
fprintf(stderr, " %d", i);
fflush(stderr);
irqcount++;
}

```

코드 521. RTC를 이용하는 예제 프로그램(계속)

이 부분은 앞에서 enable시켜둔 Update Interrupt가 발생하는지를 5번에 걸쳐서 RTC를 읽어 보는 것을 보여준다. 즉, RTC의 `rtc_read()` 함수가 호출되면서 알게되는 값은 RTC의 `interrupt`가 어떤 것이 일어났는지를 아는 것이기 때문이다. `fflush()`함수는 파일에 대한 입출력을 위해서 버퍼에 대기하고 있는 데이터들을 쓰라는 뜻을 가진다. 이때 발생하는 `interrupt`의 개수는 `irqcount`가 가진다.

```

fprintf(stderr, "\nAgain, from using select(2) on /dev/rtc:");
fflush(stderr);
for (i=1; i<6; i++) {
    struct timeval tv = {5, 0}; /* 5 second timeout on select */
    fd_set readfds;

    FD_ZERO(&readfds);
    FD_SET(fd, &readfds);
    /* The select will wait until an RTC interrupt happens. */
    retval = select(fd+1, &readfds, NULL, NULL, &tv);
    if (retval == -1) {
        perror("select");
        exit(errno);
    }
    /* This read won't block unlike the select-less case above. */
    retval = read(fd, &data, sizeof(unsigned long));
    if (retval == -1) {
        perror("read");
        exit(errno);
    }
    fprintf(stderr, " %d", i);
    fflush(stderr);
    irqcount++;
}

```

코드 522. RTC를 이용하는 예제 프로그램(계속)

앞에서의 RTC에 대한 `read`는 RTC에서 데이터가 있을 때까지 `wait queue`에서 대기하는 일을 하게 된다. 여기서는 동일한 일을 `timeout`을 주고, `select()` 시스템 콜을 사용해서 처리한다. 이때는 RTC의 `rtc_poll()` 함수가 호출될 것이다. 따라서, 디바이스 드라이버로부터 읽을 데이터가 있는지를 확인하고, 없다면 `wait queue`에서 대기할 것이며, 대기하는 동안 `signal`과 같은 것을 받게되면, 다시 깨어나 수행하게 될 것이다. 만약 대기하는 동안 `timeout`이 발생했을 경우에도 깨어나게 될 것이다. 어쨌든 `select()` 이후의 부분은 이렇게해서 깨어났을 때 데이터를 읽어서, 이 값을 표준 에러 출력(standard error)으로 쓰는 일을 수행한다. 역시 인터럽트가 발생한 회수는 `irqcount`로 나타낸다.

```

/* Turn off update interrupts */
retval = ioctl(fd, RTC_UIE_OFF, 0);
if (retval == -1) {
    perror("ioctl");
}

```

```

        exit(errno);
    }

/* Read the RTC time/date */
retval = ioctl(fd, RTC_RD_TIME, &rtc_tm);
if (retval == -1) {
    perror("ioctl");
    exit(errno);
}
fprintf(stderr, "\n\nCurrent RTC date/time is %d-%d-%d, %02d:%02d:%02d.\n",
        rtc_tm.tm_mday, rtc_tm.tm_mon + 1, rtc_tm.tm_year + 1900,
        rtc_tm.tm_hour, rtc_tm.tm_min, rtc_tm.tm_sec);

/* Set the alarm to 5 sec in the future, and check for rollover */
rtc_tm.tm_sec += 5;
if (rtc_tm.tm_sec >= 60) {
    rtc_tm.tm_sec %= 60;
    rtc_tm.tm_min++;
}
if (rtc_tm.tm_min == 60) {
    rtc_tm.tm_min = 0;
    rtc_tm.tm_hour++;
}
if (rtc_tm.tm_hour == 24)
    rtc_tm.tm_hour = 0;
retval = ioctl(fd, RTC_ALM_SET, &rtc_tm);
if (retval == -1) {
    perror("ioctl");
    exit(errno);
}

```

코드 523. RTC를 이용하는 예제 프로그램(계속)

이전 Update Interrupt Enable을 OFF시키기 위해서 RTC_UIE_OFF를 ioctl()의 command 코드로 사용한다. RTC의 날짜와 시간을 얻기 위해서 RTC_RD_TIME을 사용하고, 읽을 데이터는 rtc_time 구조체로 정의된 rtc_tm이 가지도록 포인터를 넘겨준다. 이렇게 읽은 데이터를 출력해주고(fprintf()), 읽은 데이터로 부터 5초 후에 alarm이 발생하도록 만든다(RTC_ALM_SET).

```

/* Read the current alarm settings */
retval = ioctl(fd, RTC_ALM_READ, &rtc_tm);
if (retval == -1) {
    perror("ioctl");
    exit(errno);
}
fprintf(stderr, "Alarm time now set to %02d:%02d:%02d.\n",      rtc_tm.tm_hour, rtc_tm.tm_min, rtc_tm.tm_sec);

/* Enable alarm interrupts */
retval = ioctl(fd, RTC_AIE_ON, 0);
if (retval == -1) {
    perror("ioctl");
    exit(errno);
}
fprintf(stderr, "Waiting 5 seconds for alarm..."); fflush(stderr);

/* This blocks until the alarm ring causes an interrupt */
retval = read(fd, &data, sizeof(unsigned long));
if (retval == -1) {

```

```

        perror("read");
        exit(errno);
    }
irqcount++;
fprintf(stderr, " okay. Alarm rang.\n");

/* Disable alarm interrupts */
retval = ioctl(fd, RTC_AIE_OFF, 0);
if (retval == -1) {
    perror("ioctl");
    exit(errno);
}

```

코드 524. RTC를 이용하는 예제 프로그램(계속)

이번에는 앞에서 설정한 alarm이 제대로 설정되어 있는가를 알아보는 부분이다. RTC_ALM_READ를 사용해서 rtc_time 구조체인 rtc_tm에 RTC의 alarm과 관련된 데이터를 읽어와서 출력한다. 이전 설정된 alarm 시간이 되었을 때, 인터럽트를 발생시키도록 RTC_AIE_ON을 사용한다. 설정후 바로 RTC에 대해서 read를 호출해서, 인터럽트가 발생했을 때 인터럽트 데이터를 가져오도록 한다. 역시 irqcount는 인터럽트를 counting한다. 이전 앞에서 설정한 alarm interrupt enable을 OFF시키기 위해서 RTC_AIE_OFF를 사용했다.

```

/* Read periodic IRQ rate */
retval = ioctl(fd, RTC_IRQP_READ, &tmp);
if (retval == -1) {
    perror("ioctl");
    exit(errno);
}
fprintf(stderr, "\nPeriodic IRQ rate was %ldHz.\n", tmp);

fprintf(stderr, "Counting 20 interrupts at:");
fflush(stderr);

```

코드 525. RTC를 이용하는 예제 프로그램(계속)

RTC_IRQP_READ는 RTC에 설정된 periodic interrupt의 인터럽트 발생빈도를 읽어온다. 이 값을 화면에 표시하도록 한다. 이미 앞에서 주기적인(periodic interrupt)의 발생빈도를 RTC의 Register A를 통해서 알 수 있다는 것은 보았다. 발생빈도는 Hz의 단위로 표시될 것이다.

```

/* The frequencies 128Hz, 256Hz, ... 8192Hz are only allowed for root. */
for (tmp=2; tmp<=64; tmp*=2) {
    retval = ioctl(fd, RTC_IRQP_SET, tmp);
    if (retval == -1) {
        perror("ioctl");
        exit(errno);
    }
    fprintf(stderr, "\n%ldHz:\t", tmp);
    fflush(stderr);
    /* Enable periodic interrupts */
    retval = ioctl(fd, RTC_PIE_ON, 0);
    if (retval == -1) {
        perror("ioctl");
        exit(errno);
    }
    for (i=1; i<21; i++) {
        /* This blocks */
        retval = read(fd, &data, sizeof(unsigned long));
    }
}

```

```

        if (retval == -1) {
            perror("read");
            exit(errno);
        }
        fprintf(stderr, " %d", i);
        fflush(stderr);
        irqcount++;
    }
/* Disable periodic interrupts */
retval = ioctl(fd, RTC_PIE_OFF, 0);
if (retval == -1) {
    perror("ioctl");
    exit(errno);
}
fprintf(stderr, "\n\n\t\t *** Test complete ***\n");
fprintf(stderr, "\nTyping \'cat /proc/interrupts\' will show %d more events on IRQ 8.\n", irqcount);
close(fd);
return 0;
} /* end main */
}

```

코드 526. RTC를 이용하는 예제 프로그램(계속)

이젠 128 Hz에서부터 64 Hz까지 주기적인 인터럽트를 발생시켜주는 test를 실행한다. tmp 변수로 주어진 값이 주기적인 인터럽트가 발생할 Hz를 가진다고 보면 될 것이다. RTC_IRQP_SET으로 RTC에 대한 주기적인 인터럽트를 설정하기 위해서 사용한다. 주기적인 인터럽트를 ON시키기 위해서는 RTC_PIE_ON을 사용하며, 다시 OFF시키기 위해서는 RTC_PIE_OFF를 사용한다. ON과 OFF중간에 read를 호출해서 주기적인 인터럽트가 발생했을때 데이터를 읽어오도록 대기하게 만든다. 한번의 주기에 대해서 총 20번의 인터럽트가 발생할 때까지 수행되도록 한다. 마지막은 RTC에 대한 test 결과를 보여주는 부분이다. irqcount가 RTC에서 발생한 총 인터럽트의 회수를 가진다. 물론 이것은 사용자가 알아차린(detect) 인터럽트의 수이다.¹⁹¹

이상에서 우리는 Linux에서 높은 정밀도를 가지는 주기적인 clock을 사용하고 싶다면, RTC를 사용할 수 있다는 것과 RTC의 구현 및 어떻게 응용 프로그램에서 사용하는가를 이번장에서 알 수 있었다. 또한 이를 이용해서 시스템의 성능을 어떻게 알 수 있는가도 대략적으로 알 수 있었다. 응용프로그램의 특성이 RTC를 요구한다면, 그 적절한 사용 방법도 중요하지만 실제 내부구조는 어떻게 되어 있는가를 이해하는 것이, 디바이스 드라이버와 application의 상호작용(interaction)을 이해하는데 핵심이 된다는 것을 간과해선 안될 것이다. Open source의 좋은 점이 바로 이러한 것을 모두 공개한다는 점이기에 공부하는데도 많은 도움을 줄 것이다.

¹⁹¹ RTC를 이용하는 프로그램 중에서 WEB을 통해서 얻을 수 있는 “realfeel”이라는 프로그램이 있다. 이 프로그램은 RTC에 특정 주기로 interrupt를 발생하도록 만들어서, 얼마나 잘 real-time event에 반응하는가를 측정해서 이를 histogram의 형태로 표시해 주는 프로그램이다. RedHat과 같은 곳에서 자신들의 real-time patch의 성능을 검증하기 위해서 인위적으로 시스템의 load를 발생시키는 툴(tool)과 같이 사용하기에 유용하고 적절한 툴이라고 생각한다. 이 프로그램의 특징은 x86 계열의 PC에서만 작동한다는 것¹⁹¹과 측정한 결과를 GNU Plot과 같은 것을 통해서 나타낼 수 있도록 해준다는 것이다.

8. Power Management in Linux

Power management란 시스템의 전원을 관리하는 것이다. 이것은 항상 전원에 맞물려서 사용되는 시스템¹⁹² 및 점차 그 중요성을 더해가는 mobile(이동형) 기기에서 사용된다. 물론, 이동형 기기는 이와 같은 것이 핵심으로 구현되어야 하는 기능이다. 이번 장에서는 Linux에서는 어떤 방식으로 전원을 관리하는지를 내부구조의 분석을 통해서 알아보고, 커널 programmer의 입장에서 어떻게 하면 전원 관리 기능을 구현할 수 있는지를 보도록 하겠다. 참고로 이곳에서 보는 코드들은 `~/linux/include/pm.h`와 `~/linux/kernel/pm.c`에 있는 것들이다.

8.1. Power Management를 위한 자료구조

Linux에서는 power management를 제공하는 디바이스들은 `pm_dev`라고 하는 구조체를 정의해서 사용해야 한다. 즉, `pm_dev` 구조체에는 power management를 하기 위한 디바이스에서 필요한 모든 정보를 가질 수 있는 필드들이 존재한다. 정의는 아래와 같다.

```
struct pm_dev
{
    pm_dev_t type;          /* Power management device의 type */
    unsigned long id;        /* Power management device의 ID */
    pm_callback callback;    /* Callback function */
    void *data;              /* Callback function에 대한 데이터 포인터 */
    unsigned long flags;     /* 현재(Linux Kernel version 2.4.18)로는 사용되지 않음*/
    int state;               /* Power management device의 상태 */
    int prev_state;          /* Power management device의 이전 상태 */
    struct list_head entry;  /* pm_dev 구조체의 연결 리스트 */
};
```

코드 527. `pm_dev` 구조체의 정의

`pm_dev_t`은 `integer`로 정의된 값이며, power management를 할 디바이스가 어떤 디바이스 `class`에 속하는지를 나타내준다. 아래와 같다.

```
enum
{
    PM_UNKNOWN_DEV = 0, /* generic */
    PM_SYS_DEV,        /* system device (fan, KB controller, ...) */
    PM_PCI_DEV,        /* PCI device */
    PM_USB_DEV,        /* USB device */
    PM_SCSI_DEV,       /* SCSI device */
    PM_ISA_DEV,        /* ISA device */
    PM_MTD_DEV,        /* Memory Technology Device */
};
```

코드 528. `pm_dev_t`의 값에 대한 정의

위에서 comment로 처리된 부분에 어떤 디바이스가 속하지를 보여주고 있다. `PM_UNKNOWN_DEV`가 일반적인 값으로 사용되며 0을 가진다는 것을 볼 수 있으며, SCSI와 MTD(Memory Technology Device)가 하나의 `class`로 이루어지고 있음을 알 수 있다. PCI 및 ISA는 시스템의 bus type에 따라서 분류된 것이다. 자세히 보면 알 수 있겠지만, 각각의 `class`가 전부 시스템에서 device를 어떤 bus(혹은 serial bus)를 이용해서 부착시키는가에 따라서 분류하고 있다고 봐도 좋을 것이다. 그외에 시스템에 필수적인 디바이스들은 `PM_SYS_DEV`에 속한다고 본다.

¹⁹² 물론 이와 같은 시스템에서는 전원이 항상 연결된 상태이므로 따로이 관리할 필요성이 줄어들지만, 소비전력을 줄인다는 의미에서 시스템을 운용하는 비용을 절감할 수 있기 때문에 필요하다.

```

/*
 * System device hardware ID (PnP) values
 */
enum
{
    PM_SYS_UNKNOWN = 0x00000000, /* generic */
    PM_SYS_KBC = 0x41d00303, /* keyboard controller */
    PM_SYS_COM = 0x41d00500, /* serial port */
    PM_SYS_IRDA = 0x41d00510, /* IrDA controller */
    PM_SYS_FDC = 0x41d00700, /* floppy controller */
    PM_SYS_VGA = 0x41d00900, /* VGA controller */
    PM_SYS_PCMCIA = 0x41d00e00, /* PCMCIA controller */
};

/*
 * Device identifier
 */
#define PM_PCI_ID(dev) ((dev)->bus->number << 16 | (dev)->devfn)

```

코드 529. Power management에서 사용하는 device ID에 대한 정의

pm_dev 구조체의 id필드에 들어갈 값을 보면, 먼저 해당 device가 시스템을 구성하는 기본적인 hardware인 경우에는 이에 해당하는 PNP(Plug & Play) ID값을 주도록 한다. 각각은 PM_SYS_XXX로 나타내진다. 이것에 해당하는 디바이스로는 keyboard, serial port, IrDA, floppy, VGA, PCMCIA가 해당한다. 그외의 디바이스 ID를 구하는 방법으로는 PCI 디바이스의 경우에는 고유한 ID값을 얻기위해서 device가 맞물려 있는 bus의 번호를 좌측으로 16bit shift한 값과 현재 device가 동작하고 있는 function값을 OR시켜서 사용한다¹⁹³.

```
typedef int (*pm_callback)(struct pm_dev *dev, pm_request_t rqst, void *data);
```

코드 530. pm_callback 함수의 정의

pm_callback은 power management를 위한 callback함수에 대한 정의를 가진다. 즉, 각각의 power management를 필요로하는 디바이스가 자신이 수행할 power management에 관련된 action을 이와 같은 callback에 저장해서 pm_request_t에 맞는 action들을 이 함수내에서 수행하면 될 것이다. Callback함수가 넘겨받는 값은 pm_dev구조체와 pm_request_t 구조체로 만들어진 power management 요청, 그리고 마지막으로 임의의 데이터를 가르킬 수 있는 포인터이다.

```

/*
 * Power management requests
 */
enum
{
    PM_SUSPEND, /* enter D1-D3 : 디바이스의 동작을 중단하라. */
    PM_RESUME, /* enter D0 : 디바이스의 동작을 다시 시작하라. */
    PM_SAVE_STATE, /* save device's state : 디바이스의 이전 상태를 저장하라. */
    /* enable wake-on */
    PM_SET_WAKEUP, /* 디바이스를 wakeup( 깨어남 ) 상태로 만들어라. */
    /* bus resource management */
    PM_GET_RESOURCES, /* 디바이스가 사용할 bus의 resource(자원)을 다시 얻어라. */

```

¹⁹³ 일반적으로 PCI device는 다양한 function을 구현할 수 있다. 즉, 예를 들어 LAN에 및 전화선을 이용한 networking을 둘다 지원하는 경우의 디바이스가 있을 수 있다. 따라서, 현재 디바이스가 어떤 function을 enable하고 있는지를 알아둘 필요가 있다. 이 값은 PCI configuration register SPEC.에 있는 특정 메모리 공간을 읽어내서 PCI device의 연결리스트를 만들어주는 PCI subsystem을 초기화 하는 과정에서 만들어지며, 경우에 따라서는 디바이스 드라이버와 같은 곳에서도 설정할 수 있다.

```

PM_SET_RESOURCES, /* 디바이스가 사용할 bus의 자원을 설정하라.*/
/* base station management */
PM_EJECT, /* 디바이스에 insert된 장치를 배출하라.*/
PM_LOCK, /* 디바이스를 locking하라. Button에 대한 event에 반응하지 않음.*/
};

typedef int pm_request_t;

```

코드 531. pm_request_t의 정의와 power management request의 종류

pm_request_t은 디바이스가 power management에 어떻게 반응해야 할 것인가를 정의해둔 값으로 생각하면 될 것이다. 앞에서 본 callback함수를 통해서 전달되는 값이며, OR가 되어서 사용될 수도 있는데, 디바이스 드라이버를 만들고자 한다면, 이러한 값에 따라서 디바이스를 제어(control)해 주어야 할 것이다.

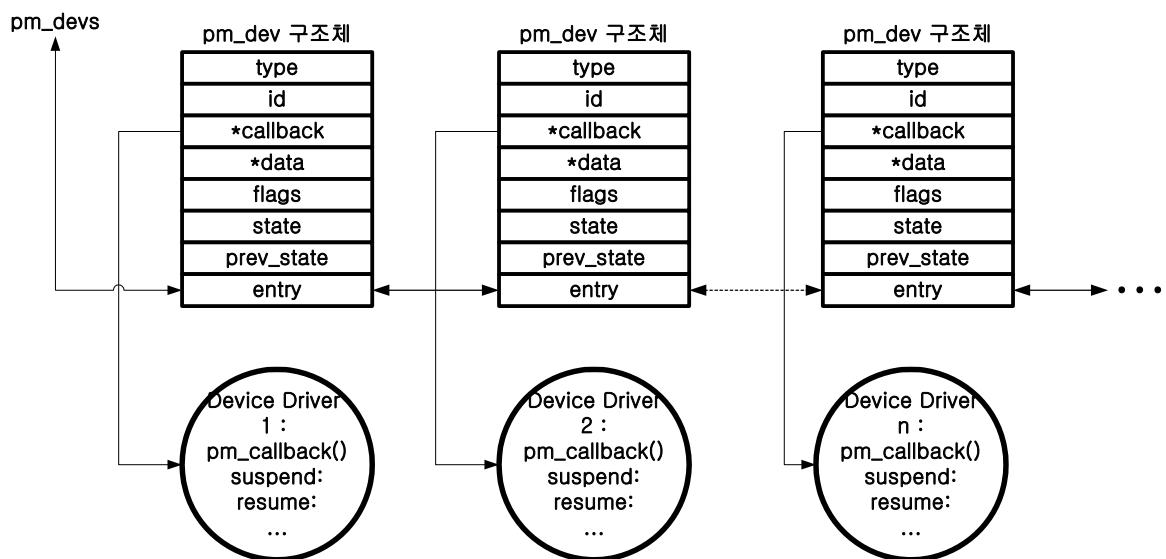


그림 68. Linux의 Power Management 구성

앞에서 본 것 이외에 power management를 위해서 사용하는 전역변수로는 pm_active라는 것과 power management의 리스트를 관리하기 위한 lock인 pm_devs_lock, 그리고 마지막으로 power management를 위한 디바이스의 연결리스트의 head인 pm_devs가 있다. pm_active는 시스템이 power management를 지원한다면 1로 설정되며, 그외의 경우에는 0을 가진다¹⁹⁴.

8.2. Power Management의 등록

```

struct pm_dev *pm_register(pm_dev_t type, unsigned long id, pm_callback callback)
{
    struct pm_dev *dev = kmalloc(sizeof(struct pm_dev), GFP_KERNEL);
    if (dev) {
        memset(dev, 0, sizeof(*dev));
        dev->type = type;
        dev->id = id;
        dev->callback = callback;

        down(&pm_devs_lock);
    }
}

```

¹⁹⁴ 현재로서는 pm_active의 설정이 ACPI와 APM에 달려있다. 이곳에서 지원 여부나 loading 여부에 따라서, 1이나 0의 값으로 설정된다.

```

        list_add(&dev->entry, &pm_devs);
        up(&pm_devs_lock);
    }
    return dev;
}

```

코드 532. pm_register() 함수의 정의

디바이스 드라이버가 power management를 제공한다면, pm_register() 함수를 호출해서 자신이 power management에 관심이 있다는 것을 알려야 할 것이다. pm_register() 함수에서는 pm_dev구조체를 생성해서 이를 pm_devs 연결리스트에 두는 일을 수행한다. 먼저 pm_dev구조체를 할당하기 위해서 kmalloc()을 수행하고, pm_dev필드를 0으로 초기화 한 후, 각각의 필드에 해당하는 값을 넣도록 한다. pm_register() 함수의 호출에서 따라오는 인자 값이 이곳에 들어갈 것이다. Type과 id, 그리고 callback() 함수를 설정하는 값이 들어가게 되며, pm_dev_lock에 semaphore을 설정하고(down()), list_add()를 통해서 pm_devs에 이렇게 초기화된 pm_dev구조체를 연결한다. 연결을 마치고나면 semaphore을 해제하기 위해서 up()을 호출한다. 복귀 값은 새로 할당된 pm_dev구조체의 포인터이거나 시스템의 메모리가 없을 경우에는 NULL이 될 것이다.

8.3. Power Management의 해제

```

void pm_unregister(struct pm_dev *dev)
{
    if (dev) {
        down(&pm_devs_lock);
        list_del(&dev->entry);
        up(&pm_devs_lock);

        kfree(dev);
    }
}

```

코드 533. pm_unregister() 함수의 정의

pm_register() 함수가 power management 를 위한 자료구조와 함수를 등록했다면, 이를 해제하는 것은 pm_unregister() 함수이다. pm_unregister()는 pm_dev구조체의 포인터를 넘겨받는다. 먼저 power management를 위한 lock을 설정한 후(down()), pm_dev 구조체를 pm_devs list에서 제거한다(list_del()). 이후 앞에서 설정했던 lock을 해제하고(up()), 마지막으로 pm_dev 구조체로 할당된 메모리를 해제한다(kfree()). 즉, 단지 앞에서 power management로 등록한 pm_dev구조체를 해제하는 일을 하는 것이 pm_unregister() 함수이다.

```

static void __pm_unregister(struct pm_dev *dev)
{
    if (dev) {
        list_del(&dev->entry);
        kfree(dev);
    }
}

/**
 * pm_unregister_all - unregister all devices with matching callback
 * @callback: callback function pointer
 *
 * Unregister every device that would call the callback passed. This
 * is primarily meant as a helper function for loadable modules. It
 * enables a module to give up all its managed devices without keeping
 * its own private list.
 */

```

```

void pm_unregister_all(pm_callback callback)
{
    struct list_head *entry;

    if (!callback)
        return;

    down(&pm_devs_lock);
    entry = pm_devs.next;
    while (entry != &pm_devs) {
        struct pm_dev *dev = list_entry(entry, struct pm_dev, entry);
        entry = entry->next;
        if (dev->callback == callback)
            __pm_unregister(dev);
    }
    up(&pm_devs_lock);
}

```

코드 534. pm_unregister_all() 함수의 정의

두번째로 power management의 해제와 관련된 것은 pm_unregister_all() 함수이다. 함수의 이름에서 알 수 있듯이 이 함수는 pm_devs 리스트에 등록된 pm_dev 구조체를 해제하는 역할을 수행한다. 하지만, 이 함수는 완전히 함수의 이름에서 의미하는 것을 하지는 않는다. 즉, 이 함수는 pm_callback이라고 하는 것과 같은 callback 함수를 가지는 pm_dev 구조체를 해제하는 일을 수행한다. 이 함수를 호출하는 것은 당연히 디바이스 드라이버와 같은 곳이 될 것이다. 즉, 자신이 등록한 power management callback 함수를 해제하고 싶을 때 이 함수를 호출할 것이다.

먼저 넘겨받은 callback 인자가 NULL인 경우에는 그대로 복귀한다. 그렇지 않다면, pm_devs_lock에 lock을 설정하고, pm_devs list의 next를 통해서 연결된 pm_dev 구조체를 접근하기 위해 entry를 초기화 한다. 이전 pm_devs 리스트를 하나씩 callback과 같은 callback 함수를 가지는 것이 있는지를 살펴보는 단계이다. 먼저 entry로부터 pm_dev 구조체의 한 entry를 가져온다(list_entry()). 이것을 dev로 두고, dev의 callback 필드가 넘겨받은 callback과 같은지를 비교한 후, 만약 같다면, __pm_unregister() 함수를 호출하도록 한다. 이렇게 모든 pm_devs 리스트에 있는 pm_dev 구조체를 살펴보았다면, 앞에서 설정한 lock을 해제하고(up()) 복귀 값 없이 그냥 return 한다.

__pm_unregister() 함수는 앞에서 본 pm_unregister() 함수와 동일하다. 다만 한가지 차이가 나는 점은 앞의 pm_unregister() 함수는 lock을 설정하지 않고 호출되기에 내부적으로 lock을 설정하지만, __pm_unregister() 함수는 이미 lock이 설정된 상황에서 호출된다는 점이다.

8.4. Power Management의 찾기

```

struct pm_dev *pm_find(pm_dev_t type, struct pm_dev *from)
{
    struct list_head *entry = from ? from->entry.next:pm_devs.next;
    while (entry != &pm_devs) {
        struct pm_dev *dev = list_entry(entry, struct pm_dev, entry);
        if (type == PM_UNKNOWN_DEV || dev->type == type)
            return dev;
        entry = entry->next;
    }
    return 0;
}

```

코드 535. pm_find() 함수의 정의

`pm_find()` 함수는 특정 power management type¹⁹⁵에 속하는 `pm_dev`를 찾는 역할을 수행한다. 이때 찾아야 할 위치도 함께 넘겨준다(`from`). 먼저 찾기를 시작할 `entry`를 결정한다. 만약 `from`에 어떤 값이 있다면, `from`의 `entry` 필드가 가르키는 `next`필드로부터 찾기를 시작하고, 그렇지 않다면 `pm_devs`의 리스트의 `next` 필드부터 시작한다. 찾기는 `entry`가 `pm_devs`와 같지 않을 때까지 계속된다. 먼저 `entry`로부터 `pm_dev` 구조체를 하나 가지고와서 `dev`가 가르키도록 만든다. 이전 `type`과 같은 가를 찾도록 한다. 만약 `type`이 `PM_UNKNOWN_DEV`이거나, 혹은 `dev`의 `type`필드가 `type`과 일치한다면, 바로 `dev`를 돌려주고 복귀한다. 그렇지 않다면, `entry`는 다음을 가르키도록 만든다(`entry=>next`). 이 함수가 찾기를 마쳤고, 찾는 power management 디바이스가 없다면, 0을 돌려준다.

8.5. Power Management의 명령 내리기

```
int pm_send(struct pm_dev *dev, pm_request_t rqst, void *data)
{
    int status = 0;
    int prev_state, next_state;

    if (in_interrupt())
        BUG();

    switch (rqst) {
    case PM_SUSPEND:
    case PM_RESUME:
        prev_state = dev->state;
        next_state = (int) data;
        if (prev_state != next_state) {
            if (dev->callback)
                status = (*dev->callback)(dev, rqst, data);
            if (!status) {
                dev->state = next_state;
                dev->prev_state = prev_state;
            }
        }
        else {
            dev->prev_state = prev_state;
        }
        break;
    default:
        if (dev->callback)
            status = (*dev->callback)(dev, rqst, data);
        break;
    }
    return status;
}
```

코드 536. `pm_send()` 함수의 정의

`pm_send()` 함수는 power management와 관련된 명령을 내리는 함수이다. 이 함수가 호출되는 곳은 APM(Advanced Power Management) 모듈이나, 혹은 Intel의 APIC(Advanced Programmable Interrupt Controller)와 같은 곳이다¹⁹⁶. `pm_send()` 함수가 하는 일은 앞에서 등록된 `pm_devs` 리스트의 `pm_dev` 구조체들에 있는 callback 함수들을 호출하는 것이다.

¹⁹⁵ 앞에서 이미 power management의 device type들에 대해서 이야기를 했었다. 즉, `PM_UNKNOWN_DEV`, `PM_SYS_DEV`, `PM_PCI_DEV`, `PM_USB_DEV`, `PM_SCSI_DEV`, `PM_ISA_DEV`, `PM_MTD_DEV` 등을 이야기 한다.

¹⁹⁶ StrongARM 및 Machintosh의 경우에 대해서도 커널 코드에서 찾을 수 있을 것이다.

넘겨받는 인자는 먼저 pm_dev 구조체에 대한 포인터와 power management 요청(request)를 가지는 pm_request_t 구조체인 rqst, 그리고 마지막으로 해당 call back함수를 호출할 때 사용할 void 포인터를 가지는 data이다. 여기서 pm_request_t은 앞에서 이미 보았지만, integer로 선언된 값이다.

만약 현재 interrupt를 service하고 있는 중(in_interrupt())이라면, BUG()를 통해서 문제가 있음을 보고 하고, 복귀 한다. 즉, interrupt service하고 있는 도중에 power management를 실행할 수 없다. 그리고나서는 switch() 문을 통해서 어떤 power management 요청을 처리할지를 결정한다. 기본적으로 PM_SUSPEND와 PM_RESUME을 처리하도록 하며, 나머지에 대해서는 디바이스 드라이버의 specific이라고 생각해서 그냥 call back함수를 호출하기만 한다. PM_SUSPEND와 PM_RESUME의 처리 절차를 보면, 먼저 현재 power management의 상태를 구해와서 prev_state로 둔다. 그리고, 다음번에 가질 상태(next_state)를 data에서 가져온다. 만약 현재 상태와 다음 상태가 같지 않은 경우에 대해서만, 디바이스 드라이버에서 제공한 call back함수가 있다면, 이를 호출해서 결과 값을 status에 둔다. 만약 status가 0인 값을 가진다면, 오류가 없었다는 것으로 생각해, 디바이스의 power management상태값을 next_state로 변경한다. 그리고, 당연히 이전의 상태를 기록하는 prev_state 필드는 prev_state로 바뀔 것이다. 만약 이전의 상태와 다음 상태가 같다고 생각되면, prev_state만 갱신한다. switch() 문의 default값으로 실행되는 것은 callback함수가 있을 경우, 이를 실행해주는 것이다. 이때 실행될 call back함수에서 data에 대한 처리를 하게 될 것이다. 복귀 값은 call back함수를 수행한 결과 값이 주어진다.

```
int pm_send_all(pm_request_t rqst, void *data)
{
    struct list_head *entry;

    down(&pm_devs_lock);
    entry = pm_devs.next;
    while (entry != &pm_devs) {
        struct pm_dev *dev = list_entry(entry, struct pm_dev, entry);
        if (dev->callback) {
            int status = pm_send(dev, rqst, data);
            if (status) {
                /* return devices to previous state on
                 * failed suspend request
                 */
                if (rqst == PM_SUSPEND)
                    pm_undo_all(dev);
                up(&pm_devs_lock);
                return status;
            }
        }
        entry = entry->next;
    }
    up(&pm_devs_lock);
    return 0;
}
```

코드 537. pm_send_all() 함수의 정의

pm_send_all() 함수는 pm_devs 리스트에 등록된 모든 pm_dev 구조체들에 power management와 관련된 것을 처리하도록 하는 일을 한다. 넘겨받는 값은 당연히 power management와 관련된 요청(request)이미 call back함수를 호출하기 위한 data의 포인터이다.

먼저 power management에 대한 lock을 설정하고(down()), pm_devs 구조체의 next 필드를 통해서 등록된 pm_dev 구조체를 접근하도록 한다(entry). entry가 pm_devs와 같지 않은 동안 다음과 같은 일이 진행될 것이다. 먼저 pm_devs에서 하나씩 pm_dev 구조체를 하나씩 가져와서 dev가 가르키도록 만든다. 만약 해당 pm_dev구조체의 callback함수가 있다면, 이 함수를 수행하기 위해서 앞에서 본 pm_send() 함수를 호출해 주도록 한다. 넘겨주는 값은 앞에서 얻은 pm_dev구조체에 대한 포인터와 요청(request), 그리고 넘겨받은 data가 될 것이다. pm_send()의 호출 결과를 status에 두고, 만약 status가 0이 아닌 경우(즉, 원가 오류가 있었다면), 요청(rqst)이 PM_SUSPEND인 경우라면, pm_undo_all()을 호출해서 해당 device에 대한

PM_SUSPEND요청을 무효로 하도록 한다. 그리고나서, lock을 해제한 후(up()), status 값을 복귀 값으로 돌려준다. 그렇지 않다면, 해당 디바이스에 대해서 power management연산이 제대로 다 수행된 것이기에 entry는 다음으로 이동 시키도록 한다(entry=entry->next). 이 while() loop를 다 수행하고 나면, 등록된 power management에 대한 것을 다 처리한 것이므로, 앞에서 설정한 lock을 해제하고(up()) 0을 돌려준다.

```
static void pm_undo_all(struct pm_dev *last)
{
    struct list_head *entry = last->entry.prev;
    while (entry != &pm_devs) {
        struct pm_dev *dev = list_entry(entry, struct pm_dev, entry);
        if (dev->state != dev->prev_state) {
            /* previous state was zero (running) resume or
             * previous state was non-zero (suspended) suspend
             */
            pm_request_t undo = (dev->prev_state
                                ? PM_SUSPEND:PM_RESUME);
            pm_send(dev, undo, (void*) dev->prev_state);
        }
        entry = entry->prev;
    }
}
```

코드 538. pm_undo_all() 함수의 정의

pm_undo_all() 함수는 앞서 실행되었던 power management관련 요청을 다시 원래의 상태로 돌리는 일을 한다. 즉, 이는 디바이스의 power management와 관련된 상태를 이전의 상태로 복귀 시킨다는 의미를 가진다. 넘겨받는 것은 pm_dev에 대한 포인터이다. 여기서 한가지 중요한 점은 이 함수는 앞에서 pm_send_all()을 호출하는 도중에 오류에 의해서 수행된다는 점이다. 따라서, 해당 디바이스에 대한 power management를 실패한 경우가 된는 것으로, 그 이전의 디바이스들에 대한 power management 명령도 원상태로 복귀 시켜야 한다는 것이다. 따라서, 현재 pm_dev구조체로 부터 이전에 이미 수행된 pm_dev 구조체들에 대한 포인터를 얻어와서 이를 처리한다. 따라서, 넘겨받은 last의 entry에서 prev 필드를 통해서 pm_dev 구조체에 접근하는 것이 타당하다. 이 요청은 entry가 pm_devs와 같지 않을 때까지 역으로 진행한다(entry=entry->prev). pm_devs 리스트에서 하나의 해당 디바이스 pm_dev 구조체의 entry를 가져와서 dev로 가르키도록 두고, pm_request_t 타입의 undo에는 디바이스의 power management에 대한 이전 상태에 따라서, PM_SUSPEND나 PM_RESUME로 두도록 한다. PM_SUSPEND가 이전의 상태라면, PM_RESUME이 될 것이며, PM_RESUME이 이전의 상태라면, PM_SUSPEND가 될 것이다. 이렇게 만든 data 값을 가지고, 원래의 상태로 복귀 시키도록 하기 위해 pm_send() 함수를 호출하도록 한다.

8.6. Power Management in PCI

이전 power management에 대한 한가지 예를 보기로 하자. 여기서 볼 예는 PCI(Peripheral Controller Interface)에서 power management이다. PCI 인터페이스는 현재 많이 사용되고 있는 주변기기 연결 방법이므로, power management의 예로서 적당하리라 생각한다.

```
void __devinit pci_init(void)
{
    ...
#ifndef CONFIG_PM
    pm_register(PM_PCI_DEV, 0, pci_pm_callback);
#endif
}
```

코드 539. pci_init() 함수에서의 power management 등록

PCI 하부 시스템(subsystem)은 자신을 초기화 하는 pci_init() 함수에서 CONFIG_PM 설정이 있으면, pm_register() 함수를 호출해서 pm_dev 구조체를 pm_devs 리스트에 등록한다. 이때 callback으로 수행될

함수는 `pci_pm_callback()` 함수이다. 앞에서 이미 설명했듯이 PCI와 관련된 디바이스의 `type`에는 `PM_PCI_DEV`를 사용한다. `pm_register()`함수를 호출하고 나면, PCI 하위 시스템은 이제 power management 요청을 처리할 준비가 완료된 것으로 생각할 것이다.

```
static int pci_pm_callback(struct pm_dev *pm_device, pm_request_t rqst, void *data)
{
    int error = 0;

    switch (rqst) {
    case PM_SAVE_STATE:
        error = pci_pm_save_state((u32)data);
        break;
    case PM_SUSPEND:
        error = pci_pm_suspend((u32)data);
        break;
    case PM_RESUME:
        error = pci_pm_resume();
        break;
    default: break;
    }
    return error;
}
```

코드 540. `pci_pm_callback()` 함수의 정의

앞에서 보았듯이 `callback` 함수는 시스템에서 power management의 요청을 내리기 위해서 `pm_send()`를 호출할 때 호출되는 함수이다. 따라서, PCI 하위 시스템도 power management를 위해서 `pci_pm_callback()` 함수를 통해서 자신이 원하는 power management를 하면 된다. 넘어오는 요청 중에서 `PM_SAVE_STATE`와 `PM_SUSPEND`, `PM_RESUME`를 처리하고, 나머지 경우에 대해서는 에러 코드로 0을 넘겨주도록 한다¹⁹⁷. 각각의 경우에 대해서 `pci_pm_save_state()`, `pci_pm_suspend()`, `pci_pm_resume()` 함수가 호출되어 요청을 처리하고, 함수의 처리 중에 발생한 복귀 값을 `pci_pm_callback()`의 복귀 값으로 사용한다.

```
static int pci_pm_save_state(u32 state)
{
    struct list_head *list;
    struct pci_bus *bus;
    int error = 0;

    list_for_each(list, &pci_root_buses) {
        bus = pci_bus_b(list);
        error = pci_pm_save_state_bus(bus,state);
        if (!error)
            error = pci_pm_save_state_device(bus->self,state);
    }
    return error;
}
```

코드 541. `pci_pm_save_state()` 함수의 정의

시스템에 있는 모든 PCI bus는 `pci_root_buses`라는 리스트에 연결되어 있다. 따라서, 우린 이렇게 연결된 bus들 상에 있는 PCI 디바이스들에게 power management와 관련된 연산을 하도록 해주어야 할 것이다. 이를 위해서 각 bus를 하나씩 가져오도록 한다(`bus = pci_bus_b()`). 이젠 이 bus상에 놓인 디바이스들에

¹⁹⁷ 에러 복귀 코드로 0은 오류를 나타내는 것이 아니라는 것을 `pm_send()` 함수에서 확인할 수 있을 것이다.

PM_SAVE_STATE라는 명령을 주기 위해서 pci_pm_save_state_bus()를 호출한다. 호출의 결과가 0이라면, 이전 bridge가 되는 디바이스에 대해서 PM_SAVE_STATE를 수행하기 위해서, pci_pm_save_state_device()를 호출하도록 한다. 호출의 인자로 넘겨주는 것은 pci_bus 구조체의 필드인 self¹⁹⁸와 state이다.

```
static int pci_pm_save_state_bus(struct pci_bus *bus, u32 state)
{
    struct list_head *list;
    int error = 0;

    list_for_each(list, &bus->children) {
        error = pci_pm_save_state_bus(pci_bus_b(list),state);
        if (error) return error;
    }
    list_for_each(list, &bus->devices) {
        error = pci_pm_save_state_device(pci_dev_b(list),state);
        if (error) return error;
    }
    return 0;
}
```

코드 542. pci_pm_save_state_bus() 함수의 정의

pci_pm_save_state_bus() 함수는 자신의 bus이하에 놓인 PCI bus들 및 각 bus상에 있는 device들에 대한 PM_SAVE_STAE 요청을 처리한다. 먼저 자신의 bus상에 놓인 디바이스들에 대한 요청을 하기 전에 자신 보다 하위에 있는(children) bus들에 PM_SAVE_STATE 명령을 요청하기 위해서 recursive하게 자기 자신을 다시 호출한다. 물론 이때 주어지는 인자는 하위 bus에 대한 연결 리스트의 entry가 될 것이다(list). 에러가 있었다면 곧바로 error 코드를 돌려주고 복귀한다. 에러가 없었다면, 이젠 자신의 bus상에 놓인 디바이스들에 대한 PM_SAVE_STATE 요청을 처리하기 위해서 pci_pm_save_state_device() 함수를 호출한다. 이때는 자신의 bus상에 놓인 PCI 디바이스에 대한 연결 리스트를 넘겨주도록 한다(bus->devices). 에러가 있다면(error != 0) 에러 코드를 돌려주고 복귀하고, 그렇지 않다면 0을 복귀 값으로 사용한다.

```
static int pci_pm_save_state_device(struct pci_dev *dev, u32 state)
{
    int error = 0;
    if (dev) {
        struct pci_driver *driver = dev->driver;
        if (driver && driver->save_state)
            error = driver->save_state(dev,state);
    }
    return error;
}
```

코드 543. pci_pm_save_state_device() 함수의 정의

pci_pm_save_state_device() 함수는 PCI bus상에 놓인 디바이스에 대한 PM_SAVE_STATE 요청을 수행하는 일을 한다. pci_driver 구조체를 접근해서 save_state 필드에 설정된 함수가 있을 경우, 이를 호출하는 역할을 한다. 여기서 잠시 pci_driver 구조체를 보면 아래와 같다.

```
struct pci_driver {
    struct list_head node;
    char *name;
    const struct pci_device_id *id_table; /* NULL if wants all devices */
    int (*probe) (struct pci_dev *dev, const struct pci_device_id *id); /* New device inserted */
```

¹⁹⁸ self 필드는 pci_dev 구조체로 선언되어 있다.

```

void (*remove)(struct pci_dev *dev); /* Device removed (NULL if not a hot-plug capable driver) */
int (*save_state)(struct pci_dev *dev, u32 state); /* Save Device Context */
int (*suspend)(struct pci_dev *dev, u32 state); /* Device suspended */
int (*resume)(struct pci_dev *dev); /* Device woken up */
int (*enable_wake)(struct pci_dev *dev, u32 state, int enable); /* Enable wake event */
};

```

코드 544. pci_driver 구조체의 정의

pci_driver 구조체를 보면 power management와 관련된 3개의 필드를 볼 수 있을 것이다. 즉, save_state()와 suspend(), resume()등이 PCI 디바이스 드라이버에서 power management를 위해서 필요한 부분들이다. 따라서, PCI 디바이스 드라이버를 만들고자 하는 사람이라면, power management를 하기를 원하면 이 부분들에 대해서 함수를 정의해 두어야 할 것이다. 나중에 이렇게 정의된 함수는 PCI 디바이스로 등록할 때 pci_device 구조체의 driver 필드로 부터 접근 가능할 것이다. 이하에서는 suspend()와 resume()에 대해서는 어떤 식으로 호출되는지를 보게되겠지만, 앞에서 본 것과 별반 다른 것이 없다.

pci_pm_save_state_device()에서는 에러가 없었다면, 0을 돌려주면 된다. 물론 이것은 pci_driver 구조체의 save_state() 함수를 수행한 결과값이거나 혹은 그와 같은 필드가 정의되지 않았을 경우에는 0이 될 것이다.

```

static int pci_pm_suspend(u32 state)
{
    struct list_head *list;
    struct pci_bus *bus;

    list_for_each(list, &pci_root_buses) {
        bus = pci_bus_b(list);
        pci_pm_suspend_bus(bus,state);
        pci_pm_suspend_device(bus->self,state);
    }
    return 0;
}

```

코드 545. pci_pm_suspend() 함수의 정의

pci_pm_suspend() 함수는 PM_SUSPEND 요청을 처리하기 위해서 호출된다. 이 함수가 호출되면, 앞에서 보았던 PM_SAVE_STATE와 마찬가지로 PCI root bus로 부터 시작해서 각각의 PCI bus상에 놓인 PCI 디바이스들에 대해서 차례로 PM_SUSPEND 요청을 처리하도록 pci_driver에 있는 suspend() 함수가 호출될 것이다.

먼저 pci_root_buses 리스트로부터 pci_bus 구조체를 하나 가져온다. 이 구조체를 통해서 pci_pm_suspend_bus() 함수를 호출하고, 다시 pci_pm_suspend_device() 함수를 호출한다. 즉, 하위의 PCI bus들에 대한 PM_SUSPEND요청을 처리하도록 하고, 자신의 bus에 있는 디바이스들에 대한 PM_SUSPEND처리를 요청하는 것이다. 특이할만한 것은 항상 복귀 값이 0이라는 점이다. 즉, 이것은 특정 PCI bus나 PCI 디바이스들에 대한 PM_SUSPEND 요청의 결과 값에 신경쓰지 않고, 항상 에러가 없다고 보겠다는 점이다. 이것은 나중에 나올 pci_pm_resume() 함수에서도 마찬가지다.

```

static int pci_pm_suspend_bus(struct pci_bus *bus, u32 state)
{
    struct list_head *list;

    /* Walk the bus children list */
    list_for_each(list, &bus->children)
        pci_pm_suspend_bus(pci_bus_b(list),state);

    /* Walk the device children list */
    list_for_each(list, &bus->devices)

```

```

        pci_pm_suspend_device(pci_dev_b(list),state);
    return 0;
}

```

코드 546. pci_pm_suspend_bus() 함수의 정의

마찬가지로 pci_pm_suspend_bus() 함수의 복귀 값도 항상 0이다. 하위(bus->children)에 있는 PCI bus에 대한 PM_SUSPEND 요청을 처리하기 위해서 자신을 재귀적으로 다시 호출하고 있다. 이렇게 하위 디바이스들에 PM_SUSPEND 요청을 전달했다면, 이젠 자신의 bus상에 있는 디바이스들(bus->devices)에 PM_SUSPEND 요청을 전달하기 위해서 pci_pm_suspend_device() 함수를 호출한다.

```

static int pci_pm_suspend_device(struct pci_dev *dev, u32 state)
{
    int error = 0;
    if (dev) {
        struct pci_driver *driver = dev->driver;
        if (driver && driver->suspend)
            error = driver->suspend(dev,state);
    }
    return error;
}

```

코드 547. pci_pm_suspend_device() 함수의 정의

pci_pm_suspend_device() 함수는 pci_dev 구조체의 driver 필드에 있는 pci_driver 구조체의 suspend() 함수가 설정되어 있는지를 확인하고, 있다면 이를 실행한다. 예러값은 드라이버의 suspend() 함수를 호출한 결과 값이다.

```

static int pci_pm_resume(void)
{
    struct list_head *list;
    struct pci_bus *bus;

    list_for_each(list, &pci_root_buses) {
        bus = pci_bus_b(list);
        pci_pm_resume_device(bus->self);
        pci_pm_resume_bus(bus);
    }
    return 0;
}

```

코드 548. pci_pm_resume() 함수의 정의

pci_pm_resume() 함수도 앞에서와 같은 과정을 수행한다. 다만 여기서 한가지 주의해야 할 것은 pci_pm_suspend() 함수에서 본 bus, 디바이스의 순서로 PM_RESUME을 처리하는 것이 아니라, 먼저 자신의 bus상에 놓인 디바이스들에 대해서 PM_RESUME 요청을 처리하고, 다시 하위의 bus들에 대해 PM_RESUME 요청을 처리하도록 pci_pm_resume_bus()를 호출한다는 점이다. 이때도 PM_RESUME 요청에 대한 bus나 디바이스들에 대한 복귀 값은 사용되지 않으며, 0을 항상 돌려준다.

```

static int pci_pm_resume_device(struct pci_dev *dev)
{
    int error = 0;
    if (dev) {
        struct pci_driver *driver = dev->driver;
        if (driver && driver->resume)
            error = driver->resume(dev);
    }
}

```

```
    return error;
}
```

코드 549. pci_pm_resume_device() 함수의 정의

pci_pm_resume_device() 함수는 pci_dev 구조체의 driver 필드를 접근해서 resume 필드에 정의된 함수를 수행하는 역할을 한다. 근데 여기서 재미있는 것을 볼 수 있다. 앞에서 설명한 save_state()와 suspend()와는 달리, resume() 함수에서는 state가 인자로서 주어지지 않는다는 것이다. 즉, 앞의 두 함수들은 특정 상태로의 디바이스 상태 변경을 요청하기 위해서 사용되지만 resume()의 경우에는 디바이스를 다시 동작시키도록 만들어 주는 역할 밖에 없다는 뜻이된다. 따라서, pci_dev 구조체를 이용해서 디바이스 드라이버를 작성하는 사람은 디바이스를 다시 동작시켜줄 수 있도록 만들어야 할 것이다. 따라서, 이전의 상태를 어느정도 유지할 필요가 있으며, 이를 위해서 사용할 수 있는 pci_dev 구조체의 필드는 driver_data¹⁹⁹가 될 수 있다. 복귀 코드는 드라이버에서 정의하고 있는 resume() 함수의 결과 값이다.

```
static int pci_pm_resume_bus(struct pci_bus *bus)
{
    struct list_head *list;

    /* Walk the device children list */
    list_for_each(list, &bus->devices)
        pci_pm_resume_device(pci_dev_b(list));

    /* And then walk the bus children */
    list_for_each(list, &bus->children)
        pci_pm_resume_bus(pci_bus_b(list));
    return 0;
}
```

코드 550. pci_pm_resume_bus() 함수의 정의

pci_pm_resume_bus() 함수는 자신의 bus 및 하위 bus(bus->children)에 있는 디바이스들에 대한 PM_RESUME 요청을 처리하는 함수이다. 먼저 자신의 bus상에 있는 디바이스들에 대한 PM_RESUME 요청을 처리하기 위해서 pci_pm_resume_device() 함수를 호출하고, 하위 bus들에 대한 PM_RESUME 요청의 처리를 위해서 재귀적으로 자신을 다시 호출한다.

```
static struct pci_driver eepro100_driver = {
    name:          "eepro100",
    id_table:      eepro100_pci_tbl,
    probe:         eepro100_init_one,
    remove:        __devexit_p(eepro100_remove_one),
#ifndef CONFIG_PM
    suspend:       eepro100_suspend,
    resume:        eepro100_resume,
#endif /* CONFIG_PM */
```

코드 551. Intel Ethernet Express Pro 100에서 pci_driver 구조체의 정의 예

그럼 실제 디바이스 드라이버에서는 어떤 식으로 power management를 다루고 있는지 조금 보도록 하겠다. Intel의 Ethernet Express Pro 100(eepro100.c)을 보면 pci_driver 구조체의 필드에 CONFIG_PM이 정의되었을 경우에 사용하게 되는 suspend()와 resume()을 위한 eepro100_suspend() 및 eepro100_resume()가 있음을 알 수 있다.

¹⁹⁹ 이 필드에 PCI 디바이스 드라이버가 사용할 private 데이터를 커널 메모리에서 할당받아(kmalloc()) 사용할 수 있을 것이다.

```
#ifdef CONFIG_PM
static int eepro100_suspend(struct pci_dev *pdev, u32 state)
{
    struct net_device *dev = pci_get_drvdata(pdev);
    struct speedo_private *sp = (struct speedo_private *)dev->priv;
    long ioaddr = dev->base_addr;

    pci_save_state(pdev, sp->pm_state);

    if (!netif_running(dev))
        return 0;

    netif_device_detach(dev);
    outl(PortPartialReset, ioaddr + SCBPort);

    /* XXX call pci_set_power_state ()? */
    return 0;
}
...
```

코드 552. eepro100_suspend() 함수의 정의

Intel의 Ethernet Express Pro 100은 fast ethernet을 지원하는 네트워크 디바이스 드라이버이다. eepro100_suspend() 함수는 이러한 network device의 특성을 반영해 주어야 한다. 여기서는 특정 레지스터에 대한 접근은 생략하고, 우리가 보고자 하는 power management에 대한 것만을 보도록 하자. pci_save_state() 함수는 PCI configuration space에 있는 것들을 저장하기 위해서 호출한다. 저장되는 곳은 네트워크 드라이버의 private 필드에 해당하는 priv 필드에 있는 pm_state이다. 만약 현재 네트워크 인터페이스가 running상태가 아니라면(if!(netif_running())) 바로 0을 돌려주고 복귀한다. 즉, 현재 이 네트워크 인터페이스가 사용되고 있지 않기 때문에 PM_SUSPEND 요청을 처리할 필요가 없다. 그렇지 않다면, 일단 network 인터페이스로부터 이 디바이스를 분리시키기 위해서 netif_device_detach()를 호출한다. 이것을 마치고 나면, 이젠 진짜 hardware적으로 suspend상태로 가도록 명령을 내려주게된다(outl()). 복귀 값은 0이다.

```
...
static int eepro100_resume(struct pci_dev *pdev)
{
    struct net_device *dev = pci_get_drvdata(pdev);
    struct speedo_private *sp = (struct speedo_private *)dev->priv;
    long ioaddr = dev->base_addr;

    pci_restore_state(pdev, sp->pm_state);

    if (!netif_running(dev))
        return 0;

    outw(SCBMaskAll, ioaddr + SCBCmd);
    speedo_resume(dev);
    netif_device_attach(dev);
    sp->rx_mode = -1;
    sp->flow_ctrl = sp->partner = 0;
    set_rx_mode(dev);
    return 0;
}
#endif /* CONFIG_PM */
```

코드 553. eepro100_resume() 함수의 정의

pci_get_drvdata()는 앞에서 본 pci_driver의 driver_data 필드를 돌려준다. 현재 network 디바이스로 등록되어 있기에 net_device 구조체를 사용한다. 이곳에서 드라이버의 private 자료구조에 접근하기 위해서 priv 필드를 접근하고, 이전 앞에서 저장해 두었던 PCI configuration space에 있는 내용을 복구하도록 한다(pci_restore_state()). 만약 네트워크 인터페이스가 running상태가 아니라면(if(!netif_running()), 앞에서와 마찬가지로 바로 0을 돌려주고 복귀 한다. 이전 다시 hardware를 이전의 상태로 변경하기 위해서 레지스터에 직접 접근해서 값을 써주고(outw()), resume()을 위해서 정의된 speedo_resume() 함수를 호출한다. 이 함수에서도 hardware에 대한 접근이 일어나게 되며, 데이터를 관리하기 위한 Ring 버퍼에 대한 연산이 있게 된다. 이전 디바이스를 다시 네트워크 인터페이스에 연결하기 위해서 netif_device_attach() 함수를 수행한다. 나머지는 디바이스가 다시 동작하도록 만드는 명령들이다. 복귀 값은 0이다.

이번 장에서는 Linux에서의 power management를 다루었다. 하지만, 실제적으로 power management 어떤 순간에 필요한가는 다루지 않았다. 즉, power management를 하기 위해서는 먼저 CPU에 대한 자세한 이해가 병행해야 하기 때문에 이것을 남겨두었다. 각각의 CPU는 나름대로 power management를 해야 할 순간을 포착할 수 있도록 설계되었으며, 이를 디바이스 드라이버에서 지원하기 위해서는 앞에서와 같은 pm_dev 구조체와 같은 것을 만들어야 하며, 이를 pm_register()를 통해서 등록하고, 만약 더 이상 필요치 않다고 생각되면, pm_unregister()로 해제해 주어야 한다. 또한 디바이스는 자신의 power management와 관련된 call back 함수를 가져서, 이 call back 함수내에서 디바이스를 제어해 power의 소비를 억제시켜야 한다는 것이다. 이 정도만 가지고서도 일단 Linux에서 디바이스 드라이버와 같은 것을 programming하는데 필요한 power management는 이해할 수 있으리라고 생각한다. APM이나 APIC와 같은 것은 저자도 아직 자세히 알지 못하기에 나중을 위해서 남겨두도록 하겠다.

9. Advanced Power Management(APM)

Advanced Power Management(APM)은 PC에는 BIOS와 같은 곳에서 시스템의 전원을 관리하기 위해서 제공하는 기능이다. Linux의 커널을 compile하다보면 APM과 관련된 설정을 찾을 수 있으며, 이는 반드시 BIOS를 동반하지 않으면, 사용할 수 없는 기능이다. 물론 조금더 진보된 기능으로 ACPI(Advanced Configuration and Power Interface)가 있기는 하지만, Linux에서는 아직 실험적인 레벨(level)에서 이를 제공하고 있으며, ACPI역시 ACPI와 호환되는 BIOS를 가지고 있어야만 사용가능하다. 일반적으로 Intel architecture를 사용하는 요즘 나오는 PC의 BIOS에는 대부분이 이 기능을 가지고 있기에 사용할 수 있는 기능이다. 하지만, embedded 시스템에서는 i386계열의 CPU를 사용하지 않는 경우가 대부분이기에 APM과 ACPI는 사용할 수 없다.²⁰⁰

Linux는 실제로 커널이 완전히 동작(working)하는 중에는 BIOS의 기능을 사용하지 않는다. 하지만, 예외적으로 PC에서는 커널이 완전히 동작하는 동안에도 APM기능을 사용하기 위해서 BIOS에 접근하고 있다. 이를 위해서 i386 계열의 PC환경에는 APM이 BIOS를 접근하기 위한 환경을 booting하는 과정에서 만들어주고 있으며, APM은 자신이 수행될 순간이 있으면 이러한 BIOS function들을 이용해서 각종 시스템의 전원에 대한 관리를 처리한다. 이미 앞에서 Power Management(PM)을 위해서 커널이 기본적으로 제공하는 함수들에 대해서는 어느정도 보았기에, 이베 여기서부터는 APM의 실제 구현이 어떻게 되어 있는가를 보도록 하겠다. 시스템의 booting과정에서부터 어떻게 APM이 동작하는지를 차례 차례 보도록 하자.²⁰¹

9.1. APM BIOS

APM BIOS에 대해서 개념적인 것을 좀더 살펴보도록 하자. 기본적으로 APM은 다음과 같은 가정을 가진다. 즉, 하나나 그 이상의 software layer로 구성되며, 이러한 software layer로는 APM Driver 및 APM-Aware Application 및 APM-Aware Device Driver등이 있다. 기본적으로 OS에 의존적이지 않은 software layer라고 할 수 있는 APM BIOS가 있다고 가정하며, OS와의 이러한 BIOS를 연결하는 역할을 해주는 것이 APM Driver이다. 각각의 APM-Aware Application과 APM-Aware Driver는 이 APM Driver를 통한 interface를 제공받을 수 있으며, Power Management(PM)와 관련된 event가 발생하면, 자신들이 관리하는 디바이스에 대한 PM을 처리할 수 있게 된다. 따라서, Application layer에서 동작할 수 있는 일종의 Daemon을 가지게 되는데, 이것이 Linux에서 볼 수 있는 APM Daemon(kapmd)이다.

²⁰⁰ 하지만, 유사한 기능을 구현할 수는 있을 것이다. 즉, APM과 같은 역할을 하는 daemon을 생성해서 이를 처리하도록 만들 수 있을 것이다.

²⁰¹ 여기서보는 커널은 2.4.19이다. 될 수 있으면 현재 나와 있는 안정된 버전의 커널을 보도록 앞으로도 계속 노력할 것이다.

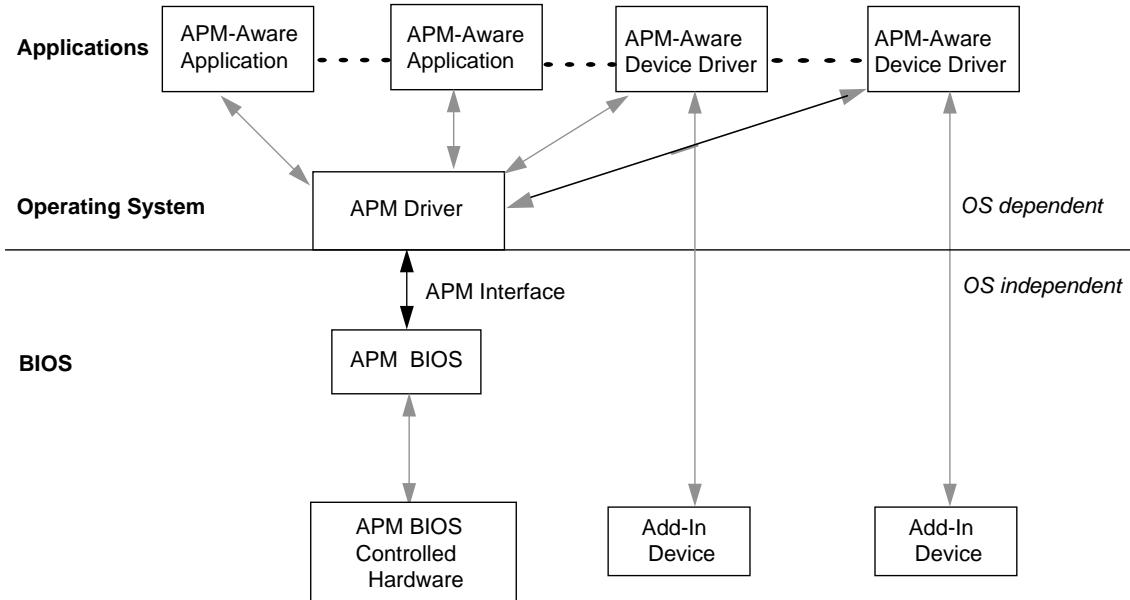


그림 69. APM의 구성 요소

APM은 3가지에 대한 PM state를 정의하고 있다. 각각은 System Power States, Device Power States, CPU Core Power State이다. 물론 Device Power State나 CPU Core Power State는 Device Control 및 CPU Core Control이라고 해도 무방하다. 각각에 대해서 다음과 같은 상태들이 존재한다.

- **System Power States** : 시스템 전체적인 입장에서 보는 Power States이다. 각각은 Full On, APM Enabled, APM Standby, APM Suspend, Off로 구분된다. 이들간의 차이점은 working state로 가는데 있어서 지연시간(Latency)을 가지고 구분된다.
 - **Full On** : 기본 모드로서 시스템이 PM을 하지 않는 상태이다.
 - **APM Enabled** : 시스템은 working하고 있지만, 일부 사용되지 않는 디바이스들은 전원(power)이 들어오지 않은 상태이다.
 - **APM Standby** : 짧은 동안의 비활성(inactivity) 이후에 시스템이 진입하게 되는 상태이다. APM Standby 상태에서 APM Enabled상태로의 진입은 동시에 일어나는 것처럼 여겨진다.
 - **APM Suspend** : 상대적으로 긴 비활성 이후에 시스템이 진입하는 상태이다. 따라서, APM Suspend 상태에서 APM Enabled 상태로 회복하기 위해서는 상대적으로 긴 시간을 요한다.
- **Device Power States(Device Control)** : 디바이스들은 일반적으로 시스템의 BIOS나 OS에 의해서(정확히 말하자면, OS를 지원하는 Device Driver라고 할 수 있다.) 전원이 관리될 수 있다. 즉, APM BIOS나 APM Driver에 의해서 전원이 관리된다. 다음과 같은 상태가 디바이스의 Power State로 볼 수 있다.
 - **Device On** : 디바이스가 완전히 동작하는 상태이다. 따라서, 모든 디바이스의 특성들을 다 사용할 수 있는 상태이다.
 - **Device Power Managed** : 디바이스는 동작하지만, 특정 디바이스의 특성들은 동작하지 않을 수 있거나, 혹은 낮은 성능(Performance)을 보일 수 있다. 전원은 유지되며, 이를 위해서 디바이스의 동작을 결정하는 여러 요소들을 저장할 필요가 있다.
 - **Device Low Power** : 디바이스가 동작하지 않는다. 전원은 유지되며, 이를 위해서 디바이스의 동작을 결정하는 여러 요소들을 저장할 필요가 있다.
 - **Device Off** : 디바이스가 완전히 동작하지 않는다. 즉, 전원이 Off된 상태로서, 디바이스의 동작에 관련된 여러 요소들도 유지되지 않는다.
- **CPU Core Power State(CPU Core Control)** : CPU는 일반적으로 다른 디바이스와는 다르게 관리된다. 즉, CPU는 마지막으로 전원이 Off되어야 하는 디바이스이며, 가장 먼저 전원이 On되어야 하는 디바이스이기 때문이다. 전원 관리의 대상이 될 수 있는 것으로는 CPU clock, cache, system bus, system timer 등을 들 수 있으며, 시스템의 전원 상태(Power State)를 바꾸기 위해서 반드시 필요하다. 아래와 같은 상태들이 존재한다.

- **Full On** : Full Speed로 동작하며, 많은 Power를 소모하고, 가장 높은 레벨의 성능을 보장한다.
- **Slow Clock** : 상대적으로 낮은 speed로 동작하며, Power의 소모를 줄이고, 성능도 이에 맞춰서 낮아진다. Interrupt들에 의해서 Full On 상태로 변경되며, 순간적으로 발생하는 것처럼 보인다.
- **Stop** : 단지 clock을 동작시키는 hard ware interrupt만이 있으며, 역시 Full On상태로의 진입은 순간적인 것처럼 보인다.

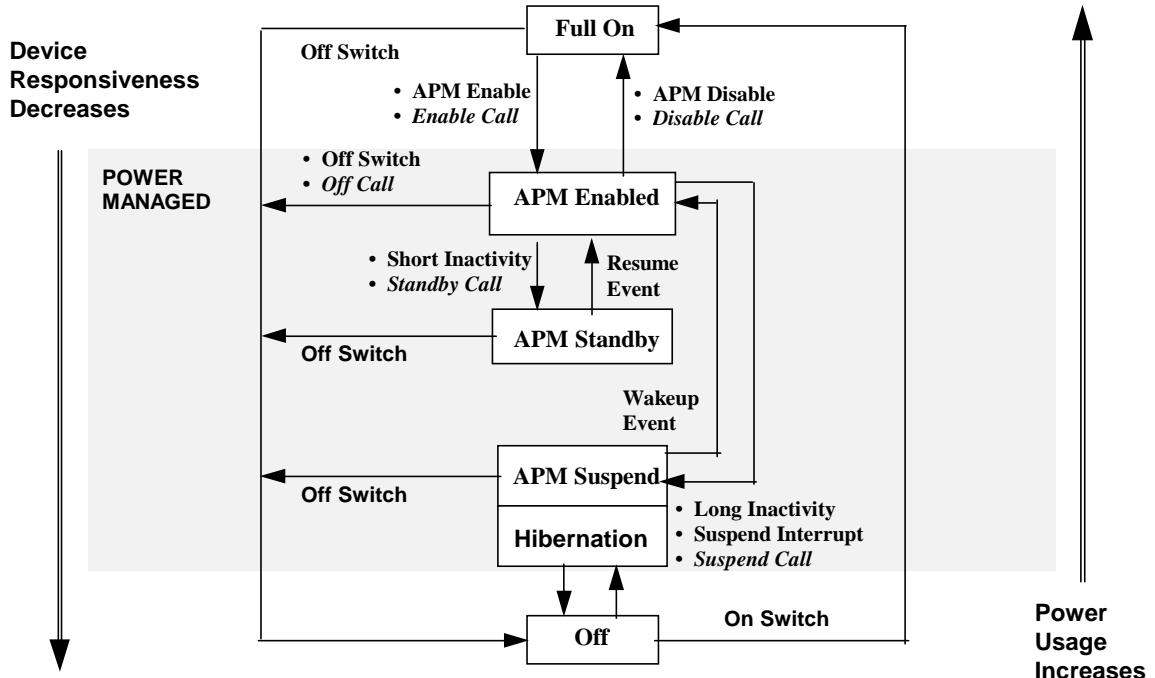


그림 70. System Power State의 천이(Transition)

이중에서 시스템의 Power State에 대한 것을 좀더 자세히 보면, [그림 70]와 같이 볼 수 있다. 여기서 시스템의 상태는 사각형에 나타나 있고, 상태의 천이는 화살표로 표시된다. 또한 화살표 옆에 있는 것들은 상태의 천이를 발생시키는 event들이다. Full On으로 갈 수록 사용되는 Power의 양은 커지게 되며, Off 상태로 갈 수록 시스템의 반응 속도는 낮아지게 된다. 중요한 점은 APM BIOS와 APM Driver가 서로 상호 작용해서 이러한 일이 발생한다는 것이다. 즉, XXX call로 표현된 것이 이러한 APM BIOS와 APM Driver간에 발생하는 interaction이다. 이러한 Call에는 다음과 같은 것들이 있다.

- **Off Switch or Off Call** : 어떤 상태에서도 on/off switch를 Off로 하는 것은 시스템을 Off 상태로 바꾸어준다.
- **On Switch** : 시스템을 On시킨다. 이것은 시스템을 다시 초기화 시키는 과정으로 운영체제의 loading이 필요하다.
- **APM Enable or Enable Call** : 시스템을 Full On 상태에서 APM Enabled 상태로 바꿔준다. 이것은 APM BIOS와 APM Driver간에 connect가 일어난 후나 혹은 소프트웨어가 APM이 enable되어야 한다는 것을 알려준 후에 생기게 된다.
- **APM Disable or Disable Call** : 시스템을 APM Enabled 상태에서 Full On 상태로 바꿔준다.
- **Short Inactivity or Standby Call** : 명시된 시간동안 시스템이 inactive할 경우에 시스템을 APM Enabled 상태에서 APM Standby 상태로 변경하도록 만든다.
- **Long Inactivity or Suspend Interrupt or Suspend Call** : 명시된 긴 시간동안 시스템이 inactive할 경우, APM Enabled 상태에서 APM Suspend 상태로 바꿔준다. 이것은 Suspend button이 눌려지거나, 혹은 battery low alarm 및 lid closure와 같은 interrupt를 받았을 때도 발생될 수 있다.

- **Resume Event** : 마우스를 움직인다던가, keyboard에 입력이 있는 경우, 혹은 어떤 디바이스를 접근(access)하는 경우에 시스템의 상태를 APM Standby 상태에서 APM Enabled 상태로 변경하도록 한다.
- **Wakeup Event** : 특정 인터럽트는(예를 들어 resume button이 놀린다거나, modem에 어떤 input이 있거나, 혹은 real-time clock이 alarm 하는 경우) 시스템의 상태를 APM Standby 상태에서 APM Enabled 상태로 바꿔준다.

PC와 같은 환경에서 동작하는 Linux에서는 기본적으로 다음과 같은 APM을 위한 것을 갖추고 있다. 즉, 앞에서 설명한 APM BIOS는 PC hardware architecture에 따라 존재한다고 가정하고, APM-Aware Application으로 kapmd 프로세스가 동작 중이어야 하며, APM driver module이 커널에 link되어야 있어야 한다.

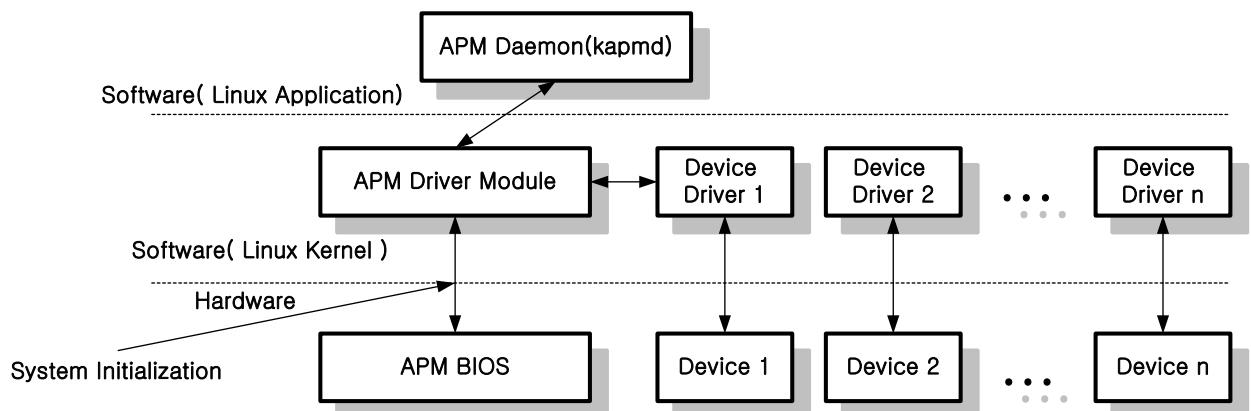


그림 71. Linux의 APM Architecture

[그림 71]는 Linux에서 가지고 있는 PC상의 APM에 대한 구성을 보여주는 것이다. 일반적으로 Linux 커널은 시스템의 초기화 시에 APM을 위한 과정을 가지고 있으며, 또한 관련된 APM Driver는 커널의 모듈형태나 static linking으로 설정되어 포함되게 된다. 또한, 이것을 관리하는 process(APM Daemon)은 실제로는 커널 address space에서 동작하게 되며, APM module에서 정의하고 있는 function을 수행하기 위해 process로 존재하게 된다.

이것으로 대략적인 APM과 APM BIOS와의 관계를 어느정도 이해했다고 보고, 이젠 Linux에서 어떻게 APM을 구현하고 있는지를 알아보기로 하자.

9.2. Booting에서의 APM관련 부분에 대한 분석

먼저 보아야 할 부분은 `~/linux/arch/i386/kernel/head.S` 파일이다. 이 파일에서 GDT(Global Descriptor Table)에 대한 정의에 APM에서 사용하는 code와 data segment에 대한 정의를 아래와 같이 찾을 수 있을 것이다.

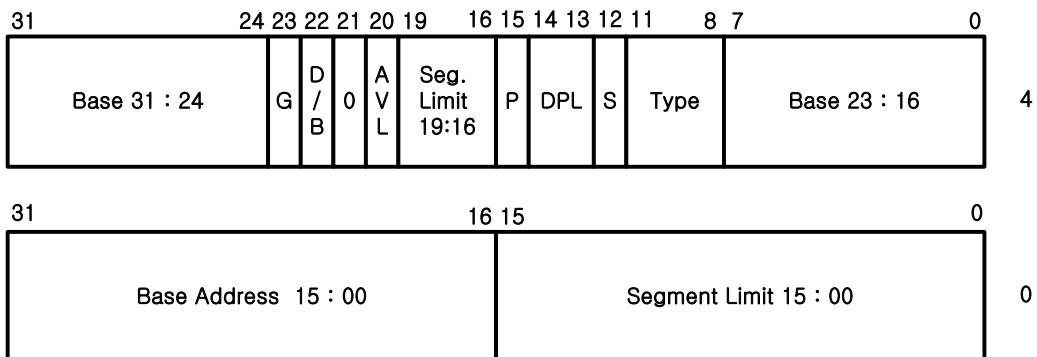
```

ENTRY(gdt_table)
...
/*
 * The APM segments have byte granularity and their bases
 * and limits are set at run time.
 */
.quad 0x0040920000000000 /* 0x40 APM set up for bad BIOS's */
.quad 0x00409a0000000000 /* 0x48 APM CS code */
.quad 0x00009a0000000000 /* 0x50 APM CS 16 code (16 bit) */
.quad 0x0040920000000000 /* 0x58 APM DS data */
...

```

코드 554. Global Descriptor Table의 정의

GDT의 각 필드들에 대한 구성은 아래의 그림과 같이 되어 있다. 즉, GDT의 한 entry는 64 bit으로 이루어져 있으며, 최대 8192개까지의 entry를 가질 수 있다.²⁰² 각각의 entry는 segment의 시작주소와 끝, 그리고 access 권한등의 정보를 표현하는 field로 구성되어 있다.



AVL – Available for use by system software

BASE – Segment base address

D/B – Default operation size (0 = 16-bit segment; 1 = 32-bit segment)

DPL – Descriptor privilege level

G – Granularity

LIMIT – Segment Limit

P – Segment present

S – Descriptor type(0 = system; 1 = code or data)

TYPE – Segment type

그림 72. Segment Descriptor의 필드 정의

[그림 72]은 GDT의 한 entry가 될 수 있는 segment descriptor의 필드를 보여주는 그림이다. 따라서, 위에서 정의한 APM의 code와 data는 모두 base address와 segment limit은 0으로 설정되어 있기에, 나중에 이는 실행시에 다시 설정되어야 할 것이다. 이 필드들을 제외한 다음으로 볼 수 있는 APM을 위한 GDT의 segment descriptor의 정의 0x4092, 0x409A, 또는 0x009A가 됨을 알 수 있다. 즉, G bit으로 이 segment가 granularity가 어떤지를 나타내며, P 및 S bit은 항상 ON되며, type 필드에는 0x0A 값이 오거나 2라는 값이 온다는 것을 볼 수 있다. 여기 G bit은 segment limit이 4-Kbyte의 단위(unit)으로 해석된다는 것을 뜻하기 위해서 설정했으며 설정되어 있지 않으면 segment limit byte 단위로 해석된다. P bit이 설정되어 있으면 현재 segment가 메모리에 항상 존재한다는 것을 나타내고, S bit을 설정하면 system의 segment가 아니라 code나 data로 사용되는 세그먼트라는 것을 나타낸다. Type 필드는 0x02와 0x0A에 대해서 각각, data segment를 가지며 read/write가 가능하다는 것과 code segment를 가지며 execute(실행)과 read가 가능하다는 것을 나타낸다.

BIOS 코드는 Linux가 32 bit protected mode로 진입하기 전에만 호출이 가능하다. 따라서, APM BIOS에 대한 접근에 필요한 작업은 kernel이 booting하는 과정에서 16 bit real-mode로 동작하는 때만 가능하다. ~/linux/arch/i386/boot/setup.S에 아래와 같은 코드를 찾을 수 있을 것이다.

```
...
#endif defined(CONFIG_APM) || defined(CONFIG_APM_MODULE)
# Then check for an APM BIOS...
        # %ds points to the bootsector
        movw    $0, 0x40          # version = 0 means no APM BIOS
        movw    $0x05300, %ax      # APM BIOS installation check
        xorw    %bx, %bx
        int     $0x15
        jc     done_apm_bios      # Nope, no APM BIOS
```

코드 555. setup.S에서의 APM BIOS 초기화

²⁰² 이것은 i386에서 segment selector가 13 bit으로 indexing하기 때문이다.

setup.S에서 이 부분의 코드까지 진행했다면, DS(Data Segment) 레지스터는 boot sector를 가지고 있을 것이라고 가정한다 0x40에 있는 APM BIOS의 버전 번호를 나타내는 값으로 일단 0을 넣어두도록 하자. 그리고나서 AX 레지스터에 0x05300을 넣고, BX레지스터를 0으로 만든다. Interrupt 0x15는 BIOS function으로 AX 레지스터의 AH:AL에 0x05300을 넣고 호출하면, APM이 설치되어 있는지를 확인하는 일을 한다. 또한 BX 레지스터에 0x00을 넣는 것은 시스템 BIOS의 디바이스 ID를 설정하는 것이며, 호출의 결과로 만약 Carry flag이 설정되었다면, 오류가 있었다는 것을 의미한다. 여기서는 오류가 발생했다면, done_apm_bios로 JUMP하도록 했다(jc). 오류가 없다면 AX 레지스터의 AH에는 APM BIOS의 major version 번호가, AL 레지스터에는 minor versoin 번호가 들어갈 것이다. 또한, BX 레지스터에는 “PM”이라는 것을 뜻하는 0x504D라는 값이 있을 것이다. 그리고, CX 레지스터에는 아래와 같은 값들이 있을 것이다.

Bits	Description
0	16-bit protected mode interface가 지원된다.
1	32-bit protected mode interface가 지원된다.
2	CPU idle call이 processor(CPU)의 속도를 저하시킨다.
3	BIOS power management가 불가하다(disabled).
4	BIOS power management에 자유롭다(disengaged) : APM version 1.1에서.
5-7	Reserved

표 53. CX 레지스터의 INT 0x15 BIOS function 호출 결과

이중에서 우린 32-bit interface가 있는지²⁰³를 확인하기 위해서 CX 레지스터의 1번 bit을 보아야 할 것이다. 아래에서 이를 볼 수 있을 것이다.

cmpw	\$0x0504d, %bx	# Check for "PM" signature
jne	done_apm_bios	# No signature, no APM BIOS
andw	\$0x02, %cx	# Is 32 bit supported?
je	done_apm_bios	# No 32-bit, no (good) APM BIOS
movw	\$0x05304, %ax	# Disconnect first just in case
xorw	%bx, %bx	
int	\$0x15	# ignore return code
movw	\$0x05303, %ax	# 32 bit connect
xorl	%ebx, %ebx	
xorw	%cx, %cx	# paranoia :-)
xorw	%dx, %dx	# ...
xorl	%esi, %esi	# ...
xorw	%di, %di	# ...
int	\$0x15	
jc	no_32_apm_bios	# Ack, error.

코드 556. setup.S에서의 APM BIOS 초기화(계속)

이젠 호출의 결과가 옳바른지를 확인하는 일이다. 먼저 BX 레지스터에 0x0504D가 있는지 확인한다. 없다면 done_apm_bios로 JUMP(jne)한다. 이젠 32 bit protected mode에서의 인터페이스가 존재하는지를 알기 위해서 CX 레지스터의 두번째 bit(1)을 본다. 만약 지원하지 않는다면 done_apm_bios로 다시 JUMP한다(je).

혹시 먼저 있을지도 모를 연결(connection)을 끊기 위해서 APM BIOS에 대한 disconnect를 요청하려고 AX 레지스터에 0x05304를 넣고, BX를 0으로 만든 후 INT 0x15를 호출한다. 이때 들려받는 값(return 값)은

²⁰³ 즉, Linux는 완전한 32-bit protected mode에서 i386계열에서는 동작하기 때문에 반드시 이와 같은 interface를 가지고 있어야만 Linux에서 APM을 사용할 수 있다.

사용하지 않는다. 즉, 우린 이미 BIOS에서 APM을 지원한다는 것을 알고 있으며, 단순히 초기화를 위해서 disconnect를 요청한 것일 뿐이기 때문이다. 그리고나서, 32-bit protected mode에서의 APM BIOS연결에 대한 요청을 하기 위해 AX 레지스터에 0x05303을 넣고, EBX, CX, DX, ESI, DI 레지스터를 전부 0으로 만든 후(xor), INT 0x15를 호출한다. 실제로는 INT 0x15의 0x05303은 AX와 BX²⁰⁴레지스터만 사용해서 호출할 수 있지만, 위에서와 같이 다른 여러 레지스터들을 0으로 만든 것은 복귀값이 이 레지스터들을 사용하기 때문이다. 하지만, 실제로는 별로 영향이 없을 것 같다. 만약 호출의 결과로 Carry Flag이 설정되었다면, 오류가 있었다는 말이며, no_32_apm_bios로 JUMP(jc)하도록 한다. 복귀값으로 가지는 값은 아래와 같다.

Register	Description
CF	성공이라면 clear(=0)된다.
AX	Protected-mode 32-bit code segment의 real-mode segment base address를 가진다. 예러시에는 AH 레지스터에 error code값으로 0x02, 0x05, 0x07, 0x08, 0x09등의 값을 가진다.
EBX	Entry point의 offset을 가진다.
CX	Protected-mode 16-bit code segment의 real-mode segment base address를 가진다.
DX	Protected-mode 16-bit data segment의 real-mode segment base address를 가진다.
SI	APM version 1.1에서 APM BIOS의 code segment의 길이를 가진다.
DI	APM version 1.1에서 APM BIOS의 data segment의 길이를 가진다.

표 54. INT 15 BIOS function 0x05303의 호출 결과를 가지는 레지스터들에 대한 정의

즉, BIOS 함수 INT 15의 0x05303을 통해서 앞에서 정해준 APM을 위한 code 및 data segment의 주소와 길이를 결정할 수 있는 것이다.

```

movw %ax, (66)           # BIOS code segment
movl %ebx, (68)           # BIOS entry point offset
movw %cx, (72)           # BIOS 16 bit code segment
movw %dx, (74)           # BIOS data segment
movl %esi, (78)           # BIOS code segment lengths
movw %di, (82)           # BIOS data segment length

# Redo the installation check as the 32 bit connect
# modifies the flags returned on some BIOSs
    movw $0x05300, %ax          # APM BIOS installation check
    xorw %bx, %bx
    xorw %cx, %cx              # paranoia
    int $0x15
    jc  .apm_disconnect        # error -> shouldn't happen

    cmpw $0x0504d, %bx          # check for "PM" signature
    jne .apm_disconnect         # no sig -> shouldn't happen

    movw %ax, (64)              # record the APM BIOS version
    movw %cx, (76)              # and flags
    jmp .done_apm_bios

```

코드 557. setup.S의 APM BIOS 초기화(계속)

이젠 구한 각종 APM BIOS에 대한 정보를 저장해야 할 것이다. 이미 DS 레지스터가 bootsect를 가르키고 있기에 여기에 있는 자료구조의 offset 66부터 시작해서 차례로 레지스터의 크기에 맞게 저장하도록

²⁰⁴ 앞에서와 마찬가지로, 이때 BX는 system BIOS의 device번호를 가진다.

한다. 저장되는 레지스터의 순서는 AX, EBX, CX, DX, ESI, DI 순이다²⁰⁵. 이제 다시 installation check(설치 점검)을 하기 위해서 0x05300을 AX 레지스터에 넣고, BIOS function INT 0x15를 호출한다. 역시 BX에는 system BIOS의 device ID로 0을 주었다. 만약 이때 다시 Carry Flag이 설정된다면, `apm_disconnect`로 JUMP(jc)하도록 한다. 하지만, 이와같은 일은 일어나지 않을 것이라고 생각한다. 앞에서와 마찬가지로 BX에 0x0504D가 있는지 확인하고, 없다면 `apm_disconnect`로 JUMP한다(jne). 이것 역시 일어나지 않을 것이다. 이젠 AX에 넘어온 호출의 결과값인 APM BIOS version 번호를 offset 64에 저장하고, CX에 넘어온 APM BIOS flag값을 offset 76에 저장한다. 이것을 마치면 APM을 32-bit protected mode에서 사용할 수 있도록 하기 위한 BIOS function의 호출을 끝나게된다. `apm_disconnect`에서는 앞에서 설정한 32-bit protected mode APM BIOS support에 대한 connect 요청을 다시 복구하기 위한 부분이다.

```

apm_disconnect:
    movw $0x05304, %ax          # Tidy up
    xorw %bx, %bx               # Disconnect
    int $0x15                   # ignore return code
    jmp done_apm_bios
no_32_apm_bios:
    andw $0xffffd, (76)         # remove 32 bit support bit
done_apm_bios:
#endif
...

```

코드 558. setup.S의 APM BIOS 초기화(계속)

`apm_disconnect`에서는 0x05304를 AX 레지스터에 넣고, BX를 0으로 만든 다음 BIOS function INT 0x15를 호출한다. 즉, 앞에서 보았던 것과 마찬가지로 APM BIOS에 대한 disconnect를 효청하는 것이다. `no_32_apm_bios`에서는 offset 76에 있는 APM BIOS installation check의 결과값으로 주어진 APM BIOS flag값에서 32-bit protected mode interface 지원 bit을 지우는 일을 한다. `done_apm_bios`가 APM BIOS에 대한 초기화 및 설정에 대한 마지막 부분을 가르키는 레이블로 사용되었다.

```

...
/*
 * This is set up by the setup-routine at boot-time
 */
#define PARAM ((unsigned char *)empty_zero_page)
#define SCREEN_INFO (*(struct screen_info *) (PARAM+0))
#define EXT_MEM_K (*(unsigned short *) (PARAM+2))
#define ALT_MEM_K (*(unsigned long *) (PARAM+0x1e0))
#define E820_MAP_NR (*(char*) (PARAM+E820NR))
#define E820_MAP ((struct e820entry *) (PARAM+E820MAP))
#define APM_BIOS_INFO (*(struct apm_bios_info *) (PARAM+0x40))
#define DRIVE_INFO (*(struct drive_info_struct *) (PARAM+0x80))
#define SYS_DESC_TABLE (*(struct sys_desc_table_struct*)(PARAM+0xa0))
#define MOUNT_ROOT_RDONLY (*(unsigned short *) (PARAM+0x1F2))
#define RAMDISK_FLAGS (*(unsigned short *) (PARAM+0x1F8))
#define ORIG_ROOT_DEV (*(unsigned short *) (PARAM+0x1FC))
#define AUX_DEVICE_INFO (*(unsigned char *) (PARAM+0x1FF))
#define LOADER_TYPE (*(unsigned char *) (PARAM+0x210))
#define KERNEL_START (*(unsigned long *) (PARAM+0x214))
#define INITRD_START (*(unsigned long *) (PARAM+0x218))
#define INITRD_SIZE (*(unsigned long *) (PARAM+0x21c))
#define COMMAND_LINE ((char *) (PARAM+2048))
#define COMMAND_LINE_SIZE 256

```

²⁰⁵ 이곳은 setup routine에서 설정되는 커널의 parameter들이 저장되는 공간으로 `empty_zero_page` (=0xC0100000 + x4000)로 시작되는 부분의 offset이다. 나중에 다시 보겠지만, 여기에 APM BIOS에 대한 정보를 저장하기 위해서 64 bytes(0x40)의 공간이 있다.

...

코드 559. ~/linux/arch/i386/kernel/setup.c 파일의 setup-routine에 의해서 설정되는 kernel parameter들

`empty_zero_page`는 여기서 일종의 기본 주소(base address)로 사용하는 기준점이다. 이것(PARAM)을 기준으로 각각의 커널에 필요한 정보(혹은 parameter)들이 전부 offset으로 나타난다. 우리가 현재 관심을 두고 있는 것은 APM BIOS_INFO로서, 이것은 PARAM+0x40(= `empty_zero_page` address + 0x40)에서 시작해서 `amp_bios_info` 구조체를 가진다. 이것은 다시 APM BIOS_INFO라는 것으로 표현한다.

```
struct apm_bios_info { /* Field : Offset */
    unsigned short version; /* APM BIOS의 version : 64*/
    unsigned short cseg; /* APM BIOS의 code segment : 66*/
    unsigned long offset; /* APM BIOS의 entry point offset : 68*/
    unsigned short cseg_16; /* APM BIOS의 16-bit code segment : 72*/
    unsigned short dseg; /* APM BIOS의 data segment : 74*/
    unsigned short flags; /* APM BIOS의 flag : 76*/
    unsigned short cseg_len; /* APM BIOS의 code segment의 길이 : 78*/
    unsigned short cseg_16_len; /* APM BIOS의 16-bit code segment의 길이 : 80*/
    unsigned short dseg_len; /* APM BIOS의 data segment의 길이 : 82*/
};
```

코드 560. apm_bios_info 구조체의 정의

`apm_bios_info` 구조체는 `~/linux/include/linux/apm_bios.h` 파일에 정의되어 있다. 이미 앞에서 APM BIOS의 정보를 저장하면서 관련된 field가 각각의 offset에 맞춰서 저장됨을 보았다. APM BIOS의 정보 구조체인 `amp_bios_info`로 정의된 전역 변수로는 `~/linux/arch/i386/kernel/setup.c`에 있는 `apm_info`가 있으며, 아래와 같은 자료구조를 가진다.

```
struct apm_info {
    struct apm_bios_info bios; /* APM BIOS에 대한 정보 */
    unsigned short connection_version; /* Connection의 versoin */
    int get_power_status_broken; /* Broken getPowerStatus() call 여부 flag */
/*
    int get_power_status_swabinminutes; /* DMI에서 사용 */206
    int allow_ints; /* APM에서 interrupt 허가여부 flag */
    int realmode_power_off; /* Real mode power off 지원여부 flag */
    int disabled; /* APM이 disable되었는가를 나타내는 flag */
};
```

코드 561. apm_info 구조체의 정의

`apm_info` 구조체는 `~/linux/include/linux/apm_bios.h`에 정의되어 있으며, 나중에 APM에서 BIOS에 대한 접근을 하게 될 때 사용될 것이다. `apm_info` 전역 변수의 초기화는 `~/linux/arch/i386/kernel/setup.c`에서 `setup_arch()`²⁰⁷ 함수가 호출될 때 이루어진다. `apm_info` 전역 변수에 앞에서 정의한 APM BIOS_INFO를 그대로 넣어주기만 하면 된다.

²⁰⁶ DMI/Desktop Management Interface)로 주로 laptop과 같은 곳에서 사용한다. APM BIOS에서 minutes(분)을 알려주기 위해서 byte 단위로 swapping을 해야한다는 것을 나타낸다.

²⁰⁷ `setup_arch()` 함수는 `~/linux/init/main.c`에서 `start_kernel()` 함수에서 커널에 lock을 설정한 후, Linux의 banner를 출력하고나서 바로 호출되는 함수이다. 넘겨주는 값은 커널의 command line을 가르키는 `command_line`의 포인터이다.

자, 이젠 APM BIOS를 사용하기 위해서 커널의 booting에서 일어나는 일들에 대해서, 어느정도 이해를 했으리라 생각한다. 먼저 booting의 초기단계에서 APM BIOS에 정보를 얻기위해서 BIOS function INT 0x15를 사용해서 각종 정보를 얻어왔고, 이와 같은 정보를 저장한 후, 이를 `apm_info`라는 전역변수에 넣어주었다. 이후에서 설명하는 것은 `~/linux/arch/i386/kernel/apm.c`에서 위에서 저장한 정보들이 어떻게 사용되는지를 추적해 보도록 하겠다.

9.3. APM Module의 적재와 해제

APM도 커널에 module로 적재되는 것이 가능하다. 따라서, 일반적인 커널 모듈에 대해서 이야기할 때와 마찬가지로 module의 적재와 해제부터 보자.

```
...
module_init(apm_init);
module_exit(apm_exit);
...
static int __init apm_init(void)
{
    struct proc_dir_entry *apm_proc;

    if (apm_info.bios.version == 0) {
        printk(KERN_INFO "apm: BIOS not found.\n");
        return -ENODEV;
    }
    printk(KERN_INFO
           "apm: BIOS version %d.%d Flags 0x%02x (Driver version %s)\n",
           ((apm_info.bios.version >> 8) & 0xff),
           (apm_info.bios.version & 0xff),
           apm_info.bios.flags,
           driver_version);
    if ((apm_info.bios.flags & APM_32_BIT_SUPPORT) == 0) {
        printk(KERN_INFO "apm: no 32 bit BIOS support\n");
        return -ENODEV;
    }
    if (allow_ints)
        apm_info.allow_ints = 1;
    if (broken_psr)
        apm_info.get_power_status_broken = 1;
    if (realmode_power_off)
        apm_info.realmode_power_off = 1;
    /* User can override, but default is to trust DMI */
    if (apm_disabled != -1)
        apm_info.disabled = apm_disabled;
```

코드 562. `apm_init()` 함수의 정의

`apm_init()` 함수는 APM을 초기화하기 위해서 호출되는 함수이다. 이미 앞에서 APM BIOS에 대한 정보들을 다 구한 상황이다. APM에 대한 proc 파일 시스템 엔트리(entry)를 생성하기 위해서 `apm_proc`이라는 변수를 정의한다. 이전에 구한 APM BIOS의 `version`이 올바른 값을 가지는지 확인하고, 만약 0이라는 값을 가진다면 APM을 사용할 수 없으므로 `-ENODEV`를 돌려준다. 그 다음은 APM BIOS에서 얻은 정보를 화면에 보여주는 것이다. APM BIOS의 버전과 flag 설정 및 APM BIOS driver의 버전 정보를 보여준다.²⁰⁸

얻어온 APM BIOS의 정보 중에서 32-bit protected mode에서의 지원이 없다면(`APM_32_BIT_SUPPORT`), Linux에서는 사용할 수 없기에 에러 메시지를 출력한 후 `-ENODEV`를 돌려준다. 이젠 커널 compile 옵션으로 주어진 것들에 대한 것을 처리할 차례이다. 즉, `apm_info` 전역 변수의 필드 중에서

²⁰⁸ 해당 정보를 보기를 원한다면, `/var/log/messages` 파일에서 찾기를 바란다. 당연히

apm_bios_info 필드를 제외한 필드들에 대한 설정을 해준다. APM BIOS call에서도 interrupt가 발생하는 것을 허용한다면(allow_ints : CONFIG_APM_ALLOW_INTS), allow_ints²⁰⁹ 필드를 1로 설정하며, broken_psr이 있는 경우에는 get_power_status_broken을 1로 설정해 준다. broken_psr²¹⁰은 깨어진(broken) getPowerStatus call을 BIOS가 가지고 있다는 뜻인데, APM 모듈의 parameter로 주어질 수 있다. realmode_power_off는 CONFIG_APM_REAL_MODE_POWER_OFF가 커널 컴파일 옵션으로 주어진 경우에 1로 설정되며, apm_disabled가 -1로 설정되지 않았다면, 즉, 모듈의 로딩/loading)에서 parameter로 APM을 disable하도록 설정되었다면, APM을 disable시키기 위해서 1로 설정한다.

```
/*
 * Fix for the Compaq Contura 3/25c which reports BIOS version 0.1
 * but is reportedly a 1.0 BIOS.
 */
if (apm_info.bios.version == 0x001)
    apm_info.bios.version = 0x100;
/* BIOS < 1.2 doesn't set cseg_16_len */
if (apm_info.bios.version < 0x102)
    apm_info.bios.cseg_16_len = 0; /* 64k */
if (debug) {
    printk(KERN_INFO "apm: entry %x:%lx cseg16 %x dseg %x",
           apm_info.bios.cseg, apm_info.bios.offset,
           apm_info.bios.cseg_16, apm_info.bios.dseg);
    if (apm_info.bios.version > 0x100)
        printk(" cseg len %x, dseg len %x",
               apm_info.bios.cseg_len,
               apm_info.bios.dseg_len);
    if (apm_info.bios.version > 0x101)
        printk(" cseg16 len %x", apm_info.bios.cseg_16_len);
    printk("\n");
}
if (apm_info.disabled) {
    printk(KERN_NOTICE "apm: disabled on user request.\n");
    return -ENODEV;
}
if ((smp_num_cpus > 1) && !power_off) {
    printk(KERN_NOTICE "apm: disabled - APM is not SMP safe.\n");
    return -ENODEV;
}
if (PM_IS_ACTIVE()) {
    printk(KERN_NOTICE "apm: overridden by ACPI.\n");
    return -ENODEV;
}
pm_active = 1;
```

코드 563. apm_init() 함수의 정의(계속)

Compaq과 같은 곳에서 APM BIOS의 버전을 reporting하는 것이 오류가 있을 수 있기에 여기서 다시 보정한다. 0x001을 가지고 있다면 0x100으로 조정한다(APM BIOS version 1.0을 따른다). 또한 APM BIOS 버전이 1.2보다 작다면²¹¹, 16-bit code segment 길이(length)를 나타내는 bit을 않기에 이를 반영하기 위해서

²⁰⁹ apm_ints는 모듈의 parameter로 주어질 수 있다. 이와같이 모듈의 parameter로 주어질 수 있는 것으로는 debugging을 목적으로 하는 debug, power_off, bounce_interval, broken_psr, realmode_power_off, idle_threshold, idle_period등이 있다. apm.c 파일의 마지막 부분에 정의되어 있으므로, 설명(description)과 같이 한번 보도록 하자.

²¹⁰ 정확히 어떤 목적으로 사용되는 가는 아직 완전히 파악하지 못했지만, APM에서 32-bit에 대한 getPowerStatus() 함수의 지원이 미흡하다는 뜻으로 해석할 수 있을 것이다.

²¹¹ 현재(2002년 10월 2일) APM Spec의 버전은 1.2가 가장 최신이다.

apm_info의 bios.csseg_16_len에 0을 넣는다. debug이 설정되어 있다면, 여기서 debugging 정보를 표시하게 될 것이다. 만약 disabled필드가 설정된 경우에는 사용자가 어떤 형식으로든 APM을 disable시켰다는 이야기가 되므로, -ENODEV를 돌려준다. 뒤에서 APM의 setup 함수로 사용되는 `apm_setup()`이라는 곳에서 command line을 parsing할 수 있기에 kernel command-line argument로 APM을 disable시켜줄 수도 있다. APM은 SMP(Symmetry Multi-Processor)는 지원하지 않으므로, 만약 시스템의 CPU 개수가 1보다 크거나, CONFIG_SMP(power_off를 선언만 해준다.)가 커널 컴파일 옵션으로 지정되어 power_off에 0이 설정된 경우에는 APM을 disable시켜야 하기에 -ENODEV를 돌려준다. power_off는 power_off event가 발생할 때 power_off를 해줄 handler를 설정하는 곳에 사용된다. PM_IS_ACTIVE()는 이미 power management가 활성화 되어있는가를 보는 것인데, 이때 또 다른 PM의 일종인 ACPI²¹² 가 있는지를 확인하는 것이다. 만약 있다면, -ENODEV를 돌려줘서 시스템에 혼동이 없도록 만든다. PM_IS_AVTIVE()는 `~/linux/include/linux/pm.h`에 정의된 매크로로, 단순히 pm_active²¹³라는 변수의 현재 값을 돌려주는 역할을 한다. 여기까지 해서 아무런 문제가 없었다면, PM이 활성화 중이라는 뜻으로 pm_active를 1로 설정한다.

```
/*
 * Set up a segment that references the real mode segment 0x40
 * that extends up to the end of page zero (that we have reserved).
 * This is for buggy BIOS's that refer to (real mode) segment 0x40
 * even though they are called in protected mode.
 */
set_base(gdt[APM_40 >> 3],
         __va((unsigned long)0x40 << 4));
_set_limit((char *)&gdt[APM_40 >> 3], 4095 - (0x40 << 4));
apm_bios_entry.offset = apm_info.bios.offset;
apm_bios_entry.segment = APM_CS;
set_base(gdt[APM_CS >> 3],
         __va((unsigned long)apm_info.bios.cseg << 4));
set_base(gdt[APM_CS_16 >> 3],
         __va((unsigned long)apm_info.bios.cseg_16 << 4));
set_base(gdt[APM_DS >> 3],
         __va((unsigned long)apm_info.bios.dseg << 4));
#endif APM_RELAX_SEGMENTS
    if (apm_info.bios.version == 0x100) {
#endif
        /* For ASUS motherboard, Award BIOS rev 110 (and others?) */
        _set_limit((char *)&gdt[APM_CS >> 3], 64 * 1024 - 1);
        /* For some unknown machine. */
        _set_limit((char *)&gdt[APM_CS_16 >> 3], 64 * 1024 - 1);
        /* For the DEC Hinote Ultra CT475 (and others?) */
        _set_limit((char *)&gdt[APM_DS >> 3], 64 * 1024 - 1);
#endif APM_RELAX_SEGMENTS
    } else {
        _set_limit((char *)&gdt[APM_CS >> 3],
                   (apm_info.bios.cseg_len - 1) & 0xffff);
        _set_limit((char *)&gdt[APM_CS_16 >> 3],
                   (apm_info.bios.cseg_16_len - 1) & 0xffff);
        _set_limit((char *)&gdt[APM_DS >> 3],
                   (apm_info.bios.dseg_len - 1) & 0xffff);
    }
#endif
```

²¹² ACPI(Advanced Configuration and Power Interface)는 현재 Intel 및 Microsoft에서 정한 Spec.으로 버전 2.0까지 존재한다. Linux에서는 커널 컴파일 옵션중에서 experimental code를 enable시켜서 compile할 경우에만 보이게 된다.

²¹³ pm_active는 현재 Power Management 활성화가 이미 이루어 졌는지를 알 수 있도록 만들어주는 flag이다. 정의는 `~/linux/kernel/pm.c`에 있다.

코드 564. `apm_init()` 함수의 정의(계속)

이전 BIOS에서 얻은 정보로 GDT(Global Descriptor Table)에 있는 APM의 Code와 Data segment descriptor의 entry를 설정해 줄 차례이다. 이를 위해서 `_set_base()`와 `_set_limit()`라는 매크로가 사용된다. 이 매크로의 정의는 `~/linux/include/asm-i386/system.h`에 아래와 같이 나와 있다.

```
...
#define _set_base(addr,base) do { unsigned long __pr; \
__asm__ __volatile__ ("movw %%dx,%1\n\t" \
    "rorl $16,%%edx\n\t" \
    "movb %%dl,%2\n\t" \
    "movb %%dh,%3" \
    :"=d" (__pr) \
    :"m" (*(addr)+2)), \
    "m" (*(addr)+4)), \
    "m" (*(addr)+7)), \
    "0" (base) \
); } while(0)

#define _set_limit(addr,limit) do { unsigned long __lr; \
__asm__ __volatile__ ("movw %%dx,%1\n\t" \
    "rorl $16,%%edx\n\t" \
    "movb %2,%%dh\n\t" \
    "andb $0xf0,%%dh\n\t" \
    "orb %%dh,%%dl\n\t" \
    "movb %%dl,%2" \
    :"=d" (__lr) \
    :"m" (*(addr)), \
    "m" (*(addr)+6)), \
    "0" (limit) \
); } while(0)

#define set_base(ldt,base) _set_base( ((char *)&(ldt)) , (base) )
#define set_limit(ldt,limit) _set_limit( ((char *)&(ldt)) , ((limit)-1)>>12 )
...
```

코드 565. GDT 및 LDT table의 Segment Descriptor의 Base Address와 Length를 설정하는 Macro

`set_base()` 및 `set_limit()`(혹은 `_set_base()` `_set_limit()`)매크로는 `addr`로 넘겨받은 기억장소(memory)에 해당 값(base or limit)를 넣는 일을 수행하는 함수이다. 여기서 `set_limit()`는 `_set_limit()`와는 달리 `limit`를 4096 bytes의 한 페이지 단위로 정렬해서 한계치(limit)를 설정하기 위해서 사용된다.

```
#define APM_40          0x40
#define APM_CS           (APM_40 + 8)
#define APM_CS_16         (APM_CS + 8)
#define APM_DS            (APM_CS_16 + 8)
```

코드 566. APM Segment Descriptor에 관련된 상수값 정의

APM_40은 0x40(= 64)으로 GDT에서 APM의 entry를 나타내는 기준값이며, 하나의 GDT 엔트리가 8 bytes씩 차지하므로, APM_CS, APM_CS_16, APM_DS가 차례로 8씩 증가된 값으로 설정되어 있다. 따라서, 0x40이라는 segment의 virtual address(= 0xC0000000 + 0x400)에 잘못된 APM BIOS code segment 설정을 가질 수 있는 설정을 먼저 설정해 주고, 나머지 APM BIOS의 GDT entry에는 앞에서 구한 정보들로 채워주도록 한다. 나머지 부분은 각 BIOS vendor에 따라, 조금씩 달라지는 부분을 GDT의 APM관련 segment의 limit에 반영하기 위한 것이다.

```
static struct {
```

```
unsigned long      offset;
unsigned short    segment;
} apm_bios_entry;
```

코드 567. apm_bios_entry 변수의 정의

apm_bios_entry 변수는 APM BIOS의 entry에 대한 offset과 code segment의 주소를 저장하기 위해서 사용한다. 각각 앞에서 구한 apm_info 전역 변수의 BIOS관련 정보를 넣도록 한다. 이 값은 나중에 APM 모듈에서 APM BIOS의 함수를 호출할 때 사용될 것이므로 기억하도록 하자.

```
apm_proc = create_proc_info_entry("apm", 0, NULL, apm_get_info);
if (apm_proc)
    SET_MODULE_OWNER(apm_proc);
kernel_thread(apm, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD);
if (smp_num_cpus > 1) {
    printk(KERN_NOTICE
        "apm: disabled - APM is not SMP safe (power off active).\n");
    return 0;
}
misc_register(&apm_device);
if (HZ != 100)
    idle_period = (idle_period * HZ) / 100;
if (idle_threshold < 100) {
    original_pm_idle = pm_idle;
    pm_idle = apm_cpu_idle;
    set_pm_idle = 1;
}
return 0;
}
```

코드 568. apm_init() 함수의 정의(계속)

이전 하드웨어적으로 준비해야하는 과정은 다 마쳤기에 소트웨어적으로 커널에서 APM을 관리하고, application과의 interface를 만들어주기 위한 절차로 들어간다. proc 파일 시스템에 대한 entry를 생성하기 위해서 create_proc_info_entry()를 호출해서, 관련 entry와 callback함수를 알려준다. apm_get_info는 나중에 /proc/apm을 읽을 때 사용되는 함수가 될 것이다. 제대로 생성되었다면, 이전 SET_MODULE_OWNER() 매크로를 사용해서 현재 module의 소유자(owner)를 가르켜주도록 한다.

여기서 kernel_thread() 함수를 호출해서 APM을 위한 kernel level의 daemon 프로세스를 생성한다. 따라서, “ps -aux”와 같은 명령으로 “kapmd”라는 커널 daemon이 있다는 것을 확인할 수 있을 것이다. 만약 smp_num_CPU가 1보다 크다면, APM을 잘 못 사용하고 있는 것이므로, 경고(Notice) 메시지를 뿌려주고, 더이상 진행하지 않고, 0을 돌려준다. misc_register()는 Miscellaneous Device를 등록하게 되는데, 이때 등록되는 APM 디바이스 드라이버는 major 번호로 10번²¹⁴을 부여받으며, minor번호로는 134를 사용한다. 따라서, /dev/apm_bios가 major 번호 10에 minor 번호 134를 가진다는 것을 알 수 있다.

HZ²¹⁵는 PC의 경우에는 100이란 값으로 주어지지만, PC가 아닐 경우에는 이야기가 다르다. 따라서, idle_period(IDLE task가 깨어나게된느 간격)은 idle_period x HZ를 한 후, 이를 100으로 나누어준 값을 가지도록 만든다. idle_threshold가 100이하의 값을 가진다면, original_pm_idle에는 현재 설정된 idle function인 pm_idle을 넣고, pm_idle에 apm_cpu_idle을 설정하도록 한다. set_pm_idle은 pm_idle에 apm_cpu_idle이 설정되어 있음을 나타내는 flag으로 사용한다.

```
...
/*
 * idle percentage above which bios idle calls are done
 */
```

²¹⁴ Miscellaneous Device Class에 속한다고 등록되므로, MISC device들이 사용하는 major번호를 쓸 것이다.

²¹⁵ 초당 시스템 timer interrupt가 발생하는 회수이다. Jiffies값을 증가시키는데 사용된다.

```
#ifdef CONFIG_APM_CPU_IDLE
#define DEFAULT_IDLE_THRESHOLD      95
#else
#define DEFAULT_IDLE_THRESHOLD      100
#endif
#define DEFAULT_IDLE_PERIOD    (100 / 3)
...
```

코드 569. IDLE과 관련된 상수값의 정의

커널의 compile 옵션으로 주어진 CONFIG_APM_CPU_IDLE이 있으면, DEFAULT_IDLE_THRESHOLD는 95란 값을 가지며, 그렇지 않다면 100을 가지게 될 것이다. 이 값과 DEFAULT_IDLE_PERIOD는 idle_threshold와 idle_period를 설정하는 기본(default) 값으로 사용된다. 물론 둘다 모듈의 loading시나 커널의 command-line으로 주어지는 값으로 overriding될 수 있다. DEFAULT_IDLE_PERIOD로 유추할 수 있겠지만, 어쨌든. 대략 초당 3번정도의 IDLE주기를 가진다고 볼 수 있다. 또한 100이하의 IDLE threshold값을 가진다면, 그냥 APM에서 제공하는 IDLE 함수를 수행하도록 해주고 있다.

```
static void __exit apm_exit(void)
{
    int      error;

    if (set_pm_idle)
        pm_idle = original_pm_idle;
    if (((apm_info.bios.flags & APM_BIOS_DISENGAGED) == 0)
        && (apm_info.connection_version > 0x0100)) {
        error = apm_engage_power_management(APM_DEVICE_ALL, 0);
        if (error)
            apm_error("disengage power management", error);
    }
    misc_deregister(&apm_device);
    remove_proc_entry("apm", NULL);
    unregister_sysrq_key('o', &sysrq_poweroff_op);
    if (power_off)
        pm_power_off = NULL;
    exit_kapmd = 1;
    while (kapmd_running)
        schedule();
    pm_active = 0;
}
```

코드 570. apm_exit() 함수의 정의

apm_exit() 함수는 apm_init()함수에서 시스템에 가했던 설정을 원래의 상태로 되돌리면 된다. BIOS에 대한 information까지는 없앨 필요가 없기에 관련된 일은 없다. set_pm_idle이 설정되어 있다면, APM에서 제공하는 IDLE function을 사용하고 있다는 말이기에, 원래의 IDLE function으로 되돌려준다 (original_pm_idle). APM BIOS의 flag 정보 중에서 APM_BIOS_DISENGAGED가 설정되지 않았고, APM의 connection version이 0x0100보다 크다면, apm_engage_power_management() 함수에 APM_DEVICE_ALL과 0을 주어서 호출한다. 만약 에러가 있었다면, 에러 메시지를 출력한다(apm_error()). 이전에 등록된 miscellaneous 디바이스를 해제하기 위해서 misc_deregister() 함수를 호출하고, proc 파일 시스템 엔트리를 제거하기 위해서 remove_proc_entry() 함수를 호출한다. 또한 시스템 요청 키(system request key) 중에서 전원차단(power off) 기능을 가지는 함수에 대한 등록을 해제하기 위해서 unregister_sysrq_key()²¹⁶를 호출한다. 넘겨주는 함수는 sysrq_poweroff_op이다. 만약 power_off핸들러가

²¹⁶ OI 함수는 ~/linux/include/linux/sysrq.h에 inline으로 정의된 함수로 CONFIG_MAGIC_SYSRQ라는 커널 compile 옵션이 있을 경우에만 사용된다. 정의되어 있지 않다면, -EINVAL을 호출의 결과 값으로 돌려줄

설정되어 있다면 pm_power_off에는 NULL을 넣어주고, exit_kapmd에 1을 넣어서 kernel APM daemon의 수행을 중단 시키도록 만든다. kapmd_running flag은 kernel APM daemon이 동작(running) 중이라는 것을 가르키며, 이 값이 있는 동안 schedule()를 호출해서 kernel APM daemon이 동작을 중지하도록 만든다. pm_active는 이제 0을 가질 것이다.

```
/***
 *      @apm_engage_power_management - enable PM on a device
 *      @device: identity of device
 *      @enable: on/off
 *
 *      Activate or deactivate power management on either a specific device
 *      or the entire system (%APM_DEVICE_ALL).
 */
static int apm_engage_power_management(u_short device, int enable)
{
    u32      eax;

    if ((enable == 0) && (device == APM_DEVICE_ALL)
        && (apm_info.bios.flags & APM_BIOS_DISABLED))
        return APM_DISABLED;
    if (apm_bios_call_simple(APM_FUNC_ENGAGE_PM, device, enable, &eax))
        return (eax >> 8) & 0xff;
    if (device == APM_DEVICE_ALL) {
        if (enable)
            apm_info.bios.flags &= ~APM_BIOS_DISENGAGED;
        else
            apm_info.bios.flags |= APM_BIOS_DISENGAGED;
    }
    return APM_SUCCESS;
}
```

코드 571. apm_engage_power_mangement() 함수의 정의

apm_engage_power_management() 함수는 특정 디바이스에 대한 PM 기능을 enable 혹은 disable 시키기 위해서 호출된다. 즉, 넘겨받는 argument에는 특정 디바이스를 나타내는 device와 enable/disable를 나타내는 enable이라는 flag이 있다. enable이 0이고, device가 APM_DEVICE_ALL로 되어있다면, BIOS의 flag값에 APM_BIOS_DISABLED이 설정되어 있는지를 확인해서 설정되어 있다면, APM_DISABLED를 돌려준다. 즉, 모든 device에 대해서 disable되었다는 말이다. 그렇지 않다면, apm_bios_call_simple() 함수에 APM_FUNC_ENGAGE_PM와 넘겨받은 device 및 enable을 주고 호출한다. 이때 추가적으로 EAX register의 값을 저장할 eax의 포인터도 같이 넘겨주도록 한다. 이때 복귀 값이 0이 아니라면, eax의 값을 8 bit 오른쪽으로 SHIFT해서 0xFF와 AND한 값을 retun 값으로 돌려주게 된다. 만약 device에 APM_DEVICE_ALL이 들어있다면, enable의 값에 따라 BIOS의 flag값에 APM_BIOS_DISENGAGED bit을 1이면 clear하고 0이면 set한다. 여기까지 아무런 에러가 없었다면 APM_SUCCESS를 돌려준다.

여기서 잠시 APM 모듈에서 사용하는 상수 값들에 대해서 조금 보도록 하자. Flag bit을 check하는 것과 에러 코드들, 그리고, APM BIOS가 관리하는 디바이스의 ID들이다. 정의는 ~/linux/include/linux/apm_bios.h에 있다.

```
/* Results of APM Installation Check */
#define APM_16_BIT_SUPPORT          0x0001
#define APM_32_BIT_SUPPORT          0x0002
```

것이다. 이와 같은 시스템 요청 키들은 특정 키의 조합에 대한 입력에 반응해서 시스템의 특정 연산(혹은 일)을 수행하도록 만들어주기 위해서 사용된다. Key의 입력에 반응하는 핸들러를 가지고 있어서 특정 키가 입력되면, 이 핸들러 함수를 수행하도록 만든다.

```
#define APM_IDLE_SLOWS_CLOCK 0x0004
#define APM_BIOS_DISABLED      0x0008
#define APM_BIOS_DISENGAGED    0x0010
```

코드 572. APM Installation Check에서의 결과 값

APM installation check에서 이미 APM BIOS의 flag값은 보았다. 이 값들은 APM 모듈에서 설정값을 검사할 때 사용된다.

```
/*
 * Error codes
 */
#define APM_SUCCESS           0x00
#define APM_DISABLED          0x01
#define APM_CONNECTED         0x02
#define APM_NOT_CONNECTED     0x03
#define APM_16_CONNECTED      0x05
#define APM_16_UNSUPPORTED    0x06
#define APM_32_CONNECTED      0x07
#define APM_32_UNSUPPORTED    0x08
#define APM_BAD_DEVICE        0x09
#define APM_BAD_PARAM          0x0a
#define APM_NOT_ENGAGED       0x0b
#define APM_BAD_FUNCTION      0x0c
#define APM_RESUME_DISABLED   0x0d
#define APM_NO_ERROR          0x53
#define APM_BAD_STATE          0x60
#define APM_NO_EVENTS          0x80
#define APM_NOT_PRESENT        0x86
```

코드 573. APM Module의 error code 정의

APM 모듈의 error 코드는 `apm_error()`과 같은 함수에서 에러값을 돌려주기 위해서 사용한다. `apm_error()` 함수는 아래와 같이 정의되어 있다.

```
static void apm_error(char *str, int err)
{
    int i;

    for (i = 0; i < ERROR_COUNT; i++)
        if (error_table[i].key == err) break;
    if (i < ERROR_COUNT)
        printk(KERN_NOTICE "apm: %s: %s\n", str, error_table[i].msg);
    else
        printk(KERN_NOTICE "apm: %s: unknown error code %#2.2x\n",
               str, err);
}
```

코드 574. `apm_error()` 함수의 정의

`apm_error()`은 모든 APM 모듈에서 발생하는 에러에 대한 메시지를 출력하는 것을 책임진다. 모든 에러들은 `lookup_t`이라는 구조체의 `table`의 형태로 저장되어 있으며 아래와 같은 정의를 가진다.

```
typedef struct lookup_t {
    int key;
    char * msg;
} lookup_t;
/*
```

```

/*
 * The BIOS returns a set of standard error codes in AX when the
 * carry flag is set.
 */
static const lookup_t error_table[] = {
/* N/A { APM_SUCCESS,           "Operation succeeded" },
{ APM_DISABLED,            "Power management disabled" },
{ APM_CONNECTED,           "Real mode interface already connected" },
{ APM_NOT_CONNECTED,       "Interface not connected" },
{ APM_16_CONNECTED,        "16 bit interface already connected" },
/* N/A { APM_16_UNSUPPORTED,   "16 bit interface not supported" },
{ APM_32_CONNECTED,        "32 bit interface already connected" },
{ APM_32_UNSUPPORTED,      "32 bit interface not supported" },
{ APM_BAD_DEVICE,          "Unrecognized device ID" },
{ APM_BAD_PARAM,           "Parameter out of range" },
{ APM_NOT_ENGAGED,         "Interface not engaged" },
{ APM_BAD_FUNCTION,        "Function not supported" },
{ APM_RESUME_DISABLED,     "Resume timer disabled" },
{ APM_BAD_STATE,           "Unable to enter requested state" },
/* N/A { APM_NO_EVENTS,        "No events pending" },
{ APM_NO_ERROR,             "BIOS did not set a return code" },
{ APM_NOT_PRESENT,          "No APM present" }
};

#define ERROR_COUNT  (sizeof(error_table)/sizeof(lookup_t))

```

코드 575. error_table[] 배열의 정의

즉, `apm_error`은 넘겨받은 에러 값을 가지고 해당 테이블의 엔트리를 찾아서 어떤 에러인가를 화면(혹은 `/var/log/messages`)에 나타내주는 일을 한다.

```

/*
 * APM Device IDs
 */
#define APM_DEVICE_BIOS           0x0000
#define APM_DEVICE_ALL            0x0001
#define APM_DEVICE_DISPLAY        0x0100
#define APM_DEVICE_STORAGE        0x0200
#define APM_DEVICE_PARALLEL       0x0300
#define APM_DEVICE_SERIAL         0x0400
#define APM_DEVICE_NETWORK        0x0500
#define APM_DEVICE_PCMCIA         0x0600
#define APM_DEVICE_BATTERY        0x8000
#define APM_DEVICE_OEM             0xe000
#define APM_DEVICE_OLD_ALL        0xffff
#define APM_DEVICE_CLASS          0x00ff
#define APM_DEVICE_MASK           0xff00

```

코드 576. APM BIOS에서 사용하는 Device ID 정의

APM 모듈에서 사용하는 디바이스의 ID는 `APM_DEVICE_XXX`라는 것으로 표현된다. APM에서 전원 관리의 대상이 되는 디바이스들은, APM BIOS 자체를 나타내는 `APM_DEVICE_BIOS`와 기타 display, storage, parallel, serial, network, pcmcia, battery 등이 있으며, 모든 디바이스를 나타내는데 사용하는 `APM_DEVICE_ALL`과 예전 APM 모듈과의 호환을 위해서 존재하는 `APM_DEVICE_OLD_ALL` 등이 있으며, 디바이스 ID의 mask 값으로 사용하는 `APM_DEVICE_MASK`가 있다. 또한 디바이스 class의 mask 값으로 사용하는 `APM_DEVICE_CLASS`와 기타등등의 OEM으로 제공되는 디바이스들에 대해서 사용할 수 있는 `APM_DEVICE_OEM`이 있다.

```
static u8 apm_bios_call_simple(u32 func, u32 ebx_in, u32 ecx_in, u32 *eax)
```

```

{
    u8                  error;
    APM_DECL_SEGS
    unsigned long      flags;

    __save_flags(flags);
    APM_DO_CLI;
    APM_DO_SAVE_SEGS;
    {
        int      cx, dx, si;

        /*
         * N.B. We do NOT need a cld after the BIOS call
         * because we always save and restore the flags.
         */
        __asm__ __volatile__(APM_DO_ZERO_SEGS
            "pushl %%edi\n\t"
            "pushl %%ebp\n\t"
            "lcall %%cs:" SYMBOL_NAME_STR(apm_bios_entry) "\n\t"
            "setc %%bl\n\t"
            "popl %%ebp\n\t"
            "popl %%edi\n\t"
            APM_DO_POP_SEGS
            : "=a" (*eax), "=b" (error), "=c" (cx), "=d" (dx),
              "=S" (si)
            : "a" (func), "b" (ebx_in), "c" (ecx_in)
            : "memory", "cc");
    }
    APM_DO_RESTORE_SEGS;
    __restore_flags(flags);
    return error;
}

```

코드 577. `apm_bios_call_simple()` 함수의 정의

여기서 간단히 APM BIOS에 대한 호출 과정을 알기위해서 `apm_bios_call_simple()`함수를 보도록 하자. 아마 APM 모듈이 APM BIOS와 어떻게 interface를 하는지를 알 수 있는 가장 핵심적인 함수일 것이다. 넘겨받은 인자(argument)값으로 APM BIOS function의 값과 EBX 및 ECX 레지스터에 전달될 값, 그리고 연산의 결과로 전달 받을 EAX 값을 저장할 포인터가 있다. APM_DECL_SEGS 및 APM_DO_SERO_SEGS, APM_DO_POP_SEGS, APM_DO_RESTORE_SEGS등의 매크로는 APM_ZERO_SEGS라는 값이 정의된 경우에만 사용한다. APM_ZERO_SEGS는 모든 data와 관련된 세그먼트를 APM 모듈에서 BIOS 함수를 호출할때 바꾸지 않고 사용하도록 만들기 위한 것이다. 즉, BIOS call의 내부에서 이전에 데이터 관련 세그먼트 레지스터를 저장해 놓고, 이 레지스터들을 0으로 만든 후, BIOS function를 호출한다. 나중에 다시 저장된 세그먼트 레지스터들을 복귀 전에 반드시 복구 해주어야 할 것이다. APM_DO_CLI는 BIOS function의 호출에서 인터럽트의 허용여부를 알려주는 `apm_info` 구조체의 `allow_ints` 필드가 정의된 경우에는 `_sti()`²¹⁷와 같은 것으로 인터럽트를 허가하도록 만들어주고, 그렇지 못할 때는 `_cli()`²¹⁸를 호출해서 인터럽트가 발생되지 못하도록 만든다.

```

#define savesegment(seg, where) \
    __asm__ __volatile__("movl %%#seg ,%0" :"=m" (where))
...
#define APM_DO_CLI      \
    if (apm_info.allow_ints) \

```

²¹⁷ Set Interrupt로 인터럽트를 허가한다.

²¹⁸ Clear Interrupt로 인터럽트를 불가하게 만든다.

```

        __sti(); \
    else \
        __cli();

#endif APM_ZERO_SEGS
# define APM_DECL_SEGS \
    unsigned int saved_fs; unsigned int saved_gs;
# define APM_DO_SAVE_SEGS \
    savesegment(fs, saved_fs); savesegment(gs, saved_gs)
# define APM_DO_ZERO_SEGS \
    "pushl %%ds\n\t" \
    "pushl %%es\n\t" \
    "xorl %%edx, %%edx\n\t" \
    "mov %%dx, %%ds\n\t" \
    "mov %%dx, %%es\n\t" \
    "mov %%dx, %%fs\n\t" \
    "mov %%dx, %%gs\n\t"
# define APM_DO_POP_SEGS \
    "popl %%es\n\t" \
    "popl %%ds\n\t"
# define APM_DO_RESTORE_SEGS \
    loadsegment(fs, saved_fs); loadsegment(gs, saved_gs)
#else
# define APM_DECL_SEGS
# define APM_DO_SAVE_SEGS
# define APM_DO_ZERO_SEGS
# define APM_DO_POP_SEGS
# define APM_DO_RESTORE_SEGS
#endif

```

코드 578. 인터럽트 및 세그먼트 레지스터 관련 매크로의 정의

`savesegment()` 매크로는 해당 세그먼트 레지스터를 특정 변수에 저장하기 위해서 사용하는 매크로이다. `APM_DO_CLI()`는 앞에서 이야기 했듯이, `apm_info.allow_ints` 값에 따라 `__sti()`나 `__cli()`를 호출해서 인터럽트의 가부를 결정한다. `APM_ZERO_SEGS`가 정의된 경우에는 `APM_DECL_SEGS`가 FS, GS 세그먼트 레지스터의 저장 변수를 선언하게 되며, `APM_DO_SAVE_SEGS`는 FS 및 GS 세그먼트를 저장한다. `APM_DO_ZERO_SEGS`는 DS와 ES 세그먼트를 STACK에 저장하고, DS, ES, FS, GS 세그먼트 레지스터를 전부 0으로 만든다. `APM_DO_POP_SEGS`는 STACK에 저장된 ES와 DS 레지스터를 이전 값으로 복구하게되며, `APM_DO_RESTORE_SEGS`는 저장된 FS와 GS 세그먼트 레지스터를 이전 값으로 복구 한다. `APM_ZERO_SEGS`가 정의되지 않았다면, 이와 같은 모든 매크로는 전부 NULL space가 될 것이다. `loadsegment`²¹⁹는 `~/linux/include/asm-i386/system.h`에 정의된 것으로 세그먼트 레지스터에 특정 값을 넣도록 만든다.

따라서, `apm_bios_simple_call()` 함수는 `apm_bios_entry`에 있는 함수를 호출하는 일을 한다는 것을 알 수 있다. 이때, EAX, EBX, ECX 레지스터들이 각각 APM BIOS 함수 호출에 대해서 인자를 넘겨주는 역할을 하게되며, eax라는 변수에 함수 호출의 결과를 저장해서 돌려주는 일을 한다.

```

static u8 apm_bios_call(u32 func, u32 ebx_in, u32 ecx_in, u32 *eax, u32 *ebx, u32 *ecx, u32 *edx, u32 *esi)
{
    APM_DECL_SEGS
    unsigned long      flags;

```

²¹⁹ `loadsegment()` 매크로는 세그먼트 레지스터에 특정 값을 넣어주는 것 이외에 .fixup section에 세그먼트 레지스터를 0으로 만들어주는 코드와 exception table에 이를 위한 entry를 추가적으로 넣어주는 코드가 들어가 있다. 즉, exception이 발생할 경우에 대해서 대비하는 일을 처리한다.

```

__save_flags(flags);
APM_DO_CLI;
APM_DO_SAVE_SEGS;
/*
 * N.B. We do NOT need a cld after the BIOS call
 * because we always save and restore the flags.
 */
__asm__ __volatile__(APM_DO_ZERO_SEGS
    "pushl %%edi\n\t"
    "pushl %%ebp\n\t"
    "lcall %%cs:" SYMBOL_NAME_STR(apm_bios_entry) "\n\t"
    "setc %%al\n\t"
    "popl %%ebp\n\t"
    "popl %%edi\n\t"
    APM_DO_POP_SEGS
    : "=a" (*eax), "=b" (*ebx), "=c" (*ecx), "=d" (*edx),
      "=S" (*esi)
    : "a" (func), "b" (ebx_in), "c" (ecx_in)
    : "memory", "cc");
APM_DO_RESTORE_SEGS;
__restore_flags(flags);
return *eax & 0xff;
}

```

코드 579. `apm_bios_call()` 함수의 정의

거의 동일한 일을 하는 `apm_bios_call()`이라는 함수도 있다. 이 함수와 앞에서 설명한 `apm_bios_call_simple()`은 넘겨주는 인자만 달라질 뿐 내부적으로는 동일하다. `apm_bios_call_simple()`에서는 `error`라는 변수를 선언해서 `error` 값을 돌려주는데 사용하지만, `apm_bios_call()`에서는 `eax`에 저장된 값 중에서 AL 레지스터가 차지하는 8 bit만을 돌려준다. 마찬가지로 `apm_bios_entry`를 호출한다. 이때 호출하는데 사용하는 명령어가 `lcall`인데, 이것은 long call²²⁰을 사용해서 APM BIOS의 function을 호출하기 때문이다. 역시 Carry Flag이 설정된 경우에는 이를 AL(앞에서는 `error`에 저장했다.) 레지스터에 저장해서 복귀값으로 사용한다.

```

static int apm_get_info(char *buf, char **start, off_t fpos, int length)
{
    char *          p;
    unsigned short   bx;
    unsigned short   cx;
    unsigned short   dx;
    int             error;
    unsigned short ac_line_status = 0xff;
    unsigned short battery_status = 0xff;
    unsigned short battery_flag   = 0xff;
    int             percentage   = -1;
    int             time_units   = -1;
    char            *units        = "?";
    p = buf;
    if ((smp_num_cpus == 1) &&
        !(error = apm_get_power_status(&bx, &cx, &dx))) {
        ac_line_status = (bx >> 8) & 0xff;
        battery_status = bx & 0xff;
        if ((cx & 0xff) != 0xff)
            percentage = cx & 0xff;

```

²²⁰ Segment와 Offset을 모두 명시해서 함수를 호출하는 것이다.

```

        if (apm_info.connection_version > 0x100) {
            battery_flag = (cx >> 8) & 0xff;
            if (dx != 0xffff) {
                units = (dx & 0x8000) ? "min" : "sec";
                time_units = dx & 0x7fff;
            }
        }
    }
    p += sprintf(p, "%s %d.%d 0x%02x 0x%02x 0x%02x 0x%02x %d%% %d %s\n",
        driver_version,
        (apm_info.bios.version >> 8) & 0xff,
        apm_info.bios.version & 0xff,
        apm_info.bios.flags,
        ac_line_status,
        battery_status,
        battery_flag,
        percentage,
        time_units,
        units);

    return p - buf;
}

```

코드 580. `apm_get_info()` 함수의 정의

`apm_get_info()` 함수는 APM 모듈과 사용자 application간의 인터페이스로 사용되는 함수이다. 즉, 사용자가 `/proc/apm`에 대해서 일기와 같은 동작을 수행할 때 호출되는 함수이다. `smp_num_cpus`가 1인 경우(즉, 시스템의 CPU개수가 한개인 경우에만), `apm_get_power_status()` 함수를 호출해서 시스템의 현재 power 상태를 읽어온다. 이때 읽어오는 시스템의 power 상태는 AC adapter의 연결에 대한 전원 정보와 battery의 현재 전원 상태이다. `apm_get_power_status()`에 넘겨주는 argument로는 현재 AC adapter의 전원 상태 및 battery의 전원 상태를 나타내는 `status(bx)`와 현재 battery에 충전된 상태를 나타내는 `bat(cx)`, 그리고, 지속 가능한 시간을 나타내는 `life(dx)`가 있다. 넘겨준 `bx`의 상위 8bit에는 AC adapter의 전원 상태를 가지고 있고, 하위 8 bit에는 battery의 전원 상태를 담고 있기에 각각을 bit field에 맞게 `ac_line_status`와 `battery_status`에 넣었다. `cx`의 하위 8 bit에는 충전된 `percentage`를 담고 있기에 이를 `percentage`라는 변수에 넣는다. 또한 `cx`의 상위 8 bit에는 APM BIOS의 connection version이 1.0이상인 경우에는 battery의 flag를 담고 있는데, high, low, critical, selected battery not present, no system battery 등이 0번 bit에서 시작해서 7번 bit까지 ch(CX 레지스터의 상위 8 bit)을 차지한다. 따라서, 이러한 정보를 `battery_flag`에 저장한다. `dx`에는 이미 남은 battery 시간을 가진다고 이야기 했는데, `dx`의 16번째 bit이 설정된 경우에는 분(minutes)을 나타내고, 그렇지 않다면 초(seconds)를 나타낸다. 남아 있는 15 bit가 시간값을 가진다. 만약 `dx`가 0xFFFF을 가진다면, 알려지지 않은 값을 가진다고 생각하도록 한다.²²¹ 나머지는 알려온 정보를 그대로 buffer에 저장하고, 저장된 크기를 복귀 값을 돌려준다.

```

static int apm_get_power_status(u_short *status, u_short *bat, u_short *life)
{
    u32      eax;
    u32      ebx;
    u32      ecx;
    u32      edx;
    u32      dummy;

    if (apm_info.get_power_status_broken)
        return APM_32_UNSUPPORTED;
    if (apm_bios_call(APM_FUNC_GET_STATUS, APM_DEVICE_ALL, 0,

```

²²¹ 이것은 APM BIOS Spec.의 0x0A의 BIOS function call에 해당한다. Spec.에 있는 것을 그대로 인용했을 뿐이다.

```

        &eax, &ebx, &ecx, &edx, &dummy))
return (eax >> 8) & 0xff;
*status = ebx;
*bat = ecx;
if (apm_info.get_power_status_swabinminutes) {
    *life = swab16((u16)edx);
    *life |= 0x8000;
} else
    *life = edx;
return APM_SUCCESS;
}

```

코드 581. `apm_get_power_status()` 함수의 정의

`apm_get_power_status()` 함수는 APM BIOS의 Get Power Status(0x0A)를 호출하는 함수이다. 만약 `apm_info.get_power_status_broken0` 있다면, APM_32_UNSUPPORTED를 복귀 값을로 돌려주어 32-bit APM BIOS function call interface를 지원하지 않는다는 것을 알려준다. 그렇지 않다면, `apm_bios_call()` 함수를 불러서 APM BIOS를 호출하게 된다. 이때 에러가 있었다면, EAX 레지스터의 값을 저장하는 eax의 ah에 있는 8bit만을 돌려준다. 나머지 ebx의 내용은 status에 넣고, ecx의 내용은 bat에 넣어서 돌려준다. 만약 `apm_info.get_power_status_swabinminutes` flag이 설정되어 있다면, batter life를 뜻하는 edx의 내용을 byte 단위로 swap시켜서 돌려주게 된다. 이때는 minutes 단위인 경우에 한해서 swap하라는 flag이 이미 설정되어 있으므로, life가 가르키는 데이터의 최상위 1 bit(MSB)을 반드시 1로 만들어주어야 할 것이다. 즉, 분단위로 측정한 결과 이기 때문이다. 그렇지 않다면, 그대로 edx의 내용을 전달한다. 여기까지 진행되었다면, APM_SUCCESS를 돌려주어 성공적으로 호출되었음을 알려준다.

9.4. APM BIOS function의 종류

우린 앞에서 이미 APM BIOS의 function을 호출하는 것을 보았다. 그렇다면, APM에서 정의하고 있는 function에는 어떠한 것들이 있으며, 이를 각각이 하는 역할이 뭔지를 볼 필요가 있겠다. 다음의 표를 보도록 하자. 정의는 `~/linux/include/linux/apm_bios.h`에 있다.

이름	값	설명
APM_FUNC_INST_CHECK	0x5300	APM Installation Check. APM 드라이버에서 시스템의 BIOS가 APM 기능을 지원하는지와, 만약 지원한다면 version이 어떻게 되는지를 묻는데 사용한다.
APM_FUNC_REAL_CONN	0x5301	APM Real Mode Interface Connect. APM 드라이버와 APM BIOS간의 real-mode 인터페이스를 설정한다.
APM_FUNC_16BIT_CONN	0x5302	APM Protected Mode 16-bit Interface Connect. APM 드라이버(Caller)와 APM BIOS간의 16-bit protected mode 인터페이스를 초기화 한다. 이 인터페이스는 APM BIOS 함수들을 real-mode나 virtual-86-mode로의 전환 없이 사용하는 것을 허락한다.
APM_FUNC_32BIT_CONN	0x5303	APM Protected Mode 32-bit Interface Connect. APM 드라이버(Caller)와 APM BIOS간의 32-bit protected mode를 초기화 한다. 마찬가지로 real-mode나 virtual-86-mode로의 전환 없이 APM BIOS 함수들을 사용할 수 있도록 해준다.
APM_FUNC_DISCONNECT	0x5304	APM Interface Disconnect. APM BIOS와 APM 드라이버간의 connection을 끊어서, PM에 대한 정책(policy)을 APM BIOS에게 넘겨준다.
APM_FUNC_IDLE	0x5305	CPU Idle. APM 드라이버에서 APM BIOS에게 현재 시스템이 IDLE이라는 것을 알려준다. 이때 APM BIOS는 interrupt와 같은 시스템의 event가 일어날 때까지 시스템을 suspend시켜준다. APM 드라이버는 APM BIOS의 IDLE

		함수(routine)으로부터 다시 제어를 받으면, 반드시 처리해야 할 일이 있는가를 확인해야 하며, 만약 없다면 다시 CPU IDLE을 발생(issue)시켜준다.
APM_FUNC_BUSY	0x5306	CPU Busy. APM 드라이버가 현재 시스템이 BUSY라고 판단했음을 APM BIOS에게 알려준다. 이때 APM BIOS는 CPU의 clock rate를 full speed로 복구시켜준다. 이 함수는 단지 CPU IDLE 함수가 이전에 불려지고 난 후에만 필요하다. CPU IDLE이 불려지면, CPU의 clock rate는 낮아지게(slow)된다. APM BIOS가 CPU IDLE 동안에 clock rate를 낮아지게 했다는 것을 알기 위해서는 APM Installation Check 함수의 복귀 값인 CX레지스터의 2번 bit을 보아야 할 것이다.
APM_FUNC_SET_STATE	0x5307	Set Power State. 특정 디바이스나 시스템이 요구된 전원 상태로 들어가도록 설정한다.
APM_FUNC_ENABLE_PM	0x5308	Enable/Disable Power Management. APM BIOS의 자동적인(automatic) PM을 enable/disable 시켜준다. 만약 disable되어 있다면, APM BIOS는 Standby, Suspend, Power saving과 같은 것을 자동으로 수행하지 않는다.
APM_FUNC_RESTORE_BIOS	0x5309	Restore APM BIOS Power-On Defaults. 모든 Power-on 기본값(defaults)을 다시 초기화 시켜준다.
APM_FUNC_GET_STATUS	0x530A	Get Power Status. 현재 시스템의 전원 상태를 돌려준다.
APM_FUNC_GET_EVENT	0x530B	Get PM Event. 현재 Pending되어 있는 PM 관련 event를 돌려주거나, 아무런 PM event가 Pending되어 있지 않는지를 알려준다. 이 함수는 더 이상의 PM event가 pending되어 있지 않을 때까지 계속 호출되어야 한다. 또한 APM 드라이버는 적어도 1초에 한번씩은 새로운 event가 있는지를 확인하기 위해서 polling을 해야 할 것이다. 각각의 event들은 시스템이나 디바이스에 적용된다.
APM_FUNC_GET_STATE	0x530C	Get Power State. 특정의 디바이스 ID가 주어진 경우, 이 디바이스의 전원 상태를 돌려준다. 주어지는 디바이스의 ID에는 모든 디바이스 혹은 디바이스 class, 및 특정의 한 디바이스를 가질 수 있다.
APM_FUNC_ENABLE_DEV_PM	0x530D	Enable/Disable Device Power Management. APM BIOS에게 명시된 특정 디바이스 ID에 대해서 자동적인 전원 관리를 enable/disable시키도록 한다. Disable되어 있을 경우, APM BIOS는 해당 디바이스 ID에 대해서 자동적으로 전원 관리를 수행하지 않는다. 시스템이 Reset된 상황에서는 모든 디바이스들에 대해서 enable된 상태이다.
APM_FUNC_VERSION	0x530E	APM Driver Version. APM BIOS에 대해서 APM 드라이버가 지원하는 version을 알려주기 위해서 사용한다. 이때 APM BIOS에서는 APM connection version 번호를 돌려준다.
APM_FUNC_ENGAGE_PM	0x530F	Engage/Disengage Power Management. APM 드라이버와 APM BIOS간의 전원 관리에 있어서의 협동관계를 설정하거나 해제한다. Disengage된 상태라면, APM BIOS에서 자동적으로 시스템이나 디바이스에 대해서 전원 관리를 책임진다. 시스템이 Reset이 된 후에는 disengage로 되어 있다.
APM_FUNC_GET_CAP	0x5310	Get Capabilities. 특정 APM BIOS(version 1.2)의 구현이 지원하는 특징들을 돌려준다.
APM_FUNC_RESUME_TIMER	0x5311	Get/Set/Disable Resume Timer. 시스템 resume timer에 대해서 얻기/설정하기/해제(disable)하기 등을 요청한다.

APM_FUNC_RESUME_ON_RING	0x5312	Enable/Disable Resume on Ring Indicator. 시스템에 resume이라는 event를 알려줄 수 있는 링(ring : modem 등에서) 기능을 사용할 것인지를 결정한다.
APM_FUNC_TIMER	0x5313	Enable/Disable Timer Based Requests. 비활동 timer를 바탕으로해서(based) 전역적인(global) Standby나 Suspend 요청을 생성하는 것을 enable/disable시킨다. 기본적으로는 APM BIOS에서 Power-on이나 기본값(defaults)들을 설정하도록 요청받았을 경우에는 enable상태로 되어 있다.

표 55. APM BIOS function의 요약(Summary)²²²

[표 55]에서 사용하는 값들은 실제로 BIOS의 함수를 호출하기 위해서 AX레지스터에 들어가는 값이다. 즉, AH와 AL에 각각 나누어져 들어가서 해당 BIOS 함수를 호출하는데 사용된다. 표에서 보여준 함수들 이외에 OEM-Defined APM 함수들과 OEM APM 함수가 있지만, 여기서는 보지 않도록 하겠다. 나중에 이들 각 함수들이 어떻게 사용되는 지는 코드를 통해서 알 수 있을 것이다. 따라서, 그때 그때 정확한 사용을 좀더 자세히 보도록 하겠다.

9.5. APM Setup String의 해석

Setup 문자열(string)을 해석하는 함수는 커널에서 넘겨받는 parameter를 해석(parsing)하는 역할을 수행한다. 이와같은 역할을 수행하기 위해서는 __setup()이라는 매크로가 필요하다. 해석되기를 원하는 parameter가 “apm=”으로 주어지면, 이것을 해석할 함수로 `apm_setup()`을 수행하도록 만들어준다.

```
#ifndef MODULE
static int __init apm_setup(char *str)
{
    int      invert;

    while ((str != NULL) && (*str != '\0')) {
        if (strncmp(str, "off", 3) == 0)
            apm_disabled = 1;
        if (strncmp(str, "on", 2) == 0)
            apm_disabled = 0;
        if ((strncmp(str, "bounce-interval=", 16) == 0) ||
            (strncmp(str, "bounce_interval=", 16) == 0))
            bounce_interval = simple_strtol(str + 16, NULL, 0);
        if ((strncmp(str, "idle-threshold=", 15) == 0) ||
            (strncmp(str, "idle_threshold=", 15) == 0))
            idle_threshold = simple_strtol(str + 15, NULL, 0);
        if ((strncmp(str, "idle-period=", 12) == 0) ||
            (strncmp(str, "idle_period=", 12) == 0))
            idle_period = simple_strtol(str + 12, NULL, 0);
        invert = (strncmp(str, "no-", 3) == 0) ||
                  (strncmp(str, "no_", 3) == 0);
        if (invert)
            str += 3;
        if (strncmp(str, "debug", 5) == 0)
            debug = !invert;
        if ((strncmp(str, "power-off", 9) == 0) ||
            (strncmp(str, "power_off", 9) == 0))
            power_off = !invert;
        if ((strncmp(str, "allow-ints", 10) == 0) ||
            (strncmp(str, "allow_ints", 10) == 0))
            apm_info.allow_ints = !invert;
    }
}
```

²²² 더 자세히 알고 싶다면, APM BIOS version 1.2 Spec.을 참조하기 바란다.

```

        if ((strncmp(str, "broken-psr", 10) == 0) ||
            (strncmp(str, "broken_psr", 10) == 0))
            apm_info.get_power_status_broken = !invert;
        if ((strncmp(str, "realmode-power-off", 18) == 0) ||
            (strncmp(str, "realmode_power_off", 18) == 0))
            apm_info.realmode_power_off = !invert;
        str = strchr(str, ',');
        if (str != NULL)
            str += strspn(str, ", \t");
    }
    return 1;
}
__setup("apm=", apm_setup);
#endif

```

코드 582. **apm_setup()** 함수의 정의

apm_setup() 함수는 일종의 모듈로 컴파일 되었을 경우에 모듈의 파라미터를 해석한다고 생각해 볼 수 있다. 하는 역할도 동일하다. 해석하는 문자열에는 “off”, “on”에 따른 **apm_disabled**의 값을 정하는 것, **bounce_interval**을 정하는 “bounce-interval”, **idle_threshold**의 값을 정하는 “idle-threshold”, **idle_period**의 값을 정하는 “idle-period”가 있으며, “no-“라는 문자열이 주어진 경우에는 **invert**값을 1로 설정해서 다음번에 비교 대상이 되는 string의 값들에 영향을 주도록 만든다. **invert**라는 변수에 영향을 받는 값으로는 **debug**, **power_off**, **allow_ints**, **broken_psr**, **realmode_power_off**등이 있으며, 이에 해당하는 문자열이 있을 경우에 **invert**의 값을 NOT해서 사용하도록 만든다. **strchr()** 함수는 문자열 내에서 해당 문자를 찾아서, 위치를 옮기는 역할을 수행하며, **strspn()** 함수는 해당 문자열이 몇개가 연속적인가를 알아내는 함수로 공백문자와 같은 것을 지워, 다음 문자열의 엔트리로 이동하도록 만든다. **Setup** 문자열은 여기서 **NULL**이거나 혹은 ‘\0’를 가질 때까지 계속 수행될 것이다. 복귀 값은 1이다. 여기서 수행된 결과에 따라서, APM 모듈의 동작이 달라지게 될 것이다. 한번더 말하거나와 이 함수는 커널에 **static**하게 **link**되어 컴파일 되는 경우에만 수행될 것이다. 모듈로서 컴파일해서 사용하려면 모듈을 로딩/loading)하는데 파라미터로 넘겨주어야 할 것이다.

9.6. APM BIOS의 파일 연산 벡터(File Operation Vector)

```

static struct file_operations apm_bios_fops = {
    owner:           THIS_MODULE,
    read:            do_read,
    poll:            do_poll,
    ioctl:           do_ioctl,
    open:            do_open,
    release:         do_release,
};

static struct miscdevice apm_device = {
    APM_MINOR_DEV,
    "apm_bios",
    &apm_bios_fops
};

```

코드 583. APM 모듈의 파일 연산 벡터의 정의

APM 모듈은 miscellaneous 디바이스 드라이버로 등록된다. 따라서, major 번호는 10번을 가질 것이며, minor번호(APM_MINOR_DEV)는 134번을 가진다. 이렇게 등록된 APM 디바이스 드라이버는 호출되는 순간은 misc 디바이스 드라이버를 통해서만 가능할 것이다. 즉, misc 디바이스 드라이버와 응용 프로그램이 인터페이스를 하게되고, misc 디바이스 드라이버 내에서 APM 디바이스 드라이버를

호출하게될 것이다. 어차피 문자열 디바이스 드라이버로 APM이 등록되기에 문자 디바이스 드라이버들이 갖추고 있는 파일 연산 벡터를 지녀야한다.

9.6.1. Open 함수의 분석

```
static int do_open(struct inode * inode, struct file * filp)
{
    struct apm_user * as;

    as = (struct apm_user *)kmalloc(sizeof(*as), GFP_KERNEL);
    if (as == NULL) {
        printk(KERN_ERR "apm: cannot allocate struct of size %d bytes\n",
               sizeof(*as));
        return -ENOMEM;
    }
    as->magic = APM_BIOS_MAGIC;
    as->event_tail = as->event_head = 0;
    as->suspends_pending = as->standbys_pending = 0;
    as->suspends_read = as->standbys_read = 0;
    /*
     * XXX - this is a tiny bit broken, when we consider BSD
     * process accounting. If the device is opened by root, we
     * instantly flag that we used superuser privs. Who knows,
     * we might close the device immediately without doing a
     * privileged operation -- cevans
     */
    as->suser = capable(CAP_SYS_ADMIN);
    as->writer = (filp->f_mode & FMODE_WRITE) == FMODE_WRITE;
    as->reader = (filp->f_mode & FMODE_READ) == FMODE_READ;
    as->next = user_list;
    user_list = as;
    filp->private_data = as;
    return 0;
}
```

코드 584. do_open() 함수의 정의

do_open() 함수는 apm_user라는 구조체를 만들고, 이를 user_list에 연결하는 일을 제외하고는 실제적인 hardware에 대한 연산은 아무것도 없다. apm_user라는 구조체를 할당할 수 없다면 -ENOMEM을 돌려주고, 할당 했다면, 이 구조체를 초기화 시켜준다. 나중에 다른 함수들에서도 사용할 수 있도록 하기위해서 filp->private_data에 저장하는 일을 수행한다.

struct apm_user {		
int magic;		/* APM BIOS의 magic number : APM_BIOS_MAGIC = 0x4101
*/		
struct apm_user * next;		/* 다음 apm_user 구조체의 포인터 */
int suser: 1;		/* Super user(root)가 사용중을 나타내는 flag */
int writer: 1;		/* Writer가 있음을 나타내는 flag */
int reader: 1;		/* Reader가 있음을 나타내는 flag */
int suspend_wait: 1;		/* Wait하고 있음을 나타내는 flag */
int suspend_result;		/* APM의 결과를 대기중임을 나타내는 flag */
int suspends_pending;		/* Pending되어 있음을 나타내는 flag */
int standbys_pending;		/* 실행준비가 되어 있음을 나타내는 flag */
int suspends_read;		/* Read를 기다고 있음을 나타내는 flag */
int standbys_read;		/* Read가 준비되었음을 나타내는 flag */
int event_head;		/* PM과 관련된 event들의 배열의 시작을 나타내는 index */

```

int          event_tail;      /* PM과 관련된 event들의 배열의 끝을 나타내는 index */
apm_event_t events[APM_MAX_EVENTS]; /* PM 관련 event들의 배열 */
};

```

코드 585. `apm_user` 구조체의 정의

`apm_user` 구조체는 현재 누가 APM 디바이스를 사용하고 있는가를 나타내는 자료구조이다. `capable()`은 현재 응용프로그램의 사용자가 누구인가를 확인한다. 이때 사용하는 `CAP_SYS_ADMIN`은 시스템을 관리할 수 있는 능력(혹은 권한 : capability)을 가지고 있는지 확인해서 이를 `suser`필드에 넣는다. `writer`와 `reader`는 넘어온 `argument`의 값을 보고 설정된다.

9.6.2. Release 함수의 분석

```

static int do_release(struct inode * inode, struct file * filp)
{
    struct apm_user * as;

    as = filp->private_data;
    if (check_apm_user(as, "release"))
        return 0;
    filp->private_data = NULL;
    lock_kernel();
    if (as->standbys_pending > 0) {
        standbys_pending -= as->standbys_pending;
        if (standbys_pending <= 0)
            standby();
    }
    if (as->suspends_pending > 0) {
        suspends_pending -= as->suspends_pending;
        if (suspends_pending <= 0)
            (void) suspend(1);
    }
    if (user_list == as)
        user_list = as->next;
    else {
        struct apm_user * as1;

        for (as1 = user_list;
             (as1 != NULL) && (as1->next != as);
             as1 = as1->next)
            ;
        if (as1 == NULL)
            printk(KERN_ERR "apm: filp not in user list\n");
        else
            as1->next = as->next;
    }
    unlock_kernel();
    kfree(as);
    return 0;
}

```

코드 586. `do_release()` 함수의 정의

`do_release()` 함수는 사용자 응용프로그램에서 `close()`함수를 파일에 대해서 호출한 경우에 사용된다. 따라서, 앞에서 했던 `open()` 함수의 역을 수행하게될 것이다. 다만 이때는 이미 진행중인 연산들에 대해서 대기해야하는 경우에는 그 연산이 수행을 마치도록 기다릴 것이다. 먼저 `filp->private_data`에

저장해 두었던 `apm_user` 구조체를 얻어오도록 한다. `check_apm_user()`함수는 해당 사용자가 올바른 사용자인가를 확인한다. 간단히 `apm_user` 구조체의 magic number 필드에 들어가 있는 번호를 비교한다.

```
static int check_apm_user(struct apm_user *as, const char *func)
{
    if ((as == NULL) || (as->magic != APM_BIOS_MAGIC)) {
        printk(KERN_ERR "apm: %s passed bad filp\n", func);
        return 1;
    }
    return 0;
}
```

코드 587. `check_apm_user()` 함수의 정의

비교해서 동일하지 않다면 1을 넘겨주고, 같다면 0을 넘겨준다. 이전 `filp->private_data`를 더 이상 사용할 일이 없으므로, 당연히 NULL을 넣어준다. 같은 사용자가 다른 함수를 호출하더라도, 같은 일을 반복하지는 않을 것이다. 이전 커널의 자료구조로 선언해준 `user_list`에 대한 연산에 들어가므로, `lock_kernel()`을 사용해서 커널 전체에 대해서 lock을 설정한다. 만약 `standbys_pending`필드가 0이상을 가진다면, `standbys_pending`이라는 전역 변수에서 현재 사용자의 `standbys_pending`값을 빼고, 이렇게해서 `standbys_pending`이 0보다 작거나 같은 경우에는 `standby()`를 호출한다.

```
static void standby(void)
{
    int      err;

    /* If needed, notify drivers here */
    get_time_diff();
    err = set_system_power_state(APM_STATE_STANDBY);
    if ((err != APM_SUCCESS) && (err != APM_NO_ERROR))
        apm_error("standby", err);
}
```

코드 588. `standby()` 함수의 정의

`standby()` 함수는 CMOS에 들어있는 시간과 현재시간을 비교해서 차이를 저장하도록 하는 `get_time_diff()`를 호출하고, 시스템의 상태를 STANDBY로 바꾼다(APM_STATE_STANDBY). 만약 오류가 있었다면, `apm_error()`를 호출해서 처리하도록 한다.

```
static int set_power_state(u_short what, u_short state)
{
    u32      eax;

    if (apm_bios_call_simple(APM_FUNC_SET_STATE, what, state, &eax))
        return (eax >> 8) & 0xff;
    return APM_SUCCESS;
}
static int set_system_power_state(u_short state)
{
    return set_power_state(APM_DEVICE_ALL, state);
}
```

코드 589. `set_power_state()` 함수 및 `set_system_power_state()` 함수의 정의

`set_power_state()` 함수는 현재 시스템에 있는 특정 디바이스의 전원 상태를 설정하기 위해서 호출되는 함수이다. `set_system_power_state()` 함수는 시스템 전체의 디바이스들에 대한 전원 상태를 설정하기 위해서 호출되는 함수이다. 따라서, `set_system_power_state()`는 `set_power_state()` 함수를 호출할 때, 대상이 되는 디바이스의 ID에 APM_DEVICE_ALL(=0x0001)을 두고 호출한다.

`set_power_state()` 함수는 `apm_bios_call_simple()` 함수를 호출하는데, 이때 사용하는 function 번호는 `APM_FUNC_SET_STATE`이다. 대상이 되는 디바이스는 what으로 나타내며, 설정할 전원 상태는 state로 `eax`는 복귀 값을 받기 위해서 사용한다. 예러가 있었다면, `eax`의 AH부분에 해당하는 에러코드를 돌려줄 것이다. 성공적이었다면, `APM_SUCCESS`를 돌려줄 것이다.

여기서 잠시 `set_power_state()` 함수에 대해서 알아보기로 하자. 먼저 이 함수가 사용하는 device class에 대한 것을 보면 아래와 같이 `~/linux/include/linux/apm_bios.h`에 정의되어 있다.

<code>#define APM_DEVICE_BIOS</code>	0x0000 /* APM BIOS를 지정한다. */
<code>#define APM_DEVICE_ALL</code>	0x0001 /* 모든 디바이스를 지정한다. */
<code>#define APM_DEVICE_DISPLAY</code>	0x0100 /* Display class중 첫번째 디바이스를 지정한다. */
<code>#define APM_DEVICE_STORAGE</code>	0x0200 /* Storage class중 첫번째 디바이스를 지정한다. */
<code>#define APM_DEVICE_PARALLEL</code>	0x0300 /* Parallel class중 첫번째 디바이스를 지정한다. */
<code>#define APM_DEVICE_SERIAL</code>	0x0400 /* Serial class중 첫번째 디바이스를 지정한다. */
<code>#define APM_DEVICE_NETWORK</code>	0x0500 /* Network class중 첫번째 디바이스를 지정한다. */
<code>#define APM_DEVICE_PCMCIA</code>	0x0600 /* PCMCIA class중 첫번째 디바이스를 지정한다. */
<code>#define APM_DEVICE_BATTERY</code>	0x8000 /* Reserved라고 version 1.2에 명시됨. Battery class */
<code>#define APM_DEVICE_OEM</code>	0xe000 /* OEM-defined Power 디바이스 ID */
<code>#define APM_DEVICE_OLD_ALL</code>	0xffff /* 이전 버전의 spec.에서 모든 디바이스를 지정한다. */
<code>#define APM_DEVICE_CLASS</code>	0x00ff /* 각 class이 전체 디바이스의 bit-wise OR mask 값 */
<code>#define APM_DEVICE_MASK</code>	0xff00 /* 각 class에 대한 bit-wise AND mask 값 */

코드 590. APM Power Device ID값의 정의

또한, 설정할 전원 상태는 다시 아래와 같은 값으로 정의될 수 있다. 이것 역시 같은 파일에 정의되어 있다.

<code>/* Set Power State(07H) */</code>	
<code>#define APM_STATE_READY</code>	0x0000 /* APM이 Enable되었다. */
<code>#define APM_STATE_STANDBY</code>	0x0001 /* 대기 모드 */
<code>#define APM_STATE_SUSPEND</code>	0x0002 /* 일시 정지 모드 */
<code>#define APM_STATE_OFF</code>	0x0003 /* 전원 차단 모드 */
<code>#define APM_STATE_BUSY</code>	0x0004 /* 마지막 요청을 처리한다고 바쁘다. */
<code>#define APM_STATE_REJECT</code>	0x0005 /* 마지막 요청이 거부되었다. */
<code>#define APM_STATE_OEM_SYS</code>	0x0020 /* OEM에서 정의한 시스템 상태 */
<code>#define APM_STATE_OEM_DEV</code>	0x0040 /* OEM에서 정의한 디바이스의 상태 */
<code>/* Enable/Disable Device Power Management(0DH) : Function Code */</code>	
<code>#define APM_STATE_DISABLE</code>	0x0000
<code>#define APM_STATE_ENABLE</code>	0x0001
<code>/* Engage/Disengage Power Management(0FH) : Function Code */</code>	
<code>#define APM_STATE_DISENGAGE</code>	0x0000
<code>#define APM_STATE_ENGAGE</code>	0x0001

코드 591. APM Power State에 대한 정의

위에서 정의한 APM 상태(state)중에서 enable/disable과 engage/disengage는 APM의 상태를 표시하기 보다는 특정 APM BIOS의 function을 호출할 때 사용하는 function code로 쓴다. 즉, 0FH 및 08H function에 대해서 CX 레지스터에 들어가는 내용이 될 것이다.

다시 앞에서 하던 이야기를 계속하면, `suspends_pending`에 대해서도 같은 식으로 해준다. 이때는 `suspend()` 함수에 1을 넘겨주어서 호출하도록 한다. 만약 `user_liset`가 현재의 `apm_user` 구조체와 같다면, `user_list`에서 제거하기 위해서 다음(next) 구조체를 가르키도록 만든다. 그렇지 않다면, 해당 `apm_user` 구조체를 찾기 위해서 loop를 돌게되고, 찾을 수 없다면, 예러 메시지를 보인다. 복귀하기 전에 앞에서 설정했던 kernel의 lock을 풀고(`unlock_kernel()`), `apm_user` 구조체를 해제(`kfree()`)한 다음 0을 돌려준다.

```

static int suspend(int vetoable)
{
    int             err;
    struct apm_user *as;

    if (pm_send_all(PM_SUSPEND, (void *)3)) {
        /* Vetoed */
        if (vetoable) {
            if (apm_info.connection_version > 0x100)
                set_system_power_state(APM_STATE_REJECT);
            err = -EBUSY;
            ignore_sys_suspend = 0;
            printk(KERN_WARNING "apm: suspend was vetoed.\n");
            goto out;
        }
        printk(KERN_CRIT "apm: suspend was vetoed, but suspending anyway.\n");
    }
    get_time_diff();
    __cli();
    err = set_system_power_state(APM_STATE_SUSPEND);
    reinit_timer();
    set_time();
    ignore_normal_resume = 1;
    __sti();
    if (err == APM_NO_ERROR)
        err = APM_SUCCESS;
    if (err != APM_SUCCESS)
        apm_error("suspend", err);
    err = (err == APM_SUCCESS) ? 0 : -EIO;
    pm_send_all(PM_RESUME, (void *)0);
    queue_event(APM_NORMAL_RESUME, NULL);
out:
    for (as = user_list; as != NULL; as = as->next) {
        as->suspend_wait = 0;
        as->suspend_result = err;
    }
    wake_up_interruptible(&apm_suspend_waitqueue);
    return err;
}

```

코드 592. suspend() 함수의 정의

suspend() 함수의 역할은 궁극적으로 시스템의 전원 상태를 SUSPEND상태로 바꾸는 역할을 수행한다. 먼저 pm_send_all()²²³을 호출해서 PM에 등록된 모든 BUS나 디바이스 드라이버들에 PM_SUSPEND를 전달한다. 이때 만약 오류가 있었다면, 특정 BUS나 디바이스 드라이버에 의해서 시스템이 SUSPEND상태로 전환하는데 필요한 일을 할 수 없다는 뜻이되므로, vetoable을 살펴서 이를 적절히 처리한다. 즉, SUSPEND상태로 가지 못하는 것이다. 오류 값은 -EBUSY가 될 것이며, 만약 connection version이 1.0보다 크다면, 시스템의 상태를 다시 APM_STATE_REJECT로 변경되도록 한다.

get_time_diff()는 앞에서 설명한 것과 같이, 현재 시스템의 시간과 CMOS에 저장된 시간의 차이를 업데이트(update)하고, __cli()를 호출해서 시스템에 interrupt가 발생하지 못하도록 만든다. 이전 시스템의 전원 상태를 APM_STATE_SUSPEND로 바꾸고, timer를 다시 초기화하고(reinit_timer()), 시스템의 시간을 갱신(set_time())한 후, 일반적인 RESUME(재개)를 허가하지 않는다는 뜻으로 ignore_normal_resume을 1로 설정한다. 이때 시스템은 이제 SUSPEND상태로 진행할 것이므로,

²²³ 커널에서 사용하는 PM에 대한 것을 이야기 할 때 볼 수 있을 것이다.

이것을 마치면 interrupt를 허가하도록 __sti()를 호출한다. 만약 위의 과정에서 오류가 없었다면(APM_NO_ERROR), 복귀 코드로는 APM_SUCCESS가 주어질 것이다. 그렇지 않다면, `apm_error()` 함수를 호출해서 오류 메시지를 출력한다. 다시 복귀 코드값을 비교해서 APM_SUCCESS일 경우에는 0을 그렇지 않다면 -EIO²²⁴로 설정하도록 하고, `pm_send_all()` 함수를 호출해서 각각의 BUS와 디바이스 드라이버에 PM_RESUME을 전달한다. `queue_event()` 함수는 APM_NORMAL_RESUME을 받을 event로 설정하고, `user_list`에 있는 모든 `apm_user` 구조체의 `suspend_wait`와 `suspend_result`를 각각 0과 에러 값으로 설정한다. 여기서 `wake_up_interruptible()` 함수를 호출해서 APM의 suspend wait queue에서 대기중인 모든 process들을 깨워주게 된다. 복귀 값은 앞의 연산의 결과를 가지는 err이다.

```
static void get_time_diff(void)
{
#ifndef CONFIG_APM_RTC_IS_GMT
    unsigned long      flags;

    /*
     * Estimate time zone so that set_time can update the clock
     */
    save_flags(flags);
    clock_cmos_diff = -get_cmos_time();
    cli();
    clock_cmos_diff += CURRENT_TIME;
    got_clock_diff = 1;
    restore_flags(flags);
#endif
}
```

코드 593. `get_time_diff()` 함수의 정의

`get_time_diff()` 함수는 CONFIG_APM_RTC_IS_GMT옵션이 선택되어 있는 경우에 대해서만, `get_cmos_time()`²²⁵ 함수를 호출해서 현재 CMOS에 저장되어 있는 시간을 구해서 `clock_cmos_diff`에 저장하고, 현재 시간(CURRENT_TIME)²²⁶과의 차이를 다시 `clock_cmos_diff`에 넣는다. `got_clock_diff`는 시간의 차이를 계산했다는 것을 나타내는 flag으로 사용하며, CURRENT_TIME을 구하기 전에 `cli()`를 사용해서 인터럽트의 발생을 막아준다. 나중에 `restore_flags()`에서 EFLAG register의 내용을 다시 지정하기 때문에 `sti()`를 해줄 필요가 없다.

```
static void reinit_timer(void)
{
#ifndef INIT_TIMER_AFTER_SUSPEND
    unsigned long      flags;

    save_flags(flags);
    cli();
    /* set the clock to 100 Hz */
    outb_p(0x34,0x43);          /* binary, mode 2, LSB/MSB, ch 0 */
    udelay(10);
    outb_p(LATCH & 0xff , 0x40); /* LSB */
    udelay(10);
    outb(LATCH >> 8 , 0x40);   /* MSB */
    udelay(10);
    restore_flags(flags);
#endif
}
```

²²⁴ Error In I/O.

²²⁵ `get_cmos_time()` 함수는 `~/linux/arch/i386/kernel/time.c`에 정의되어 있다.

²²⁶ CURRENT_TIME은 `~/linux/include/linux/sched.h`에 다음과 같이 정의되어 있다. `#define CURRENT_TIME (xtime.tv_sec)`, 즉, timer tick에 의해서 관리되는 시간중에서 초를 가져오는 매크로이다.

```
#endif
}
```

코드 594. reinit_timer() 함수의 정의

reinit_timer() 함수는 INIT_TIMER_AFTER_SUSPEND가 정의된 경우에만 일을 한다. 나머지는 Programmable Interval Timer(PIT) chip인 8254 CMOS chip에 대해서 새롭게 프로그램해주는 일이다. 8254 chip이 사용하는 port는 0x40에서 0x43까지의 4개의 port이다. LATCH는 clock tick이 얼마나 발생할지를 결정하는 변수이다. 이것을 두번에 걸쳐서 0x40 port에 LSB, MSB의 순서로 쓴다(outb()).

```
static void set_time(void)
{
    unsigned long      flags;

    if (got_clock_diff) {          /* Must know time zone in order to set clock */
        save_flags(flags);
        cli();
        CURRENT_TIME = get_cmos_time() + clock_cmos_diff;
        restore_flags(flags);
    }
}
```

코드 595. set_time() 함수의 정의

set_time() 함수는 _got_clock_diff라는 변수가 설정되어 있다면, 이를 현재의 시간(CURRENT_TIME)에 반영하는 일을 한다. 즉, get_cmos_time() 함수를 호출해서 CMOS에 저장된 시간과 clock_cmos_diff를 더한 값을 CURRENT_TIME에 저장한다. 이때 get_cmos_time()을 호출하기전에 interrupt의 발생을 막기 위해서 cli()를 호출하고 있다.

```
/*
 * Power management requests
 */
enum
{
    PM_SUSPEND, /* enter D1-D3 */           /* 전원 차단 모드로 전환 */
    PM_RESUME,  /* enter D0 */             /* 전원 재사용 모드로 전환 */
    PM_SAVE_STATE,                      /* 디바이스의 상태를 저장하라 */
    PM_SET_WAKEUP,                     /* 디바이스가 시스템을 깨울 수 있도록 만들라 */
    PM_GET_RESOURCES,                 /* 버스(BUS)가 사용하는 자원 내역을 구하라 */
    PM_SET_RESOURCES,                 /* 버스가 사용하는 자원의 내역을 설정하라 */
    PM_EJECT,                          /* EJECT */
    PM_LOCK,                           /* LOCK */
};
typedef int pm_request_t;
```

코드 596. Power Management Request(pm_request)의 정의

Power Management의 요청(request)는 pm_request_t이라는 타입(type)으로 정의된다. pm_request_t는 정수(integer) 타입으로 정의된 변수로 PM_XXX와 같은 형태의 상수들이 들어갈 수 있다. 이와 같은 PM 요청은 나중에 각각의 버스와 디바이스에 내려지는 명령이 될 것이다.

9.6.3. Read 함수의 분석

Read() 함수는 사용자 application에서 해당 디바이스 드라이버 모듈에서 데이터를 얻기 위해서 호출된다. 따라서, 이 함수에는 파일 구조체(디바이스의 node를 의미한다) 및 읽은 데이터를 저장하는 buffer의 포인터와 읽을 데이터의 크기를 알려준는 count, 현재 파일에서의 위치(position)등이 넘어온다.

```

static ssize_t do_read(struct file *fp, char *buf, size_t count, loff_t *ppos)
{
    struct apm_user * as;
    int i;
    apm_event_t event;

    as = fp->private_data;
    if (check_apm_user(as, "read"))
        return -EIO;
    if (count < sizeof(apm_event_t))
        return -EINVAL;
    if ((queue_empty(as)) && (fp->f_flags & O_NONBLOCK))
        return -EAGAIN;
    wait_event_interruptible(apm_waitqueue, !queue_empty(as));
    i = count;
    while ((i >= sizeof(event)) && !queue_empty(as)) {
        event = get_queued_event(as);
        if (copy_to_user(buf, &event, sizeof(event))) {
            if (i < count)
                break;
            return -EFAULT;
        }
        switch (event) {
        case APM_SYS_SUSPEND:
        case APM_USER_SUSPEND:
            as->suspends_read++;
            break;
        case APM_SYS_STANDBY:
        case APM_USER_STANDBY:
            as->standbys_read++;
            break;
        }
        buf += sizeof(event);
        i -= sizeof(event);
    }
    if (i < count)
        return count - i;
    if (signal_pending(current))
        return -ERESTARTSYS;
    return 0;
}

```

코드 597. do_read() 함수의 정의

file 구조체의 private_data로 부터 apm_user 구조체의 포인터를 얻는다. 현재 사용자가 유효한 apm_user 구조체를 가지는 가를 확인하고(check_apm_user()), 그렇지 않다면 -EIO(Error in I/O)를 돌려준다. 넘겨받은 count값이 apm_event_t보다 작은 값을 가진다면, event queue에 들어있는 값을 제대로 다 복사해 줄 수 없으므로 -EINVAL(Error Invalid)를 돌려준다. queue_empty()는 event queue가 비어있는지를 확인하게되며, 만약 non-blocking 모드로 디바이스 노드를 open했다면(O_NONBLOCK), 여기서 바로 -EAGAIN(Error, try it again)을 돌려준다. 즉, 읽으려는 데이터가 없고, 데이터가 생기기 전까지 기다리지 않겠다(non-blocking)는 것을 밝힌 것이므로 이렇게 하는 것이 옳다. 그렇지 않다면, wait_event_interruptible()를 호출해서 queue가 비지 않을 때까지 대기(wait)하게 된다(!queue_empty()). 이때 여기서 깨어나게되면, event queue가 비지 않았을 것이므로 읽어갈 데이터가 존재하게 된다.

복사는 while() loop를 돌면서 수행된다. get_queued_event()는 해당 `apm_user`의 구조체로 부터 event²²⁷을 가져오는 역할을 수행한다. 이렇게 가져온 event는 `copy_to_user()` 함수를 통해서 사용자 application의 주소 영역에 있는 버퍼(buf)로 복사되며, 오류가 있었을 경우에는 현재 복사한 용량을 가지고 있는 i 값이 원래 복사할 양을 나타내는 count보다 작을 가지는 경우에는 while() loop를 빠져나오게 되지만, 그렇지 않은 경우에는 -EFAULT(Error Fault)를 돌려준다. 즉, 복사를 하는 마지막 단계에서 오류가 발생한 것이기에 곧바로 복귀한다. 이전 queue에서 빠져나온 event의 개수를 반영하기 위해서 event값에 따라 다음과 같은 일을 한다. APM_SYS_SUSPEND나 APM_USER_SUSPEND인 경우에는 `apm_user` 구조체의 `suspends_read`를 증가시키고, APM_SYS_STANDBY나 APM_USER_STANDBY인 경우에는 `standbys_read`를 증가시킨다. 다음번 while() loop를 수행하기 전에 버퍼(buf)의 포인터를 event의 크기만큼 이동시키고, 남은 복사할 데이터의 양을 나타내는 i 값을 event 크기만큼 감소시킨다. while() loop를 빠져나와서 만약 i 값이 count값보다 작다면 복귀 값으로 count에서 i 값을 뺀 값을 돌려주어 얼마나 많은 event들을 읽었는지를 알려줄 수 있도록 한다. 만약 i 값이 count와 크거나 같은 값²²⁸을 가진다면 그 이하를 실행할 것이다. 즉, 일어야 할 데이터를 다 잘읽었다면, 현재 프로세스에 pending된 signal이 있는지를 확인한다(`signal_pending()`). 만약 pending된 signal이 있다면, -ERESTARTSYS(Error Restart System Call)을 돌려준다. 복귀 값은 0이다. 이것은 앞에서 `wait_event_interruptible()` 호출에서 문제가 있었는가를 확인하는 작업이라고 보면 될 것이다.

```
static int queue_empty(struct apm_user *as)
{
    return as->event_head == as->event_tail;
}

static apm_event_t get_queued_event(struct apm_user *as)
{
    as->event_tail = (as->event_tail + 1) % APM_MAX_EVENTS;
    return as->events[as->event_tail];
}
```

코드 598. queue_empty() 함수와 get_queued_event() 함수의 정의

`queue_empty()` 함수는 `apm_user` 구조체에 있는 event queue가 비었는가를 확인하는 역할을 한다. `event_head`와 `event_tail`이 같은 값을 가진다면 비어있는 것이다. `get_queued_event()` 함수는 queueing된 event를 하나 읽어서 돌려주는 일을 한다. `event_tail`의 값을 증가시키고, 여기에 있는 event를 가져온다.

9.6.4. I/O Control 함수의 분석

I/O Control은 디바이스의 동작에 세밀한 영향을 미치는 연산을 적용할 경우에 주로 사용한다. 즉, 사용자 application에서 디바이스의 동작을 제어하고 싶은 경우에 이 함수를 사용할 수 있다.

```
static int do_ioctl(struct inode * inode, struct file *filp,
                   u_int cmd, u_long arg)
{
    struct apm_user * as;

    as = filp->private_data;
    if (check_apm_user(as, "ioctl"))
        return -EIO;
    if ((!as->suser) || (!as->writer))
        return -EPERM;
    switch (cmd) {
```

²²⁷ `apm_event_t`으로 정의된 것으로 `apm_event_t`은 `~/linux/include/linux/apm_bios.h`에 `unsigned short`로 정의된 것이다.

²²⁸ i 같은 event queue의 크기를 초과 할수는 없다. 하지만, 읽어야 할 count값보다는 항상 작거나 같은 값을 가질 것이다.

```

case APM_IOC_STANDBY:
    if (as->standbys_read > 0) {
        as->standbys_read--;
        as->standbys_pending--;
        standbys_pending--;
    } else
        queue_event(APM_USER_STANDBY, as);
    if (standbys_pending <= 0)
        standby();
    break;
case APM_IOC_SUSPEND:
    if (as->suspends_read > 0) {
        as->suspends_read--;
        as->suspends_pending--;
        suspends_pending--;
    } else
        queue_event(APM_USER_SUSPEND, as);
    if (suspends_pending <= 0) {
        return suspend(1);
    } else {
        as->suspend_wait = 1;
        wait_event_interruptible(apm_suspend_waitqueue,
                               as->suspend_wait == 0);
        return as->suspend_result;
    }
    break;
default:
    return -EINVAL;
}
return 0;
}

```

코드 599. do_ioctl() 함수의 정의

do_ioctl() 함수는 먼저 `apm_user` 구조체를 얻기위해서 `file` 구조체의 `private_data`를 이용한다. 그리고, 이 구조체가 유효한가를 확인한 후(`check_apm_user()`), 그렇지 않다면 `-EIO`를 돌려준다. 만약 현재 APM 모듈의 사용자가 `superuser`가 아니거나 `writer`가 아니라면²²⁹ `-EPERM`(Permission Error)값을 돌려준다. 나머지 과정은 사용자 application에서 내린 명령에 따라 수행하는 부분으로 구분된다.

크게 사용자가 내리는 명령은 `APM_IOC_STANDBY`와 `APM_IOC_SUSPEND`가 있다. 나머지 경우에 대해서는 `-EINVAL`(Error Invalid)을 돌려준다. `APM_IOC_STANDBY`를 받았을 경우에는 `apm_user` 구조체에 `standbys_read`가 있는지를 확인한다. 만약 0이상의 값을 가진다면, `standbys_read`와 `standbys_pending` 및 전역 변수인 `standbys_pending`을 1씩 감소시키고, 그렇지 않다면, `APM_USER_STANDBY` 요청을 `apm_user`의 `event queue`에 넣어준다(`queue_event()`). 이젠 이렇게 해서 변경될 수 있는 전역 변수인 `standbys_pending`이 0이하의 값을 가지는 가를 확인한다. 만약 그렇다면, `standby()`를 호출해서 시스템을 대기(stanby) 모드의 전원관리로 상태를 바꾼다. `APM_IOC_SUSPEND` 명령이 내려졌을 경우에는 `suspends_read`가 0이상인 값을 가지고 있는지를 확인해서 `suspends_read`나 `suspends_pending`, 전역 변수인 `suspends_reading`의 값을 1씩 감소시키고, 그렇지 않을 경우에는 `APM_USER_SUSPEND`를 `apm_user`의 `event queue`에 저장한다(`queue_event()`). 만약 전역 변수인 `suspends_pending`이 0이하의 값을 가진다면, `suspend()` 함수에 1을 넘겨주고 호출한 결과값을 복귀값을 돌려준다. 0이하의 값을 `suspends_pending`이 가지지 않는다면, `apm_user` 구조체의 `suspend_wait`에 1을 설정하고, `wait_event_interruptible()`를 호출해서 현재 `process`가 `suspend_wait`가 0이 되는 `event`가 발생할 때까지 `apm_suspend_waitqueue`에서 대기(wait)하도록 만든다. 물론 이것은 `sleep`의 한 방법으로 특정 `event`에 대해서 `interruptible sleep`을 하는 경우에 해당한다. 이때의 결과 값은 `suspend_result`라는 `apm_user` 필드의 값이 될 것이다.

²²⁹ 즉, `writer`로 APM 디바이스 모듈을 열지 않았다면 이에 해당한다.

```

static void queue_event(apm_event_t event, struct apm_user *sender)
{
    struct apm_user * as;

    if (user_list == NULL)
        return;
    for (as = user_list; as != NULL; as = as->next) {
        if ((as == sender) || (!as->reader))
            continue;
        as->event_head = (as->event_head + 1) % APM_MAX_EVENTS;
        if (as->event_head == as->event_tail) {
            static int notified;

            if (notified++ == 0)
                printk(KERN_ERR "apm: an event queue overflowed\n");
            as->event_tail = (as->event_tail + 1) % APM_MAX_EVENTS;
        }
        as->events[as->event_head] = event;
        if ((!as->suser) || (!as->writer))
            continue;
        switch (event) {
        case APM_SYS_SUSPEND:
        case APM_USER_SUSPEND:
            as->suspends_pending++;
            suspends_pending++;
            break;

        case APM_SYS_STANDBY:
        case APM_USER_STANDBY:
            as->standbys_pending++;
            standbys_pending++;
            break;
        }
    }
    wake_up_interruptible(&apm_waitqueue);
}

```

코드 600. queue_event() 함수의 정의

queue_event()는 apm_user 구조체의 event queue에 넘겨받은 event를 queueing하는 일을 처리한다. 만약 APM 사용자의 리스트(user_list)가 NULL을 가진다면 그냥 복귀 한다. 이전 해당 apm_user가 있는가를 확인하고, event queue에 event를 넣는 일을 한다. 이 과정은 user_list를 처음부터 시작해서 순차적으로 loop를 돌면서 찾아나가는 구조로 되어 있다. 즉, 넘겨받은 apm_user구조체(sender)와 user_list의 엔트리가 같거나, reader가 아닌 경우에는 해당 apm_user의 event queue에 event를 queueing할 필요가 없기에 다음 번 loop로 돌아간다. 그렇지 않다면, 다음과 같은 일을 처리한다. 즉, user_list에 있는 다른 apm_user구조체들의 event queue에 event를 queueing하기 위해서 event_head는 1을 증가시키고, 만약 event_head와 event_tail이 같다면, event queue가 FULL인 상태이므로, overflow가 발생했다는 것을 알리기 위해서 notified를 1증가시킨다. overflow를 회복하기 위해, 하나의 공간을 주기 위해서, 다시 event_tail을 1증가시킨다. 이전 해당 apm_user의 event queue인 events[] 배열에 넘겨받은 event를 넣어준다. 이때 현재 사용자가 super user가 아니거나 writer가 아니라면, 다시 다음번 loop로 이동한다. 이전 넘겨받은 event가 어떤 것인가를 찾아서, 해당 변수들의 값을 update할 차례이다. APM_SYS_SUSPEND와 APM_USER_SUSPEND가 event로 주어졌을 경우에는 user_list에 있는 apm_user 구조체들의 suspends_pending과 전역변수 suspends_pending의 값을 증가시켜주며, APM_SYS_STANDBY나 APM_USER_STANDBY가 주어졌을 경우에는 standbys_pending과 전역 변수 standbys_pending의 값을 증가시킨다. 모든 apm_user 구조체들에 대해서 event queueing이 끝나면, apm_waitqueue에서 event가 발생하기를 대기(wait)하고 있는 프로세스들을 깨우기 위해서 wake_up_interruptible()를 호출한다.

9.6.5. Poll 함수의 분석

Poll함수는 디바이스에게 사용자가 읽기나 쓰기를 하기 위해서 준비가 되었는가를 물기 위해서 호출되는 함수이다. APM 모듈에 대해서는

```
static unsigned int do_poll(struct file *fp, poll_table * wait)
{
    struct apm_user * as;

    as = fp->private_data;
    if (check_apm_user(as, "poll"))
        return 0;
    poll_wait(fp, &apm_waitqueue, wait);
    if (!queue_empty(as))
        return POLLIN | POLLRDNORM;
    return 0;
}
```

코드 601. do_poll() 함수의 정의

do_poll() 함수에는 file 구조체의 private_data영역에 저장된 apm_user구조체의 포인터를 얻어서 사용한다. check_apm_user() 함수는 apm_user 구조체에 저장된 자료구조에 오류가 있는지를 확인하는 함수이며, poll_wait()를 호출해서, 현재 파일 구조체의 POLL table에 event를 알려줄 수 있는 queue를 더한다. queue_empty()는 현재 apm_user의 queue가 비었는가(tail과 head가 같은 위치를 가르킨다.)를 확인하는 함수이며, 비지 않아다면, POLLIN과 POLLRDNORM을 OR시킨 값을 돌려준다. 즉, 일반적인 읽기가 가능한 상태라는 것을 알려준다.

9.7. System IDLE 상태에서의 수행

시스템의 IDLE 프로세스는 수행해야하는 다른 프로세스가 없는 경우에 실행된다. ACPI(Advanced Configuration and Power Interface)를 사용하지 않고, APM을 사용한다면 APM 모듈에서 제공하는 IDLE 함수를 수행하게 될 것이다.

start_kernel()이 커널로의 진입점이라고 할 때, init 프로세스는 rest_init()라는 함수에서 커널의 thread로 생성된다. 이 커널 thread를 생성한 후, 곧바로 현재 프로세스가 scheduling이 되기를 요구한다는 need_resched 필드를 설정하고, 바로 cpu_idle() 함수를 호출하게 된다.

```
static void rest_init(void)
{
    kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
    unlock_kernel();
    current->need_resched = 1;
    cpu_idle();
}
```

코드 602. rest_init() 함수의 정의

rest_init() 함수는 init(PID 1)을 생성하는 일을 수행하고, 커널에 설정된 lock을 해제하며, 커널에 scheduling을 요청한다(current->need_resched = 1). 이는 현재 이미 init(PID 0)가 IDLE thread로서 생성이 된 상황이며, cpu_idle() 함수는 이 IDLE thread의 context에서 수행되는 것이다.

```
/*
 * The idle thread. There's no useful work to be
 * done, so just try to conserve power and have a
 * low exit latency (ie sit in a loop waiting for
 * somebody to say that they'd like to reschedule)
 */
void cpu_idle (void)
```

```
{
    /* endless idle loop with no priority at all */
    init_idle();
    current->nice = 20;
    current->counter = -100;

    while (1) {
        void (*idle)(void) = pm_idle;
        if (!idle)
            idle = default_idle;
        while (!current->need_resched)
            idle();
        schedule();
        check_pgt_cache();
    }
}
```

코드 603. cpu_idle() 함수의 정의

cpu_idle() 함수는 각각의 architecture마다 다르게 가질 수 있다. 여기서 보는 것은 i386 architecture에서의 cpu_idle() 함수다. init_idle() 함수는 현재 프로세스(IDLE Process)를 RUN QUEUE에서 제거하고, 스케줄링에 필요한 데이터를 만들어주는 일을 한다. IDLE process는 이제 자신의 스케줄링 우선 순위 계산에 필요한 nice값과 counter값을 다시 설정하게되는데, 이때 NICE값²³⁰으로는 20을 counter값²³¹은 -100을 넣도록 한다. 즉, 일반적인 프로세스들보다는 우선 순위에 있어서 밀리게 된다.

나머지는 IDLE process가 무한 loop를 돌면서 수행하는 부분이다. 여기서 우리가 APM과 관련되어 중점적으로 봐야하는 것이 바로, pm_idle이라는 변수이다. 이 변수는 IDLE 함수를 가르키는 포인터로서 정의는 ~/linux/arch/i386/kernel/process.c에 정의되어 있는 전역 변수로서, 위의 코드에서는 이 pm_idle에 Power Management에서 사용하는 IDLE 함수가 정의된 경우에는 이것을 사용하고, 그렇지 않을 경우에는 default_idle이 가르키는 함수를 사용하도록 하고 있다.

이때, while() loop를 돌면서 시스템에 스케줄링을 요하는 작업이 발생할 때까지 계속 무한 loop를 돌면서 대기하게 된다. 만약 스케줄링을 요구하는 작업이 발생한다면(current->need_resched == 1), schedule() 함수를 호출해서 새로운 프로세스를 수행하도록 요청한다. check_pgt_cache()함수²³²는 페이지 테이블의 cached가 특정한 값 범위에 항상 있도록 만들어 주는 함수이다.

```
void __init init_idle(void)
{
    struct schedule_data * sched_data;
    sched_data = &aligned_data[smp_processor_id()].schedule_data;

    if (current != &init_task && task_on_rqueue(current)) {
        printk("UGH! (%d:%d) was on the rqueue, removing.\n",
               smp_processor_id(), current->pid);
        del_from_rqueue(current);
    }
    sched_data->curr = current;
```

²³⁰ NICE라는 말은 다른 프로세스들의 입장에서 보았을 때, 이 프로세스가 자신의 우선 순위를 낮춘다는 뜻에서 좋다(nice)라는 것이다. 이 NICE가 일반적으로 양수값을 가지면 우선 순위는 낮아지게되며, 음수 값을 가지게 되면 우선 순위는 올라간다. 그리고, -20에서 20사이의 값을 사용한다.

²³¹ COUNTER값은 우선 순위계산 더 높은 값을 가질 수록 더 높은 우선순위를 가진다고 본다. 따라서, 음수 값을 가질 경우에는 우선 순위가 낮아지는 현상이 발생한다. IDLE 프로세스는 다른 실행준비된 프로세스가 없을 경우에 실행되므로, 우선 순위는 될 수 있으면 낮은 것이 좋다.

²³² check_pgt_cache()함수는 ~/linux/mm/memory.c에 있으며, 이 함수 내부에서 다시 do_check_pgt_cache()를 호출해서 적절한 값 범위에 페이지 테이블에 대한 cache의 free한 엔트리가 얼마나 있는지를 알려준다. 만약 특정 범위를 넘어서는 값을 가진다면, 범위안에 오도록 만들어 주는 역할도 수행한다.

```

    sched_data->last_schedule = get_cycles();
    clear_bit(current->processor, &wait_init_idle);
}

```

코드 604. init_idle() 함수의 정의

init_idle() 함수는 현재 수행중인 프로세스가 동작중인 프로세서(processor : CPU)의 스케줄링 데이터를 sched_data가 가르키게 하고, 현재 프로세스가 init_task가 아니고 RUN queue에 있다면(task_on_runqueue()), IDLE 프로세스를 RUN queue에서 제거하기 위해서 del_from_runqueue()를 호출한다. 이하는 현재 CPU의 스케줄링 정보를 update하는 일이다. IDLE 프로세스가 수행중인 프로세서(CPU)에 대해서는 wait_init_idle의 해당 bit을 지워준다(clear). 즉, IDLE 프로세스가 초기화 대기를 기다리는 다른 작업들이 진행할 수 있도록 만들어주는 것이다.

```

void default_idle(void)
{
    if (current_cpu_data.hlt_works_ok && !hlt_counter) {
        __cli();
        if (!current->need_resched)
            safe_halt();
        else
            __sti();
    }
}

```

코드 605. default_idle() 함수의 정의

default_idle() 함수는 current_cpu_data.hlt_works_ok가 설정되어 있고, hlt_counter가 0인 경우에 대해서만 동작한다. 즉, __cli()를 호출해서 인터럽트의 생성을 불가(disable)로 만들고, 현재 프로세스가 스케줄링을 요구하지 않는다면(current->need_resched == 0), safe_halt()²³³를 호출한다. 그렇지 않다면, 다시 인터럽트를 enable시키기 위해서 __sti()를 호출한다.

결과적으로 cpu_idle()이라는 함수를 통해서 우린 pm_idle이란 변수에 설정되는 APM의 IDLE function을 분석하면 IDLE process에서 수행하는 PM을 알 수 있을 것이다. 이미 앞에서 apm_init()함수를 보면서, pm_idle이라는 변수에 apm_cpu_idle이라는 함수의 주소가 들어간다는 보았다..

```

/***
 * apm_cpu_idle          -      cpu idling for APM capable Linux
 *
 * This is the idling function the kernel executes when APM is available. It
 * tries to do BIOS powermanagement based on the average system idle time.
 * Furthermore it calls the system default idle routine.
 */

static void apm_cpu_idle(void)
{
    static int use_apm_idle; /* = 0 */
    static unsigned int last_jiffies; /* = 0 */
    static unsigned int last_stime; /* = 0 */

    int apm_idle_done = 0;
    unsigned int jiffies_since_last_check = jiffies - last_jiffies;
}

```

²³³ ~/linux/include/asm-i386/system.h에 "#define safe_halt() __asm__ __volatile__("sti; hlt": : : "memory")"로 정의된 매크로이다. 즉, 인터럽트를 enable 시키고, hlt 인스트럭션(instruction)을 수행한다. 시스템의 상태는 halt(정지) 상태가 되지만, 인터럽트등과 같은 것으로 시스템을 다시 수행할 수 있도록 할 수 있다.

```

unsigned int bucket;

recalc:
    if (jiffies_since_last_check > IDLE_CALC_LIMIT) {
        use_apm_idle = 0;
        last_jiffies = jiffies;
        last_stime = current->times.tms_stime;
    } else if (jiffies_since_last_check > idle_period) {
        unsigned int idle_percentage;

        idle_percentage = current->times.tms_stime - last_stime;
        idle_percentage *= 100;
        idle_percentage /= jiffies_since_last_check;
        use_apm_idle = (idle_percentage > idle_threshold);
        last_jiffies = jiffies;
        last_stime = current->times.tms_stime;
    }
}

```

코드 606. `apm_cpu_idle()` 함수의 정의

`apm_cpu_idle()` 함수는 IDLE 프로세스에 의해서 불려지는 함수이다. 즉, 시스템의 IDLE 프로세스의 context상에서 수행된다고 보면 될 것이다. `jiffies_since_last_check`는 마지막으로 `apm_cpu_idle`을 수행한 시간과 현재 시간²³⁴과의 차이 값을 가진다. 만약 이 값이 `IDLE_CALC_LIMIT`(= HZ X 100 = 100 X 100 = 10000)보다 크다면, 즉 10000 X 10 ms(= 10 secs)보다 큰 값을 가진다면, `use_apm_idle` 값을 0으로 만들고, `last_jiffies`에는 현재 `jiffies`값을 넣어주게 되며, `last_stime`에는 현재 프로세스(IDLE process)가 가진 시스템내에서 소비한 시간을 넣어주게된다. 그렇지 않다면, `idle_period`²³⁵ 와 `jiffies_since_last_check`를 비교한다. 만약 더 큰 값을 가진다면, 현재 프로세스의 시스템내에서 수행된 시간과 마지막에 수행되었던 시간과의 차이를 구해서 이를 퍼센트(%)화 시킨다. 물론 이때는 `jiffies_since_last_check` 중에서 얼마나 많은 시간을 IDLE 프로세스가 시스템 내에서 소모했는가를 계산하는 과정이다. 이렇게 해서 이 값이 `idle_threshold`²³⁶ 보다 큰 값을 가진다면, `use_apm_idle`을 설정한다(set). 다음번의 새로운 계산을 위해서 `last_jiffies`에는 이제 현재 `jiffies`를 넣고, `last_stime`에도 현재 프로세스의 `stime`을 넣어주게 된다.

```

bucket = IDLE_LEAKY_MAX;
while (!current->need_resched) {
    if (use_apm_idle) {
        unsigned int t;

        t = jiffies;
        switch (apm_do_idle()) {
            case 0: apm_idle_done = 1;
                if (t != jiffies) {
                    if (bucket) {
                        bucket = IDLE_LEAKY_MAX;
                        continue;
                    }
                } else if (bucket) {
                    bucket--;
                    continue;
                }
                break;
            case 1: apm_idle_done = 1;
        }
    }
}

```

²³⁴ 여기서의 시간은 `jiffies`가 가지는 단위가 될 것이다. 대략 10ms가 1 `jiffies`라고 보면 된다.

²³⁵ 명시되지 않아다면, `DEFAULT_IDLE_PERIOD`(= 100/3)를 가질 것이다.

²³⁶ 명시되지 않은 경우에 95나 100의 값을 가진다.

```

        break;
    default: /* BIOS refused */
    }
}
if (original_pm_idle)
    original_pm_idle();
else
    default_idle();
jiffies_since_last_check = jiffies - last_jiffies;
if (jiffies_since_last_check > idle_period)
    goto recalc;
}
if (apm_idle_done)
    apm_do_busy();
}

```

코드 607. **apm_cpu_idle()** 함수의 정의(계속)

IDLE_LEAKY_MAX(= 16)은 얼마나 시간이 흘러갔나를 확인하기 위해서 사용한다. while() loop로 들어갈 때, 현재 프로세스가 새로운 스케줄링을 요구하지 않는 동안 계속 수행되도록 만들고, use_apm_idle이 설정된 경우에 대해서 다음과 같은 일을 한다. 먼저 t에는 jiffies(현재 시스템의 시간)값을 저장하고, apm_do_idle()을 호출한다. 이 호출의 결과가 0일 경우에는 apm_do_idle()함수를 제대로 수행했으며, clock 영향을 주지 않았다는 뜻이 되므로, t가 jiffies와 같지 않을 때 bucket이 0이 아닌 값을 가지는 경우에 대해서 bucket에는 IDLE_LEAKY_MAX를 주고 다음번의 loop를 수행하도록 한다. 만약 t가 jiffies와 같다면, 시간이 조금 흘렀다는 말이 되며, bucket에 0이 아닌 값이 있다면, 이 값을 감소시키고 다음번 loop로 들어가게 된다. 만약 1이란 값을 apm_do_idle()이 돌려준다면, 이때는 APM BIOS의 IDLE을 수행하며, 시간이 느려졌다는 뜻으로 사용한다. 단순히 apm_idle_done을 1로 설정한다. 만약 그외에 값을 apm_do_idle() 함수가 돌려준다면, 이는 APM BIOS function 호출에 오류가 있었다는 뜻이되며, 이때는 그냥 빠져나오도록 한다. 만약 이전에 정의한 IDLE 함수가 존재한다면(original_pm_idle : 저장해 둔 IDLE function)을 수행하고, 그렇지 않다면 default_idle() 함수를 수행하도록 한다. 이는 ACPI와 같은 것을 고려한 것이라고 볼 수 있다. 즉, ACPI와 APM을 동시에 사용할 수도 있다는 말이다.

이젠 jiffies_since_last_check를 최근 값으로 업데이트(update)한다. 만약 이 값이 idle_period보다 크다면, 제어는 recalc로 다시 넘어가서 앞에서 했던 일을 반복적으로 수행할 것이다. while() loop를 빠져나오는 방법은 어쨌든 시스템에 rescheduling을 해주어야하는 event가 발생하는 길 밖에 없다. 만약 이러한 event가 발생한다면, apm_idle_done이 이 설정되어 있다면 apm_do_idle()을 호출할 하고, 그렇지 않다면 함수는 복귀하게 될 것이다.

```

/**
 *      apm_do_idle      -      perform power saving
 *
 *      This function notifies the BIOS that the processor is (in the view
 *      of the OS) idle. It returns -1 in the event that the BIOS refuses
 *      to handle the idle request. On a success the function returns 1
 *      if the BIOS did clock slowing or 0 otherwise.
 */
static int apm_do_idle(void)
{
    u32      eax;

    if (apm_bios_call_simple(APM_FUNC_IDLE, 0, 0, &eax)) {
        static unsigned long t;

        if (time_after(jiffies, t + 10 * HZ)) {
            printk(KERN_DEBUG "apm_do_idle failed (%d)\n",
                   (eax >> 8) & 0xff);
            t = jiffies;
        }
    }
}

```

```

        }
        return -1;
    }
    clock_slowed = (apm_info.bios.flags & APM_IDLE_SLOWS_CLOCK) != 0;
    return clock_slowed;
}

```

코드 608. `apm_do_idle()` 함수의 정의

`apm_do_idle()` 함수는 APM BIOS에 있는 IDLE 함수를 수행하는 역할을 한다. 즉, `apm_bios_call_simple()`에 `APM_FUNC_IDLE`을 전달해서 호출한다. 이때 오류가 있었다면, `if`절을 수행한다. `time_after()`²³⁷은 단순히 뒤의 값에서 앞의 값을 빼서 0과 비교한 결과를 알려주는 매크로이다. `t`는 static 변수로 정의되었기 때문에 초기 값으로 0을 가질 것이다. 따라서, 처음에는 이 값에 10 X HZ한 값을 더해서 `jiffies`와 비교해본다. 만약 `jiffies`가 `t + 10 x HZ`보다 큰 값을 가진다면, `apm_do_idle()`이 실패한 것으로 생각한다. 오류 코드는 `apm_bios_call_simple()` 함수의 APM BIOS function 호출의 결과 값이 저장된 `eax` 레지스터를 8 bit 우측으로 SHIFT한 값²³⁸을 사용한다. 그리고나서, `t`값을 `jiffies`값으로 변경한다. 복귀 값은 -1을 가질 것이다. 만약 오류가 없이 `apm_bios_call_simple()`을 호출했다면, `clock_slowed`에는 `apm_info` 구조체에 `APM_IDLE_SLOWS_CLOCK`이 0인가를 확인해서 0이라면 1을 돌려주고, 그렇지 않다면 0을 돌려준다. 여기서 한가지 중요한 보고넘어가야 할 점은 `jiffies`값과 `t + 10 x HZ`를 비교하는 부분이다. 즉, 시스템이 BIOS의 IDLE 함수를 제대로 수행했다면, `jiffies`와 같은 값을 증가시켜주지 않을 것이다. 즉, 이 시간과 BIOS를 호출하는데 걸리는 시간상의 차이를 보고, 예러가 정말 발생했는가 그렇지 않은가를 확인하고 있는 것이다. `clock_slowed`는 APM BIOS가 시스템의 `clock`까지도 멈추게 했는가를 알기 위해서 사용한다.

```

/**
 *      apm_do_busy      -      inform the BIOS the CPU is busy
 *
 *      Request that the BIOS brings the CPU back to full performance.
 */
static void apm_do_busy(void)
{
    u32      dummy;

    if (clock_slowed || ALWAYS_CALL_BUSY) {
        (void) apm_bios_call_simple(APM_FUNC_BUSY, 0, 0, &dummy);
        clock_slowed = 0;
    }
}

```

코드 609. `apm_do_busy()` 함수의 정의

`apm_do_busy()` 함수는 `clock_slowed` 값이 설정되어 있거나, 혹은 `ALWAYS_CALL_BUSY`가 있는 경우에 한해서 호출된다. 이 함수는 BIOS에 CPU가 바쁘다(busy)는 것을 알리기 위한 것으로 BIOS가 CPU를 다시 완전 동작 상태로 바꾸도록 요청한다. 호출하는 APM BIOS function은 `APM_FUNC_BUSY`이다. 호출후에는 `clock_slowed`를 0으로 바꾼다.

9.8. APM Kernel Daemon

APM Kernel Daemon도 역시 `apm_init()`에서 커널 thread로 생성된다. APM Kernel Daemon은 주기적으로 깨어나서 APM 관련 event를 처리하는 것을 담당한다.

```

static int apm(void *unused)
{

```

²³⁷ `time_after()` 매크로는 `~linux/include/linux/time.h`에 `((long)(b) - (long)(a) < 0)`라고 정의되어 있다.

²³⁸ Error 코드가 AH레지스터에 설정되어 넘어오기 때문이다.

```

unsigned short bx;
unsigned short cx;
unsigned short dx;
int error;
char * power_stat;
char * bat_stat;

kapmd_running = 1;
daemonize();
strcpy(current->comm, "kapmd");
sigfillset(&current->blocked);
if (apm_info.connection_version == 0) {
    apm_info.connection_version = apm_info.bios.version;
    if (apm_info.connection_version > 0x100) {
        if (apm_info.connection_version > 0x0102)
            apm_info.connection_version = 0x0102;
        error = apm_driver_version(&apm_info.connection_version);
        if (error != APM_SUCCESS) {
            apm_error("driver version", error);
            apm_info.connection_version = 0x100;
        }
    }
}
if (debug)
    printk(KERN_INFO "apm: Connection version %d.%d\n",
           (apm_info.connection_version >> 8) & 0xff,
           apm_info.connection_version & 0xff);

```

코드 610. apm() 함수의 정의

apm() 함수는 먼저 APM 데몬이 이전 동작한다는 것을 알려주기 위해서 kapmd_running flag를 1로 설정한다. daemonize()라는 함수는 현재 프로세스(APM 커널 thread)를 데몬²³⁹으로 만들어주기 위한 일을 처리하며, 이 프로세스가 가지는 comm이란 필드에 ‘kapmd’를 넣어서, 현재 실행중인 프로세스가 무엇인가를 나타내준다.²⁴⁰ sigfillset()은 inline으로 정의된 함수로 현재 실행중인 프로세스가 어떤 signal도 받지 않도록 만들기 위해서 blocked(blocking 할 signal의 집합)를 전부 1로 설정한다. apm_info 구조체의 connection_version이 00이라면, APM BIOS의 version으로 connection_version을 설정하고, 만약 이 값이 0x100보다 큰 경우, 0x0102(APM BIOS version 1.2)보다 크다면 0x0102로 수정하도록 한다. 현재 APM 데몬이 지원하는 APM BIOS 버전은 1.2까지라는 뜻이다. APM BIOS의 드라이버 버전을 구하기 위해서 apm_driver_version()을 호출한다. 에러가 발생했다면(!= APM_SUCCESS), apm_error()을 호출하고, connection_version을 0x100으로 설정한다. APM 드라이버의 version function은 반드시 APM BIOS와의 connection이 설정(establish)된 후에 호출되어야 하며, APM version 1.1 이상의 버전에서만 사용할 수 있다. 따라서, 오류가 발생한다면 APM version 1.0이 된다. 나머지는 debugging 메시지를 출력하는 부분이다.

```

static int __init apm_driver_version(u_short *val)
{
    u32 eax;

    if (apm_bios_call_simple(APM_FUNC_VERSION, 0, *val, &eax))
        return (eax >> 8) & 0xff;
    *val = eax;
    return APM_SUCCESS;
}

```

²³⁹ 일반적으로 데몬(Daemon)은 시스템의 background로 수행되는 프로세스로 시스템의 관리기능을 수행하는 프로세스이다.

²⁴⁰ “ps” 명령으로 kapmd를 찾을 수 있을 것이다.

코드 611. `apm_driver_version()` 함수의 정의

`apm_driver_version()` 함수는 `apm_bios_call_simple()` 함수를 이용해서 `APM_FUNC_VERSION` function을 호출한다. 오류가 없었다면 `APM_SUCCESS`를 돌려줄 것이지만, 그렇지 않다면 `eax`의 값중에서 AH레지스터에 해당 오류 코드를 넣어서 보여줄 것이다. 성공적인 호출에서는 `val`에 AH와 AL레지스터를 이용해서 APM connection의 major와 minor 번호를 각각 넘겨줄 것이다.

```
#ifdef CONFIG_APM_DO_ENABLE
    if (apm_info.bios.flags & APM_BIOS_DISABLED) {
        error = apm_enable_power_management(1);
        if (error) {
            apm_error("enable power management", error);
            return -1;
        }
    }
#endif
if ((apm_info.bios.flags & APM_BIOS_DISENGAGED)
    && (apm_info.connection_version > 0x0100)) {
    error = apm_engage_power_management(APM_DEVICE_ALL, 1);
    if (error) {
        apm_error("engage power management", error);
        return -1;
    }
}
```

코드 612 `apm()` 함수의 정의(계속)

만약 `CONFIG_APM_DO_ENABLE`이 컴파일 옵션으로 주어져 있다면, booting시에 APM을 enable시키라는 뜻이므로, 현재 APM BIOS가 disable이 되어 있는지를 확인한 후, `apm_enable_power_management()` 함수를 호출해서 전원 관리를 시작하도록 만든다. 이 역시 APM 데몬이 처음 시작되는 순간에 실행될 것이므로, 시스템이 booting하거나 APM 모듈이 로딩되는 시간에 일어날 것이다. 에러가 있었다면 `apm_error()`가 처리하고 -1을 돌려주도록 한다. 만약 `APM_BIOS_DISENGAGED`가 BIOS의 flag에 설정되어 있고, APM connection version이 1.0(= 0x0100)보다 크다면, `apm_engage_power_management()`를 호출해서 시스템이나 디바이스가 전원 관리에 관여하도록 만든다. 이렇게 하지않으면, APM BIOS에서 자동적으로 시스템이나 디바이스의 전원을 관리하게 될 것이다. 시스템이 reset되었을 때에는 APM이 지원되는 디바이스들은 모두 disengaged 상태로 남아 있게된다. 따라서, 이때 모든 디바이스들과 시스템이 전원관리에 동참하도록 만들어주는 과정이다.

```
if (debug && (smp_num_cpus == 1)) {
    error = apm_get_power_status(&bx, &cx, &dx);
    if (error)
        printk(KERN_INFO "apm: power status not available\n");
    else {
        switch ((bx >> 8) & 0xff) {
        case 0: power_stat = "off line"; break;
        case 1: power_stat = "on line"; break;
        case 2: power_stat = "on backup power"; break;
        default: power_stat = "unknown"; break;
        }
        switch (bx & 0xff) {
        case 0: bat_stat = "high"; break;
        case 1: bat_stat = "low"; break;
        case 2: bat_stat = "critical"; break;
        case 3: bat_stat = "charging"; break;
        default: bat_stat = "unknown"; break;
        }
    }
}
```

```

        printk(KERN_INFO
            "apm: AC %s, battery status %s, battery life ",
            power_stat, bat_stat);
        if ((cx & 0xff) == 0xff)
            printk("unknown\n");
        else
            printk("%d%%\n", cx & 0xff);
        if (apm_info.connection_version > 0x100) {
            printk(KERN_INFO
                "apm: battery flag 0x%02x, battery life ",
                (cx >> 8) & 0xff);
            if (dx == 0xffff)
                printk("unknown\n");
            else
                printk("%d %s\n", dx & 0x7fff,
                    (dx & 0x8000) ?
                    "minutes" : "seconds");
        }
    }
}

```

코드 613. `apm()` 함수의 정의(계속)

debug이 정의되어 있고, CPU의 개수가 1개라면, 디버깅을 위한 정보를 보여준다. `apm_get_power_status()`는 현재 시스템의 전원 상태를 알기위해서 호출하는 함수이다. 이 함수의 파라미터로서 주어지는 bx, cx, dx는 호출의 결과 값을 가지게 된다. 에러가 발생했다면, 시스템의 전원 상태를 알 수 없다는 메시지를 출력하고, 그렇지 않다면 호출의 결과 값에 맞는 출력을 보여준다. 이때 오류가 없었다면, BH는 AC line의 상태를, BL은 battery의 상태를, CH는 battery flag를 가지며, CL은 남아있는 battery의 시간(life : percentage of charge)을, DX에는 시간단위(second나 minute)의 남아 있는 battery 시간(life)을 가진다. 따라서, 이를 각각에 대한 정보를 잘 분석해서 보여주는 일이 나머지 과정이다. 여기서 CX 값이 만약 0xFF라는 값을 가진다면, 이때는 unknown값으로 설정된 것을 의미하며, connection version이 0x100(1.0)보다 큰 경우에 대해서만 battery flag가 유효한 값을 가진다. 또한 DX 값이 0xFFFF라는 값을 가지는 경우도 역시 unknown으로 처리한다.²⁴¹

```

static int apm_get_power_status(u_short *status, u_short *bat, u_short *life)
{
    u32     eax;
    u32     ebx;
    u32     ecx;
    u32     edx;
    u32     dummy;

    if (apm_info.get_power_status_broken)
        return APM_32_UNSUPPORTED;
    if (apm_bios_call(APM_FUNC_GET_STATUS, APM_DEVICE_ALL, 0,
                      &eax, &ebx, &ecx, &edx, &dummy))
        return (eax >> 8) & 0xff;
    *status = ebx;
    *bat = ecx;
    if (apm_info.get_power_status_swabinminutes) {
        *life = swab16((u16)edx);
        *life |= 0x8000;
    } else
        *life = edx;
    return APM_SUCCESS;
}

```

²⁴¹ APM BIOS version 1.2 Spec.에서 APM function 0x0A에 Get Power Status를 참조하기 바란다.

}

코드 614. `apm_get_power_status()` 함수의 정의

먼저 `apm_info` 구조체에 `get_power_status_broken` 필드가 설정되어 있는지 확인한다. 만약 설정되어 있다면, 시스템의 전원 상태 정보를 구할 수 없다는 뜻이되며, 따라서, `APM_32_UNSUPPORTED`라는 값을 돌려준다. 즉, 32 bit 연산을 APM BIOS에서 지원하고 있지 않는다는 것을 나타낸다. 그렇지 않다면, `apm_bios_call()` 함수를 호출해서 시스템의 전원 상태(`APM_FUNC_GET_STATUS`)를 가져온다. 이때 복귀 값(Carry Flag)이 1로 설정되어 있다면, EAX중 AH의 값을 취해서 에러 코드로 돌려준다. 발생할 수 있는 에러에는 Unrecognized Device ID(=09H), Parameter Value out of Range(=0AH), APM not Present(=86H) 등이 있다. 돌려주는 값들은 각각이 해당하는 자리에 들어갈 수 있도록 만든다. `status`에는 `ebx`를, `bat`에는 `ecx`를 넣고, `get_power_status_swabinminutes`가 `apm_info`구조체에 설정되어 있을 경우에는 `edx`의 내용을 `swab16()`을 통해서 변환해서 `life`에 설정하고, 분(minutes)단위 시간을 명시하기 위해서 0x8000을 OR시켜준다. 그렇지 않다면, `edx`를 그냥 `life`에 준다. 복귀 코드는 `APM_SUCCESS`가 될 것이다.

```
/* Install our power off handler.. */
if (power_off)
    pm_power_off = apm_power_off;
register_sysrq_key('o', &sysrq_poweroff_op);
if (smp_num_cpus == 1) {
#endif
    if defined(CONFIG_APM_DISPLAY_BLANK) && defined(CONFIG_VT)
        console_blank_hook = apm_console_blank;
#endif
    apm_mainloop();
#ifndef CONFIG_APM_DISPLAY_BLANK
    console_blank_hook = NULL;
#endif
}
kapmd_running = 0;
return 0;
}
```

코드 615. `apm()` 함수의 정의(계속)

`power_off`라는 flag가 설정되어 있는 경우에는 `pm_power_off`라는 시스템의 전역 변수에 `apm_power_off`를 넣어주도록 한다. 이 변수에 설정된 함수는 나중에 `machine_power_off()`²⁴²라는 함수를 통해서 시스템이 rebooting이 될 때 수행될 것이다.

`register_sysrq_key()` 함수는 `CONFIG_MAGIC_SYSRQ`가 커널 컴파일 옵션으로 선언된 경우에만 동작하는 키들을 등록하는 inline함수(혹은 매크로)이다. 즉, PC의 경우 “Sys Rq”라는 키와 다른 키의 조합으로 동작하는 키를 등록한다. 만약 CPU의 개수가 하나라면, `CONFIG_APM_DISPLAY_BLANK`와 `CONFIG_VT`가 정의된 경우에 대해서 `console_blank_hook`에 `apm_console_blank`를 등록하려준다. `console_blank_hook`은

이전 드디어 APM 데몬이 수행할 main loop를 만나게 된다. APM 데몬은 시스템이 가동중인 동안에는 별다른 일이 없을때 이 loop에서 빠져나오지 않을 것이다. 만약 빠져나온다면, 앞에서 설정했을 가능성이 있는 `console_blank_hook`을 다시 NULL로 만들어준다. 마지막으로 커널 APM 데몬 thread가 가동중이지 않다는 뜻으로 `kapmd_running` flag를 0으로 설정하고, 0을 돌려주면서 복귀한다.

```
static void apm_power_off(void)
{
    unsigned char po_bios_call[] = {
```

²⁴² `machine_power_off()` 함수는 `~/linux/arch/i386/kernel/process.c`에 정의되어 있으며, `sys_reboot()`라는 함수에서 호출된다. `sys_reboot()` 함수는 Linux에서 제공하는 시스템 콜(88번으로 정의 : `~/linux/include/asm/unistd.h` 참조)로서, 정의는 `~/linux/kernel/sys.c`에 있으며, 사용자 프로세스에서 호출할 수 있다.

```

        0xb8, 0x00, 0x10, /* movw $0x1000,ax */
        0x8e, 0xd0,      /* movw ax,ss           */
        0xbc, 0x00, 0xf0, /* movw $0xf000,sp   */
        0xb8, 0x07, 0x53, /* movw $0x5307,ax   */
        0xbb, 0x01, 0x00, /* movw $0x0001,bx   */
        0xb9, 0x03, 0x00, /* movw $0x0003,cx   */
        0xcd, 0x15         /* int $0x15          */

};

#endif CONFIG_SMP
/* Some bioses don't like being called from CPU != 0 */
while (cpu_number_map(smp_processor_id()) != 0) {
    kernel_thread(apm_magic, NULL,
                  CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD);
    schedule();
}
#endif
if (apm_info.realmode_power_off)
    machine_real_restart(po_bios_call, sizeof(po_bios_call));
else
    (void) set_system_power_state(APM_STATE_OFF);
}

```

코드 616. `apm_power_off()` 함수의 정의

`apm_power_off()` 함수에서 한가지 재미있는 점은 `po_bios_call[]`이란 문자 배열에 직접 machine instruction을 넣어서 사용하고 있다는 점이다. SMP(Symmetric Multi-Processor) machine인 경우에는 현재 CPU의 ID를 구해서(`smp_processor_id()`), 0 값이 0이 아닌지를 확인한다(`cpu_number_map()`).²⁴³ 즉, 현재 `apm_power_off()` 함수를 수행하고 있는 CPU의 번호가 0번이 아닌 경우에는, 다시 커널 thread를 생성하는 것이다. 이때 생성되는 커널 thread가 수행할 일은 `apm_magic()`이라는 함수이다. 커널 thread가 생성되고난 다음에는 `schedule()`을 호출해서 CPU ID가 0인 것이 수행되도록 만들어 줄 것이다. 따라서, CPU ID 0이 power off를 완수 할 때까지 다른 CPU들은 전부 계속 scheduling만 요청하는 형태로 존재하게 될 것이다. `while()` loop를 빠져나왔다면, `apm_info` 구조체에 `realmode_power_off` flag가 설정된 경우에 `machine_real_restart()`을 호출한다. 그렇지 않다면, `set_system_power_state()`를 호출해서 시스템의 전원 상태를 `APM_STATE_OFF`로 만들도록 한다. `machine_real_restart()`²⁴⁴는 시스템을 real-mode로 바꾸고 전달된 코드(`po_bios_call`)를 수행하는 일을 한다. 이때 넘겨지는 code의 길이는 100 bytes 이하라고 가정한다.

```

#endif CONFIG_SMP
static int apm_magic(void * unused)
{
    while (1)
        schedule();
}
#endif

```

코드 617. `apm_magic()` 함수의 정의

`apm_magic()` 함수는 계속해서 `schedule()` 함수를 호출해서 다른 프로세스들이 수행되도록 만들어주는 함수(여기서는 thread)이다.

```

void handle_poweroff (int key, struct pt_regs *pt_regs, struct kbd_struct *kbd, struct tty_struct *tty)
{
    apm_power_off();
}

```

²⁴³ `cpu_number_map()`은 i386 architecture에서는 아무런 일도 하지 않는다. 단순히 넘겨받는 인자를 다시 돌려주기만 할 뿐이다.

²⁴⁴ `machine_real_restart()`은 `~/linux/arch/i386/kernel/process.c`에 정의되어 있다.

```

}

struct sysrq_key_op      sysrq_poweroff_op = {
    handler:        handle_poweroff,
    help_msg:       "Off",
    action_msg:     "Power Off\n"
};

```

코드 618. handle_poweroff() 함수의 정의

handle_poweroff() 함수는 “Sysrq” Key에 의해서 동작되는 전원 차단을 다루는 함수이다. 이 함수는 register_sysrq_key()에 의해서, 시스템 요청(request) 키에 반응하도록 만드는데, 이를 위해서 sysrq_key_op라는 구조체가 필요하다. 간략히 말한다면, 이 구조체는 키다 눌려졌을때 반응하는 핸들러 함수와 이 키와 관련된 도움(help) 말을 출력하기 위해서 필요한 문자열(help_msg), 그리고, 실제 반응이 시작될 때 출력할 문자열(action_msg)를 가진다. 따라서, handle_poweroff() 함수는 시스템 요청 키에 대해서 호출되는 함수이다. 이 함수는 다시 apm_power_off()를 호출한다.

```

#if defined(CONFIG_APM_DISPLAY_BLANK) && defined(CONFIG_VT)
static int apm_console_blank(int blank)
{
    int      error;
    u_short state;

    state = blank ? APM_STATE_STANDBY : APM_STATE_READY;
    /* Blank the first display device */
    error = set_power_state(0x100, state);
    if ((error != APM_SUCCESS) && (error != APM_NO_ERROR)) {
        /* try to blank them all instead */
        error = set_power_state(0x1ff, state);
        if ((error != APM_SUCCESS) && (error != APM_NO_ERROR))
            /* try to blank device one instead */
            error = set_power_state(0x101, state);
    }
    if ((error == APM_SUCCESS) || (error == APM_NO_ERROR))
        return 1;
    apm_error("set display", error);
    return 0;
}
#endif

```

코드 619. apm_console_blank() 함수의 정의

apm_console_blank() 함수는 전역변수로 정의된 console_blank_hook로 지정되는 함수이다. 즉, 이 함수는 console을 빙화면(blank)으로 만들기 위해서 호출되는 함수이다²⁴⁵. 이 함수에서 받는 값은 오로지 빙화면으로 만들것인지 말것인지를 결정하는 blank라는 flag값이다. 만약 blank에 1인 값을 넘겨받는다면, APM_STATE_STANDBY가, 그렇지 않다면 APM_STATE_READY가 state에 들어가게 되며, 이 설정된 state값에 따라서, set_power_state() 호출에서 시스템의 전원 상태를 결정해 줄 것이다. 이때 호출에서 사용하는 device ID가 set_power_state() 함수의 첫번째 파라미터로 주어지는데, 0x100은 display에 사용되는 첫번째 디바이스라는 것을 나타낸다. set_power_state() 함수의 복귀 값이 APM_SUCCESS가 아니고, APM_NO_ERROR도 아니라면, set_power_state()에서 오류가 발생한 것이기에, 모든 display에 관련된 디바이스에 대해서 전원 상태를 설정하기 위해서 0x1FF를 사용해서 set_power_state() 함수를 다시 한번 호출한다. 이때도 역시 오류가 발생한다면, 이전 두번째 display class에 속하는 디바이스를 그냥 끌려고

²⁴⁵ Console에서 주기적으로 timer를 설정해서, blank로 만들것인가를 결정하게 되는데, 이때 blank 시켜야 한다고 판단될 때 console_blank_hook에 설정된 함수를 호출하게된다. ~/linux/drivers/char/console.c를 참조하기 바란다.

시작한다. 오류가 없었다면 1을 돌려주지만, 오류가 발생한다면 `apm_error`를 호출해서 오류 메시지를 출력한다. 복귀 값은 0이다.

```
static void apm_mainloop(void)
{
    DECLARE_WAITQUEUE(wait, current);

    add_wait_queue(&apm_waitqueue, &wait);
    set_current_state(TASK_INTERRUPTIBLE);
    for (;;) {
        schedule_timeout(APM_CHECK_TIMEOUT);
        if (exit_kapmd)
            break;
        /*
         * Ok, check all events, check for idle (and mark us sleeping
         * so as not to count towards the load average).. .
         */
        set_current_state(TASK_INTERRUPTIBLE);
        apm_event_handler();
    }
    remove_wait_queue(&apm_waitqueue, &wait);
}
```

코드 620. `apm_mainloop()` 함수의 정의

`apm_mainloop()`는 APM 커널 데몬이 순환적으로 계속 수행하는 부분이다. 먼저 APM 데몬이 사용하는 `wait queue`를 정의한다(`DECLARE_WAITQUEUE()`). 현재 프로세스를 `wait queue`에 넣기 위해서 `add_wait_queue()`를 호출해서 `apm_waitqueue`에 `current task`(APM 데몬)를 넣도록 한다. 그리고, `task`의 상태는 `TASK_INTERRUPTIBLE`로 바꾼다. 이젠 `task`가 스케줄러에 의해서 scheduling이 되지는 않는다. 하지만, 수행은 여기서 멈추지 않고 계속 진행한다. `schedule_timeout()`을 호출할 때 `APM_CHECK_TIMEOUT`(= 100)을 넣어서, 1초(= $100 * 1 \text{ jiffies} = 1000 \text{ ms} = 1 \text{ sec}$) 후에는 깨어나도록 만든다. 이때 `schedule_timeout()` 내에서 스케줄러가 호출되면서 APM 커널 데몬은 실행을 잠시 멈출 것이다. 이젠 timeout이 발생했을 때 수행되는 부분이다. 만약 다시 깨어났을 때 `exit_kapmd`가 설정되어 있다면, 이는 APM 데몬을 더 이상 수행하지 말라는 뜻이므로, `for()` loop에서 나오게 된다. 그렇지 않다면, 다시 APM 커널 데몬의 상태를 `TASK_INTERRUPTIBLE`로 바꾸고, `apm_event_handler()`를 호출한다. 아직까지 `apm_wait_queue`에서 빠져나온 것은 아니다. 따라서, `for()` loop를 빠져나오게 되면, `remove_wait_queue()`를 호출해서 `wait queue`에서 APM 커널 데몬을 빼낸다.

```
static void apm_event_handler(void)
{
    static int pending_count = 4;
    int err;

    if ((standbys_pending > 0) || (susPENDS_pending > 0)) {
        if ((apm_info.connection_version > 0x100) &&
            (pending_count-- <= 0)) {
            pending_count = 4;
            if (debug)
                printk(KERN_DEBUG "apm: setting state busy\n");
            err = set_system_power_state(APM_STATE_BUSY);
            if (err)
                apm_error("busy", err);
        }
    } else
        pending_count = 4;
    check_events();
```

{}

코드 621. `apm_event_handler()` 함수의 정의

APM 커널 데몬은 깨어나게되면, pending되어 있는 APM 관련 event는 처리하려고 할 것이다. 따라서, 이는 user_list상에 있는 APM user들이 요청한 event들을 처리하는 일이 될 것이다. standbys_pending이나 suspends_pending이 0 이상의 값을 가진다면, 다음과 같은 일을 처리한다. 즉, `apm_info` 구조체의 connnection_version이 0x1000이상의 값을 가지고, pending_count가 0보다 작은 값을 가지는 경우에는 pending_count를 초기값인 4로 만들고, debug이 설정된 경우에는 APM 디버깅 정보를 표시한다. 이젠 시스템을 원래의 완전 기동 상태로 만들기 위해서 `APM_STATE_BUSY`로 전원 상태를 바꾼다(`set_system_power_state()`). 이때 에러가 발생한다면, `apm_error()`를 호출하도록 한다. 즉, 시스템에서 pending된 요청(request)가 있기에 이를 처리하기 위해서, 시스템의 전원 상태를 `APM_STATE_BUSY`로 바꾸는 역할을 수행하는 것이다. 만약 아무것도 pending된 요청이 없다면, pending_count에는 다시 4를 넣고, `check_events()`를 호출해서 BIOS에 pending된 event들이 있는지를 확인하도록 한다. 있다면, 이것을 다시 APM에 반영하도록 한다.

10. USB(Universal Serial Bus) Device Driver

USB(Universal Serial Bus)는 사용의 편리와, 넓은 확장성 그리고, PC와 전화와의 연결을 위해서 1994년 Compaq, Intel, Microsoft, Nec등이 주축이 되어 개발되었다. 크게 두 가지 종류의 chip이 PC쪽에서 USB와의 연결을 담당하며 Intel과 Compaq에서 제공된다. 현재 USB Spec. 2.0이 나와 있는 상태이며, Spec 1.1에서는 최대 12Mbps를 그리고, Spec. 2.0에서는 최대 120Mbps를 지원한다.

USB는 hierarchily한 구조를 구성하며, master/slave protocol을 사용해서 USB device와 통신한다. 즉, Device 측에선 Host측으로의 통신선로를 만들지 못한다. 따라서, Host(PC)에서만 통신의 시작을 할 수 있다. USB에 맞물릴 수 있는 device의 개수는 최대 127개이다.

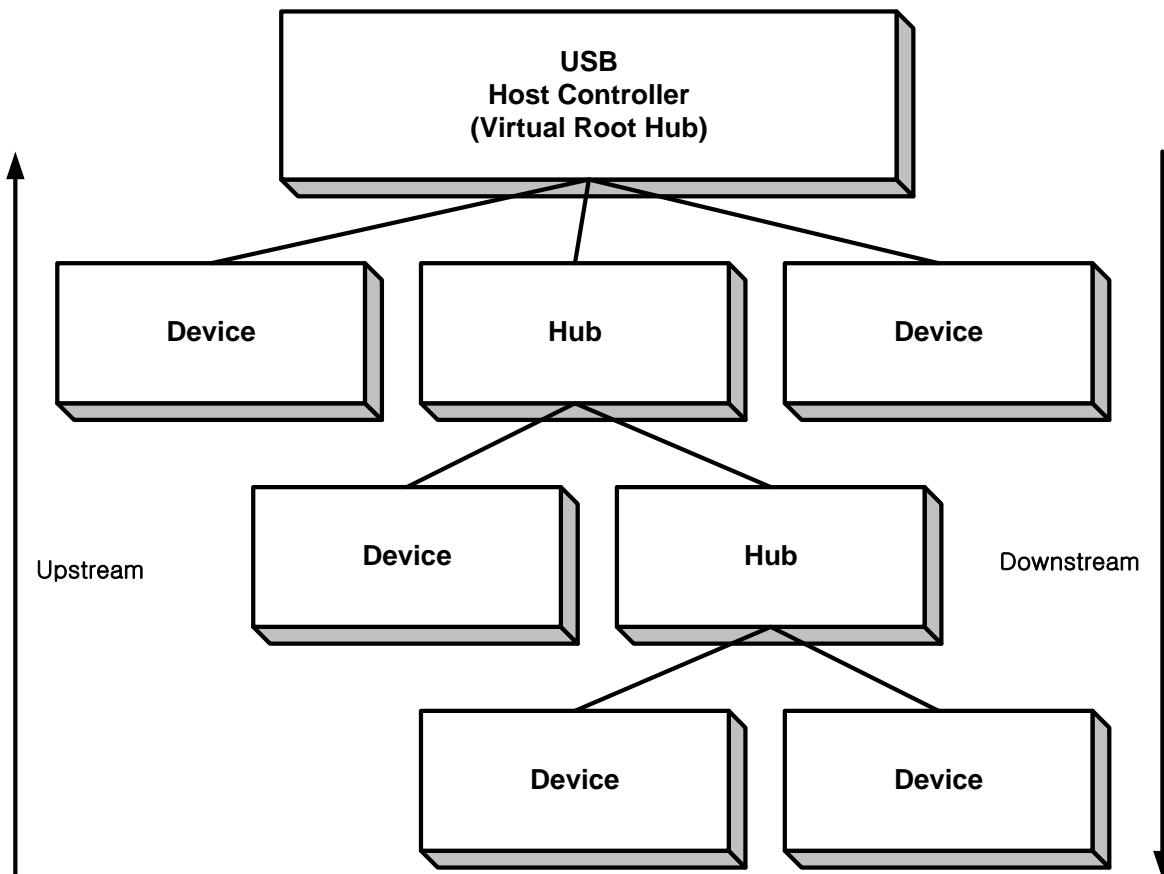


그림 73. USB의 Bus Topology

[그림 58]은 USB의 Bus Topology를 보여준다. 그림에서 상위로 가는 것을 upstream이라고 표현하며, 아래로 가는 방향을 downstream이라고 한다. USB의 Host Controller가 가상의 Root Hub 역할을 하며, 각각의 디바이스는 직접 Hub에 연결되던가, 아니면 새로운 Hub를 통해서 다시 Root Hub에 연결된다.

10.1. USB Host Controller

오늘날의 USB host controller는 대부분 PC의 motherboard에 들어가 있다. 이러한 controller들을 서로 호환성이 있으며, 크게 OHCI (Open Host Controller Interface : Compaq)와 UHCI(Universal Host Controller Interface : Intel)로 나뉜다. 서로간에 호환성이 있으며, 같은 capability를 제공하기에 USB device는 host controller의 종류에 대해서 걱정할 필요가 없다. 기본적으로 UHCI hardware가 조금 더 간단하기에 좀 더 복잡한 device driver가 필요하며, 따라서 CPU에 조금 더 load를 준다고 볼 수 있다. 다음과 같은 구조를 가지게 된다.([그림59]를 참조)

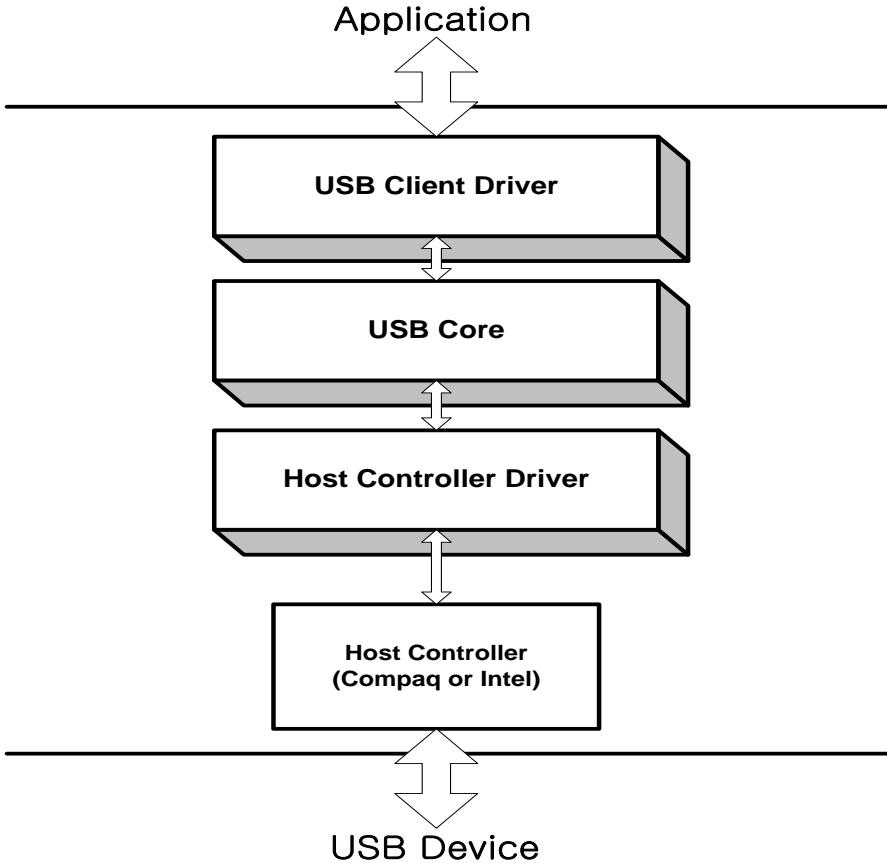


그림 74. USB Device Driver Stack

[그림59]에서 Application program은 USB Client Driver와 interface를 하게 되며, USB Client Driver는 Host Controller driver를 이용하기 위해서 USB Core를 이용해서 접근하게 된다. 물론 이것은 Linux에서의 USB 구현에 대한 부분이기에 이런 구조를 가진다. 다른 OS에서의 구현도 이와 유사한 구조를 가지게 되지만 USB core에 해당하는 부분에 다른 것이 들어가게 될 것이다. USB Host Controller Driver는 대부분 이미 구현된 상태이며, 안정된 되었기에, Device driver를 만들고자 하는 사람은 USB Client Driver만을 구현하면 될 것이다. 따라서 이후에서 살펴볼 부분도 USB client driver를 위주로 살펴보게 될 것이다.

USB가 다른 운영체제에서는 어떤 구조를 가지고 있는지를 보기위해서 Windows NT에서의 구현을 살펴보면, Windows에서는 위에서 설명한 구조와 약간의 차이를 보인다. 하지만, 근본적으로 그렇게 다르지는 않다. [그림60]을 보기로 하자. USB device가 Bus에 놓여지게 되면, 실제로는 PCI Bus상에 device가 장착되는 것과 같은 역할을 하게 된다. 즉, PCI enumerator가 USB device가 PCI Bus에 놓임을 알아내게 된다. 이것은 다시 모든 USB driver stack이 자동적으로 load되도록 한다. 각각의 Driver layer에서 하는 역할을 살펴보면 다음과 같다.

- ◆ UHCI/OHCI Driver : Linux와 마찬가지고 두 가지의 USB device를 연결하는 고리를 형성하는 standard이다. 특정의 USB interface에 대해서 둘 중 하나의 driver가 load된다. USB driver중에서 가장 하위에 놓인다.
- ◆ USB Class Driver : USBD.SYS이며, USB transaction²⁴⁶을 처리하는 layer이다. Power management나 bus enumeration을 책임진다. 주로 우리가 작성하는 device driver가 통신하게 되는 driver이다.
- ◆ USB Hub Class Driver : USBHUB.SYS로 USB Class Driver가 root hub를 enumerate하게 되면 load된다. 또한 downstream에서 hub가 enumerate될 때 재귀적으로 자신을 호출해서 reference count를 증가시키고, 새로운 hub를 위한 data structure를 할당한다. 모든 hub에 대한 transaction의

²⁴⁶ 뒤에서 USB transaction에 대한 설명을 참조하기 바란다.

처리를 담당하는데, 이러한 것으로는 port의 초기화, 특정 hub에 붙은 downstream device에 대한 초기화, 그리고 USB의 관리적 측면의 function들이 있다. USB driver stack에서 가장 상위에 위치하며, USB Device Interface (USBDI)를 정의한다. USBDI 상위에는 USB driver stack을 사용하게 될 client driver들이 load된다.

각각의 driver layer를 비슷한 역할을 해준다는 점과, 정확한 interface의 정의를 바탕으로 상위의 USB client driver와 통신한다는 점에서 Linux와 다르지 않다고 할 수 있다. 물론 내부적인 mechanism에서는 차이가 있을 것이다.

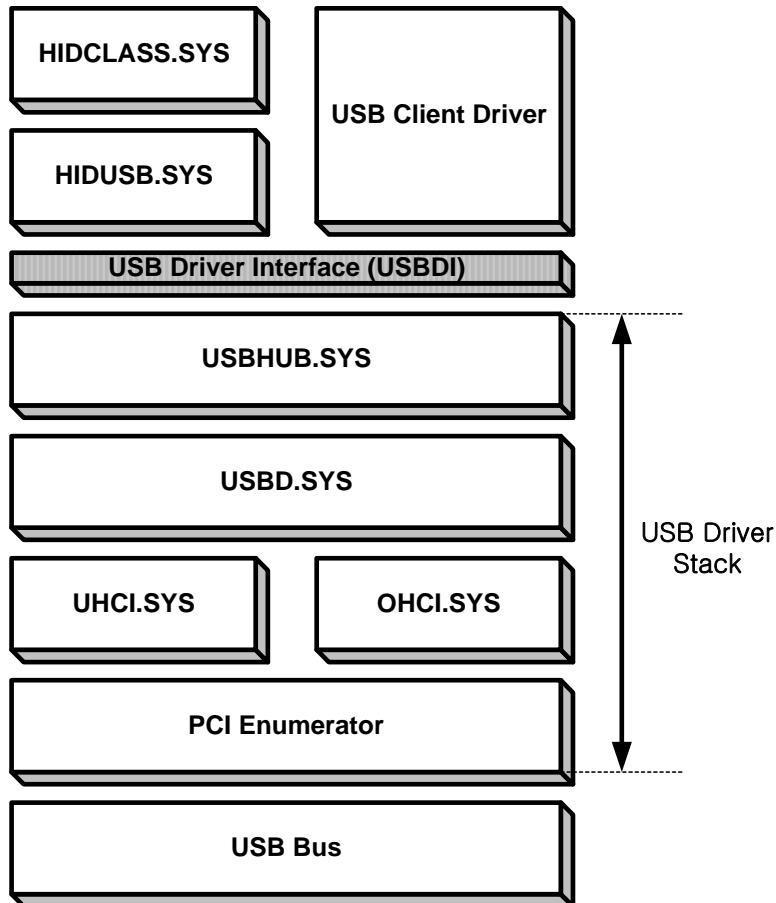


그림 75. Windows USB Driver Architecture.

Linux와 window에서의 구현을 살펴보았는데, USB의 특성상 많은 종류의 device가 올라 올 수 있다는 점에서 Layer architecture를 취했다는 점, 그리고 이러한 많은 종류의 device driver에 일관된 interface를 제공한다는 점에 특히 유의해서 보아야 할 것이다.

10.2. USB Device들과 Transfer Characteristics

USB device로는 다양한 종류가 있으며, 여러 가지의 다른 목적으로 사용된다. 또한 Device는 bus로부터 혹은 자신이 직접, 아니면 둘 다를 사용해서 전원을 공급 받을 수 있다. 두 가지 경우를 다 지원하는 device의 경우에는 외부전원이 연결될 경우 self-powered mode로의 switch를 지원한다.

또한 USB device로는 속도의 차이가 있을 수 있다. USB Spec.은 low speed device로는 joysticks, keyboard, mouse 등이 있으며, high speed device로는 audio/video device 및 network device 등이 있다. 이와 같은 device들은 1.5Mbps에서 10Mbps의 다양한 속도를 지원한다.²⁴⁷

²⁴⁷ Protocol overhead로 2Mbps정도가 사용된다고 볼 수 있다.

USB device들은 다음과 같이 나누어서 생각해 볼 수 있다.

10.2.1. USB Hubs

Computer에는 두개 내지 네 개의 USB port가 후면부에 있다. 이러한 port들은 USB device를 연결할 때나 혹은 USB Hub를 연결할 때 사용된다. Hub는 USB device를 2개내지 8개까지 연결할 수 있도록 port를 연장하는데 사용된다. Hub 자체는 Bus powered 이거나 혹은 self powered일 수 있으며 full speed device이다. 일반적으로 physical ports들은 virtual root hub에 의해서 handle되며, host controller에 의해서 hub가 simulate되어 bus topology을 unify하는데 도움을 준다. 따라서, 모든 port들은 USB subsystem의 hub driver에 의해서 같은 식으로 handling되어진다.

10.2.2. Data Flow

USB의 data flow로는 다음과 같은 것이 있다.(여기서 말하는 data flow라는 것은 전송의 형태를 말하는 것이다.)([그림58]을 참고하라.)

- Downstream쪽으로 향하는 것으로 OUT으로 표현된다.
- Upstream으로 향하는 것으로 IN으로 표현된다.

각각에 대해서 다음과 같은 종류의 전송(Transfer)가 존재한다.

1. **Control Transfer** : 작은 data를 전송하기 위해서 사용되며, USB device를 configure하기 위해서 사용되며 적어도 최소한의 control command를 가진다.
2. **Bulk Transfer** : Data를 packet의 형태로 reliable하게 전송하기 위해서 사용된다. 전체 Bandwidth를 차지할 수도 있다.
3. **Interrupt Transfer** : 주기적으로 polling된다. 한번 주어지면, Host Controller는 주기적으로 자동적으로 request를 하게 된다.
4. **Isochronous Transfer** : 일정한 속도로 일정량의 data를 전송하기를 원하는 경우에 사용한다. 이러한 것으로 Stream이 있으며, reliable한 전송을 보장하지는 않는다. 주로 Audio나 Video의 data를 전송하는 경우 유용할 것이다.

위와 같은 전송 형태를 가지고, 연결하고자 하는 Device를 명시해 주어야 하며, 이것은 USB class를 나누는 기준이 될 수 있을 것이다. 이하에서 좀더 자세히 보도록 할 것이다.

10.3. Descriptor

USB나 혹은 다른 PCI device같은 것을 다루게 될 때, device의 configuration을 알기 위해선 device의 descriptor들을 읽어와야 한다. 이렇게 읽어온 data를 기초로 어떤 device가 연결되었는지를 알 수 있게 되며, 또한 device로 data를 보내고자 할 때, device의 특성에 맞게 data의 양을 조절해 줄 수도 있게 된다. USB device들은 bus에 붙게 되면, USB subsystem에 의해서 enumeration이 된다.(이것은 일종의 PNP-Plug and Play라고 보아도 될 것이다.) 즉, 고유한 device 번호가 할당된다. 그리고 나선 device의 descriptor가 읽혀지게 되며, 이렇게 읽어진 descriptor에는 device와 properties들이 정의 되어있다. USB Spec.에는 다음과 같은 Descriptor들이 정의 되어 있다.([그림61]을 참조하라.)

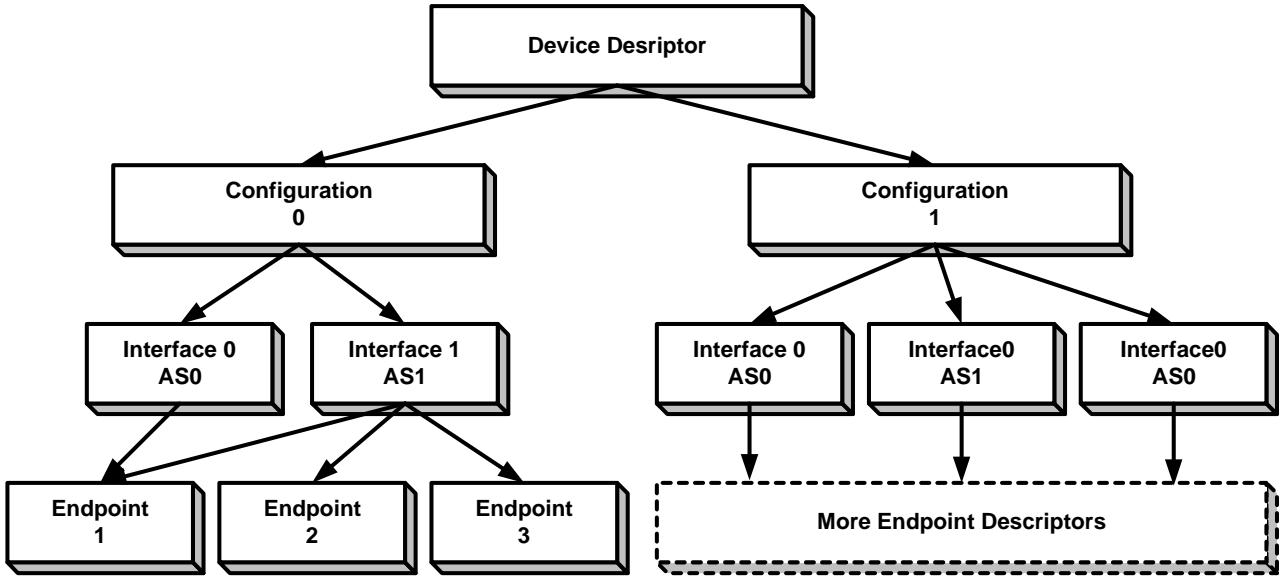


그림 76. USB Descriptor Hierarchy

- **Device descriptor** : USB device에 대한 일반적인 정보를 제공한다. 단지 하나의 descriptor만이 존재하며 그 USB device에 전역적으로 영향을 미친다.
- **Configuration descriptor** : 특정의 device configuration에 대한 정보를 제공한다. USB device는 하나나 그 이상의 configuration descriptor를 가질 수 있으며, 각각의 descriptor는 또한 하나나 그 이상의 interface를 가질 수 있다. 물론 각각의 interface를 역시 0이나 여러 개의 endpoint를 가질 수 있다. 이러한 endpoint들은 같은 interface의 다른 setting에 의해서 사용되지 않는다면, 단일의 configuration내에서 interface간에 공유될 수 없다. 하지만 이러한 제한이 없는 다른 configuration내의 interface에 의해서는 공유가 될 수 있다. configuration들은 control transfer에 의해서 배타적으로 activate될 수 있으며, global한 device setting을 변화시키기 위해서 사용될 수 있다.(여기서 사용되는 control command로는 *set_configuration*이다.)
- **Interface descriptor** : 하나의 configuration내의 특정 interface를 말하며, 0이나 혹은 여러 개의 endpoint를 가질 수 있다. 물론 이러한 endpoint들은 하나의 configuration내에서 고유한 set을 이룬다. Interface는 또한 alternate setting을 가질 수 있으며, 이것은 endpoint를 변화시키거나 endpoint의 특성을 device가 configure된 후에 변화시키기 위해서 사용될 수 있다. Interface에 대한 default setting은 항상 alternate setting 0이다. Alternate setting은 *set_interface control command*로 배타적으로 select될 수 있다. 이러한 예로서는 multi-functional device를 들 수 있다. 즉, 하나의 device가 camera로 사용되거나 Microphone으로 사용될 수 있는 경우이다. 이 경우 두 가지 모두로 사용될 수도 있을 것이다. 즉, 화상 회의 용으로 사용되어 image 및 음성 data의 capture를 동시에 하는 경우라 할 수 있다.
- **Endpoint descriptor** : host가 각각의 endpoint의 bandwidth requirements를 결정하기 위해서 필요하다. 간단히 이야기 해서 endpoint라는 것은 USB device에 있어서는 data의 논리적인 source가 되거나 sink가 된다고 할 수 있다.²⁴⁸ 다른 말로 표현하자면 pipe라고도 한다. 가장 기본적으로 Endpoint 0은 항상 모든 control transfer에 사용되며, descriptor가 존재하지 않는다.
- **String descriptor** : optional한 것이며, 사람이 읽을 수 있는 unicode format으로 된 string을 가진다. 이것은 vendor나 device name 혹은 serial number 등을 가질 수 있을 것이다.

이것을 정확히 이해하는 것이 이후의 프로그램에서 device를 검출하고, 사용하고자 하는 device의 feature를 선택하는데 있어서 중요하다. [그림 4]를 보면 device의 descriptor들이 상위에서 하위로 계층적인 구조를 가지고 정의 되어 있으며, 하나의 device descriptor에 configuration, interface, endpoint descriptor들이 있음을 볼 수 있다.

²⁴⁸ Source는 data의 생성을 담당하고, sink는 생성된 data의 최종 목적지라고 생각하면 된다.

이상에서 USB를 기본적으로 이해하기 위한 것들을 살펴 보았다. 하지만, 실제로 있어서는 USB device들은 특정의 USB class내에 속하게 되며, 특정 class들에 대한 spec은 따로 존재한다. 따라서, USB device driver를 작성하고자 하는 사람은 USB class Spec.에 대해서도 상세히 알고 있어야 할 것이다. Device descriptor들에 대한 실제적인 구현 code는 아래의 코드와 같다. 보면 알 수 있듯이, 상위의 descriptor가 하위의 descriptor를 reference하도록 구현되어져 있다.

```
/* Device descriptor */
struct usb_device_descriptor {
    __u8 bLength;
    __u8 bDescriptorType;
    __u16 bcdUSB;
    __u8 bDeviceClass;
    __u8 bDeviceSubClass;
    __u8 bDeviceProtocol;
    __u8 bMaxPacketSize0;
    __u16 idVendor;
    __u16 idProduct;
    __u16 bcdDevice;
    __u8 iManufacturer;
    __u8 iProduct;
    __u8 iSerialNumber;
    __u8 bNumConfigurations;
} __attribute__((packed));

/* Endpoint descriptor */
struct usb_endpoint_descriptor {
    __u8 bLength;
    __u8 bDescriptorType;
    __u8 bEndpointAddress;
    __u8 bmAttributes;
    __u16 wMaxPacketSize;
    __u8 bInterval;
    __u8 bRefresh;
    __u8 bSynchAddress;

    unsigned char *extra; /* Extra descriptors */
    int extralen;
} __attribute__((packed));

/* Interface descriptor */
struct usb_interface_descriptor {
    __u8 bLength;
    __u8 bDescriptorType;
    __u8 bInterfaceNumber;
    __u8 bAlternateSetting;
    __u8 bNumEndpoints;
    __u8 bInterfaceClass;
    __u8 bInterfaceSubClass;
    __u8 bInterfaceProtocol;
    __u8 iInterface;

    struct usb_endpoint_descriptor *endpoint;

    unsigned char *extra;
    int extralen;
} __attribute__((packed));
```

```

struct usb_interface {
    struct usb_interface_descriptor *altsetting;

    int act_altsetting;          /* active alternate setting */
    int num_altsetting; /* number of alternate settings */
    int max_altsetting;         /* total memory allocated */

    struct usb_driver *driver;   /* driver */
    void *private_data;
};

/* Configuration descriptor information.. */
struct usb_config_descriptor {
    __u8 bLength;
    __u8 bDescriptorType;
    __u16 wTotalLength;
    __u8 bNumInterfaces;
    __u8 bConfigurationValue;
    __u8 iConfiguration;
    __u8 bmAttributes;
    __u8 MaxPower;

    struct usb_interface *interface;
} __attribute__((packed));

/* String descriptor */
struct usb_string_descriptor {
    __u8 bLength;
    __u8 bDescriptorType;
    __u16 wData[1];
} __attribute__((packed));

```

코드 622. Linux에서의 USB descriptor의 데이터 구조 정의

각 필드들에 대해서 잠시 보도록 하자. 먼저 USB device descriptor는 [표 50]과 같다. 주로 device에 대한 개략적인 설명과 몇 개의 configuration이 오는지에 대한 것을 알려준다. 또한 endpoint 0에 대해서 최대전송 크기를 명시해서 bus의 초기화에 필요한 control endpoint로 전송할 수 있는 최대 packet size를 알 수 있게 된다.

Offset	Field	Size(Bytes)	Value
0	Blength	1	Descriptor의 크기를 byte수로 나타냄
1	BDescriptorType	1	Device descriptor인 것을 나타냄
2	BcdUSB	2	USB Spec. release number를 BCD format으로 나타냄
4	BdeviceClass	1	USB Device의 class code
5	BDeviceSubClass	1	USB Device의 subclass code
6	BDEviceProtocol	1	USB Device의 protocol code
7	BMaxPacketSize	1	Endpoint 0에 대한 maximum packet size
8	IdVendor	2	Vendor ID
10	IdProduct	2	Product ID
12	BcdDevice	2	Device release number를 BCD format으로 나타냄
14	IManufacturer	1	Manufacturer를 나타내는 string descriptor에 대한 index값이거나 0
15	Iproduct	1	Productor를 나타내는 string descriptor에 대한 index값이거나 0

16	ISerialNumber	1	Product의 serial number를 나타내는 string descriptor에 대한 index값이거나 0
17	bNumConfigurations	1	가능한 configuration의 수

표 56. USB device의 device descriptor의 Field들에 대한 설명.

Offset	Field	Size(Bytes)	Value
0	BLength	1	Descriptor의 크기를 byte수로 나타냄
1	BDescriptorType	1	Descriptor의 type이 configuration이라고 표시함
2	WTotalLength	2	Configuration 및 interface descriptor, endpoint descriptor, class/vendor specific한 descriptor, 모든 string descriptor를 포함한 크기
4	BNumInterfaces	1	0이 configuration descriptor에서 지원되는 interface의 수
5	bConfigurationValue	1	Configuration을 선택하기 위해서 사용되는 값
6	IConfiguration	1	Configuration을 위한 string descriptor의 index값
7	BmAttributes	1	Configuration특성을 나타내는 값. (bit 7 : bus-powered, bit 6 : self-powered, bit 5 : remote wakeup, other bits : reserved)
8	MaxPower	1	Bus-powered USB device에서 최대로 소모 power
9	*interface	Sizeof(struct usb_interface *)	USB device의 interface descriptor를 가리키는 pointer

표 57. USB device의 configuration descriptor

Offset	Field	Size(bytes)	Description
0	BLength	1	Interface descriptor의 크기를 byte로 나타냄
1	BDescriptorType	1	Interface descriptor라는 것을 표시
2	BInterfaceNumber	1	Interface의 개수
3	BAlternateSetting	1	Interface의 alternate setting을 나타냄
4	BNumEndPoints	1	Endpoint 0를 제외한 0이 interface에서 사용되는 endpoint의 수
5	BInterfaceClass	1	Class code
6	BInterfaceSubClass	1	Subclass code
7	BInterfaceProtocol	1	Protocol code
8	IInterface	1	Interface를 위한 string descriptor의 index
9	*endpoint	Sizeof(struct usb_endpoint_descriptor *)	Endpoint descriptor를 가리키는 pointer
	*extra	Sizeof(unsigned char *)	Extra descriptor를 가리키는 pointer
	Extralen	Sizeof(int)	Extra descriptor의 length

표 58. USB device의 interface descriptor

Interface descriptor에서 Interface class field는 0인 경우에 USB-specific device class에 속하지 않으며, 0xFF인 경우에는 vendor-specific하게 결정된다. 또한 interface subclass field의 경우, interface class가 0이면 0이어야 하며, interface subclass field가 0xFF가 아니라면, reserved되어 있다. Interface protocol field의 경우에는 0일 경우, class specific protocol을 사용하지 않음을 나타내고, 0xFF인 경우 vendor specific protocol을 사용한다.

Offset	Field	Size(bytes)	Description
0	Blength	1	Descriptor의 크기를 byte수로 나타냄
1	BDescriptorType	1	Endpoint descriptor임을 나타냄
2	bEndpointAddress	1	0이 descriptor에 의해서 나타내지는 USB device의

			endpoint address를 나타냄. (bit 0 ~ 3 : endpoint number, bit 4 ~ 6 : reserved 0, bit 7 : direction – control transfer의 경우는 무시, 0은 OUT endpoint, 1은 IN endpoint)
3	BmAttributes	1	Configuration descriptor에서 bConfigurationValue를 이용해서 configure될 때 사용되는 endpoint attributes. (00 : control, 01 : Isochronous, 10 : Bulk, 11 : Interrupt, 나머지 bit들은 reserve됨)
4	wMaxPacketSize	2	Maximum packet size
6	BInterval	1	Millisecond 단위의 data transfer를 위한 endpoint를 polling하는 interval 값
7	bRefresh	1	No description
8	bSynchAddress	1	No description
9	*extra	Sizeof(unsigned char *)	Extra descriptor에 대한 pointer
	Extralen	Sizeof(int)	Extra descriptor에 대한 length

표 59. USB device의 endpoint descriptor

Extra descriptor란 vendor specific한 추가적인 descriptor가 더 올 수 있다는 것을 나타내며, 이것이 어디에 있는지 찾을 수 있고, 그 크기가 얼마인지를 명시한다.

Offset	Field	Size	Description
0	BLength	1	Descriptor의 크기를 byte 단위로 나타냄
1	bDescriptorType	1	String Descriptor임을 나타냄
2	bData[1]	bLength -1	Unicode character string

표 60. USB device의 string descriptor

String descriptor의 마지막 field는 NULL로 끝나지 않는 Unicode character로 되어 있다. 따라서 copy하기 위해서는 크기를 나타내는 field를 access해서 그 크기만큼을 buffer로 copy해서 사용해야 할 것이다. 또한 unicode²⁴⁹라는 것을 명심하도록 하자.

다음으로는 USB device descriptor와 관련된 window 쪽의 정의를 보도록 하겠다. 이해의 수준을 넓히기 위한 시도이므로 참고적으로 알아두면 좋을 것이다. 주로 data structure만을 간단히 보자.

10.4. Windows device descriptors for USB

Windows Driver Model (WDM)²⁵⁰에서도 관련된 device descriptor들을 정의하고 있다. 아래의 코드를 보도록 하자. Device, endpoint, configuration, interface, string descriptor들의 정의를 볼 수 있다. 이외에도 power descriptor, hub descriptor들이 있다. 각 field들에 대한 설명은 위에서 설명한 것을 토대로 하면 될 것이다.

```
typedef struct _USB_DEVICE_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT bcdUSB;
    UCHAR bDeviceClass;
    UCHAR bDeviceSubClass;
    UCHAR bDeviceProtocol;
```

²⁴⁹ 16 bits를 사용해서 하나의 character를 나타낸다. 따라서, ASCII는 unicode의 하나의 subset으로 들어간다.

²⁵⁰ Microsoft에서는 Windows Driver Model(WDM)이라는 driver 작성 model을 제공해서 각각의 device driver가 해주어야 할 것들을 명확히 define하고, 서로간의 interface는 어떻게 할 것인가를 정의하고 있다. 특징으로는 비슷한 역할을 하는 driver들을 묶어서 driver class 개념을 사용하고 있다는 점이다.

```

UCHAR bMaxPacketSize0;
USHORT idVendor;
USHORT idProduct;
USHORT bcdDevice;
UCHAR iManufacturer;
UCHAR iProduct;
UCHAR iSerialNumber;
UCHAR bNumConfigurations;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR;

typedef struct _USB_ENDPOINT_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bEndpointAddress;
    UCHAR bmAttributes;
    USHORT wMaxPacketSize;
    UCHAR bInterval;
} USB_ENDPOINT_DESCRIPTOR, *PUSB_ENDPOINT_DESCRIPTOR;

typedef struct _USB_CONFIGURATION_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT wTotalLength;
    UCHAR bNumInterfaces;
    UCHAR bConfigurationValue;
    UCHAR iConfiguration;
    UCHAR bmAttributes;
    UCHAR MaxPower;
} USB_CONFIGURATION_DESCRIPTOR, *PUSB_CONFIGURATION_DESCRIPTOR;

typedef struct _USB_INTERFACE_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bInterfaceNumber;
    UCHAR bAlternateSetting;
    UCHAR bNumEndpoints;
    UCHAR bInterfaceClass;
    UCHAR bInterfaceSubClass;
    UCHAR bInterfaceProtocol;
    UCHAR iInterface;
} USB_INTERFACE_DESCRIPTOR, *PUSB_INTERFACE_DESCRIPTOR;

typedef struct _USB_STRING_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    WCHAR bString[1];
} USB_STRING_DESCRIPTOR, *PUSB_STRING_DESCRIPTOR;

typedef struct _USB_COMMON_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
} USB_COMMON_DESCRIPTOR, *PUSB_COMMON_DESCRIPTOR;

```

코드 623. Windows에서의 USB Descriptor의 구현

Window에서는 Linux와는 달리 상위의 descriptor가 하위의 descriptor들을 reference하지는 않는다. 다만 가정에서 모든 descriptor들이 선형적으로 연속되어서 나온다는 것을 알 수 있을 것이다. 이것이 linux에서의 descriptor구현과 가장 큰 차이점이라 하겠다. 나머지는 기본 spec.을 따르므로 차이가 없다.

10.5. Device Class

일반적인 USB device와 interface descriptor는 classification에 대한 field를 가지고 있다. 이러한 classification의 정보로 사용되는 것으로는 Class, Sub-class, Protocol이 있으며, Host에서 device 혹은 interface를 device driver에 연관시키기 위해서 사용된다. 물론 Spec.으로 정의되어 있다. Valid한 값은 USB Device Working Group에 의해서 정해진다.

이렇게 device 혹은 interface를 특정의 class로 묶어서 취급하고, Class Spec.에서 각각의 USB device class에 대한 성질들을 명시하는 것은 class에 바탕 한 여러 가지의 implementation을 다루어야 하는 host software의 개발에 많은 도움을 준다. 따라서, host software는 device에 의해서 주어지는 description에 따라서 특정의 device 혹은 interface에 대한 operation으로 한정할 수 있게 된다. Class specification은 최소의 device 혹은 interface의 operation을 정의하며, 이것으로 어떤 class의 member인지를 알 수 있게 된다.

따라서, device driver의 writer들은 읽어온 descriptor정보에 바탕해서 device가 자신이 작성한 driver가 사용하게 될지를 결정하게 되며, 최소한의 지원되어야 할 operation을 알게 된다.(여기서 말하는 operation이란, 가령 write를 한다든지 통계적인 수치를 읽는 등의 driver에서 상위의 application에 제공하는 것을 지칭한다.) 다음과 같은 class가 존재한다.

- Display class : monitor.
- Communication : modem, ATM, ethernet.
- Audio : speaker.
- Mass Storage : hard drive.
- Human Interface(HID-Human Interface Device) : keyboard, pointing device.

각 각의 class에 대한 code정의는 다음과 같다.

```
/*
 * USB device and/or Interface Class codes for LINUX
 */
#define USB_CLASS_PER_INTERFACE          0      /* for DeviceClass */
#define USB_CLASS_AUDIO                  1
#define USB_CLASS_COMM                  2
#define USB_CLASS_HID                   3
#define USB_CLASS_PRINTER               7
#define USB_CLASS_MASS_STORAGE          8
#define USB_CLASS_HUB                   9
#define USB_CLASS_DATA                  10
#define USB_CLASS_VENDOR_SPEC           0xff

/*
 * USB device class code for Windows
 */
#define USB_DEVICE_CLASS_RESERVED        0x00
#define USB_DEVICE_CLASS_AUDIO           0x01
#define USB_DEVICE_CLASS_COMMUNICATIONS 0x02
#define USB_DEVICE_CLASS_HUMAN_INTERFACE 0x03
#define USB_DEVICE_CLASS_MONITOR         0x04
#define USB_DEVICE_CLASS_PHYSICAL_INTERFACE 0x05
#define USB_DEVICE_CLASS_POWER           0x06
#define USB_DEVICE_CLASS_PRINTER         0x07
#define USB_DEVICE_CLASS_STORAGE         0x08
#define USB_DEVICE_CLASS_HUB             0x09
#define USB_DEVICE_CLASS_VENDOR_SPECIFIC 0xFF
```

코드 624. USB 디바이스 Class Code에 대한 Linux와 Windows의 정의

관련된 문서로는 “USB Class Specification 1.1”을 읽어보기 바란다. 각각의 class들에 대해서 특정 class code가 정해져 있으며, 이것은 device descriptor를 보고 판단한다. 자, 이제는 Linux상에서 USB device driver를 어떻게 구현할 것인가에 대한 이야기를 하도록 하자.

10.6. USB Device Driver

USB Device Driver는 USB에 대한 application의 입출력 요구를 적절한 형태로 USB core쪽으로 전송하는 역할을 하며, 또한 그 처리의 결과를 application으로 되돌려 주는 역할을 한다. 여기서 말하는 application은 사용자 수준의 program이 될 수도 있을 것이며, kernel내에 들어가는 또 다른 device driver도 될 수 있을 것이다. 따라서, 상위의 application은 하위의 USB 구현 구조에 상관없이 일관되게 interface를 할 수 있어야 하기에 USB Device Driver에서는 일관된 interface를 제공해 주어야 한다. 또한 USB Device Driver는 이렇게 주어진 요구에 맞춰서, 적절히 USB core와 통신할 수 있어야 할 것이다. 이것을 위해서 도입된 것이 URB(USB Request Block)이다.

10.7. Linux USB Subsystem

이상에서 Linux에서의 USB가 어느 부분을 차지 하는지를 대략적으로 알아보았다. 이젠 조금 더 하위로 들어가 사용하게 되는 data structure와 상위의 Kernel과 driver의 interface에 대해서 알아보도록 하겠다. 실제적인 driver의 개발에 있어서는 이러한 것들이 큰 도움을 주리가 생각된다. 즉, kernel과 기존의 interface가 어떻게 USB라는 새로운 Peripheral interface와 연결 될 수 있는지를 지켜보게 될 것이다.

Linux에서의 USB driver는 하위의 layer를 사용할 수 있도록 하기위해서 API를 제공한다. 이러한 API를 사용해서 사용자 USB driver는 하위의 driver를 interface하게 된다. [그림62]는 이것을 간단히 나타낸 것이다.

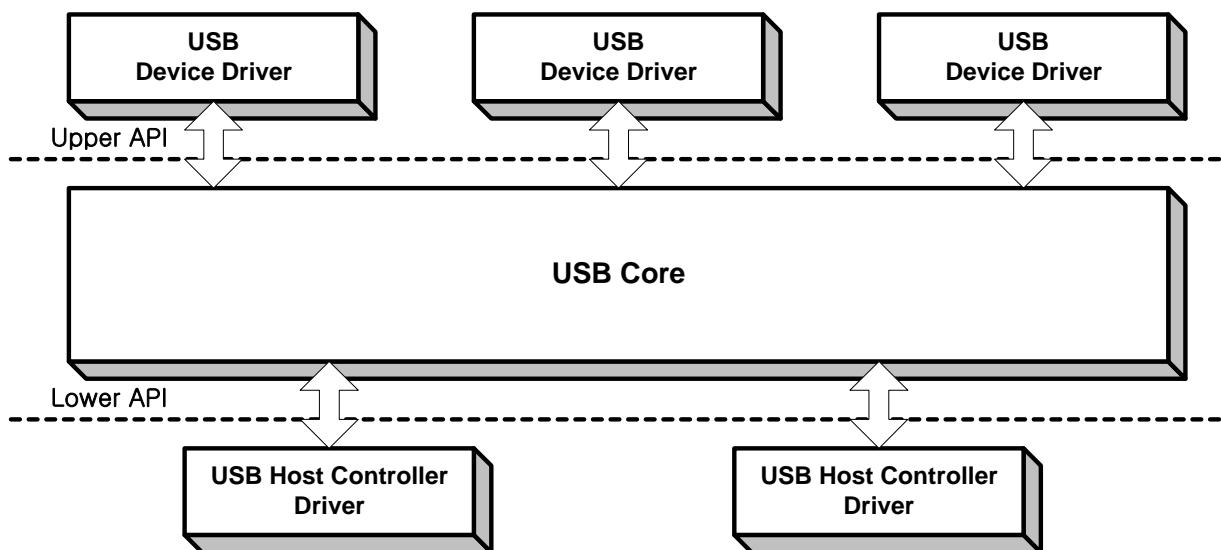


그림 77. USB Core API Layers

[그림62]에서 보듯이 USB Core가 하는 역할은 Hardware와 device dependent한 부분을 data structure와 macro를 정의해서 abstract하게 하는 역할을 한다. 즉, USB core는 USB device driver들과 Host controller driver에 일반적인 routine들을 가진다. 이것의 주된 목적은 hardware나 device에 dependent한 부분들에 대해서 data structure와 function, macro 등을 제공하는 것이다. 이렇게 함으로 해서 상위에 올라갈 program들이 이것을 이용해서 hardware나 device에 손쉽게 접근하게 되는 것이다. API는 크게 두 가지 부분으로 나누어지며, Upper API layer와 Lower API layer로 구분된다. USB driver writer는 단순히 USB Device driver부분만을 담당하므로 Upper API만 알면 될 것이다. 하위는 이미 안정된 되었다고 볼 수 있다. 이하에서는 USB core를 사용하기 위한 data structure와 그것을 다루기 위한 function들에 대해서 알아보기로 하자.

10.8. USB device driver의 frame work

여기서 설명하고자 하는 것은 USB device driver가 가져야 할 기본적인 routine나 interface이다. 즉, 이것을 통해서 상위의 kernel이 device에 접근한다고 가정하는 것이다. 물론 이렇게 해주기 위해선 kernel에 이러한 interface를 사용하는 부분이 있다는 것을 알아야 할 것이다. 또한 모든 USB device는 적어도 여기서 설명할 것에 대한 구현을 가지고 있어야 할 것이다.

USB device들을 사용되기 위해선 subsystem에 등록(register)되어야 한다. 또한 사용이 다 끝났을 경우에는 등록을 취소해 주어야 한다. 이것은 USB와 같은 device들은 system에 항상 attach되어 있다고 가정하지 않는데 기인하며, system이 켜진 상황에서도 설치 및 제거가 된다는 특성에서도 연유된다고 볼 수 있다. Driver는 두개의 entry point와 자신의 이름을 등록 해야 한다. 또한 driver에 따라서는 여러 개의 file operation과 minor number를 등록하는 경우도 있다.²⁵¹ 이럴 경우에는 16개까지의 minor number가 할당 될 수 있을 것이다. 즉, USB device를 16개까지 같은 것을 연결할 수 있다는 말이 된다. 즉, 하나의 driver가 16개의 같은 device를 관리한다고 보면 될 것이다. 하지만 모든 USB device의 major number는 1800이다.

```
struct usb_driver{
    const char *name;
    void *(*probe) (struct usb_device *, unsigned int);
    void (*disconnect) (struct usb_device *, void *);
    struct list_head driver_list;
    struct file_operations *fops;
    int minor;
}
```

코드 625. `usb_driver`구조체의 정의

모든 USB와 관련된 function 혹은, data structure들은 “usb_”로 시작한다는 규칙을 가진다. `usb_driver` 구조체는 USB device를 등록하기 위해서 필요한 data structure를 보여주며, USB driver 구조체에서 각각의 field가 하는 역할은 다음과 같다.

- ◆ Name : module의 이름을 주면 된다.
- ◆ Probe : device를 검출하는 function을 가리킨다. 이것은 kernel에서 정확한 USB driver를 선택할 수 있도록 도와주는 function이며, 새로운 device가 검출될 때 호출된다.
- ◆ Disconnect : 더 이상 USB device가 사용되지 않을 때 호출되며, probe에서 해주었던 일을 이곳에서 되돌려준다고 생각하면 된다.
- ◆ Driver_list : subsystem에서 내부적으로 사용되며, {NULL, NULL}로 초기화 된다.
- ◆ Fops : file operation들을 명시하는 부분이다. Driver에 대한 정상적인 file operation들을 나타내주면 된다.
- ◆ Minor : device에 할당된 base minor number를 명시한다. 이 값은 16의 배수이어야 한다.

여기서 Probe에 대한 것과 disconnect에 대한 것을 조금 더 살펴보도록 하자. 이것은 USB device driver에서 반드시 있어야 하는 function이며, 대부분의 초기화와 관련된 일과 해제에 관련되어서 해주어야 할 일들이 처리되는 곳이다.

- ◆ void *probe(struct usb_device *dev, unsigned int interface); - 이 function은 BUS²⁵²에 새로운 device가 물리게 되면, 호출되며, device driver는 새로운 device에 대한 내부적인 data structure를 하나 만들게 된다. 각각의 field가 하는 역할은 다음과 같다.
- ✓ Dev : device context를 명시한다. 모든 USB descriptor에 대한 pointer를 가지고 있다.

²⁵¹ 이러한 예가 될 수 있는 것으로는 terminal io interface를 덧붙이는 경우가 있을 수 있을 것이다. 하나의 computer에 io USB를 통해서 붙을 수 있는 여러 개의 terminal interface가 덧붙을 수 있기 때문이다.

²⁵² 간혹 USB device driver를 이야기하면서 BUS라는 사실을 잊어먹는다. 항상 기억하도록 하자. Bus에 device가 붙게 되면, 주소가 정해져야만 access를 할 수 있으며, 또한 device를 나타내는 ID값들이 존재하게 된다. IEEE 1394의 경우도 마찬가지이며, PCI device에 대해서도 적용된다.

- ✓ Interface : interface number를 명시한다. 만약 USB driver가 특정의 device와 interface에 자신을 bind하고자 할 경우에는 device driver의 context structure에 대한 pointer를 return해야 한다.

Probe는 driver가 자신이 control할 device가 있는지를 확인하는 것으로 vendor와 product ID를 확인하거나 class와 subclass 정의를 확인하는 것이다. 만약 일치한다면 interface number가 driver에 의해서 지원되는 것과 비교되어지게 된다. 또한 class based한 방법으로 이러한 비교가 일어날 경우에는 USB descriptor가 더 비교되어지게 되는데, 이것은 device의 properties가 많이 다른 경우가 있을 수 있기 때문이다. 아래의 코드는 probe()에 대한 간단한 예를 보여준다.

```
Void *probe( struct usb_device *dev, unsigned int interface )
{
    struct driver_context *context;

    /* Device의 Vendor ID와 Product ID 및 interface값을 비교한다.*/
    if( dev->descriptor.idVendor == 0xXXXX &&
        dev->descriptor.idProduct == 0xXXXX && interface == 1 )
    {
        MOD_INC_USE_COUNT; /* Module의 usage count를 늘린다.*/

        /* Driver에서 사용하게 될 resource를 할당한다.*/
        Context = allocate_driver_resources();

        /* Bind할 목적으로 device의 context를 return한다.*/
        return context;
    }
    return NULL;
}
```

코드 626. probe()의 구현 예.

- ◆ Void disconnect (struct usb_device *dev, void *drv_context); - driver에 의해서 service받는 device가 제거 될 때 호출된다.

- ✓ Dev : device의 context를 명시한다.
- ✓ Drv_context : 이전에 probe에서 등록된 driver_context를 가리킨다. USB frame work는 disconnect function으로부터 return하게되면, 모든 device에 연관된 data structure들을 deallocate한다. 따라서, USB driver에서는 usb_device structure를 더 이상 사용할 수 없게 된다.

아래의 코드는 disconnect() 함수의 간단한 예를 보여준다.

```
Static void disconnect( struct usb_device *usbdev, void *drv_context)
{
    /* driver_context에 대한 pointer를 가져온다.*/
    struct driver_context *s = drv_context;

    /* remove pending flag 을 setting 한다.*/
    s->remove_pending = 1;

    /* 모든 driver의 sleeping part를 깨운다.*/
    wake_up( &s->wait );

    /* Driver가 device와 연관된 data structure들을 release할 수 있을 준비가 될 때까지
    기다린다.*/
    sleep_on( &s->remove_ok );
```

```

/* 사용된 resource들을 전부 free시킨다.*/
free_driver_resources(s);
/* Module usage count를 감소시킨다.*/
MOD_DEC_USE_COUNT;
}

```

코드 627. disconnect()함수의 예

이상에서 간단한 USB device의 검출 및 해제에 관련된 routine들을 살펴보았다. 이것은 USB driver를 이루는 entry function들이며, 다음으로 보아야 할 것은 USB core에서 제공하는 frame function에 대한 것들이다.

10.9. Framework function

여기서 보게 될 function들은 USB device driver를 등록하고, interface를 요구하거나, 혹은 이미 요구되어졌는지를 확인하거나, 또는 등록을 취소하거나 interface를 release하는 function들로서 주로 entry function들의 내부에서 사용될 수 있을 것이다. 물론 사용하는 도중에 더 요구하는 특별한 상황이 있을 수도 있지만 거의 없을 것이다.

- ◆ Int `usb_register(struct usb_driver *drv);` - 새로운 USB device driver를 subsystem에 등록하고자 할 때 사용한다. 성공 했다면 0을, 실패했을 때는 error value를 return한다.
- ◆ Void `usb_deregister(struct usb_driver *drv);` - 이전에 등록된 USB device driver를 등록 해지 한다.
- ◆ Void `usb_driver_claim_interface(struct usb_driver *driver, struct usb_interface *iface, void *drv_context);` - Probing시에 device에 대해서 하나 이상의 interface를 필요로 할 경우에 USB device driver에 의해서 사용된다. Iface는 USB interface structure를 가리키는 pointer이며, 이것은 `usb_config_descriptor`의 일부이다. 물론 `usb_config_descriptor`는 `usb_device` structure를 통해서 access 가능하다.²⁵³
- ◆ Int `usb_interface_claimed(struct usb_interface *iface);` - 다른 device driver가 특정 interface를 요구(claim)를 했는지를 확인하는 function이다. 만약 요구 되어지지 않았다면, 0을 return한다.
- ◆ Void `usb_driver_release_interface(struct usb_driver *driver, struct usb_interface *iface);` - 이전에 요구한 interface를 release하기 위해서 사용한다. Disconnect에서는 probe에서 부가적으로 요구된 interface에 대해서 release하지 않아도 된다.

이상에서 USB device driver를 등록하고, 다시 interface를 요구/해제 하는 것 등에 대해서 살펴보았다. 다음에서 보아야 할 것은 이렇게 요구되거나 등록시키고자 하는 USB device driver에서 USB device를 configuration하는 방법을 보도록 하겠다.

10.10. Configuring USB devices

다음에서 살펴보게 될 API들은 descriptor에 대한 선택(select)과 질의(query), 혹은 device의 setting을 configure하거나 바꾸어주는 역할을 하는 function들이다. 이와 같은 것들은 USB device에 대한 control transfer의 형태로 주어진다.²⁵⁴

먼저 보아야 할 것은 USB device structure이다. 이것은 `usb.h`에 정의 되어 있으며, USB subsystem에 의해서 저장된다. 저장된 정보들은 각종의 descriptor에 대한 정보를 찾거나 바꾸고자 할 때 유용한 구실을 할 수 있다. 앞에서 이러한 정보들로는 어떠한 것들이 있는지 살펴보았었다.

²⁵³ 이것은 또한 Probe function에서 argument로 넘겨 받게 된다.

²⁵⁴ Control, bulk data, interrupt, isochronous data transfer 등이 있음을 상기하기 바란다. 여기서 isochronous data transfer란 등속 전송을 말하는 것으로 일정한 양의 data를 일정한 속도에 맞춰서 계속적으로 전송하는 것을 말한다. 주로 multimedia data의 전송에서 그 예를 찾을 수 있겠다. 예를 들어서 USB PC camera에서 image를 읽어오는 것을 들 수 있다.

```

struct usb_device {
    int devnum;                      /* Device number on USB bus */
    int slow;                         /* Slow device? */

    atomic_t refcnt;                 /* Reference count */

    unsigned int toggle[2];           /* one bit for each endpoint ([0] = IN, [1] = OUT) */
    unsigned int halted[2];            /* endpoint halts; one bit per endpoint # & direction; */
                                      /* [0] = IN, [1] = OUT */

    struct usb_config_descriptor *actconfig; /* the active configuration */
    int epmaxpacketin[16];            /* INput endpoint specific maximums */
    int epmaxpacketout[16];           /* OUTput endpoint specific maximums */

    struct usb_device *parent;
    struct usb_bus *bus;              /* Bus we're part of */

    struct usb_device_descriptor descriptor; /* Descriptor */
    struct usb_config_descriptor *config; /* All of the configs */

    int have_langid;                /* whether string_langid is valid yet */
    int string_langid;               /* language ID for strings */

    void *hcpriv;                   /* Host Controller private data */

/* usbdevfs inode list */
    struct list_head inodes;
    struct list_head filelist;

/*
 * Child devices - these can be either new devices
 * (if this is a hub device), or different instances
 * of this same device.
 *
 * Each instance needs its own set of data structures.
 */
    int maxchild;                    /* Number of ports if hub */
    struct usb_device *children[USB_MAXCHILDREN];
};


```

코드 628. `usb_device` 구조체의 정의

`usb_device` structure는 모든 USB의 descriptor들에 대한 root이다. 따라서, 이것을 통해서 다른 descriptor에 대한 접근이 가능하게 된다. 때에 따라서는 `device`를 `configure`하거나 `transfer request`를 적절하게 설정하기 위해서 이러한 descriptor들을 parsing할 필요가 있다. 아래의 코드는 이와 같은 경우에 모든 descriptor들을 접근하는 방법을 보여준다.

```

/* 모든 가능한 configuration descriptor를 access하는 방법 */
for ( I = 0; I < dev->descriptor.bNumConfigurations; I++)
{
    struct usb_config_descriptor *cfg = &dev->config[I];
    ...
}

/* 특정의 configuration descriptor에서 모든 가능한 interface descriptor를 access 하는 방법 */
for ( j = 0; j < cfg->bNumInterfaces; j ++ )
{
    struct usb_interface *ifp = &cfg->interface[j];
    ...
}

/* 특정 interface에서 모든 alternate setting을 access하는 방법 */
for ( k = 0; k < ifp->num_altsetting; k ++ )
{
    struct usb_interface_descriptor *as = &ifp->altsetting[k];
    ...
}

/* 특정 alternate setting 의 모든 endpoint descriptor를 access 하는 방법 */
for ( l = 0; l < as->bNumEndpoints; l ++ )
{
    struct usb_endpoint_descriptor *ep = &as->endpoint[l];
}

```

코드 629. 모든 usb device 디스크립터에 대한 접근 예

Active한 configuration을 접근하고자 할 경우에는 dev->actconfig를 사용하면 될 것이다. 이것은 항상 현재 active한 configuration을 가리키는 역할을 하기 때문이다. 또한 active alternate setting을 access하기 위해서는 interface의 &ifp->altsetting[ifp->act_altsetting]을 이용하면 된다.²⁵⁵

이전 USB device에 기본적인 function들을 알아보기로 한다. 여기서 알아볼 것들은 주로 device의 status나 property와 관련된 것들이다. 즉, device의 configuration을 바꾸거나, 특정 device의 descriptor를 읽어보는 경우에 사용되게 된다. 다음과 같다.

- ◆ Int usb_set_configuration(struct usb_device *dev, int configuration); - argument로 주어진 특정의 configuration을 활성화(activate)하는 방법이다.
- ◆ Int usb_set_interface(struct usb_device *dev, int interface, int alternate); argument로 주어진 interface로부터 특정 alternate setting을 활성화 시킨다.
- ◆ Int usb_get_device_descriptor(struct usb_device *dev); - 특정 device로부터 전체 descriptpor를 새로이 읽어 들인다. 이것은 bus에 새로운 device가 붙어서 자동으로 실행되거나, 혹은 USB descriptor의 변화가 있을 경우에 사용될 수 있을 것이다.
- ◆ Int usb_get_descriptor(struct usb_device *dev, unsigned char desctype, unsigned char descindex, void *buf, int size); - 단일의 USB descriptor를 device로부터 읽어 들일 수 있는데, 이 function은 extended나 vendor specific descriptor를 parsing하는데 사용되어진다. 사용하기 위해서는 관련된 USB spec.을 읽어보아야 할 것이다.
- ◆ Int usb_get_string(struct usb_device *dev, unsigned short langid, unsigned char index, void *buf, int size); - string descriptor를 읽어 들이는데 사용한다. USB spec.에서는 string descriptor가 unicode로 되어있다는 점을 유의해야 한다. 성공한다면 0을 그렇지 않다면 error code를 return한다.
- ◆ Int usb_string(struct usb_device *dev, int index, char *buf, size_t size); - 위와 같은 function이지만 ascii string으로 converting한 다음에 return한다는 점이 다르다.
- ◆ Int usb_get_status(struct usb_device *dev, int type, int target, void *data); - USB spec.을 참고하기 바란다.

²⁵⁵ 여기서 ifp는 interface를 가리키는 pointer이다.

- ◆ Int `usb_clear_halt(struct usb_device *dev, int pipe);`; 만약 endpoint가 STALL²⁵⁶상태라면, 이 function을 call해서 STALL condition을 clear할 수 있다.

HID(Human Interface Device)와 관련된 function으로는 다음과 같은 것이 있다. 이와 관련된 사항을 알고자 한다면 HID class specification을 읽어보기 바란다.

- ◆ Int `usb_get_protocol(struct usb_device *dev);`
- ◆ Int `usb_set_protocol(struct usb_device *dev , int protocol);`
- ◆ Int `usb_get_report(struct usb_device *dev, unsigned char type, unsigned char id, unsigned char index, void *buf, int size);`
- ◆ Int `usb_set_idle(struct usb_device *dev, int duration, int report_id);`

이상에서 USB와 관련된 기본적인 구조와 function들에 대한 이야기를 했다. USB는 사용되는 device의 종류가 매우 다양하기에 이를 지원하기 위한 구조와 접근하기 위해서 어떤 driver들이 layer를 이루고 있는지를 살펴보았다. 또한 Windows와의 비교를 통해서 비슷한 구조가 window에서도 USB에 꽂힌 device를 접근하기 위해서 사용된다는 것을 알게 되었다. 다음에서는 실제적인 USB를 이용한 device와 application간의 data 및 control의 transfer에 대해서 알아볼 것이며, 실제로 구현된 code를 통해서 어떻게 구현되었는지를 보게 될 것이다.

10.11. USB Data Transfer

USB data의 전송에는 크게 control, bulk, interrupt, isochronous와 같은 4가지의 종류가 있다. 또한 이것과 관련된 데이터 구조로는 USB Request Block(URB)라는 것을 가지고 있으며, 관련된 함수와 매크로도 가지고 있다.. 이하에서는 이러한 것들에 대해서 살펴보도록 한다.

먼저 USB에서의 데이터의 전송은 transaction이라고 부르는 여러 개의 조각으로 나누어서 요구(request)가 처리된다. 매 밀리세컨드(millisecond)마다 transaction은 프레임(frame)으로 구성되어서 전송되게 되는데, [그림63]을 참고하도록 하자.

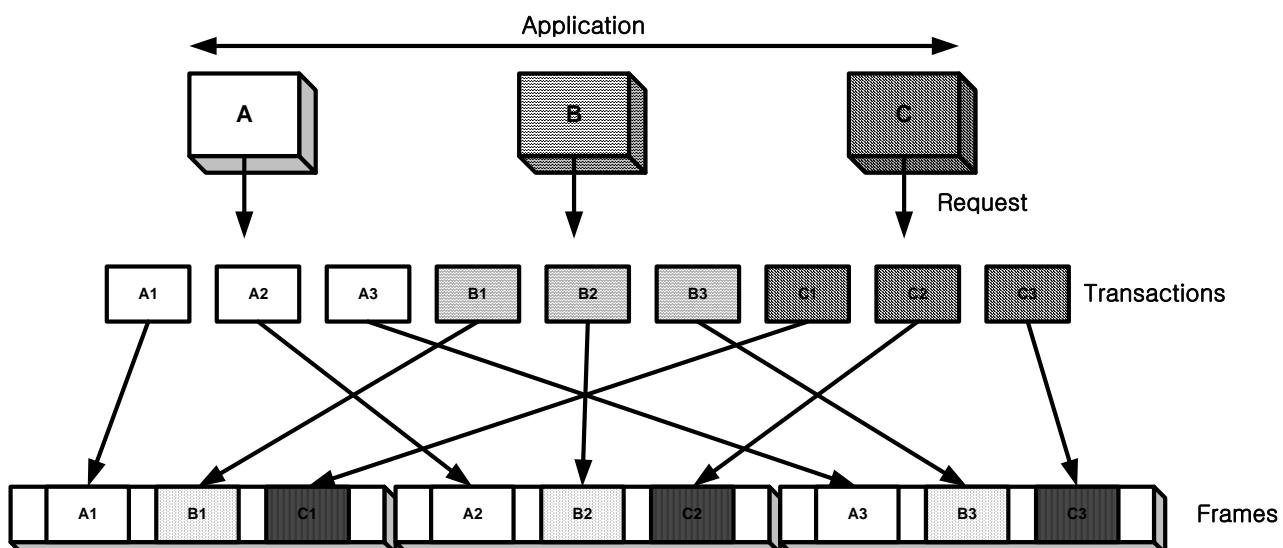


그림 78. USB의 data transfer 구조.

²⁵⁶ Data를 주고 받을 수 없는 상태를 말하거나, control pipe request가 지원되지 않을 경우를 가리킨다.

그림에서 보듯이 하나의 요구는 여러 개의 transaction으로 나누어져서 전송되어지며, 또한 transaction은 프레임에 여러 개가 동시에 실려서 전송되어짐을 볼 수 있다. Transaction에 대해서 조금 더 살펴보면 [그림 64]와 같이 이루어져 있다.

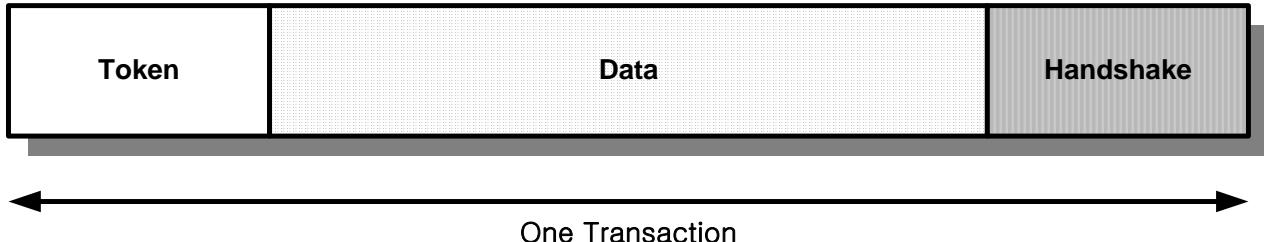


그림 79. USB Transaction의 구조.

Transaction은 phase라는 것으로 이루어져 있으며, 이러한 phase에는 token, data, handshake packet이 있다. Transaction의 형(type)에 따라서 데이터와 handshake phase가 선택적으로(optional) 따라오게 된다. Token phase에는 디바이스 주소(address)와 endpoint 수(number)가 주어지며, 단지 지정된(addressed) 장치만이 transaction을 처리하게 된다. Data phase동안에는 데이터가 버스(bus)상에 올려지며, input과 output에 따라서, host와 장치의 역할이 정해지게 된다. 즉, input일 경우에는 host가 데이터를 읽어드리고, output일 경우에는 장치가 데이터를 읽어드리게 된다. 물론 데이터를 버스에 놓는 것은 그 반대가 될 것이다. Handshake phase의 경우에는 host나 장치가 상태(status) 정보를 주는 패킷(packet)을 교환한다. 이러한 상태정보를 교환하는 패킷으로는 ACK, NAK, STALL이 있다. ACK는 host 혹은 장치가 데이터를 수용(accept)했다는 것을, NAK는 바쁜거나(busy) 받지 못했음을, 그리고 STALL은 transaction은 받아드렸으나, 논리적으로 문제가 있는 경우를 나타내준다. 만약 handshake 패킷이 없다면, 에러(error)를 가리키며, transaction이 다시 한번 전송(transmit)된다.

Transaction이 일어나게 되는 궁극적인 장소는 endpoint로서 [그림65]는 endpoint의 상태(state)를 나타낸 것이다. 즉, endpoint는 각각의 상태(state)에 맞게 transaction에 반응을 하게 된다.

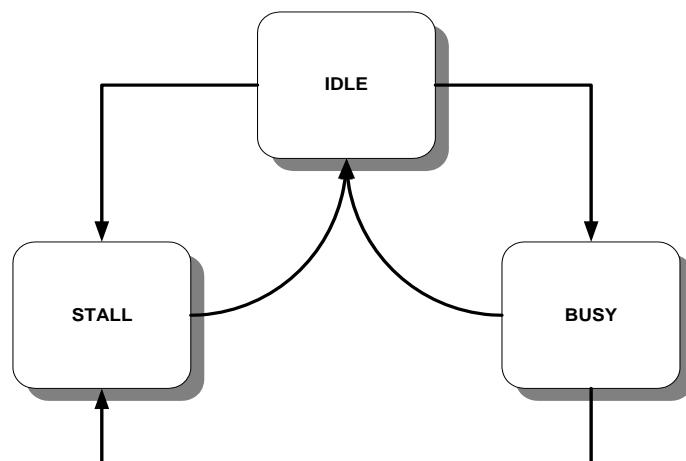


그림 80. Endpoint의 State Transition.

Idle 상태에서는 endpoint는 host에 의해서 시도되는 새로운 transaction을 처리할 준비가 된 상태이며, busy 상태는 transaction을 처리한다고 바쁜 상태로서 새로운 transaction을 처리하지 못한다. 만약 busy 상태인 endpoint에 transaction의 처리를 부탁하게 되면, device는 NAK handshake 패킷으로 답하게 되며, host는 다시 한번 나중에 처리를 요구하게 된다. Device가 내부적으로 자신의 험수에서 에러를 탐지하게 되면, 장치는 STALL handshake 패킷을 현재의 transaction에 대해서 host로 보내고, stall 상태로 움직이게 된다. Control endpoint는 자동적으로 새로운 transaction을 받게 되면, 이러한 stall 상태에서 자동적으로 회복하게

되지만, 만약 다른 stall 상태에 있는 endpoint로 host에서 요구를 보내고자 한다면, clear시키라는 control 요구를 먼저 보내주어야 한다.

이제는 각각의 요구에 대해서 알아보도록 하자. 즉, control, bulk, interrupt, isochronous 요구에 대한 것을 하나하나 알아보도록 하자.

- ◆ Control transfer : control information을 control endpoint로부터 받거나, 전송하고자 할 때 사용한다. 예를 들어서 device에 있는 descriptor를 host가 읽고자 하는 경우에 input control transfer를 할 수 있을 것이다. [그림66]은 control transfer와 관련된 phase를 나타낸 것이다.

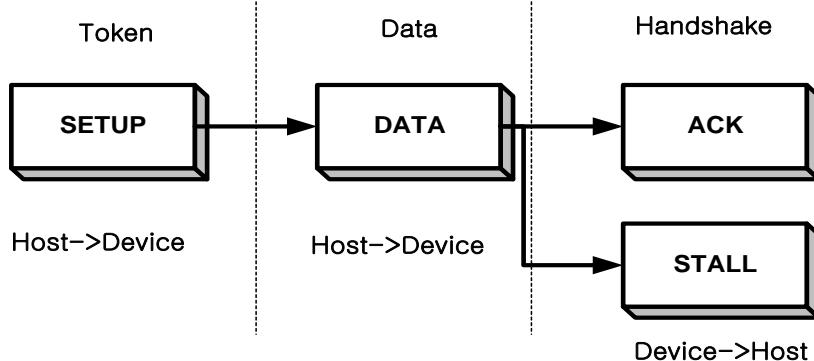


그림 81. Control Transfer Phase

- ◆ Bulk transfer : bulk endpoint로부터 64 bytes의 data를 전송하는데 사용한다. IN/OUT의 두 가지 경우가 있으며, IN의 경우에는 읽어오기, OUT의 경우에는 data를 device로 쓰기가 될 것이다. Control transfer와 같이 lossless한 data의 전송을 보장하지만, 보장된(guaranteed) latency가 없다. 각각의 패킷들은 전송되기 전에 스케줄(schedule)되기 때문에, host는 전송을 위한 충분한 대역폭(bandwidth)이 확보되었을 때 bulk transfer를 스케줄링 한다. [그림67]은 bulk transfer의 phase를 보여준다.

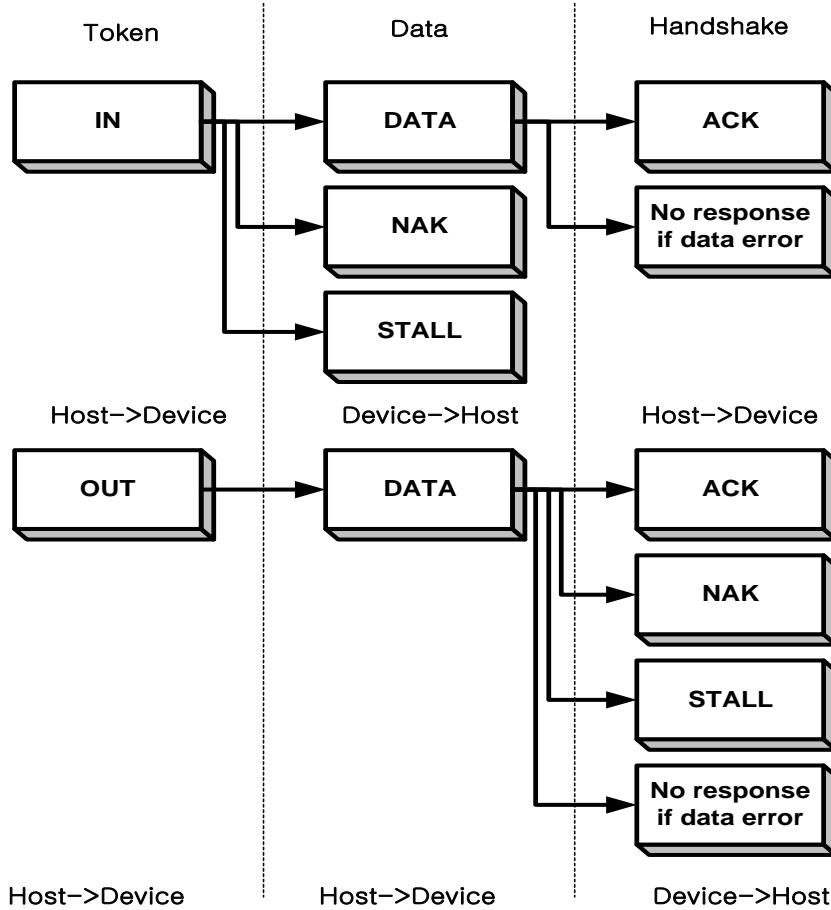


그림 82. Bulk/Interrupt transfer phase.

Host측에서는 장치에 데이터나 인터럽트를 전송하기 위해서 IN token 패킷을 보내서, 장치가 전송할 것을 명하고, 장치에서는 그에 대한 상태(status)나 데이터를 다시 host로 전송한다. 이에 대해서 다시 host에서는 데이터 패킷에 대한 상태(status)를 device로 알려주게 된다. Host에서 장치로 데이터를 전송하기를 원할 때는 OUT token을 보내고, 다시 데이터나 인터럽트를 장치에 전송하게 되며, 이에 대한 상태(status)를 장치는 host에 다시 알려주게 된다.

- ◆ **Interrupt transfer :** 기본적으로 bulk transfer와 동일하나 interrupt endpoint로부터, 혹은 interrupt endpoint로 64 bytes의 데이터를 전송한다. 특이한 점은 interrupt endpoint와 관련되어 polling interval 값으로 1에서 255 millisecond값을 가진다는 것이다. Host는 적어도 polling interval에 endpoint로 IN 혹은 OUT transaction을 하기위한 충분한 대역폭을 예약해야 한다. 여기서 한가지 주의해야 할 점은 USB는 비동기 인터럽트를 지원하지 않고, polling기반으로 움직인다는 점이다.²⁵⁷([그림74]를 참고하기 바란다.)
- ◆ **Isochronous transfer**는 1023 bytes의 data를 isochronous endpoint로부터, 혹은 isochronous endpoint로 전송하는 한다. 이것은 시간에 민감한 data의 전송에 적합하며, 주로 음성(audio) 데이터와 같은 것을 전송할 때 사용 되어질 수 있다. 주기적이며, 일정한 양의 데이터 크기를 보내주어야 할 경우가 될 것이다. 하지만, data의 오류가 있더라도 재전송(retransmission)은 행해지지 않는다는 점에서 다른 transfer 형(type)과는 다른 면을 보인다. [그림68]은 isochronous transfer의 phase를 보여준다.

²⁵⁷ 기본적으로 interrupt에는 두 가지 방식이 있으며, asynchronous interrupt인 경우에는 interrupt가 외부에서 어느 시점에서나 일어날 수 있으며, 이때 적절한 처리를 해주어야 한다는 것이며, polling의 경우는 주기적으로 CPU가 interrupt가 외부에서 발생하는지를 확인한다는 차이점이 있다.

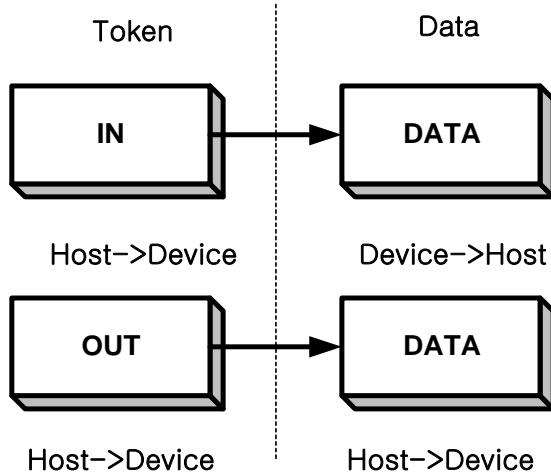


그림 83. Isochronous transfer phase.

지금까지 우리는 USB에서 일어날 수 있는 여러 가지의 data transfer 형(type)들에 대해서 알아보았다. 이제 실제적인 것으로 조금 더 깊이 들어가서, 데이터의 전송과 관련된 코드에서 사용하기 위한 자료구조와 함수 및 연관된 매크로에 대해서 알아보도록 하겠다.

10.12. URB(USB Request Block)

가장 먼저 살펴볼 것은 URB(USB Request Block)에 대한 것이다. 모든 USB의 전송(transfer)에 관련되어 하위의 드라이버와 통신하기 위한 기본 자료구조(data structure)라고 생각하면 될 것이다.²⁵⁸ 기본적인 구조는 아래와 같다.

```

// Isochronous packet descriptor.
typedef struct
{
    unsigned int offset;
    unsigned int length;           // expected length
    unsigned int actual_length;
    unsigned int status;
} iso_packet_descriptor_t, *piso_packet_descriptor_t;

struct urb;
typedef void (*usb_complete_t)(struct urb *);

// URB data structure
typedef struct urb
{
    spinlock_t lock;              // lock for the URB
    void *hcpriv;                // private data for host controller
    struct list_head urb_list;   // list pointer to all active urbs
    struct urb *next;             // pointer to next URB
    struct usb_device *dev;       // pointer to associated USB device
    unsigned int pipe;            // pipe information
    int status;                  // returned status
    unsigned int transfer_flags; // USB_DISABLE_SPD | USB_ISO_ASAP | USB_URB_EARLY_COMPLETE
    void *transfer_buffer;        // associated data buffer
}

```

²⁵⁸ Windows에서도 마찬가지로 URB라는 것을 정의해서 USBDI와 통신을 하고 있다. SCSI interface에서는 SCSI Request Block이라는 것을 두고 있다는 점에서 유사하다고도 할 수 있을 것이다. 즉, 모든 data의 이동은 정해진 interface format에 따르도록 하고 있다.

```

int transfer_buffer_length;           // data buffer length
int actual_length;                 // actual data buffer length
unsigned char *setup_packet; // setup packet (control only)
//
int start_frame;                  // start frame (iso/irq only)
int number_of_packets;            // number of packets in this request (iso/irq only)
int interval;                     // polling interval (irq only)
int error_count;                  // number of errors in this transfer (iso only)
int timeout;                      // timeout (in jiffies)
//
void *context;                   // context for completion routine
usb_complete_t complete;          // pointer to completion routine
//
iso_packet_descriptor_t iso_frame_desc[0];
} urb_t, *purb_t;

```

코드 630. USB 데이터 구조체의 정의

Linux 하위 시스템(subsystem)에서는 단지 하나의 자료구조를 사용하는데, 그것이 URB이다 이 자료구조는 USB에서의 모든 전송(transfer)을 설정(setup)하는데 필요한 매개변수(parameter)를 포함하고 있다. 모든 전송요구는 USB core쪽으로 비동기적으로(asynchronous) 전달되며, 요구의 완료(completion)은 callback 함수로 알려진다(signaled).

크게 URB 자료구조에서의 필드 값들은 input, output 혹은 input과 output으로 사용되어진다. 각각의 의미를 중요한 field를 중심으로 살펴보도록 하자.

- ◆ Dev : usb_device structure에 대한 포인터이다. (Probe 함수를 참고하기 바란다.) 이 field는 input으로 반드시 필요하다.
- ◆ Pipe : Endpoint 번호와 특성(properties)을 encoding하기 위해서 필요한 부분이다. [그림69]는 파이프와 연관되어 Host와 USB 장치의 관계를 조금 더 명확히 보여준다. 즉, Control 파이프는 USB 장치와 시스템 소프트웨어 간의 논리적이(logical) 부분에 대한 데이터 전송을, 데이터 파이프는 USB 장치의 기능적인(functional) 부분과 Host 컴퓨터의 응용 프로그램 소프트웨어간의 통신을 전달한다. 따라서 통신을 위한 가장 기본적인 조건은 파이프가 된다.

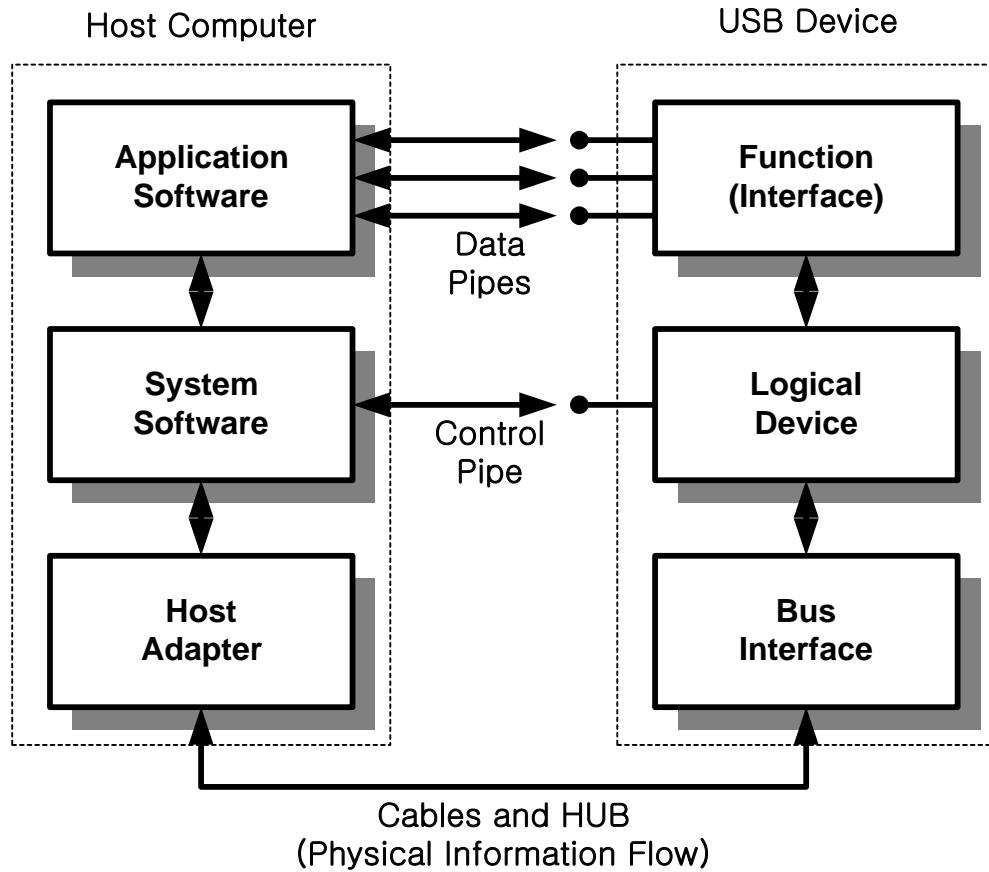


그림 84. Host와 device간의 pipe를 이용한 통신.

관련된 사용되는 매크로로는 다음과 같은 것이 있다.

- Pipe = `usb_sndctrlpipe(dev, endpoint)`, `pipe = usb_rcvctrlpipe(dev, endpoint)` – 주어진 endpoint에 대해서 downstream(send)과 upstream(receive) control 전송을 위한 파이프를 생성한다. Dev는 `usb_device structure`를 가리키고, endpoint는 0이다.
- Pipe = `usb_sndbulkpipe(dev, endpoint)`, `pipe = usb_rcvbulkpipe(dev, endpoint)` – 주어진 endpoint에 대해서 downstream(send)과 upstream(receive) bulk 전송을 위한 파이프를 생성한다. Endpoint의 값으로는 1에서 활성화(active)된 endpoint descriptor에 따라서 15까지의 값이 대입된다.
- Pipe = `usb_sndintpipe(dev, endpoint)`, `pipe = usb_rcvintpipe(dev, endpoint)` – 주어진 endpoint에 대해서 downstream(send)과 upstream(receive) interrupt 전송을 위한 파이프를 생성한다. 역시 활성화된 endpoint descriptor에 따라서 endpoint의 값은 1에서 15까지의 값을 가질 수 있다.
- Pipe = `usb_sndisopipe(dev, endpoint)`, `pipe = usb_rcvisopipe(dev, endpoint)` – 주어진 endpoint에 대해서 downstream(send)과 upstream(receive) isochronous 전송을 위한 파이프를 생성한다. Endpoint의 값으로는 1에서 15까지 활성화된 endpoint descriptor의 값에 따라서 정해진다.
- ◆ Transfer_buffer : input으로 반드시 있어야 하며, 장치로 흘은 장치로부터의 전송될 데이터를 가지고 있는 부분이다. 이 버퍼는 페이지ing이 되지 않는 연속된 물리적 메모리 블록(non-pageable²⁵⁹ contiguous physical memory block)으로 할당(allocate)되어야 하며, 다음의 함수를 이용해서 가능하다.

²⁵⁹ Device driver에서 사용하는 memory들은 kernel mode내에서 사용되기 때문에, paging이 되어버린 memory라는 다시 system으로 읽어드려야 사용이 가능하다. 이것은 kernel에 다시 제 진입이 되어 문제를 일으킬 소지가 많아진다. 때문에 device driver에서 pageable한 memory리를 중요한 data에 대해서 쓰는

- Void *kmalloc(size_t, GFP_KERNEL);
- ◆ Transfer_buffer_length : input을 위해서 반드시 있어야 하며, 전송버퍼(transfer buffer)의 크기를 byte단위로 나타낸다. Endpoint가 전송할 수 있는 최대 크기는 endpoint descriptor에서 wMaxPacketSize 필드를 보면 된다. 하지만 모든 control 전송에 사용되는 default endpoint 0에 대해서는 descriptor가 존재하지 않기에 usb_device structure의 maxpacketsize를 참고한다. 인터럽트나 control 전송에 대해서는 endpoint와 연관된 maximum packet size보다 작거나 같아야 하며, bulk transfer에 대해서는 wMaxPacketSize보다 커진 경우에 자동적으로 더 작은 조각들로 나누어진다.
- ◆ Complete : input에 대해서 선택적인(optional) 부분이다. USB 요구는 비동기적으로(asynchronous) 처리가 되기에 complete는 처리가 끝난 시점에서 특정한 함수가 호출되는 것을 허락한다. 이 함수의 목적은 가능한 한 빨리 요구와 관련된 특정한 부분을 처리하기 위한 것으로, 주로 host controller의 하드웨어 인터럽트와 연관되어 있다. 예를 들면 전송 후나 데이터를 받았을 경우에 버퍼를 관리하는 부분이 있을 수 있을 것이다. 이것들은 전부 요구가 끝난 후에 바로 해주어야 할 일들이다.
- ◆ Context : input에 대해서 선택적인 부분이다. 요구와 연관된 환경(context)으로 주어진다. 아래의 코드는 위의 complete handler와 context의 간단한 사용 예를 보여준다.

```
void complete( struct urb *purb )
{
    struct device_context *s = purb->context;
    /* wake up sleeping requester */
    wake_up( &s->wait );
}
```

코드 631. complete() 함수에서 context의 간단한 사용 예

예에서 보듯이 하나의 요구가 끝나면, 나머지 그 동안 요구가 끝나기를 기다리고 있던 다른 requester들에 대해서 신호(signal)를 보내서 새로이 요구가 진행되어지도록 만들어 준다.

- ◆ Transfer_flags : input과 return 값으로 선택적인 부분이다. Transfer request를 처리에 변화를 주기 위해서 사용한다. 다음과 같은 플래그(flag)들이 있다.
 - USB_DISABLE_SPD – short packet을 사용하지 못하게 한다. Short packet이란 요구와 관련된 endpoint의 maximum packet size보다 작은 데이터를 upstream으로 전송하게 될 때 발생하게 된다.
 - USB_URB_EARLY_COMPLETE – 이 플래그가 명시되면, completion 핸들러가 다른 URB(linked URB)가 제 전송(resubmit)되기 전에 호출되도록 한다.
 - USB_ISO_ASAP – isochronous 요구를 스케줄링하게 될 때 host controller가 가능한 한 빨리 전송(transfer)를 시작하도록 한다. 이것은 isochronous 전송이 현재의 프레임 번호와 동기화(synchronous)될 필요가 없을 경우에 유용하다. 현재 프레임 번호(Current frame number)란 11 bit크기의 카운터로서 매 millisecond마다 증가한다.
 - USB_ASYNC_UNLINK – URB가 취소(cancel)되어야 할 경우 비동기적, 혹은 동기적으로 일어날 수 있다. 이 플래그가 명시되면, 비동기 unlink(cancel)로 바꿔(switch)게 된다.
 - USB_TIMEOUT_KILLED – host controller에 의해서 설정(set)되며 URB가 timeout에 의해서 끝났음을(kill) 나타낸다.
 - USB_QUEUE_BULK – bulk 요구에 대한 큐잉(queuing)을 지원하기 위해서 사용된다. 일반적으로 특정 장치의 endpoint에 대해서 하나의 bulk 전송만이 큐잉될 수 있다.
- ◆ Next : input 매개변수로 선택적인(optional)이다. 여러 개의 URB들을 next 포인터를 이용해서 연결(link)시킬 수 있는데, 이것은 USB 요구를 연속적으로 USB core로 전달할 수 있도록 해준다. 연결은 NULL로 끝나며, 연속된 data stream을 URB로 바꿀 때 사용할 수 있을 것이다.

것은 좋지 않다. 이것은 window용의 device driver를 개발하는 경우에도 마찬가지로 적용된다고 할 수 있다. 또한 연속적인(contiguous) memory를 사용하는 것은 device가 훔어진 data를 일일이 찾아서 모아야 할 필요를 덜기에 좋다고 하겠다. 주로 DMA를 사용하는 device driver에서 이런 경우를 많이 찾아볼 수 있다. 물론 scatter and gather라는 것을 지원하는 경우에는 memory에 신경을 쓰는 경우가 줄 것이다.

- ◆ Status : URB에 대한 return 값으로 사용된다. 즉, 현재 진행중인 요구나 이미 끝난 요구에 대해서 상태(status) 값을 돌려줄 때 사용할 수 있다. 만약 성공적으로 요구를 USB core로 전송했다면, -EINPROGRESS가 되돌려질 것이며, 성공적인 completion일 경우에는 0이 return 할 것이다.
- ◆ Actual_length : return 값으로 사용된다. 요구가 완료(complete)된 후에 실제적으로 전송된 byte수를 해아리는데 사용된다.
- ◆ 그 외의 필드로는 각각의 전송 형(transfer type)과 관련된 것으로 각각의 형에 대해서 다음과 같은 필드들이 있다.
 - Bulk transfer : 추가적인 필드가 없음.
 - Control transfer
 - setup_packet : input으로 반드시 있어야 한다. Control 전송은 2내지 3개의 단계(stage)로 구성되며, 첫번째 단계는 setup 패킷을 downstream으로 전송하는 것이다. 이 필드는 setup 데이터에 대한 포인터를 담고 있다. 이 버퍼는 반드시 페이징이 안 되는 연속된 물리적인 메모리 블록에서 할당되어야 한다.
 - Interrupt transfer
 - start_frame : return 값으로 사용된다. 인터럽트가 스케줄링될 때, 첫번째 프레임 번호(frame number)를 나타낸다. -1인 경우에는 인터럽트 전송을 가능한 한 빨리 시작한다.
 - interval : 반드시 있어야 할 input 파라미터로 인터럽트 전송에 대한 간격(interval)을 millisecond 단위로 명시하고자 할 때 사용한다. 1에서 255까지의 값을 가질 수 있으며, 0으로 명시할 경우 한번의 인터럽트만을 발생 시킨다. Interrupt endpoint에 대한 endpoint descriptor의 bInterval 필드를 보면 될 것이다.
 - Isochronous Transfers
 - start_frame : input과 return 값으로 사용된다. Isochronous 패킷의 스케줄링인 경우, 첫번째 프레임 번호를 명시한다. start_frame을 설정(setting)하는 것은 endpoint로부터의 동기적인(synchronous) 전송을 허용하며, USB_ISO_ASAP 플래그가 있을 때 isochronous 전송의 스케줄링인 경우 첫번째 프레임 임을 나타내주기 위해서 되돌려 진다.
 - number_of_packets : input으로 반드시 주어야 한다. 여러 개의 요구로 isochronous 전송이 USB core로 전달되며, 하나의 요구 데이터의 크기는 endpoint의 최대 패킷 크기까지 될 수 있다. 이 필드는 isochronous 전송 패킷의 수를 명시한다.
 - error_count : return값이다. 요구가 종료되고 난 다음 URB 상태(status)가 -EINPROGRESS가 아닌 경우 에러가 있는 패킷의 수를 돌려준다. 하나 하나의 전송 요구에 대한 구체적인 정보는 iso_frame_desc 구조체를 살펴보기 바란다.
 - timeout : input으로 주어진다. Jiffies²⁶⁰ 단위로 주어지며, host controller 스케줄링에서 자동적으로 URB를 제거하기 위해서 사용될 수 있다. 만약 timeout이 일어나면 USB_TIMEOUT_KILLED가 전송 플래그에 설정(set)된다. Timeout을 발생시킨 USB 상태(status)에 대한 것은 실제적인 전송 상태(transfer status)에서 알려준다.
 - iso_frame_desc : input으로 반드시 주어야 하며, 모든 isochronous URB의 끝에 붙게 되는 배열이다. 모든 단일의 요구 패킷에 대해서 전송 파라미터를 설정(setup)한다. Offset과 length, actual_length, status를 가진다.

이상에서 우리는 리눅스에서 사용되는 하위의 드라이버 계층(layer)에 대한 요구를 어떻게 하는가에 대해서 URB를 중심으로 살펴보았다. 윈도우에서도 이와 비슷한 인터페이스를 가지고 있으며, 참고로 하자면 아래의 코드와 같다.

```
typedef struct _URB {
    union {
        struct _URB_HEADER
        UrbHeader;
```

²⁶⁰ Kernel에서 내부적으로 경과된 시간을 표시하는 32bit의 값이다. 매 clock tick마다 하나씩 증가한다고 생각하면 된다.

```

struct _URB_SELECT_INTERFACE           UrbSelectInterface;
struct _URB_SELECT_CONFIGURATION       UrbSelectConfiguration;
struct _URB_PIPE_REQUEST              UrbPipeRequest;
struct _URB_FRAME_LENGTH_CONTROL      UrbFrameLengthControl;
struct _URB_GET_FRAME_LENGTH          UrbGetFrameLength;
struct _URB_SET_FRAME_LENGTH          UrbSetFrameLength;
struct _URB_GET_CURRENT_FRAME_NUMBER UrbGetCurrentFrameNumber;
struct _URB_CONTROL_TRANSFER          UrbControlTransfer;
struct _URB_BULK_OR_INTERRUPT_TRANSFER UrbBulkOrInterruptTransfer;
struct _URB_ISOCH_TRANSFER            UrbIsochronousTransfer;

// for standard control transfers on the default pipe
struct _URB_CONTROL_DESCRIPTOR_REQUEST UrbControlDescriptorRequest;
struct _URB_CONTROL_GET_STATUS_REQUEST UrbControlGetStatusRequest;
struct _URB_CONTROL_FEATURE_REQUEST   UrbControlFeatureRequest;
struct _URB_CONTROL_VENDOR_OR_CLASS_REQUEST UrbControlVendorClassRequest;
struct _URB_CONTROL_GET_INTERFACE_REQUEST UrbControlGetInterfaceRequest;
struct _URB_CONTROL_GET_CONFIGURATION_REQUEST UrbControlGetConfigurationRequest;
};

} URB, *PURB;

```

코드 632. Windows USB Request Block(URB)의 데이터 구조

코드에서 보듯이 각각의 요구들에 대한 정의와 그에 연관된 자료 구조들을 정의해 주고 있다. 이것을 받게 되는 하위의 USB driver에서는 각각의 요구에 맞게 형 변환(type cast)을 해서 각 필드들을 접근(access)해 주면 될 것이다. 리눅스에서의 구현과 차이점이라면, 요구에 특정한(specific) 자료 구조를 사용함으로써 필드들에 대한 신경을 덜 쓰고 요구 자체에 관심을 더 주겠다는 의도로 여겨진다. 하지만, 근본적인 차이는 없다고 보면 될 것이다.²⁶¹ 윈도우에서도 URB와 관련된 초기화 함수와 매크로 등을 제공하고 있으며, 이점에 대한 리눅스에서의 지원은 아래에서 살펴볼 것이다.

10.13. URB와 관련된 함수 및 매크로들

URB를 다루기 위해서 제공되는 USB core의 함수로는 다음과 같이 4가지가 있다.

1. `purb_t usb_alloc_urb(int iso_packets);` - URB 구조(structure)가 필요한 경우에 사용한다. Argument로 주어지는 `iso_packets`는 isochronous 전송을 위해, URB 구조의 끝에 들어갈 필요한 수만큼의 `iso_frame_desc` 구조의 개수를 명시해주며, 만약 성공했을 경우에는 0으로 초기화(initialization)된 URB 구조의 포인터를 되돌려주며, 그렇지 않을 경우에는 NULL 포인터가 되돌려진다.
2. `void usb_free_urb(purb_t purb);` - `usb_alloc_urb`로 할당된 URB를 해제 하고자 할 때 사용한다.
3. `int usb_submit_urb(purb_t purb);` - USB core로 전송요구를 비동기적으로(asynchronous) 전달한다. Purb는 할당되어 초기화된 URB에 대한 포인터이다. 성공적이라면 0을, 그렇지 않을 경우에는 적절한 에러 코드를 돌려준다. 함수는 항상 즉각적으로 복귀(return)하기에 블로킹(blocking)이 일어나지 않는다. 따라서, 여러 개의 URB를 다른 endpoint에 대해 기다리지 않고, 보낼 수 있다. 심지어, Isochronous endpoint에 대해서는 하나의 endpoint에 대해서 여러 개의 URB를 보내는 것도 가능하다. 이러한 것은 USB 프록토콜의 control, bulk, interrupt 전송에 대한 에러 처리(error handling)와 재전송(retransmission)에 기인한다.
4. `int usb_unlink_urb(purb_t purb);` - 스케줄링²⁶²된 요구를 마치기 전에 취소(cancel)하기 위해서 사용된다. Purb는 이미 전달(submit)된 URB 구조체에 대한 포인터이며, `transfer_flag`이 `USB_ASYNC_UNLINK`이냐에 따라서 synchronous 혹은 asynchronous하게 함수가 호출될 수 있다.

²⁶¹ 물론 이것은 각각의 request에 대한 data structure를 살펴보면 확인 할 수 있을 것이다. 예를 들어서 read/write request와 관련된 data structure를 보면 될 것이다.

²⁶² 이전에서도 schedule이란 말을 사용했는데, 모든 USB request들은 전송되기 전에 schedule된다는 점을 기억하기 바란다.

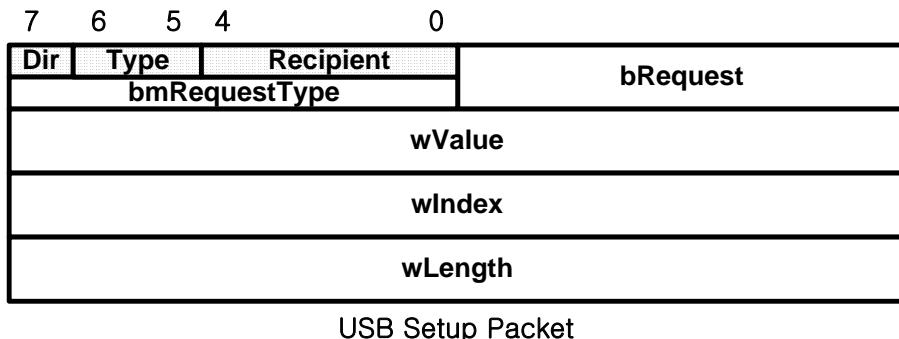
만약 동기적으로 호출되었다면 1ms를 기다리게 되며, 인터럽트나 completion handler에서는 호출되지 않는다. 비동기적으로 호출된다면, 함수는 즉각적으로 복귀하게 되며, return값이 -EINPROGRESS일 경우에는 함수가 성공적으로 시작되었음을 나타낸다. 만약 usb_unlink_urb가 호출된 후에, completion handler가 호출되었고, 동기적인 호출이라면 -ENOENT를, 비동기적으로 호출된 경우에는 -ECONNRESET를 URB 상태(status)가 가지게 된다. 인터럽트 전송 URB에 대해서도 사용될 수 있으며, 인터럽트 전송은 자동적으로 재 스케줄링(rescheduling) 된다. “one shot interrupt” 대해서도 마찬가지다.

URB와 관련되어 지원되는 USB core의 함수들에 대해서 살펴보았다. 이러한 function들 이외에 간단히 URB의 초기화와 관련된 매크로들도 있다.

- FILL_CONTROL_URB(purb, dev, pipe, setup_packet, transfer_buffer, transfer_buffer_length, complete, context);
- FILL_BULK_URB(purb, dev, pipe, transfer_buffer, transfer_buffer_length, complete, context);
- FILL_INT_URB(purb, dev, pipe, transfer_buffer, transfer_buffer_length, complete, context, interval);

즉, control, bulk, interrupt 전송에 대한 URB의 필드들을 초기화한다. 또한 이전 버전(version)의 API와의 호환(compatibility)을 해결할 목적으로 다음과 같은 함수들도 제공한다. 이러한 함수들 중에는 blocking control 전송이나 blocking bulk 전송을 발생시키는데 여전히 사용되는 것도 있다.

- Int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request, __u8 requesttype, __u16 value, __u16 index, void *data, __u16 size, int timeout); - blocking standard control 요구를 발생시킨다. 성공적이라면 전송된 byte수를 돌려하게 되고, 그렇지 않다면 에러 코드를 돌려준다. Control 메시지는 데이터가 전달되기 전에 SETUP 패킷을 만든다. 여기에 들어가는 형식(format)은 [그림70]을 참조로 하기 바란다. [그림70]에서 bmRequestType에 들어가는 값이 requesttype 매개변수에 해당한다.



Dir : 0 – Hosts to Device 1 – Device to Host	Recipient : 0 – Device 1 – Interface 2 – Endpoint 3 – other 4~31 – Reserved
Type : 0 – Standard 1 – Class 2 – Vendor 3 – Reserved	

그림 85. USB Setup Packet의 구조

Setup 패킷의 형식에서 보듯이 control 메시지에 들어가는 파라미터 값과 실제적인 관계를 가지고 있다. Setup 패킷 이후에는 전송하고자 하는 실제적인 데이터 값이 오게 되며, 뒤에 따라오는 필드의 의미는 요구형(request type)에 따라서 다르다는 사실도 기억하기 바란다. 자세한 사항을 알고자 한다면 USB Spec. 2.0의 9.3절을 읽어보기 바란다. 주로 표준 장치(standard device request)에 대한 요구와 요구형(request type)에 대해서 자세히 논하고 있다. 만약 공급자(vendor)에 특정한 control 메시지를 보내고자 한다면, 공급자가 제공하는 매뉴얼을 자세히

읽어보아야 할 것이다. 리눅스에서는 이곳에서 쓰고자 정의한 요구에 대한 코드를 제공한다. 이 값들은 [그림70]의 bRequest 필드에 들어갈 것이다. 아래의 코드를 참고하기 바란다.

```
/*
 * Standard requests
 */
#define USB_REQ_GET_STATUS          0x00
#define USB_REQ_CLEAR_FEATURE       0x01
#define USB_REQ_SET_FEATURE         0x03
#define USB_REQ_SET_ADDRESS         0x05
#define USB_REQ_GET_DESCRIPTOR      0x06
#define USB_REQ_SET_DESCRIPTOR      0x07
#define USB_REQ_GET_CONFIGURATION   0x08
#define USB_REQ_SET_CONFIGURATION   0x09
#define USB_REQ_GET_INTERFACE        0x0A
#define USB_REQ_SET_INTERFACE        0x0B
#define USB_REQ_SYNCH_FRAME         0x0C

/*
 * HID requests
 */
#define USB_REQ_GET_REPORT          0x01
#define USB_REQ_GET_IDLE            0x02
#define USB_REQ_GET_PROTOCOL         0x03
#define USB_REQ_SET_REPORT           0x09
#define USB_REQ_SET_IDLE             0x0A
#define USB_REQ_SET_PROTOCOL          0x0B
```

코드 633. USB Request들에 대한 정의

- Int `usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe, void *data, int len, unsigned long *actual_length, int timeout);` - blocking bulk transfer를 발생시킨다. Actual_length field는 실제로 전송된 byte수를 가지는 변수에 대한 pointer이다.

다음에서 설명하는 것은 더 이상 사용되지 않는 것들이다.

- Void *`usb_request_bulk(struct usb_device *dev, unsigned int pipe, usb_device_irq handler, void *data, int len, void *dev_id);`
- Int `usb_terminate_bulk(struct usb_device *dev, void *first);`
- Int `usb_request_irq(struct usb_device *dev, unsigned int pipe, usb_device_irq handler, int period, void *dev_id, void **handler);`
- Int `usb_release_irq(struct usb_device *dev, void *handler, unsigned int pipe);`

이상에서 기본적으로 USB client device driver를 작성하기 위한 기본적인 것들을 보았다. 다음에서 볼 것은 실제적인 코드를 보는 것이다. USB client device driver와 관련되어 dabusb driver가 sample로서 usb kernel code에 제공된다. 이 driver가 보여주는 예는 여러 개의 device를 하나의 driver로 제공하는 방법과 interface를 요구(claim)하기, configuration과 alternate setting을 설정하기, .control과 bulk URB를 전달(submitting)하는 방법, isochronous data stream을 읽는 것, 그리고 전원이 켜진 상태에서 USB device를 unplug하는 방법 등을 보여준다.

10.14. USB Error Code & Status Code

```
/*
 * USB-status codes:
 * USB_ST* maps to -E* and should go away in the future
 */
#define USB_ST_NOERROR          0
#define USB_ST_CRC              (-EILSEQ)
```

#define USB_ST_BITSTUFF	(-EPROTO)	
#define USB_ST_NORESPONSE	(-ETIMEDOUT)	/* device not responding/handshaking */
#define USB_ST_DATAOVERRUN	(-EOVERFLOW)	
#define USB_ST_DATAUNDERUN	(-EREMOTEIO)	
#define USB_ST_BUFFEROVERRUN	(-ECOMM)	
#define USB_ST_BUFFERUNDERUN	(-ENOSR)	
#define USB_ST_INTERNALERROR	(-EPROTO)	/* unknown error */
#define USB_ST_SHORT_PACKET	(-EREMOTEIO)	
#define USB_ST_PARTIAL_ERROR	(-EXDEV)	/* ISO transfer only partially completed */
#define USB_ST_URB_KILLED	(-ENOENT)	/* URB canceled by user */
#define USB_ST_URB_PENDING	(-EINPROGRESS)	
#define USB_ST_REMOVED	(-ENODEV)	/* device not existing or removed */
#define USB_ST_TIMEOUT	(-ETIMEDOUT)	/* communication timed out, also in urb->status**/
#define USB_ST_NOTSUPPORTED	(-ENOSYS)	
#define USB_ST_BANDWIDTH_ERROR	(-ENOSPC)	/* too much bandwidth used */
#define USB_ST_URB_INVALID_ERROR	(-EINVAL)	/* invalid value/transfer type */
#define USB_ST_URB_REQUEST_ERROR	(-ENXIO)	/* invalid endpoint */
#define USB_ST_STALL	(-EPIPE)	/* pipe stalled, also in urb->status */

코드 634. USB error 코드 및 status 코드의 정의

참고적으로 앞에서 나열한 함수들의 호출 결과로 가질 수 있는 Linux의 USB 구현에서 사용하고 있는 USB error code 및 status code들에 대한 정의이다. 관련된 함수를 호출했다면, return 되어서 돌아오는 호출 결과를 위에서 보인 값들과 비교해서 어떤 error가 발생했는지, 혹은 USB device가 어떤 상태를 가지고 있는지를 확인해 보아야 할 것이다.

10.15. 예제를 통한 USB device driver의 분석

리눅스에서 예제로서 제공되는 USB driver로는 커널의 소스아래의 device driver이하에 *usb/dausb.c*가 있다. 하지만 우리는 plusb.c를 분석하도록 한다. 이 드라이버는 간단히 두 대의 PC를 연결하여 데이터를 주고 받는 것을 목적으로 한다. 우리는 이 device driver를 분석해 봄으로서 향후에 driver를 작성하는데 있어서 필요한 기초지식을 습득하는 것을 목적으로 하며, 네트워크와 USB를 어떻게 연결해서 볼 것인가를 익히도록 하겠다. 참고로서 볼 수 있는 코드로는 *usb/acm.c* 정도가 더 있을 수 있다. 만약 USB class specification 1.1에서 나와있는 Abstract Control Model과 비교해서 보고자 한다면 *usb/acm.c*를 보는 것이 좋을 것이다. 특히 특정 USB class와 관련된 control 메시지들에 대한 요구를 어떻게 처리하는 가를 보기에는 좋은 참고가 될 것이다. 이하에서는 plusb.c 코드를 참고로 해서 USB 및 network device driver와 관련된 것을 한가지씩 보도록 한다.

10.15.1. USB device driver의 자료구조

아래의 코드는 plusb.c에서 사용하는 USB device를 위한 자료구조를 정의해 놓고 있다. 데이터 구조체의 각 필드의 의미는 comment로 정의되어 있다.

```
typedef struct
{
    struct usb_device usbdev; /* usb device 자료구조를 위한 저장소 */
    int status; /* device driver의 현재의 상태를 나타내는 pointer */
    int connected; /* 연결 상태를 나타냄 */
    int in_bh; /* bottom half가 진행 중 인지를 명시 */
    int opened; /* device가 open되었는지를 나타냄 */
    spinlock_t lock; /* 공유변수를 접근하기 위한 spin lock */
    urb_t inturb; /* interrupt urb를 전송 */
    urb_t bulkurb; /* bulk urb를 전송 */
    struct list_head tx_skb_list; /* 전송측 socket buffer의 list */
    struct list_head free_skb_list; /* 사용 가능한 socket buffer의 list */
    struct tq_struct bh; /* task의 scheduling을 위한 구조체 */
}
```

```

struct net_device net_dev; /* network device로 등록된 device 자료구조를 위한 저장소 */
struct net_device_stats net_stats; /* network device로 등록되어 사용될 때, 상위의 protocol stack에
전송될 device의 상태 정보를 위한 저장소 */
} plusb_t,*ppplusb_t;

```

코드 635. plusb.c의 USB 디바이스 드라이버를 위한 자료구조 정의

먼저 USB device 자료 구조의 포인터(pointer)를 설정하고 있다. 이것은 추후에 각각의 함수로 전달될 변수로서 기억할 목적을 가진다. 각각의 필드(field)에 대한 설명은 리스트 속의 설명으로도 충분하리라 생각하지만, 간단히 덧붙인다면 다음과 같다.(spinlock에 대해서는 이후에 설명하도록 하겠다.)

먼저 우리가 하려는 일은 간단한 PC간의 연결을 지원하는 것이다. 따라서, plusb.c는 network device로서도 커널에 등록되어 있어야 하며, 또한 USB device로도 커널에 등록이 되어 있어야 한다. 전송이 일어나는 데이터의 형식은 소켓버퍼(socket buffer)를 이용하게 되기에, 이것들을 기억할 소켓버퍼 리스트(list)에 대한 자료구조도 있어야 할 것이다. 마지막으로 배터적인 드라이버에 대한 접근을 위한 필드와 bottom half처리를 위한 필드들로 구성된다. 여기서 bottom half란 커널에서 빠른 데이터의 처리를 요구할 경우에 사용되는 방법으로 하드웨어와 관련된 것을 먼저 처리하고 이후에 다시 소프트웨어적으로 뒤따르는 처리들을 해주도록 커널에 스케줄링을 요구하는 것이다. 이는 지나친 자원의 점유를 막고 커널의 효율을 증대 시킬 목적으로 사용되는 방법이다. 정확한 정의는 나중에 안전한 시간을 골라서 수행되도록 스케줄링되는 함수(routine)를 말한다.

USB device driver를 위해서 반드시 있어야 할 자료구조로는 다음과 같은 것이 있다. 이에 대한 설명은 앞에서 설명하였다.

```

static struct usb_driver plusb_driver =
{
    name: "plusb",
    probe: plusb_probe,
    disconnect: plusb_disconnect,
};

```

코드 636. USB device driver로 반드시 가져야 할 usb_driver구조체의 정의

먼저, 장치의 이름을 나타내는 필드와 USB subsystem이 해당하는 드라이버를 검출할 때 사용하는 함수와 사용을 그만하고자 할 때 사용하는 함수에 대한 것이 필드의 값으로 온다. 이 함수들에 대해서는 아래에서 보기로 한다.

10.15.2. USB device driver의 초기화 및 제거하기

USB device driver를 초기화하는 것은 먼저 모듈(module)을 초기화 하는 것과 그와 관련된 함수부분에서 USB device를 찾아서 초기화 하는 부분으로 이루어 졌다. 모듈을 초기화 하기 위해서는 [리스트 4]와 같이 하면 된다.

```

int __init init_module (void)
{
    return plusb_init ();
}

void __exit cleanup_module (void)
{
    plusb_cleanup ();
}

```

코드 637. 모듈의 초기화

즉, init_module()과 cleanup_module 두개의 함수가 주어져서 커널이 모듈을 적재(load)하고 내리는데(unload) 사용하게 되는 것이다. 여기서 중요한 것은 이 코드는 반드시 MODULE로 정의된 후에

실행된다는 것이다. 그렇지 않을 경우에는 단순히 `_init`와 `_exit`뒤에 관련된 함수를 선언해 주도록 하자. 이것은 커널이 device driver를 초기화 하는 부분이 된다.

USB device driver를 위한 초기화 부분을 살펴보도록 하자. 이곳에서 하는 일은 주로 사용하게 될 device driver의 데이터 구조에 대한 초기화와 등록이다.

```
int __init plusb_init (void)
{
    unsigned u;
    dbg("plusb_init");

    /* initialize struct */
    for (u = 0; u < NRPLUSB; u++) {
        plusb_t s = &plusb[u];           /* plusb[]에서 하나의 구조에 대한 포인터 얻기*/
        memset (s, 0, sizeof (plusb_t));   /* 0로 초기화 한다.*/
        s->bh.routine = (void (*) (void ))plusb_bh; /* bottom half 처리를 위한 함수의 포인터 보관*/
        s->bh.data = s;                /* bottom half 처리를 위해 넘겨줄 데이터
보관*/
        INIT_LIST_HEAD (&s->tx_skb_list);      /* Tx socket buffer에 대한 초기화 */
        INIT_LIST_HEAD (&s->free_skb_list);    /* free socket buffer에 대한 초기화 */
        spin_lock_init (&s->lock);          /* spin lock에 대한 초기화 */
    }
    /* register misc device */
    usb_register (&plusb_driver);
    dbg("plusb_init: driver registered");
    return 0;
}
```

코드 638. plusb에 대한 초기화

spin lock은 여러 개의 프로세스가 같은 device driver를 접근하게 될 때, 전역적인 변수를 동시에 access하지 않도록 하는 방법을 제공한다. 즉, 하나의 프로세스만이 전역적으로 사용되는 변수를 접근해서 다른 프로세스와 동기화를 맞추게 되어, 자료구조상의 문제가 생길 여지를 없애주기 위한 것이다. 특히 이러한 것이 많이 사용되는 곳은 device driver에서 있어서는 queue의 관리가 될 것이다. DMA(Direct Memory Access)를 위한 메모리 버퍼들의 queue를 접근할 때 사용하면 될 것이다.

10.15.3. USB device driver의 해제

USB device driver의 해제는 driver에 할당된 자원(resource)들을 해제 하는 것과 커널에 등록된 USB device driver를 제거하는 부분으로 이루어 진다.

```
void __exit plusb_cleanup (void)
{
    unsigned u;

    dbg("plusb_cleanup");
    for (u = 0; u < NRPLUSB; u++) {
        plusb_t s = &plusb[u];           /* USB device driver 자료구조를 하나 선택한다.*/
        if(s->net_dev.name[0]) {
            dbg("unregistering netdev: %s",s->net_dev.name);
            unregister_netdev(&s->net_dev); /* network device driver로 등록된 것을 해제
한다.*/
        }
    }
    usb_deregister (&plusb_driver);      /* USB device driver로 등록된 것을 해제 한다.*/
}
```

```

    dbg("plusb_cleanup: finished");
}

```

코드 639. plub에 대한 해제

현재 작성하고 있는 device driver는 network device driver로서의 역할도 하고 있으므로 이것도 해제해 주어야 할 것이다. 초기화에서 새로이 할당된 버퍼가 없기에 이곳에서는 단순히 driver로 등록된 것을 해제 하는 것으로 일을 끝내게 된다.

10.15.4. USB device의 검출과 연결 끊기

USB device를 검출(probe)하는 것은 아래의 코드와 같다. 크게 보면 USB의 device descriptor를 읽는 부분과 network device로 등록하는 부분으로 이루어 졌다.

```

static void *plusb_probe (struct usb_device *usbdev, unsigned int ifnum)
{
    plusb_t *s;

    dbg("plusb: probe: vendor id 0x%x, device id 0x%x ifnum:%d",
        usbdev->descriptor.idVendor, usbdev->descriptor.idProduct, ifnum);
    /* descriptor를 읽어서 우리가 원하는 device인지를 확인한다. 즉, vendor ID와 product ID를 비교*/
    if (usbdev->descriptor.idVendor != 0x067b || usbdev->descriptor.idProduct > 0x1)
        return NULL;
    /* configuration0이 여러 개일 경우에는 다루지 않도록 한다.*/
    if (usbdev->descriptor.bNumConfigurations != 1)
        return NULL;

    /* 빈 USB device driver의 structure를 하나 찾는다.*/
    s = plusb_find_struct ();
    if (!s)
        return NULL;
    /* USB device에 대한 pointer를 저장한다.*/
    s->usbdev = usbdev;

    /* 쓰고자 하는 configuration을 활성화 한다.*/
    if (usb_set_configuration (s->usbdev, usbdev->config[0].bConfigurationValue) < 0) {
        err("set_configuration failed");
        return NULL;
    }
    /* 쓰고자 하는 interface를 활성화 한다.*/
    if (usb_set_interface (s->usbdev, 0, 0) < 0) {
        err("set_interface failed");
        return NULL;
    }
    /* network device로 등록하고 초기화에 필요한 부분을 등록한다.*/
    if (!s->net_dev.name[0]) {
        strcpy(s->net_dev.name, "plusb%d");
        s->net_dev.init=plusb_net_init;
        s->net_dev.priv=s;
        if(!register_netdev(&s->net_dev))
            info("registered: %s", s->net_dev.name);
        else {
            err("register_netdev failed");
            s->net_dev.name[0] = '\0';
        }
    }
    /* 연결된 상태로 만든다.*/

```

```
s->connected = 1;

if(s->opened) {
    dbg("net device already allocated, restarting USB transfers");
    plusb_alloc(s);
}

info("bound to interface: %d dev: %p", ifnum, usbdev);
/* 모듈의 사용 카운트를 증가 시킨다.*/
MOD_INC_USE_COUNT;
return s;
}
```

코드 640. USB 디바이스 드라이버의 검출과 network 디바이스 드라이버로의 등록

MOD_INC_USE_COUNT를 사용해서 현재 module이 사용되고 있는 회수를 증가 시킴으로써, kernel이 module을 제거하지 않도록 만들어주며, 또한 *register_netdev()*를 사용해서 network device driver로의 역할을 하도록 만들어주고 있다.

비어있는 USB device structure의 자리를 찾는 것은 USB device structure의 배열을 차례로 검색하여 연결(connect)되지 않은 것을 찾아서 돌려주면 된다.

```
static plusb_t *plusb_find_struct (void)
{
    int u;

    for (u = 0; u < NRPLUSB; u++) {
        plusb_t *s = &plusb[u];
        if (!s->connected)           /* 연결상태인지를 확인하고 그렇지 않다면 돌려준다.*/
            return s;
    }
    return NULL;
}
```

코드 641. plusb의 빈 USB 디바이스 구조체 찾기

관련된 부분을 조금 더 찾아서 들어가면, network device를 위한 커널 인터페이스 부분을 찾을 수 있다. 이곳에서 필요한 함수들에 대한 포인터들을 명시해준다.

```
int plusb_net_init(struct net_device *dev)
{
    dbg("plusb_net_init");

    dev->open=plusb_net_open;          /* driver에 대한 open요구를 처리하는 함수를 저장 */
    dev->stop=plusb_net_stop;         /* driver에 대한 stop요구를 처리하는 함수를 저장 */
    dev->hard_start_xmit=plusb_net_xmit; /* driver에 대한 전송(transmission)요구를 처리하는 함수를
저장*/
    dev->get_stats     = plusb_net_get_stats; /* driver의 상태정보를 알려주는 함수에 대한 저장 */
    ether_setup(dev);                /* device structure의 ethernet과 관련된 부분에 대한
초기화*/
    dev->tx_queue_len = 0;           /* Tx queue에 대한 초기화 */
    dev->flags = IFF_POINTOPOINT|IFF_NOARP; /* PPP연결에 대한 setting과 ARP를 하지 않도록
해준다.*/
    dbg("plusb_net_init: finished");
    return 0;
}
```

코드 642. Network 디바이스 드라이버로의 초기화

이곳에서는 PPP로 연결할 경우의 network device driver에 대한 설정을 해주었다. 즉, *dev->flags = IFF_POINTTOPOINT|IFF_NOARP*를 network device의 flag으로 명시해 줌으로서 PPP연결과 ARP(Address Resolution Protocol)을 사용하지 않음으로 표시했다.

나중에 커널에서 network device에 대한 open요구 시에 호출될 부분과, 소켓 버퍼의 형식을 가진 데이터를 전송할 *xmit*함수, 그리고 kernel의 device 상태 정보 요구를 처리할 함수에 대한 pointer도 설정해 주었다. 자세한 것은 앞에서 네트워크 디바이스를 다룰 때 했던 이야기를 보기 바란다.

```
static void plusb_disconnect (struct usb_device *usbdev, void *ptr)
{
    plusb_t *s = ptr;

    dbg("plusb_disconnect");
    s->connected = 0;                                /* 연결되지 않은 상태로 만든다.*/
    plusb_free_all(s);                             /* 관련된 data structure를 해제한다.*/

    if(!s->opened && s->net_dev.name) {           /* network device의 이름은 있으나 open되지 않은
경우*/
        dbg("unregistering netdev: %s",s->net_dev.name);
        unregister_netdev(&s->net_dev);          /* network device driver로의 등록을 해제한다.*/
        s->net_dev.name[0] = '\0';                  /* network device의 이름을 초기화 한다.*/
    }

    dbg("plusb_disconnect: finished");
    MOD_DEC_USE_COUNT;                            /* 모듈의 사용 카운트를 감소 시킨다.*/
}
```

코드 643. plusb의 연결 끊기

USB device driver의 연결을 끊는(disconnect) 부분은 크게 사용된 자료구조와 network device driver로 등록된 것을 해제하는 하는 부분으로 되어 있다. 이곳에서 현재 모듈의 사용 카운트를 감소시키는 것을 볼 수 있다.

이제 남은 것은 실제적인 데이터의 입출력에 대한 요구와 요구의 처리에서 마지막에 행해질 것들에 대한 것이다. 주로 요구가 끝났을 경우에는 버퍼 queue에 대한 연산이 일어나게 되므로 전역적인 변수의 배타적인 사용에 유념해서 보기 바란다. 먼저 보아야 할 것은 buffer의 할당이다.

10.15.5. Buffer의 할당과 해제

커널로부터의 입출력 요구는 network device driver의 경우에는 소켓 버퍼로써 데이터를 넘겨 받는다. 이렇게 넘겨 받은 소켓버퍼는 전송을 위해서 device driver의 queue에 저장되며, bulk data의 전송에서 전달된다. 이곳에서는 이렇게 사용하게 될 소켓 버퍼와 파이프(pipe)를 위한 버퍼의 할당과 해제에 대해서 살펴보자.

```
static int plusb_alloc(plusb_t *s)
{
    int i;
    skb_list_t *skb;

    dbg("plusb_alloc");

    for(i=0 ; i < _SKB_NUM ; i++) {
        /* free socket buffer list는 non-pagable memory에서 할당한다.*/
        skb=kmalloc(sizeof(skb_list_t), GFP_KERNEL);
        if(!skb) {
```

```

        err("kmalloc for skb_list failed");
        goto reject;
    }
    /* 할당한 메모리를 0으로 초기화 */
    memset(skb, 0, sizeof(skb_list_t));
    /* free buffer list에 삽입한다.*/
    list_add(&skb->skb_list, &s->free_skb_list);
}

dbg("inturb allocation:");
/* interrupt의 전송을 위한 URB를 할당한다.*/
s->inturb=usb_alloc_urb(0);
if(!s->inturb) {
    err("alloc_urb failed");
    goto reject;
}

dbg("bulkurb allocation:");
/* bulk data의 전송을 위한 URB를 할당한다.*/
s->bulkurb=usb_alloc_urb(0);
if(!s->bulkurb) {
    err("alloc_urb failed");
    goto reject;
}
/* interrupt URB의 pipe 및 각 field들을 초기화하고, buffer를 pageable memory에서 할당한다.*/
dbg("bulkurb/inturb init:");
s->inturb->dev=s->usbdev;
s->inturb->pipe=usb_rcvintpipe (s->usbdev, _PLUSB_INTPipe);
s->inturb->transfer_buffer=kmalloc(64, GFP_KERNEL);
if(!s->inturb->transfer_buffer) {
    err("kmalloc failed");
    goto reject;
}
s->inturb->transfer_buffer_length=1;
/* interrupt 받기 처리가 끝났을 때 수행될 함수를 등록한다.*/
s->inturb->complete=plusb_int_complete;
s->inturb->context=s; /* 함수 수행 시 넘겨질 필요한 data구조를 context에 보관한다.*/
s->inturb->interval=10; /* polling주기를 설정함 */

dbg("inturb submission:");
/* interrupt URB에 대한 설정을 마치고, 이것을 하위의 드라이버로 전송함.*/
if(usb_submit_urb(s->inturb)<0) {
    err("usb_submit_urb failed");
    goto reject;
}
/* bulk URB에 대한 pipe 및 각 field들에 대해서 설정한다.*/
dbg("bulkurb init:");
s->bulkurb->dev=s->usbdev;
/* 받을 pipe에 대한 설정*/
s->bulkurb->pipe=usb_rcvbulkpipe (s->usbdev, _PLUSB_BULKINPIPE);
/* 받을 data의 buffer를 non-pageable kernel memory에서 할당 받는다.*/
s->bulkurb->transfer_buffer=kmalloc(_BULK_DATA_LEN, GFP_KERNEL);
if(!s->bulkurb->transfer_buffer) {
    err("kmalloc failed");
    goto reject;
}
}

```

```

s->bulkurb->transfer_buffer_length=_BULK_DATA_LEN;
s->bulkurb->complete=plusb_bulk_complete; /* bulk data 받기가 끝났을 경우에 호출될 함수 설정.*/
s->bulkurb->context=s; /* 함수의 호출 시 사용될 context 저장 */
dbg("plusb_alloc: finished");
return 0;
reject:
dbg("plusb_alloc: failed");

plusb_free_all(s);
return -ENOMEM;
}

```

코드 644. plusb의 소켓 버퍼 할당 및 PIPE의 초기화

보내는 측에서는 넘겨받은 소켓 버퍼들을 리스트로 유지하기 위해서 소켓버퍼의 리스트 구조체가 필요하며, 이것은 큐를 구성하게 된다. 하지만 받는 측에서는 새로이 소켓버퍼를 할당해서 받은 데이터를 소켓 버퍼에 차워넣는 과정이 더 필요하다. 이렇게 해서 만들어진 소켓 버퍼는 상위의 모듈로 전달 되게 된다. 이것 역시도 non-pageable 커널 메모리에서 할당 받아야 하며, 해제하는 것은 상위의 모듈이 담당한다. 나중에 이것에 대해서도 살펴보게 될 것이다.

또한 인터럽트와 bulk 데이터의 전송을 위해서 먼저 파이프를 만들고 초기화하는 과정이 필요하다. 즉, 이곳에서는 모든 데이터의 할당과 초기화가 일어난다고 볼 수 있겠다. 이렇게 만들어진 버퍼와 파이프는 나중에 다시 해제를 해주어야 하며, 아래의 코드와 같이 해준다.

```

static void plusb_free_all(plusb_t *s)
{
    struct list_head *skb;
    skb_list_t *skb_list;

    dbg("plusb_free_all");
    /* tq_immediate queue에 있는 작업들에 대한 실행을 요구한다.*/
    run_task_queue(&tq_immediate);
    /*현재 전송 중일지도 모를 interrupt URB를 하나 떼어낸다.*/
    if(s->inturb) {
        dbg("unlink inturb");
        usb_unlink_urb(s->inturb);
    }
    /* 할당된 transfer buffer를 다시 kernel로 돌려준다.*/
    if(s->inturb && s->inturb->transfer_buffer) {
        dbg("kfree inturb->transfer_buffer");
        kfree(s->inturb->transfer_buffer);
        s->inturb->transfer_buffer=NULL;
    }
    /* 할당된 interrupt URB를 해제한다.*/
    if(s->inturb) {
        dbg("free_urb inturb");
        usb_free_urb(s->inturb);
        s->inturb=NULL;
    }
    /* 현재 사용중일지 모를 bulk URB를 하나 떼어낸다.*/
    if(s->bulkurb) {
        dbg("unlink bulkurb");
        usb_unlink_urb(s->bulkurb);
    }
    /* bulk URB를 위해서 할당된 buffer를 kernel memory로 돌려준다.*/
    if(s->bulkurb && s->bulkurb->transfer_buffer) {
        dbg("kfree bulkurb->transfer_buffer");
    }
}

```

```

        kfree(s->bulkurb->transfer_buffer);
        s->bulkurb->transfer_buffer=NULL;
    }
    /* bulk URB를 해제한다.*/
    if(s->bulkurb) {
        dbg("free_urb bulkurb");
        usb_free_urb(s->bulkurb);
        s->bulkurb=NULL;
    }
    /* 할당된 free socket buffer들을 해제한다.*/
    while(!list_empty(&s->free_skb_list)) {
        skb=s->free_skb_list.next;
        list_del(skb);
        skb_list = list_entry (skb, skb_list_t, skb_list);
        kfree(skb_list);
    }
    /* 현재 사용하고 있을지도 모를 Tx측의 socket buffer들을 해제한다. 이것은 kernel에서 넘겨
    받았으므로 이곳에서 해제해 주어야 한다.*/
    while(!list_empty(&s->tx_skb_list)) {
        skb=s->tx_skb_list.next;
        list_del(skb);
        skb_list = list_entry (skb, skb_list_t, skb_list);
        kfree(skb_list);
    }
    dbg("plusb_free_all: finished");
}

```

코드 645. plusb의 소켓 버퍼 해제 및 PIPE해제

모듈의 사용이 끝났을 경우 즉각적으로 커널에서 할당된 메모리는 해제되어야 하며, 그렇지 않을 경우 커널에서 메모리를 사용하지 못하게 되는 경우를 유발 시킨다. 따라서, 드라이버의 개발자는 자신이 사용하는 메모리에 대해서 정확히 추적해서 사용이 끝났을 경우 반드시 해제해 주어야 할 것이다. 이렇게 해서 발생한 문제는 찾기가 힘들기에 디버깅(debugging)하는 수고를 덜고자 하면 꼭 기억하도록 하자.

마지막으로 한가지 짚고 넘어가자면, *run_task_queue(&tq_immediate)*에 대한 것이다. 이 함수는 작업 큐(task queue)에 있는 모든 작업들을 수행하고자 할 때 사용한다. 인수로서 넘겨진 *tq_immediate*는 *<linux/tqueue.h>*에 정의되어 있으며, 시스템 콜의 복귀(return)나 스케줄러(scheduler)가 실행되며, 가능한 한 빨리 실행되는 큐이다. 이 큐는 인터럽트 time에 소모된다. 여기서 말하는 인터럽트 time이라는 것은 어떠한 현재 실행중인 프로세스(process)와도 관련을 맺지 않고 있음을 의미한다. 이와 같은 큐를 사용하는 이유는 드라이버가 인터럽트에 의지하지 않고, 나중에 다시 한번 수행할 기회를 가지기 위해서이며, 커널에서 하드웨어와 관련된 처리를 빨리 해주기 위한 것이다. 이것과 그 외의 작업 queue들을 정리하면 다음과 같은 것들이 있다.

Queue	Description
tq_schedule	스케줄러가 실행될 때 소모되며, 스케줄러는 스케줄링 되어 나가게 될 프로세스의 환경(context)상에서 실행 되기에 스케줄러에서 실행될 작업(task)은 어떠한 일이라도 할 수 있다. 하나 작업은 인터럽트 time에 실행될 수는 없다.
tq_timer	Timer tick에 의해서 실행되며, tick이란 interrupt time에 실행되기에, 이 큐에 속한 모든 작업들은 인터럽트 time에서 실행(run)된다.
tq_immediate	시스템 콜이 복귀(return)하던가 아니면, 스케줄러가 실행될 때 가능한 한 빨리 실행되는 큐이다. 큐는 인터럽트 time에 소비(consume)된다.
tq_disk	커널 1.2에는 없으며, 메모리관리에서 내부적으로 사용된다. 모듈에서는 사용할 수 없다.

표 61. 작업 큐

나중에 데이터의 전송과 관련된 부분에서 큐의 사용 예를 보게 될 것이다.

10.15.6. PLUSB에 대한 network driver 입출력 함수.

먼저 네트워크 드라이버 측면에서 제공되는 함수들에 대한 것을 살펴보고, 그리고 나서 USB 측면에서 어떻게 그러한 요구를 처리하는지를 보도록 하겠다. 먼저 네트워크 드라이버 측면에서 제공되는 함수를 나열하자면, open, stop, hard_start_xmit, get_stats들이 있다. 아래의 코드는 open() 함수를 나타낸 것이다.

```
static int plusb_net_open(struct net_device *dev)
{
    plusb_t *s=dev->priv;

    dbg("plusb_net_open");
    /* 필요한 자료구조를 할당한다.*/
    if(plusb_alloc(s))
        return -ENOMEM;
    s->opened=1;                      /* 현재 open되었음을 표시한다.*/
    MOD_INC_USE_COUNT;                /* 모듈의 사용 카운트를 증가 시킨다.*/

    dbg("plusb_net_open: success");
    return 0;
}
```

코드 646. plusb의 open()함수

Open 함수는 간단히 필요한 자료를 할당하고 open되었음을 표시한 후에 복귀한다. 여기서 유의할 점은 open 시에 USB에 대한 interrupt, bulk 및 소켓 버퍼들이 할당된다는 것이다.

아래의 코드는 stop() 함수를 보여준다. 여기서 하는 일은 이전에 open될 때 할당된 자료구조를 반환하고 연결이 끝났음을 명시해 주고, 모듈의 사용 카운트를 감소시키는 것이다.

```
static int plusb_net_stop(struct net_device *dev)
{
    plusb_t *s=dev->priv;

    dbg("plusb_net_stop");

    plusb_free_all(s);           /* 할당된 자료구조를 반환 */
    s->opened=0;                /* open flag를 지운다.*/
    MOD_DEC_USE_COUNT;          /* 모듈의 사용카운트를 감소시킨다.*/
    dbg("plusb_net_stop:finished");
    return 0;
}
```

코드 647. plusb의 stop()함수

아래의 plusb_net_xmit() 함수는 실제적으로 데이터 전송 요구가 상위에서 주어졌을 경우에 호출되는 함수이다. 이 함수에서 유의해서 살펴볼 부분은 넘겨 받은 소켓버퍼를 list의 형태로 만들어주고 나중에 스케줄링을 요구한다는 것이다.

```
static int plusb_net_xmit(struct sk_buff *skb, struct net_device *dev)
{
    plusb_t *s=dev->priv;
```

```

skb_list_t *skb_list;
int ret=NET_XMIT_SUCCESS; /* return 값으로 일단 성공(success)을 표시한다.*/

dbg("plusb_net_xmit: len:%d i:%d",skb->len,in_interrupt());
/* 현재 connected되지 않았거나, 혹은 free socket buffer list가 없을 경우에
   congestion이 일어났다고 return값을 표시한다.*/
if(!s->connected || list_empty(&s->free_skb_list)) {
    ret=NET_XMIT_CN;
    goto lab;
}
/* free socket buffer list에서 하나를 선택한 다음,
   Tx socket buffer list에 넣고, 이것이 넘겨받은 socket buffer를 가리키도록 만들어준다.
   즉, queue와 같이 만들어준다.*/
plusb_add_buf_tail(s, &s->tx_skb_list, &s->free_skb_list);
skb_list = list_entry(s->tx_skb_list.prev, skb_list_t, skb_list);
skb_list->skb=skb;
skb_list->state=1;

lab:
/* 이미 bottom half가 표시되었는지 확인한다.*/
if(s->in_bh)
    return ret;

dbg("plusb_net_xmit: queue_task");
/* bottom half가 표시 되어있지 않으면, 이것을 표시한 다음 tq_schedule에 넣어서 스케줄링을
   요구한다.*/
s->in_bh=1;
queue_task(&s->bh, &tq_scheduler);

dbg("plusb_net_xmit: finished");
return ret;
}

```

코드 648. plusb_net_xmit()함수

함수 `queue_task(&s->bh, &tq_scheduler)`를 보도록 하자. 이 함수의 정의는 다음과 같다.

- `void queue_task(struct tq_struct *task, task_queue *list);` - task를 큐잉(queueing)하는 함수이다. 경쟁상황(race condition)을 배제하기 위해서 인터럽트를 끊쓰게(disable) 만들며, 모듈의 어디에서도 호출 될 수 있다. 이곳에서 사용되는 `tq_struct`는 다음과 같이 정의 된다.

```

struct tq_struct {
    struct tq_struct *next; /* linked list of active bh's */
    unsigned long sync; /* must be initialized to zero */
    void (*routine)(void *); /* function to call */
    void *data; /* argument to function */
};

```

즉, 스케줄링 되기를 요구하는 함수에 대한 포인터와 처리가 일어날 때 함께 넘겨질 데이터에 대한 포인터를 만들어준 후, 이것에 대해서 `queue_task()`함수를 호출을 하면 되는 것이다.

여기서는 초기화 부분에서 이미 bottom half에서 처리가 일어날 함수와 넘겨질 데이터에 대해서 선언해주었다. 나중에 스케줄러가 실행되면 이렇게 등록한 함수를 하나씩 큐의 리스트를 움직이면서 처리해 줄 것이다. 따라서, 실제적인 데이터의 전송이 일어나게 되는 것은 bottom half로 선언된 함수이다.

plusb_bh()함수는 PLUSB에서 사용된 bottom half 함수를 보여준다. 이곳에서는 스케줄링 되었을 때 Tx 소켓 버퍼 리스트에 있는 소켓 버퍼를 하나씩 bulk transfer로 전송하는 일을 한다.

```
static void plusb_bh(void *context)
{
    plusb_t *s=context;
    struct net_device_stats *stats=&s->net_stats;
    int ret=0;
    int actual_length;
    skb_list_t *skb_list;
    struct sk_buff *skb;

    dbg("plusb_bh: i:%d",in_interrupt());
    /* Tx socket buffer list가 비어 있지 않은 동안 실행한다.*/
    while(!list_empty(&s->tx_skb_list)) {
        /* Tx에 문제가 있다면 break */
        if(!(s->status&_PLUSB_TXOK))
            break;
        /* Tx socket buffer list에서 하나의 socket buffer를 가지고 온다.*/
        skb_list = list_entry (s->tx_skb_list.next, skb_list_t, skb_list);
        /* socket buffer list가 준비되었는지 확인하고, 만약 그렇지 않을 경우 다른 작업이 scheduling되도록 만들어준다.*/
        if(!skb_list->state) {
            dbg("plusb_bh: not yet ready");
            schedule();
            continue;
        }
        /* data를 전송 bulk transfer형태로 전송한다.*/
        skb=skb_list->skb;
        ret=plusb_my_bulk(s, usb_sndbulkpipe (s->usbdev, _PLUSB_BULKOUTPIPE),
                          skb->data, skb->len, &actual_length);
        /* 제대로 data의 전송이 되었는지를 확인하고 그렇지 않을 경우 제 전송을 요구한다.*/
        if(ret || skb->len != actual_length ||!(skb->len%64)) {
            plusb_my_bulk(s, usb_sndbulkpipe (s->usbdev, _PLUSB_BULKOUTPIPE),
                          NULL, 0, &actual_length);
        }
        /* network device driver로서의 상태정보를 기록한다.*/
        if(!ret) {
            stats->tx_packets++;
            stats->tx_bytes+=skb->len;
        }
        else {
            stats->tx_errors++;
            stats->tx_aborted_errors++;
        }
        dbg("plusb_bh: dev_kfree_skb");
        /* 전송한 socket buffer는 이곳에서 해제한다.*/
        dev_kfree_skb(skb);
        /* 준비가 안된 상태로 만든다.*/
        skb_list->state=0;
        /* Tx socket buffer list에서 free socket buffer list로 하나의 entry를 옮겨놓는다.*/
        plusb_add_buf_tail (s, &s->free_skb_list, &s->tx_skb_list);
    }

    dbg("plusb_bh: finished");
}
```

```
/* 마지막으로 bottom half요구가 없음을 표시한다.*/
s->in_bh=0;
}
```

코드 649. plusb_bh()함수

또한 전송되어진 상태 정보의 기록과 소켓 버퍼의 해제, 그리고 소켓버퍼의 리스트에 대한 관리를 담당하고 있다. Bottom half에 대한 처리가 끝났으므로 다시 초기화 시켜주는 것도 잊지 말도록 하자.

plusb_net_get_stats()함수는 network driver로서의 상태 정보를 상위에 알려주기 위한 함수이다. 이것은 `ifconfig -a`와 같은 명령을 실행했을 때 나오는 정보들을 제공해 주기에 반드시 있어야 할 함수이다.

```
static struct net_device_stats *plusb_net_get_stats(struct net_device *dev)
{
    plusb_t *s=dev->priv;

    dbg("net_device_stats");

    return &s->net_stats;           /* net_stats구조체에 대한 포인터를 돌려준다.*/
}
```

코드 650. plusb의 get_stats()함수

이 함수는 단순히 내부에 유지하고 있는 `net_device_stats`구조체에 대한 포인터를 돌려주는 것으로 충분하다. `net_device_stats`구조체는 다음과 같은 필드들로 이루어져 있으며, 이것은 `<linux/netdevice.h>`에 정의되어 있다. 이것 역시 앞에서 이미 network 디바이스 드라이버를 다룰 때 한번 보았던 것이지만, 여기선 간단히 그것을 떠올리기 위해서만 살펴보자.

```
Struct net_device_stats
{
    unsigned long      rx_packets;          /* total packets received */
    unsigned long      tx_packets;          /* total packets transmitted */
    unsigned long      rx_bytes;            /* total bytes received */
    unsigned long      tx_bytes;            /* total bytes transmitted */
    unsigned long      rx_errors;           /* bad packets received */
    unsigned long      tx_errors;           /* packet transmit problems */
    unsigned long      rx_dropped;          /* no space in linux buffers */
    unsigned long      tx_dropped;          /* no space available in linux */
    unsigned long      multicast;           /* multicast packets received */
    unsigned long      collisions;          /* */

    /* detailed rx_errors: */
    unsigned long      rx_length_errors;    /* receiver ring buff overflow */
    unsigned long      rx_over_errors;       /* recvd pkt with crc error */
    unsigned long      rx_crc_errors;        /* recv'd frame alignment error */
    unsigned long      rx_frame_errors;     /* recv'r fifo overrun */
    unsigned long      rx_fifo_errors;       /* receiver missed packet */
    unsigned long      rx_missed_errors;    /* */

    /* detailed tx_errors */
    unsigned long      tx_aborted_errors;
    unsigned long      tx_carrier_errors;
    unsigned long      tx_fifo_errors;
    unsigned long      tx_heartbeat_errors;
    unsigned long      tx_window_errors;

    /* for cslip etc */
}
```

```

    unsigned long      rx_compressed;
    unsigned long      tx_compressed;
};

```

코드 651. net_device_stats구조체의 정의

여기서 중요한 점은 데이터의 전송에 관련된 사항을 항상 추적해서 일일이 상태정보로 남겨 주어야 한다는 점이다. 따라서 데이터 전송 함수의 끝부분이나 혹은 데이터를 받는 함수의 끝부분에서 위의 필드들을 증가 시켜주어야 한다. 나중에 데이터 전송과 받기 부분에서 이것을 사용하는 예를 볼 수 있을 것이다.

USB에 관련된 함수들로 넘어가기 전에 마지막으로 한가지 더 볼 것은 리스트에 대한 연산이다. 드라이버는 여러 개의 연산이 동시에 일어날 가능성이 있기에 서로간의 동기화(synchronization)를 고려하지 않으면 잘 못될 가능성이 많다. 이중에서도 특히 중요한 것이 큐와 관련한 연산이다.

```

/* -----
static int plusb_add_buf_tail (plusb_t *s, struct list_head *dst, struct list_head *src)
{
    unsigned long flags;
    struct list_head *tmp;
    int ret = 0;

    spin_lock_irqsave (&s->lock, flags); /* 배타적인 접근을 위해서 spin lock을 설정한다.*/

    if (list_empty (src)) {           /* 리스트가 비었다면 error를 돌려준다.*/
        // no elements in source buffer
        ret = -1;
        goto err;
    }
    tmp = src->next;                /* source에서 하나의 entry를 선택해서 destination에 넣어준다.*/
    list_del (tmp);
    list_add_tail (tmp, dst);

err:   spin_unlock_irqrestore (&s->lock, flags); /* spin lock을 해제한다. */
    return ret;
}

```

코드 652. plusb의 list관리

PLUSB에서 유념해서 보아야 할 것은 리스트에 대한 연산을 하기 전에 spin lock을 설정한다는 것이다. 물론 이것을 위해선 미리 spin_lock_init()함수를 사용해서 spin lock을 위한 변수를 초기화 시켜주어야 할 것이다. 또한 spin_lock_irqsave()를 호출해서 인터럽트가 일어나지 못하게 하며, 현재의 인터럽트 허용수준을 저장한다. 나중에 이것을 다시 spin_lock_irqrestore()를 호출해서 복귀 restore()시켜준다.

10.15.7. PLUSB에 대한 USB 입출력 함수

이전 PLUSB에서 실제적인 데이터의 입출력이 일어나는 부분에 대해서 보기로 하자. 주로 interrupt와 bulk transfer에 대해서만 다루기로 한다.

```

static int plusb_my_bulk(plusb_t *s, int pipe, void *data, int size, int *actual_length)
{
    int ret;

    dbg("plusb_my_bulk: len:%d",size);
    /* bulk transfer를 행한다. Timeout 값으로 500을 설정함 */
    ret=usb_bulk_msg(s->usbdev, pipe, data, size, actual_length, 500);

```

```

if(ret<0) {
    err("plusb: usb_bulk_msg failed(%d)",ret);
}
/* 현재 pipe가 stall상태에 있으므로 CLEAR_FEATURE요구를 보낸다.*/
if( ret == -EPIPE ) {
    warn("CLEAR_FEATURE request to remove STALL condition.");
    /* HALT상태를 제거(clear)한다.*/
    if(usb_clear_halt(s->usbdev, usb_pipeendpoint(pipe)))
        err("request failed");
}
dbg("plusb_my_bulk: finished act: %d", *actual_length);
return ret;
}

```

코드 653. plusb의 bulk transfer

`plusb_my_bulk()`함수는 bulk transfer가 실제로 일어나는 부분이다. 데이터를 전송하기 위해서 bulk transfer를 위한 pipe를 이용해서 `usb_bulk_msg()`함수로 보내고 있다. 만약 보낸 결과가 멈춤(stall) 상태라면, 이것을 없애기 위해 다시 `usb_clear_halt()`함수를 호출한다.

```

static void plusb_bulk_complete(urb_t *purb)
{
    plusb_t *s=purb->context;
    /* bulk transfer에 대한 처리 결과(status)를 출력*/
    dbg("plusb_bulk_complete: status:%d length:%d",purb->status,purb->actual_length);
    /* 연결 상태가 아니라면 더 이상처리할 필요가 없다.*/
    if(!s->connected)
        return;
    /* 처리결과가 올바른지 확인*/
    if( !purb->status) {
        struct sk_buff *skb;
        unsigned char *dst;
        int len=purb->transfer_buffer_length;
        struct net_device_stats *stats=&s->net_stats;
        /* 새로운 socket buffer를 할당 한다.*/
        skb=dev_alloc_skb(len);
        /* 할당에 문제가 있을 경우 받은 데이터를 drop한다.*/
        if(!skb) {
            err("plusb_bulk_complete: dev_alloc_skb(%d)=NULL, dropping frame",len);
            stats->rx_dropped++;
            return;
        }
        /* 새로이 생성된 buffer의 데이터 부분을 가리키는 포인터를 만든다.*/
        dst=(char *)skb_put(skb, len);
        /* 받은 데이터를 socket buffer로 복사한다.*/
        memcpy( dst, purb->transfer_buffer, len);
        /* socket buffer의 field들을 초기화 한다.*/
        skb->dev=&s->net_dev;
        skb->protocol=eth_type_trans(skb, skb->dev);
        /* 현재 driver의 상태정보를 update한다.*/
        stats->rx_packets++;
        stats->rx_bytes+=len;
        /* socket buffer의 처리를 상위에 알려준다.*/
        netif_rx(skb);
    }
}

```

```

    purb->status=0;
}

```

코드 654. Bulk transfer를 위한 completion 함수

`plub_bulk_complete()`함수는 bulk transfer를 위한 completion 처리(routine)이다. 이것은 미리 bulk transfer를 위한 URB를 할당할 때 이미 정해진 것으로, host controller의 bulk transfer에 대한 인터럽트 핸들러로 호출된다. 모든 인터럽트 핸들러 처리(routine)에 해당하는 제한이 이곳에서도 적용된다고 볼 수 있다. 이곳에서 소켓 버퍼관련 함수와 매크로에 대해서 잠시 정리하고 넘어가도록 하자. 먼저, 소켓 버퍼 구조체의 필드들에 대해서 보면 아래의 표와 같다. (이곳에서는 중요한 필드만 보도록 하겠다. 더 자세히 알고싶다면 `<linux/skbuff.h>`를 참고하기 바란다.)

Field	Description
<code>struct device *dev;</code>	Buffer를 주고 받을 장치를 명시한다.
<code>unsigned char *head;</code> <code>unsigned char *data;</code> <code>unsigned char *tail;</code> <code>unsigned char *end;</code>	데이터를 지정하기 위해서 사용되는 포인터 들이다.
<code>unsigned short protocol;</code>	프로토콜을 나타낸다.
<code>unsigned long len;</code>	실제 데이터의 길이(<code>skb->tail - skb->head</code>)
<code>unsigned char ip_summed;</code>	TCP/UDP checksum에 사용되는 필드로 받은 데이터에 대해서 드라이버가 설정해준다.
<code>unsigned char pkt_type;</code>	PACKET_HOST, PACKET_BROADCAST, PACKET_MULTICAST, 또는 PACKET_OTHERHOST등의 값을 가질 수 있으며, 드라이버가 받은 데이터의 형에 따라서 설정해 준다. Ethernet driver의 경우는 <code>eth_type_trans()</code> 가 이 부분을 처리해준다.

표 62. 소켓버퍼 구조체의 중요 필드들에 대한 설명.

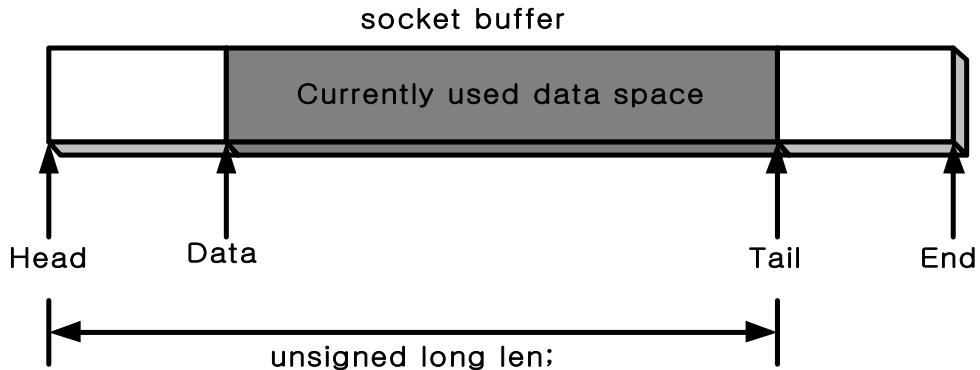


그림 86. socket buffer구조체.

[그림71]은 소켓 버퍼의 head, data, tail, end 필드들 간의 관계를 보여준다. 실제적으로 데이터를 기록하는 것은 tail쪽에서 일어나게 되며, head쪽에는 프로토콜과 관련된 데이터들이 들어간다. 중요 필드들이 다 채워진 소켓 버퍼는 `netif_rx(socket_buffer)`함수를 이용해서 상위로 전달된다.

```

static void plusb_int_complete(urb_t *purb)
{
    plusb_t *s=purb->context;
    /* status 정보를 받는다.*/
    s->status=((unsigned char*)purb->transfer_buffer)[0]&255;
    /* 참고로 아래의 PLUSB에서 사용하는 status정보에 대한 bit 필드 설명을 보도록 하자.

```

```

*   7   6   5   4   3   2   1   0(bit)
*   tx rx  1   0
*-----
*   1   1   10 0000 rxdata
*   1   0   10 0000 idle
*   0   0   10 0000 tx over
*   0   1   10      tx over + rxd
*   _PLUSB_RXD 0x40
*/
#endif
#if 0
if((s->status&0x3f)!=0x20) {
    warn("invalid device status %02X", s->status);
    return;
}
#endif
/* 연결상태인지 확인하고 그렇지 않다면 return한다.*/
if(!s->connected)
    return;
/* Interrupt URB가 Rx쪽 status정보를 가졌는지 확인한다.*/
if(s->status&_PLUSB_RXD) {
    int ret;
    /* Rx bulk URB가 사용 중인지를 확인한다.*/
    if(s->bulkurb->status) {
        err("plusb_int_complete: URB still in use");
        return;
    }
    /* 사용 중이지 않다면, 받기를 요구한다.- receive URB이므로*/
    ret=usb_submit_urb(s->bulkurb);
    /* Return값을 확인한다.*/
    if(ret && ret!= -EBUSY) {
        err("plusb_int_complete: usb_submit_urb failed");
    }
}
/* Status정보를 출력한다.*/
if(purb->status || s->status!=160)
    dbg("status: %p %d buf: %02X", purb->dev, purb->status, s->status);
}

```

코드 655. plusb의 인터럽트 completion함수

plusb_int_complete()함수는 Receive 인터럽트 파이프에 대한 completion 함수를 나타낸 것이다. 즉, 들어온 interrupt에 대해서 Rx(받기)인지를 확인하고, 만약 데이터를 받아야 한다면 Rx를 위한 bulk URB를 .usb_submit_urb() 함수를 이용해 USB 하위 시스템으로 보내서 받겠다고 알려준다. 그리고, Rx bulk에 대한 처리가 끝나면 다시 이전에 설명한 bulk complete이 호출되게 된다²⁶³.

²⁶³ 이곳에서 보여준 USB에 대한 설명은 전부 커널 버전 2.2.X에 대한 것이다. 따라서, 차후에 커널 2.4.0 버전에 대한 설명을 더 보강해 나가기로 하겠다.

11. IEEE 1394 for Linux

IEEE 1394는 USB와 비슷한 bus구조를 지니고, 디바이스들이 컴퓨터에 연결될 때 해당하는 드라이버를 loading해서 사용할 수 있도록 한다. IEEE 1394에 대한 지원은 커널 2.2.X버전에 대한 patch에서부터 보이고 있으며, 현재 IEEE1394를 사용해서 할 수 있는 것으로는 예를 들어서 디지털 비디오 캠코더를 연결해서 스트리밍을 받아서 플레이 할 수 있는 것을 들 수 있을 것이다. Window에 대한 경우에는 WDM(Windows Driver Model)을 사용해서 드라이버 스택(stack)이란 개념으로 각각의 해당하는 드라이버를 원할 때 즉각적으로 띄울 수 있도록 하고 있다. 리눅스에서도 유사하게 모듈화 된 드라이버들을 사용해서 드라이버 스택을 구현하고 있으며, [그림72]와 같은 구조를 지닌다.

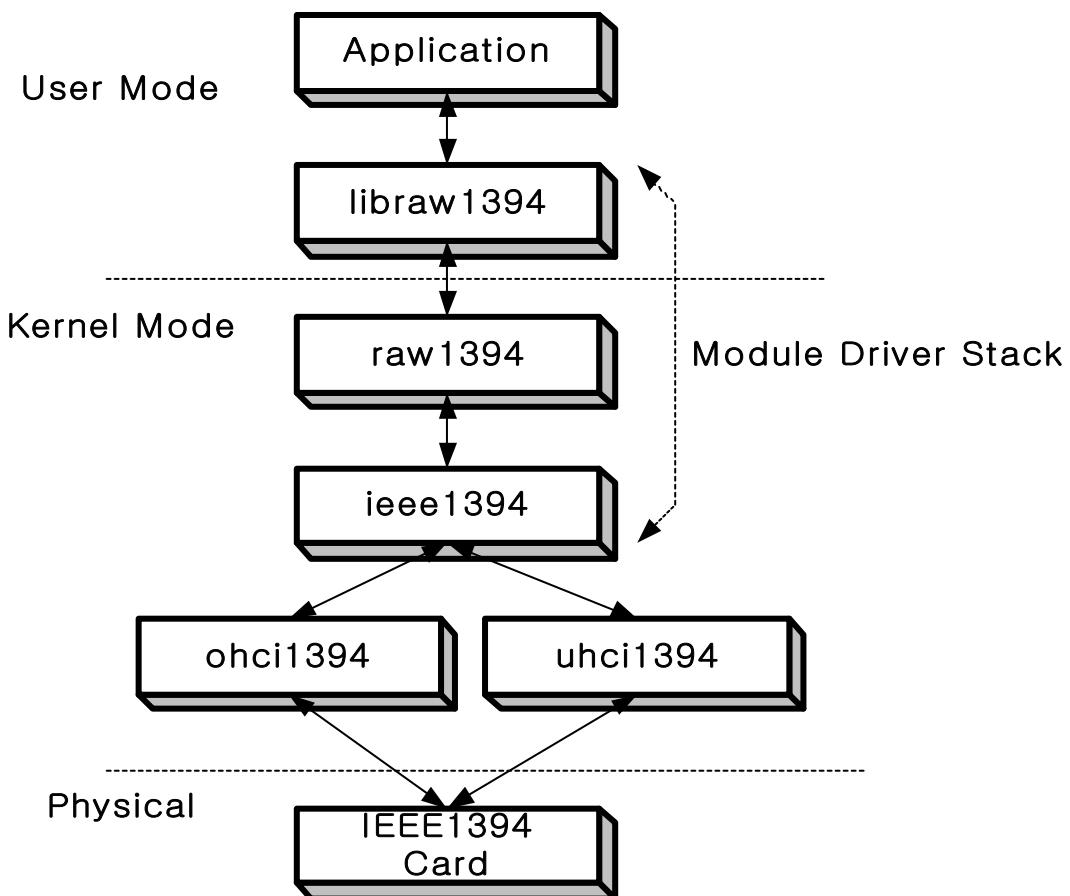


그림 87. IEEE1394 Device driver의 구성

먼저 드라이버의 계층 구조를 보도록 하자²⁶⁴. [그림72]는 PC상에서 리눅스로 IEEE1394의 구현을 보여주는 것으로 IEEE1394의 핵심은 ieee1394 모듈이다. 이 모듈이 전체 IEEE1394의 상하 레벨의 드라이버들을 관리하며, transaction의 처리를 다루며, 사건(event)의 triggering에 대한 메커니즘을 제공한다. ieee1394모듈의 아래에는 하위 레벨의 하드웨어 드라이버 모듈이 있으며, 이에 대해서 3개의 드라이버 모듈이 존재한다. 각각은 아래와 같다.

1. aic5800 – Adaptec AIC-5800 PCI-IEEE1394 chip 드라이버
2. pcilynix – Texas Instruments PCILynx 드라이버
3. ohci1394 – 1394 Open Host Controller 인터페이스 드라이버

²⁶⁴ 이하의 내용은 Soren Thing Andersen<soeren@thing.dk>가 쓴 것을 정리한 것이다.

이하에서는 OHCI호환 카드에 대한 것을 설명하기로 한다. 3개의 드라이버를 동시에 적재해서 활성화시키는 것도 가능하다. 하위 드라이버의 기본적인 최대 수는 4개이며, 모든 하위 레벨 드라이버는 적어도 하나 이상의 카드를 제어한다.

ieee1394 모듈의 위에는 상위 레벨의 드라이버 모듈이 오며, 이러한 것의 예로서 raw1394가 있다. 이 모듈은 사용자 주소 공간의 응용 프로그램들이 raw 1394 버스에 접근하는 인터페이스를 제공한다. 또 다른 상위 레벨의 드라이버로는 SBP2 드라이버가 있는데, 이는 Mark Halte가 계발 중에 있다. 사용자 공간에서 raw 1394 버스에 접근하기 위해서는 응용 프로그램은 반드시 libraw1394를 링크해야 하며, libraw1394가 raw 1394 상위 레벨 드라이버와의 통신을 처리하게 된다.

~/drivers/ieee1394 디렉토리에 있는 모든 파일은 아래와 같이 정리해서 볼 수 있다.

파일	설명
raw1394.*	상위 레벨 드라이버로서 raw1394에 관련된 파일들을 가진다.
csr.*	상위 레벨 드라이버로 local CSR들에 대해서 read/write/lock을 처리한다.
highlevel.*	상위 레벨 드라이버의 등록과 관리를 맡는다.
ieee1394_core.*	IEEE1394 드라이버 구현의 핵심(core)이다. 모든 패킷은 이곳을 통과하며, timeout을 다룬다(handle).
ieee1394_transactions.*	일반적인 블록킹 read/write/lock 함수들과 utility 함수를 가진다.
events.*	Event handler 등록과 event의 처리(dispatch)를 맡고 있다.
hosts.*	하위 레벨 드라이버의 등록과 관리를 맡고 있다.
ohci1394.*	1394 Open Host Controller 인터페이스 드라이버이다.
aic5800.*	Adaptec AIC-5800 PCI-IEEE1394 chip의 드라이버이다.
pcilynix.*	Texas Instruments PCILynx 드라이버이다.
ieee1394.h	일반적인 IEEE1394에 대한 정의를 가지고 있다. 예를 들어 transaction코드를 담고 있다.
ieee1394_types.h	여러 리눅스 헤더파일에 대한 include문과 사용하게 될 타입의 정의를 담고 있다.
ieee1394_syms.c	모듈의 사용을 위한 심벌들에 대한 export를 담고 있다.
Makefile, Config.in	커널의 컴파일을 위한 내용을 담고 있다.

표 63. IEEE 1394구현에 관련된 파일들

코드를 보게 될 때 나오는 “hpsb_”와 같은 형태의 이름들은 High Performance Serial Bus의 약자로 생각하면 될 것이다. 그리고, event.*로 시작하는 event의 처리와 관련된 기본 골격(framework)은 현재는 사용되지 않는다.

Linux에서의 IEEE 1394 디바이스 드라이버에 대한 것은 인터넷을 통해서 거의 자료를 구할 수 없는 상황이라, 이곳에서는 간단히 IEEE 1394를 이용해서 어떤 드라이버를 만들 수 있는 것을 논하기로 하겠다. 여기서 예로 사용하는 드라이버는 네트워킹을 구현할 수 있는 것으로 이와 같은 목적에 충분히 부합한다고 생각된다. IP over 1394의 구현을 보기 위해서 잠시 상위 레벨 드라이버에 대해서 먼저 알아보도록 하자.

11.1. 상위 레벨 드라이버란?

리눅스 IEEE 1394 subsystem의 상위 레벨 드라이버들은 hpsb_register_highlevel() 함수를 불러서 ieee1394 모듈에 자신을 등록하는 루틴(routine)들이다. hpsb_register_highlevel() 함수는 다음과 같은 네 개의 함수에 대한 포인터를 가지는 구조체를 포함하며, 이 함수들은 등록하는 상위 레벨 드라이버에서 제공한다. 즉, add_host(), remove_host(), host_reset(), iso_receive() 함수들이 상위 레벨 드라이버에서 제공해야 하는 함수이다. 이러한 함수들은 하위 시스템에서 적절한 사건(event)가 발생할 때 호출 될 것이며, 가령 예를

들어 add_host() 함수의 경우는 새로운 호스트²⁶⁵를 발견할 때마다 호출 될 것이다. iso_receive() 함수는 하위 레벨의 드라이버로부터 isochronous 데이터를 받을 때마다 호출된다. hpsb_register_highlevel()이 호출되면, 즉각적으로 ieee1394 모듈에 의해서 이미 알고 있는 모든 호스트들에 대해서 add_host() 함수가 호출될 것이다.

상위 레벨 드라이버는 또한 local 호스트들의 1394 주소 공간 영역(area)에 대한 read/write/lock transaction들을 다루도록 자신을 등록할 수 있다. 이렇게 하기 위해서는 상위 레벨 드라이버 자신이 hpsb_register_addrspace() 함수를 호출 해야 할 것이며, 넘겨주는 구조체로 주소 영역(address range)과 네 개의 함수를 상위 레벨 드라이버에서 제공해야 한다. 제공해야 하는 함수로는 read(), write(), lock(), 과 lock64() 함수가 있다. 이러한 함수들은 하위 시스템에서 적절한 사건이 있을 때 호출될 것이며, 예로서 read() 함수는 명시된 주소 영역에 속하는 주소의 read 요청을 시스템이 받았을 때 호출된다. 만약 시스템이 상위 레벨 드라이버 자신이 등록하지 않은 주소에 대한 요청을 받으면 response 코드로 “resp_address_error”를 돌려줄 것이다.

위에서 설명한 두개의 등록 함수를 부를 때, 포인터로 넘겨진 것들에 대해서 NULL로 두는 것이 허락되며, 만약 hpsb_register_addrspace() 함수의 인자 중 단지 read()만이 NULL이 아니라면, 물음에 대한 주소 영역은 1394 버스에서 read-only이 될 것이다. 같은 방법으로 hpsb_register_highlevel() 함수의 인자인 iso_receive()가 NULL이라면, isochronous 데이터를 받았을 경우에 상위 레벨 드라이버가 호출되는 것을 막을 수 있다.

IEEE1394 표준은 64bit으로 이루어진 주소 공간을 사용한다. 이중 상위 16bits는 노드(node)의 ID를 나타내기에, 상위 레벨 드라이버는 하위의 48bits의 주소 공간에 대해서만 연산을 등록할 수 있다. 이것은 상위 레벨 드라이버가 모든 local 호스트 상의 명시된 주소들에 대해서 호출 될 것이라는 것을 의미한다. 또한 상위 레벨 드라이버는 필요하다면 호스트 들을 다루는(handling) 방법을 달리 할 수 있는데, 문제가 되고 있는 호스트의 레퍼런스가 호출과 같이 전달 되기 때문이다.

이상에서 상위 레벨 드라이버가 제공하는 것과 등록에 대한 것을 간략히 알아보았다. 즉, 하위에 관심을 두지 않는다면, 상위 레벨 드라이버만을 구현하는데 어떤 것들이 필요한지를 본 것이다. 이젠 이것을 어떻게 사용하는지를 보도록 하자.

11.2. 상위 레벨 드라이버의 초기화 설정

이 부분에서는 IEEE 1394를 리눅스 상에서 어떻게 사용하는지에 대해서 보기로 하자. 즉, 모듈로서 적재하는 insmod 명령을 수행했을 때 어떤 일들이 일어나는지 살펴보자.

```
#insmod ieee1394.o
```

위와 같은 명령을 수행하면, 모듈의 초기화 함수인 ieee1394_core.c에 있는 init_module() 함수가 호출된다. 다시 이 함수는 init_hpsb_highlevel()과 init_csr()²⁶⁶, init_ieee1394_guid() 함수를 호출한다. 이중 init_hpsb_highlevel() 함수는 highlevel.c에 정의되어 있으며, 상위 레벨 드라이버의 주소 공간과 제공하는 함수들에 대한 리스트를 초기화 한다. 이곳에서 찾을 수 있는 addr_space 변수는 상위 레벨 드라이버가 등록하는 주소 영역에 대한 리스트이다. 이 주소는 단지 하나의 상위 레벨 드라이버만이 처리를 담당한다. 초기화에서는 함수들의 리스트와 주소 공간의 리스트 모두 비어있게 된다.

init_csr() 함수는 csr.c에 정의되어 있으며, 상위 레벨 드라이버의 초기화가 진행된 후에 호출된다. init_csr() 함수는 “standard registers”라고 불리 우는 상위 레벨 드라이버를 등록하게 되며, 이 상위 레벨 드라이버가 표준 명령 및 상태 레지스터들, configuration ROM, topology map, speed map 등에 대한 처리를 담당한다. 여기서 사용하는 등록 함수로는 hpsb_register_highlevel()과 hpsb_register_addrspace() 함수가 있다.

만약, 하위의 드라이버가 필요하다면, insmod 명령을 통해서 적재되며, init_module()에서 hpsb_register_lowlevel() 함수가 호출될 것이다. hpsb_register_lowlevel() 함수가 넘겨받는 인수에는 호스트를 찾거나 지우고, 패킷을 전송하는 등의 일을 하는 함수들과 호스트의 이름을 가지는 hpsb_host_template를

²⁶⁵ IEEE 1394 인터페이스 카드

²⁶⁶ Configuration and Status Register

넘겨받는다. `hpsb_register_lowlevel()` 함수는 `hosts.c`에 구현되어 있으며, 이 함수는 하위 레벨의 드라이버를 자신이 가지고 있는 드라이버의 리스트에 더하고, 하위 레벨 드라이버가 알고 있는 모든 시스템에 붙은 호스트(1394 adapter)를 찾아서 초기화 및 등록시켜준다. 여기서부터는 하위의 드라이버 및 장치가 모두 기동 되어 가동 상태가 된다. 하지만, 사용자 모드의 프로세스가 `raw` 버스에 접근하기 위해서는, 다른 하나의 모듈이 더 필요하다. 이것이 바로 `raw1394` 모듈이다.

```
#insmod raw1394
```

`raw1394` 모듈은 자신을 상위의 드라이버로 `hpsb_register_highlevel()` 함수를 이용해 모듈로 적재될 때 등록한다. 하지만, 1394 주소 공간의 어떤 부분도 자신이 다루지 않기에 `hpsb_register_addrspace()` 함수를 호출하지는 않는다. 또한 `raw1394` 모듈은 자신을 문자 모드 디바이스로 `register_chrdev()` 함수를 이용해서 등록한다. 즉, `/dev/raw1394`를 통해서 접근할 수 있는 창구를 열어주고 있다. 이 문자 모드 디바이스 노드(`/dev/raw1394`)는 다시 `libraw1394`에 의해서, 사용자 모드에서 `raw1394`를 접근하는데 사용된다. 따라서, 여기서부터는 `libraw1394`를 써서 응용 프로그램이 리눅스에서 지원하는 IEEE1394 하위 시스템(subsystem)을 사용할 준비가 된 것이다.

자, 이제 어느 정도 상위 레벨 드라이버에 대한 이야기를 했으므로, 상위 레벨 드라이버의 한 예가 될 수 있는 IP over 1394 드라이버의 구현을 상세히 논하도록 하겠다. 부족한 부분이 많지만, 어쨌든 참고자료는 충분하리라 본다.

11.3. IP over 1394의 구현

IP over 1394란 Firewire라고 불리 우는 IEEE 1394 카드를 이용해서 간단히 네트워킹을 구현하고자 할 때 사용된다. 이러한 예로서는 가정에 있는 컴퓨터 들을 기존의 10 Base T와 같은 LAN cable이 아니라 1394 cable로 연결해서 네트워크를 만드는 것을 들 수 있다. 홈 네트워크(Home Network)를 구성하는 기본이라고 할 수 있겠다. 이곳에서 볼 내용은 쉽게 말하면, 기존에 있는 TCP/IP protocol stack의 하위에 IEEE 1394연결을 사용할 수 있도록 가상의 디바이스 드라이버를 하나 집어 넣는 것이다. 따라서, 기존의 IP 연결을 사용하는 모든 응용 프로그램들이 IEEE 1394를 사용해서 다른 컴퓨터와 통신할 수 있게 된다.

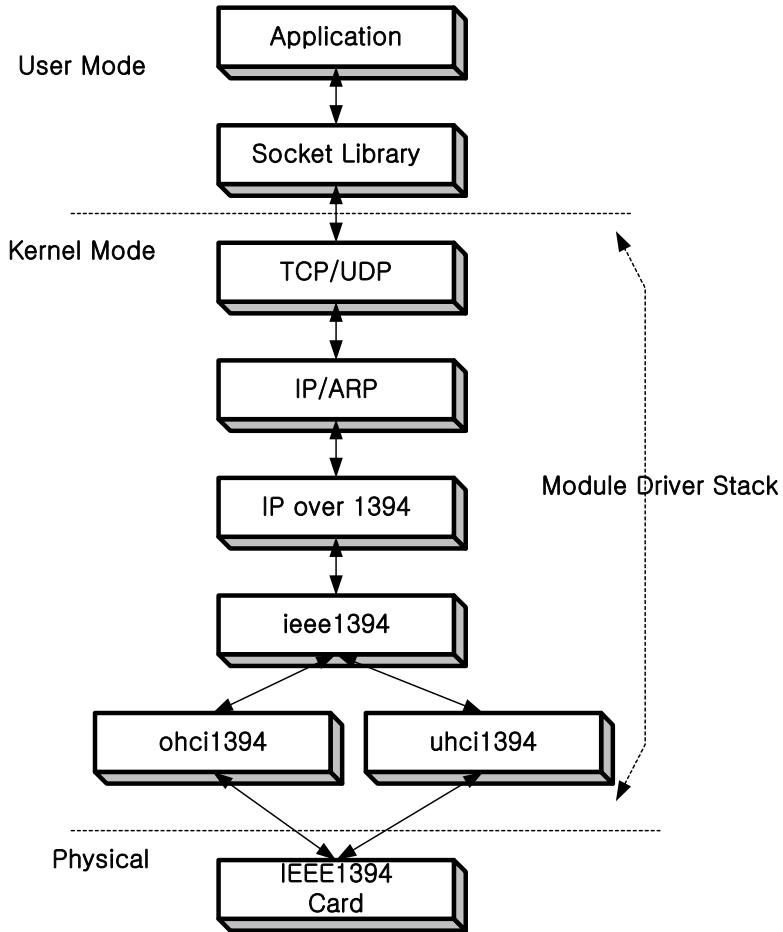


그림 88. IP over 1394의 Architecture

[그림 88]은 IP over 1394를 구현하는 방법에 대한 전체 architecture를 그린 것이다. 여기서 우리가 구현하고자 하는 모듈은 바로 IP over 1394이다. 즉, IEEE 1394 상위 driver로 동작하면서, TCP/IP stack에는 마치 ethernet driver와 같은 역할을 해주는 것이다.

프로그램의 source 코드는 http://www.s.netic.de/gfiala/IP_over_1394.html 에 있는 ip1394.tgz 파일이다. 어떤 식으로 가상의 드라이버를 구현하는지 만을 보기 바란다. 이와 같은 개념은 다른 시스템에서도 자주 사용하게 되는 것이므로, 코드에 너무 치중하지 말고 상위 및 하위의 인터페이스만 만족시킨다면, 어떠한 형태의 드라이버도 구현할 수 있다는 점에 유념하기 바란다. 자세한 것은 그 홈페이지에 있는 내용을 참조하도록 하자.

11.3.1. Driver의 초기화와 삭제

먼저 module로 컴파일된 드라이버를 사용하기 위해서는 README 파일을 읽어보면 알 수 있겠지만, ieee1394 모듈과 자신이 가지고 있는 IEEE 1394 카드의 인터페이스 스펙(spec.)이 원지에 따른 OHCI(혹은 PCILYNX) 모듈이 먼저 로드(load)가 되어야 한다. 그리고 나서, 해당 모듈을 다음과 같이 로드 한다.

```
% insmod ip1394.o
```

이렇게 했다고 끝나는 것이 아니라, 자신이 만들고자 하는 네트워크 정보에 따라서 네트워크 인터페이스를 설정해 주어야 한다. README 파일에는 다음과 같은 예를 보여주고 있다.

```
% ifconfig eth0 up 192.168.0.1 ( 이것은 eth0를 사용하는 네트워크 인터페이스를 가동(up)시키고, 이 인터페이스에 사용할 주소로 192.168.0.1을 준다는 의미이다. )
```

만약 다른 네트워크 인터페이스가 있다면, 조금 다른 setting을 해주어야 할 것이지만, 이것만으로도 기동하는 데는 문제가 없을 것이다. 이것으로 디바이스의 설정에 대한 이야기를 마치고 본론으로 들어가도록 하겠다.

```
#ifdef MODULE
int init_module(void)
{
    return init_ip1394();
}
void cleanup_module(void)
{
    cleanup_ip1394();
    return;
}
#endif
```

코드 656. IP over 1394 드라이버의 초기화와 해제

모듈을 등록할 때 제일 먼저 수행되는 것은 init_module() 함수이다. 해제는 cleanup_module() 함수가 처리한다. 각각의 함수는 init_ip1394() 함수와 cleanup_ip1394() 함수를 호출한다.

```
int init_ip1394(void)
{
    /* IEEE1394 highlevel driver registering */
    hl_handle = hpsb_register_highlevel(IP1394_DEVICE_NAME, &ip1394_hl_ops);
    if (hl_handle == NULL) {
        HPSB_ERR("No more memory for driver %s", IP1394_DEVICE_NAME);
        return -ENOMEM;
    }
    /* Listen incoming ethernet packets */
    hpsb_register_addrspace(hl_handle, &ip1394_addr_ops, MEM_IP_1394_START,
                           MEM_IP_1394_END);
    /* IPover1384 driver registering */
    ip1394_dev.init = ip1394_init;
    if (register_netdev(&ip1394_dev)) {
        printk("Error during network driver init %s", ip1394_dev_name);
        return -ENODEV;
    }
    printk("Successfully load %s driver\n", ip1394_dev_name);
    return 0;
}

void cleanup_ip1394(void)
{
    kfree(ip1394_dev.priv);
    unregister_netdev(&ip1394_dev);
    hpsb_unregister_highlevel(hl_handle);
}
```

코드 657. init_ip1394() 함수와 cleanup_ip1394() 함수의 정의

init_ip1394() 함수는 IEEE 1394의 드라이버 스택(stack)의 상위 드라이버로 등록하는 일과 이 드라이버가 네트워크 인터페이스로 사용된다는 것을 커널에게 알려준다. 상위 드라이버로의 등록은 hpsb_register_highlevel() 함수가 처리하게 되며, 넘겨주는 파라미터 값으로는 드라이버의 이름(IP1394_DEVIE_NAME = ip1394)과 상위 레벨의 드라이버가 제공해야 되는 연산 벡터이다(ip1394_hl_ops). 함수의 호출 값으로 넘겨받는 것은 등록된 상위 레벨 드라이버에 대한

핸들로서 NULL 값이라면 등록되지 못했다는 것을 나타낸다. -ENOMEM은 등록하지 못한 이유가 더 이상 상위드라이버를 위해서 시스템의 메모리를 할당하지 못했다는 말이다.

이젠 들어오는 packet들을 감지하기 위해서 자신이 사용하는 주소공간²⁶⁷ 을 등록할 차례다. hpsb_register_addrspace() 함수를 호출해서 이것을 처리한다. 앞에서 할당 받은 상위 레벨 드라이버의 핸들과 주소 공간에 대한 연산 벡터, 주소 공간의 시작(MEM_IP_1394_START = 0xfffff0200000ULL)과 끝(MEM_IP_1394_END = 0xfffff0200800ULL)이 파라미터로 넘겨진다. 해당 영역에 대한 연산(operation)이 있을 경우에 등록된 연산 벡터의 함수가 호출될 것이다. 이젠 네트워크 디바이스의 초기화를 담당할 함수를 설정하고(ip1394_init()), register_netdev() 함수를 호출해서 Linux의 네트워크 디바이스로 등록한다. 이때 초기화 함수(ip1394_init())도 호출될 것이다. 예러가 있다면 -ENODEV를 돌려준다. 복귀 값은 0이다. cleanup_ip1394() 함수는 초기화에서 할당하는 커널 메모리 공간(ip1394_dev.priv)을 해제하고, 네트워크 드라이버 등록 해제 및 IEEE 1394 상위 드라이버 등록 해제를 수행하는 것이 주요 역할이다.

종합해서 말하자면, 현재 보고 있는 IP over 1394 드라이버는 상위 커널에 대해서는 네트워크 디바이스 드라이버로 등록 시켜주며 하위의 IEEE 1394 드라이버에는 상위 레벨의 드라이버로서 등록되어 실제적인 디바이스를 제어하기 보다는 가상의 디바이스를 만들어주는 역할을 한다. Window와 같은 경우에는 이러한 역할을 해주는 것으로, 다른 드라이버의 상위에 존재하는 filter 드라이버와 같은 것이 될 수 있을 것이다.

더 진행하기에 앞서 간단히 상위 레벨 드라이버의 연산 벡터에 대한 정의와 주소공간 연산 벡터에 대한 정의를 좀더 보기로 하자. 코드는 Linux 커널의 drivers/ieee1394/highlevel.h에 있다.²⁶⁸

```
struct hpsb_highlevel_ops {
    /* 아래에서 보여주는 필드는 iso_receive를 제외하고는 NULL이 될 수 있다. iso_receive 필드는
       채널에 대한 요청이 없는 경우에만 NULL이 될 수 있을 것이다.*/
    /* 새로운 IEEE 1394 host가 초기화 될 때 호출된다. 또한 hpsb_register_highlevel() 함수가 모든
       등록된 host들에 대해서 호출될 때도 역시 실행될 것이다.*/
    void (*add_host)(struct hpsb_host *host);
    /* Host가 제거되려고 할 때 호출될 것이다. 또한 hpsb_unregister_highlevel()이 각각의 host들에
       대해서 한번씩 호출될 때도 사용될 것이다.*/
    void (*remove_host)(struct hpsb_host *host);
    /* Host가 설정 변경(configuration change)과 같은 있을 수 있는 bus reset과 같은 상황에서
       호출될 것이사. 주의할 것은 이러한 상황이 interrupt나 bottom half handling과 같은 곳에서도
       일어날 수 있으며, hpsb_reads() 와 같은 것을 할 수 있으리라고는 예측해선 안된다.*/
    void (*host_reset)(struct hpsb_host *host);
    /* Isochronous 패킷을 받을 경우에 호출된다. 채널은 편의를 위해 채널 번호를 가질 수 있으며,
       채널 번호는 받은 패킷의 해더에 있는 첫 4 bytes에도 CRC를 제외하고 들어가 있다.
       또한 요청하지 않은 channel과 host의 조합에서도 불려질 수 있다.
    void (*iso_receive)(struct hpsb_host *host, int channel, quadlet_t *data, unsigned int length);
    /* FCP_COMMAND(direction 파라미터가 0)나 FPC_RESPONSE(direction 파라미터가 1)
       레지스터인 상황에서 write 요청을 받을 수 있다. cts 파라미터는 데이터의
       첫번째 bytes를 가진다.269 */
    void (*fcp_request)(struct hpsb_host *host, int nodeid, int direction,
                        int cts, u8 *data, unsigned int length);
};

struct hpsb_address_ops {
```

²⁶⁷ IEEE 1394 디바이스들은 기기들간에 연결되면 자신이 사용하는 주소 공간영역이 결정되며, 특정 디바이스에 데이터를 쓴다는 것은 실제적으로는 그 주소 공간영역에 데이터를 쓴다는 말과 동일하다. 즉, 시스템의 bus와 같은 역할을 하기 때문이다.

²⁶⁸ 주석은 코드에 있는 것을 그대로 한글로 옮겼다. IEEE 1394 프로젝트를 진행하는 당사자들이 제시한 것인만큼 신용이 가기 때문이다.

²⁶⁹ 관련된 자료로서는 IEEE 1394의 CSR(Configuration and Status Register) Architecture로서 IEC 61883-1/FDIS spec.을 참조하라.

```

/* NULL로 선언된 필드의 함수 포인터 각각은 RCODE_TYPE_ERROR이라는 에러코드를
돌려줄 것이다. 이것은 read-only 속성을 가진 레지스터들에 대한 구현을 쉽게 하기 위한
것이다. 선언된 함수 포인터는 적절한 IEEE 1394 RCODE를 돌려주어야(return) 한다.
사용될 수 있는 RCODE에는 다음과 같은 것이 있다.

#define RCODE_COMPLETE          0x0  - 에러가 없다.
#define RCODE_CONFLICT_ERROR    0x4  - 충돌 에러가 발생했다.
#define RCODE_DATA_ERROR        0x5  - 데이터에 에러가 있다.
#define RCODE_TYPE_ERROR         0x6  - Type 에러가 있다.
#define RCODE_ADDRESS_ERROR     0x7  - 주소 영역 에러가 있다.
*/
/* 다음의 함수들은 block read를 구현해야 한다.*/
int (*read)(struct hpsb_host *host, int nodeid, quadlet_t *buffer, u64 addr, unsigned int length);
int (*write)(struct hpsb_host *host, int nodeid, int destid, quadlet_t *data, u64 addr, unsigned int length);
/* Lock transactions이다. ext_tcode에 주어진 연산의 write 결과는 *store에 저장된다.*/
int (*lock)(struct hpsb_host *host, int nodeid, quadlet_t *store, u64 addr, quadlet_t data, quadlet_t arg,
           int ext_tcode);
int (*lock64)(struct hpsb_host *host, int nodeid, octlet_t *store, u64 addr, octlet_t data, octlet_t arg,
              int ext_tcode);
};

}

```

코드 658. hpsb_highlevel_ops 및 hpsb_address_ops 구조체의 정의

위와 같이 정의된 구조체를 IP over 1394에서는 다음과 같이 정의하고 있다. 즉, 해당하는 연산 부분에 자신이 정의한 함수를 삽입함으로써 하위 레벨에서 이벤트(event)가 발생했을 때 호출될 함수를 정의해서 넣어준 것이다.

```

/* Functions for incoming ip1394 packets */
struct hpsb_address_ops ip1394_addr_ops = {
    NULL,
    write_ip1394,
    NULL,
    NULL
};

/* OHCI1394 highlevel driver functions */
struct hpsb_highlevel_ops ip1394_hl_ops = {
    ip1394_add_host,
    ip1394_remove_host,
    ip1394_host_reset,
    NULL,
    NULL,
};

```

코드 659. ip1394_addr_ops와 ip1394_hl_ops의 정의

따라서, ip1394_hl_ops 구조체가 정의하고 있는 연산 벡터로부터 유추할 수 있는 것은 host를 새로 찾거나, 제거 혹은 bus의 reset상황에서 코드에서 정의하고 있는 함수가 호출되기를 바란다는 것이고, 상대방 host에서 현재의 host가 가진 주소 영역에 대해 write가 있을 경우에 호출될 함수를 정의하고 있다는 것을 확인할 수 있다. 각각의 함수에 대해서는 해당하는 부분에서 다시 자세히 보도록 하겠다.

11.3.2. Network 디바이스 드라이버의 초기화

앞에서 network디바이스 드라이버로 등록하기 바로 앞에서 설정한 ip1394_init() 함수가 네트워크 디바이스를 위한 초기화를 담당할 것이다.

```
int ip1394_init(struct net_device *dev)
```

```
{
    int i ;
    struct list_head *lh;
    struct host_info *hi;

    /* Fill in the fields of the device structure with ip1394 values. */

    dev->open = ip1394_open;
    dev->stop = ip1394_release;
    dev->set_config = ip1394_config;
    dev->hard_start_xmit = ip1394_tx;
    dev->tx_timeout = ip1394_tx_timeout;
    dev->watchdog_timeo = IP1394_TIMOUT;
    dev->hard_header = ip1394_header;

    dev->rebuild_header = ip1394_rebuild_header;
    dev->set_mac_address = NULL ; /* Don't support changing HW addr manual */
    dev->hard_header_cache = NULL ; /* ip1394_header_cache; this is for raw socket */
    dev->header_cache_update = NULL ; /* ip1394_header_cache_update; : this is for raw socket */
    dev->hard_header_parse = NULL ; /* ip1394_header_parse; : this is for raw socket */

    dev->do_ioctl = ip1394_ioctl;
    dev->get_stats = ip1394_stats;
    dev->change_mtu = NULL ; /* ip1394_change_mtu is not supported; */
    dev->hard_header_len = IP1394_UF_HLEN + IP1394_WRITEB_HLEN; /* this is used only for incoming packets
*/
    dev->addr_len = IP1394_HW_ADDR_LEN;
    dev->type = ARPHRD_1394;
    dev->tx_queue_len = 100; /* ip1394 wants good queues */

    /* New-style flags. */
    dev->flags = IFF_BROADCAST /* | IFF_MULTICAST */ ;
    dev_init_buffers(dev);

    /* Then, allocate the priv field. This encloses the statistics
     * and a few private fields.
    */
    dev->priv = kmalloc(sizeof (struct ip1394_priv), GFP_KERNEL);
    if (dev->priv == NULL)
        return -ENOMEM;
    memset(dev->priv, 0, sizeof (struct ip1394_priv));
}
```

코드 660. ip1394_init() 함수의 정의

ip1394_init() 함수는 Linux 커널에 네트워크 드라이버의 역할을 하도록 만들어주는 부분이다. 네트워크 드라이버로서의 역할을 하기 위해선 net_device 구조체의 각 필드를 적절한 값으로 채워주면 된다. 기본적으로 정의된 field들은 다음과 같다.

```
static struct net_device ip1394_dev = {
    "ip1394", /* name is in a statically allocated memory : 디바이스의 이름 */
    0, 0, 0, 0, /* shmem addr : 공유 메모리 영역 설정 */
    0x0000, /* IOport : I/O port 메모리 공간에 대한 설정 */
    0, /* irq : 사용할 interrupt 번호 */
    0, 0, 0, /* flags : 기본 flag들 */
    NULL, /* next : 다음 네트워크 인터페이스에 대한 포인터 */
    NULL, /* init = null, should be filled by init_ip1394() : 네트워크 초기화 함수 */
};
```

코드 661. ip1394_dev 구조체의 정의

ip1394_dev구조체를 보면, I/O port 및 공유 메모리, IRQ 부분이 없다는 것을 확인할 수 있을 것이다. 이것은 현재 보고 있는 드라이버가 software적인 가상의 드라이버처럼 동작하기 때문이다. 그리고, 위의 구조체에 덧붙여 다시 다음과 같은 것들이 초기화 부분에서 설정 된다.

- Open 함수 - 디바이스에 대한 open() 연산을 처리한다(ip1394_open()).
- Stop 함수 - 디바이스에 대한 stop() 연산을 처리한다(ip1394_release()).
- Set_config 함수 - 디바이스에 대한 설정을 변경한다(ip1394_config()).
- Hard_start_xmit 함수 - 보내는 연산을 처리한다(ip1394_tx).
- Tx_timeout 함수 - 보내는 연산의 timeout을 처리한다(ip1394_tx_timeout()).
- Watchdog_timeo - watchdog timer의 timeout 간격을 설정한다(IP1394_TIMEOUT : HZ/10).
- Hard_header 함수 - 하드웨어 packet의 헤더를 만든다(ip1394_header()).
- Rebuild_header 함수 - 패킷을 보내기 전에 하드웨어 헤더를 재구성한다(ip1394_rebuild_header()) - 우리가 헤더를 직접 구성해 주어야 한다.)
- Set_mac_address 함수 - MAC(Media Access Control) 주소를 설정한다(NULL - 하드웨어 주소를 인위적으로 바꾸는 것을 지원하지 않으므로 NULL로 둔다. 실제로는 IP over 1394의 경우 ethernet과 같은 주소체계를 사용하지 않고 있다.)
- Hard_header_cache 함수 - ARP를 사용할 경우 ARP query의 결과로 hh_cache구조체를 채우기 위해서 사용한다(NULL).
- Header_cache_update 함수 - 주소 변경과 같은 상황에 대한 hh_cache 구조체에 있는 목적지 주소를 update하는 방법(method)이다(NULL).
- Hard_header_parse 함수 - 소켓버퍼 구조체에 있는 패킷으로부터 source의 주소를 얻기 위한 방법이다(NULL).
- Do_ioctl 함수 - 디바이스에 대한 I/O Control 연산을 적용한다(ip1394_ioctl()).
- Get_stats 함수 - 디바이스에 대한 통계정보를 얻는다(ip1394_stats()).
- Change_mtu 함수 - 전송 path의 MTU(Maximum Transfer Unit) 값을 변경한다(NULL).
- Hard_header_len - 하드웨어 헤더의 길이를 나타낸다(IP1394_UF_HLEN + IP1394_WRITEB_HLEN = 4 + 16 = 20 Bytes).
- Addr_len - 하드웨어 주소의 길이를 나타낸다(IP1394_HW_ADDR_LEN = 2 Bytes).
- Type : 네트워크 인터페이스의 type을 나타낸다(ARPHRD_1394 = 9 : 이 부분은 Linux 커널의 include/linux/if_arp.h에 들어가 있어야 할 것이다.)²⁷⁰
- Tx_queue_len - Tx시에 디바이스의 전송 queue에 들어갈 수 있는 최대 frame의 갯수를 정한다. 일반적으로 ethernet을 사용할 경우에는 ether_setup()이라는 함수에서 100으로 설정할 것이다. 여기서도 100으로 설정하고 있다.
- Flags - 네트워크 디바이스의 상태나 capability를 나타낸다고 볼 수 있다.(IFF_BROADCAST | IFF_MULTICAST : broadcasting과 multicasting을 지원한다.)

위와같은 net_device 구조체에 대한 설정을 거의 마쳤다면, 이전 디바이스를 위한 버퍼 큐를 초기화 하기 위해서 dev_init_buffers() 함수를 호출하고, 디바이스를 위한 고유한 영역(private data area)로 사용할 공간을 커널 메모리 공간에서 할당받는다(dev->priv = kmalloc()). 만약 할당받을 수 없다면 -ENOMEM을 돌려준다. 할당받은 메모리 공간을 0으로 초기화 해주는 것도 잊어서는 안될 것이다(memset()).

```
/* We choose only the first card found : */
/* TO BE : all card must be choosed */
if (host_count > 0) {
    spin_lock_irq(&host_info_lock);
    lh = ip1394_host_indo_list.next;
    hi = list_entry(lh, struct host_info, list);
    /** Registering card **/
    ((struct ip1394_priv *) (dev->priv))->host = hi->host;
```

²⁷⁰ README 파일과 같은 곳에서 이렇게 하도록 요구하고 있다.

```
((struct ip1394_priv *) (dev->priv))->eui_64_hi = hi->host->csr.rom[3] ;
((struct ip1394_priv *) (dev->priv))->eui_64_lo = hi->host->csr.rom[4] ;
spin_unlock_irq(&host_info_lock);
} else {
    printk("Error : no card found !\n");
    return -ENODEV;
}
/* we use this card's max data payload size as default mtu */
dev->mtu = (1<<(((hi->host)->csr.rom[2]>>12)&0xf)+1)) - IP1394_UF_HLEN ;
for (i= 0 ; i < 64 ; i++) {
    max_rec[i] = 0x10 ; /* default max packet size = 2***(0x10+1) */
    sspd[i] = 0 ;
    fifo_offset[i] = MEM_IP_1394_START ; /* 0xfffff0200000ULL */
}
return 0;
}
```

코드 662. ip1394_init() 함수의 정의(계속)

host_count 변수는 host가 발견되면 1씩 증가되고, host가 제거될 경우에 1씩 감소한다. 처음에 모듈이 설치될 경우에 한번 증가가 되기에 host_count는 0보다 큰 값을 가질 것이다. 그렇지 않다면 -ENODEV를 돌려준다.²⁷¹

일단 host_info_lock에 인터럽트를 받지 않도록 lock을 설정하고(spin_lock_irq()), host card에 대한 정보를 수집하도록 한다. 시스템에 있는 host card들의 정보(host_info 구조체)는 ip1394_host_ino_list라는 연결 리스트의 next를 참조하면 될 것이다(lh). 여기서 하나의 entry를 추출해서 이 정보를 가지고 private 데이터 영역을 초기화 한다.

```
struct ip1394_priv {
    u32 eui_64_hi ; /* this is this card's 64bits extended unique id high 32bits */
    u32 eui_64_lo ; /* this is this card's 64bits extended unique id low 32bits */
    struct hpsb_host *host; /* The ieee1394 card used */
    struct net_device_stats stats;
};
```

코드 663. ip1394_priv 구조체의 정의

ip1394_priv 구조체는 host card에 대한 정보를 eui_64_hi, eui_64_lo에 CSR ROM의 4번째와 5번째 entry를 가지고 초기화하고, hosst_info 구조체의 host필드를 가지고 hpsb_host 구조체의 포인터를 초기화 한다. 여기서 대부분의 자료구조는 이미 add_host()와 같은 함수에 의해서 넣어진 정보이므로 그대로 설정만 하도록 한다. 나중에 get_stats()와 같은 함수에서 원하는 데이터는 net_device_stats 구조체를 가지는 stats가 사용될 것이다. 이것을 끝내고 나면 앞에서 설정한 lock을 해제한다(spin_unlock_irq()).

디바이스의 최대 전송 단위(MTU)은 CSR ROM의 3번째 엔트리(entry)를 차지한다.²⁷² 여기서 구한 값은 최대 데이터의 recod 크기를 2의 승수로 나타낸 것이다. MTU의 크기는 하드웨어 헤더의 길이는 제외해야 할 것이다. IEEE 1394의 동일한 serial bus에 한번에 물릴 수 있는 노드(node)의 갯수가 63개까지 이므로 인덱스 값으로 64를 주어서 max_rec[]와 sspd[], fifo_offset[]에 대한 default값을 결정한다. 복귀 값은 0이 될 것이다.

11.3.3. Highlevel Driver Operations

더 이상 이야기를 진행하기에 앞서 IEEE 1394 protocol stack과의 인터페이스를 보기 위해서, 여기서는 앞에서 정의한 hpsb_highlevel_ops 구조체의 각각의 필드에 정의된 함수들을 분석해 보도록 하자.

²⁷¹ 여기서 할당받은 커널 메모리 공간을 다시 해제해 주어야 할 거라고 생각한다.

²⁷² Configuration ROM spec.중에 들어있다. Bus Infomation Block을 참고하기 바란다.

11.3.3.1. Add Host

Add host 함수는 카드가 발견될 때마다 호출된다. 일반적으로는 모듈의 로딩/loading)시에 호출되어 현재 어떤 host card들이 있는지를 찾을 것이다.

```
static void ip1394_add_host(struct hpsb_host *host)
{
    struct host_info *hi;
    hi = (struct host_info *) kmalloc(sizeof (struct host_info), SLAB_KERNEL);
    if (hi != NULL) {
        INIT_LIST_HEAD(&hi->list);
        hi->host = host;

        spin_lock_irq(&host_info_lock);
        list_add_tail(&hi->list, &ip1394_host_ino_list);
        host_count++;
        spin_unlock_irq(&host_info_lock);
    }
}
```

코드 664. ip1394_add_host() 함수의 정의

Host card에 대한 정보를 기록하기 위한 구조체를 할당 받는다(host_info). 할당 받은 구조체가 NULL이 아니라면, if() 이하를 실행한다. 할당 받은 자료구조의 list 연결을 초기화 하고(INIT_LIST_HEAD), host의 정보를 넘겨받은 포인터(host)로 초기화 시킨다. 이젠 이렇게 만든 자료구조를 전체 host의 정보를 가지는 ip1394_host_ino_list에 연결시켜준 후, 발견한 host card의 개수를 증가 시켜준다(host_count++). 이때 전역 변수들에 대한 접근이 있기에 spinlock을 설정해서 사용한다.

11.3.3.2. Remove Host

이 함수는 host card가 제거될 때 호출된다. 그러나 실제로 이러한 일은 일어나지 않기에 신경 쓰지 않아도 될 것이다.

```
static void ip1394_remove_host(struct hpsb_host *host)
{
    struct host_info *hi;
    printk("Calling ip1394_remove_host function\n");
    spin_lock_irq(&host_info_lock);
    hi = find_host_info(host);
    if (hi == NULL) {
        spin_unlock_irq(&host_info_lock);
        printk(KERN_ERR "Can't remove unknown host : 0x%p\n", host);
        return;
    }
    list_del(&hi->list);
    host_count--;
    spin_unlock_irq(&host_info_lock);
    kfree(hi);
}
```

코드 665. ip1394_remove_host() 함수의 정의

단지 host의 정보를 제거하는 일을 이곳에서 수행한다. 제거하려는 host의 찾고(find_host_info()), 찾은 host의 정보를 앞에서 구성한 리스트에서 제거한다. Host의 count는 1 감소할 것이다. 찾은 host 자료구조는 반드시 메모리에서 할당을 해제해야 한다(kfree()). 앞에서와 마찬가지로, 전역 자료구조에 대한 접근을 하기 전에 lock을 설정하고, 다시 해제해 주는 일을 한다.

```
static struct host_info *find_host_info(struct hpsb_host *host)
```

```
{
    struct list_head *lh;
    struct host_info *hi;
    lh = ip1394_host_indo_list.next;
    while (lh != &ip1394_host_indo_list) {
        hi = list_entry(lh, struct host_info, list);
        if (hi->host == host) {
            return hi;
        }
        lh = lh->next;
    }
    return NULL;
}
```

코드 666. find_host_info() 함수의 정의

find_host_info() 함수는 전체 host의 정보를 가지는 ip1394_host_indo_list를 검사해서 해당하는 host 정보가 있는지를 찾은 후 그 entry에 대한 포인터를 돌려준다. 전역 자료구조에 대한 변동이 없기에 특별히 lock을 설정할 이유는 없다.

11.3.3.3. Reset Host

Bus의 설정이 바뀌는 경우나 새로운 node가 bus에 삽입이 되는 등의 상황이 발생하면, reset 함수가 호출될 것이다.

```
static void ip1394_host_reset(struct hpsb_host *host)
{
    u16 node_id = host->node_id ;
    u16 bcast_node = 0xffff;
    int phy_id = node_id & NODE_MASK ;

    //ip1394_dev.tbusy = 1; /* can't transmit now */
    netif_stop_queue(&ip1394_dev);
    printk("%s function is called \n", __FUNCTION__);

    max_rec[phy_id] = (host->csr.rom[2] >> 12) & 0xf ;
    sspd[phy_id] = host->speed_map[(phy_id << 6)+phy_id] ;
    /* fifo_offset[phy_id] = user_defined_value */
    node_id = htons(node_id) ;
    memcpy(ip1394_dev.dev_addr, &node_id, ip1394_dev.addr_len ) ;
    memcpy(ip1394_dev.broadcast, &bcast_node, ip1394_dev.addr_len ) ;

    //ip1394_dev.tbusy = 0; /* can transmit now */
    netif_wake_queue(&ip1394_dev);
}
```

코드 667. ip1394_host_reset() 함수의 정의

노드의 ID를 얻기 위해서 넘겨받은 host의 정보에서 node_id 필드를 참조한다. Broadcast node(bcast_node)는 0xFFFF로 설정하고, 물리적인 ID(phy_id)는 node_id를 다시 NODE_MASK로 AND한 값을 사용한다. 이때 더 이상 전송이 일어나지 않도록 netif_stop_queue() 함수를 호출해서 디바이스에 관련된 queue를 잠시 멈추도록 만든다. 이전 해당 phy_id의 max_rec[] (최대 record 크기), sspd[] (speed) 배열에 들어갈 정보를 갱신시켜서 해당하는 노드의 물리적인 ID와 관련된 물리적인 정보의 변동이 있었음을 알려준다. 다시 네트워크 디바이스의 주소를 변경하기 위해서 node_id를 network byte ordering으로 변경한다(htons()) . 이렇게 구한 값으로 디바이스의 물리적인 주소를 다시 지정하고, 디바이스의 broadcast 주소도 변경한다. 이것을 마치고나면 다시 네트워크 인터페이스를 가동시키기

위해서 `netif_wake_queue()` 함수를 호출한다. 코멘트(comment)로 처리된 `tbusy`의 설정은 커널 버전 2.0등에 대해서 사용한다.

11.3.4. Highlevel Driver Address Space Operations

Highlevel Driver 연산 벡터와 관련된 함수의 정의는 단 하나 밖에 없다. 즉, 상대방이 자신의 host가 사용하는 주소 공간 영역에 `write`를 할 경우(패킷을 전달받는 경우)이다.

```
static int write_ip1394(struct hpsb_host *host, int nodeid, quadlet_t * data, u64 addr, unsigned int length)
{
    data -= 4 ; /* to include 1394 write block request header .
                  This doesn't need to modify 1394 subsystem */
    ip1394_rx(&ip1394_dev, length+16, (char *) data);

    data[0] |= ((NODE_MASK)<<16) ;
    return RCODE_COMPLETE;
}
```

코드 668. `write_ip1394()` 함수의 정의

`write_ip1394()` 함수는 단지 `ip1394_rx()` 함수를 호출해주는 역할만 한다. 이때, 데이터의 길이를 나타내는 `length`에 16을 더하고, 데이터의 포인터를 나타내는 `data`에는 -4(4 bytes를 4만큼 감소 시켜준다. 즉, 16 bytes 만큼 감소가 된다. 따라서, 받는 데이터의 길이도 16을 더해준 것이다.)를 해준다. 그리고, 받은 데이터의 destination ID를 나타내는 부분에는 `NODE_MASK(=0x003F)` 값을 좌측으로 16 bit shift해서 bitwise OR시켜준 값을 넣어 어느 `node`에서 전달했는지를 넣어둔다.²⁷³ 복귀 값은 앞에서 말했듯이 `RCODE_COMPLETE`가 된다.

11.3.5. Network Interface Operations

여기서부터는 이전 하위 인터페이스에 대해서 보다는 상위의 protocol stack에 대한 인터페이스 역할을 수행하는 함수들에 대해서 보도록 하겠다. 앞에서 이미 디바이스 드라이버가 초기화 될 때 관련된 함수들이 다 초기화가 되었으므로, 이젠 각각의 함수만을 보면 대부분의 분석은 마무리가 될 것이다.

11.3.5.1. Open

`Open`은 `ifconfig`과 같은 명령어를 사용해서 네트워크 인터페이스를 기동 시키게 되면 호출될 것이다. 여기서 여러 가지 `net_device`의 필드들에 대한 부분이 `comment`로 처리되어 있는데, 이는 디바이스 드라이버를 나타내는 `net_device`의 큐를 기동 시키는 것으로 처리가 가능하기 때문이다.

```
int ip1394_open(struct net_device *dev)
{
    struct ip1394_priv *priv = (struct ip1394_priv *) dev->priv ;
    struct hpsb_host *host = (struct hpsb_host *) priv->host ;

    printk("Opening interface %s\n", ip1394_dev_name);

    ip1394_host_reset(host) ;
//    dev->interrupt = 0;
//    dev->start = 1;
//    dev->tbusy = 0;
    netif_start_queue(dev);
    MOD_INC_USE_COUNT;
    return 0;
}
```

코드 669. `ip1394_open()` 함수의 정의

²⁷³ Bus의 ID가 10 bit이며, 나머지 node의 ID가 6 bit을 차지한다.

넘겨받은 net_device 구조체로 부터 priv필드를 접근해서 앞에서 저장해 두었던(ip1394_init()), host의 정보를 가져온다. 이 정보를 이용해서 ip1394_host_reset()을 호출해서 해당하는 node에 대해 reset을 걸어준다. 이젠 네트워크 인터페이스를 제대로 서비스할 모든 준비가 다 되었으므로, netif_start_queue()를 실행해 준다. 모듈의 사용 카운터도 증가시켜주도록 한다(MOD_INC_USE_COUNT). 복귀 값은 0이다.

11.3.5.2. Release

Release는 네트워크 인터페이스의 사용을 더 이상 하지 않을 경우에 호출된다. 예를 들어 ifconfig과 같은 명령어를 사용해서 down시키는 경우가 될 것이다.

```
int ip1394_release(struct net_device *dev)
{
    printk("Closing interface %s\n", ip1394_dev_name);
    // dev->interrupt = 0;
    // dev->start = 0;
    // dev->tbusy = 1; /* can't transmit any more */
    netif_stop_queue(dev);
    MOD_DEC_USE_COUNT;
    return 0;
}
```

코드 670. ip1394_release() 함수의 정의

Open의 반대과정을 밟는다. netif_stop_queue()로 더 이 네트워크 인터페이스를 사용하지 않는다는 것을 알리고, 모듈의 사용 카운트를 낮춘다(MOD_DEC_USE_COUNT). 복귀 값은 0이다.

11.3.5.3. Hard Header

Hard header를 사용하는 사용자는 커널이다. 즉, 커널은 패킷을 보내기 전에 목적지와 소스(source) 주소를 가지고, 하드웨어 헤더를 구성해준다. 기본적으로 ethernet과 같이 일반화된 네트워크 인터페이스 프로토콜의 경우에는 신경 쓸 필요가 없지만, 만약 이와 다른 네트워크 인터페이스를 구현하고자 한다면, 이것은 직접 디바이스 드라이버에서 해결해 주어야 한다. 따라서, 넘겨받은 정보를 이용해서 특정 디바이스에 맞는 헤더를 새로 구성해 주어야 할 것이다. Ethernet인 경우에는 이것을 처리하는 함수가 eth_header()와 같은 것이 있으며, 이 함수는 ether_setup()을 호출해서 네트워크 디바이스를 등록하는 동안에 처리할 수 있다.

```
int ip1394_header(struct sk_buff *skb, struct net_device *dev, unsigned short type, void *daddr, void *saddr, unsigned len)
{
    unsigned char * daddr_;
    struct ip1394_hdr *hdr = (struct ip1394_hdr *) skb_push(skb, IP1394_UF_HLEN); /* IP1394_UF_HLEN : 4 */
    memset(hdr, 0, 4);
    hdr->u.uf_hdr.type = htons(type); /* We use only a unfragmented datagram */
    hdr->u.uf_hdr.lf = 0;

    /* Set the source hardware address. */
    daddr_ = (unsigned char *)skb_push(skb, dev->addr_len); /* IP1394_HW_ADDR_LEN : 8 */
    /* Anyway, the loopback-device should never use this function... */
    if (dev->flags & (IFF_LOOPBACK | IFF_NOARP)) {
        memset(daddr_, 0, dev->addr_len);
        return (dev->addr_len + IP1394_UF_HLEN);
    }
    if (daddr) {
        memcpy(daddr_, daddr, dev->addr_len);
        return (dev->addr_len + IP1394_UF_HLEN);
    }
    return -dev->hard_header_len;
```

{}

코드 671. ip1394_header() 함수의 정의

ip1394_header() 함수는 보내는 데이터가 들어있는 sk_buff 구조체와 네트워크 디바이스를 나타내는 구조체 및 패킷의 탑입, 목적지 주소, 소스 주소, 헤더의 길이 정보를 넘겨받는다. 넘겨받은 소켓 버퍼에서 IP over 1394가 사용할 헤더 만큼의 공간을 확보하고(skb_push), 이것을 ip1394_hdr구조체를 가르키는 hdr로 둔다. 헤더의 길이는 IP1394_UF_HLEN(= 4)이다. 헤더의 type 필드는 네트워크 byte order로 바뀐 type 값을 준다. 이젠 하드웨어 주소를 주기 위해서 다시 소켓버퍼에 헤더의 공간을 만든다(skb_push). 하드웨어 주소를 가르키는 것은 daddr_이다. 만약 네트워크 인터페이스가 현재 Loopback 디바이스 이거나 ARP를 사용하지 않는다면, 0으로 이 값을 설정한 후, 하드웨어 주소의 길이에 앞에서 IP over 1394의 헤더 길이를 더한 값을 복귀 값으로 준다. 그렇지 않다면, 다시 목적지 주소가 있는 경우 이를 하드웨어 주소의 길이 만큼을 daddr_에 복사하고, 다시 앞에서와 마찬가지 값을 돌려주면서 복귀 한다. 최종적인 복귀 값은 하드웨어 헤더의 길이를 음수로 바꾼 값이 될 것이다.²⁷⁴

11.3.5.4. Rebuild Header

Rebuild header는 패킷이 전송되기 바로 전에 하드웨어 헤더를 다시 만들어주는 역할을 한다. 기본적으로 ethernet인 경우에는 아무것도 설정되지 않았다면, 아직 찾지 못한 정보를 구하기 위해서 ARP를 이용할 것이다. 즉, eth_rebuild_header()와 같은 함수를 사용할 것이다.

```
int ip1394_rebuild_header(struct sk_buff *skb)
{
    unsigned char *daddr = skb->data ;
    struct ip1394_hdr *hdr = (struct ip1394_hdr *) (daddr + 8) ;
    struct net_device *dev = skb->dev ;

    switch (hdr->u.uf_hdr.type) {
#ifdef CONFIG_INET
        case __constant_htons(ETH_P_IP) :
            return arp_find(daddr, skb) ;
#endif
        default :
            memcpy(daddr, dev->dev_addr, dev->addr_len) ;
            break ;
    }
    return 0 ;
}
```

코드 672. ip1394_rebuild_header() 함수의 정의

ip1394_rebuild_header() 함수 역시 eth_rebuild_header() 함수와 기본적으로 동일하다. 다만 참조하는 필드가 앞에서 hard_header() 함수에서 설정한 type정보에 기초한다는 사실만 조금 달라질 뿐이다. 만약 이 type 필드가 ETH_P_IP로 설정된 것이라면, arp_find() 함수를 사용해서 목적지 주소에 대한 하드웨어 주소를 구하려고 할 것이다. 그렇지 않다면, 소스 주소에(여기서는 daddr이라고 되어있다.) 디바이스가 사용하는 하드웨어의 주소를 복사해서 넣어준다. 복귀 값은 0이다. arp_find() 함수는 커널에서 제공하는 함수이다.

11.3.5.5. Receive

네트워크 디바이스 드라이버를 만드는데 있어서, 아마 앞에서 설명했던 과정들을 다 마쳤다면 이젠 정상적으로 동작하는가를 알기를 원할 것이다. 가장 먼저 해볼 수 있는 것은 제대로 패킷들을 받는가를

²⁷⁴ 이 부분은 ethernet과 관련된 eth_header() 함수를 참조하기 바란다. 거의 비슷하며, 소스 주소를 설정하는 부분만이 빠져있음을 확인할 수 있을 것이다. 정의는 Linux/net/eth.c에 있으며, 다음에 볼 rebuild_header와 관련된 ethernet 함수(eth_rebuild_header())도 찾을 수 있을 것이다.

알아보는 것이다. Ethernet과 같은 LAN에서는 네트워크를 통해서 다양한 패킷을 계속 주고 받기 때문에, 이것을 확인해 보기 가 쉽다.

```
void ip1394_rx(struct net_device *dev, int len, unsigned char *buf)
{
    struct sk_buff *skb;
    struct ip1394_priv *priv = (struct ip1394_priv *) dev->priv;

    skb = dev_alloc_skb(len + 2);
    if (!skb) {
        HPSB_PRINT(KERN_ERR, "ip1394 rx: low on mem");
        priv->stats.rx_dropped++;
        return;
    }
    skb_reserve(skb, 2); /* align IP on 16B boundary */
    memcpy(skb_put(skb, len), buf, len);

    /* Write metadata, and then pass to the receive level */
    skb->dev = dev;
    skb->protocol = ip1394_type_trans(skb, dev);
    skb->ip_summed = CHECKSUM_UNNECESSARY; /* don't check it */

    switch (ntohs(skb->protocol)) {
        case 0x0800 : /* IP packet */
        case 0x0806 : /* ARP packet */
        case 0x8861 : /* MCAP packet */
            break ;
        default:
            priv->stats.rx_errors++;
            priv->stats.rx_dropped++;
            return ;
    }
    /* Statistics */
    priv->stats.rx_packets++;
    priv->stats.rx_bytes += skb->len;
    /* We give the packet to the upper level */
    netif_rx(skb);
    return;
}
```

코드 673. ip1394_rx() 함수의 정의

ip1394_rx() 함수는 앞에서 초기화하는 과정을 통해서 주소공간에 대한 write 함수에서 불린다는 것을 이미 알고 있을 것이다. 넘겨주는 파라미터는 net_device 구조체(dev)의 포인터와 데이터의 길이(len), 그리고 실제 데이터가 들어가 있는 buffer(buf)이다. 이곳에서는 받은 데이터를 sk_buff와 같은 소켓 버퍼 구조체로 변화시키는 과정이 수반된다. 이렇게 변환된 데이터는 상위의 protocol 모듈로 전달될 것이다. 이것은 상위의 protocol 모듈에서 처리할 수 있는 데이터가 sk_buff 구조체이기 때문이다.

먼저 실제 데이터 보다 2 bytes정도 큰 크기의 sk_buff 구조체를 할당 받는다. 할당 받을 수 없다면, 받은 packet은 버린다고 생각하기에 통계정보를 가지는 priv->stats.rx_dropped를 증가시켜준다. skb_reserve()를 호출해서 소켓 버퍼의 첫 두 bytes를 먼저 예약해 둔다. 이것은 앞에서 2 bytes정도 크게 할당한 것과 마찬가지 이유로 IP header와 같은 것을 16 bytes에 정렬하기 위한 것이다.²⁷⁵ 이렇게 할당 받은 sk_buff에 앞에서 넘겨받은 buf 구조체의 값을 복사한다(memcpy()). 소켓 버퍼와 관련된 디바이스를 현재의

²⁷⁵ 32 bit machine의 경우 4 bytes씩 접근한다고 생각하는 것이 더 좋을 것이다. 메모리에 대한 연산이 정렬된 경우에는 더 빠르게 수행될 수 있다. 그렇지 않다면, 같은 데이터를 읽는데 한번 더 읽기를 수행해야 한다.

디바이스로 두고(dev), 프로토콜 번호는 ip1394_type_trans()를 호출해서 얻는다. 그리고, IP layer에서의 checksum 계산을 별도로 하지 않게 하기 위해서 CHECKSUM_UNNECESSARY를 ip_summed 필드에 둔다. 이전 앞에서 구한 프로토콜 번호를 가지고 어떤 패킷인지를 구분한다. 만약 IP, ARP, MCAP²⁷⁶ 패킷이 아니라면, 통계정보를 갱신한 후 바로 복귀 한다. 그렇지 않다면, 받은 패킷의 개수와 읽은 패킷의 길이 정보를 갱신하고, 상위의 프로토콜 모듈에 패킷이 도착했음을 알려주기 위해서 netif_rx()를 호출한다. 이것을 마치면 올바르게 패킷을 받았으므로 이전 그냥 돌아가면 될 것이다.

ip1394_type_trans() 함수는 받은 패킷에서 프로토콜 번호를 구하는데 사용하는 함수이다. 소켓 버퍼 구조체와 net_device 구조체를 넘겨받는다.

```
unsigned short ip1394_type_trans(struct sk_buff *skb, struct net_device *dev)
{
    struct write_reqb_hdr *raw_hdr;
    struct ip1394_hdr *hdr;
    struct hpsb_host *host = ((struct ip1394_priv *) (dev->priv))->host; /* The ieee1394 cards used */

    skb->mac.raw = skb->data;
    skb_pull(skb, dev->hard_header_len);
    /* hard_header_len= 20 ( 1394 write block request header + unfragmented encapsulation header)*/
    raw_hdr = (struct write_reqb_hdr *) skb->mac.raw;
    hdr = (struct ip1394_hdr *) (raw_hdr + 1);

    if ((raw_hdr->dst_id & 0x3F)==0x3F)
        skb->pkt_type = PACKET_BROADCAST;
    else if (raw_hdr->dst_id != host->node_id)
        skb->pkt_type = PACKET_OTHERHOST;

    if (hdr->u.uf_hdr.lf) {
        skb->protocol = 0x0;
        return 0 ;
    }
    if (hdr->u.uf_hdr.type == htons(ETH_P_ARP)) {
        convert_1394arp_to_arp(skb, dev ) ;
    }
    return hdr->u.uf_hdr.type;
};
```

코드 674. ip1394_type_trans() 함수의 정의

ip1394_type_trans() 함수는 받은 패킷으로부터 IP 혹은 ARP와 관련된 정보를 추출하는 역할을 하는 함수이다. 앞에서 이미 소켓 버퍼의 protocol 필드에 대한 설정을 보았다. Ethernet을 사용하지 않는 드라이버의 경우에는 protocol 필드 이외에도 pkt_type필드와 mac.raw 필드도 따로 처리를 해주어야 할 것이다²⁷⁷. mac.raw 필드의 경우에는 ARP(Address Resolution Protocol)과 같은 곳에서 사용하는 것으로서 net_device 구조체의 type필드에 일치하는 기계 주소(혹은 물리적인 주소)를 가르키고 있어야 한다. 여기서 보이는 코드에 대한 도움을 얻고자 하면, ether_type_trans()를 참조하기 바란다.

skb_pull() 함수를 사용해서 일단 하드웨어 헤더 부분을 소켓버퍼의 포인터를 옮겨서 제거한다(물론, 실제적으로는 제거되지 않는다. 단순히 데이터의 포인터만 이동할 뿐이다.). skb->mac.raw를 IEEE 1394 write request block header(write_reqb_hdr 구조체)를 가르키는 포인터로 변환해서 raw_hdr이 가르키도록 한다. 여기서 데이터 부분을 가르키기 위해서 hdr를 사용한다. hdr은 ip1394_hdr 구조체를 가르키게 될 것이다. 이전 패킷의 type을 결정하기 위해서 목적지의 주소를 본다(dst_id). 만약 이 값이 0x3F와 AND시켜서 0x3F가 나온다면 모든 host로 전달된다는 말이 되므로, 소켓 버퍼의 패킷 type필드에 PACKET_BROADCAST로 둔다. 그렇지 않고, 이 부분이 현재 호스트의 ID와 다르다면, 당연히

²⁷⁶ 커널 버전 2.4.16에서는 MCAP에 대한 것은 보이지 않는다. Linux/include/linux/if_ether.h에 다른 프로토콜들에 대한 정의를 찾을 수 있을 것이다.

²⁷⁷ Ethernet인 경우에는 Linux 커널에서 제공하는 ether_type_trans()와 같은 함수에서 이것을 처리해준다.

PACKET_OTHERHOST가 될 것이다. 만약 받은 IP over 1394 패킷의 헤더에 있는 If필드가 설정된 경우에는 소켓 버퍼의 protocol 필드에는 0을 두고, 바로 0으로 복귀한다. 만약 받은 패킷의 type이 ARP 패킷이라면, convert_1394arp_to_arp() 함수를 호출해서 IP over 1394의 ARP 패킷을 다시 원래의 ARP 패킷으로 변환한다. 복귀 값은 패킷의 type을 돌려줄 것이다. 이 패킷 type은 이미 네트워크 byte order로 되어 있다.

11.3.5.6. Transmit

패킷의 수신 과정을 보았으므로, 이제 볼 것은 패킷을 전달하는 것이다. 역시 이미 앞에서 초기화 된 인터페이스를 통해서 패킷은 전달된다. 이 경우 상위 프로토콜 모듈에서 받는 데이터의 형식은 소켓 버퍼(sk_buff) 구조체이다.

```
int ip1394_tx(struct sk_buff *skb, struct net_device *dev)
{
    int retval = 0;
    struct ip1394_priv *priv = (struct ip1394_priv *) dev->priv;

    if (test_bit(__LINK_STATE_XOFF, &dev->state)) { //shouldn't happen
        printk("Interface ip1394 is busy (but shouldn't)\n");
        priv->stats.tx_dropped++;
        priv->stats.tx_errors++;
        // retval = -EBUSY;
        // This is safer ,grayrain : I agree this
        retval = 0;
        // and this too
        //dev->tbusy = 0;
        netif_wake_queue(dev);
        goto tx_done;
    }
    if (skb == NULL) {
        return 0;
    }
    /* YOU COULD TRY TO COMMENT THIS LINE IF SOME PROBLEMS OCCUR */
    //dev->tbusy = 1; /* transmission is busy */
    netif_stop_queue(dev);
    dev->trans_start = jiffies; /* save the timestamp */
    retval = ip1394_hw_tx(skb, dev);
tx_done:
    dev_kfree_skb(skb); /* release it */
    return retval;
    /* zero == done; nonzero == fail */
    /* but if nonzero,especially -EINVAL, is returned, kernel get in panic */
}
```

코드 675. ip1394_tx() 함수의 정의

현재 네트워크 인터페이스의 연결(link)의 상태가 __LINK_STATE_XOFF(일반적으로 network과 관련된 queuing을 담당하는 layer에서 사용된다. 값은 0이며, 정의는 Linux/include/linux/netdevice.h에 있다.)이면, 상위 프로토콜 레이어(protocol layer)로부터 받은 패킷을 버린다. 물론 이때 통계정보에 대한 업데이트와 큐를 새로 깨어나게 하는 일이 필요할 것이다(netif_wake_queue()). 받은 소켓버퍼는 이곳에서 제거한다. 복귀 값은 0이 될 것이다. 즉, 상위 프로토콜 레이어는 Tx가 제대로 된 것으로 간주하게 될 것이다.²⁷⁸ 만약 넘겨받은 소켓 버퍼가 NULL이라면, 이것도 에러가 되며 0을 돌려준다. 이젠 하나의 패킷을 보내게 되므로, 더 이상 queue를 동작하는 것을 멈추게 하기 위해서 netif_stop_queue() 함수를 호출한다. 패킷의

²⁷⁸ 이것은 조금 문제가 있는 부분이다. 실제로 패킷이 전달이 되지 않았고, 네트워크 인터페이스의 상태가 OFF이므로 -EBUSY보다는 -EAGAIN과 같은 것을 돌려주는 것이 좋을 것이라고 생각한다.

전달 시간을 기록하기 위해서 jiffies를 trans_start에 넣고, 실제적인 전달은 ip1394_hw_tx() 함수를 호출한다. 전달이 끝나면, 해당 소켓 버퍼를 제거해 주어야 할 것이다.

```
int ip1394_hw_tx(struct sk_buff *skb, struct net_device *dev)
{
    u16 dst_node_id = *(u16*) skb->data ;
    struct ip1394_priv *priv = (struct ip1394_priv *) dev->priv;
    int retval ;

    skb_pull(skb, dev->addr_len) ;

    if ( skb->protocol == __constant_htons(ETH_P_ARP) ) {
        convert_arp_to_1394arp(skb, dev) ;
    }
    retval = write_no_wait(priv->host, ntohs(dst_node_id), (quadlet_t *) skb->data, skb->len);
    /* Statistics : */
    if (!retval) {
        priv->stats.tx_bytes += skb->len;
        priv->stats.tx_packets++;
    } else {
        printk("Write transaction problem\n");
        priv->stats.tx_dropped++;
        priv->stats.tx_errors++;
        //dev->tbusy = 0;
        netif_wake_queue(dev);
    }
    return retval;
}
```

코드 676. ip1394_hw_tx() 함수의 정의

ip1394_hw_tx() 함수가 하는 역할은 넘겨받은 소켓 버퍼의 데이터를 보고, 이 패킷이 어떤 패킷인지를 구별해서 적절한 처리를 해주는 것이다. 만약 ARP 패킷이라면 convert_arp_to_1394arp()를 호출해주어야 한다. 실제 전달은 write_no_wait()에서 일어나게 된다. Tx의 여러 유무에 따라 통계정도도 갱신될 것이다. 이때, 전달의 대상이 되는 하드웨어 주소를 소켓버퍼에서 떼어내기 위해 skb_pull() 함수를 사용했다. __constant_htons()는 호스트의 byte order를 네트워크 byte order로 바꾸기 위해서 사용했고, 이를 다시 상수 값으로 변환해서 소켓 버퍼의 protocol 필드와 비교했다. ETH_P_ARP는 ARP 프로토콜의 값을 의미한다.

```
void convert_arp_to_1394arp(struct sk_buff *skb, struct net_device *dev)
{
    struct ip1394_priv * priv = (struct ip1394_priv * ) dev->priv ;
    struct hpsb_host *host = priv->host ;
    struct ip1394_hdr * hdr = (struct ip1394_hdr*)skb->data ;
    struct arphdr * arp = (struct arphdr * )(hdr + 1);
    unsigned char * arp_ptr = (unsigned char * )(arp+1) ;
    struct ip1394_arp * arp1394 = (struct ip1394_arp * ) arp_ptr ;
    u16 phy_id = host->node_id & NODE_MASK ;

    skb_put(skb, sizeof(struct ip1394_arp) - ((arp->ar_hln << 1) + 8)) ;

    arp_ptr += dev->addr_len ;
    memcpy(&arp1394->sip, arp_ptr, 4) ;
    arp_ptr += dev->addr_len + 4 ;
    memcpy(&arp1394->tip, arp_ptr, 4) ;
    memcpy(&arp1394->suid_hi, &priv->eui_64_hi, 4) ;
    memcpy(&arp1394->suid_lo, &priv->eui_64_lo, 4) ;
```

```

arp1394->max_rec = max_rec[phy_id] ;
arp1394->sspd = sspd[phy_id] ;
arp1394->su_fifo_hi = fifo_offset[phy_id] >> 32 ;
arp1394->su_fifo_lo = fifo_offset[phy_id] & 0xffffffff ;
}

```

코드 677. convert_arp_to_1394arp() 함수의 정의

convert_arp_to_1394arp() 함수는 상위 ARP 프로토콜로부터 받은 ARP 패킷을 IP over 1394에서 사용하는 ARP 패킷으로 변환하기 위한 함수이다. 즉, 넘겨받은 소켓 버퍼는 ARP 패킷을 가지며, IP over 1394 헤더를 제외한 부분에 ARP를 위한 헤더(arphdr 구조체)를 가지고 있다. 이를 arp가 가르키도록 만든다. 그리고, 실제 ARP 패킷 데이터는 이 ARP 헤더의 정보 뒤에 들어가게 된다. 따라서, 다음의 [그림 89]과 같이 볼 수 있다.

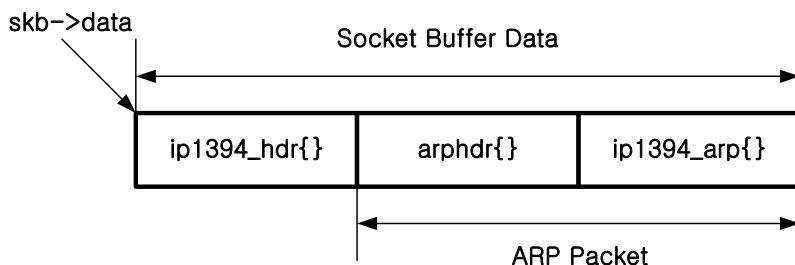


그림 89. IP over 1394의 ARP packet의 구조

문제는 이 ARP 패킷의 데이터에 들어가야 하는 것이 IP over 1394에서 사용하는 ARP정보로 되어있어야 하는 것이다. 따라서, 호스트의 정보(host)에서 node_id 필드중 물리적인 ID(phy_id)를 가지는 부분을 찾고, IP over 1394에서 사용할 ARP packet을 위한 ip1394_arp 구조체가 들어갈 자리를 확보하기 위해서 `skb_put()` 함수를 호출한다. 이전 소스와 타겟(target, 혹은 목적지 주소)를 이 ARP 패킷에 복사해 넣도록 한다. 이때 보내는 측(소스(source))의 주소를 저장했던 private 필드(priv)로부터 얻어와서 적절히 복사해 넣는다. 나머지 ARP 패킷의 필드들도 이곳에서 처리해 줄 수 있을 것이다. 복귀 값은 없다.

```

void convert_1394arp_to_arp(struct sk_buff *skb, struct net_device *dev)
{
    struct arphdr *arp = (struct arphdr *) skb->data ;
    unsigned char * arp_ptr = (unsigned char *) (arp+1) ;
    struct ip1394_arp * arp1394 = (struct ip1394_arp *) arp_ptr ;
    u16 node_id = ((struct write_reqb_hdr *) skb->mac.raw ) -> src_id ;
    int phy_id = node_id & NODE_MASK ;

    //arp_ptr += IP1394ARP_HW_ADDR_LEN ; /* arp->hln can be used */
    if (phy_id ^ NODE_MASK) {
        register_max_rec_sspd( dev, arp1394 , phy_id ) ;
    }
    fifo_offset[phy_id]=((u64)arp1394->su_fifo_hi << 32)|(arp1394->su_fifo_lo) ;
    arp->ar_hln = dev->addr_len ;
    node_id = htons(node_id) ;
    memcpy(arp_ptr, &node_id, dev->addr_len) ;
    arp_ptr+= dev->addr_len ;
    memcpy(arp_ptr, &arp1394->sip, 4) ;
    arp_ptr+= 4 ;
    memset(arp_ptr, 0, dev->addr_len) ; /* target HW addr is set to 0 */
    arp_ptr+= dev->addr_len ;
    memcpy(arp_ptr, &arp1394->tip, 4) ;
    //skb->len -= (sizeof(struct ip1394_arp) - ((arp->ar_hln << 1) + 8)) ;
}

```

코드 678. convert_1394arp_to_arp() 함수의 정의

convert_1394arp_to_arp() 함수는 앞에서 했던 일을 반대로 수행하는 것이다. 즉, 받은 IP over 1394 ARP 패킷을 다시 원래의 ARP 패킷으로 변환해 주는 일을 한다.²⁷⁹ 전달 받은 패킷으로부터(skb->data) ARP 헤더 자료구조를 찾고, 여기서 다시 헤더를 제외한 ARP 데이터를 ip1394_arp 구조체로 변환한 후, 소켓 버퍼의 mac.raw로부터 소스의 ID를 구해 node의 ID로 두고(node_id), 이 정보를 이용해 다시 물리적인 ID(phy_id)를 구하게 된다. 물리적인 ID의 bit 값이 전부가 다 1이 아니라면, register_max_rec_sspd() 함수를 호출해서 최대 레코드 크기와 스피드 맵(speed map)을 갱신한다. 패킷의 fifo 옵셋도 이곳에서 갱신될 수 있을 것이다. 이젠 ARP 패킷의 데이터를 원래의 ARP 패킷으로 만들어주는 일이 될 것이다. node_id를 하드웨어 주소 길이 만큼을 복사해서 ARP 데이터에 넣어주고, 소스의 IP주소(sip)도 복사해 준다. 목적지 주소는 0으로 설정하고, 다시 목적지 IP 주소(tip)도 복사해 준다.

```
int write_no_wait(struct hpsb_host *host, u16 node_id, quadlet_t * buffer, size_t length)
{
    struct hpsb_packet *packet;
    struct xmit_task *xtask;

    if (length == 0) {
        return -EINVAL;
    }
    packet = hpsb_make_writebpacket(host, node_id, fifo_offset[node_id & NODE_MASK], length );
    if (!packet) {
        printk("Couldn't create a packet\n");
        return -ENOMEM;
    }
    packet->expect_response = 0 ;
    xtask = create_xtask((void (*)(void *)) write_completed, packet);
    queue_task(xtask->tq, &packet->complete_tq);
    /* We copying only existing bytes */
    memcpy(packet->data, buffer, length);
    /* We send the packet end we go */
    return !hpsb_send_packet(packet);
}
```

코드 679. write_no_wait() 함수의 정의

다시 원래 Tx에 관련된 부분으로 돌아가서 실제적으로 하위의 IEEE 1394쪽에 대한 보내기를 보면, write_no_wait() 함수가 처리한다. 만약 보내고자 하는 데이터의 길이가 0이라면 당연히 -EINVAL을 돌려줄 것이다. 그렇지 않다면, hpsb_make_writebpacket() 함수를 호출해서 보내고자 하는 패킷을 write request block 패킷으로 만들어 준다. 필요한 정보는 호스트 정보와 노드의 ID, 목적지가 되는 노드가 사용하는 FIFO의 주소와 데이터 길이가 될 것이다. 이 함수의 복귀 값은 write request block 패킷에 대한 포인터(packet)가 된다. 만약 이 packet 변수의 값이 NULL이라면, 더 이상 할당할 수 있는 패킷을 위한 시스템 메모리가 없다는 말이되므로 -ENOMEM을 돌려준다. hpsb_packet 구조체(packet)의 예상하는 응답 시간에는 0을 주고, Tx가 마무리 되었을 때 호출될 함수를 만든다(create_xtask()). 만들고자 하는 task²⁸⁰가 수행할 일은 write_completed() 함수이다. 생성된 task는 관련된 패킷의 complete_tq에 삽입된다. 이젠 하위 레이어에 데이터를 보내기에 앞서 만든 packet에 실제 데이터를 실어준다(memcpy()). 하위 레이어에 대한 보내는 요청은 hpsb_send_packet() 함수이다.

```
struct xmit_task {
    struct hpsb_packet *p; /* 상위 레이어에서 사용하는 hpsb_packet 구조체에 대한 포인터 */
```

²⁷⁹ 즉, 여기서 보여주는 것은 IEEE 1394 card들 간에 통신하는 것이므로, 전송 미디어에만 변화가 있을 뿐이지, 상위 프로토콜 레이어는 변화가 없다는 것을 명심하기 바란다.

²⁸⁰ 여기서 말하는 task는 일반적인 process나 thread와는 다르다. 즉, 커널의 일부로서 호출될 function 정도로 생각하면 될 것이다. 일종의 callback 함수가 된다.

```
struct tq_struct *tq; /* Bottom half로 수행된 task의 구조체에 대한 포인터 */
};
```

코드 680. xmit_task 구조체의 정의

xmit_task는 하위 레이어에서 callback을 위해서 사용하는 bottom half를 위한 구조체이다. Bottom half로 수행될 함수는 write_completed() 함수라는 것은 앞에서 이미 보았다.

```
void write_completed(struct xmit_task *xtask)
{
    struct ip1394_priv *priv = (struct ip1394_priv *) ip1394_dev.priv;
    // printk("Unified write has succeeded, freeing tq packet: %p\n", xtask->tq);
    // ip1394_dev.interrupt = 1;
    free_tlabel(priv->host, xtask->p->node_id, xtask->p->tlabel);
    kfree(xtask->tq);
    free_hpsb_packet(xtask->p);
    kfree(xtask);
    netif_wake_queue(&ip1394_dev); // ip1394_dev.tbusy = 0; /** we are ready */
    // ip1394_dev.interrupt = 0;
}
```

코드 681. write_completed() 함수의 정의

write_completed() 함수는 보내고자 하는 패킷이 전달된 후에 호출되는 함수이다. 이 함수가 하는 일은 보내기가 끝났으므로 관련된 커널 메모리 및 앞에서 보내기 연산을 수행하기 전에 멈춘 네트워크 인터페이스 드라이버의 큐를 다시 가동시켜서(netif_wake_queue()), 다른 보내고자 하는 패킷을 받아드리는 것이다. 해당 tq_struct 구조체를 해제(free) 시켜주고, 패킷을 해제하기 위해 free_hpsb_packet() 함수를 호출한다. 또한 xmit_task 구조체 자체도 해제하기하고 netif_wake_queue() 함수를 호출한다.

```
struct xmit_task *create_xtask(void (*routine)(void *), struct hpsb_packet *packet)
{
    int kmflags = in_interrupt() ? GFP_ATOMIC : GFP_KERNEL;
    struct tq_struct *task = kmalloc(sizeof(struct tq_struct), kmflags);
    struct xmit_task *xtask = kmalloc(sizeof(struct xmit_task), kmflags);
    if ((task == NULL) || (xtask == NULL)) {
        return NULL;
    }
    /* This are task data : */
    xtask->p = packet;
    xtask->tq = task;
    /* This is our task : */
    // task->next = NULL;
    task->sync = 0;
    task->routine = routine;
    task->data = xtask;
    return xtask;
}
```

코드 682. create_xtask() 함수의 정의

create_xtask() 함수는 callback으로 사용할 함수를 생성하는 역할을 수행한다. 먼저 tq_struct 구조체를 메모리 공간에서 할당 받기 위해서, 현재 인터럽트가 진행중인 경우와 그렇지 않은 경우에 대해서 flag를 설정한다. 인터럽트가 진행 중이라면 GFP_ATOMIC을 사용하고, 그렇지 않다면 GFP_KERNEL을 사용한다. GFP_ATOMIC은 주로 인터럽트 핸들러와 같은 곳에서 프로세스의 context와는 상관없이 메모리를 할당하려는 경우에 사용하며, GFP_KERNEL은 일반적으로 사용하는 커널 메모리 할당 flag이다. 보내는 task를 위한 xtask도 같이 할당 받는다. 만약 할당 받을 수 없다면 NULL을 돌려준다. 이전 할당 받은 자료구조 들을 초기화 해주면 될 것이다.

```
/* Linux/include/linux/tqueue.h에서 */
struct tq_struct {
    struct list_head list;          /* 활성화된 bottom half의 연결 리스트 */
    unsigned long sync;             /* 반드시 0으로 초기화 해주어야 한다. */
    void (*routine)(void *);        /* Bottom half로 호출하고자 하는 함수의 포인터 */
    void *data;                     /* Bottom half 함수에 대한 인자(argument) */
};
```

코드 683. tq_struct 구조체의 정의

즉, task의 routine 필드에서는 앞에서 본 write_completed() 함수가 될 것이며, data 필드에는 할당 받아서 초기화한 xtask 변수가 될 것이다.

11.3.5.7. Config

Config은 네트워크 인터페이스의 설정(configuration)을 바꿀 경우에 사용된다. 예를 들어 사용하는 기본 I/O를 위한 주소를 변경한다던가, 혹은 인터럽트 번호등을 변경하는데 사용될 수 있다. 넘겨지는 파라미터는 ifmap이라는 구조체의 정의를 따른다. ifmap 구조체의 정의는 다음과 같다.

```
struct ifmap {
    unsigned long mem_start;          /* 공유 메모리의 시작주소 */
    unsigned long mem_end;            /* 공유 메모리의 끝주소 */
    unsigned short base_addr;         /* 네트워크 인터페이스의 I/O 기본 주소 */
    unsigned char irq;                /* 할당할 IRQ 번호 */
    unsigned char dma;                /* DMA위한 channel 번호 */
    unsigned char port;               /* Multiport 사용 기기의 경우 사용할 port */
    /* 3 bytes spare */
};
```

코드 684. ifmap 구조체의 정의

위와 같이 넘겨온 자료 구조를 사용해서 해당 네트워크 인터페이스의 각종 디바이스 정보를 변경해줄 수 있다. 하지만, 이것은 어디까지나 하드웨어적인 정보에 기초해야 하므로 지원하지 않는다면 에러를 돌려주어도 된다.

```
int ip1394_config(struct net_device *dev, struct ifmap *map)
{
    printk("Calling ip1394_config function\n");
    if (dev->flags & IFF_UP)           /* can't act on a running interface */
        return -EBUSY;
    if (map->base_addr != dev->base_addr) {
        return -EOPNOTSUPP;
    }
    if (map->irq != dev->irq) {
        return -EOPNOTSUPP;
    }
    return 0;
}
```

코드 685. ip1394_config() 함수의 정의

ip1394_config() 함수에서는 만약 현재 네트워크 인터페이스를 사용 중이라면(IFF_UP), -EBUSY를 돌려준다. 또한 ifmap의 base_addr과 현재 사용 중인 base_addr이 다르거나, 혹은 irq가 다르다면, -EOPNOTSUPP(Error Operation Not Support)를 돌려준다. 그 외의 값에 대해서는 단지 0을 돌려줄 것이다.

따라서, 새로운 설정은 적용하지 않는다. 따라서, 이 함수는 단순히 네트워크 인터페이스로서의 구성을 갖추기 위한 함수로 생각된다.

11.3.5.8.Ioctl

디바이스 드라이버에 대한 I/O control은 특정한 정보를 구한다거나 debugging과 같은 곳에서 사용할 수 있다. 만약 디바이스 드라이버의 특정한 특징을 살리고자 한다면, 이를 이용해도 좋을 것이다.

```
/** IOCTL commands */
int ip1394_ioctl(struct net_device *dev, struct ifreq *rq, int cmd)
{
    switch (cmd)
    {
        // case IP_ADD_MEMBERSHIP :
        //     break ;
        // case IP_DROP_MEMBERSHIP :
        //     break ;
        // case IP_MULTICAST_IF :
        //     break ;
    }
    printk("Calling ioctl function\n");
    return 0;
}
```

코드 686. ip1394_ioctl() 함수의 정의

실제적으로 ip1394_ioctl() 함수에서 해주는 일을 아무것도 없다. 단지 0만 돌려준다. 만약 특정한 것을 설정해서 사용하고 싶다면, 이런 곳이 적당하리라고 생각한다.

11.3.5.9.Statistics

통계정보는 ifconfig과 같은 명령어를 사용해서 네트워크 디바이스 드라이버가 현재 어떤 상황인가를 알고자 할 경우에 커널에서 호출될 것이다.

```
/** Return statistics to the caller */
struct net_device_stats *ip1394_stats(struct net_device *dev)
{
    struct ip1394_priv *priv = (struct ip1394_priv *) dev->priv;
#ifndef DEBUG
    printk("Calling stats function\n");
#endif
    return &priv->stats;
}
```

코드 687. ip1394_stats() 함수의 정의

이미 Tx나 Rx에서 디바이스 드라이버의 통계정보에 대한 일을 수행해 왔다. 따라서, 디바이스의 private 데이터 저장 공간의 통계정보를 위한 필드 주소를 복귀 값을 넘겨주는 것으로 함수는 끝난다.

11.4. IEEE 1394 for Linux에서 남은 일

리눅스상에서 구현된 IEEE1394 디바이스 드라이버에 대한 자세한 설명을 거의 찾아볼 수 없기에 내용의 부족함이 보일 것이다. 차후에 이에 대한 정리를 더 곁들일 것임을 밝힌다. 시간이 허락한다면, 직접 IEEE 1394 디바이스 드라이버의 코드를 분석 및 정리해 나갈 것이다. 일단 계략적인 정보만을 알게 된 것에 만족하도록 하자. 더 자세한 정보를 원한다면 직접 소스코드를 보고 공부하기 바란다. 내가 할 수 있는 한계는 아직 여기까지이다.

12. PCMCIA for Linux

12.1. PCMCIA란?

PCMCIA(Personal Computer Memory Card International Association)은 컴퓨터관련 회사들의 연합으로, 컴퓨터에 연결되는 장치에 대한 독점적인지 않은 소프트웨어와 하드웨어 표준을 제정하는 단체이다. PCMCIA 표준은 명세(specification)와 장치(device), 그리고, 관련된 모든 것을 포괄적으로 포함하며, 다시 이것은 하드웨어와 관련된 모든 구성요소(component)를 PC Card라는 용어를 사용해서 나타낸다.

PC Card Adapter는 PC Card가 연결된 host 시스템²⁸¹과의 모든 인터페이스를 수행하는 chip이다. 만약 PC Card라는 말이 없이 adapter라는 말만을 사용할 때에는 일반적인 경우 시스템의 slot에 삽입되는 adapter card를 의미한다. 또한, PC Card의 표준에 CardBus라는 말을 더함으로써, 16-bit PC Card Adapter와 CardBus PC Card Adapter라는 기능적인 구분이 생겨나게 되었다. 결과적으로 PC Card System은 PC Card들을 지원하는 컴퓨터 시스템을 일컫는 말로 사용되며, 하나나 그 이상의 PC Card들을 수용할 수 있는 Socket을 가진다.

PC Card Socket이라는 것은 PC Card가 삽입될(insert) 수 있는 저장소(receptacle)이다. 즉, 마치 diskette drive와 같이 디스크을 삽입하는 곳과 같은 뜻이다. 따라서, socket은 PC Card가 잘 정렬(align)되어서 삽입되도록 guide를 옆쪽에 가지고 있으며, 빼기 버튼(ejection button)을 가지고 있다.

PC Card의 종류로는 크게 Type I, Type II, Type III, Type IV가 있다. Type I은 일반적으로 메모리 카드를 의미하며, Type II는 I/O card나 network adapter 및 data/fax modem을, Type III는 hard drive를 구현하는데 사용된다. Type IV는 아직 PC Card 표준에서 지원되지 않는 것으로 Type II보다 거의 세배 크기의 두께를 가진다. 이 이외에도 각각의 Type에 대한 Extension이 있는데, 이는 카드의 길이를 늘려서 필요한 기능을 두는 것이다. 예를 들어서, 안테나를 필요로 하는 PC Card를 만들기를 원할 때 사용될 수 있을 것이다.

CardBus PC Card는 High-speed LAN이나, SCSI, Multimedia 기기들을 만들고자 할 때 사용된다. 즉, 중요한 성능상의 향상을 도모한다고 볼 수 있다. CardBus를 위한 PC Card 표준은 PC Card를 위한 BAR(Base Address Register)와 CIS(Configuration Information and Status) format 및, 이러한 디바이스를 지원하기 위한 host PC Card 소프트웨어의 변경들을 모두 포함한다.

PC Card Adapter는 16-bit PC Card Adapter와 32-bit PC Card Adapter로 다시 나누어진다. PC Card Adapter는 먼저 시스템 버스와 PC Card 확장(expansion) 버스 사이에서 변환(conversion)을 담당한다. 예를 들어서, ISA 버스와 PC Card 사이에서 PC Card Adapter가 변환을 수행하는 것을 들 수 있다. 또한 PC Card Adapter는 PC Card Socket와의 인터페이스를 담당한다. 즉, 인터럽트의 전달과 파워 컨트롤, 시간에 관련된 연산에 대한 접근 등을 담당한다. PC Card Bus는 하나의 인터럽트를 PC Card로부터 받아서, PC Card Adapter로 보내며, 다시 PC Card Adapter는 이러한 인터럽트를 해당하는 IRQ line으로 보낸다. 발생될 수 있는 인터럽트로는 카드의 상태 변경 인터럽트(Card Status Change Interrupt)²⁸²와 일반적인 PC Card의 I/O로부터 기인한 I/O 연산 인터럽트가 있다. 이 두 가지의 인터럽트는 반드시 구분되어야 하며, 모든 PC Card가 두 가지의 인터럽트를 발생시키는 것이 가능하다. 마지막으로 PC Card 표준 2.X버전에 속하는 PC Card Adapter들은 PC Card 메모리를 시스템의 메모리로, 메모리 mapping이나 I/O 윈도우(window)를 사용해서 mapping하는 역할을 책임진다. 대부분의 PC Card Adapter들은 2개의 Socket을 지원할 수 있는데, 시스템이 만약 2개보다 많은 수의 Socket을 가지고 있다면, 둘 이상의 PC Card Adapter를 가질 것이다. 실제로는 최대 16개의 PC Card Adapter를 PC Card 시스템이 가질 수 있으며, 따라서, 전체 32개의 Socket이 존재할 수 있다.

32-bit PC Card Adapter는 32-bit 시스템을 위한 것으로 PCI나 혹은 CardBus Bridge라고 불리며, 인터페이스 컨트롤러 chip이다. 이러한 chip은 시스템의 board에 장착되어서 PC Card Socket에 연결되어 있다. 16-bit의 PC Card Adapter와 동일한 역할을 수행하지만, 32-bit PC Card Adapter의 경우에는 PCI 버스를 CardBus

²⁸¹ 즉, 시스템 버스가 될 것이다. ISA나 Microchannel, 혹은 PCI가 된다.

²⁸² Card detect, read/busy, battery low등의 상태 변경이 있을 수 있다.

버스에 연결하는 일을 한다. CardBus Adapter들은 메모리나 I/O를 mapping하지 않으며, 대신에 host 시스템에 대한 접근을 직접적으로 decoding한다. 또한 16-bit PC Card들은 PC CardBus Socket에서도 동일하게 동작할 것이다.

이전 PC Card가 동작하는 환경에 대해서 알아보도록 하자. 이것에는 크게 Card Services(CS), Socket Services(SS), Client Device Driver(CDD)가 있다.

12.2. Card Services(CS)

Card Services는 운영체제에 의존적인 PC Card 클라이언트 디바이스 드라이버(client device driver)에 대한 API의 표준을 제공하는 역할을 한다. 즉, 이러한 API를 통해서 클라이언트 디바이스 드라이버는 요청(request)을 할 수 있으며, 시스템의 자원과 PC Card와 관련된 사건(event)들을 받을 수 있게 된다. 즉, client는 PC Card를 지원하기 위해서 시스템에 있는 물리적인 하드웨어를 이해할 필요가 없게 된다.

Card Service는 운영체제에 의존적인 부분이기 때문에, 다른 운영체제에 내장된 함수나 디바이스 드라이버와 같은 분리된 구성요소로서 제공되는 것이 가장 이상적이며, client 디바이스 드라이버와는 Client/Server 모델로 동작하도록 구성된다. 예를 들어, 클라이언트 디바이스 드라이버가 자원에 대한 요청을 한다면, Card Service는 서버로 동작해서 사용 자원들을 요청에 맞게 할당하는 처리하도록 한다. 따라서, Card Service는 사용 가능한 자원에 대한 관리를 해주어야 하며, 때로는 resource manager의 도움을 받아 resource map이란 것으로 이러한 기능을 가능하도록 한다.

Resource Manager의 기능은 컴퓨터에 의해서 이미 사용중인 자원들을 결정하고, 이를 Card Service에 알려주는 일을 한다. 예를 들어서, 만약 COM1이 이미 시스템이 사용중이라면, 이를 요구하는 모든 클라이언트 디바이스의 요구에 대해서는 거부를 돌려줄 것이다.

또한, Card Service는 PC Card 시스템의 하드웨어를 관리하기 위해서 Socket Service(SS)에 의존한다. Socket Service는 아래에서 보도록 하겠다. 따라서, Card Service는 클라이언트 디바이스 드라이버와 Socket Service간의 중간에 위치한다고 볼 수 있다.

12.3. Socket Service(SS)

Socket Service(SS)²⁸³는 CS와 같이 API로 구성된다. 하지만, CS가 많은 client를 가지고 있는데 반해서, Socket Service는 두 가지의 client만을 가진다. 즉, CS와 Memory Technology Driver만이 Socket Service의 client로 사용된다. 또한, 다양한 PC Card adapter들이 시장에서 사용되기 때문에, Socket Service의 구현도 여러 가지가 있을 수 있다. 예를 들어서, Card Bus를 지원하는 Socket Service는 ISA에 기초를 둔 PC Card를 지원하는 Socket Service와는 달라진다. 결과적으로 각각의 PC Card Adapter들은 자신만의 Socket Service를 가질 필요가 있게 된다.

Socket Service는 직접적으로 시스템에 있는 PC Card 하드웨어에 대한 접근을 하기에, 시스템의 BIOS에 구현되거나 혹은, 디바이스 드라이버의 형태로 구현되어 CS와 인터페이스 한다. 따라서, CS는 하나만이 시스템에 존재하게 되며, 각각의 PC Card 하드웨어에 대해서 여러 개의 Socket Service가 존재할 수 있다.

12.4. Client Device Drivers(CDD)

Client Device Driver는 CS에 의존적인 디바이스 드라이버이다. 즉, Client Device Driver는 CS가 제공하는 API를 사용해서 시스템의 자원을 할당 받고, 요구되는 일을 처리할 수 있게 된다. 즉, CS의 API를 통해서 시스템에서 발생되는 PC Card에 대한 event를 전달 받을 수 있다. 여기서 중요한 점은 CS와 SS는 PC Card에 대한 설정에 책임이 없으며, Client Device Driver가 모든 역할을 수행한다는 점이다. 단지 CS와 SS는 Client Device Driver의 요청에 반응하는 역할만을 수행할 따름이다.

이상과 같은 CS, SS, CDD를 간단히 표현하자면, [그림]와 같이 될 것이다. 그림에서 보다시피 시스템에는 하나의 CS와 각각의 BUS 타입에 따른 SS, 그리고, SS가 관리하는 PC Card Adapter가 있으며, 이를 통해서 PC Card Socket에 PC Card가 연결됨을 알 수 있다.

²⁸³ 여기서 말하는 Socket과 네트워크의 관리에서 논의한 Socket을 혼동하지 말기를 바란다. 이곳에서 말하는 Socket이란 단순히 PC Card가 제공하는 API의 집합이라고 생각하도록 하자.

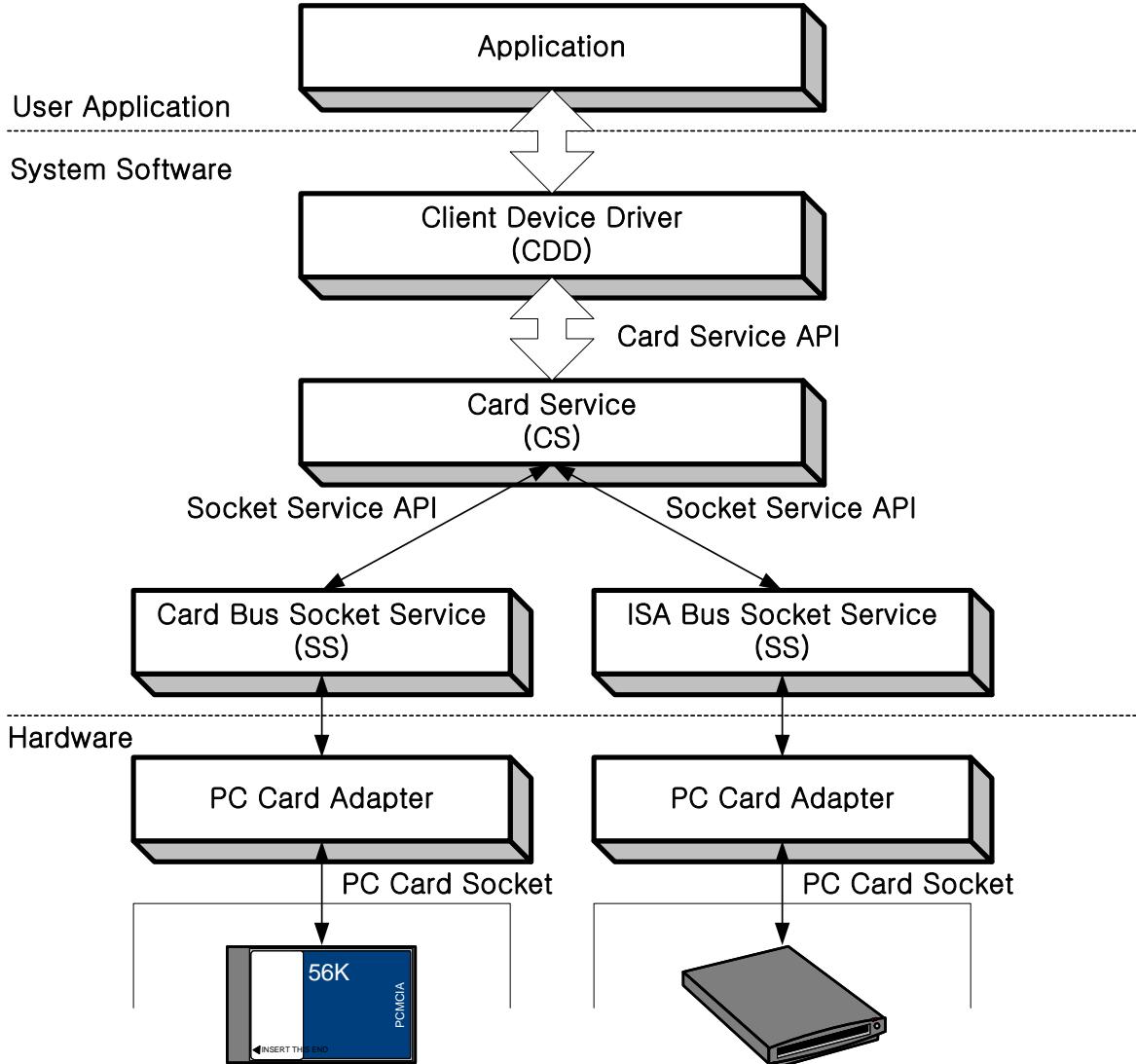


그림 90. PC Card Software의 Architecture

Client Device Driver의 역할을 좀더 자세히 들여다보면, Client Device Driver는 CS와의 인터페이스를 책임지는 프로그램으로, CS의 도움을 받아서 여러 개의 동일한 PC Card를 관리하거나, 혹은 특정한 PC Card의 관리를 책임진다. 이러한 Client Device Driver의 특수한 형태로는 Super Client Driver가 있는데, 이는 universal card installer라고도 불리운다. 이것은 여러 개의 Client Device Driver의 모음으로 이미 알려진 PC Card의 데이터 베이스를 가지고, PC Card가 시스템에 삽입될 때, 자신의 데이터 베이스를 보고 해당하는 Client Device Driver를 가져오는 역할을 한다. 하지만, 이러한 방식으로 Super Client Driver를 구성하는 것은 데이터 베이스가 항상 새롭게 갱신되어야 한다는 단점이 있다. 이것을 극복하기 위한 방법이 바로 CIS(Card Information Structure)를 가지는 것이다. 즉, 이 PC Card의 CIS에 저장된 정보를 바탕으로 PC Card를 설정하는 것이다. CIS에 대해서는 나중에 다시 보도록 할 것이다. 이곳에서는 이러한 것에 있다는 사실만 알기 바란다.

Client Device Driver의 종류(Type)는 PC Card Adapter에 연결된 디바이스가 어떤 것인가에 따라서 구분되며, 이에는 앞에서 보았듯이 MTD(Memory Technology Device), I/O Device들과 같은 형태의 디바이스를 다루는 디바이스 드라이버로 구분될 것이다.

그럼, 여기서 잠시 Client Device Driver와 CS를 관련 지어서, 어떻게 Client Device를 구현할 수 있는지를 보도록 하자. 먼저 대부분의 PC Card Client Driver들은 디바이스 드라이버 형태로 구현된다. 따라서, 다른 디바이스 드라이버와 마찬가지로 PC Card Client Driver는 동작하기를 원하는 운영체제의 구성에 따라야 할 것이다. 만약 Linux상에서 구현되는 Client Device Driver는 당연히 Linux에서 동작하는 다른 디바이스 드라이버의 모델이랑 같게 구현될 것이다.

가장 먼저 실행되는 부분은 항상 초기화(initialization)가 될 것이다. 이 부분은 한번만 수행되기에 수행을 마친 후에는 메모리를 차지 하지 않아도 될 것이다²⁸⁴. Call Back 핸들러 함수는 CS API를 호출해서 등록되는 함수로서 특정 event의 처리를 맡는다. 발생될 수 있는 event로서는 PC Card의 insert와 eject 및 Power Management(PM)등이 있을 수 있을 것이다. 만약 프로그램 하려는 디바이스가 PM 기능을 제공한다면 Client Device Driver역시 지원해 주어야 할 것이다. 따라서, Callback 핸들러 함수는 등록된 event가 발생될 때마다 호출될 것이며, 핸들러는 어떤 event가 발생했는지를 물어서, 해당하는 처리를 해 줄 수 있어야 할 것이다. 이것은 PC Card가 portable하게 구성된다는 점에서 기인한다고 보면 될 것이다. 한가지 주의할 점은 Callback 핸들러 함수의 수행은 CS에 의해서 시작되기에, 핸들러의 수행이 끝나면 다시 CS로 제어가 돌아간다는 점이다. ISR(Interrupt Service Routine)은 디바이스에서 발생되는 인터럽트를 처리하는 부분이다. 즉, 디바이스에 대한 입출력 요구가 완료되는 시점에서 인터럽트는 발생되며, 이것을 처리하는 것은 디바이스 드라이버에서 정해준 ISR이 맡는다.

이제 우린 기본적인 PC Card 시스템에 대한 이해를 했다. 다음으로 볼 것은 실제 시스템에서 어떻게 PC Card 디바이스 드라이버가 구현되는 가를 볼 차례이다. 실제에서 중요한 부분은 이미 Card Service와 Socket Service는 구현되어 있기에, 우리가 볼 부분은 Client Device Driver 부분으로 한정될 것이다. 앞으로의 내용도 이것을 중점적으로 분석하게 된다.

12.5. Linux에서의 PCMCIA 구현

Linux 커널에서의 PCMCIA 구현은 크게 3개의 주요 component로 되어 있다. 가장 하위의 level에 있는 것이 socket driver들이며, 그 상위에 Card Service 모듈이 올라간다. 특정 card에 대한 드라이버들은 Card Service layer위에 올라가게 된다. 특별히 Card Service client중에서 Driver Service라고 불리우는 component는 사용자 level의 utility program과 kernel 간의 link 역할을 하는 프로그램으로 동작한다.²⁸⁵

Socket driver layer는 Socket Service API에 기초하며, 기본적으로 2개의 socket driver 모듈을 가지고 있다. 이것이 바로 tcic 모듈과, i82365 모듈이며, 각각은 Databook TCIC-2계열의 호스트 controller와 Intel 82365sl계열을 지원한다. 특히 i82365 모듈의 경우에는 다양한 Intel호환의 controller들을 지원하는데, Cirrus Logic, VLSI, Ricoh, Vadim 칩등을 포함한다. 또한 i82365 모듈의 경우에는 Yenta register-level spec.을 따르는 CardBus controller에 대한 지원도 구현하고 있다.

Card Service는 PCMCIA package에서 가장 큰 모듈로서 MS-DOS의 Card Service와 비슷한 API를 제공하며, UNIX환경에 맞게 구현되었다. Linux의 경우에는 Solaris에서 사용하는 인터페이스 spec.을 부분적으로 기반하고 있으며, pcmcia_core 모듈에 구현되어 있다. 대부분의 PCMCIA 2.1 spec.의 특성들이 구현되어 있으며, 몇몇 PC Card 95 특성들도 가지고 있다.

Driver Service는 사용자 모드의 pseudo-device로서 구현되어 있으며, utility프로그램에서 Card Service 함수들을 접근하기 위한 것이다. 우리가 앞으로 보게될 Client Driver의 경우에도 이 Driver Service를 사용하게 되는데, 이와 같은 것이 가능하기 위해서는 모든 Client Driver들을 관리하고, 물리적인 socket들과 driver들을 일치(match)시켜주는 역할을 Driver Service가 맡게 된다. 구현하고 있는 모듈은 ds이다.

위에서 설명한 것을 간략히 그림으로 나타내면, [그림 91]와 같이 될 것이다. 보기에서 우리가 접근하고 관심을 가진 것은 하위의 Socket Service, Card Service, Driver Service가 아니라 Client Driver임을 명심하도록 하자.

²⁸⁴ Linux에서는 초기화에 관련된 부분을 해제할 수 있도록 하고 있다.

²⁸⁵ 이곳에서 이야기 하는 것은 PCMCIA의 programmer's guide의 내용을 기초로해서 작성한 것이다.

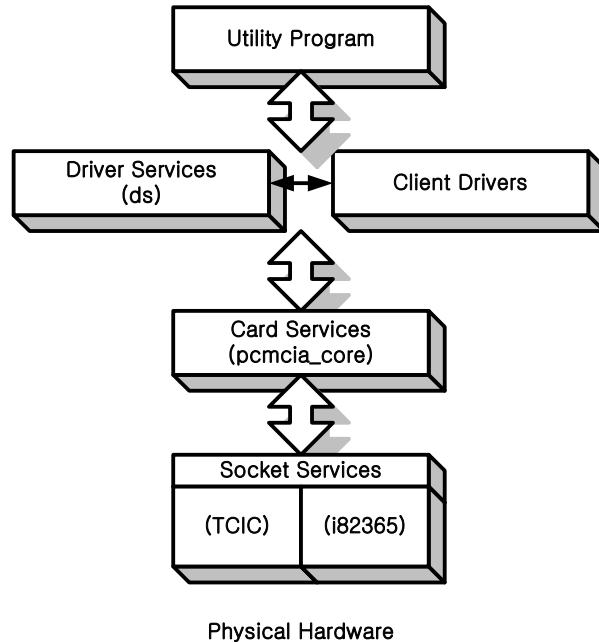


그림 91. Linux에서의 PCMCIA 구현 architecture

실제로 나중에 보게될 Client Device Driver는 Driver Service와 Card Service만을 이용할 수 있도록 구성되며, Socket Service에 대한 것은 직접적인 호출이 보이지 않을 것이며, Driver Service는 Client Driver의 등록과 해제 및 interrupt dispatching과 같은 역할을 해줄 것이다.

12.5.1. Socket interface

PC Card 버스는 기본적으로 2가지의 동작 모드를 가지며, 각각이 memory-only과 memory and I/O로 나누어진다. Memory-only 모드는 원래 Version 1.0 spec.에서 정의된 것으로, 단지 간단한 메모리 card를 지원하기 위한 것이다. 두번째 모드는 Version 2.0 spec.에서 정의된 것으로, 몇몇 메모리 card 제어 시그널들에 대해서 I/O port addressing과 I/O interrupt 시그널처리를 지원하기 위해서 다시 정의하고 있다.

PC Card 장치들은 두개의 메모리 영역을 가진다. 각각은 attribute 메모리와 common 메모리로 구분된다. 인터페이스는 각 타입의 메모리에 대해서 16MBytes까지의 주소를 사용할 수 있으며, attribute 메모리는 특히 descriptive 정보와 설정 레지스터들을 담는데 사용된다. Common 메모리의 경우는 메모리 card의 대용량 저장장치를 포함하며, I/O card들의 경우에는 디바이스 buffer들을 가지게 된다. Version 2.0 spec.의 PC Card와 호환되는 모든 card들은 반드시 CIS(Card Information Structure)를 attribute 메모리에 가져야 하며, 이것으로 card에 대한 정보를 기술하거나 어떻게 설정되어서 사용되는지를 나타내게 된다.

앞에서 설명한 제어를 위한 시그널들은 card들이 자신의 동작 상태를 host controller에 보고할 수 있도록 하고 있으며, 이러한 시그널들은 card의 검출, ready/busy, write protect, battery low, battery dead와 같은 것을 포함하고 있다. 따라서, 이러한 시그널들에 대해서 적절히 대응할 수 있도록 소프트웨어의 구성이 필요하게 된다.

메모리와 I/O 인터페이스 모드는 card가 I/O port로 64Kbytes의 주소 공간을 사용할 수 있도록 해주고 있다. 또한 card들에 I/O interrupt 신호(signal)를 알릴 수 있도록 하며, 한 card의 output을 host 시스템의 speaker로 전달(route)할 수 있도록 한다. 이 모드에서는 여러개의 메모리 card의 제어 신호(control signal)은 사용할 수 없는데, 이는 이러한 신호를 전달하는 pin이 여분(extra)의 I/O card 신호(signal)를 나르는데 사용되기 때문이다. 어떤 card에서는 이러한 신호들이 attribute 메모리에 있는 특정한 설정(configuration) 레지스터로 부터 읽혀질 수 있으며, 이러한 목적으로 사용하는 레지스터를 “Pin Replacement Register”라고 한다.

12.5.2. Socket controller

Socket controller는 PC Card 디바이스와 시스템의 버스(예를 들어서 PCI bus)간의 bridge 역할을 한다. Socket controller의 종류로는 여러가지가 있지만, 기본적인 기능면에서는 동일한 것을 공유하고 있다. Socket Service layer에 속하는 소프트웨어 layer는 host controller를 어떻게 프로그램할 것인가에 대한 모든 자세한 특성들에 대한 처리를 맡고 있다. 즉, socket을 통해서 접속하는 디바이스들은 socket 측의 host controller를 통해서 접근되며, 이 접근을 맡아서 처리하는 것이 Socket Service인 것이다.

Socket controller가 하는 일은 host의 메모리 및 I/O 공간에 있는 주소 window를 card가 가진 주소 window로 mapping하는 것이다. 모든 지원되는 controller는 적어도 4개의 독립적인 메모리 window와 2개의 I/O window를 socket 하나당 지원한다. 각각의 메모리 window는 host 주소 공간에 있는 기본 주소(base address)와 card에 있는 기본 주소 및 window의 크기로 정의된다. 어떤 controller들은 메모리 window를 자신이 가진 정렬 방법(alignment rule)에 맞게 위치하도록 하고 있지만, 모든 controller들은 window 크기가 적어도 4Kbytes를 차지하며, 2의 승수(power)로 정해지고, window 크기의 배수로 기본 주소를 정하고 있다. 또한, 각각의 window는 attribute나 common 메모리를 pointing하도록 프로그램될 수 있다.

I/O window는 host address가 I/O window내에 있을 경우, 이 주소가 I/O card에 넘겨지기 전에는 변동(modify)되지 않는다는 점에서 메모리 window와는 차이가 난다. 즉, 메모리 window는 card에 이 부분이 전달되어 사용된다고 이야기 되기 전에 내용의 변경(modify)이 가능하다는 말이다. 실제로는 host나 card의 주소 공간에 있는 window의 기본 주소는 동일하다. I/O window는 또한 정렬(alignment)이나 크기에 있어서의 제한이 없기에, 64Kbytes의 I/O 주소 공간의 어떤 byte 경계(boundary)에서도 시작하거나 끝날 수 있다.

PC Card 버스는 card로부터 controller로 전달되어 들어오는 하나의 interrupt를 정의하고 있다. 따라서, controller는 이렇게 전달되어 들어온 interrupt를 적절한 interrupt request line으로 돌리는(steering) 일을 하게되며, 실제로 모든 controller들은 card의 I/O interrupt를 어떤 사용되지 않는 interrupt line으로도 돌릴 수 있다. 이렇게 interrupt를 interrupt request line으로 돌리는 것은 controller에서 일어나기에, card 자체는 어떤 interrupt를 자신이 사용하는지 알 필요가 없다.

모든 PC Card controller는 card의 상태 변화에 따라서 interrupt를 발생시킬 수 있다. 이러한 interrupt들은 I/O card가 생성하는 I/O interrupt와는 성질이 다르며(distinct), 분리된 interrupt line을 사용한다. 인터럽트를 발생 시킬 수 있는 신호(signal)로는 card의 감지(detect), ready/busy, write protect, battery low 및 battery dead 등이 있다.

이젠 각각의 구현을 좀더 깊이 들어가서 보기로 하자. 먼저 Client Service 인터페이스를 보고, 다음으로 Driver Service 인터페이스를 보자. 기본 구현에 대한 것을 살펴보고나서 실제적인 PCMCIA Client Driver의 모듈을 선택해서 분석하도록 할 것이다.

12.6. Card Services에 대한 인터페이스

Linux가 PCMCIA 디바이스 드라이버를 위해서 정의한 기본적인 자료구조 들은 ~/include/pcmcia 및 ~/include/linux²⁸⁶에 있다. 이곳에서 있는 자료구조 중에서 중심적으로 설명하고자 하는 부분은 Card Service API와 관련되어 어떤 것들이 있는지를 보는 것과, 하위의 Card Service 및 Socket Service가 어떻게 동작되는지를 알아보는 것이다.

가장 기본적인 Card Service API는 CardServices() 함수이다. 아래와 같은 정의를 가진다. 정의는 ~/include/pcmcia/cs.h에 있으며, 구현은 ~/drivers/pcmcia/cs.c에 있다.

```
/* ~/include/pcmcia/cs.h에서 */
...
```

²⁸⁶ 이미 Linux Kernel Version 2.4.30이 나와있는 상황이므로 PCMCIA 부분에서는 Linux Kernel Version 2.4.3을 기준으로 할 것이다.

```
#ifdef IN_CARD_SERVICES
extern int CardServices(int func, void *a1, void *a2, void *a3);
#else
extern int CardServices(int func, ...);
#endif

...
/* ~/drivers/pcmcia/cs.c에서 */
int CardServices(int func, void *a1, void *a2, void *a3)
{
#ifdef PCMCIA_DEBUG
    if (pc_debug > 2) {
        int i;
        for (i = 0; i < SERVICE_COUNT; i++)
            if (service_table[i].key == func) break;
        if (i < SERVICE_COUNT)
            printk(KERN_DEBUG "cs: CardServices(%s, 0x%p, 0x%p)\n",
                   service_table[i].msg, a1, a2);
        else
            printk(KERN_DEBUG "cs: CardServices(Unknown func %d, "
                   "0x%p, 0x%p)\n", func, a1, a2);
    }
#endif
    switch (func) {
    ...
    case RegisterClient:
        return register_client(a1, a2); break;
    ...
    default:
        return CS_UNSUPPORTED_FUNCTION; break;
    }
} /* CardServices */
```

코드 688. CardServices() 함수

CardServices() 함수는 일반적으로 다음과 같이 정의될 수 있다. 즉, func 파라미터에는 CS가 제공하는 Card Service 함수의 번호가, a1과 a2, a3에는 Card Service API 함수에서 사용할 argument가 들어간다. 각 부분에 argument로 들어가는 내용중에서 func부분에 사용하는 것은 나중에 다시 보게 될 것이다. 예를 들어서, 만약 CardServices() 함수를 사용해서 하나의 Client Device Driver가 Callback 핸들러 함수를 등록하기를 원한다면, 아래와 같이 정의된 형태로 CardServices() 함수를 호출할 것이다.

```
#include "cs_types.h"
#include "cs.h"
int CardServices(RegisterClient, client_handle_t *client, client_reg_t *reg);
```

예제 4. Client의 등록

즉, CardServices() 함수는 func에 해당하는 값으로 switch()문을 통해서 분기할 것이며, 각각의 case 문에서 해당하는 함수를 불러줄 것이다. 따라서, RegisterClient 함수에 해당하는 pcmcia_register_client(a1, a2)가 호출될 것이다.

CardServices() 함수를 통해서 제공되는 func은 아래와 같은 것들이 있다. 이것은 ~/include/pcmcia/cs.h에 아래와 같이 정의되어 있다.

```
/*
 * The main Card Services entry point
 */
enum service {
    AccessConfigurationRegister, AddSocketServices,
```

```

AdjustResourceInfo, CheckEraseQueue, CloseMemory, CopyMemory,
DeregisterClient, DeregisterEraseQueue, GetCardServicesInfo,
GetClientInfo, GetConfigurationInfo, GetEventMask,
GetFirstClient, GetFirstPartition, GetFirstRegion, GetFirstTuple,
GetNextClient, GetNextPartition, GetNextRegion, GetNextTuple,
GetStatus, GetTupleData, MapLogSocket, MapLogWindow, MapMemPage,
MapPhySocket, MapPhyWindow, ModifyConfiguration, ModifyWindow,
OpenMemory, ParseTuple, ReadMemory, RegisterClient,
RegisterEraseQueue, RegisterMTD, RegisterTimer,
ReleaseConfiguration, ReleaseExclusive, ReleaseIO, ReleaseIRQ,
ReleaseSocketMask, ReleaseWindow, ReplaceSocketServices,
RequestConfiguration, RequestExclusive, RequestIO, RequestIRQ,
RequestSocketMask, RequestWindow, ResetCard, ReturnSSEntry,
SetEventMask, SetRegion, ValidateCIS, VendorSpecific,
WriteMemory, BindDevice, BindMTD, ReportError,
SuspendCard, ResumeCard, EjectCard, InsertCard, ReplaceCIS,
GetFirstWindow, GetNextWindow, GetMemPage
};

}

```

코드 689. Main Card Service의 entry point

각각의 entry point가 하는 역할은 다시 아래와 같이 간단히 정의할 수 있다. 여기서 중요한 점은 모든 Client Device Driver가 여기서 나열된 Card Service를 다 사용하지는 않는다는 점이다. 따라서, 구현하고자 하는 Client Device Driver에 따라서, 사용되는 Card Service의 내용도 달라질 것이다.

Service	Description
AccessConfigurationRegister	Config Register 영역(area)으로부터 특정 offset만큼 떨어진 곳에 있는 CIS configuration register에 한 byte를 읽거나 쓰는 일을 한다. 이것은 RequestConfiguration으로 설정(configure)된 socket에 대해서만 사용될 수 있다. Offset으로 사용될 수 있는 값은 cistpl.h에 정의되어 있다.
AddSocketServices	이미 Card Service가 사용중인 Socket Service 핸들러 들에 새로운 Socket Service 핸들러를 추가한다.
AdjustResourceInfo	Card Service에 PC Card 디바이스에 의해서 할당될 수 있거나 할당될 수 없는 자원(resource)들에는 어떤 것이 있는지를 알려주는데 사용된다. 이것은 일반적인 Linux의 자원관리 시스템이외에 사용자 수준의 새로운 제어레벨(level)을 제공한다.
CheckEraseQueue	Card Service에 새로운 erase 요청이 RegisterEraseQueue 호출로 이미 등록된 큐에 있다는 것을 알려준다. 일반적으로 client는 초기에 각각의 erase 큐의 엔트리(entry)들에 ERASE_IDLE이란 상태 값을 할당(assign)하고, 다시 새로운 요청을 queue에 더하게 되면, client는 ERASE_QUEUED로 상태로 바꾸고나서, CheckEraseQueue를 호출하게 된다. 만약 client가 erase 완료(completion) event를 받게(notify) 되면, 요청이 성공했는지를 확인하기 위해서 state 필드를 확인(check)하게 된다.
CloseMemory	OpenMemory에서 넘겨받은 메모리 핸들을 해제(release)하는데 사용한다. Client는 반드시 DeregisterClient Card Service를 호출하기 전에 모든 메모리 핸들을 해제해야 한다.
CopyMemory	특정(specified)한 논리적인 socket에 있는 PC Card로부터 데이터를 읽고, 이 데이터를 같은(region) 영역의 다른 곳으로 쓴다(write).
DeregisterClient	Client가 모든 자신이 할당한 resource를 해제(release)한 후에 호출하며, client와 Card Service와의 연결을 끊어준다.
DeregisterEraseQueue	RegisterEraseQueue 호출로 등록된 큐를 해제(free)한다. 만약 pending인 요청이 명시된 큐에 대해서 있다면, 호출은 실패할 것이다.
GetCardServicesInfo	현재 Card Service에 대한 갱신(revision) 정보를 돌려준다.

GetClientInfo	Client를 기술하는 정보를 돌려준다.
GetConfigurationInfo	RequestIO나 RequestIRQ, RequestConfiguraton등에 의해서 이미 설정된 현재 사용중인 socket의 설정사항을 돌려준다.
GetEventMask	특정한 Client에 대한 event mask를 돌려준다.
GetFirstClient	Card Service에 등록된 client의 첫번째 client 핸들을 돌려준다.
GetFirstPartition	PC Card의 CIS에 기초한, 특정 socket에 있는 PC Card상의 첫번째 partition에 대한 디바이스 정보를 돌려준다.
GetFirstRegion	GetNextRegion과 함께 card의 CISTPL_DEVICE, CISTPL_JEDEC, CISTPL_DEVICE_GEO tuple들에 있는 정보를 요약(summarize) 정리한다.
GetFirstTuple	Card의 CIS를 검사해서 주어진 tuple에 해당하는 첫번째 tuple을 돌려준다.
GetNextClient	Card Service에 등록된 다음번의 client 핸들을 돌려준다.
GetNextPartition	PC Card의 CIS에 기초한, 특정 socket에 있는 PC Card상의 다음 partition에 대한 디바이스 정보를 돌려준다.
GetNextRegion	GetFirstRegion과 함께 card의 CISTPL_DEVICE, CISTPL_JEDEC, CISTPL_DEVICE_GEO tuple들에 있는 정보를 요약(summarize) 정리한다
GetNextTuple	GetFirstTuple과 같으며, 단지 일치하는 다음번의 tuple을 돌려준다.
GetStatus	현재 client의 소켓 상태를 돌려준다. I/O 모드로 설정된 Card들에 대해서 GetStatus는 Pin Replacement Register와 Extended Status Register를 가지고, card의 상태를 결정하는데 사용한다.
GetTupleData	명시된 tuple로부터 연속된 data를 끄집어온다. Tuple은 반드시 GetFirstTuple이나 GetNextTuple 호출에서 넘겨받은 것이어야 한다.
MapLogSocket	Card Service의 logical socket을 그것이 가지고 있는 Socket Service의 physical adapter와 socket 번호로 mapping한다.
MapLogWindow	Handle argument값으로 넘겨받은 Card Service의 Window 핸들을 그것이 가진 Socket Service의 physical adapter와 window로 mapping한다.
MapMemPage	Card의 메모리 주소를 결정하는데 사용되며, card 메모리는 메모리 윈도우의 기본(base)에 mapping된다. 윈도우는 이미 RequestWindow 호출을 사용해서 생성되어 있어야 한다.
MapPhySocket	Socket Service의 physical adapter와 그것이 가진 socket 값을 Card Service의 logical socket으로 mapping한다.
MapPhyWindow	Socket Service의 physical adapter와 window 값을 Card Service의 logical Window 핸들로 mapping한다.
ModifyConfiguration	Socket의 특성(attribute)들을 변경하는데 사용된다. Socket은 이미 RequestConfiguration 호출에서 설정되었다고 가정한다.
ModifyWindow	이전의 RequestWindow호출에서 돌려진 window 핸들의 특성(attribute)을 변경한다.
OpenMemory	다른 대량(bulk) 메모리 서비스를 통해서 메모리 영역(region)에 대한 접근을 하기위해서 사용되는 핸들을 얻는데(obtain) 사용한다.
ParseTuple	GetTupleData 호출에서 넘겨받은 tuple data를 해석(interpret)한다. cistpl.h파일에 돌려주는 구조체의 정의가 나와 있다.
ReadMemory	WriteMemory와 함께, 명시된(specified) 메모리 핸들에 의해서 정의되는 card 메모리 영역(area)에 대한 read/write를 한다. 이곳에서 사용되는 메모리 핸들은 OpenMemory에서 넘겨받은 것이다.
RegisterClient	Client의 핸들(handle)을 다른 서비스를 사용하기 전에 구하기 위해서 사용한다. 즉, Client Driver와 Card Service간의 링크(link)를 만들어주며, Client Driver를 적당한 Socket에 연결시켜준다.
RegisterEraseQueue	Card Service에 지우기(erase) 요청의 큐를 등록하는데 사용한다. 만약 client가 CheckEraseQueue를 호출하면, Card Service는 큐를 검색(scan)하고, 새로운 요청에 대해서 비동기적인 처리를 시작할 것이다.
RegisterMTD	Card Service에 client MTD가 특정(specified) 메모리 영역(region)에 대한 요청을

	처리(handle)할 것이라는 것을 알려준다.
RegisterTimer	Card Service에 callback 구조체를 등록한다. 주어진 tick 카운터에 기초해서 Card Service는 time 기간이 만료되고 Card Service 인터페이스가 사용가능하면, client를 다시 호출하게 될 것이다.
ReleaseConfiguration	RequestConfiguration 호출에서 이미 설정된 socket을 un-configure한다.
ReleaseExclusive	Client를 위한 Socket에 있는 PC Card의 배타적인 사용을 해제한다.
ReleaseIO	앞에서 호출한 RequestIO에 사용된 I/O port를 un-reserve한다.
ReleaseIRQ	앞에서 호출한 RequestIRQ에 사용된 interrupt를 un-reserve한다.
ReleaseSocketMask	현재 사용중인 socket의 상태 변화에 대해서 client가 더 이상 알기를 원하지 않는다는 것을 요청한다.
ReleaseWindow	RequestWindow에서 이미 할당된 메모리 window를 해제한다.
ReplaceSocketServices	Card Service가 사용중인 현재 존재하는 Socket Service의 핸들러를 새로운 Socket Service 핸들러로 바꾸는 것을 허락한다.
RequestConfiguration	실제적으로 socket을 설정(configuration)하는 역할을 수행한다. 이것은 전압(voltage)의 설정과 CIS configuration register들의 설정, I/O port window들 및 interrupt들의 설정을 포함한다.
RequestExclusive	Client를 위해서 socket에 있는 PC Card의 배타적인(exclusive) 사용을 요청한다.
RequestIO	Card를 위해서 I/O port를 예약(reserve)한다.
RequestIRQ	Card에서 사용할 목적으로 interrupt line을 예약한다.
RequestSocketMask	적용되는 Socket에 대한 상태 변화를 client가 알기를 원한다는 것을 나타낸다.
RequestWindow	Card 메모리의 window를 시스템의 메모리에 mapping시킨다.
ResetCard	Client의 socket을 reset시킨다. 이 호출(call)이 이루어지면, Card Service는 모든 client에게 CS_EVENT_RESET_REQUEST event를 보내게 되며, 만약 어떤 client가 이 event를 거절하면, Card Service는 이 호출을 시작한 client에게 CS_EVENT_RESET_COMPLETE event를 event_callback_args.info 필들에 요청을 거절한 client의 복귀 코드를 설정해서 보내게 된다. 만약 모든 client들이 이 요청에 따른다면, Card Service는 CS_EVENT_RESET_PHYSICAL event를 보내게 되며, socket을 reset한다. Socket이 ready되었다고 알려오면, CS_EVENT_CARD_RESET event가 생성된다. 마지막으로 CS_EVENT_RESET_COMPLETE event가 호출을 시작한 client로 전달되며, event_callback_args.info에는 0(zero)이 설정된다.
ReturnSSEntry	Socket Service를 호출하는데 사용될 수 있는 entry point에 대한 pointer를 돌려준다.
SetEventMask	어떤 event에 대해서 현재 client가 통고(notify) 받기를 원하는지 결정하는 mask를 갱신(update)한다.
SetRegion	Client가 PC Card region의 특성(characteristic)을 설정하는 것을 허락한다.
ValidateCIS	Card가 올바른 CIS(Card Information Structure)를 가지고 있는지 확인한다. 이것은 Chains라고 불리는 곳에서 찾은 tuple의 수를 돌려주게 되며, 만약 CIS가 해석(interpret)할 수 없는 형태라면, Chains에는 0이 설정될 것이다.
VendorSpecific	Vendor Specific한 Card Service 요청을 만들기 위해서 사용한다.
WriteMemory	ReadMemory와 함께, 명시된(specified) 메모리 핸들에 의해서 정의되는 card 메모리 영역(area)에 대한 read/write를 한다. 이곳에서 사용되는 메모리 핸들은 OpenMemory에서 넘겨받은 것이다.
BindDevice	디바이스 드라이버와 특정 socket을 관련 짓는다. 이것은 일반적으로 Driver Service들로부터 호출되며, 새로이 삽입된 card를 찾고 난 후에 있게 된다. 드라이버가 일단 socket에 연결(bind)되면, 그 소켓의 client로 등록 될 수 있다. 주의할 점은 이 호출은 argument로 client의 handle을 받지 않는다는 점이다. 또한 socket의 번호를 argument로 받는 유일한 Card Service이다.

BindMTD	MTD(Memory Technology Device)를 메모리 영역(region)과 관련(associate) 시킨다.
ReportError	주어진 Card Service 함수의 코드와 그것의 복귀 코드를 가지고, 커널의 에러 메시지를 만드는데 사용한다. 만약 argument로 사용되는 client의 핸들이 유효(valid)하다면, 에러는 client의 드라이버 이름이 앞에 붙을 것이다.
SuspendCard	Card Service는 모든 client들에 CS_EVENT_PM_SUSPEND event를 보내게 된다. 그리고 나서, socket을 shutdown시키고 power를 끈다.
ResumeCard	Socket의 power를 회복 restore한 후, Card Service는 모든 client에 CS_EVENT_PM_RESUME event를 보낸다.
EjectCard	Card Service는 모든 client에게 eject event를 보낸다. 그리고 나서 socket을 shutdown하고 power를 끈다. Driver Service를 제외한 모든 client는 소켓과의 link를 해제할 것이다.
InsertCard	Card Service는 현재 사용중인 socket의 모든 client들에 insertion event를 보낸다. 일반적으로 client는 단지 Driver Service만이 될 것이다.
ReplaceCIS	Client가 card상에 있는 CIS를 대치할 수 있는 것을 Card Service에 보낼 수 있도록 한다. 이것은 완전하지 못하거나 정확하지 않는 CIS 정보를 가진 card를 위한 응용프로그램에서 사용될 수 있으며, 만약 정확한 CIS가 card에 대해서 유용한 다른 정보에서 유추할 수 있다면, 이러한 정보들이 client에 제공될 수 있도록 한다. 다른 방법으로는 client의 쓰스 코드에 CIS 오류를 가진 card를 위해서 정정된 것을 담아두는 방법이 있을 수 있다. 바꿔진 CIS는 card가 eject되기 전까지는 남아있게 되며, 모든 tuple과 관련된 service들은 card의 실제 CIS보다는 대체된 CIS를 사용할 것이다.
GetFirstWindow	Socket의 모든 메모리 window들에 대해서 window 설정(configuration) 정보를 가져온다. GetFirstWindow는 client의 window 핸들을 메모리 윈도우 핸들로 대치하게 되며, 메모리 윈도우 핸들은 다시 GetNextWindow 호출에서 갱신(update)될 것이다.
GetNextWindow	GetFirstWindow를 참조하라. 즉, client의 window 핸들을 대치하게 되는 메모리 윈도우 핸들을 갱신한다.
GetMemPage	현재 card의 메모리 윈도우에 대한 주소 mapping을 구한다(retrieve).

표 64. Card Service의 주요 Service의 정의

위의 표와 같은 많은 CardServices()의 함수들이 존재하지만, 이를 각각에 대해서 논의한다는 것은 많은 시간을 필요로 한다. 따라서, 나중에 실제적인 PC Card용 디바이스 드라이버를 살펴볼 때 관련된 함수로서 살펴보게 될 것이다. 여기선 이러한 Card Services의 함수들이 존재한다는 사실만 알면 될 것이다. CardServices()가 돌려주는 Error Code에는 아래와 같은 값이 같은 파일에 들어 있다.

#define CS_SUCCESS	0x00	/* 성공적인 호출이다. */
#define CS_BAD_ADAPTER	0x01	/* 잘못된 adapter 오류 */
#define CS_BAD_ATTRIBUTE	0x02	/* 잘못된 attribute 오류 */
#define CS_BAD_BASE	0x03	/* 잘못된 base address 오류 */
#define CS_BAD_EDC	0x04	/* 잘못된 error detection code 오류 */
#define CS_BAD_IRQ	0x06	/* 잘못된 interrupt request 오류 */
#define CS_BAD_OFFSET	0x07	/* 요청된 card의 offset이 메모리 영역을 벗어남 */
#define CS_BAD_PAGE	0x08	/* Page의 값이 0이 아님 */
#define CS_READ_FAILURE	0x09	/* Read 실패 */
#define CS_BAD_SIZE	0x0a	/* 요청된 전달의 크기가 메모리 영역을 벗어남 */
#define CS_BAD_SOCKET	0x0b	/* 명시된 socket의 번호가 유효하지 않음 */
#define CS_BAD_TYPE	0x0d	/* 인터페이스 type이 일치하지 않음 */
#define CS_BAD_VCC	0x0e	/* Vcc voltage를 지원하지 않음 */
#define CS_BAD_VPP	0x0f	/* 요청된 Vpp1/Vpp2를 지원하지 않음 */

```
#define CS_BAD_WINDOW 0x11      /* 요청된 window가 유효하지 않음 */
#define CS_WRITE_FAILURE 0x12      /* Write 실패 */
#define CS_NO_CARD 0x14          /* 존재하지 않는 card에 대한 연산을 수행했음 */
#define CS_UNSUPPORTED_FUNCTION 0x15      /* 지원하지 않는 기능임 */
#define CS_UNSUPPORTED_MODE 0x16      /* 지원하지 않는 모드임 */
#define CS_BAD_SPEED 0x17          /* 전송 속도 오류 */
#define CS_BUSY 0x18              /* Card Service가 사용중임 */
#define CS_GENERAL_FAILURE 0x19      /* 일반적인 오류 */
#define CS_WRITE_PROTECTED 0x1a      /* Write가 Protect되어 있음 */
#define CS_BAD_ARG_LENGTH 0x1b      /* Argument의 길이가 잘못됨 */
#define CS_BAD_ARGS 0x1c          /* 잘못된 argument를 사용함 */
#define CS_CONFIGURATION_LOCKED 0x1d      /* Configuration에 이미 lock이 설정되 있음 */
#define CS_IN_USE 0x1e              /* 시스템의 메모리에서 free window를 찾을 수 없음 */
#define CS_NO_MORE_ITEMS 0x1f      /* Socket에 대해서 더 이상 설정할 window가 없음 */
#define CS_OUT_OF_RESOURCE 0x20      /* 더 이상의 메모리 원도우를 할당할 수 없음 */
#define CS_BAD_HANDLE 0x21          /* 잘못된 client handle 임 */
#define CS_BAD_TUPLE 0x40          /* 잘못된 tuple임 */
```

코드 690. CardServices() 함수의 error code

위에서 정의한 CardService의 error code들은 특정 Card Service API에 대해서 subset으로 사용되며, 적절한 error 정보를 넘겨주도록 사용될 것이다.

12.7. Card Services의 Event Handling

Card Service에서 발생하는 event는 다음과 같은 분류로 나누어 볼 수 있다. 즉, 이것은 event를 발생시키는 source가 무엇인가에 따른 것이다.

- Card의 상태에 변화가 있었음을 하위 level의 socket driver가 알려옴.
- Card Service 자체에서 발생시킨 인위적인 event.
- APM(Advanced Power Management)와 관련된 event.
- 다른 Card Service client로 부터 발생한 event.

이중에서 하위의 socket driver가 발생시키는 event는 interrupt-driven이나 polling 방식을 취할 수 있다. 즉, asynchronous하게 발생하는 interrupt를 이용하거나, 주기적으로 check를 하는(polling) 방식을 취할 수 있다.

이와 같은 event가 발생했을 때, 이를 처리할 event handler는 다음과 같은 형식으로 정의된다. 여기서 정의한 event handler는 client에 의해서 Card Service에 callback 함수로 등록된다.

```
int (*event_handler)(event_t event, int priority, event_callback_args_t *args);
```

여기서, priority는 CS_EVENT_PRI_LOW를 가지거나, CS_EVENT_PRI_HIGH를 가질 수 있으며, 각각은 일반적인 우선 순위와 즉각적으로 반응해야하는 우선 순위를 나타낸다. 이중에서 높은 우선 순위를 가지는 event로는 나중에 나오겠지만, CS_EVENT_CARD_REMOVAL이 있다. 즉, client 드라이버는 Card Service가 다른 client들에게 빠리 이 event를 알릴 수 있도록 최대한 효율적으로 이 event를 처리하는 핸들러를 만들어 주어야 할 것이다. 여기서 사용하는 event_callback_args_t 구조체는 아래와 같이 정의된다.

```
typedef struct event_callback_args_t {
    client_handle_t client_handle; /* Event에 대해 책임이 있는, socket을 가지는 client의 핸들 */
    void *info;                  /* ResetCard에 대한 호출 결과를 가진다. */
    void *mtdrequest;            /* MTD(Memory Technology Device)에 대한 요청을 가진다. */
    void *buffer;                /* 데이터의 buffer 영역 */
}
```

```

void *misc;           /* 기타정보 */
void *client_data;   /* 디바이스와 관련된 local data 구조체를 가진다.*/
struct bus_operations *bus;    /* Socket에 대한 I/O primitive(명령)들의 entry point가 되는 테이블에 대한
포인터이다. 즉, 버스 연산자 들을 가진다.*/
} event_callback_args_t;

```

코드 691. event_callback_args_t 구조체의 정의

실제로는 이중에서 misc, buffer, mtdrequest는 아직 사용하고 있는 필드가 아니며, client_handler은 드라이버가 여러개의 socket에 bind된 경우에 유용하게 사용할 수 있다. 또한, socket이 직접적으로 host의 I/O나 메모리 공간에 card를 mapping하지 않는 경우에 bus 필드가 소켓에 대한 I/O 명령(혹은 연산)의 entry point를 가지도록 하고 있다.

Card Service에서 발생할 수 있는 event는 아래와 같은 것이 같은 파일에 정의되어 있다. 등록하려는 Client Driver는 이와 같은 event에 대해서 핸들러를 선언해서 Card Service가 해당 event가 발생할 때 호출하도록 만들 수 있다.

```

/* Events */
#define CS_EVENT_PRI_LOW          0
#define CS_EVENT_PRI_HIGH         1
#define CS_EVENT_WRITE_PROTECT    0x0000001
#define CS_EVENT_CARD_LOCK        0x0000002
#define CS_EVENT_CARD_INSERTION   0x0000004
#define CS_EVENT_CARD_REMOVAL     0x0000008
#define CS_EVENT_BATTERY_DEAD     0x0000010
#define CS_EVENT_BATTERY_LOW      0x0000020
#define CS_EVENT_READY_CHANGE     0x0000040
#define CS_EVENT_CARD_DETECT      0x0000080
#define CS_EVENT_RESET_REQUEST    0x000100
#define CS_EVENT_RESET_PHYSICAL   0x000200
#define CS_EVENT_CARD_RESET       0x000400
#define CS_EVENT_REGISTRATION_COMPLETE 0x000800
#define CS_EVENT_RESET_COMPLETE   0x001000
#define CS_EVENT_PM_SUSPEND       0x002000
#define CS_EVENT_PM_RESUME        0x004000
#define CS_EVENT_INSERTION_REQUEST 0x008000
#define CS_EVENT_EJECTION_REQUEST 0x010000
#define CS_EVENT_MTD_REQUEST      0x020000
#define CS_EVENT_ERASE_COMPLETE   0x040000
#define CS_EVENT_REQUEST_ATTENTION 0x080000
#define CS_EVENT_CB_DETECT        0x100000
#define CS_EVENT_3VCARD           0x200000
#define CS_EVENT_XVCARD           0x400000

```

코드 692. Client가 받는 Event에 대한 정의

이들 event들을 정리하면, 아래의 표와 같다. 즉, Card Service에서 발생할 수 있는 event들에 적당한 handler들을 설치해서 card의 event에 대해서 대응할 수 있도록 해준다.

Event	Value	Description
CS_EVENT_PRI_LOW	0	일반적인 우선 순위를 가지는 event를 가르킨다.
CS_EVENT_PRI_HIGH	1	즉각적인 반응을 보여야 하는 우선 순위를 가지는 event를 가르킨다.
CS_EVENT_WRITE_PROTECT	0x0000001	Write protect signal의 변화를 나타낸다.
CS_EVENT_CARD_LOCK	0x0000002	Card에 lock이 설정되었다.

CS_EVENT_CARD_INSERTION	0x000004	Card가 insert될 때 발생하는 event이다. 만약 드라이버가 이미 점유된 socket에 대해서 bind가 되어 있다면, Card Service가 드라이버에 insert event를 인위적으로 보내줄 것이다.
CS_EVENT_CARD_REMOVAL	0x000008	Card가 제거되었다. 이 event는 최소의 delay를 가지고 처리되어야 한다. 따라서, Card Service는 client에 최대한 빨리 알려줄 것이다.
CS_EVENT_BATTERY_DEAD	0x000010	Battery dead를 알려준다.
CS_EVENT_BATTERY_LOW	0x000020	Battery low를 알려준다.
CS_EVENT_READY_CHANGE	0x000040	Ready 상태임을 알려준다.
CS_EVENT_CARD_DETECT	0x000080	Card가 검출(detect)되었음을 알려준다.
CS_EVENT_RESET_REQUEST	0x000100	Client가 ResetCard를 호출할 경우에 전달된다. Event 핸들러는 failure를 돌려줌으로써 거부 할 수 있다.
CS_EVENT_RESET_PHYSICAL	0x000200	Reset 신호가 card에 전달되기전에 모든 client로 전달된다.
CS_EVENT_CARD_RESET	0x000400	Reset 연산이 끝났음을 알려준다. GetStatus를 통해서 reset이 잘되었는지를 확인해야 한다.
CS_EVENT_REGISTRATION_COMPLETE	0x000800	RegisetClient를 성공적으로 호출한 후에 전달된다.
CS_EVENT_RESET_COMPLETE	0x001000	ResetCard를 호출한 client에 reset의 처리가 끝났음을 알리기 위해서 전달된다.
CS_EVENT_PM_SUSPEND	0x002000	Card Service가 사용자나 APM(Advanced Power Management)의 suspend 요청을 받았다는 것을 알린다. Event handler는 failure를 돌려줌으로써 거부할 수 있다.
CS_EVENT_PM_RESUME	0x004000	시스템의 제기동(resume)을 알려준다.
CS_EVENT_INSERTION_REQUEST	0x008000	Card의 삽입(insertion) 요청이 있음을 알려준다.
CS_EVENT_EJECTION_REQUEST	0x010000	Card의 배출(ejection) 요청이 있음을 알려준다.
CS_EVENT_MTD_REQUEST	0x020000	MTD(Memory Technology Device)의 메모리 연산을 시작하는데 사용하는 것으로, callback 함수의 argument중 mtdrequest 필드에 요청을 기술하게 되며, buffer 필드에 host의 buffer 주소가 전달된다.
CS_EVENT_ERASE_COMPLETE	0x040000	큐잉(queueing)된 erase 연산이 끝났음을 client에 알려준다. Callback 함수의 argument중 info 필드에 지워진(erase) queue의 entry가 전달된다.
CS_EVENT_REQUEST_ATTENTION	0x080000	주의를 요한다고 알려준다.
CS_EVENT_CB_DETECT	0x100000	Socket0이 CardBus 디바이스에 의해서 점유되었음을 알려준다.
CS_EVENT_3VCARD	0x200000	3.3 Volt 연산을 지원한다는 것을 알려준다.
CS_EVENT_XVCARD	0x400000	X.X Volt 연산을 지원한다는 것을 알려주는 것으로 실제 voltage는 현재 명세서(spec.)에 정의되어 있지 않다.

표 65. Client Event에 대한 설명

Client Driver는 반드시 CS_EVENT_CARD_INSERTION과 CS_EVENT_CARD_REMOVAL event에 대해서 socket을 설정하거나, 설정해제하는 식으로 반응을 해야한다. 또한 드라이버는 높은 우선 순위의 event(앞에서 이미 CS_EVENT_CARD_REMOVAL가 있음을 보았다.)가 들어올 경우에, 즉시 socket에 대한 I/O 연산을 중단(block)해야 할 것이다. 이것은 아마도 디바이스 구조체의 flag에 대한 설정을 수반할 것이며,

timer interrupt를 사용해서 모든 shutdown과 관련된 처리(processing)가 나중에 일어나도록 scheduling할 것이다.

만약 CS_EVENT_PM_RESET_REQUEST event를 받는다면, 드라이버는 반드시 I/O를 block시켜야 하며, lock된 socket 설정(configuration)을 해제해야 한다. 또한 CS_EVENT_CARD_RESET을 받는다면, 드라이버는 반드시 socket 설정을 복구 시켜야 하며, block된 I/O를 unblock 시켜야 할 것이다.

CS_EVENT_PM_SUSPEND event를 받는다면, 드라이버는 반드시 CS_EVENT_PM_RESET_REQUEST event를 받았을 때와 유사하게 처리를 해야기에, I/O는 반드시 block되고, socket의 설정은 해제되어야 할 것이다. CS_EVENT_PM_RESUME event를 받으면, 드라이버는 card가 재설정(reconfigure)될 준비가 되어 있다고 생각할 것이며, CS_EVENT_CARD_RESET event를 받았을 때와 유사하게 동작할 것이다.

따라서, 앞에서 열거한 client가 처리할 책임이 있는 event들은 CS_EVENT_PM_RESET_REQUEST, CS_EVENT_CARD_RESET, CS_EVENT_PM_SUSPEND, CS_EVENT_PM_RESUME이 있겠다.

12.8. Driver Service에 대한 인터페이스

Card Service 이외에 리눅스에서는 Driver Service라는 것을 더 가지고 있다. Driver Service는 Card Service의 client driver들과 같은 사용자 모드 utility간의 link를 제공하는 역할을 한다. 따라서, 이것은 Card Service의 “Super-Client”정도 된다고 생각할 수 있겠다. Driver Service는 BindDevice 함수를 사용해서 해당하는 card와 client driver간의 링크를 생성한다. 또한 다른 client들과는 달리, Driver Service는 항상 card가 삽입되고 제거되는 모든 socket에 대해서 영구적으로 bind된 상태로 존재한다.

Driver Service는 설치되어 socket에 붙을(attach) 준비가 된 모든 Client Driver들을 관리한다. Client Driver들은 하나의 물리적인 card를 관리하는데 필요한 모든 것을 가지는 디바이스 인스턴스(device instance) 생성하고, 지우기 위한 entry point를 가질 필요가 있다. 각각의 Client Driver들은 고유한 32 글자의 꼬리표(tag)로 구분되어지며, 이 꼬리표는 dev_info_t이라는 특별한 자료구조를 가진다. 이것은 ~/include/pcmcia/cs_types.h에 아래와 같이 정의되어 있다.

```
#ifndef DEV_NAME_LEN
#define DEV_NAME_LEN 32
#endif

typedef char dev_info_t[DEV_NAME_LEN];           /* 디바이스 이름을 가지는 배열 */
```

코드 693. dev_info_t의 정의

또한 디바이스 인스턴스는 dev_link_t 구조체로 표현되는데, 이것은 ~/include/pcmcia/ds.h에 아래와 같이 정의되어 있다.

```
typedef struct dev_link_t {
    dev_node_t      *dev;          /* PC Card client 드라이버에 대한 포인터 */
    u_int           state, open;   /* 디바이스의 상태와 open되었는지를 나타내는 flag값 */
    wait_queue_head_t pending;    /* 처리되지 않은(pending) 요청을 한 프로세스의 대기 큐 */
    struct timer_list release;    /* Release timer의 list */
    client_handle_t handle;      /* Client Driver의 핸들 */
    io_req_t        io;           /* I/O 요청 */
    irq_req_t       irq;          /* IRQ 요청 */
    config_req_t    conf;         /* 설정(configure) 요청 */
    window_handle_t win;          /* 메모리 윈도우 핸들 */
    void            *priv;         /* Private 데이터를 보관하는 영역 */
    struct dev_link_t *next;       /* 다음 연결로의 포인터 */
} dev_link_t;
```

코드 694. dev_link_t 구조체의 정의

여기서 보여주는 `dev_link_t` 구조체의 대부분의 필드에 대한 것은 CIS(Card Information Structure) tuple을 읽어야지 알 수 있는 것이며, 또한 이렇게 읽어들인 자료를 이용해서 Card Services를 호출해야 사용할 수 있게 된다. 또한 여려개의 tuple이 있으므로 이러한 tuple들을 읽어서 자신이 원하는 것인지를 확인할 필요도 있을 것이다. CIS에 대한 것은 나중에 다시 살펴 보도록 할 것이다. 또한 priv 필드는 디바이스 드라이버가 local data를 자신만을 위해서 보관할 때 사용할 수 있는 영역이다. 다른 디바이스 드라이버를 구현할 때와 마찬가지로 드라이버가 사용하기를 원하는 전역 데이터와 같은 것을 이곳에 저장할 수 있을 것이다. state 변수는 디바이스의 현재 상태를 나타내기 위한 것으로 사용되며, 현재 디바이스가 설정이 되었는지, 혹은 pending된 상태인지를 기억하기 위해서 쓰인다. 또한 pending된 경우를 대비해서 wait queue를 사용해서 프로세스들의 대기 상태로 있을 수 있도록 만들었으며, release가 곧바로 일어나지 않고 일정시간 delay를 주기 위한 timer list를 가진다. 이외의 필드들에 대한 정의는 client driver를 분석할 때 볼 기회가 있을 것이다.

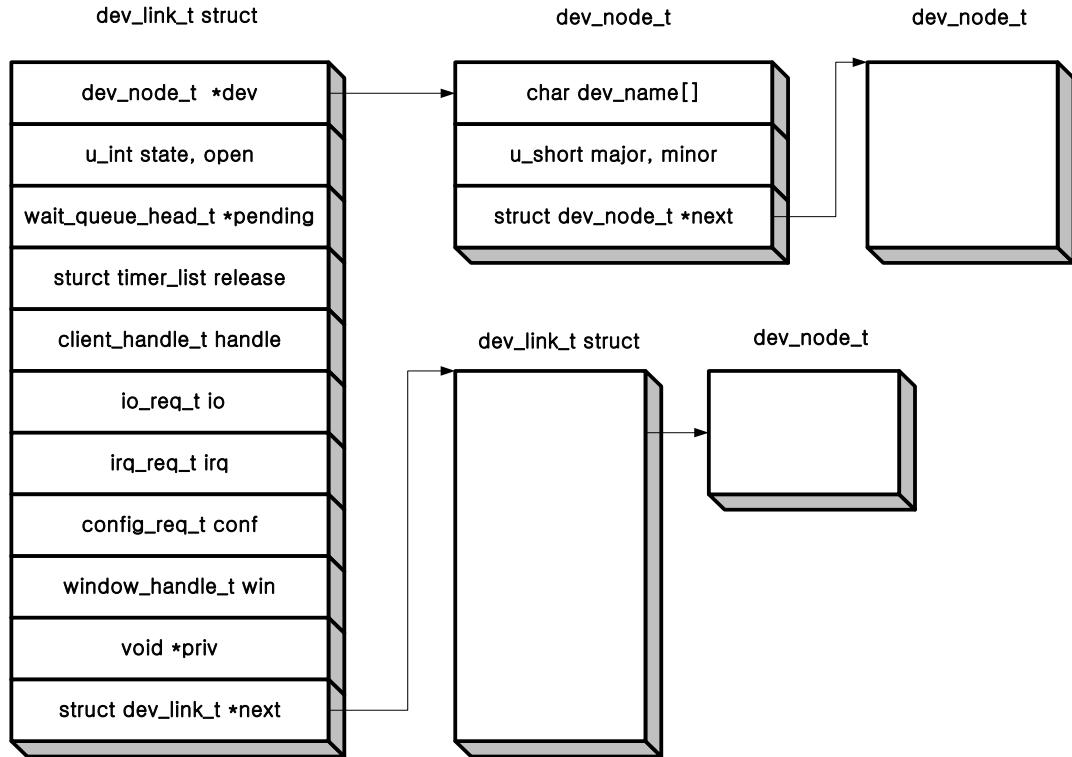
`dev_link_t` 구조체 필드 중에서 `dev_node_t`의 정의는 `~/include/pcmcia/driver_ops.h`에 아래와 같이 되어 있다.

```
#ifndef DEV_NAME_LEN
#define DEV_NAME_LEN 32
#endif
...
typedef struct dev_node_t {
    char          dev_name[DEV_NAME_LEN];/* 이 디바이스를 접근하기 위한 device 파일 이름 */
    u_short       major, minor;           /* 이 디바이스를 접근하기 위한 major와 minor 번호 */
    struct dev_node_t *next;             /* 다음에 연결된 dev_node_t에 대한 포인터 */
} dev_node_t;
```

코드 695. `dev_node_t` 구조체의 정의

`dev_node_t` 구조체에서 `dev_name` 필드는 디바이스 파일의 이름을 가지는데, 예를 들어서, `serial_cs` 드라이버²⁸⁷ 같은 경우에는 “`ttyS`”에 속자가 따로오는 디바이스 파일 이름을 사용한다. [그림 92]는 `dev_link_t`과 `dev_node_t` 간의 관계를 보여준다.

²⁸⁷ 이 디바이스 드라이버는 Linux 커널의 source에는 들어 있지 않으며, sourceforge.net과 같은 곳에서 받은 `pcmcia-cs-X.X.X.tar.gz`에 들어가 있다. 이후에는 이 파일에 들어있는 pcmcia 디바이스 드라이버들이 예로서 사용될 것이기에 download 받아서 보도록 하자.

그림 92. `dev_link_t` 구조체와 `dev_node_t` 구조체의 관계

하나의 `dev_link_t`에 여러 개의 `dev_node_t`이 연결될 수 있으며, 각각의 `dev_node_t`은 자신과의 인터페이스를 위해서 필요한 디바이스 이름과 major, minor 번호를 가지고 있음을 볼 수 있다. Major, minor 번호는 Unix 계열에서 하나의 디바이스를 접근하기 위해서 필요한 모든 정보가 됨을 익히 알고 있을 것이다. Driver Service는 앞에서 정의한 `dev_link_t` 구조체의 정보를 `DS_GET_DEVICE_INFO` ioctl을 이용해서 사용자 모드의 프로그램에 전달해 줄 수 있다. `dev_link_t` 구조체의 `state` 필드는 현재 디바이스의 상태를 알기 위해서 사용하는 것으로 다음과 같은 값이 정의되어 있다.

- `DEV_PRESENT` : 현재 card가 존재한다는 것을 나타낸다. 이 bit은 반드시 드라이버의 event 핸들러에 의해서 card의 삽입과 삭제시에 set되거나 clear되어야 할 것이다.
- `DEV_CONFIG` : card가 사용되기 위해서 설정(configure)되었음을 나타낸다.
- `DEV_CONFIG_PENDING` : 설정 연산이 진행중이다.
- `DEV_SUSPEND` : 현재 card가 suspend되어 있다.
- `DEV_BUSY` : I/O 연산이 진행중이라서 busy한 상태이다. 이 bit은 여러 프로세스가 동시에 접근하는 것을 방지하기 위해서 사용된다.
- `DEV_STALE_CONFIG` : 어떤 드라이버들에 대해서는 card가 배출(eject)된 상태에서, socket이 card에 해당하는 몇몇 디바이스들이 close되기 전까지는 설정 해제(unconfigure)가 되지 않아야만 한다. 따라서, 이 flag bit는 socket이 디바이스를 close할 때, 설정 해제가 되어야 함을 나타낸다. 즉, 설정 해제가 되어야 하는 드라이버의 socket과 설정 해제가 되지 않아야 하는 socket을 구분하기 위함이다.
- `DEV_STALE_LINK` : 드라이버의 instance는 자신의 모든 자원(resource)을 해제하기 전에 제거(delete)되지 않아야 할 것이다. 따라서, 이 flag bit는 socket이 설정 해제가 되자마자, driver instance가 해제(free)되어야 함을 나타낸다. 즉, 사용하고 있는 socket도 하나의 자원이 되므로, 이 socket의 제거가 있어야지만 driver instance를 제거할 수 있을 것이다.

`dev_link_t` 구조체의 `open` 필드는 이 디바이스의 사용 카운트(usage count) 역할을 하며, 디바이스는 이 `open`의 값이 0이 될 때만 해제될 것이다. `pending` 필드는 디바이스를 사용하기 위해서 대기(wait)하고 있는 process들의 queue를 관리하기 위한 것이며, `release` 필드는 card가 배출될 때 디바이스의 shutdown

처리를 스케줄링하는데 사용한다. Card의 제거는 event를 발생시키며, 높은 우선 순위를 가진다. 따라서, 드라이버의 event 핸들러는 디바이스의 state 필드에 있는 DEV_PRESENT를 재설정(reset)시키는 것으로 처리할 것이며, 나중에 shutdown 절차를 처리하기 위해서 scheduling할 것이다.

dev_link_t 구조체의 handle, io, irq, conf 및 win 필드는 평범한 PC Card I/O 디바이스를 설정하는데 필요한데 일반적인 자료 구조를 구성하는 부분이며, priv 필드는 디바이스를 관리하기 위해서 필요한 어떤 종류의 데이터 구조를 유지하는데도 사용될 수 있을 것이다. 즉, 이것은 file object와 같은 곳에서 사용하는 private 필드와 같은 역할을 한다고 보면 될 것이다. next 필드는 dev_link_t 구조체의 연결 리스트를 만들기 위해서 쓰이며, 이와 같은 경우에는 한 드라이버가 여러개의 디바이스 instance를 관리할 수 있을 것이다. 즉, 드라이버가 관리하는 디바이스 instance가 여러개가 존재해서 각각이 다른 minor 번호를 부여받아, 특정 operation에 대해서 해당 minor번호를 사용해서 접근하는 형태가 될 것이다.

```
typedef struct config_req_t {
    u_int      Attributes;           /* config_req_t의 attributes */
    u_int      Vcc, Vpp1, Vpp2;       /* Card의 voltage 설정 */
    u_int      IntType;             /* Interface type에 대한 설정 */
    u_int      ConfigBase;          /* Configuration Register의 Base Address */
    u_char     Status, Pin, Copy, ExtStatus; /* Configuration Register들 */
    u_char     ConfigIndex;         /* Configuration의 index 값 */
    u_int      Present;             /* 어떤 register들이 있는지를 알려주는 bitmap */
} config_req_t;
```

코드 696. config_req_t구조체의 정의

config_req_t 구조체는 RequestConfiguration과 같은 CardService에서 사용하는 구조체로서, socket과 같은 것을 설정하는데 사용된다. 여기서 잠시 card의 configuration과 관련된 register를 보도록 하자. 이하에서도 관련된 부분에서 많은 사용이 보일 것이다. 일단 다음과 같은 register들로 configuration register가 구성된다.

- Configuration Option Register : 모든 I/O function들에 대해서 반드시 존재해야 하며, 자원(resource)을 할당하는데 사용된다.
- Configuration and Status Register : 옵션이며, 다양한 function을 포함하는 것으로 status 변화 감지 및 알림(reporting), PCMCIA host expansion bus interface의 size, audio enable, power down control, interrupt pending status등과 같은 것을 포함한다.
- Pin Replacement Register : 옵션이며, HBA(Host Bus Adapter)의 메모리 interface를 위한 event change의 상태를 알려주기 위한 function들을 대체하기 위해서 사용한다.
- Socket and Copy Register : 옵션이며, I/O card가 시스템내에 하나나 그 이상의 동일한 card를 가질 수 있도록 하며, 동일한 I/O address 범위(range)에 반응하도록 하는데 사용된다.
- Extended Status Register : 옵션이며, STSCHG pin을 통해서 보고되는 event의 수를 확장(extend)하기 위해서 추가된 것으로서, software에서 이러한 event를 감지하고, 지우기 위해서 사용된다.
- I/O Base Address Register(s) : Multi-functional PC Card들에 대해서 반드시 있어야 하며, multi-functional 카드들이나 single function 카드들에서 I/O를 위한 기본(base) 주소를 알려주기 위해서 사용된다.
- I/O Limit Register : 옵션이며, I/O 기본 주소 register들에 관련된 것으로 기본 주소(base address)에 mapping될 수 있는 최대 I/O address의 범위를 나타낸다.

위와 같은 각각의 register들은 아래의 그림과 같이 그 위치 및 bit field가 정해진다. 각각의 레지스터는 전부 2byte boundary에 맞춰지도록 되어있다.

Offset	7	6	5	4	3	2	1	0
0								Configuration Option Register
	SRESET	LEVIREQ						Function Configuration Index
2								Configuration and Status Register

	Changed	SigChg	IOIS8	RFU	Audio	PwrDwn	Intr	IntrAck
4				Pin Replacement Register				
	CBVD1	CBVD2	CREADY	CWProt	RBVD1	RBVD2	PREADY	RWProt
6				Socket and Copy Register				
	RFU		Copy Number			Socket Number		
8				Extended Status Register				
	Event3	Event2	Event1	Req Attn	Enable3	Enable2	Enable1	Req Attn Enable
10				I/O Base 0				
12				I/O Base 1				
14				I/O Base 2				
16				I/O Base 3				
18				I/O Limit				

표 66. Configuration Register의 포맷

이러한 configuration register들은 CIS의 tuple에서 configuration과 관련된 tuple에서 attribute 메모리 공간에 있는 레지스터의 기본 주소를 찾을 수 있다. Tuple에 대한 것을 다룰때 이것을 볼 수 있을 것이다. 각각의 필드에 대한 정의도 나중에 다시 보도록 하고, 여기서는 이와 같은 register들이 있다는 것만 기억하기 바란다.

12.8.1. Client Driver를 Driver Service에 등록하기

Client Driver가 Driver Service에 등록하기 위해서는 앞에서 잠시본 register_pccard_driver() 함수를 호출한다. 이 함수를 통해서 Client Device Driver가 위에서 설명한 dev_link_t 및 dev_node_t과 같은 자료구조를 등록한다. register_pccard_driver()의 코드는 ~/drivers/pcmcia/ds.c에 아래와 같은 정의되어 있다.

```
int register_pccard_driver(dev_info_t *dev_info, dev_link_t *(*attach)(void), void (*detach)(dev_link_t *))
{
    driver_info_t *driver;
    socket_bind_t *b;
    int i;

    DEBUG(0, "ds: register_pccard_driver('%s')\n", (char *)dev_info);
    for (driver = root_driver; driver; driver = driver->next)
        if (strncmp((char *)dev_info, (char *)driver->dev_info,
                    DEV_NAME_LEN) == 0)
            break;
    if (!driver) {
        driver = kmalloc(sizeof(driver_info_t), GFP_KERNEL);
        if (!driver) return -ENOMEM;
        strncpy(driver->dev_info, (char *)dev_info, DEV_NAME_LEN);
        driver->use_count = 0;
        driver->status = init_status;
        driver->next = root_driver;
        root_driver = driver;
    }
    driver->attach = attach;
    driver->detach = detach;
    if (driver->use_count == 0) return 0;
    /* Instantiate any already-bound devices */
    for (i = 0; i < sockets; i++)
        for (b = socket_table[i].bind; b; b = b->next) {
            if (b->driver != driver) continue;
            b->instance = driver->attach();
            if (b->instance == NULL)
                printk(KERN_NOTICE "ds: unable to create instance "
```

```

        "of '%s'!\n", driver->dev_info);
}

return 0;
} /* register_pccard_driver */

```

코드 697. register_pccard_driver() 함수의 정의

register_pccard_driver() 함수는 dev_info_t과 dev_link_t 구조체에 대한 포인터를 돌려주는 attach() 함수의 포인터 및 detach() 함수의 포인터를 파라미터 값으로 넘겨받는다. root_driver 변수로부터 넘겨받는 dev_info과 같은 값을 가지는 것이 있는지를 차례로 확인해서 만약 없다면, 제일 마지막을 가르키는 NULL이 driver의 값으로 주어질 것이며, driver_info_t을 새로이 생성해서 이것을 driver가 가르키도록 만든 후, driver가 가지는 driver_info_t 구조체의 dev_info필드에 dev_info를 복사해 넣는다. use_count필드는 아직 사용되지 않기에 0을 가지며, status필드는 init_status를, next에는 root_driver를 가르키도록 만든다. 마지막으로 root_driver는 이 driver를 가르키도록 해준다.

driver가 가지는 driver_info_t의 attach와 detach는 넘겨받은 attach와 detach 함수의 주소로서 초기화 해주고, 만약 driver의 use_count필드가 0이라면 즉시 복귀한다. 이때의 복귀 코드는 0이다. use_count 필드가 0이 아니라고 한다면, 이것은 이미 사용중인 것을 말하므로 이미 bind된 디바이스 각각에 대해서 instance를 하나씩 생성한다.

for() loop가 하는 일이 위에서 설명한 instance들을 하나씩 생성하는 부분이다. 즉, 소켓의 개수만큼 socket_table[] 배열이 있을 것으로 각각의 배열에 대해서 socket_table[].bind를 살펴본다. 만약 bind된 driver가 다르다면 더 이상 진행할 필요가 없으며, 같다면 driver의 attach()함수를 호출한 값을 socket_table[].bind의 instance필드에 넣는다. 만약 같은 driver인데, socket_table[].bind의 instance필드가 NULL이라면, 즉, driver의 attach() 함수를 호출한 값이 NULL이라면, instance를 만드는데 실패한 것이다. 최종적인 복귀코드는 0이다.

정리하자면, register_pccard_driver() 함수는 Driver Service에 Client Driver가 있으며, socket과 bind를 준비가 되었다는 것을 알려주는 역할을 한다. 만약 Driver Service가 DS_BIND_REQUEST ioctl을 받았을 때, 이것이 Driver Service에 등록된 Client Driver의 dev_info와 같다면, Driver Service는 Client Driver의 attach() 함수를 불러줄 것이다. 만약 Driver Service가 DS_UNBIND_REQUEST를 받는다면, detach() 함수가 호출될 것이다. 즉, Client Driver에서는 이 각각의 attach()와 detach()함수에 대한 진입 점(entry point)을 만들어서 socket과의 bind에 대비할 수 있게 된다.

12.8.2. Client Driver를 Driver Service에서 제거하기

Client Driver는 Driver Service에서 자신의 등록을 해제하기 위해서 unregister_pccard_driver() 함수를 호출한다. unregister_pccard_driver()의 정의는 ~/drivers/pcmcia/ds.c에 아래와 같이 되어 있다.

```

int unregister_pccard_driver(dev_info_t *dev_info)
{
    driver_info_t *target, **d = &root_driver;
    socket_bind_t *b;
    int i;

    DEBUG(0, "ds: unregister_pccard_driver('%s')\n",
          (char *)dev_info);
    while ((*d) && (strncpy((*d)->dev_info, (char *)dev_info,
                           DEV_NAME_LEN) != 0))
        d = &(*d)->next;
    if (*d == NULL)
        return -ENODEV;

    target = *d;
    if (target->use_count == 0) {
        *d = target->next;
        kfree(target);
    }
}

```

```

} else {
    /* Blank out any left-over device instances */
    target->attach = NULL; target->detach = NULL;
    for (i = 0; i < sockets; i++)
        for (b = socket_table[i].bind; b; b = b->next)
            if (b->driver == target) b->instance = NULL;
}
return 0;
} /* unregister_pccard_driver */

```

코드 698. unregister_pccard_driver() 함수의 정의

unregister_pccard_driver() 함수가 넘겨받는 것은 앞의 register_pccard_driver() 함수를 호출할 때 사용했던 dev_info_t 구조체이다. 먼저 모든 driver_info_t 구조체를 관리하는 root_driver 변수에 대한 포인터를 얻어서 d로 둔다. 그리고 나서, while() loop를 돌면서 등록된 driver_info_t 구조체에서 현재 등록 해지를 원하는 것과 같은 dev_info를 가지는 것을 찾는다. 물론 이때 d가 가르키는 것은 NULL이 아니어야 하며, DEV_NAME_LEN만큼만 비교하면 된다.

만약 이렇게 찾는 연산의 결과, 같은 것이 없다면, -ENODEV를 돌려주게 되며, 있다면 target에 d를 넣는다. target의 use_count가 0이라면 현재 사용중인 것이 아니므로, target를 빼어내서 가지고 있는 메모리를 해제한다(kfree()). 만약 use_count필드가 0이 아니라면, 다른 곳에서 사용 중이므로 socket과 bind된 인스턴스가 존재할 수 있다. 따라서, 먼저 target의 attach와 detach 필드에 NULL을 주고, socket_table[].bind를 보고 같은 드라이버가 socket에 bind되어 있는지를 찾아서, instance에 NULL을 넣어준다. 즉, 아젠 모든 socket과의 bind가 제거된 것이다. 마지막으로 돌려주는 복귀코드는 0이다.

정리하자면, unregister_pccard_driver() 함수는 Driver Service에게 Client Driver가 더 이상 특정 socket과 bind될 필요가 없다는 것을 나타내주는 함수이다.

register_pccard_driver() 함수에서 다루지 않은 것은 중요한 것으로는 driver_info_t 구조체와 socket_bind_t 구조체 및 socket_table[] 배열이다. 이를 각각에 대해서 알아보도록 하자. 이것은 ~/drivers/pcmcia/ds.c에 아래와 같이 정의되어 있다.

```

typedef struct driver_info_t {
    dev_info_t           dev_info;          /* dev_info_t 구조체 */
    int                 use_count, status;   /* 사용카운터 및 상태 */
    dev_link_t          *(*attach)(void);    /* attach() 함수에 대한 포인터 */
    void                (*detach)(dev_link_t *); /* detach() 함수에 대한 포인터 */
    struct driver_info_t *next;             /* 다음 driver_info_t 구조체에 대한 포인터 */
} driver_info_t;

typedef struct socket_bind_t {
    driver_info_t *driver;           /* driver_info_t 구조체에 대한 포인터 */
    u_char            function;       /* socket의 function */
    dev_link_t        *instance;      /* dev_link_t를 가르키는 포인터로 instance생성에 쓰임 */
    struct socket_bind_t *next;      /* 다음 socket_bind_t 구조체에 대한 포인터 */
} socket_bind_t;

/* Device user information */
#define MAX_EVENTS     32
#define USER_MAGIC     0x7ea4
#define CHECK_USER(u) \
    (((u) == NULL) || ((u)->user_magic != USER_MAGIC))

typedef struct user_info_t {
    u_int              user_magic;        /* Client driver에 대한 magic number */
    int               event_head, event_tail; /* event_t 배열의 시작과 끝을 가르키는 index */
} user_info_t;

```

```

event_t          event[MAX_EVENTS]; /* event_t의 배열로서 발생한 event를 가진다.*/
struct user_info_t *next;           /* 다음 user_info_t 구조체에 대한 포인터 */
} user_info_t;

/* Socket state information */
typedef struct socket_info_t {
    client_handle_t handle;          /* Client Driver에 대한 handle */
    int state;                      /* 현재 socket의 상태 */
    user_info_t *user;              /* user_info_t에 대한 포인터 */
    int req_pending, req_result;    /* pending된 request의 수와 request의 결과를 가진다.*/
    wait_queue_head_t queue, request; /* 대기 큐와 request의 큐 */
    struct timer_list removal;      /* 제거 타임머 */
    socket_bind_t *bind;            /* 연결된 socket_bind_t에 대한 포인터 */
} socket_info_t;

```

코드 699. driver_info_t 와 socket_bind_t 및 socket_info_t 구조체의 정의

driver_info_t 구조체의 dev_info_t 구조체 필드인 dev_info는 단순히 문자열을 가지는 array이다. 이 필드는 설치된 driver의 이름을 가지도록 해주면 될 것이다. 실제로 예제에서 다루게될 디바이스 드라이버의 경우에는 “dummy_cs”라는 값으로 정의하고 있다. use_count는 현재 driver의 사용자 수가 얼마나되는지를 나타내는 값으로 설정되며, status는 현재로서는 커널내에 static link로 결합되어 있는지 아니면, module로서 동작하는지를 나타내기 위한 field로 드라이버가 등록될 때, init_status라는 변수의 값을 상속하도록 하고 있다. attach()와 detach() 함수는 각각 디바이스의 attach시와 detach시에 호출되는 함수의 포인터로서 attach에서 client의 각종 함수 및 알림을 받길 원하는 event의 등록이 일어날 수 있을 것이다. Detach는 앞에서 본 attach함수가 하는 일의 역을 행할 것이다. 만약 드라이버에 local 데이터 structure가 필요하다면, attach에서 할당하고, 다시 detach에서 해제해 주면 될 것이다.

socket_bind_t 구조체는 socket_info_t 구조체의 하위 필드를 구성하는 것으로서, socket이 어떤 드라이버와 연결되어 있는지를 알려주는 자료구조이다. driver_info_t 구조체의 포인터로 정의된 driver는 이미 앞에서 보았듯이, bind된 driver에 대한 정보를 알려주는 필드이다. u_char로 정의된 function 필드는 socket이 bind될 card function을 나타내는 것으로, multifunctional client driver인 경우에는 BIND_FN_ALL(0xFF)로 주어질 경우에는 card가 지원하는 모든 function들에 대해서 driver를 bind시켜준다. 실제로 드라이버는 자신이 bind된 function에 대한 CIS tuple 접근만이 허용된다. 관련된 필드로는 CISTPL_LONGLINK_MFC를 참조하도록 하자. dev_link_t 구조체에 대한 포인터로 정의된 instance 필드는 client driver가 생성하고 지워주는, 하나의 물리적인 디바이스를 관리하기 위한 모든 것을 가지는 것이다. 이것은 또한 driver_info_t 구조체로 정의된 driver 필드의 detach()와 같은 함수의 인자에서도 사용된다. socket_info_t 구조체는 PCMCIA card와 host간의 인터페이스 역할을 하는 하나의 socket에 대한 모든 정보를 가지는 구조체로서, handle 필드는 현재 이 socket과 bind된 client driver에 대한 handle이며, state필드는 socket의 상태를 나타내는 값으로 아래와 같은 값을 가질 수 있다.

심벌	값	설명
SOCKET_PRESENT	0x01	소켓이 존재한다.
SOCKET_BUSY	0x02	소켓이 현재 사용중이다.
SOCKET_REMOVAL_PENDING	0x10	소켓이 현재 removal 요청이 pending상태이다.

표 67. socket_info_t 구조체의 state 필드 값 정의

user_info_t 구조체의 포인터인 user는 이 socket을 사용중인 client 드라이버에 대한 magic number와, client driver가 받기를 원하는 event 값들로 이루어져 있는 자료구조에 대한 포인터이다. 따라서, socket에서 발생된 event가 현재 client 드라이버에서 설정한 event mask와 비교해서 관련된 event가 발생했다면, 이를 client driver에 알려주기 위한 목적으로 사용될 수 있다. req_pending과 req_result는 각각 pending된 요청과 요청의 결과 값을 가지는 필드이며, 대기 큐로 사용할 queue 및 request의 대기를 위한 request 필드가 존재한다. 또한 socket에서 client driver를 떼어내기 위한 것으로, 일정시간의 delay후에 이를 적용하기 위한 timer list인 removal 필드가 있다.

12.8.3. CardBus Client의 인터페이스

CardBus란 PC Card가 16-bit의 ISA에 기반을 두고 있는 반면, 32-bit의 PCI에 기초한 새로운 PCMCIA의 규격이다. 즉, 고속으로 대용량의 데이터를 주변기기와 전달하기 위한 확장이라고 생각하면 될 것이다. 대략 8에서 20Mbytes정도의 데이터를 초당 전송할 수 있는 PC Card가 CardBus로 확장되면서, 33MHz로 동작하는 PCI bus상에서 이론적으로 최대 132Mbytes까지 전송이 가능하게 되었기 때문이다.

일반적으로 CardBus card는 표준 PCI chip set을 사용해서 디자인 되었으며, client driver가 CardBus card를 쉽게 설정하고, 기존의 Linux 커널의 PCI driver와 source 레벨에서의 공유를 극대화 시키기 위해, “super client”라는 것을 제공하고 있다. 이것이 바로 “cb_enabler”라는 모듈이다. 이 cb_enabler 모듈은 16-bit PC Card에서 사용하는 Driver Services와 비슷한 개념으로 디바이스의 설정과 끝내기(shutdown) 및 power management를 다루기 위한 여러개의 함수를 제공한다. cb_enabler 모듈은 card의 설정과 Card Services의 event를 관리하게 되기에, 모든 CardBus와 관련된 코드들은 cb_enabler에 들어가고, 기타 부가적인 최소의 코드만이 PCI driver 자체에 들어가게 된다.²⁸⁸ 하지만, CardBus의 client가 반드시 cb_enabler를 사용할 필요는 없다. 만약 특별한 client driver가 cb_enabler 모듈에서 제공되는 CardBus의 설정보다도 더 직접적인 제어를 원한다면, 16-bit client driver와 마찬가지로 직접 Card Services와 함께 등록되어, Card Services에서 제공하는 호출들을 사용할 수 있다.

이러한 cb_enabler 모듈은 두개의 함수로 자신의 entry point를 기술하는데, 각각은 아래와 같이 정의된다. 등록과 해제에 관련된 함수들이다.

- int register_driver(struct driver_operation *ops);
- void unregister_driver(struct driver_operation *ops);

두 함수가 모두 공통적으로 사용하는 driver_operation 구조체는 다시 아래와 같으며, 정이는 ~/include/pcmcia/driver_ops.h에 있다.

```
typedef struct driver_operations {
    char          *name;           /* Driver operation의 이름 */
    dev_node_t     *(*attach)(dev_locator_t *loc); /* Attach() 함수에 대한 포인터 */
    void         (*suspend)(dev_node_t *dev); /* Suspend() 함수에 대한 포인터 */
    void         (*resume)(dev_node_t *dev); /* Resume() 함수에 대한 포인터 */
    void         (*detach)(dev_node_t *dev); /* Detach() 함수에 대한 포인터 */
} driver_operations;
```

코드 700. driver_operations 구조체의 정의

attach() 함수와 detach() 함수는 앞에서 이미 보았을 것이다. 나머지 suspend() 함수와 resume() 함수는 디바이스의 power management와 관련된 함수로, power saving모드로 전환하는 경우에 suspend()함수를 호출할 것이며, 다시 원래의 모드로 돌아오록 resume()이 호출될 것이다.

이 함수들이 사용하는 자료구조로는 dev_node_t 구조체와 dev_locator_t 구조체가 있다며, 다시 아래와 같은 정의를 가진다.

```
typedef struct dev_node_t {
    char          dev_name[DEV_NAME_LEN]; /* 제어하는 디바이스의 이름을 준다.*/
    u_short       major, minor;           /* 디바이스의 major와 minor 번호 */
    struct dev_node_t *next;             /* 다음 dev_node_t 구조체에 대한 포인터 */
} dev_node_t;

typedef struct dev_locator_t {
```

²⁸⁸ 실제로 pcmcia의 source코드를 보면, CardBus를 사용하고 있는 디바이스 드라이버들은 대부분의 코드는 원래의 것을 그대로 사용하고, 단순히 디바이스를 등록하고, 해제하는 경우와 power management부분에 한정적으로 cb_enabler를 이용하고 있다. 이렇게 함으로써, 추가적인 개발 노력을 단축할 수 있다는 점에서 cb_enabler와 같은 접근방식은 옳은 것 같다.

```

enum { LOC_ISA, LOC_PCI } bus;           /* 버스 type */
union {
    struct {
        u_short    io_base_1, io_base_2;      /* 기본 I/O address 1 과 2 */
        u_long     mem_base;                 /* Mapping된 시스템 메모리의 기본 주소 */
        u_char     irq, dma;                /* 사용하고 있는 IRQ 번호와 DMA channel 번호
    } isa;                                /* ISA device에 대한 구조체의 정의 */
    struct {
        u_char     bus;                   /* PCI Bus ID */
        u_char     devfn;                /* 디바이스의 function */
    } pci;                                /* PCI device에 대한 구조체의 정의 */
} b;
} dev_locator_t;

```

코드 701. dev_locator_t 구조체 및 dev_node_t 구조체의 정의

dev_locator_t 구조체는 attach() 함수에서 사용되며, 어디에서 자신이 사용할 디바이스를 찾을 것인가를 기술해 주는 자료구조이다. attach() 함수는 이와 같은 구조체를 이용해서 해당하는 dev_node_t entry를 돌려주거나, 혹은 NULL과 같은 값을 error로 돌려주어야 한다. 참고로, DEV_NAME_LEN은 디바이스가 가질 수 있는 최대 이름의 길이를 나타내는 값으로 32를 가진다.

12.8.3.1. CardBus Driver의 등록

```

int register_driver(struct driver_operations *ops)
{
    int i;

    DEBUG(0, "register_driver('%s')\n", ops->name);

    for (i = 0; i < MAX_DRIVER; i++)
        if (driver[i].ops == NULL) break;
    if (i == MAX_DRIVER)
        return -1;

    MOD_INC_USE_COUNT;
    driver[i].ops = ops;
    strcpy(driver[i].dev_info, ops->name);
    register_pccard_driver(&driver[i].dev_info, driver[i].attach,
                          &cb_detach);
    return 0;
}

```

코드 702. register_driver() 함수의 정의

CardBus driver의 실제적인 처리는 register_pccard_driver()에서 일어난다. 따라서, register_driver() 함수는 단순히 driver service에서 제공하는 API를 이용해서 새로운 client driver를 등록시켜주는 역할만 할 뿐이다. 일단 MAX_DRIVER(=4) 만큼의 갯수만큼 증가시키면서, driver[] array에 빈자리가 있는지를 확인한다. 여기서 만약 빈자리가 없다면(i==MAX_DRIVER) -1을 복귀 값으로 돌려준다.

```

typedef struct driver_info_t {
    dev_link_t          *(*attach)(void);
    dev_info_t          dev_info;
    driver_operations   *ops;
    dev_link_t          *dev_list;
} driver_info_t;

```

```

static dev_link_t *cb_attach(int n);
#define MK_ENTRY(fn, n) \
static dev_link_t *fn(void) { return cb_attach(n); }

#define MAX_DRIVER      4

MK_ENTRY(attach_0, 0);
MK_ENTRY(attach_1, 1);
MK_ENTRY(attach_2, 2);
MK_ENTRY(attach_3, 3);

static driver_info_t driver[4] = {
    { attach_0 }, { attach_1 }, { attach_2 }, { attach_3 }
};

```

코드 703. driver_info_t 구조체의 정의

driver[] 배열은 driver_info_t으로 정의된 전역 변수로서, 하나의 client driver당 하나의 entry를 가지는 array이다. 먼저 driver_info_t 구조체는 attach() 함수에 대한 pointer를 가지며, dev_info_t 구조체를 가지는 dev_info 필드와 driver의 연산에 대한 포인터를 가지는 ops 필드, 그리고 각 device의 instance들에 대한 연결 리스트에 대한 포인터를 가지는 dev_list 필드로 구성되어 있다. MK_ENTRY() 매크로는 각각의 드라이버에 대한 index를 argument로 넘겨주어 cb_attach() 함수를 호출하도록 만들어주고 있으며, 초기화시에 driver[] 배열은 4개의 entry에 대해서 default로 attach_X라는 함수의 포인터로 설정된다. Default로 cb_attach()를 호출하고, dev_link_t 구조체에 대한 pointer를 돌려준다.

이전 이 모듈의 사용 카운트를 하나 증가시켜서, 이 모듈이 사용되는 회수를 증가 시킨다(MOD_INC_USE_COUNT). 이전 이 함수가 넘겨받은 driver_operations 구조체의 포인터를 빙 driver[] 배열의 entry에 있는 ops 필드로 두고, dev_info에는 driver_operations에서 얻을 수 있는 디바이스의 이름을 copy하도록 한다. 마지막으로 위에서 얻은 정보를 기준으로 Driver Services에서 제공하는 API인 register_pccard_driver()를 호출한다. 넘겨주기 위한 argument로는 dev_info_t 구조체와 attach() 및 detach() 함수²⁸⁹이다. 여기서 한가지 주의할 점은 detach() 함수로서 cb_enabler에서 제공하는 cb_detach()를 넘겨준다는 점이다. 예러가 없었다면, 복귀 값은 0이다.

12.8.3.2. CardBus Driver의 해제

```

void unregister_driver(struct driver_operations *ops)
{
    int i;

    DEBUG(0, "unregister_driver('%s')\n", ops->name);
    for (i = 0; i < MAX_DRIVER; i++)
        if (driver[i].ops == ops) break;
    if (i < MAX_DRIVER) {
        unregister_pccard_driver(&driver[i].dev_info);
        driver[i].ops = NULL;
        MOD_DEC_USE_COUNT;
    }
}

```

²⁸⁹ cb_enabler가 제공하는 attach(), detach(), release(), config(), event() 등의 함수는 나중에 다시 분석해 보도록 할 것이다. 여기서는 단지 Driver Services를 사용하기 위해서, default로 이것들을 register_pccard_driver()의 인자(argument)로 넘겨준다는 사실만 기억하도록 하자. 결국, 이렇게 설정된 함수들이 먼저 호출될 것이며, 이 함수들 내에서 우리가 만들어준 driver_operations 구조체에 정의된 연산들이 사용될 수 있을 것이다.

코드 704. unregister_driver() 함수의 정의

unregister_driver() 함수는 앞에서 했던 연산을 역으로 수행한다. 먼저 넘겨받은 driver_operations 구조체와 같은 것이 driver[] 배열에 있는지를 확인한다. 없다면($i \geq MAX_DRIVER$) 에러이며, 수행은 그냥 끝나게된다. 같은 값이 존재한다면, Driver Services가 제공하는 unregister_pccard_driver() 함수를 호출해서 client driver에 대한 해제를 요청하고, driver[] 배열에서 차지하고 있던 부분의 driver_operations 구조체의 포인터 필드를 NULL로 주어서, 연결을 완전히 제거한다. 최종적으로 이 모듈의 사용 카운터를 감소시켜주면 함수는 복귀할 것이다(MOD_DEC_USE_COUNT).

12.8.3.3. CardBus Enabler의 Attach() 함수

CardBus enabler의 attach() 함수는 CardBus가 일종의 Driver Services에 대한 client driver로서의 역할을 하기에 client driver가 등록될 때 호출되는 함수이다. 이 함수에서는 새로운 client driver를 등록하는 역할을 수행한다.

```
struct dev_link_t *cb_attach(int n)
{
    client_reg_t client_reg;
    dev_link_t *link;
    int ret;

    MOD_INC_USE_COUNT;
    DEBUG(0, "cb_attach(%d)\n", n);

    link = kmalloc(sizeof(struct dev_link_t), GFP_KERNEL);
    if (!link) {
        MOD_DEC_USE_COUNT;
        return NULL;
    }

    memset(link, 0, sizeof(struct dev_link_t));
    link->conf.IntType = INT_CARDBUS;
    link->conf.Vcc = 33;

    /* Insert into instance chain for this driver */
    link->priv = &driver[n];
    link->next = driver[n].dev_list;
    driver[n].dev_list = link;
```

코드 705. cb_attach() 함수의 정의

먼저 등록할 client driver에 대한 자료구조를 만들어 주기 위해서 client_reg_t 구조체를 선언한다(client_reg). 또한 device의 instance 역할을 하게될 dev_link_t 구조체에 대한 포인터를 선언하는데, 이것은 나중에 커널 메모리에서 할당 받기 위한 것이다. 진행하기에 앞서, 이곳에서 다시 모듈의 사용 카운트를 하나 증가 시켜서(MOD_INC_USE_COUNT), 새롭게 추가될 client driver가 이 모듈을 사용한다는 것을 알려준다. link는 dev_link_t 구조체를 커널에서 할당받아 가르키도록 만든다. 만약 할당 받을 수 없다면, 모듈의 사용 카운트를 낮춰주고(MOD_DEC_USE_COUNT), NULL을 복귀 코드를 돌려준다. 할당 받은 메모리는 0으로 초기화시켜주고(memset()), 인터페이스 타입(IntType)으로 INT_CARDBUS(=0x04)를 준다. 초기 전압(voltage) 값으로는 33을 주어서 3.3 Volt임을 알려준다. 이젠 이렇게 생성된 instance를 driver[] array의 dev_list에 추가시켜주도록 한다.

```
/* Register with Card Services */
client_reg.dev_info = &driver[n].dev_info;
client_reg.Attributes = INFO_IO_CLIENT | INFO_CARD_SHARE;
client_reg.event_handler = &cb_event;
client_reg.EventMask = CS_EVENT_RESET_PHYSICAL |
```

```

CS_EVENT_RESET_REQUEST | CS_EVENT_CARD_RESET |
CS_EVENT_CARD_INSERTION | CS_EVENT_CARD_REMOVAL |
CS_EVENT_PM_SUSPEND | CS_EVENT_PM_RESUME;

client_reg.Version = 0x0210;
client_reg.event_callback_args.client_data = link;
ret = CardServices(RegisterClient, &link->handle, &client_reg);
if (ret != 0) {
    cs_error(link->handle, RegisterClient, ret);
    cb_detach(link);
    return NULL;
}
return link;
}

```

코드 706. cb_attach() 함수의 정의(계속)

이전 client driver를 등록시켜주는 일이 남았다. 먼저 client_reg_t 구조체의 필드를 채우도록 한다. 드라이버의 이름을 나타내는 dev_info 필드는 이미 초기화한 driver[] array의 dev_info 필드를 두도록 한다. Attributes에는 INFO_IO_CLIENT(=0x02) 및 INFO_CARD_SHARE(=0x10)²⁹⁰로 설정해서, 이것이 I/O client driver라는 것을 알려준다. 또한 event handler로는 cb_event() 함수를 등록하고, 이 등록하려는 함수가 보기일 원하는 event로는 CS_EVENT_RESET_PHYSICAL, CS_EVENT_RESET_REQUEST, CS_EVENT_CARD_RESET, CS_EVENT_CARD_INSERTION, CS_EVENT_CARD_REMOVAL, CS_EVNET_PM_SUSPEND, CS_EVENT_PM_RESUME과 같은 것이 있다는 것을 알려준다. 또한 등록하려는 client의 version은 Card Services의 version을 나타내지만, 현재로서는 의미가 없다. 마지막으로 client driver를 등록하기 위해서 RegisterClient CardServices를 호출한다. 만약 복귀(return) 값이 0이 아니라면, 오류가 있었다는 말이므로, cs_error() 함수를 호출해서 error를 reporting하고, cb_detach() 함수를 호출해서 앞에서 했던 일을 undo하게되며, 복귀 값으로 NULL을 돌려준다. 호출이 성공적이었다면, 복귀값은 새로 생성된 instance를 나타내는, dev_link_t 구조체를 가르키는 link가 될 것이다.

```

static void cs_error(client_handle_t handle, int func, int ret)
{
    error_info_t err = { func, ret };
    CardServices(ReportError, handle, &err);
}

```

코드 707. cs_error() 함수의 정의

앞에서 잠시 나온 cs_error() 함수는 단순히 error_info_t 구조체를 생성해서, ReportError Card Services를 호출하는 일만을 한다. error_info_t 구조체는 error를 일으킨 Card Services와 그것의 return code값을 필드로 가진다.

12.8.3.4. CardBus Enabler의 Detach() 함수

CardBus의 detach() 함수는 cb_attach() 함수가 했던 일을 되돌리는 일을한다. 이 함수는 card가 removal이 될 때 수행될 것이다. 따라서, client driver가 사용했던 커널의 자료구조들에 대한 제거가 주 역할이 될 것이다.

```

static void cb_detach(dev_link_t *link)
{
    driver_info_t *dev = link->priv;
    dev_link_t **linkp;
    bus_info_t *b = (void *)link->win;
}

```

²⁹⁰ 뒤에서 실제로 PC Card client driver를 분석할 때 자세히 보게될 것이다. 이곳에서는 각 필드가 가진 의미만을 보기로 하자. 두번째 있는 INFO_CARD_SHARE는 아무런 역할을 하지 않고, 다만 호환성 때문에 존재한다.

```

DEBUG(0, "cb_detach(0x%p)\n", link);

/* Locate device structure */
for (linkp = &dev->dev_list; *linkp; linkp = &(*linkp)->next)
    if (*linkp == link) break;
if (*linkp == NULL)
    return;

if (link->state & DEV_CONFIG)
    cb_release((u_long)link);

/* Don't drop Card Services connection if we are the bus owner */
if (b && (b->flags != 0) && (link == b->owner)) {
    link->state |= DEV_STALE_LINK;
    return;
}

if (link->handle)
    CardServices(DeregisterClient, link->handle);

*linkp = link->next;
kfree(link);
MOD_DEC_USE_COUNT;
}

```

코드 708. cb_detach() 함수의 정의

cb_attach() 함수에서 dev_link_t 구조체의 priv 필드에 해당 driver[] array의 entry에 대한 pointer를 저장했었다. 이것을 dev 변수로 가르키도록 한다. 또한 device instance를 찾기 위한 변수로 dev_link_t 구조체의 포인터를 가르키는 linkp를 선언하고, device instance가 사용하는 메모리 원도우에 대한 handle을 가지는 b(bus_info_t 구조체의 포인터)를 정의한다. DEBUG() 매크로는 단순히 어디까지 연산이 진행되었는가를 보여주기 위한 것이다. bus_info_t 구조체는 아래와 같은 정의를 가진다.

```

typedef struct bus_info_t {
    u_char             bus;           /* Bus ID */
    int               flags, ncfg, nuse; /* Request type, configuration, use count */
    dev_link_t        *owner;        /* 현재 bus를 소유하고 있는 device instance */
} bus_info_t;

```

코드 709. bus_info_t 구조체의 정의

위와 같이 정의된 bus_info_t 구조체는 cb_enabler에서는 자신만을 위해서 독특하게 사용하고 있다. cb_enable에서 사용하는 모든 bus에 대한 정보는 bus_table[] array가 가지며, 각각의 driver에 대해서 하나의 entry를 가진다. 또한 이것을 저장하기 위해서 device의 instance가 사용하는 win 필드를 이용한다. 이것은 Driver Service와 같은 곳에서 사용하는 것과는 다른 형태이다. 이것이 가능한 이유는 단순히 (*void)로 type casting해서 포인터로 사용하기 때문이다.²⁹¹

linkp를 driver[] array가 가지는 dev_list로 초기화시키고, 이것이 우리가 detach하려는 것과 같은 것인지를 찾는다. 만약 없다면 그냥 복귀하도록 한다. 현재 device instance의 상태(state)가 DEV_CONFIG라면, 설정을 마친 상태이므로 시스템의 resource를 점유하고 있기에, cb_release()를 먼저 호출하도록 한다. 넘겨주는 것은 device instance를 가지는 link 변수이다.

²⁹¹ 참고적으로 cb_enabler의 cb_config() 함수를 보도록 하자. 이곳에서는 window_handle_t 구조체 가르키는 포인터를 가지는 win필드를 어떻게 bus_info_t 구조체의 포인터로 사용하고 있는지를 찾을 수 있다.

만약 현재의 device instance가 bus에 대한 owner라면, client driver를 바로 unregister하지 않고, link가 잘못되었다는 것만을 나타낸 후(state |= DEV_STALE_LINK), 바로 복귀한다. 이것은 bus에 대한 operation이 끝난 후에 다시 해제하도록 하기 위한 것이다.

이전 시스템에 설정했던 자원들에 대한 해제는 이미 되었으므로, 등록된 client driver가 있는지를 확인한 후(link->handle), DereRegisterClient Card Services를 호출하도록 한다. 마지막으로 driver[] array에서 dev_list에 있는 device의 instance에 대한 link를 제거한 후, 할당했던 dev_link_t 구조체를 메모리에서 제거한다(kfree()). 이 모듈에 대해서 사용하고 있는 client driver가 하나 제거되었으므로, MOD_DEC_USE_COUNT를 호출해서 모듈 사용 카운트를 감소시켜준다.

12.8.3.5. CardBus Enabler의 Config() 함수

cb_config() 함수는 새로운 card가 socket에 insert될 때 호출되어, 이 카드에 대한 것을 설정(configuration)하기 위해서 호출되는 함수이다. 나중에 볼 event handler에서 이것을 호출하는 부분을 찾을 수 있을 것이다. 여기서 하는 주요한 일은 카드에 설정된 정보를 이용해서 시스템에서 사용할 자원을 예약하는 것이다.

```
static void cb_config(dev_link_t *link)
{
    client_handle_t handle = link->handle;
    driver_info_t *drv = link->priv;
    dev_locator_t loc;
    bus_info_t *b;
    config_info_t config;
    u_char bus, devfn;
    int i;

    DEBUG(0, "cb_config(0x%p)\n", link);
    link->state |= DEV_CONFIG;

    /* Get PCI bus info */
    CardServices(GetConfigurationInfo, handle, &config);
    bus = config.Option; devfn = config.Function;

    /* Is this a new bus? */
    for (i = 0; i < MAX_DRIVER; i++)
        if (bus == bus_table[i].bus) break;
}
```

코드 710. cb_config() 함수의 정의

cb_config() 함수가 넘겨받는 argument로는 device instance를 가르키는 link이다. 여기서, client driver에 대한 handle를 얻어서 handle 변수를 초기화한다. link의 priv 필드에 저장된 driver_info_t 구조체는 dev로 두고, 디바이스를 찾기 위한 dev_locator_t를 loc로 둔다. bus_info_t 구조체를 가르키는 b는 나중에 device instance의 win에 저장될 값이며, config_info_t를 가지는 config는 card를 설정(configuration)하기 위해서 사용한다. DEBUG() 매크로는 현재 cb_config() 함수를 진행하고 있음을 보여준다.

Device instance의 상태(state)는 이제 설정으로 들어가기에 DEV_CONFIG로 둔다. CardBus는 PCI configuration register를 가지므로, 이를 얻기 위해서 GetConfigurationInfo Card Services를 호출한다. 여기서, PC Card와 CardBus가 차이나는 점이 생긴다. CardBus의 경우에는 GetConfiguration Card Services가 사용하는 config_info_t 구조체의 ConfigBase 필드가 PCI vendor/device ID로 설정되고, Option 필드는 CardBus PCI bus 번호를 가지도록 만든다. 또한 Function 필드는 PCI device의 function 번호를 나타낸다. 따라서, 코드에서는 이러한 값을 이용해서 bus와 devfn을 설정하고 있다.

이전 이렇게 찾은(detect) 디바이스가 이미 있는지를 확인하기 위해서 bus에 대한 정보를 가지고 있는 bus_table[] array를 검사하도록 한다. 같은 것이 있다면, 바로 빠져나온다.

```
if (i == MAX_DRIVER) {
    for (i = 0; i < MAX_DRIVER; i++)
```

```

        if (bus_table[i].bus == 0) break;
        b = &bus_table[i]; link->win = (void *)b;
        b->bus = bus;
        b->flags = 0;
        b->ncfg = b->nuse = 1;

        /* Special hook: CS know what to do... */
        i = CardServices(RequestIO, handle, NULL);
        if (i != CS_SUCCESS) {
            cs_error(handle, RequestIO, i);
            return;
        }
        b->flags |= DID_REQUEST;
        b->owner = link;
        i = CardServices(RequestConfiguration, handle, &link->conf);
        if (i != CS_SUCCESS) {
            cs_error(handle, RequestConfiguration, i);
            return;
        }
        b->flags |= DID_CONFIG;
    } else {
        b = &bus_table[i]; link->win = (void *)b;
        if (b->flags & DID_CONFIG) {
            b->ncfg++; b->nuse++;
        }
    }
    loc.bus = LOC_PCI;
    loc.b.pci.bus = bus;
    loc.b.pci.devfn = devfn;
    link->dev = drv->ops->attach(&loc);

    link->state &= ~DEV_CONFIG_PENDING;
}

```

코드 711. cb_config() 함수의 정의(계속)

만약 찾은 디바이스가 새로운 것일 때는($i==MAX_DRIVER$), bus_table[] array에 빈 자리가 있는지를 확인하고, 빈 자리를 b 변수가 가르키도록 만든 후, device의 instance를 나타내는 link의 win 필드가 b를 가르키도록 만든다²⁹². bus_info_t 구조체를 가르키는 b의 bus와 flag을 찾은 bus ID값과 0으로 초기화 하고, ncfg와 nuse에는 각각 1을 넣어서, 설정과 사용 카운트를 초기화 한다.

이전 I/O port window 영역에 대한 예약을 하기 위해서 RequestIO Card Services를 호출할 차례이다. 이미 client driver를 등록하는 곳에서 handle을 알고 있다. 또한 io_req_t 구조체는 CardBus client에서는 사용하지 않기에 NULL로 주었다. 에러가 있다면($i!=CS_SUCCESS$)라면 에러 상황을 알린 후 복귀한다.

bus_info_t 구조체를 가르키고 있는 b의 flag은 I/O에 대한 request를 성공적으로 수행했으므로, DID_REQUEST(=1)를 설정하도록 한다. owner field는 이 bus를 사용하고 있는 device의 instance를 가르키도록 만든다.

RequestConfiguration Card Services가 사용하는 config_req_t 구조체 필드의 대부분은 CardBus에서는 무시되며, 모든 카드의 function들은 앞에서 호출한 RequestIO Card Services에서 얻은 데이터를 기준으로 설정된다. 즉, CardBus bridge를 설정하고, Command, Base Address, Interrupt Line register들과 같은 PCI configuration register들이 초기화된다는 것이다. 여기서 한가지 주의할 것은 config_req_t의 필드중에서 IntType(Interface Type)은 반드시 INT_CARDBUS로 되어있어야 할 것이다. 이것은 이미 attach() 함수에서 초기화해 주었다. 에러가 있었다면, 다시 cs_error() 함수를 호출하고 바로 복귀한다. 이것을 마치면, 설정이 끝났으므로, DID_CONFIG(=2)를 bus_info_t 구조체의 flag에 설정한다.

²⁹² 이미 앞에서 detach()에서 이 필드를 어떻게 사용하고 있는지를 잠시 보았었다. 여기서는 이 필드에 대한 초기화를 담당하고 있다.

만약 찾으려는 디바이스가 이미 있다면, 해당하는 디바이스의 bus_info_t 구조체를 b변수로 놓고, 이것을 device instance를 나타내는 link의 win으로 다시 두도록 한다. 또한 만약 이 디바이스가 이미 설정이 된 상태라면(configuration), ncfg와 nuse를 하나씩 증가시켜서, 설정된 회수와 사용된 회수를 올려준다. dev_locator_t 구조체를 나타내는 loc의 bus 필드에 LOC_PCI로 두고, bus ID와 function을 설정한 후 client driver가 제공하는 driver_operations에 있는 attach() 함수를 호출해서, client driver가 attach시에 해주어야 할 일을 처리하도록 만든다. 따라서, client driver에서는 이곳에서 디바이스에 대한 초기화를 수행하고, 역할을 커널에 등록할 수 있는 기회를 얻게된다. 마지막으로 device instance의 상태(state)에서 DEV_CONFIG_PENDING을 지워서, 설정이 끝났음을 알려준다.

12.8.3.6. CardBus Enabler의 Release() 함수

CardBus enabler의 release() 함수는 디바이스의 detach시와 removal event에서 호출된다. 이 함수가 하는 일은 client driver가 사용하는 시스템의 resource에 대한 것을 해제하는 일을 한다. cb_release() 함수는 아래와 같이 정의된다.

```
static void cb_release(u_long arg)
{
    dev_link_t *link = (dev_link_t *)arg;
    driver_info_t *drv = link->priv;
    bus_info_t *b = (void *)link->win;

    DEBUG(0, "cb_release(0x%p)\n", link);

    if (link->dev != NULL) {
        drv->ops->detach(link->dev);
        link->dev = NULL;
    }
    if (link->state & DEV_CONFIG) {
        /* If we're suspended, config was already released */
        if (link->state & DEV_SUSPEND)
            b->flags &= ~DID_CONFIG;
        else if ((b->flags & DID_CONFIG) && (--b->ncfg == 0)) {
            CardServices(ReleaseConfiguration, b->owner->handle,
                        &b->owner->conf);
            b->flags &= ~DID_CONFIG;
        }
        if ((b->flags & DID_REQUEST) && (--b->nuse == 0)) {
            CardServices(ReleaseIO, b->owner->handle, NULL);
            b->flags &= ~DID_REQUEST;
        }
        if (b->flags == 0) {
            if (b->owner && (b->owner->state & DEV_STALE_LINK))
                cb_detach(b->owner);
            b->bus = 0; b->owner = NULL;
        }
    }
    link->state &= ~DEV_CONFIG;
}
```

코드 712. cb_release() 함수의 정의

cb_release() 함수가 넘겨받는 값은 device instance를 나타내는 dev_link_t 구조체의 포인터이다. 이것으로 link 변수를 초기화한다. 다시 link로부터 driver_info_t 구조체를 얻기 위해서 priv 필드를 사용한다. 이것을 drv 변수로 둔다. bus_info_t 구조체는 이미 link의 win 필드에 config()시에 들어가 있으므로, 여기서 얻도록 한다. DEBUG() 매크로는 cb_release() 함수가 호출되었음을 알려준다.

만약 넘겨받은 device instance의 dev필드가 NULL이 아니라면, client driver가 detach()를 호출하지 않았다는 말이 되므로, 여기서 client driver에게 detach()를 호출할 수 있는 기회를 제공한다. 호출이 끝나면, link의 dev필드는 NULL로 둔다.

만약 device instance의 상태(state) DEV_CONFIG이라면, 설정을 해제하기 위해서 다음과 같은 일을 할 것이다. 먼저 state가 DEV_SUSPEND라고 한다면, 단순히 bus_info_t 구조체를 가르키는 b의 flag에서 DID_CONFIG을 지워주기만하고, 만약 DID_CONFIG이 설정되었는데, ncfg를 감소시켜서 0이 된다면, 즉, 완전히 configuration의 해제를 해야될 때라면, ReleaseConfiguration Card Services를 호출해서 설정을 해제한다. 이것을 마치고나면, 다시 DID_CONFIG을 지워주어야 할 것이다.

만약 bus_info_t 구조체의 flags에 DID_REQUEST가 설정되어 있으며, nuse를 감소시켰을 때 0이라는 값을 가진다면, 이때도 역시 I/O 영역에 대한 해제(release)를 해주어야 할 때이므로, ReleaseIO Card Services를 호출하도록 한다. 마찬가지로 호출이 끝나면, DID_REQUEST를 지워야 할 것이다.

이전 bus_info_t 구조체의 flag에 앞에서 했던 연산의 결과로 완전히 0이라는 값을 가진다면, 즉, configuration에 대한 release와 I/O 영역에 대한 request가 다 해제가 되었다면, bus_info_t 구조체의 owner필드에 어떤 값(device instance 값)을 가지고, 그 상태(state)가 DEV_STALE_LINK라면, cb_detach()를 호출해서 완전히 제거하게 된다. bus와 owner에는 각각 0과 NULL을 주어서 초기의 설정으로 바꿔준다. 또한, 함수를 복귀하기에 앞서 DEV_CONFIG 필드를 state에서 지워주는 것을 잊지 말아야 할 것이다.

12.8.3.7. CardBus Enabler의 Event() 함수

CardBus enabler의 event handler의 역할을 하는 함수는 cb_event() 함수이다. 이 함수는 앞에서 client driver를 등록하는 곳에서 설정한, 받고자 하는 event가 발생했을 때 처리를 맡는 함수이다. 아래와 같다.

```
static int cb_event(event_t event, int priority,
                    event_callback_args_t *args)
{
    dev_link_t *link = args->client_data;
    driver_info_t *drv = link->priv;
    bus_info_t *b = (void *)link->win;

    DEBUG(0, "cb_event(0x%06x)\n", event);

    switch (event) {
    case CS_EVENT_CARD_REMOVAL:
        link->state &= ~DEV_PRESENT;
        if (link->state & DEV_CONFIG)
            cb_release((u_long)link);
        break;
    case CS_EVENT_CARD_INSERTION:
        link->state |= DEV_PRESENT | DEV_CONFIG_PENDING;
        cb_config(link);
        break;
    case CS_EVENT_PM_SUSPEND:
        link->state |= DEV_SUSPEND;
        /* Fall through... */
    case CS_EVENT_RESET_PHYSICAL:
        if (link->state & DEV_CONFIG) {
            if (drv->ops->suspend != NULL)
                drv->ops->suspend(link->dev);
            b->ncfg--;
            if (b->ncfg == 0)
                CardServices(ReleaseConfiguration, link->handle,
                            &link->conf);
        }
        break;
    }
```

코드 713. cb_event() 함수의 정의

`cb_event()` 함수는 발생한 event에 대한 것을 알려주는 `event_t` 구조체인 `event` 변수와 `event` 처리의 우선순위를 가지는 `priority`, 그리고 마지막으로 `event handler`에서 `event`가 발생했을 때 넘겨받는 `event_callback_args_t` 구조체의 포인터인 `args`가 있다. 이 `args` 변수는 이미 client driver를 등록하는 곳(`attach()`)에서 `device`의 `instance`를 가지도록 해 주었다. 따라서, 이것을 이용해서 `link` 변수를 초기화 한다. `driver_info_t` 구조체의 포인터도 이것을 이용해서 `priv` 필드에서 얻어오게되며, 다시 `bus_info_t` 구조체에 대한 포인터인 `b`도 초기화 한다. `DEBUG()`은 현재 `cb_event()` 함수를 수행하고 있다는 것을 알려준다.

`switch()` 절은 발생한 event의 종류에 따라 분기해서 실행한다. `cb_event()` 함수에서 처리할 event는 이미 `attach()` 함수에서 client driver를 등록시킬때 정한 event와 동일한다.

`CS_EVENT_CARD_REMOVAL`은 card가 제거될 때 발생한다. `link`의 `state` 필드에는 현재 카드가 제거되었다는 것을 나타내기 위해 `DEV_PRESENT`를 지운다. 만약 card에 대한 설정(configuration)이 되어 있다면(`DEV_CONFIG`), `cb_release()` 함수를 호출해서 설정을 해제하도록 한다.

`CS_EVENT_CARD_INSERTION`은 card가 삽입될 때 발생한다. `state`에는 `DEV_PRESENT`를 설정(set)하고, 디바이스가 configuration될 필요가 있다는 것을 알리기 위해서 `DEV_CONFIG_PENDING`을 설정한다. 실제적인 디바이스에 대한 설정은 `cb_config()` 함수를 호출하는 것으로 처리할 것이다.

`CS_EVENT_PM_SUSPEND`는 power management와 관련된 event로 발생한다. 디바이스의 power를 끄도록 하는데 사용된다고 보면 될 것이다. `state`는 `DEV_SUSPEND`가 될 것이다. 이 event는 다시 `CS_EVENT_RESET_PHYSICAL` event를 처리하는 곳으로 넘어가서 계속 수행된다.

`CS_EVENT_RESET_PHYSICAL` event는 card에 실제로 reset signal이 전달되기전에 발생한다. 만약 `state`가 `DEV_CONFIG`으로 설정이 끝난 상태라면 다음과 같은 일을 한다. 먼저 등록된 `driver_operations` 구조체의 `suspend()`를 호출해서 CardBus client driver가 `suspend()` 함수를 호출할 수 있도록 보장한다. 여기서 client driver는 power suspend와 같은 일을 물리적으로 처리할 수 있는 기회를 얻게된다. `ncfg`는 하나 감소시키고, 만약 감소시킨 값이 0이 된다면, 완전히 설정을 해제한다. `ReleaseConfiguration Card Services`가 이를 처리할 것이다.

```

case CS_EVENT_PM_RESUME:
    link->state &= ~DEV_SUSPEND;
    /* Fall through... */
case CS_EVENT_CARD_RESET:
    if (link->state & DEV_CONFIG) {
        b->ncfg++;
        if (b->ncfg == 1)
            CardServices(RequestConfiguration, link->handle,
                         &link->conf);
        if (drv->ops->resume != NULL)
            drv->ops->resume(link->dev);
    }
    break;
}
return 0;
}

```

코드 714. `cb_event()` 함수의 정의(계속)

`CS_EVENT_PM_RESUME` event는 새로 card에 power를 넣으라는 event이다. `CS_EVENT_PM_SUSPEND`와 같은 event 이후에 발생할 것이다. 여기서는 단지 `state`에서 `DEV_SUSPEND`를 지우는 것만 하고, 실제적인 처리는 `CS_EVENT_CARD_RESET`에서 일어날 것이다.

`CS_EVENT_CARD_RESET` event는 reset 연산을 다 마치게 되었을 때 발생한다. 만약 디바이스가 `DEV_CONFIG` 상태(state)라면, `ncfg`의 값을 증가시켜주고, 이 값이 0에서 1이 되었다면 새로운 configuration을 하기 위해서 `RequestConfiguration Card Services`를 호출한다. 또한 client driver에게 연산(operation)을 계속 수행할 수 있도록 하기위해서 `driver_operations` 구조체의 `resume()` 함수를 호출한다. 여기서 client driver는 자신이 관리하는 물리적인 디바이스의 power를 ON 상태로 바꿀 수 있는 기회를 얻을 수 있다.

12.9. I/O Mapping and Memory Mapping

16-bit PC Card는 메모리 인터페이스를 위해서 26개의 address line을 가지고 있다. 즉, 이 address line으로 접근할 수 있는 PC Card의 메모리 공간은 최대 64 Mbytes가 된다. 이러한 메모리 공간은 다시 common memory와 attributes memory로 나누어지며, 각각은 다음과 같은 역할을 한다.

- Common Memory : 데이터나 실행 파일을 저장하기 위한 활동 주소 공간이다.
- Attribute Memory : 설정 정보를 저장하기 위한 메모리 공간으로 CIS(Card Information Structure)와 같은 설정정보 및 configuration register를 포함하고 있다.

전체 사용가능한 64Mbytes의 메모리 공간은 common memory와 attribute memory에 대해서 유효하며, 8-bit 접근 및 16-bit 접근이 가능하도록 되어 있다. 16-bit 접근의 경우는 ISA 확장 bus와 HBA(Host Bus Adapter)가 연결될 때를 가정한 것이다.

PC Card의 메모리 대한 access는 크게 다음과 같은 4가지로 나누어서 볼 수 있다. 각각은 common memory와 attribute memory에 대한 read/write이다. 여기서 덧붙여서 I/O device로 socket이 설정될 경우에는 다시 I/O에 대한 read/write 연산이 추가적으로 더 들어가게 되어 전체 6개의 연산이 된다고 볼 수 있다. 여기서 주의할 점은 PC Card의 초기에 설치될 때는 socket이 메모리 socket으로 설정되어 Card Information Structure가 attribute memory address space로부터 읽혀지는 것을 허락하지만, 일단 설정을 맡고 있는 software가 I/O device를 PC Card가 포함하고 있다는 것을 알게되면, PCMCIA HBA를 프로그램해서 I/O device에 대한 지원(support)을 재설정(reconfigure)한다.²⁹³

PCMCIA 카드에서의 메모리 공간을 host가 사용하는 address 공간으로 mapping하는 이유는 다음과 같은 3가지로 나누어 볼 수 있다.

- Host와 socket의 address 공간이 같은 크기가 아니다.
- 시스템 소프트웨어가 시스템의 주소 공간(host address space)에 card가 mapping될 수 있는 address들에 대해서 어떤 제한을 가하고 있다.
- PC Card의 address를 decoding하는 logic이 programmable이 아닌 경우가 있다.

즉, host가 addressing할 수 있는 메모리 공간이 PC Card가 가진 메모리 공간보다 크거나 혹은 작을 수 있으며, 특정 소프트웨어는 특정한 메모리 영역이 PC의 다른 device들을 위해서 메모리가 할당되어 있다고 생각할 수 있는 여지가 있기 때문이다. 또한 PC Card 자체에 내장된 address decoder가 programming이 되지 않는 경우 등이 있기 때문에, PC Card의 address 공간을 새로이 host의 address space로 remapping시켜주어야 한다는 것이다.

Address를 mapping하는데는 다음과 같은 것들이 다시 있을 수 있겠다. 이것은 앞에서 본 이유들에 기인한 것이다.

- Direct Mapping : 시스템의 address가 똑같은 주소로 PC Card의 메모리 주소 공간으로 mapping되는 경우이다. 이럴 경우에는 re-mapping하는 것이 필요없다. 하지만, 이것은 시스템의 특정 address 공간이 항상 특정 PC Card로 mapping된다는 점에서 시스템에 없는 메모리 공간에 대한 access는 어떻게 처리할 것인가에 대한 의문이 생긴다. 또한 PC의 경우에는 첫 1 Mbytes 공간내에 여러 디바이스를 위한 메모리 공간을 reserve하고 있기 때문에 이럴 경우 문제가 생길 여지가 많다.
- Remapping the Host Address to PC Cards with Fixed Addresses : 이것은 시스템의 주소 공간중의 일부를 PC Card가 address space에 mapping하는 방법으로 시스템의 메모리 공간중에서 사용되지 않는 부분의 메모리 주소 공간을 PC Card의 address space로 지정하는 방법이다. 따라서, HBA는 시스템의 address space를 PC Card의 address space로 remapping하는 방법을 구현하고 있어야 할 것이다.
- System Address Space Smaller Than Socket Address Space : 이것은 시스템의 address space가 socket의 address space보다 작은 경우이다. 이럴 경우에는 전체 PC Card의 address 공간을 access할 수 있기 위해서는 PC Card address space를 사용될 때마다 mapping을 새로이 해주는 방법을 써야 할 것이다.

²⁹³ Driver Service와 같은 곳에서 이것을 해줄 수 있을 것이다. Client driver에서는 설정된 것을 알기를 원할 것이며, CIS tuple을 접근해서 이것을 요청할 수 있다.

것이다. 예를 들어서, 16Mbytes의 address space를 가지는 시스템에서 20 Mbytes의 PC Card address space를 접근하는 경우이며, 이때 두번에 나누어서 20 Mbytes의 PC Card address space를 나누어서 각각 mapping해 주는 것이 필요하다. 따라서, 이 경우에도 역시 HBA에 address를 remapping할 수 있는 능력이 요구된다. PC와 같은 환경에서 이러한 기법을 사용하고 있다.

- System Address Space Larger Than Socket Address Space : 이것은 시스템의 address space가 PC Card의 address space보다 충분히 큰 경우를 말하는 경우이다. 이 경우에도 역시 앞에서 말한 것과 같은 방법으로 HBA를 통해서 시스템의 특정 공간 부분에 PC Card의 address space를 mapping시키는 것이 가능하다.

간략하게 나마, PC Card의 주소 공간을 시스템의 주소 공간으로 mapping하는 방법들에 대해서 보았다. PC Card의 주소 공간을 전부 한번에 mapping시키기보다는 일정 부분을 시스템의 메모리 공간에 mapping하기 위해서 사용하는 것이 바로, memory address window라는 것이다. 즉, 메모리 주소 원도우란 PC Card의 주소 공간 중에서 특정 부분을 시스템의 메모리 mapping 시킨 것이라고 생각하면 될 것이다. 이러한 메모리 원도우 각각은 다음과 같은 3개의 레지스터로 이루어질 수 있을 것이다.

- Start Window Address Register : PC Card의 메모리 원도우의 시작주소를 나타내는 레지스터이다.
- Stop Window Address Register : PC Card의 메모리 원도우의 끝을 나타내는 레지스터이다.
- Window Offset Address Register(Mapping Register) : host의 address와 더해져서, PC Card의 메모리 주소 공간의 원하는 위치로 주소를 mapping시키기 위해서 사용되는 레지스터이다. 여기서, 중요한 것은 host address가 하위의 address에 mapping되어야 하기 때문에, 실제로는 host address로부터 빼주어된다. 따라서, 2의 보수(complement)로 표현된다는 것이다.

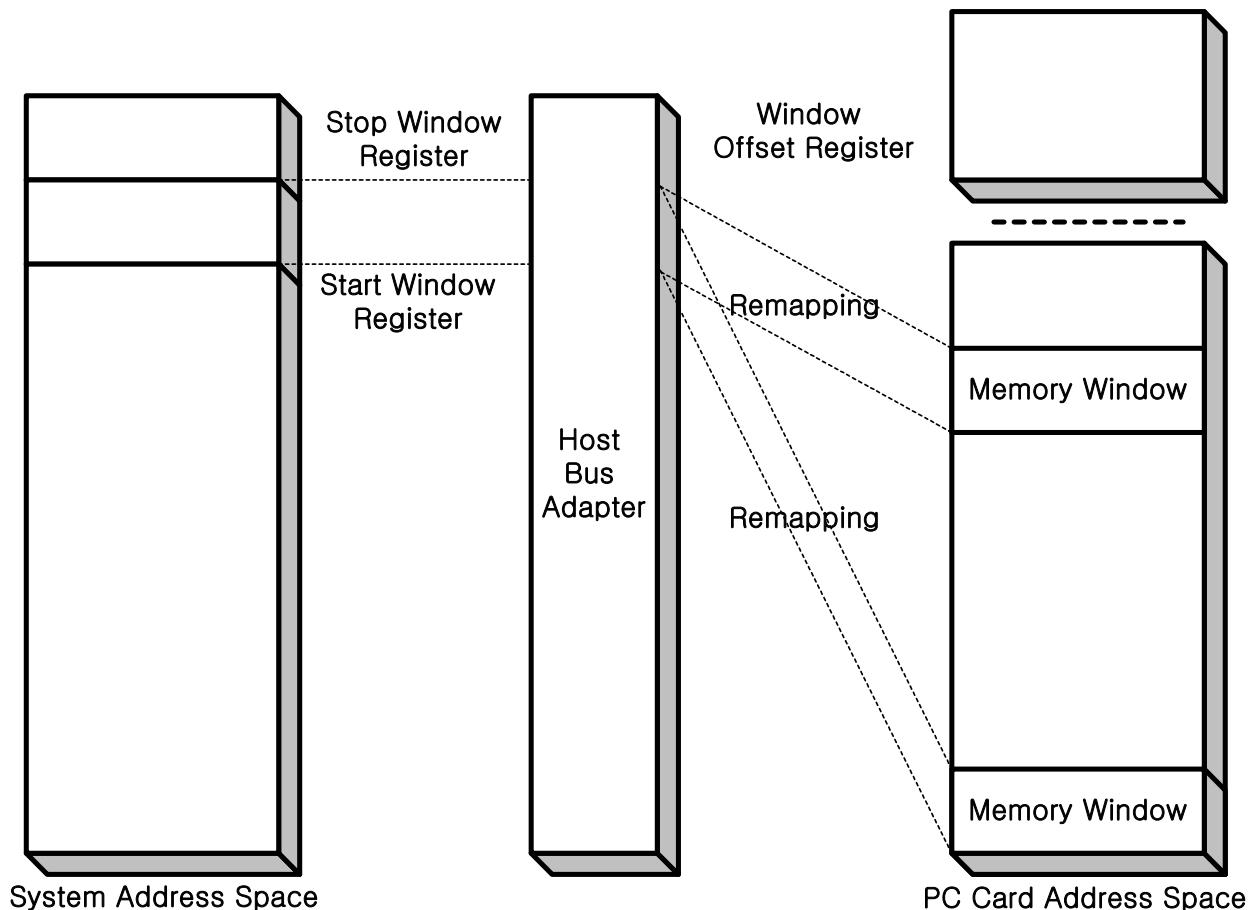


그림 93. PC Card의 메모리 원도우

앞에서는 메모리 address mapping에 대한 것을 보았다. 여기서 부터는 이제 I/O address의 mapping에 대해서 알아보기로 하자. PC Card가 I/O 레지스터를 가지는 경우에 이러한 I/O 레지스터를 접근하기 위해서는 반드시 시스템의 주소 공간에 mapping이 되어야 한다. 만약 시스템이 메모리 address 공간만을 가지고 있는 경우에는 HBA가 이러한 메모리 address를 PC Card의 I/O address 공간으로 mapping해 주어야만 한다. I/O address mapping도 다시 다음과 같이 나누어서 볼 수 있다. 여기서 가정하고 있는 것은 시스템이 I/O Address를 따로 가지고 있는 경우이다.

- Direct Mapped I/O Address : 이 경우는 앞에서 메모리 mapping에 대해서 말했던 것과 같이 PC Card의 I/O 레지스터가 직접 해당하는 시스템의 I/O address로 mapping된 경우이다. HBA는 단순히 PC Card의 I/O address에 해당하는 I/O 주소가 감지되면 그것을 그대로 PC Card로 전달하는 bridge와 같은 역할을 해준다.
- Overlapping I/O windows : 이 경우는 시스템의 다른 device가 사용하는 I/O address 공간을 PC Card의 I/O 레지스터가 같이 사용하게 되는 경우이다. 즉, I/O 레지스터를 위한 window를 크게 잡고, 다른 시스템의 device가 사용하는 address space가 이 window에 포함되도록 한 후, 실제로 PC Card가 가진 I/O register부분이 접근될 때만 반응하도록 해주는 것이다. 따라서, start window register와 stop register는 다른 시스템의 device가 사용하는 I/O address 공간을 포함하는 window를 생성할 수 있다.

위에서 살펴본 것은 PC Card의 주소 공간에 대한 접근이 어떻게 mapping이 되는가 였다. 이것과 덧붙여서 실제로 PC Card에는 다양한 속도의 디바이스가 볼 수 있으며, 또한 이러한 device에 대한 입출력 단위가 8bit, 16bit으로 달라질 수도 있다. 따라서, client driver는 CIS(Card Information Structure)²⁹⁴를 읽어서, PC Card의 속도와 접근하는 데이터 크기를 결정한 후, 이를 HBA가 관리하는 window의 정보로 알려주어야 할 것이다.

12.10. CIS(Card Information Structure)

CIS는 client driver와 같은 소프트웨어가 PC Card에 어떤 card가 설치되었으며, 그것의 속도가 얼마나 되며, 어떤 크기를 가지고 있고, 어떤 시스템의 자원을 사용하는지와 같은 정보를 담는 비 휘발성 메모리 영역이다.

이러한 정보를 얻고난 후에야 HBA는 이 PC Card에 대한 접근을 할 수 있도록 프로그램될 수 있을 것이며, card 자체도 자신의 configuration register에 어떤 값을 써 넣음으로써 초기 설정을 할 수 있게 된다. 가령 예를 들어서, CIS에 들어갈 수 있는 정보로는 power, timing, interrupt, I/O address space, 메모리 address space등과 같은 정보들이 들어가 있기 때문에, 이러한 모든 정보를 얻고난 후, 이를 시스템을 관리하는 운영체제에 해당 자원을 요청해야지만 client driver와 같은 소프트웨어에서 사용할 수 있게 된다. 참고적으로 Configuration register들은 I/O card를 위해서는 반드시 있어야 하는 레지스터이지만, 일반적인 메모리 카드에서는 옵션이다. 또한 CIS와 Configuration register는 모두 attribute 메모리 공간에 존재한다. 즉, 카드의 특성(attribute)를 나타내는 메모리 공간에 위치한다는 말이다.

일반적으로 CIS는 PC Card의 client driver에 의해서 카드를 초기화 하는 동안에 읽어들려지며, card가 어떠한 설정 옵션을 가지고 있는지를 알수 있는 자료로서 사용된다. PC Card client driver가 이러한 CIS를 접근하기 위해서는 앞에서 설명한 Card Services나 Socket Services를 이용해야 하며, 다시 Card Services와 Socket Services를 이용해서 HBA 및 PC Card를 설정한다. 일단 card에 대한 초기화를 마친 상황에서는 다시 이러한 CIS를 접근할 이유가 없다.

12.10.1. CIS의 구조

CIS는 attribute 메모리 주소 영역의 0에서부터 시작하는 공간에 매핑이된다. CIS는 tuple이라고 불리는 데이터 블록의 연결 리스트로 이루어지며, 이러한 tuple은 PC Card의 특성과 기능(function)을 기술한다. 따라서, card를 설정하는 소프트웨어는 이 데이터를 접근해서 PC Card의 특성과 필요한 설정 사항들을 결정해 주어야 할 것이다. Tuple의 종류를 나타내는 것은 각각의 tuple내에 있는 첫번째 byte이며, 고유한 code를 담고 있다.

²⁹⁴ 아래에서 살펴보게되겠지만, 이곳에는 card가 사용하는 정보를 client driver software가 알 수 있도록 설정정보를 담고 있는 메모리 공간이다.

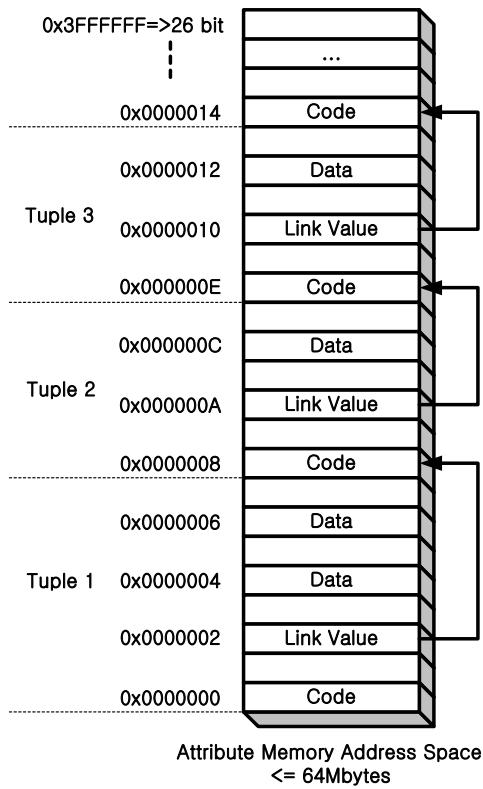


그림 94. Tuple을 가진 CIS의 예

[그림 94]은 Tuple을 가지고 있는 CIS 데이터의 예를 보여준다. 그림에서 보듯이 모든 CIS tuple의 데이터는 짹수 Byte에서 시작하며, 하나의 byte로 그 값을 표현하고 있다. 이것은 8 bit 데이터만을 처리하는 시스템을 위해 데이터 라인에서 단지 8 bit만을 사용하도록 하고 있다. 또한 하나의 tuple은 tuple의 종류를 나타내는 code와 다음 tuple의 시작을 나타내는 link value, 그리고, 해당 tuple의 데이터 값으로 표현된다.

12.10.2. What is Tuple?

Tuple이란 여러개의 원소를 가진 데이터의 집합이란 뜻이다. 즉, tuple은 하나의 특성을 나타내는 값과 그 특성에 해당하는 값으로 이루어진 데이터의 집합이다. Tuple이 가지는 데이터 포맷은 아래와 같다.

심벌	Offset	설명
TPL_CODE	0	tuple의 type code를 나타낸다.
TPL_LINK	2	다음 tuple에 대한 link 역할을 한다. (tuple에서 남은 데이터의 크기를 byte 단위로 나타낸다.)
TPL_DATA	2 * n	tuple의 특정 데이터를 나타낸다. (개개의 tuple에 의해서 정의되는 tuple 데이터의 정의, 형식(format), 길이 등을 나타낸다.)

표 68. 기본적인 tuple의 포맷

물론, 기본적인 tuple의 포맷이 외에 좀더 발전된 형식의 tuple의 포맷도 있다. 하지만, 그것은 단지 TPL_DATA의 확장이라고 보는 것이 옳을 것이다. 즉, TPL_DATA 부분에 좀더 세부적인 정보를 기술하는 것을 덧붙인 것이다.

CIS는 이러한 tuple들의 연결 리스트이다. 따라서, software에서는 이러한 CIS tuple들의 연결리스트를 읽어들여서 다시 software적으로 연결리스트를 구현하고, tuple의 데이터를 차지하는 부분을 해석해

주어야 할 것이다. 즉, CIS에 들어갈 수 있는 정보는 PC Card마다 달라지기에, 이러한 해석도 software마다 달리 할 수 있을 것이다.

12.10.3. Configuration Table Entry

I/O를 지원하는 card의 경우에는 CIS가 configuration table을 가져야한다. 단순한 메모리 카드는 이러한 CIS entry를 가지지 않아도 된다. 이러한 configuration table은 여러개의 entry로 이루어지며, configuration option이라는 집합으로 이루어지며, PC card의 일반적인 연산(operation)을 위해서 필요한 것들로 구성된다. 즉, 각각의 configuration table의 entry는 PC Card가 설정될 수 있는 가능한 자원의 설정들을 반영한다. 다음과 같은 것들이 configuration table의 entry로 포함될 수 있다.

- Power 설정 : Vcc, Vpp1, Vpp2와 같은 power의 설정과 관련된 파라미터 값들이다.
- 시간 설정 정보 : READY 신호를 deassert할 수 있는 최대 시간과 WAIT 신호의 최대 시간을 나타내는 값들이다. 이 값들에 의해서 PC card를 접근하기 위해서 대기 해야하는 시간들이 정해질 수 있을 것이다.
- I/O address 공간 정보 : 최대 16개의 I/O address 공간을 정의한다. 즉, 기본 I/O address와 이 범위(range)내에 있는 address 위치의 개수까지도 정의한다.
- Interrupt 요청 정보 : 시스템의 interrupt 요청 line을 명시한다. 이러한 interrupt line들은 하나나 그 이상의 그룹으로 될 수 있으며, 또한 전달 모드(level mode or pulse mode) 및 interrupt 공유가 지원되는지와 alternative interrupt 신호의 정의(NMI, I/O check, bus error, vendor specific interrupt등)도 포함한다.
- 메모리 address space 정보 : 최대 8개까지의 메모리 주소 영역(memory address space)을 표시할 수 있다. Host의 주소 및 PC Card의 주소가 명시될 수 있으며, host와 PC Card의 주소가 동일할 경우에는 address 변환(translation)은 필요없게되며, direct address mapping이 일어나게된다. 여기서 기술되는 것은 common address 영역에 대한 것으로 I/O address 영역에 대한 정보와는 관련이 없다. 만약 host address 영역이 명시되지 않는다면, 사용될 수 있는 host address 영역의 일부가 HBA에 의해서 common address space영역으로 mapping된다. 기본 주소(base address) 및 범위(range) 값이 이 주소영역에 있는 block을 위해서 필요할 것이다.
- 기타 정보(Miscellaneous) : 현재 PC Card의 표준에서는 두개의 bytes로 이것을 표시하는데, 첫번째 byte는 PC Card가 power down을 지원하는지, SPKR(speaker) pin을 사용하는지, 최대로 지원되는 동일한 card의 개수를 나타내며, 두번째 byte는 DMA를 지원하는지와 지원한다면 DMA의 전송 크기가 얼마나 되는지 및 DREQ 신호를 위해서 PC Card가 어떤 pin을 사용하는지를 나타낸다.
- Subtuple 정보 : 이것은 configuration table entry tuple에 대한 확장 정보를 가지는 부분으로, 이 설정이 적용되어야 하는 운영체제 및 물리적인 디바이스의 정보를 가진다.

아래의 표는 위의 configuration table entry를 추가한 CIS tuple의 포맷을 표현하는 것이다.

심벌	Offset(2 * n)	설명
TPL_CODE	0	Configuration Entry Tuple code를 나타낸다.(CISTPL_CFTABLE_ENTRY=0x1B)
TPL_LINK	1	다음 tuple에 대한 link(최소 2)
TPCE_INDX	2	Configuration table의 index byte로 entry의 index와 interface byte가 따라나오는지, 그리고, 이 entry가 default entry인지를 나타낸다.
TPCE_IF	3	Interface를 기술한다. 이 필드는 위에서 정의한 TPCE_INDX에 interface를 나타내는 bit이 설정된 경우에만 존재한다.
TPCE_FS	..	Feature Selection을 나타내는 byte이다. 즉, option으로 따라나오는 구조체가 있는지의 여부를 나타내는 byte이다.
TPCE_PD	..	Power Description을 나타내는 struture이다.
TPCE_TD	..	Configuration Timing 정보를 나타내는 structure이다.
TPCE_IO	..	I/O address space 정보를 알려주는 structure이다.
TPCE_IR	..	Interrupt Request 정보를 알려주는 structure이다.
TPCE_MS	..	Memory Address Space 정보를 알려주는 structure이다.
TPCE_MI	..	Miscellaneous 정보를 알려주는 structure이다.

TPCE_ST	n	Subtuple 속에 들어있는 설정 정보(configuration)에 대한 추가적인 정보를 제공한다.
---------	---	----------------------------------------------------------

표 69. Configuration Table Entry Tuple을 구성하는 CIS tuple의 정의

Client driver는 위와같은 configuration table entry를 얻어서, 원하는 데이터를 구하기 위해 얻어온 데이터를 parsing해야 하며, 또한 parsing한 데이터를 기초로 다시 시스템에 자신이 사용하는 자원의 할당을 요청해야 할 것이다. 기본적으로 default entry에 속하는 값으로 설정을 해줄 수 있어야 하며, 또한 추가적인 configuration table entry의 정보를 이용해서 이러한 default로 정의한 값을 재설정해 줄 수 있어야 할 것이다. 따라서, default entry는 최소한의 만족되어야 할 정보를 기록하기 위한 것이다.

12.10.4. Multiple Function in PC Card

앞에서는 single function만을 구현한 CIS에 대해서 알아보았다. 그렇다면, multi-function을 구현하는 CIS는 어떻게 구현하는가를 알아볼 차례이다. 다음의 그림을 참고하기 바란다.

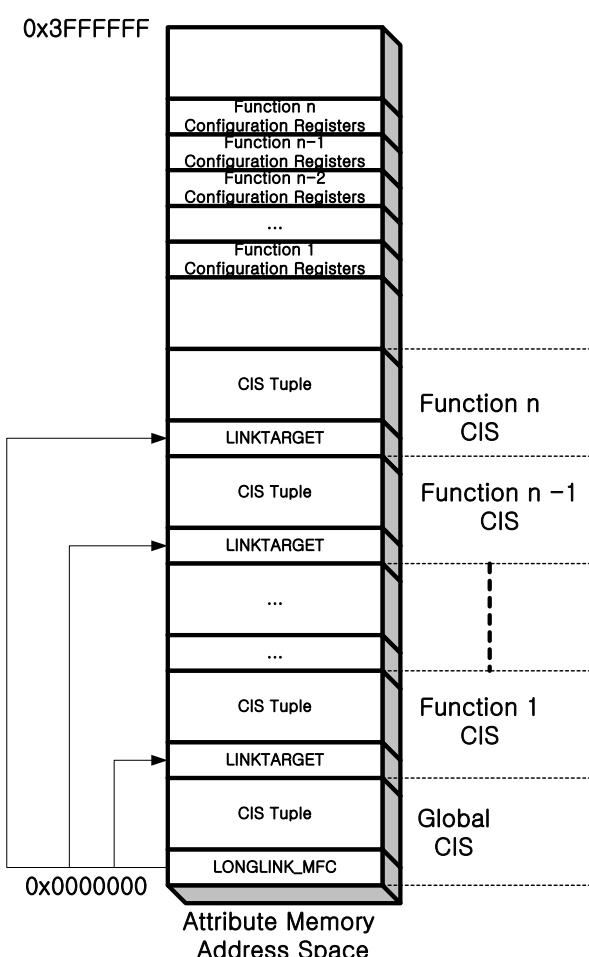


그림 95. Multifunctional PC Card의 CIS 구조

[그림 95]에서 보듯이 multifunctional PC Card의 CIS는 분리된 CIS 구조체를 attributes 메모리 영역에 가져야 하며, 각각의 function들에 대해서 또한 각각의 분리된 configuration register들의 집합을 가진다. 여기서 0x00000000에서 시작하는 CIS 구조체를 global CIS라고 하며, multifunctional PC Card를 구현하는데 있어서 반드시 필요하다. 이 global CIS는 long link multi-function tuple(LONGLINK_MFC)라는 것을 사용해서 각각의 function에 해당하는 CIS에 대한 포인터를 유지한다. 이것을 통해서 각각의 function에 대한 CIS를 접근할 수 있게되는 것이다. 이 포인터가 가르키는 CIS의 첫번째 entry에는 반드시 LINKTARGET 이라는 tuple이 있어야 하며, 이 tuple은 LONGLINK_MFC tuple의 의해서 명시된

시작주소(start address)의 정확성을 검증하는데 사용한다. 또한 각 function에서 사용하는 configuration register들의 집합은 각각의 CIS내에 있는 configuration tuple에 의해서 알 수 있게 된다.

12.11. Card Configuration Register

Card의 configuration register는 configuration tuple에 의해서 attribute 메모리 내에 어디에 있는지가 결정되며, 아래와 같은 형식을 가진다.

Offset	7	6	5	4	3	2	1	0
0								Configuration Option Register
	SRESET	LEVIREQ						Function Configuration Index
2								Configuration and Status Register
	Changed	SigChg	IOISS8	RFU	Audio	PwrDwn	Intr	IntrAck
4								Pin Replacement Register
	CBVD1	CBVD2	CREADY	CWProt	RBVD1	RBVD2	PREADY	RWProt
6								Socket and Copy Register
	RFU		Copy Number					Socket Number
8								Extended Status Register
	Event3	Event2	Event1	Req Attn	Enable3	Enable2	Enable1	Req Attn Enable
10								I/O Base 0
12								I/O Base 1
14								I/O Base 2
16								I/O Base 3
18								I/O Limit

표 70. Configuration Register의 정의

이와 관련된 configuration register들의 각각의 offset은 다음과 같은 값으로 Linux의 PCMCIA 구현에서 나타난다.

#define CISREG_COR	0x00	/* Configuration Option Register의 옵셋 */
#define CISREG_CCSR	0x02	/* Card Configuration and Status Register의 옵셋 */
#define CISREG_PRR	0x04	/* Pin Replacement Register의 옵셋 */
#define CISREG_SCR	0x06	/* Socket and Copy Register의 옵셋 */
#define CISREG_ESR	0x08	/* Extended Status Register의 옵셋 */
#define CISREG_IOBASE_00x0a		/* I/O Base Register 0의 옵셋 */
#define CISREG_IOBASE_10x0c		/* I/O Base Register 1의 옵셋 */
#define CISREG_IOBASE_20x0e		/* I/O Base Register 2의 옵셋 */
#define CISREG_IOBASE_30x10		/* I/O Base Register 3의 옵셋 */
#define CISREG_IOSIZE	0x12	/* I/O Limit Register의 옵셋 */

코드 715. Configuration Register의 각 레지스터에 대한 옵셋 정의

앞에서 이와 관련된 것을 이미 보았다. 여기서는 각각의 필드가 하는 역할들에 대한 이야기를 하도록 하겠다.

12.11.1. Configuration Option Register(COR)

COR은 프로그램할 수 있는 address decoder를 가진 PC Card를 설정하는 일을 한다. 일단 client driver가 CIS를 성공적으로 해석(parse)하고나서 사용할 시스템의 자원을 획득한 후에 이것을 COR을 통해서 할당할 수 있게 되는 것이다. 즉, CIS의 configuration table이 card가 지원하는 configuration option을 명시하고, 각각의 CIS내에 있는 entry는 card의 자원 요구의 다양한 조합을 가진다. 만약 이러한 configuration entry에 명시된 자원의 사용요구를 나타내는 configuration option이 가능하다면, 이것에 해당하는 entry의 index 번호가 COR에 write가 될 것이다. 따라서, 이 entry에 대한 index 번호는 관련된

configuration table entry에 명시된 자원을 card가 사용할 수 있도록 만들어주는 역할을 한다고 볼 수 있다. COR의 각 필드는 아래와 같은 의미를 가지고 있다.

심벌	설명
SRESET	Software reset 을 나타내는 필드이다. 이 bit을 설정하면, card를 reset 상태로 놓게되며, 이것은 RESET 신호를 assertion하는 것과 동일하다.(물론 이 SRESET을 clear하는 것을 제외한 상황을 말한다.) 이 bit을 다시 0으로 돌려주면, card는 hardware reset이후의 상황과 동일한 상태로 놓이게 된다. 이 bit은 hardware reset이나 power up시에 0으로 설정된다.
LevelReq	Level Mode IREQ 를 나타내는 필드이다. Level mode interrupt는 이 bit이 1로 설정될 경우이며, 만약 0으로 되어 있다면, pulse mode interrupt를 사용한다는 뜻이다.
Function Configuration Index	Configuration Index 를 나타내는 필드이다. 이 필드는 card의 configuration table에 있는 entry에 대한 index값을 가진다. 즉, 현재 선택된 configuration option에 해당하는 configuration table의 entry를 나타낸다고 볼 수 있다. 만약 configuration index가 0이라면, card의 I/O는 disable상태이며, 어떠한 I/O cycle들에 대해서도 반응하지 않으며, 단순히 memory-only 인터페이스만 사용할 것이다. 만약 card가 multifunctional이라면, 이 부분은 Multi-function Card Index 정의로 사용되며, 이곳에 있는 각각의 bit은 아래와 같은 의미를 가진다. Bit 0 : 특정 function을 enable/disable하는 일을 한다. 1일 경우에 enable이다. Bit 1 : I/O addressing이 사용되는지를 표시한다. 1일 경우에 I/O address는 base와 limit 레지스터에 의해서 명시된 값으로 function이 사용할 것이며, 0일 경우에는 모든 host의 I/O address가 function으로 전달될 것이다. 당연히 bit 0가 enable인 경우에만 유효하다. Bit 2 : IRQ신호의 routing을 enable시켜준다. 1인 경우에 이 function은 PC Card의 IREQ 라인에 interrupt를 전달할 것이며, 0인 경우에는 이 function에 대해서 interrupt를 disable 시켜준다. 역시 앞에서 말한 bit 0가 enable된 경우에만 유효하다. Bit 3-5 : Vendor specific으로 정해져 있다.

표 71. Configuration Option Register의 필드 정의

위와 같이 정의된 레지스터의 각 값을 다루기 위해서 Linux의 PCMCIA에서는 아래와 같은 것을 정의하고 있다.

#define COR_CONFIG_MASK	0x3f /* Configuration Index를 나타내는 masking 값 */
#define COR_MFC_CONFIG_MASK	0x38 /* Multifunctional Configuration을 위한 masking 값 */
#define COR_FUNC_ENA	0x01 /* Multifunction enable bit */
#define COR_ADDR_DECODE	0x02 /* Address decoding enable bit */
#define COR_IRQ_ENA	0x04 /* IRQ routing enable bit */
#define COR_LEVEL_REQ	0x40 /* Level IRQ bit */
#define COR_SOFT_RESET	0x80 /* Software reset bit */

코드 716. Configuration Option Register를 위한 상수 정의

12.11.2. Card Configuration and Status Register(CCSR)

이 레지스터는 card를 제어하기 위한 다양한 역할과 card의 상태를 보고하기 위한 필드들을 가진다. 이 레지스터가 맡은 중요한 역할로는 아래와 같은 것이 있다.

- Card의 상태 변화를 나타내고, 알려준다.
- PCMCIA host expansion bus의 인터페이스 크기를 알려준다.
- Audio가 enable되는지의 여부를 알려준다.
- Power를 절약하기 위해서 power down 제어를 하는지를 알려준다.
- Interrupt가 pending되었는지의 상태를 알려준다.

CCSR의 레지스터의 각각의 필드는 아래와 같은 정의를 가진다.

심벌	설명
Chng	Status change detected 를 나타내는 bit이다. 이 bit은 하나나 그 이상의 Pin Replacement Register의 bit들이 1로 설정되었음을 나타내며, 일반적으로 STSCHG 신호가 assert되도록 한다. 하지만, SigChg bit이 1이고, Card가 I/O 인터페이스를 위해서 설정되었다면, STSCHG 신호를 나타내는 pin은 이 bit이 설정되었을 때만, assert된다.
SigChg	Signal change enable/disable 을 나타내는 bit이다. 이 bit은 host에 의해서 status 레지스터로부터의 status 변화 신호를 enable/disable하기 위해서 설정되거나 해제된다. 만약 이 bit이 설정되었고, card가 I/O 인터페이스를 위해서 설정되었다면, Chng bit이 STSCHG 신호를 나타내는 pin 63을 제어하게 된다. 만약 아무런 상태변화(status change) 신호를 받지 않기를 원한다면, 이 bit은 0으로 설정되어야 하며, card가 I/O를 위해서 설정되어 있는 동안에 STSCHG 신호 pin은 deasserted로 유지될 것이다.
IOis8	I/O cycle occur only as 8-bit transfers 를 나타내는 bit이다. 만약 host가 data 0에서 7까지 만을 사용해서 I/O cycle을 제공한다면, PCMCIA software는 이 bit을 1로 설정할 것이다. 16-bit 레지스터에 대한 접근은 한번의 16-bit 접근보다는 두번에 걸친 8-bit 씩으로 처리될 것이다. 이것은 16-bit 레지스터와 8-bit 레지스터들이 overlap된 경우에 유용하다.
Resrv	0으로 예약된 부분이다.
Audio	Audio enable 을 나타내는 bit이다. 이 bit은 HBA로 speaker pin을 통해서 audio 정보가 전달되어야 한다는 것을 나타내는 것으로, I/O 인터페이스로 card가 설정된 경우이다.
PwrDn	Power down 을 나타내는 bit이다. 이 bit은 card가 power-down 상태로 가도록 요청을 하기 위해서 설정된다. PCMCIA software는 반드시 card의 READY pin이 low(Busy) 상태에 있는 동안에는 card를 power down 상태로 만들어서는 안된다.
Intr	<p>Interrupt request pending을 나타내는 bit이다. 이 bit은 인터럽트 요청의 내부 상태를 표시하는 역할을 한다. 이 값은 인터럽트가 설정되었거나 되지 않았거나에 관계없이 항상 유용(available)하며, Intr bit이 어떻게 지워(clear)지느냐는 IntrAck bit이 설정되었는가에 의존적이다. 아래와 같다.</p> <p>IntrAck가 0인 경우 : Intr은 function의 interrupt 요청 상태를 반영한다. 만약 function 내에서 인터럽트가 clear되어 있다면, Intr은 function에 의해서 reset된다.</p> <p>IntrAck가 1인 경우 : Intr은 interrupt 상황이 clear된 후에도 그대로 설정(set)된 상태로 남는다. 이때는 시스템의 소프트웨어가 다른 interrupt를 받을 준비가 되었다고 알려주기 위해서 다시 reset 되어야 한다. 이와 같은 것은 인터럽트 공유(interrupt sharing)와 같은 곳에서 사용될 수 있을 것이다.</p>
IntrAck	<p>Interrupt acknowledge를 나타내는 bit이다. 이 bit은 Intr bit의 반응을 결정한다. IntrAck bit은 PC Card의 IREQ pin을 여러개의 function들이 공유할 수 있도록 한다. 아래와 같이 사용한다.</p> <p>IntrAck가 0인 경우 : 이때는 IntrAck가 단일의 interrupt 구현을(즉, 하나의 ISR만이 이 인터럽트를 사용하도록) 지원하도록 Intr function을 reset하는 경우이다.</p> <p>IntrAck가 1인 경우 : 이것은 Intr bit이 이미 interrupt service routine이 interrupt를 이미 service한 경우에도 그대로 설정 상태로 남도록 한다. 일반적으로 interrupt service routine은 function의 specific 레지스터에 있는 interrupt pending bit을 지워서 Intr도 지우도록 한다. 하지만, 인터럽트 공유를 지원하기 위해서는 PCMCIA 특정 소프트웨어가 다음 interrupt 요청을 처리할 준비가 될때까지 Intr bit은 clear되지 않는다. 만약 PCMCIA 소프트웨어에 의해서 지워진다면, 다른 pending된 인터럽트 요청이 PC Card의 IREQ pin을 통해서 assert될 수 있을 것이다. 즉, interrupt의 ack를 요구하고 있다는 말이된다.</p>

표 72. Card Configuration and Status Register의 필드 정의

위의 표에서 보여준 각각의 필드를 위해서 Linux PCMCIA 구현에서는 아래와 같은 값을 정의하고 있다. 각 필드의 bit position에 유의하기 바란다.

#define CCSR_INTR_ACK	0x01	/* IntrAck를 위한 설정 */
-----------------------	------	----------------------

#define CCSR_INTR_PENDING	0x02	/* Intr를 위한 설정, interrupt pending */
#define CCSR_POWER_DOWN	0x04	/* Power down bit */
#define CCSR_AUDIO_ENA	0x08	/* Audio enable bit */
#define CCSR_IOIS8	0x20	/* IOis8 bit */
#define CCSR_SIGCHG_ENA	0x40	/* SigChg enable bit */
#define CCSR_CHANGED	0x80	/* Chng bit */

코드 717. Card Configuration and Status Register을 위한 상수 정의

12.11.3. Pin Replacement Register(PRR)

메모리만을 위한 인터페이스를 사용하는 card의 경우에는 자신의 상태 변화를 status change pin을 통해서 직접적으로 HBA에 알려준다. 하지만, card가 만약 I/O 인터페이스를 사용한다면, status change pin은 다른 I/O 인터페이스를 위한 신호를 전달하는 목적으로 사용된다. 결과적으로 HBA는 I/O card에서 발생할 수 있는 status change에 대해서는 아무런 것도 보지 못한다. 이와 같이 PRR은 메모리 인터페이스에 대해서 status change event들을 알려주기 위한 HBA function을 대치하는 목적으로 사용하는 레지스터이다. 아래와 같은 필드들을 가진다.

심벌	설명
CBVD1, CBVD2	Changed BVD1과 BVD2 를 나타낸다. 이 bit들은 RBVD1이나 RBVD2 bit에 변화가 있을 때 설정된다. 이 bit들은 host에 의해서 clear될 수 있다.
CREADY	Changed Ready 를 나타낸다. 이 bit은 PREADY bit의 상태 변화에 따라서 설정된다. 역시 host가 clear할 수 있다.
CWProt	Changed Write Protect 를 나타낸다. 이 bit은 RRWProt의 상태 변화에 따라서 설정된다. 마찬가지로 host에 의해서 clear될 수 있다.
RBVD1, RBVD2	Current State of BVD1과 BVD2 를 나타낸다. 이 bit들은 Battery Voltage Detect 회로의 내부 상태를 표현하는 것들로, card가 battery를 가지고 있는 경우에 해당한다. BVD1과 BVD2는 각각 62번과 63번 pin을 나타내며, 이 bit들이 설정되면, 해당하는 change를 나타내는 bit도 역시 설정된다. 하지만, 이 bit들이 clear된다고 해서 state change를 나타내는 bit도 영향을 받는 것은 아니다.
RRdy	Current State of Ready 를 나타내는 bit이다. 이 bit은 READY 신호의 내부 상태를 나타내는 bit으로 READY의 상태를 반영해준다. READY pin은 I/O card의 경우에는 Interrupt Request로 사용되기 위해서 제 할당되기 때문이다. 만약 이 bit이 설정되면, 마찬가지로 해당하는 change bit도 설정된다. Clear될 때는 change bit에는 아무런 영향도 주지 못한다.
RWProt	Current State of Write-Protect Switch 를 나타내는 bit이다. 이 bit은 Write-Protect switch의 현재 상태를 나타낸다. 만약 pin 24가 IOIS16을 위해서 사용될 때는 이 bit으로 Write Protect switch의 상태를 반영한다. 이 bit이 설정되면, 해당하는 change bit도 같이 설정되며, 지워질 경우에는 영향을 주지 못한다.

표 73. Pin Replacement Register의 필드 정의

아래의 상수들은 위에서 정의한 필드들을 위해서 사용할 수 있는 Linux의 PCMCIA 구현에 사용한 것들이다.

/* Status를 위한 상수 값들 */		
#define PRR_WP_STATUS	0x01	/* Write Protect */
#define PRR_READY_STATUS	0x02	/* Ready */
#define PRR_BVD2_STATUS	0x04	/* BVD2 상태 */
#define PRR_BVD1_STATUS	0x08	/* BVD1 상태 */
/* 상태 변화(status change)를 위한 상수 값들 */		
#define PRR_WP_EVENT	0x10	/* Write Protect Event */
#define PRR_READY_EVENT	0x20	/* Ready Event */
#define PRR_BVD2_EVENT	0x40	/* BVD2 Event */

#define PRR_BVD1_EVENT	0x80	/* BVD1 Event */
------------------------	------	------------------

코드 718. Pin Replacemant Register를 위한 상수 정의

12.11.4. Socket and Copy Register(SCR)

이 레지스터는 I/O card를 위한 것으로 하나나 그 이상의 동일한 card가 존재할 수 있도록 해주며, 동일한 I/O address 영역을 사용할 수 있도록 만들어준다. 이것은 ATA(IDE) drive들에 사용될 수 있으며, 이 경우 각각의 driver는 0과 1이라는 것으로 구분되어진다. 각각은 동일한 I/O address space를 사용하지만, socket and copy register를 사용해서 고유(unique)하게 식별될 수 있다. 즉, 첫번째 설정된 card가 copy zero로 정해지면, 그 이후의 설정될 card들은 연속적인 copy 번호를 부여받는다(copy 1, copy2, etc.). 또한 socket 번호는 이러한 copy가 점유(occupy)하고 있는 socket을 식별하기 위해서 사용된다. 아래와 같은 필드를 가지고 있다.

심벌	설명
Reserved	예약된 부분이다. 레지스터에 값을 적을 때 software적으로 0으로 설정되어야 한다.
Copy Number	Card가 동일하게 설정된(여기서는 twin card들이라는 말을 사용하고 있다.) 여러개의 card과 공존할 수 있다. 이때 특정한 card를 나타내기 위해서 이 copy number를 사용한다. 0에서부터 MAX개 까지의 twin card를 가질 수 있으며, 이때 이 필드는 n 번째의 card를 나타내기 위해서 n이란 값으로 설정되며, 첫번째로 install된 카드가 0을 가진다. 이것은 고유하게 하나하나의 card를 나타낼 수 있고, 연속적으로 번호가 매겨진다는 것을 보장한다면, 동일한 card들이 I/O port들의 일반적인 집합을 공유하도록 허락한다.
Socket Number	이 필드는 n 번째 socket에 있는 card를 표시하기 위해서 사용한다. 첫번째 socket이 0이 되며, 마찬가지로 고유하게 식별가능하다는 하기만 하면, I/O port의 일반적인 집합을 공유할 수 있도록 허락한다.

표 74. Socket and Copy Register의 필드 정의

아래의 값은 위에서 설명한 각 필드의 값을 구하기 위한 mask의 역할을 수행하는 Linux에서의 PCMCIA 구현에서 사용한 상수들이다.

#define SCR_SOCKET_NUM	0x0f	/* Socket Number를 위한 bit mask */
#define SCR_COPY_NUM	0x70	/* Copy Number를 위한 bit mask */

코드 719. Socket and Copy Register를 위한 상수 정의

12.11.5. Extended Status Register(ESR)

이 레지스터는 PC Card의 표준에 STSCHG pin을 통해서 알려질 수 있는 event의 수를 늘리기 위해서 첨가된 것이며, software가 event를 감지하고 clear시킬 수 있도록 하고 있다. 크게 이 레지스터는 상위 4bit(nibble)과 하위의 4bit으로 나누어지며, 상위의 4bit은 해당하는 function의 event가 발생했을 때 설정되며, 하위의 4bit은 CCSR의 “Changed”를 나타내는 설정을 enable/disable하는데 사용한다. 만약 status change interrupt가 발생하면, PC Card의 software는 이 레지스터를 읽어서 어떤 bit이 interrupt를 일으켰는지를 결정하기 위한 정보로서 사용할 수 있다. 아래와 같은 필드들을 가진다.

심벌	Bit	설명
Event 3	7	0 - 예약된 영역임
Event 2	6	0 - 예약된 영역임
Event 1	5	0 - 예약된 영역임
Req Attn	4	PC Card에서 발생하는 event의 1ms이내에 latch된다. 만약 이 bit이 1로 되어있고, Req Attn Enable이 1로 설정된 경우에는 Configuration and Status Register의 Changed bit도 역시 1이되며, 만약 SigChg bit이 1로 host가 설정한 경우에는 STSCHG pin이 assert된다. Host가 이 bit에 1을 쓰는 것으로 다시 reset을 설정하게되며, 0을 쓰는 것은 아무런 효과가 없다.

Enable 3	3	0 - 예약된 영역임
Enable 2	2	0 - 예약된 영역임
Enable 1	1	0 - 예약된 영역임
Req Attn Enable	0	이 bit을 1로 설정하는 것은 Configuration and Status Register의 Changed bit을 enable 시켜준다. 즉, 앞에서 설명한 것과 같이 Req Attn Bit이 설정된 경우가 이에 해당한다. 만약 이 bit이 0으로 재설정(reset)이 되면, 이러한 일은 일어나지 않을 것이다. Req Attn bit 자체는 Req Attn Enable bit의 영향을 받지 않는다.

표 75. Extended Status Register의 필드 정의

아래의 값들은 위의 표에서 예약영역을 제외한 부분을 설정하기 위한 Linux의 PCMCIA 구현의 상수 값들이다.

#define ESR_REQ_ATTN_ENA	0x01
#define ESR_REQ_ATTN	0x10

코드 720. Extended Status Register를 위한 상수 정의

ESR_REQ_ATTN_ENA와 ESR_REQ_ATTN은 각각 ESR 레지스터에서 어떤 bit이 해당하는 값을 가지는지를 나타내고 있다.

12.11.6. I/O Base and Limit Register

PC Card의 표준에서는 여러개의 I/O base address register들을 정의하고 있으며, 이를 각각은 multifunction card들에 대해서 사용된다. 하지만, 하나의 function 카드들에 대해서도 여러개의 I/O base address가 사용 가능하다. 이 레지스터들은 현재 card의 function이 I/O register들이 host 프로세서의 주소 공간에 mapping된 기본(base) I/O address를 나타낸다. 따라서, 이곳에 기본 주소를 두고 있는 레지스터의 총 개수는 프로세서에 의존적이며, 예를 들어 x86 호환과 같은 프로세서에서는 단지 64Kbytes 영역을 이를 위해서 사용할 수 있으므로 단지 첫 2개의 I/O base register만을 가지고 전체 64Kbytes 공간상의 기본주소를 나타내는데 사용할 수 있게 된다. 즉, 16bit(8bit + 8bit) address를 생성하기 위해서 두개의 I/O base address register를 사용하는 것이다.

I/O Limit Register는 base address에서 시작해서 최대로 host의 address에 mapping될 수 있는 최대 I/O address의 범위를 설정하는 레지스터이다. 이 레지스터의 포맷은 가장 중요한 bit 위치(most significant bit)이 address를 decode하는데 필요한 address line의 수를 결정하도록 되어 있다. 여기서 중요한 것은 가장 큰 I/O address space는 8bit 레지스터 전체가 1로 설정된 경우로 256 bytes의 I/O를 위한 address 범위를 결정한다. 아래와 같은 포맷을 가진다.

최대 address range	7 bit	6 bit	5 bit	4 bit	3 bit	2 bit	1 bit	0 bit
2	0	0	0	0	0	0	0	1
4	0	0	0	0	0	0	1	1
8	0	0	0	0	0	1	1	1
16	0	0	0	0	1	1	1	1
32	0	0	0	1	1	1	1	1
64	0	0	1	1	1	1	1	1
128	0	1	1	1	1	1	1	1
256	1	1	1	1	1	1	1	1

표 76. I/O Limit Register가 가질 수 있는 값과 address range의 상관관계

이 레지스터는 옵션이며, 만약 모든 card의 function이 동일한 수의 I/O register를 사용한다면, 각각의 function들에 대해서 구현될 필요는 없을 것이다.

13. PCMCIA Client Device Driver

이전 실재적으로 어떻게 PCMCIA Client Device Driver를 구현하는지에 대해서 알아볼 차례이다. 이곳에서 볼 Client Device Driver의 예는 pcmcia-cs-X.X.X.tar.gz에 Client Device Driver로 들어있는 dummy_cs.c이다. 이 Client Device Driver는 어떻게 I/O Card Client를 작성하는지를 보여주는 예제로서 일반적인 point enabler로서 카드의 CIS가 명시하는 데로 card를 설정하는 동작을 할 것이다.

13.1. dummy_cs 모듈의 적재와 해제

역시 제일 먼저 보는 부분은 모듈로서 동작을 시작하는 부분이다. 해당하는 코드는 module_init()와 module_exit() 매크로로 정의된 init_dummy_cs() 함수와 exit_dummy_cs() 함수이며, 아래와 같다.

```
static dev_info_t dev_info = "serial_cs";
static dev_link_t *dev_list = NULL;
...
static int __init init_dummy_cs(void)
{
    servinfo_t serv;
    DEBUG(0, "%s\n", version);
    CardServices(GetCardServicesInfo, &serv);
    if (serv.Revision != CS_RELEASE_CODE) {
        printk(KERN_NOTICE "dummy_cs: Card Services release "
               "does not match!\n");
        return -1;
    }
    register_pccard_driver(&dev_info, &dummy_attach, &dummy_detach);
    return 0;
}

static void __exit exit_dummy_cs(void)
{
    DEBUG(0, "dummy_cs: unloading\n");
    unregister_pccard_driver(&dev_info);
    while (dev_list != NULL) {
        del_timer(&dev_list->release);
        if (dev_list->state & DEV_CONFIG)
            dummy_release((u_long)dev_list);
        dummy_detach(dev_list);
    }
}
module_init(init_dummy_cs);
module_exit(exit_dummy_cs);
```

코드 721. dummy_cs Client Driver의 적재와 해제

init_dummy_cs() 함수는 모듈의 적재 시에 호출되며, 먼저 CardService() 함수를 GetCardServiceInfo를 넘겨주어서 호출한다. 넘겨지는 정보는 ~/include pcmcia/cs.h에 정의된 servinfo_t 구조체인 serv에 저장될 것이다. 아래에 servinfo_t에 대한 정의가 나와 있다.

typedef struct servinfo_t {	
char Signature[2];	/* Signature */
u_int Count;	/* 사용 카운터 */
u_int Revision;	/* Revision 번호 */
u_int CSLevel;	/* Card Service Level */
char *VendorString;	/* Vendor를 나타내는 String */

```
} servinfo_t;
```

코드 722. servinfo_t 구조체의 정의

servinfo_t 구조체의 Revision에는 현재 제공되는 Card Service의 Version 코드가 들어가 있는데, 이것이 CS_RELEASE_CODE²⁹⁵ 와 같지 않다면, 에러 메시지를 출력하고 -1을 돌려준다. 같다면, register_pcmcia_driver()에 dev_info_t과 detach 함수 및 attach 함수의 포인터를 넘겨주어 호출한다. 복귀 값은 0이다.

모듈의 해제 시에는 앞에서 한 일을 반대로 해주면 될 것이다. 먼저 unregister_pccard_driver() 함수에 앞에서는 달리 dev_info_t 만을 넘겨주어 호출한다. dev_list가 NULL이 아닌 동안에, dev_list->release를 argument로 해서 del_timer() 함수를 호출해서 timer를 제거하고, dev_list->state 필드가 DEV_CONFIG bit가 설정되어 있다면, dummy_release()에 dev_list를 넘겨주면서 호출한다. 이전 dummy_detach() 함수를 호출해서 Client Driver를 떼어내는 일이 남았다. 여기서 한가지 dev_list 구조체의 type인 dev_link_t을 보도록 하자.

```
typedef struct dev_link_t {
    dev_node_t          *dev;           /* PCMCIA device node에 대한 포인터 */
    u_int               state, open;   /* 현재 device의 상태와 open 여부 */
    wait_queue_head_t   pending;       /* 이 디바이스의 wait queue(대기 큐) */
    struct timer_list   release;      /* Release timer의 list */
    client_handle_t     handle;       /* Client의 핸들 */
    io_req_t            io;           /* I/O 요청 */
    irq_req_t           irq;          /* IRQ 요청 */
    config_req_t        conf;         /* 설정(configuration)에 대한 요청 */
    window_handle_t    win;          /* 메모리 window에 대한 handler */
    void                *priv;        /* Private data structure에 대한 포인터 */
    struct dev_link_t   *next;        /* 다음 연결에 대한 포인터 */
} dev_link_t;
```

코드 723. dev_link_t 구조체의 정의

dev_link_t 구조체는 아래에서 살펴볼 여러 함수들에 대한 argument로 넘겨지게 되므로, 각 함수가 현재의 context를 알 수 있는 만큼의 정보를 가질 수 있는 자료구조 되어있다. 또한 디바이스가 설정하고 있는 각종 parameter값과 디바이스에 고유한 데이터를 저장하기 위한 private 필드를 가지고 있으며, 연결 리스트로 구성하기 위한 pointer도 유지하도록 한다. 각각의 값들에 대해서는 아래에서 관련된 부분에 대한 것을 살펴볼 때 자세히 보도록 하겠다.

13.2. dummy_cs의 Attach와 Detach

먼저, dummy_attach() 함수와 dummy_detach() 함수부터 보기로 하자. 이들 각각은 Driver Service에서 Client Device Driver를 등록하고 해제하는 부분에서 호출될 것이다.

```
static dev_link_t *dummy_attach(void)
{
    local_info_t *local;
    dev_link_t *link;
    client_reg_t client_reg;
    int ret, i;

    DEBUG(0, "dummy_attach()\n");

    /* Allocate space for private device-specific data */
```

²⁹⁵ 현재 사용되고 있는 값은 ~include/pcmcia/version.h에 있는 0x3119 이다.

```

local = kmalloc(sizeof(local_info_t), GFP_KERNEL);
if (!local) return NULL;
memset(local, 0, sizeof(local_info_t));
link = &local->link; link->priv = local;

/* Initialize the dev_link_t structure */
link->release.function = &dummy_release;
link->release.data = (u_long)link;

/* Interrupt setup */
link->irq.Attributes = IRQ_TYPE_EXCLUSIVE;
link->irq.IRQInfo1 = IRQ_INFO2_VALID|IRQ_LEVEL_ID;
if (irq_list[0] == -1)
    link->irq.IRQInfo2 = irq_mask;
else
    for (i = 0; i < 4; i++)
        link->irq.IRQInfo2 |= 1 << irq_list[i];
link->irq.Handler = NULL;

```

코드 724. dummy_attach()함수

먼저 커널에서 사용하게 될 데이터 구조체에 대한 메모리를 할당 받는다(kmalloc()). 디바이스 드라이버가 사용하는 모든 메모리는 non-pageable memory에 있어야 하기 때문에 메모리 할당시 GFP_KERNEL 옵션을 주도록 한다. local변수의 타입인 local_info_t은 아래와 같이 정의된다.

```

typedef struct local_info_t {
    dev_link_t           link;          /* dev_link_t을 사용하기 위해서 */
    dev_node_t           node;          /* dev_node_t을 사용하기 위해서 */
    int                 stop;          /* 현재의 Client Driver의 stop상태를 표시 */
    struct bus_operations *bus;         /* Bus에 대해서 행할 수 있는 연산 포인터의 정의 */
} local_info_t;

```

코드 725. local_info_t의 정의

이 구조체에서 주의해서 볼 부분은 바로 bus_operations 구조체에 대한 포인터이다. 이것은 시스템의 Bus에 대해서 행할 수 있는 연산(함수)의 포인터들을 가지고 있는 구조체이다. 정의는 ~/linux/include/pcmcia/bus_ops.h에 아래와 같이 정의되어 있다.

```

typedef struct bus_operations {
    void *priv;                                /* Private 데이터의 저장공간 */
    u32 (*b_in)(void *bus, u32 port, s32 sz);   /* bus에서 하나의 input을 받기 위해 */
    void (*b_ins)(void *bus, u32 port, void *buf, u32 count, s32 sz); /* 여러 개의 input을 받기 위해 */
    void (*b_out)(void *bus, u32 val, u32 port, s32 sz); /* bus에서 하나의 output을 보내기 위해 */
    void (*b_outs)(void *bus, u32 port, void *buf, u32 count, s32 sz); /* 여러 개의 output을 보내기 위해 */
    void *(*b_ioremap)(void *bus, u_long ofs, u_long sz); /* I/O remapping */
    void (*b_iounmap)(void *bus, void *addr); /* I/O unmapping */
    u32 (*b_read)(void *bus, void *addr, s32 sz); /* Bus에 대한 read연산 */
    void (*b_write)(void *bus, u32 val, void *addr, s32 sz); /* Bus에 대한 write연산 */
    void (*b_copy_from)(void *bus, void *d, void *s, u32 count); /* Copy data from */
    void (*b_copy_to)(void *bus, void *d, void *s, u32 count); /* Copy data to */
    int (*b_request_irq)(void *bus, u_int irq, /* Interrupt 할당 요구 */
                        void (*handler)(int, void *, struct pt_regs *), /* ISR */
                        u_long flags, const char *device, void *dev_id); /* Interrupt Flag, Device와 ID */
    void (*b_free_irq)(void *bus, u_int irq, void *dev_id); /* Interrupt 할당 해제 */
} bus_operations;

```

코드 726. bus_operations 구조체의 정의

위의 bus_operations 구조체를 보면 알 수 있듯이 이는 PCMCIA가 하나의 bus로서 기능을 갖추고 있다는 것을 가정해서, 이 bus에 대해서 우리가 일반적으로 찾아 볼 수 있는 연산들을 정의해 둔 것이다. PCMCIA의 socket은 HBA(Host Bus Adapter)에 연결되며, 다시 이 HBA는 시스템의 bus에 연결된다. 즉, ISA bus나 PCI bus와 같은 곳에서 연결되어 접근할 수 있도록 만들어주는 것이다. 따라서, 이러한 bus에서 사용할 수 있는 연산들에 대한 정의를 앞에서 보여주고 있다고 생각하면 된다.

이전 적절하게 앞에서 할당받은 local 데이터 구조체의 필드를 채워주는 일을 하도록 한다. 먼저 release시에 호출된 함수(dummy_release())와 이 함수에 인자로서 넘겨질 데이터(link)를 초기화 하고, 다시 interrupt를 설정한다. 이때 인터럽트의 attribute값으로 IRQ_TYPE_EXCLUSIVE를 주고, IRQ의 information값으로 IRQ_INFO2_VALID | IRQ_LEVEL_ID를 준다. 이것은 ~/include/pcmcia/cs.h에 정의된 것으로 RequestIRQ()와 ReleaseIRQ() Card Service의 attribute값으로 사용하기 위한 것이다. 아래와 같이 정리할 수 있다.

심벌	값	설명
IRQ_TYPE	0x03	IRQ의 type을 나타낸다고 표시한다.
IRQ_TYPE_EXCLUSIVE	0x00	IRQ의 type이 현재 exclusive(배타적)으로 사용된다고 표시한다.
IRQ_TYPE_TIME	0x01	이 인터럽트는 다른 Card Service 드라이버들과 time-shared될 수 있다는 것을 나타내며, 단지 하나의 드라이버만이 특정 시간에 enable될 수 있다.
IRQ_TYPE_DYNAMIC_SHARING	0x02	이 인터럽트는 다른 인터럽트와 동적으로 공유(sharing)됨을 표시한다.
IRQ_FORCED_PULSE	0x04	인터럽트가 기본적인 level mode보다는 pulsed mode로 반드시 설정되어야 한다는 것을 나타낸다.
IRQ_FIRST_SHARED	0x08	IRQ_TYPE_TIME과 함께, shared interrupt를 요청한 첫 번째 드라이버에 의해서 설정되어야 한다.
IRQ_HANDLE_PRESENT	0x10	반드시 설치된 인터럽트 service routine에 대해서 irq_req_t 구조체의 Handler() 필드가 포인터를 가진다는 것을 나타낸다.
IRQ_PULSE_ALLOCATED	0x100	현재 IRQ가 pulse mode로 할당되었다는 것을 표시한다.

표 77. irq_req_t의 attribute field 값에 대한 정의

다시 irq_req_t 구조체에 대한 정의를 보도록 하면, 아래와 같다. 이것은 ~/include/pcmcia/cs.h를 참고하기 바란다. 앞에서는 이 구조체의 Attribute field의 정의만을 보았었다.

```
/* For RequestIRQ and ReleaseIRQ */
typedef struct irq_req_t {
    u_int      Attributes;          /* IRQ의 attribute에 대한 필드 */
    u_int      AssignedIRQ;         /* 할당된 IRQ */
    u_int      IRQInfo1, IRQInfo2;   /* IRQ의 정보를 나타내는 두개의 필드 */
    void *Handler;                 /* IRQ의 핸들러(ISR: Interrupt Service Routine)에 대한 포인터 */
    void *Instance;                /* 이 인터럽트와 연결될 client의 instance */
} irq_req_t;
```

코드 727. irq_req_t 구조체의 정의

irq_req_t 구조체의 IRQInfo1과 IRQInfo2는 각각이 CFTABLE_ENTRY²⁹⁶ tuple의 interrupt description bytes에 해당하는 것으로, 만약 IRQInfo1에 IRQ_INFO2_VALID가 설정(set)된 경우에는 IRQInfo2는 허가된

²⁹⁶ CIS tuple에 정의된 필드이다. 이것과 관련된 것은 CIS(Card Information Structure)에서 찾을 수 있다.

interrupt 값의 bit-mapped mask 역할을 하게된다. 또한 각각의 bit은 하나의 interrupt line에 해당하므로 bit 0은 irq 0, bit 1은 irq 1에 해당한다고 볼 수 있다. 따라서, 0x1100(0001 0001 0000 0000b)이라고 mask를 설정할 경우에는 irq 12와 irq 8이 사용될 수 있다. 만약 IRQ_INFO2_VALID가 설정되지 않았다면, IRQInfo1만이 요구된 인터럽트 번호를 나타낸다. 만약 요구가 받아들여져서 적절히 인터럽트를 예약(reserve)했다면, 이 인터럽트 번호는 AssignedIRQ로 들어갈 것이다.

따라서, 코드에서 irq_list[0]와 -1을 비교해서 만약 같은 값을 가지고 있다면, 현재 모듈이 로드될 때 사용할 IRQ값으로 아무것도 주지 않았다는 것이되므로 irq_mask로 irqIRQInfo2를 초기화 시킨다. 만약 모듈의 loading시에 특정 값을 주었다면, 그값으로 irqIRQInfo2를 설정한다. IRQ에 대한 핸들러는 dummy_cs에서 NULL로 설정하고 있다. 아래의 코드를 잠시 보도록 하자.

```
/* Release IO ports after configuration? */
static int free_ports = 0;

/* The old way: bit map of interrupts to choose from */
/* This means pick from 15, 14, 12, 11, 10, 9, 7, 5, 4, and 3 */
static u_int irq_mask = 0xdeb8;
/* Newer, simpler way of listing specific interrupts */
static int irq_list[4] = { -1 };

MODULE_PARM(free_ports, "i");
MODULE_PARM(irq_mask, "i");
MODULE_PARM(irq_list, "1-4i");
```

코드 728. dummy_cs에서 사용할 자원들에 대한 정의

free_ports는 I/O port를 configuration후에 release할지를 결정하기 위한 변수이며, irq_mask는 사용할 IRQ의 list를 만들기 위해서 0xDEB8(1101 1110 1011 1000b)를 사용했다(15, 14, 12, 11, 10, 9, 7, 5, 4, 3을 IRQ로 사용한다.). 그리고, 모듈의 loading시 구성을 위해서 free_ports와 irq_mask, irq_list를 MODULE_PARM() 매크로를 사용하고 있다.

```
/*
General socket configuration defaults can go here. In this
client, we assume very little, and rely on the CIS for almost
everything. In most clients, many details (i.e., number, sizes,
and attributes of IO windows) are fixed by the nature of the
device, and can be hard-wired here.
*/
link->conf.Attributes = 0;
link->conf.Vcc = 50;
link->conf.IntType = INT_MEMORY_AND_IO;

/* Register with Card Services */
link->next = dev_list;
dev_list = link;
client_reg.dev_info = &dev_info;
client_reg.Attributes = INFO_IO_CLIENT | INFO_CARD_SHARE;
client_reg.EventMask =
    CS_EVENT_CARD_INSERTION | CS_EVENT_CARD_REMOVAL |
    CS_EVENT_RESET_PHYSICAL | CS_EVENT_CARD_RESET |
    CS_EVENT_PM_SUSPEND | CS_EVENT_PM_RESUME;
client_reg.event_handler = &dummy_event;
client_reg.Version = 0x0210;
client_reg.event_callback_args.client_data = link;
ret = CardServices(RegisterClient, &link->handle, &client_reg);
if (ret != CS_SUCCESS) {
    cs_error(link->handle, RegisterClient, ret);
```

```

        dummy_detach(link);
        return NULL;
    }
    return link;
} /* dummy_attach */

```

코드 729. dummy_attach() 함수의 정의

이전 dev_link_t 구조체의 conf 필드에 대한 초기화를 하도록 한다. dev_link_t 구조체의 conf 필드는 다시 config_req_t 구조체로 정의되며, 아래와 같다.

```

/* For RequestConfiguration */
typedef struct config_req_t {
    u_int          Attributes;           /* 설정하고자 하는 attribute */
    u_int          Vcc, Vpp1, Vpp2;       /* Voltage에 대한 설정 */
    u_int          IntType;              /* 인터페이스 type */
    u_int          ConfigBase;           /* Attribute memory 내에서의 configuration register offset
*/
    u_char         Status, Pin, Copy, ExtStatus; /* Parameter 값 */
    u_char         ConfigIndex;           /* Parameter 값 */
    u_int          Present;               /* 어떤 CIS configuration register가 card에서 구현되었는가를 나타내는 bitmap
*/
} config_req_t;

```

코드 730. config_req_t 구조체의 정의

이 구조체는 RequestConfiguration Card Service를 이용해서 실제적으로 socket을 설정하기 위해서 사용할 때 쓴다. 즉, voltage에 대한 설정, CIS configuration register들에 대한 설정, I/O port window 및 interrupt를 위한 설정등에 사용한다.

IntType이 가질 수 있는 값으로는 INT_MEMORY, INT_MEMORY_AND_IO, INT_CARDBUS등이 있으며, 각각 memory와 I/O, 혹은 Cardbus type의 interface를 가진다는 것을 의미한다.²⁹⁷ 또한 Voltage값으로는 1/10 volt를 나타내도록 하며, 현재로서는 Vpp1과 Vpp2가 같은 값을 가지도록 하고 있다. dummy.cs에서는 Attributes값으로 0을 주었고, Vcc로 50(5Volts), memory와 I/O를 다 사용하도록 설정해 주었다. Attributes가 가질 수 있는 값으로는 다시 아래와 같은 것이 있을 수 있다.

심벌	값	설명
CONF_ENABLE_IRQ	0x01	RequestIRQ에 대한 이전의 호출에 의해서 예약된 I/O interrupt를 enable 시킨다.
CONF_ENABLE_DMA	0x02	이 socket에 대해서 DMA access를 enable 시킨다.
CONF_ENABLE_SPKR	0x04	이 socket으로 부터의 speaker output을 enable 시킨다.
CONF_VALID_CLIENT	0x100	유효한 client를 가지고 있다고 설정한다.

표 78. Attributes 필드 값에 대한 정의

ConfigBase가 가지는 값은 다시 아래와 같이 구분해 볼 수 있다. 여기서 보여주는 값은 현재 존재하는 configuration register로 생각하면된다. 또한, 이 값에 따라 그 아래에 정의된 값들(Status, Pin, Copy, ExtStatus)의 사용 여부가 결정될 것이다.

심벌	값	설명
PRESENT_OPTION	0x001	Configuration Option Register(COR)가 있다는 것을 명시한다. COR은 ConfigIndex를 사용해서 설정된다.

²⁹⁷ 여기에 덧붙여서 INT_ZOOMED_VIDEO도 정의되어 있으며, 이것은 video를 위한 인터페이스를 설정하는 것으로 보인다.

PRESENT_STATUS	0x002	Card Configuration과 Status Register(CCSR)가 있다는 것을 명시한다. CCSR은 Status를 사용해서 초기화 된다.
PRESENT_PIN_REPLACE	0x004	Pin Replacement Register(PRR)가 있다는 것을 명시한다. PRR은 Pin을 사용해서 초기화된다.
PRESENT_COPY	0x008	Socket and Copy Register(SCR)가 있다는 것을 명시한다. SCR은 Copy를 사용해서 초기화된다.
PRESENT_EXT_STATUS	0x010	Extended Status Register(ESR)이 있다는 것을 명시한다. ESR은 ExtStatus를 사용해서 초기화된다.
PRESENT_IOPORT_0	0x020	기본 I/O 레지스터 0이 있다는 것을 명시한다.
PRESENT_IOPORT_1	0x040	기본 I/O 레지스터 1이 있다는 것을 명시한다.
PRESENT_IOPORT_2	0x080	기본 I/O 레지스터 2가 있다는 것을 명시한다.
PRESENT_IOPORT_3	0x100	기본 I/O 레지스터 3이 있다는 것을 명시한다.
PRESENT_IOSIZE	0x200	기본 I/O 레지스터의 한계를 나타내는 I/O Limit 레지스터가 있나는 것을 의미한다.

표 79. config_reg_t 구조체의 ConfigBase의 값에 대한 정의

위에서 열거한 레지스터들은 모두 설정 레지스터들로서 attribute 메모리 공간에 매핑되어 있다고 볼 수 있다. 이러한 레지스터에 대해서 접근하게 됨으로써, driver는 자신이 control하고 있는 디바이스가 어떤 특성을 가지고 있는지를 파악할 수 있다.

이전 Card Service에 등록하는 절차이다. link->next필드를 dev_list를 가르키도록 만들고, 다시 dev_list는 link를 가지도록 한다. 이렇게 하는 것으로 현재 등록된 모든 device에 대한 list에 이 device를 연결시키도록 하고, 이하에서는 client_reg_t 구조체를 만들어서, RegisterClient() Card Service를 호출하도록 한다.

```
/* For RegisterClient */
typedef struct client_reg_t {
    dev_info_t *dev_info; /* socket과 function을 등록하려는 client와 비교하기 위해서 사용한다.*/
    u_int Attributes; /* 등록하려는 attribute를 말한다.*/
    u_int EventMask; /* 등록하려는 client가 받게될 event에 대한 mask값.*/
    int (*event_handler)(event_t event, int priority,
                        event_callback_args_t *); /* Event를 처리하는 함수에 대한 포인터 */
    event_callback_args_t event_callback_args; /* Event handler에 대한 argument */
    u_int Version; /* Card Service version level을 표시 */
} client_reg_t;
```

코드 731. client_reg_t 구조체의 정의

client_reg_t 구조체는 앞에서 이미 한번 보았을 것이다. 여기서 다시 조금 더 자세히 보도록 하겠다. RegisterClient는 client driver와 Card Service간의 link를 설정하고, client를 적절한 socket과 연결하는 일을 한다. 만약 호출이 성공적이라면, client의 핸들을 돌려받을 것이다. Attribute가 가지는 값으로는 아래와 같은 것이 있을 수 있다.

심벌	값	설명
INFO_MASTER_CLIENTS	0x01	Driver Services client만이 사용하는 것으로 이 client가 이 socket에서 card가 제거될 때 unbound되어서는 안된다는 것을 표시한다. 즉, Driver Services 역할을 하는 client를 계속 남겨둔다.
INFO_IO_CLIENT	0x02	이 client가 I/O card driver라는 것을 의미한다.
INFO_MTD_CLIENT	0x04	이 client가 MTD(Memory Technology Driver)라는 것을 의미한다.
INFO_MEM_CLIENT	0x08	이 client가 Memory Card Driver라는 것을 의미한다.
INFO_CARD_SHARE	0x10	호환성 때문에 가지고 있는 것으로 아무런 의미가 없다.

INFO_CARD_EXCL	0x20	호환성 때문에 가지고 있는 것으로 아무런 의미가 없다.
----------------	------	--------------------------------

표 80. client_reg_t 구조체의 Attributes값 정의

또한 최대로 가질 수 있는 client의 갯수는 MAX_NUM_CLIENTS로 3이라는 값을 가지고 있다. dummy_cs에서는 info 필드로 client_reg_t 구조체의 info채워주고, Attribute로는 INFO_IO_CLIENT와 INFO_CARD_SHARE를 OR시켜주었으며, EventMask로 CS_EVENT_CARD_INSERTION, CS_EVENT_CARD_REMOVAL, CS_EVENT_RESET_PHYSICAL, CS_EVENT_CARD_REST, CS_EVENT_PM_SUSPEND, CS_EVENT_PM_RESUME을 OR 시켜서 사용하도록 했다. 즉, 앞에서 우리가 본 event들에 대해서 dummy_cs가 관심이 있으니, 그러한 event들이 발생하면, 우리가 등록한 client driver에 등록된 event handler를 호출해 달라는 것이다. Event handler로는 dummy_event()를, 그리고 Card Services의 version은 0x0210을 주었다. 현재는 이 버전이 아무런 효력을 가지고 있지 않으므로 신경을 쓰지 않아도 될 것이다.²⁹⁸ Event handler에 대한 넘겨줄 argument값은 link를 가르키도록 만들어 주었고, 최종적으로 CardServices()함수를 호출해서 client driver를 등록하도록 한다. 이 함수의 복귀값이 CS_SUCCESS가 아니라면, 잘못된 호출이므로, cs_error()를 호출해서 어떤 error가 있었는지를 알리고, 다시 dummy_detach()를 호출해서 앞에서 연결된 device list에서 이 driver를 위한 device 구조체를 제거하도록 만든다. 호출이 실패했을 경우의 복귀값은 NULL이고, 성공적이었다면, dev_list_t 구조체인 link를 돌려주도록 한다.

이전 이것으로 dummy_cs client driver의 attach() 함수에 대한 이야기를 마치도록 하고, 반대되는 일을 해주는 detach() 함수를 보도록 하자. 정의는 dummy_detach()함수를 참조하도록 한다.

```
static void dummy_detach(dev_link_t *link)
{
    dev_link_t **linkp;

    DEBUG(0, "dummy_detach(0x%p)\n", link);
    /* Locate device structure */
    for (linkp = &dev_list; *linkp; linkp = &(*linkp)->next)
        if (*linkp == link) break;
    if (*linkp == NULL)
        return;
    /*
     * If the device is currently configured and active, we won't
     * actually delete it yet. Instead, it is marked so that when
     * the release() function is called, that will trigger a proper
     * detach().
     */
    if (link->state & DEV_CONFIG) {
#ifdef PCMCIA_DEBUG
        printk(KERN_DEBUG "dummy_cs: detach postponed, '%s' "
               "still locked\n", link->dev->dev_name);
#endif
        link->state |= DEV_STALE_LINK;
        return;
    }
    /* Break the link with Card Services */
    if (link->handle)
        CardServices(DeregisterClient, link->handle);
    /* Unlink device structure, and free it */
    *linkp = link->next;
    /* This points to the parent local_info_t struct */
    kfree(link->priv);
} /* dummy_detach */
```

²⁹⁸ cs.c에 Card Services의 version으로 이 코드에서 참조하는 것으로는 1.274로 되어 있다.

코드 732. dummy_detach() 함수의 정의

먼저 dummy_detach() 함수가 넘겨받는 값은 제거할 link가 될 것이다. 즉, link의 데이터 타입은 제거할 dev_link_t 구조체에 대한 포인터이다. 이렇게 넘겨받는 dev_link_t 구조체와 같은 것을 현재 연결된 디바이스들에 대한 list를 검사해서 같은 값을 찾도록 한다(dev_list를 검색). 만약 같은 값을 찾을 수 없다면, 여기서 바로 return하도록 한다. 이전 제거할 dev_link_t 구조체를 찾았으므로, 이 디바이스를 제거할 수 있는지를 확인하기 위해서 state를 검사한다. 만약 DEV_CONFIG으로 설정되어 있다면, 이미 설정(configure)되어 있으며, active하다는 것이므로, 이것을 바로 제거하지 않고 return한다. 이때, state를 DEV_STALE_LINK(잘못된 link이다.)²⁹⁹와 OR시켜서 나중에 release() 함수에서 다시 detach()가 호출될 수 있도록 만든다.

만약 핸들러가 등록되어 있다면, DeregisterClient Card Services를 호출해서 client의 등록을 해제하고, 현재의 link를 dev_list에서 제거한다. 또한 할당받은 private 데이터를 위한 구조체를 해제해 준다(kfree())²⁹⁹. DeregisterClient Card Services는 RegisterClient와 반대되는 일을 하는 Card Services 함수로 등록된 client의 핸들(client_handler_t)을 CardServices() 함수의 인자를 필요로 한다. 이것은 앞에서 이미 등록시에 자동으로 설정된 것으로 link->handle에 RegisterClient함수에 의해서 설정된 것이다.

13.3. dummy_cs의 Release

dummy_cs의 release함수는 dummy_release() 함수이다. 이 함수는 앞에서 이미 attach() 하는 함수에서 client driver와 같이 등록되었다. 즉, 이 client에 대한 release를 하기 위해서 나중에 이 함수를 자동적으로 호출해 줄 것이다. 이 함수는 card가 제거(eject)될 때 shutdown processing의 일환으로 schedule되어 실행되는 일종의 timer 함수이며, DEV_PRESENT상태가 reset되고난 후에 바로 실행될 것이다.

```
static void dummy_release(u_long arg)
{
    dev_link_t *link = (dev_link_t *)arg;

    DEBUG(0, "dummy_release(0x%p)\n", link);
    /*
     * If the device is currently in use, we won't release until it
     * is actually closed, because until then, we can't be sure that
     * no one will try to access the device or its data structures.
     */
    if (link->open) {
        DEBUG(1, "dummy_cs: release postponed, '%s' still open\n",
              link->dev->dev_name);
        link->state |= DEV_STALE_CONFIG;
        return;
    }
    /* Unlink the device chain */
    link->dev = NULL;
    /*
     * In a normal driver, additional code may be needed to release
     * other kernel data structures associated with this device.
     */
    /* Don't bother checking to see if these succeed or not */
    if (link->win)
        CardServices(ReleaseWindow, link->win);
    CardServices(ReleaseConfiguration, link->handle);
    if (link->io.NumPorts1)
        CardServices(ReleaseIO, link->handle, &link->io);
    if (link->irq.AssignedIRQ)
        CardServices(ReleaseIRQ, link->handle, &link->irq);
}
```

²⁹⁹ 뒤에서 다시 event를 처리하는 곳에서 좀더 자세히 볼 수 있을 것이다. 여기서는 단지 잘못된 link라는 것을 나타낸다고 생각하면 될 것이다.

```

link->state &= ~DEV_CONFIG;
if (link->state & DEV_STALE_LINK)
    dummy_detach(link);
} /* dummy_release */

```

코드 733. **dummy_release()** 함수의 정의

dummy_release() 함수가 넘겨받는 arg값은 앞에서 **attach()** 함수에서 등록한 **timer_list** 구조체인 **release**의 **data**가 된다. 따라서, **dev_link_t** 구조체의 포인터가 될 것이다. 이것을 이용해서 **link** 변수를 초기화 한다. **DEBUG()**은 단순히 어디에서 실행중인가를 알려줄 뿐이다.

만약 아직 **open**되지 않은 디바이스라면, 단순히 상태(**link->state**)에 **DEV_STALE_CONFIG**(아직 **configure**이 되지 않았거나, 잘못된 **configuration**을 가진다.)를 OR시켜주고 바로 **return**한다. **link**의 **dev** 필드는 **NULL**로 두어 디바이스들의 **chain**을 끊어주고, 사용하고 있는 메모리에 대한 **window**가 있다면, 이를 해제하기 위해서 **ReleaseWindow Card Services**를 호출한다. 또한 설정(**configuration**)을 해제하기 위해서는 **ReleaseConfiguration Card Services**를 호출해주고, 사용하고 있는 **port**가 있었다면, **ReleaseIO Card Services**를, 사용하고 있던 **IRQ**가 있었다면, **ReleaseIRQ Card Services**를 호출해서 다 해제하도록 한다. 이전 디바이스의 상태를 설정이 되지 않았다고 만들어 주기 위해서 **link->state**에 **~DEV_CONFIG**을 AND시켜주고, 만약 **link->state**가 **DEV_STALE_LINK**라면, **dummy_detach()**를 호출해서 처리를 다 마치지 못한 **detach()** 함수를 다시 호출해서 처리가 끝날 수 있도록 만들어준다.

여기서 다시 **Card Services**의 **release**와 관련된 **service**들을 잠시 보도록 하자. 즉, **ReleaseWindow**, **ReleaseConfiguratioin**, **ReleaseIO**, **ReleaseIRQ** 등이 될 것이다.

13.3.1. RequestIO & ReleaseIO

먼저 앞에서 본 **ReleaseIO**부터 보도록 하자. **RequestIO**와 관련된 설명은 이어서 하도록 하겠다. 중요한 것은 현재 code에 있는 내용이기에, 그렇게 정했다.

```
int CardService(ReleaseIO, client_handle_t client, io_req_t *req);
```

위와 같이 함수의 원형이 주어지며, 예약(reserve)한 I/O port window 영역을 un-reserve하는 역할을 수행한다. 앞에서 이미 **RequestIO**를 했다는 가정하에 **release**하기 위해서 호출되며, **req**는 반드시 **RequestIO**에 전달된 것과 동일한 것을 가르켜야만 할 것이다. 만약 여러 card의 function³⁰⁰들이 큰 I/O port window를 공유하고 있다면, 하나의 function에 의해서 **release**된 port는 모든 card의 function들이 I/O port window를 **release**하기 전에는 다른 card에서 사용할 수 없을 수도 있다. **CardBus**의 **client**인 경우에는 이 함수가 card에 대해서 할당된 모든 시스템의 resource를 해제해 줄 것이다. 예러 코드로는 다음과 같은 값을 가질 수 있다.

- **CS_BAD_HANDLE** : 잘못된 **client** 핸들을 가지고 호출했다.
- **CS_CONFIGURATION_LOCKED** : 이 소켓의 **configuration**은 **RequestConfiguration** 호출에 의해서 lock이 된 상태이다. 따라서, **configuration**이 **ReleaseIO**를 호출하기 전에 **release**가 먼저 되어야 한다.
- **CS_BAD_ARGS** : **req**에 전달된 파라미터 값이 **RequestIO**에서 사용했던 값과 일치하지 않는다.

다시, 여기서 조금 더 볼 부분은 **RequestIO**에서 사용하고 있는 **io_req_t** 구조체의 값이다. 아래와 같은 정의를 가지고 있으며, **RequestIO**와의 관계는 아래에서 설명하겠다.

```

/* For RequestIO and ReleaseIO */
typedef struct io_req_t {
    ioaddr_t     BasePort1;          /* 기본(base) I/O port window address를 나타낸다. */
    ioaddr_t     NumPorts1;         /* 사용하는 port의 개수를 나타낸다. */
}

```

³⁰⁰ Multi-funtion으로 동작하는 card의 경우이다. 즉, 하나의 카드가 여러가지 다른 기능을 가지는 경우가 될 것이다.

```

u_int      Attributes1;      /* Port의 attribute를 나타낸다.*/
ioaddr_t   BasePort2;       /* NumPorts2가 0이 아닌 경우에 예약될 기본 I/O port window address
*/
ioaddr_t   NumPorts2;       /* 두번째 기본 I/O port window에서 사용되는 port의 개수 */
u_int      Attributes2;     /* 두번째 I/O port의 attribute */
u_int      IOAddrLines;    /* Card의 의해서 실제적으로 decoding이 되는 address line의 개수 */
} io_req_t;

```

코드 734. io_req_t 구조체의 정의

I/O port를 할당하는 algorithm은 하위의, IOAddrLines bits를 예약하는 요청된 address들의 어떠한 alias도 받아들이며, BasePort1과 BasePort2는 실제적으로 정해진 그러한 address range를 반영하도록 update시켜준다. 이것은 3.1.4버전 이전에는 IOAddrLines가 무시되었으며, allocator가 base address가 0가 아니라면 정확한 address range를 할당하려고 시도하고, 만약 0이라면 2의 승수에서 가장 가까운 보다 큰 값으로 사용가능한 window가 할당되었다. 새로운 allocator에서는 IOAddrLines 파라미터가 요청된 window 파라미터 값에 따르는지(agree)는 확인하고, 만약 일치하지 않는 점이 발견된다면 3.1.4의 default 할당 algorithm을 사용한다.

Multi-function으로 사용되는 card의 경우에는 각각의 card function에 대해서, 모든 카드의 port들이 두개의 물리적인 socket들과 관련된 하위 레벨 I/O port window들에 의해서 맵핑될 수 있도록, I/O port를 할당할 것이다.

RequestIO의 원형은 다음과 같다. 앞에서 본 io_req_t구조체는 req가 가르키도록 만든다. dummy_cs에서는 설정(configuration)에서 사용하고 있다.

```
int CardServices(RequestIO, client_handle_t client, io_req_t *req);
```

즉, 우리가 사용할 I/O를 위한 port를 요청하는 것이다. io_req_t 구조체의 포인터는 실제로 할당된 것을 나타내므로, 나중에 이것을 해제(release)하기 위해서 보관할 필요가 있을 것이다. 이것까지를 마치면, 우리는 메모리 상에 card를 위한 I/O port를 가지게 되므로, 이제부터는 그 port를 통해서 접근할 수 있을 것이다.³⁰¹

만약 CardBus client에 의해서 이 함수가 사용된다면 io_req_t 구조체는 무시될 것이며, 대신에 Card Services는 card를 검사해서 모든 필요한 시스템의 resource를 할당(예약)할 것이다. 이러한 자원에 해당하는 것으로는 I/O와 메모리 공간(space) 이외에 필요하다면 interrupt도 들어갈 것이다. 하나의 호출로서 카드의 모든 기능(function)을 위해서 필요한 모든 필요한 자원들이 예약될 것이다. 하지만, 여기서 주의할 점은 이 호출이 실제적으로 socket의 I/O window를 설정하는 것은 아니라는 점이다. 따라서, 설정(configure)하기 위해서는 추후에 **RequestConfiguration**을 사용해야만 할 것이다.

이곳에서 사용하는 io_req_t의 Attributes1과 Attributes2가 가질 수 있는 값으로는 IO_DATA_WITH_WIDTH라는 것이다. 이것은 16 bit 접근(access)과 8 bit 접근을 위해서 각각 IO_DATA_PATH_WIDTH_16이나, 혹은 IO_DATA_PATH_WIDTH_8을 가질 수 있으며, 접근 형태에 따른 동적인 크기를 주기위해서는 IO_DATA_PATH_WIDTH_AUTO가 될 수 있을 것이다. 이외에 가질 수 있는 값으로는 다음과 같은 것이 더 있다.

심벌	값	설명
IO_SHARED	0x01	공유된 I/O port를 사용한다.
IO_FIRST_SHARED	0x02	공유된 I/O port를 사용하지만, 한번에 하나의 드라이버만이 사용가능하다.
IO_FORCE_ALIAS_ACCESS	0x04	I/O port에 대한 alias access를 사용하도록 만든다. ³⁰²

³⁰¹ 물론 나중에 설정(configuration)하는 것을 필요로 하지만, 여기서는 이러한 설정까지는 고려하지 않기로 하자.

³⁰² 이 부분은 아직 명확하지가 않다. 찾게 되면, 좀더 자세히 논하도록 하겠다. 현재로서는 이러한 값을 사용할 수 있다고만 되어있다.

IO_DATA_PATH_WIDTH	0x18	기본적으로 사용하게 될 data path의 width를 나타낸다.
IO_DATA_PATH_WIDTH_8	0x00	8 bit data path의 width를 사용하도록 한다.
IO_DATA_PATH_WIDTH_16	0x08	16 bit data path의 width를 사용하도록 한다.
IO_DATA_PATH_WIDTH_AUTO	0x10	자동적으로 사용하게 될 data path의 width를 결정하도록 한다.

표 81. RequestIO와 ReleaseIO에서의 Attributes1과 Attributes2가 가지는 값의 정의

13.3.2. RequestIRQ & ReleaseIRQ

이것은 이미 앞에서 살펴본 부분이다. 여기서는 간략하게 사용하는 함수의 원형만 보기로 하자. 아래와 같다.

```
int CardServices(RequestIRQ, client_handle_t client, irq_req_t *req);
int CardServices(ReleaseIRQ, client_handle_t client, irq_req_t *req);
```

여기서도 주의할 점은 RequestIRQ에서 예약 받은 irq_req_t 구조체에 대한 포인터를 ReleaseIRQ에 넘겨서 사용해야 한다는 점이다.

13.3.3. RequestConfiguration & ReleaseConfiguration

RequestConfiguration과 ReleaseConfiguration에서 사용하는 자료구조는 앞에서 이미 보았던 config_req_t 구조체이다. 따라서, 이곳에서는 함수의 원형에 대해서만 보도록 하겠다.

```
int CardService(RequestConfiguration, client_handle_t client, config_req_t *req);
int CardService(ReleaseConfiguration, client_handle_t client, config_req_t *req);
```

RequestConfiguration은 실제적으로 socket을 설정(configure)하는 역할을 하며, 이와 반대 작용을 하는 것이 ReleaseConfiguration이다. 역시 config_req_t의 포인터인 req는 RequestConfiguration과 ReleaseConfiguration이 동일한 것을 사용해야 할 것이다.

13.3.4. RequestWindow & ReleaseWindow

PCMCIA에서 말하는 window란 card의 특정 부분이 host의 주소 공간에 맵핑(mapping)된 것을 말한다. 각각의 socket은 최대로 active한 메모리 window를 4개까지 가질 수 있으며, 이러한 메모리는 common과 attribute³⁰³로 다시 나누어지며, 각각이 16Mbytes를 가질 수 있다. Window는 반드시 2의 승수 크기를 가져야 하며, 소켓의 capability에 따라서 호스트와 card의 주소공간에서 window 크기의 배수 경계(boundary)에 정렬되어야 할 수도 있다.

이러한 PCMCIA card의 메모리 윈도우는 RequestWindow를 통해서 초기화되며, ModifyWindow를 가지고 window의 attribute를 바꿀 수 있다. 또한 MapMemPage는 window에 맵핑된 card 메모리 부분(segment)를 변경(modify)할 수 있으며, 마지막으로 맵핑을 해제하기 위해서 ReleaseWindow를 사용한다. 여기서, 이러한 Card Services들을 사용하는데 있어서 중요한 파라미터 값이 되는 것은 바로 window_handle_t 구조체로 정의된 window handle이라는 점이다. 우리가 현재 보고자 하는 것은 RequestWindow와 ReleaseWindow Card Services이다.

RequestWindow Card Services는 아래와 같은 정의를 가진다. 여기서 사용하는 win_req_t의 정의도 같이 보도록 하자.

```
int CardServices(RequestWindow, client_handle_t *handle, win_req_t *req);
```

```
/* For RequestWindow */
typedef struct win_req_t {
    u_int          Attributes;           /* Window request의 attribute값 */
    u_long         Base;                /* Host 시스템에서의 윈도우의 기본 물리주소 */
```

³⁰³ Common memory란 데이터 실행 파일을 가지는 메모리 array가 현재 working하는 주소 공간에 맵핑된 경우를 말하며, attribute memory란 설정 정보를 가지는 메모리로 사용되는 경우를 말한다.

```

u_int      Size;          /* Byte단위로 나타낸 원도우의 크기 */
u_int      AccessSpeed;   /* Nanosecond단위로 나타낸 메모리 접근 속도 */
} win_req_t;

```

코드 735. win_req_t 구조체의 정의

win_req_t 구조체의 필드 중에서 Attributes가 가질 수 있는 값으로는 다시 다음과 같은 것들이 있다. 물론 이 값들도 역시 ~/include/pcmcia/cs.h에서 찾을 수 있을 것이다.

심벌	값	설명
WIN_ADDR_SPACE	0x0001	원도우의 주소 공간의 특성을 나타낸다.(아래를 참조)
WIN_ADDR_SPACE_MEM	0x0000	원도우가 메모리 공간이다.
WIN_ADDR_SPACE_IO	0x0001	원도우가 I/O 공간이다.
WIN_MEMORY_TYPE	0x0002	어떤 메모리 타입을 가지는지 명시한다.(아래를 참조)
WIN_MEMORY_TYPE_CM	0x0000	Common 메모리 type을 가진다.
WIN_MEMORY_TYPE_AM	0x0002	Attribute 메모리 type을 가진다.
WIN_ENABLE	0x0004	이 bit이 설정되면, window를 enable(turn on) 시킨다.
WIN_DATA_WIDTH	0x0018	메모리에 대한 데이터 접근시의 크기를 말한다.(아래를 참조)
WIN_DATA_WIDTH_8	0x0000	8 bit 접근을 말한다.
WIN_DATA_WIDTH_16	0x0008	16 bit 접근을 말한다.
WIN_DATA_WIDTH_32	0x0010	32 bit 접근을 말한다.
WIN_PAGED	0x0020	원도우가 paging되어서 접근하도록 한다.
WIN_SHARED	0x0040	원도우가 공유가 가능하도록 만든다.
WIN_FIRST_SHARED	0x0080	원도우의 공유가 가능하지만 첫번째 매핑요구에 대해서 반응하도록 한다.
WIN_USE_WAIT	0x0100	Controller에게 card의 MWAIT 신호를 관찰하도록 만든다. ³⁰⁴
WIN_STRICT_ALIGN	0x0200	매핑될 원도우의 기본 주소가 원도우 크기의 배수에 정렬되도록 요구한다.
WIN_MAP_BELOW_1MB	0x0400	원도우가 1MB 주소 공간이내에 매핑되도록 요구한다. 이것은 특정 architecture에서는 사용할 수 없다.
WIN_PREFETCH	0x0800	원도우가 매핑된 주소 공간이 prefetch ³⁰⁵ 가 가능하다.
WIN_CACHEABLE	0x1000	원도우가 매핑된 주소 공간이 cacheable하다.
WIN_BAR_MASK	0xE000	원도우의 BAR(Base Address Register) mask 값을 나타낸다.
WIN_BAR_SHIFT	0x000D	원도우의 BAR(Base Address Register) shift 값을 나타낸다.

표 82. win_req_t의 Attributes 값의 정의

앞에서 만약 Base가 0인 값을 가진다면, Card Services는 Base를 첫번째 사용가능한 window 주소로 설정할 것이다. 만약 Size가 0인 값을 가진다면, 다시 Card Services는 host controller에서 지원하는 가장 작은 크기의 window 크기로 설정할 것이다.

여기서 한가지 주의할 점은 앞에서 RequestWindow Card Services에서 넘긴 client_handle_t의 *handle은 호출의 결과값을 넘겨 받을 때 window_handle_t으로 대체된다는 점이다. 나머지, window과 관련된 함수는 0이 handle값을 사용해서 연산을 수행해야 할 것이다.

ReleaseWindow의 정의는 다음과 같다. 이 Card Services가 하는 역할을 앞에서 RequestWindow가 했던 것의 반대 일을 하는 것으로 window의 매핑을 없앤다.

³⁰⁴ 일반적으로 사용하는 time동안에 card에 대한 data의 전송을 마칠 수 없다고 생각될 때, 이것을 사용해서 그 시간을 늘려줄 수 있다. 이때, controller가 이것을 관찰하도록 만들어서, 그 이후에 다른 전송이나 연산이 발생하도록 만들어주기 위한 것이다.

³⁰⁵ 이것은 performance를 높이기 위해서 미리 읽어오는 것을 말한다.

```
int CardServices(ReleaseWindow, window_handle_t handle);
```

RequestWindow에서 handle로 넘겨받은, 이미 앞에서 본 window_handle_t 구조체를 사용한다는 점에 주의하기 바란다.

13.4. dummy_cs의 Event 처리

이제 dummy_cs가 처리하게 되는 event에 대해서 알아보기로 하자. 이것은 dummy_cs가 등록시켜둔 dummy_event() 함수가 처리를 맡고 있다. 정의는 아래와 같다.

```
static int dummy_event(event_t event, int priority,
                      event_callback_args_t *args)
{
    dev_link_t *link = args->client_data;
    local_info_t *dev = link->priv;

    DEBUG(1, "dummy_event(0x%06x)\n", event);

    switch (event) {
        ...
    }
    return 0;
} /* dummy_event */
```

코드 736. dummy_event() 함수의 정의

Switch() 문의 내용은 아래에서 살펴볼 것이며, 앞에서 이미 한번 보았지만, dummy_event() 함수가 처리하는 event는 주로 card의 물리적인 동작에 관련된 것으로 다음과 같은 표로 다시한번 정리해 보도록 하자.

Event	설명
CS_EVENT_CARD_REMOVAL	이 event는 카드가 제거(removal) 되었음을 알려준다. 이 event는 Card Service가 모든 client에 가능한 한 빨리 알려줄 수 있도록, 최소 delay를 가지도록 처리되어야 할 것이다.
CS_EVENT_CARD_INSERTION	이 event는 카드가 insert되었다는 것을 알려준다. 만약 드라이버가 이미 점유된 socket에 bind가 된 상태라면, Card Service가 드라이버에 인위적인 insertion event를 전달해 줄 것이다.
CS_EVENT_PM_SUSPEND	이 event는 Card Service가 사용자나 APM(Advanced Power Management)로부터 시작된 suspend 요청을 받았다는 것을 알려준다. Event 핸들러는 failure를 돌려줌으로써 거부의사를 표현할 수 있다.
CS_EVENT_RESET_PHYSICAL	이 event는 모든 client들에 전달되는 것으로 card에 reset 신호가 전달되기 바로 전이 될 것이다.
CS_EVENT_PM_RESUME	이 event는 시스템이 다시 on-line상태로 전환되었다는 것을 의미하는 것으로 suspend와 resume의 순환 관계를 가진다.
CS_EVENT_CARD_RESET	이 event는 reset 연산이 끝났다는 것을 알려준다. Reset의 성공이나 실패는 반드시 GetStatus()를 통해서 결정되어야 한다.

표 83. dummy_cs의 Card Service event에 대한 정의

코드를 설명하면, 먼저 전달된 argument로 부터 client driver가 사용하는 데이터 구조를 얻기 위해서 event_callback_arg_t의 client_data부분을 가져온다. 이것으로 dev_list_t 구조체에 대한 포인터를 획득하고, 다시 이것을 이용해서 드라이버에서 사용하는 private data영역인 local_info_t 구조체의 포인터를 얻는다. 이전 switch() 문을 통해서 전달받은 event를 처리하도록 한다. 처리하는 event는 위에서 정의한 표와 같은 것들이다. 각각에 대해서 살펴보도록 하자.

13.4.1. CS_EVENT_CARD_REMOVAL & CS_EVENT_CARD_INSERTION event의 처리

```
case CS_EVENT_CARD_REMOVAL:
    link->state &= ~DEV_PRESENT;
    if (link->state & DEV_CONFIG) {
        ((local_info_t *)link->priv)->stop = 1;
        mod_timer(&link->release, jiffies + HZ/20);
    }
break;
```

코드 737. dummy_cs의 CS_EVENT_CARD_REMOVAL event의 처리

CS_EVENT_CARD_REMOVAL은 카드가 제거될 때 발생되는 event이다. 따라서, 디바이스의 상태 변경이 필요하며, 디바이스 드라이버의 release() 함수를 호출해서 카드가 제거되었다는 것을 반영해 주도록 한다. 먼저 상태에 대한 정의는 ds.h에 아래와 같이 되어 있다.

```
/* Flags for device state */
#define DEV_PRESENT          0x01      /* 현재 카드가 존재한다.*/
#define DEV_CONFIG            0x02      /* 카드에 대한 설정이 되어 있다.*/
#define DEV_STALE_CONFIG      0x04      /* 현재의 설정이 잘못되었다. Close시에 release한다.*/
#define DEV_STALE_LINK         0x08      /* 현재 Link가 잘못되었다. Release시에 detach한다.*/
#define DEV_CONFIG_PENDING     0x10      /* 현재 카드의 설정(configuration)이 pending되어 있다.*/
#define DEV_RELEASE_PENDING     0x20      /* 현재 카드의 release에서 pending된 상태이다.*/
#define DEV_SUSPEND             0x40      /* 현재 디바이스가 suspending된 상태이다.*/
#define DEV_BUSY                0x80      /* 현재 디바이스가 요청을 처리하고 있는 중이다.(busy)
*/
```

코드 738. PCMCIA 디바이스 상태에 대한 설정값의 정의

앞에서 보인 코드에서는 CS_EVENT_CARD_REMOVAL이 발생하면, state에서 DEV_PRESENT bit을 clear하고, 만약 state가 DEV_CONFIG으로 디바이스가 configuration이 된 상태라면, 디바이스를 stop시키기 위해서 stop bit을 1로 설정해 주며, release함수가 수행되도록 timer를 재설정한다. 여기서 주의할 점은 바로 release함수를 호출하는 것이 아니라, 어느정도 시간의 delay를 준다는 점이다.

```
case CS_EVENT_CARD_INSERTION:
    link->state |= DEV_PRESENT | DEV_CONFIG_PENDING;
    dev->bus = args->bus;
    dummy_config(link);
break;
```

코드 739. dummy_cs의 CS_EVENT_CARD_INSERTION event의 처리

CS_EVENT_CARD_INSERTION event는 카드가 socket에 삽입될 때 발생하는 event이다. 따라서, 이때서야 비로소 카드의 정보를 읽어서 실제적인 설정(configuration)을 할 수 있는 정보를 알게된다. 먼저 state를 DEV_PRESENT와 DEV_CONFIG_PENDING으로 두어, 현재 카드가 있지만 설정 중이라는 것을 알린다. 그리고나서, argument로 받은 args의 bus field로부터 card가 사용하는 bus ID를 기억한다. 마지막으로 dummy_config() 함수를 호출해서 실제적인 card의 설정에 들어가게 된다.

13.4.2. CS_EVENT_PM_SUSPEND & CS_EVENT_PM_RESUME event의 처리

```
case CS_EVENT_PM_SUSPEND:
    link->state |= DEV_SUSPEND;
    /* Fall through... */
...
case CS_EVENT_PM_RESUME:
```

```
link->state &= ~DEV_SUSPEND;
/* Fall through... */
```

코드 740. dummy_cs의 PM(Power Management) Event의 처리

dummy_cs는 실제 하드웨어의 구성을 가지고 있지 않으므로, 여기서는 단순히 state값 만을 DEV_SUSPEND로 설정했다가 다시 이를 지워주는 것으로 끝내고 있다. 하지만, 만약 실제 하드웨어를 가진다면 이야기는 달라진다. 즉, 만약 하드웨어가 power management와 관련된 spec.0이 있다면, 여기서 이것을 제어해 주어야 할 것이다. Network card나 기타 디바이스에서는 power와 관련된 레지스터를 가진 경우가 많으므로 이 레지스터를 적절히 여기서 설정해 주면 될 것이다.

13.4.3. CS_EVENT_RESET_PHYSICAL & CS_EVENT_CARD_RESET event의 처리

```
case CS_EVENT_RESET_PHYSICAL:
    /* Mark the device as stopped, to block IO until later */
    dev->stop = 1;
    if (link->state & DEV_CONFIG)
        CardServices(ReleaseConfiguration, link->handle);
    break;
...
case CS_EVENT_CARD_RESET:
    if (link->state & DEV_CONFIG)
        CardServices(RequestConfiguration, link->handle, &link->conf);
    dev->stop = 0;
    /*
     * In a normal driver, additional code may go here to restore
     * the device state and restart IO.
     */
    break;
```

코드 741. dummy_cs의 reset event의 처리

물론 dummy_cs는 물리적인 디바이스를 가지고 있을리가 없다. 따라서, 물리적으로 reset이 가해지는 일을 없을 것이다. CS_EVENT_RESET_PHYSICAL event에서는 단지 이 디바이스에 대해서 I/O를 하려는 것이 있다면, 이것을 멈추게 할 목적으로 stop bit을 1로 설정하고, 디바이스가 현재 reset이 되었으므로 기존에 가지고 있던 configuration을 새로이 읽어야 하기 때문에, ReleaseConfiguration Card Services를 호출하는 것으로 끝내도록 했다.

CS_EVENT_CARD_RESET은 reset 연산이 무사히 잘 진행되었다는 것을 알려주는 것이다. 만약 디바이스가 아직 설정(configuration) 상태를 유지하고 있다면(DEV_CONFIG), 이것을 새롭게하기 위해서 RequestConfiguration Card Services를 호출한다. 그리고, 다시 I/O를 진행 가능한 상태로 만들도록 하기 위해서 stop bit을 0으로 바꾸어 주었다. 즉, 모든 reset이 끝나고, 다시 동작을 시작하는 것이라고 보면 될 것이다.

한가지 덧붙이고 싶은 것은, 위의 코드에도 있듯이 실제 하드웨어를 제어하는 디바이스 드라이버라면, 당연히 관련된 하드웨어에 대한 직접적인 접근이 있어야 할 것이다.

13.5. dummy_cs의 Error Reporting

CardService가 제공하는 error report 사용하기 위해서 dummy_cs는 cs_error()이라는 함수를 사용하며, 아래와 같은 정의를 가지도록 만든다.

```
static void cs_error(client_handle_t handle, int func, int ret)
{
    error_info_t err = { func, ret };
    CardServices(ReportError, handle, &err);
}
```

코드 742. cs_error() 함수의 정의

이 함수는 단순히 CardService()에 ReportError을 호출해서 에러 상황을 알려주는 역할만 하고 있다. 여기서 사용되는 데이터 구조체로는 error_info_t 구조체가 있다. 아래와 같다.

```
typedef struct error_info_t {
    int         func;           /* Error를 일으킨 함수 */
    int         retcode;        /* Error 값 */
} error_info_t;
```

코드 743. error_info_t 구조체의 정의

error_info_t 구조체는 에러를 일으킨 Card Services의 함수와 관련된 에러코드로 정의된다. 즉, 이와 같은 값을 주어서, Card Service에 어떤 error가 발생했는가를 알려주는데 사용한다. 나중에도 에러 상황을 만나면, 이와같은 데이터 구조체를 이용해서 Card Services에 알려주게 되므로 잘 알고 넘어가도록 하자.

13.6. dummy_cs의 Configuration

dummy_cs의 card 설정을 맡고 있는 함수는 dummy_config()이다. 이 함수에서는 card의 설정 정보를 알기 위해서 CIS(Card Information Structure)에 대한 접근을 해야하며, 읽은 정보를 기초로해서 I/O port 및 IRQ, 메모리 window등의 설정을 동반할 것이다. 이 함수의 정의는 아래와 같다.

```
static void dummy_config(dev_link_t *link)
{
    client_handle_t handle = link->handle;
    local_info_t *dev = link->priv;
    tuple_t tuple;
    cisparse_t parse;
    int last_fn, last_ret;
    u_char buf[64];
    config_info_t conf;
    win_req_t req;
    memreq_t map;
    cistpl_cftable_entry_t dflt = { 0 };

    DEBUG(0, "dummy_config(0x%p)\n", link);
    /*
     * This reads the card's CONFIG tuple to find its configuration
     * registers.
     */
    tuple.DesiredTuple = CISTPL_CONFIG;
    tuple.Attributes = 0;
    tuple.TupleData = buf;
    tuple.TupleDataMax = sizeof(buf);
    tuple.TupleOffset = 0;
    CS_CHECK(GetFirstTuple, handle, &tuple);
    CS_CHECK(GetTupleData, handle, &tuple);
    CS_CHECK(ParseTuple, handle, &tuple, &parse);
    link->conf.ConfigBase = parse.config.base;
    link->conf.Present = parse.config.rmask[0];
    /* Configure card */
    link->state |= DEV_CONFIG;
    /* Look up the current Vcc */
    CS_CHECK(GetConfigurationInfo, handle, &conf);
    link->conf.Vcc = conf.Vcc;
```

코드 744. dummy_config() 함수의 정의

dummy_config() 함수가 전달 받는 것은 dev_link_t 구조체이다. 이 구조체에 연산에 적용될 client driver에 대한 handle이 있으므로, 이것을 구해서 handle에 넣도록 한다. 또한 이 드라이버에서 private data structure로 사용하고 있는 local_info_t 구조체에 대한 포인터를 역시 dev가 가르키도록 한다. 또한 읽어올 CIS tuple을 값을 가질 tuple과 CIS tuple을 해석(parsing)하는데 사용하기 위한 parse 및 error reporting에 사용할 last_fn과 last_ret를 정의한다. 임시 버퍼로 사용하는 buf와 configuration에 사용할 conf, window 설정에 사용하는 req, 메모리 설정에 사용할 map도 정의하고 있다. 또한 CIS tuple의 한 entry를 값을 가지는 dflt로 이곳에서 정의한다.

먼저 DEBUG()으로 현재 어느 routine을 진행하고 있는지를 보여주고, card의 configuration tuple을 읽기 위한 준비를 한다. 여기서 잠시 tuple_t 구조체의 정의를 보기로 하자.

```
typedef struct tuple_t {
    u_int          Attributes;      /* 어떤 tuple을 사용하는지에 대한 속성(attribute)를 설정한다.*/
    cisdata_t      DesiredTuple;   /* 일치되는 tuple을 찾는 곳에서 사용한다.*/
    u_int          Flags;          /* internal use */
    u_int          LinkOffset;     /* internal use */
    u_int          CISOffset;      /* internal use */
    cisdata_t      TupleCode;      /* Tuple의 code값이다.*/
    cisdata_t      TupleLink;      /* Tuple의 link값을 명시한다.*/
    cisdata_t      TupleOffset;    /* Tuple이 있는 attribute 메모리의 offset 값을 명시한다.*/
    cisdata_t      TupleDataMax;   /* Tuple이 가질 수 있는 최대 데이터 크기를 명시한다.*/
    cisdata_t      TupleDataLen;   /* Tuple이 가지는 실제 데이터의 길이를 나타낸다.*/
    cisdata_t      *TupleData;      /* Tuple의 실제 데이터를 가르킨다.*/
} tuple_t;
```

코드 745. tuple_t 구조체의 정의

이와 같이 정의된 tuple_t 구조체는 GetFirstTuple, GetNextTuple, GetTupleData와 같은 Card Service를 호출하는데 인자로서 넘겨지게 된다. Attributes가 가지는 값들로는 아래와 같은 것이 있을 수 있겠다.

심벌	값	설명
TUPLE_RETURN_LINK	0x01	이것은 link tuple(CISTPL_LONGLINK_A, CISTPL_LONGLINK_C, CISTPL_LONGLINK_MFC, CISTPL_NOLINK, CISTPL_LINKTARGET)들을 돌려주도록 요구한다. 일반적으로는 이러한 tuple들은 내부적으로 사용되며, client driver에서는 사용할 일이 없을 것이다.
TUPLE_RETURN_COMMON	0x02	이것은 multifunction CIS의 common CIS section에 있는 tuple들을 돌려주어야 한다는 것을 나타낸다. 일반적으로 이 flag가 설정되지 않았다면, Card Service는 단순히 현재의 client에 bind된 function에 해당하는 tuple만을 돌려줄 것이다.

표 84. tuple_t 구조체의 Attributes field 값의 정의

코드에서는 요구하고 있는 tuple이 CISTPL_CONFIG(설정과 관련된 tuple)이며, Attributes로는 0을 주어 일반적인 tuple만을 가져오려고 하며, tuple의 데이터가 저장될 곳으로 buf를 TupleData가 가르키게 만들었다. 또한 최대로 가져오고자 하는 tuple data의 크기는 buf의 크기이므로 TupleDataMax는 sizeof(buf)로 설정하고, 첫번째로 가져오고자 하는 tuple이므로 offset은 0으로 두었다. 이젠 tuple을 가져오는 일을 하기 위해서 CS_CHECK(GetFirstTuple, handle, &tuple)을 사용한다. CS_CHECK()는 아래와 같은 macro로 정의되어 있다.

```
#define CS_CHECK(fn, args...) \
while ((last_ret=CardServices(last_fn=(fn),args))!=0) goto cs_failed

#define CFG_CHECK(fn, args...) \
if (CardServices(fn, args) != 0) goto next_entry
```

코드 746. dummy_cs의 CardServices() 함수의 매크로 정의

따라서, CS_CHECK()는 단순히 CardServices()의 내용 역할을 하므로, GetFirstTuple을 호출한 것과 동일하다. Tuple을 가져온 후에 이전 데이터를 얻기 위해서 GetTupleData를 호출했으며, 얻은 tuple data를 parse하기 위해서 ParseTuple을 호출했다.

```
typedef union cisparse_t {
    cistpl_device_t           device;
    cistpl_checksum_t          checksum;
    cistpl_longlink_t          longlink;
    cistpl_longlink_mfc_t     longlink_mfc;
    cistpl_vers_1_t            version_1;
    cistpl_altstr_t            altstr;
    cistpl_jedec_t             jedec;
    cistpl_manfid_t            manfid;
    cistpl_funcid_t             funcid;
    cistpl_funce_t              funce;
    cistpl_bar_t                  bar;
    cistpl_config_t              config;
    cistpl_cftable_entry_t      cftable_entry;
    cistpl_cftable_entry_cb_t   cftable_entry_cb;
    cistpl_device_geo_t         device_geo;
    cistpl_vers_2_t              vers_2;
    cistpl_org_t                  org;
    cistpl_format_t               format;
} cisparse_t;
```

코드 747. cisparse_t union의 정의

cisparse_t union을 보면 알 수 있듯이 이것은 원하는 정보를 parse하고 난 다음의 결과를 가진다는 것을 볼 수 있을 것이다. 또한 각각의 원하는 정보에 따라서, cisparse_t union도 달리 정해진다는 것이다. 여기서, 모든 cistpl_XXX_t 구조체를 보기보다는 필요한 것만을 보도록 하겠다. 일단 우리가 원하는 정보는 configuration에 관련된 것에 한정되므로, cistpl_config_t 구조체가 이에 해당한다.

```
typedef struct cistpl_config_t {
    u_char      last_idx;        /* 가장 마지막에 있는 configuration table의 entry에 대한 index 값 */
    u_int       base;            /* Attribute memory내에서의 configuration register의 위치 */
    u_int       rmask[4];         /* 어떤 configuration register가 있는지를 나타내는 bit mask 값 */
    u_char      subtuples;       /* 일반적인 tuple에 뒤따라 나오는 subtuple의 byte 수 */
} cistpl_config_t;
```

코드 748. cistpl_config_t 구조체의 정의

cistpl_config_t 구조체에 있는 subtuples은 card의 설정과 관련된 부가적인 정보를 주기 위한 tuple을 말하는 것으로 vendor에 고유하거나 PCMCIA의 확장을 위해서 존재하는 것이다.

다시 앞의 이야기로 돌아가서, dummy_cs에서는 가져온 정보를 이용해서 설정(configuration)을 요청하기 위해서 link의 config_req_t에 해당하는 conf 필드의 ConfigBase에 cisparse_t의 config에 있는 base를, Present에는 config에 있는 rmask[0]를 주었다. 이전 card에 대한 설정에 들어가게 되므로, state를 DEV_CONFIG으로 바꾸도록 한다. 이것은 기본적인 configuration register의 위치와 어떤 configuration register들이 있는지를 알게되었다.

GetConfigurationInfo CardService는 현재 사용중인 설정을 구하기 위해서 사용하는 Card Services이다. 이 함수에서 사용하는 자료 구조체는 config_info_t 구조체로서 아래와 같은 정의를 가진다.

```
/* for GetConfigurationInfo */
```

```

typedef struct config_info_t {
    u_char      Function;          /* Card의 function에 대한 index */
    u_int       Attributes;        /* 요청의 attributes */
    u_int       Vcc, Vpp1, Vpp2;   /* Voltage 설정 값 */
    u_int       IntType;          /* Interface type */
    u_int       ConfigBase;        /* Configuration register의 기본 주소 */
    u_char      Status, Pin, Copy, Option, ExtStatus; /* 사용되는 parameter 값 : configuration register */
    u_int       Present;          /* 존재 여부를 나타내는 flag */
    u_int       CardValues;        /* 어떤 configuration register들이 있는가를 표시 */
    u_int       AssignedIRQ;       /* 할당된 IRQ */
    u_int       IRQAttributes;     /* IRQ의 attribute */
    ioaddr_t   BasePort1;         /* Base port 1의 주소 */
    ioaddr_t   NumPorts1;         /* Port 1의 개수 */
    u_int       Attributes1;       /* Port 1의 attribute */
    ioaddr_t   BasePort2;         /* Base port2의 주소 */
    ioaddr_t   NumPorts2;         /* Port 2의 개수 */
    u_int       Attributes2;       /* Port 2의 attribute */
    u_int       IOAddrLines;       /* I/O address line */
} config_info_t;

```

코드 749. config_info_t 구조체의 정의

GetConfigurationInfo가 돌려주는 것은 현재 socket의 설정 정보로서, RequestIO, RequestIRQ, RequestConfiguration에 의해서 설정된 것이다. 대부분의 필드들은 socket이 완전히 설정된 상황에서만 채워지며, 이러한 것을 나타내 줄 수 있는 것이 Attributes 필드가 CONFIG_VALID_CLIENT을 가지는 것이다. 하나의 function에 bind된 일반적인 client driver인 경우에는 Function 필드가 무시되며(ignore), 그 client driver에 대한 데이터를 돌려준다. 만약 client driver가 BIND_FN_ALL로 bind된 경우에는, Function 필드가 어떤 function의 configuration을 돌려줄 것인가를 명시하도록 하고 있다.

config_info_t 구조체의 CardValues 필드가 가질 수 있는 값은 아래와 같이 정의 할 수 있다.³⁰⁶ 어떤 값을 가지는지에 따라서, configuration register에 대한 사용이 달라질 것이다.

심벌	값	설명
CV_OPTION_VALUE	0x01	Configuration Option Register를 위한 값
CV_STATUS_VALUE	0x02	Configuration and Status Register를 위한 값
CV_PIN_REPLACEMENT	0x04	Pin Replacement Register를 위한 값
CV_COPY_VALUE	0x08	Socket and Copy Register를 위한 값
CV_EXT_STATUS	0x10	Extended Status Register를 위한 값

표 85. config_info_t 구조체의 CardValues 필드가 가지는 값의 정의

따라서, 코드에서는 GetConfigurationInfo를 호출해서 Vcc값으로 client driver가 제어하는 디바이스의 현재 Vcc값을 설정해 주었다.

이후부터의 코드는 loop를 들어가게 된다. 이 loop에서는 CIS(Card Information Structure)를 검사해서 configuration table entry들을 하나씩 보게될 것이다. 만약 이러한 entry들이 유효(valid)하다면, 각각이 Voltage, I/O window, 메모리 window, 및 interrupt에 대한 설정(configuration) 정보를 가질 것이다. 일반적으로 client driver들은 자신이 현재 control하게 될 카드의 configuration 설정사항들을 대부분은 알고

³⁰⁶CardValues 필드는 현재는 documentation이 되지 않은 필드이지만, 가지는 값으로 어떤 역할을 하는지 일단 추측에 근거해서 보겠다.

있기 때문에, 자신이 구현하고 싶어하는 역할만을 수행하기 위한 정보만을 CIS로부터 얻어오면 된다. 이곳에서 보여주고자 하는 것은 단지 CIS 정보를 어떻게 이용하는가이다.

```
tuple.DesiredTuple = CISTPL_CFTABLE_ENTRY;
CS_CHECK(GetFirstTuple, handle, &tuple);
while (1) {
    cistpl_cftable_entry_t *cfg = &(parse.cftable_entry);
    CFG_CHECK(GetTupleData, handle, &tuple);
    CFG_CHECK(ParseTuple, handle, &tuple, &parse);
    if (cfg->flags & CISTPL_CFTABLE_DEFAULT) dflt = *cfg;
    if (cfg->index == 0) goto next_entry;
    link->conf.ConfigIndex = cfg->index;
```

코드 750. `dummy_config()` 함수의 정의(계속)

찾고자 하는 tuple이 원지를 나타내는 tuple_t 구조체의 DesiredTuple을 CISTPL_CFTABLE_ENTRY로 두고, GetFirstTuple Card Services를 호출한다. 즉, 첫번째 CISTPL_CFTABLE_ENTRY에 해당하는 tuple을 읽어오라는 말이다. 이후부터는 loop를 진행하면서, 원하는 CIS tuple을 계속 찾는 것이다.

cistpl_cftable_entry_t 구조체는 configuration table의 entry를 가지는 것으로 card에 의해서 제공되는 설정 option을 준다. 각각의 table entry는 부가적인 설정 option을 제공할 수 있으며, 전체 CIS내에 있는 configuration entry를 합쳐서 configuration table이라고 한다. 아래와 같이 정의되어 있다.

```
typedef struct cistpl_cftable_entry_t {
    u_char           index;          /* 현재의 동작 모드에 대한 설정 index값을 준다.
                                         이 값을 card의 Configuration Option Register에 쓰는 것으로
                                         이 동작 모드를 설정한다. */
    u_short          flags;          /* 구현하고 있는 기능들에 대한 flag값을 가진다. */
    u_char           interface;      /* 인터페이스 타입(memory, I/O and memory) */
    cistpl_power_tvcc vpp1, vpp2;   /* Power 관련 설정 */
    cistpl_timing_t  timing;        /* Timing 관련 설정 */
    cistpl_io_t       io;            /* I/O 관련 설정 */
    cistpl_irq_t      irq;           /* IRQ 관련 설정 */
    cistpl_mem_t      mem;           /* Memory 관련 설정 */
    u_char           subtuples;     /* subtuple */ } cistpl_cftable_entry_t;
```

코드 751. `cistpl_cftable_entry_t` 구조체의 정의³⁰⁷

`cistpl_cftable_entry_t` 구조체의 flags 필드가 가질 수 있는 값으로는 다시 다음과 같은 것들이 있다. 이들 값들은 PCMCIA standard에서 정의된 CISTPL_CFTABLE_ENTRY의 interface와 misc 필드에 영향을 미치도록 되어있다.

심벌	값	설명
CISTPL_CFTABLE_DEFAULT	0x0001	이것이 기본(default) configuration table entry라는 것을 표시한다.
CISTPL_CFTABLE_BVDS	0x0002	이 configuration은 BVD1과 BVD2 신호(signal) ³⁰⁸ 을 Pin Replacement Register에서 구현하고 있다는 것을 표시한다.

³⁰⁷ CIS tuple에 대한 정의를 일일이 다 나열하려면, 너무 많은 지면이 할애가 되어야 한다. 여기서는 간략하게 어떤 것들이 있는지를 위주로 보고, 자세한 것은 PCMCIA Standard를 보도록 해야 할 것이다. 현재 PCMCIA Standard는 copyright가 있는 것이기에 특별히 개발을 담당하는 engineer가 아니라면, 일반 개인이 사서 보기에는 조금 무리가 있다.

³⁰⁸ BVD(Battery Voltage Detect)signal은 battery를 가진 card상에서 BVD의 내부 상태를 나타내기 위해서 사용된다.

CISTPL_CFTABLE_WP	0x0004	이 configuration은 write protect 신호를 Pin Replacement Register ³⁰⁹ 에서 구현하고 있다는 것을 표시한다.
CISTPL_CFTABLE_RDYBSY	0x0008	이 configuration은 Ready/Busy 신호를 Pin Replacement Register에서 구현하고 있다는 것을 표시한다.
CISTPL_CFTABLE_MWAIT	0x0010	메모리에 대한 접근 cycle동안 WAIT 신호를 관찰해야 한다는 것을 나타낸다. ³¹⁰
CISTPL_CFTABLE_AUDIO	0x0800	이 configuration은 host의 speaker로 가는 audio 신호를 만들어 낸다는 것을 표시한다.
CISTPL_CFTABLE_READONLY	0x1000	Card가 이 configuration에 읽기 전용(read-only) 메모리 공간을 가진다는 것을 표시한다.
CISTPL_CFTABLE_PWRDOWN	0x2000	이 configuration이 Card Configuration과 Status Register를 통해서 power down mode를 지원하다는 것을 표시한다.

표 86. cistpl_cftable_entry_t 구조체의 flags 필드 값의 정의

이전에 읽어온 tuple의 데이터를 해독(parse)해서 이를 ParseTuple Card Services를 사용해서 cisparse_t 구조체로 변형한다. cisparse_t 구조체에는 이미 CISTPL_CFTABLE_ENTRY를 가지는 필드가 있으며, 현재 cfg가 이것을 가르키고 있으므로, parse된 데이터에 CISTPL_CFTABLE_DEFAULT인 경우 dflt에 cfg가 가르키는 값을 넣어주도록 한다. 만약 cistpl_cftable_entry_t 구조체의 index 필드가 0이라면, 다음 entry로 jump한다(goto next_entry). Index가 0이 아니라면, 현재의 configuration index값을 디바이스의 configuration index(link->conf.ConfigIndex)로 설정해서 기억하도록 한다.

```
/* Does this card need audio output? */
if (cfg->flags & CISTPL_CFTABLE_AUDIO) {
    link->conf.Attributes |= CONF_ENABLE_SPKR;
    link->conf.Status = CCSR_AUDIO_ENA;
}
/* Use power settings for Vcc and Vpp if present */
/* Note that the CIS values need to be rescaled */
if (cfg->vcc.present & (1<<CISTPL_POWER_VNOM)) {
    if (conf.Vcc != cfg->vcc.param[CISTPL_POWER_VNOM]/10000)
        goto next_entry;
} else if (dflt.vcc.present & (1<<CISTPL_POWER_VNOM)) {
    if (conf.Vcc != dflt.vcc.param[CISTPL_POWER_VNOM]/10000)
        goto next_entry;
}
if (cfg->vpp1.present & (1<<CISTPL_POWER_VNOM))
    link->conf.Vpp1 = link->conf.Vpp2 =
        cfg->vpp1.param[CISTPL_POWER_VNOM]/10000;
else if (dflt.vpp1.present & (1<<CISTPL_POWER_VNOM))
    link->conf.Vpp1 = link->conf.Vpp2 =
        dflt.vpp1.param[CISTPL_POWER_VNOM]/10000;
/* Do we need to allocate an interrupt? */
if (cfg->irq.IRQInfo1 || dflt.irq.IRQInfo1)
    link->conf.Attributes |= CONF_ENABLE_IRQ;
```

코드 752. dummy_config() 함수의 정의(계속)

³⁰⁹ Pin Replacement Register(PPR)은 일반적으로 메모리 interface의 상태 변화를 일으키는 event에 사용하는 HBA(Host Bus Adapter) function을 대체하는 역할을 하는 것으로, I/O event와 관련된 특정 interface 신호를 만들어내도록 한다.

³¹⁰ 일반적으로 PCMCIA가 메모리에 접근(access)하는 시간이 프로그램된 timing안에 반응하지 못하는 경우, 이러한 시간을 늘려주기 위해서 사용하는 것이다. 즉, WAIT 시그널이 있는 동안에는 메모리에 대한 읽기나 쓰기를 wait한다.

만약 configuration table entry(cfg)의 flags에 CISTPL_CFTABLE_AUDIO가 있다면, host system의 speaker에 대한 audio output이 있다는 말이되므로, 디바이스의 설정(conf)에 있는 Attributes에 CONF_ENABLE_SPKR을 추가한다(OR). 또한 Status 필드에는 CCSR_AUDIO_ENA로 두어서, audio가 enable되어어야 한다는 것을 표시한다.

CCSR(Card Configuration and Status Register)에 설정될 수 있는 값으로는 아래와 같은 것들이 있다. 이 값에 대한 정의는 ~/include/pcmcia/cisreg.h를 보기 바란다. 실제로 여기서 보여주는 값들은 전부 CCSR에서 하나의 bit position을 차지하고 있다. 중간에 reserve된 bit을 포함해서 CCSR은 실제로 8 bit register의 형식을 취하고 있다.

심벌	값	설명
CCSR_INTR_ACK	0x01	만약 이 bit이 설정되면(set), CCSR_INTR_PENDING bit은 software에 의해서 clear될 때까지 set된 형태로 남아있게 된다. 즉, 반드시 software에서 ACK를 해주도록 요청하기 위해서 사용한다.
CCSR_INTR_PENDING	0x02	Card가 인터럽트 요청을 하고 있다는 것을 말한다. 이것은 interrupt 공유시에 도움을 주기 위한 것이다. 즉, software들이 pending된 인터럽트를 보고, 자신을 위한 interrupt인지를 확인한 후, 만약 자신이 처리해야 할 interrupt라면 처리를 마친후, CCSR_INTR_ACK와 같은 bit을 clear 시켜주면 될 것이다. 그렇지 않다면, 다른 interrupt handling software로 제어를 넘겨주도록 한다.
CCSR_POWER_DOWN	0x04	이 bit을 설정하는 것은 card가 반드시 power down 상태로 전이해야 한다는 것을 나타내주는 것이다.
CCSR_AUDIO_ENA	0x08	이 bit은 card의 audio output이 반드시 enable되어야 한다는 것이다.
CCSR_IOIS8	0x20	Host에 의해서 사용되며, 8 bit I/O 연산만을 할 수 있다는 것을 나타낸다. 따라서, 16 bit 접근(access)인 경우에는 두번의 8 bit 접근으로 수행된다.
CCSR_SIGCHG_ENA	0x40	WP(Write Protect), READY, BVD1, BVD2와 같은 신호의 변화를 card가 SIGCHG 신호를 사용해서 나타내 주어야 한다는 것을 의미한다.
CCSR_CHANGED	0x80	PRR(Pin Replacement Register)에 있는 어떤 bit의 상태가 변하면, host에 이것을 알려주도록 설정한다.

표 87. Card Configuration and Status Register의 값 정의

이전 power 설정에 대한 부분을 보기로 하자. 코드에서는 Vcc와 Vpp에 대한 power 설정을 다루고 있다. 여기서 한가지 주의할 점은 tuple에서 정해진 설정(cfg)이 없다면, 앞에서 구한 default configuration(dflt)를 사용한다는 것이다. 따라서, 코드에서는 cfg를 먼저 보고, 이곳에 정해진 것이 없는 경우에만 dflt를 사용해서 설정하고 있다.

cistpl_power_t 구조체는 CIS tuple에서 power와 관련된 설정을 나타낸다. 정의는 아래와 같으며, cistpl_cftable_entry_t 구조체의 한 필드를 차지 한다.

```
typedef struct cistpl_power_t {
    u_char      present; /* 01 power signal에 대해서 어떤 parameter가 존재하는지를 나타내는 bitmap */
    u_char      flags;   /* 정의를 찾을 수 없다. 표준에서도 이 필드에 대한 것은 찾을 수 없다. */
    u_int       param[7]; /* 앞에서 present bitmap이 존재여부를 나타낸 parameter들의 array */
} cistpl_power_t;
```

코드 753. cistpl_power_t 구조체의 정의

present필드가 가질 수 있는 값으로는 다음의 표와 같은 것이 있다. 이 값이 하나의 entry에 대한 index의 역할을 한다.

심벌	값	설명
CISTPL_POWER_VNOM	0	명목상(nominal) 공급 voltage parameter field가 있다.
CISTPL_POWER_VMIN	1	최소 공급 voltage parameter field가 있다.
CISTPL_POWER_VMAX	2	최대 공급 voltage parameter field가 있다.
CISTPL_POWER_ISTATIC	3	연속적인(continuous) 공급 current parameter field가 있으며, 필요로 한다.
CISTPL_POWER_IAVG	4	1초동안으로 평균한 최대 current parameter field가 있다.
CISTPL_POWER_IPEAK	5	10 ms동안으로 평균한 최대 current parameter field가 있다.
CISTPL_POWER_IDOWN	6	Power down mode에서 필요한 current parameter field가 있다.

표 88. `cistpl_power_t` 구조체의 `present`필드의 값 정의

따라서, 설정(configuration)값의 Vcc를 구하기 위해서는 CISTPL_POWER_VNOM이 `present`에 있는지를 확인하고, 이 값이 있다면, 이것을 index로 해서 `param[]` 필드³¹¹로 접근해 얻어온다. 얻어온 값은 다시 voltage의 경우에는 10 microvolts로 나타내지며, current의 경우에는 100 nanoamperes로 표기된다. 만약 이렇게 구한 Vcc 값을 10000으로 나누어서, 원래 있던 Vcc값(`conf`의 Vcc 필드)과 일치하지 않는다면 다음 entry로 이동한다. 마찬가지로 Vpp1에 대해서도 CISTPL_POWER_VNOM이 있는지를 확인하고, 앞에서 이미 Vcc에 대한 값이 일치한다고 확인하였기에, 디바이스의 Vpp1값과 Vpp2값을 설정해 주도록 한다.

이전 interrupt를 설정할 차례이다. `cfg`나 `dflt`의 `IRQInfo1` 필드에 어떤 값이 있다면, interrupt를 enable시켜야 한다는 말이 되므로, 디바이스의 configuration attributes에 `CONF_ENABLE_IRQ` bit를 설정하도록 한다.

```
typedef struct cistpl_irq_t {
    u_int      IRQInfo1;          /* IRQ information 1 */
    u_int      IRQInfo2;          /* IRQ information 2 */
} cistpl_irq_t;
```

코드 754. `cistpl_irq_t` 구조체의 정의

`IRQInfo1` 필드가 가질 수 있는 값으로는 아래와 같은 것이 있으며, `IRQInfo1`이 0이라면 interrupt에 대한 정보는 사용할 수 없게 된다. 또한 `IRQ_INFO2_VALID` 필드가 설정된 경우에 대해서만 `IRQInfo2`는 허가된 interrupt 요청 번호에 대한 유효한 bit mask값을 가질 수 있도록 하고 있다.

심벌	값	설명
IRQ_MASK	0x0F	Card가 반드시 사용해야 하는 interrupt 번호를 명시한다.
IRQ_NMI_ID	0x01	만약 <code>IRQ_INFO2_VALID</code> 가 설정된 경우에 card에 지정될 수 있는 특별한 interrupt 신호를 나타내기 위해서 사용되며, 아래의 3개 값도 같이 사용된다. 여기서는 non-maskable interrupt ID를 나타낸다.
IRQ_IOCK_ID	0x02	I/O check를 나타낸다.
IRQ_BERR_ID	0x04	Bus error를 나타낸다.
IRQ_VEND_ID	0x08	Vendor가 지정한 interrupt를 나타낸다.
IRQ_INFO2_VALID	0x10	<code>IRQInfo2</code> 가 유효한 interrupt 요청 번호로 유효한 bit mask를 가지고 있다는 것을 표시한다.
IRQ_LEVEL_ID	0x20	Card가 level mode interrupt를 지원한다는 것을 나타낸다.
IRQ_PULSE_ID	0x40	Card가 pulse mode interrupt를 지원한다는 것을 나타낸다.
IRQ_SHARE_ID	0x80	Card가 interrupt 공유를 지원한다는 것을 나타낸다.

표 89. `cistpl_irq_t` 구조체의 `IRQInfo1` 필드 값의 정의

³¹¹ Parameter는 실제로 2 bytes로 표기되며, extension, mantissa, exponent로 구성되어 있다. Standard를 참조하기 바란다.

여기서, level mode와 pulse mode라는 말이 나오는데, 기본적으로 16-bit PC card의 경우에는 level mode interrupt를 지원하도록 design되어야 하며, pulse mode는 옵션으로 지정될 수 있다. Level mode에서는 interrupt가 발생했음을 알려주는 IRQ# pin이 Vcc로 끌어올려져서 인터럽트가 일어났음을 알려주게되는 것이며, 인터럽트는 ISR(Interrupt Service Routine)이 interrupt indication을 재설정(reset)하지 않으며, 계속 assert된 상태로 남아있게 된다. Pulse mode는 level mode가 인터럽트 공유를 하지 못한다는 것을 보완하기 위한 것으로 인터럽트가 발생했음을 알려주기 위해서 신호의 transition이 일어날 경우에만 interrupt를 발생시켜주는 것이다.

```
/* IO window settings */
link->io.NumPorts1 = link->io.NumPorts2 = 0;
if ((cfg->io.nwin > 0) || (dflt.io.nwin > 0)) {
    cistpl_io_t *io = (cfg->io.nwin) ? &cfg->io : &dflt.io;
    link->io.Attributes1 = IO_DATA_PATH_WIDTH_AUTO;
    if (!(io->flags & CISTPL_IO_8BIT))
        link->io.Attributes1 = IO_DATA_PATH_WIDTH_16;
    if (!(io->flags & CISTPL_IO_16BIT))
        link->io.Attributes1 = IO_DATA_PATH_WIDTH_8;
    link->io.IOAddrLines = io->flags & CISTPL_IO_LINES_MASK;
    link->io.BasePort1 = io->win[0].base;
    link->io.NumPorts1 = io->win[0].len;
    if (io->nwin > 1) {
        link->io.Attributes2 = link->io.Attributes1;
        link->io.BasePort2 = io->win[1].base;
        link->io.NumPorts2 = io->win[1].len;
    }
    /* This reserves IO space but doesn't actually enable it */
    CFG_CHECK(RequestIO, link->handle, &link->io);
}
}
```

코드 755. dummy_config() 함수의 정의(계속)

I/O window를 설정하기 위해서 디바이스의 NumPorts1과 NumPorts2를 0으로 초기화 하도록 한다. 이하에서 관련된 field를 설정할 것이다. 새로 읽어들인 configuration tuple과 default의 cistpl_io_t 구조체의 nwin이 0 이상의 값을 가진 경우에만 if 이하의 절을 수행한다. cistpl_io_t 구조체를 들어가기에 앞서 보도록 하자. 아래와 같은 정의를 가진다.

```
typedef struct cistpl_io_t {
    u_char      flags;           /* I/O window에 대한 flag */
    u_char      nwin;            /* I/O window의 개수 */
    struct {
        u_int     base;           /* Window의 base address */
        u_int     len;             /* Window의 길이(length ) */
    } win[CISTPL_IO_MAX_WIN];   /* 각각의 I/O window를 위한 array */
} cistpl_io_t;
```

코드 756. cistpl_io_t 구조체의 정의

cistpl_io_t 구조체의 flags 필드가 가질 수 있는 값은 아래와 같이 정의된다. 또한 nwin는 아래에 오는 필드에 얼마나 많은 I/O window가 있는지 알려주는 값이다.

심벌	값	설명
CISTPL_IO_LINES_MASK	0x1F	Card에 의해서 decode되는 I/O line의 수를 나타낸다.
CISTPL_IO_8BIT	0x20	Card가 16-bit I/O register에 대한 8-bit split access를 지원한다.
CISTPL_IO_16BIT	0x40	Card가 I/O register에 대한 full 16-bit access를 지원한다.
CISTPL_IO_RANGE	0x80	Card의 I/O 공간에 대한 range 정의가 올지 안올지를 나타낸다.

		만약 range가 설정된 경우에는 I/O를 위한 address와 length를 나타내는 것이 CIS tuple에서 뒤따라 온다는 것을 나타낸다. 여기서는 사용할 필요가 없을 것이다.
CISTPL_IO_MAX_WIN	16	flags이 가지는 값이 아니며, 최대 I/O window의 개수를 나타낸다.

표 90. cistpl_io_t 구조체의 flags 필드 값의 정의

코드에서는 nwin이 0이상의 값을 가지면, 이 값을 사용하도록 하고 있다. 즉, I/O를 위한 window가 존재하는지를 검사하는 것이다. 만약 0 이상이라면, io를 cfg나 dflt의 cistpl_io_t 구조체를 가르키도록 하고, 기본적으로 Data path에 대한 설정을 auto(IO_DATA_PATH_WIDTH_AUTO)로 둔 다음 각 설정에 맞게 이것을 변경한다. 즉, 8-bit access를 지원하지 않는 경우에는(CISTPL_IO_8BIT) IO_DATA_PATH_WIDTH_16을, 16-bit access를 지원하지 않는 경우에는(CISTPL_IO_16BIT) IO_DATA_PATH_WIDTH_8을 I/O를 위한 Attributes1에 지정했다. 또한 사용하는 address line에는 CISTPL_IO_LINES_MASK를 flags와 AND한 값으로 주었으며, 첫번째 port의 기본 주소로 win[0].base를, 첫번째 port의 개수로는 win[0].len을 각각 주었다. 즉, 기본 주소와 access 할 수 있는 공간의 크기를 알려주는 것이다. 또한 nwin이 1 이상인 값을 가지는 경우에는 Attribute2와 BasePort2, NumPorts2도 지정하도록 했다. Attributes2는 Attributes1과 동일하도록 만들었다. 즉, I/O window의 각 window의 attributes는 동일하게 설정하고 있다. 이젠 이렇게 구한 window를 실제 시스템에 사용한다고 등록할 차례로서 RequestIO를 Card Services를 호출하도록 한다. 넘겨주는 값은 앞에서 설정한 I/O window를 위한 값이다.³¹²

```

if ((cfg->mem.nwin > 0) || (dflt.mem.nwin > 0)) {
    cistpl_mem_t *mem =
        (cfg->mem.nwin) ? &cfg->mem : &dflt.mem;
    req.Attributes = WIN_DATA_WIDTH_16|WIN_MEMORY_TYPE_CM;
    req.Attributes |= WIN_ENABLE;
    req.Base = mem->win[0].host_addr;
    req.Size = mem->win[0].len;
    if (req.Size < 0x1000)
        req.Size = 0x1000;
    req.AccessSpeed = 0;
    link->win = (window_handle_t)link->handle;
    CFG_CHECK(RequestWindow, &link->win, &req);
    map.Page = 0; map.CardOffset = mem->win[0].card_addr;
    CFG_CHECK(MapMemPage, link->win, &map);
}
/* If we got this far, we're cool! */
break;
next_entry:
if (link->io.NumPorts1)
    CardServices(ReleaseIO, link->handle, &link->io);
    CS_CHECK(GetNextTuple, handle, &tuple);
}

```

코드 757. dummy_config() 함수의 정의(계속)

이전 common memory window에 대한 설정(setup)을 해줄 차례이다. dev_link_t 구조체를 보면, 메모리 window에 대한 필드로 window_handle_t 구조체를 정의하고 있으며, 단지 하나의 원도우에 대한 정보만을 가지도록 하고 있다. 따라서, 만약 여러개의 window를 필요로 한다던가, 기본 address(base address)를 저장하기를 원한다면, 이것들은 전부 private data 구조체에 저장되어야 할 것이다³¹³. 한가지 더 주의할

³¹² RequestIO Card Services를 참조하도록 하라. 앞에서 이미 dummy_cs의 Release에서 보았었다.

³¹³ dummy_cs에서는 local_info_t 구조체와 같은 곳에 이것을 위한 필드를 정의해서 사용할 수 있을 것이다.

점은 여기서 말하는 base address는 전부 physical address를 의미하므로, 디바이스 드라이버에서 이것을 사용하기 위해서는, 먼저 가상 주소 공간으로 mapping을 한다음에 해야할 것이다. 이를 위해서 ioremap()과 같은 함수를 사용할 수 있다.

cistpl_mem_t 구조체를 보도록 하자. cistpt_mem_t 구조체는 메모리 윈도우를 요청하기 위한 자료구를 채우는데 사용한다.

```
#define CISTPL_MEM_MAX_WIN 8

typedef struct cistpl_mem_t {
    u_char      flags;          /* 메모리 윈도우의 flag 값314 */
    u_char      nwin;           /* 메모리 윈도우의 개수 */
    struct {
        u_int      len;            /* 메모리 윈도우의 크기(길이) */
        u_int      card_addr;      /* 메모리 윈도우의 card 메모리 공간(space)에서의 주소 */
        u_int      host_addr;      /* 메모리 윈도우의 host 메모리 공간(space)에서의 주소 */
    } win[CISTPL_MEM_MAX_WIN];   /* 메모리 윈도우를 나타내는 array */
} cistpl_mem_t;
```

코드 758. cistpl_mem_t 구조체의 정의

nwin값이 0이상이면 하나 이상의 메모리 윈도우에 대한 description이 존재하므로, common 메모리에 대한 설정에 들어가도록 한다. 먼저 사용하게될 메모리 윈도우 맵에 대한 포인터로 mem을 cfg가 있을 경우에는 그곳에서 가져오고, 그렇지 않을 경우에는 default로 주어진 dflt에서 가져와서 초기화 한다. 메모리 윈도우에 대한 요청(request)을 하기 위해서 win_req_t 구조체인 req를 설정한다. 먼저 Attributes에는 data width를 16(WIN_DATA_WIDTH_16)과 common 메모리 임을 나타내는 WIN_MEMORY_TYPE_CM을 서령하고, 다시 이 메모리를 활성화 시키기 위해서 WIN_ENABLE bit을 ON 시켜준다. 메모리 윈도우가 사용할 base address에는 win[0].host_addr을 주고, 다시 그 길이에는 win[0].len을 주도록 한다. 만약 요청하는 메모리 윈도우의 크기가 0x1000(=4K)보다 작다면, 4K로 옮겨주도록 한다. 즉, 최소 4Kbytes 영역을 맵핑해서 쓰도록 만들어 주는 것이다. 현재로서는 AccessSpeed를 0으로 설정해서 하위 시스템에서 알아서 메모리에 대한 access speed를 설정하도록 했다. 이전 card의 메모리를 host의 메모리 공간에 mapping하기 위해서 RequestWindow Card Services를 호출한다. 앞에서 만든 요청(win_req_t) 구조체도 같이 넘겨주도록 한다.

이것으로 메모리에 대한 mapping은 생성되었으며, 실제적인 메모리에 대한 접근전에 메모리 윈도우의 기본 주소로 mapping된 card memory의 주소를 설정하기 위해서 MapMemPage Card Service를 호출한다. 이곳에서 사용하는 자료 구조로는 memreq_t 구조체가 있으며, 아래와 같은 정의를 가진다.

```
typedef struct memreq_t {
    u_int      CardOffset; /* 메모리 윈도우의 기본 주소에 mapping된 card 메모리주소가 설정되는 곳 */
    page_t     Page;       /* 현재로서는 사용되지 않고, 단순히 0인 값을 가지도록 한다. */
} memreq_t;
```

코드 759. memreq_t 구조체의 정의

이렇게 mapping된 메모리 윈도우의 현재의 card address mapping은 다시 GetMemPage Card Services 호출을 사용해서 다시 얻어올 수 있다. 코드에서는 매팅을 생성하기 위해서 win[0].card_addr 필드를 CardOffset으로 주어서, 첫번째 메모리 윈도우에 대한 mapping을 재설정하고 있다. 만약 여기까지 제대로 진행했다면, 더 이상의 loop는 돌지않고, 밖으로 나오도록 한다(break).

³¹⁴ 현재로서는 이 field에 대한 사용을 찾을 수 없다. 단순히 memory window에 대한 flag이며, standard에서도 찾을 수 없었다.

next_entry label이 하는 다음번 tuple을 entry에 대한 것을 요청하는 부분이다. 만약 앞에서 I/O port에 대한 것을 이미 요구한 상태라면, 이것을 해제하기 위해서 ReleaseIO Card Services를 호출하게 되며, 또한 다음번 tuple을 얻기 위해서 GetNextTuple Card Services도 호출한다. 제어(control)은 while() loop를 다시 돌게 된다.

여기까지 해서, loop의 순환은 끝나게 되며, 사용하게 될 메모리에 대한 mapping도 마치게 되었다. 이제 남은 것은 IRQ를 할당하는 것과 마지막으로 앞에서 구한 정보를 이용해서 configuration을 정하는 일이다.

```
/*
 * Allocate an interrupt line. Note that this does not assign a
 * handler to the interrupt, unless the 'Handler' member of the
 * irq structure is initialized.
 */
if (link->conf.Attributes & CONF_ENABLE_IRQ)
    CS_CHECK(RequestIRQ, link->handle, &link->irq);
/*
 * This actually configures the PCMCIA socket -- setting up
 * the I/O windows and the interrupt mapping, and putting the
 * card and host interface into "Memory and IO" mode.
 */
CS_CHECK(RequestConfiguration, link->handle, &link->conf);
/*
 * We can release the IO port allocations here, if some other
 * driver for the card is going to loaded, and will expect the
 * ports to be available.
 */
if (free_ports) {
    if (link->io.BasePort1)
        release_region(link->io.BasePort1, link->io.NumPorts1);
    if (link->io.BasePort2)
        release_region(link->io.BasePort2, link->io.NumPorts2);
}
```

코드 760. dummy_config() 함수의 정의(계속)

Interrupt line을 할당하도록 한다. 할당은 앞에서 구한 conf_info_t 구조체의 Attributes에 CONF_ENABLE_IRQ가 설정된 경우에 한하며, 이때 interrupt를 enable시키도록 요구했으므로, RequestIRQ Card Services를 사용한다. 넘겨주는 변수로는 client 핸들과 앞에서 설정한 irq_req_t 구조체³¹⁵이다. 이젠 모든 설정 정보에 대한 것을 다 얻어왔으므로 최종적으로 RequestConfiguration Card Services를 호출할 수 있게 되었다. 이것은 실제로 socket을 설정하는 호출이며, voltage, CIS configuration register, I/O port window 및 interrupt에 대한 설정을 동반한다.

free_ports 변수는 다른 driver가 load되어서, 이 드라이버가 사용하고 있는 port가 사용 가능하다고 기대할 경우를 대비해서 미리 이곳에서 port를 release하기 위해서 사용하는 전역 변수이다. Default 값으로 dummy_cs에서는 0으로 설정하고 있지만, module이 load될 때 parameter로 명시할 수 있도록 하고 있다. 만약 0이 아닌 값을 가진다면, 할당된 region을 해제하기 위해서 release_region() 커널 함수를 사용해서 영역을 제거하도록 한다. BasePort1과 BasePort2에 대해서 각각 한번씩 해주도록 한다.

```
/*
 * At this point, the dev_node_t structure(s) need to be
 * initialized and arranged in a linked list at link->dev.
```

³¹⁵ Attach와 config에서 약간씩 그 설정을 이미 해 주었다. 여기서는 이것을 사용하기 위해서 시스템에 등록한다고 생각하면 될 것이다.

```

*/
sprintf(dev->node.dev_name, "skel0");
dev->node.major = dev->node.minor = 0;
link->dev = &dev->node;
/* Finally, report what we've done */
printk(KERN_INFO "%s: index 0x%02x: Vcc %d.%d",
       dev->node.dev_name, link->conf.ConfigIndex,
       link->conf.Vcc/10, link->conf.Vcc%10);
if (link->conf.Vpp1)
    printk(", Vpp %d.%d", link->conf.Vpp1/10, link->conf.Vpp1%10);
if (link->conf.Attributes & CONF_ENABLE_IRQ)
    printk(", irq %d", link->irq.AssignedIRQ);
if (link->io.NumPorts1)
    printk(", io 0x%04x-0x%04x", link->io.BasePort1,
          link->io.BasePort1+link->io.NumPorts1-1);
if (link->io.NumPorts2)
    printk("& 0x%04x-0x%04x", link->io.BasePort2,
          link->io.BasePort2+link->io.NumPorts2-1);
if (link->win)
    printk(", mem 0x%06lx-0x%06lx", req.Base,
          req.Base+req.Size-1);
printk("\n");
link->state &= ~DEV_CONFIG_PENDING;
return;
cs_failed:
cs_error(link->handle, last_fn, last_ret);
dummy_release((u_long)link);
} /* dummy_config */

```

코드 761. dummy_config() 함수의 정의(계속)

사용하게될 디바이스의 이름은 skel0로 node의 dev_name 필드에 두도록 한다. 디바이스의 major와 minor 번호는 각각 0으로 초기화 하고, 이젠 디바이스를 연결하기 위해서 link->dev 필드에 dev->node의 주소를 넣는다. 이하는 앞에서 설정한 값들의 정보를 화면이나 커널의 log으로 만들어주기 위해서 printk()를 호출하는 부분이다. 설정을 마치게 되므로, 디바이스의 상태는 이제 DEV_CONFIG_PENDING이 아니므로 이것을 state의 bit에서 지워주도록 한다. 여기까지 했으면, dummy_config() 함수는 복귀하게 될 것이다. 만약 Card Service에 대한 호출에서 실패(failure)가 있었다면, cs_failed label 이하를 실행하게 된다. 먼저 cs_error를 호출해서 가장 마지막에 호출한 function과 return값을 넘겨주고, dummy_release()를 호출해서 device에 대한 해제를 시도하도록 한다.

우리는 앞에서 PCMCIA에 대한 것을 아주 가볍하게만 다루었다. 실제로 현재 나와있는 Linux의 PCMCIA driver에 대한 것으로는 유일한 것이 “pcmcia programming guide”라는 것이 있지만, 별로 programming에 대한 도움은 주지 못하는 것 같다. 하지만, 단순히 Card Services만을 사용하는 경우에는 충분한 정보를 줄 수 있다고 생각한다. 나머지는 열심히 여러 책을 공부하고, 이미 있는 코드를 참조하는 수 밖에 없으리라 생각한다. 이곳에서 보여준 것은 그야말로 맞보기에 불과하기에 조금 미안하지만, 코드를 열심히 읽으라는 말 밖에는 하지 못하겠다. 또한 embedded환경에서 초기화시에 PCMCIA를 접근해서 자신이 원하는 드라이버를 만든다면, 여기 있는 자료는 별로 도움이 되지 못할 것이다. 즉, Socket Service, Card Service, Driver Service와 같은 것은 아직 존재하지 않기 때문이다. 이때는 직접적으로 HBA(Host Bus Adapter)를 통해서 갈 수 밖에 없다. 물론 초기화 정보 정도만 얻을 수 있고, 메모리가 mapping이 되기만 하면, 그 다음 부터는 직접적으로 card에 있는 register들에 접근해서 원하는 일을 할 수 있으리라 생각한다. 어쨌든, 많은 시행착오를 각오해야 할 것이다. 이것으로 Linux의 PCMCIA 구현에 대한 이야기를 마친며, 부족한 부분은 항상 나중이라는 말로 대신하도록 하겠다.

14. CardBus Client Driver의 분석

앞에서는 PC Card client driver인 dummy_cs에 대한 분석을 해 보았다. 이곳에서는 CardBus를 사용하는 client driver에 대한 분석을 보도록 하겠다. 완전히 다 분석해 보는 것도 좋지만, 어떤 방식으로 제작하는지를 알기위해서 주로 CardBus와 관련된 것을 위주로 해서 코드를 분석해 보도록 하겠다.

15. Input Device Driver In Linux

15.1. Input Device Driver

Input 디바이스는 Linux상에서 동작하는 모든 input과 관련된 기기들을 지원할 목적으로 만들어진 드라이버이다. 하지만 현재 커널 버전상에서는 단지 USB 디바이스들로 한정되며, ~/drivers/input이하에 위치한다.

디바이스 드라이버들은 hardware와 통신하며, event³¹⁶들을 input.o 모듈로 알려준다. 이렇게 전달된 event들은 각각의 event handler가 받게되며, 이것은 다시 다양한 인터페이스를 통해서 전달된다. 예를 들어서 key보드를 누르는 것은 kernel 전달 되며, 마우스를 움직이는 것은 PS/2를 simulation하는 interface를 통해서 GPM이나 X등으로 전달될 것이다.

Input 디바이스를 사용하기 위해서는 다음과 같은 전형적인 예가 있을 것이다. 즉, 어떠한 모듈들이 필요한지를 나타낸다. 먼저 input.o 모듈이 있어야 할 것이며, keyboard나 mouse와 관련된 모듈³¹⁷, usb core 모듈, 그리고, USB가 사용하는 interface 모듈로 UHCI나 OHCI 모듈, HID 모듈이 필요로 하다.

OHCI와 UHCI는 host controller에 있는 USB interface chip을 제어하기 위한 모듈이며, HID 모듈은 Human Interface Device들을 위한 모듈이다. HID는 가장 크고 복잡한 모듈로서, misc, joystick, gamepad, steering wheels, keyboard, trackball, digitizer와 같은 것들을 다룬다(handle). 또한, USB에서는 HID를 또한 monitor를 제어하거나, speaker 및 UPS, LCD등과 같은 여러 다른 목적을 위해서도 사용한다.

~/drivers/input이하의 디렉토리를 보면 다음과 같은 파일들이 있을 것이다. 이 각각의 파일은 자신의 목적에 맡는 디바이스를 제어하기 위해서 사용된다.

파일	설명
evdev.c	Character 디바이스에 대한 event를 알려주는 것으로, raw input 디바이스에 대한 접근을 제공한다.
input.c	Input layer 모듈 자체이다.
joydev.c	Input driver를 위한 Joystick 디바이스 드라이버이다.
keybdev.c	Keyboard driver가 bind되는 input driver이다.
mousedev.c	PS/2나 ImPS/2 디바이스 모듈에 대한 input driver이다.

표 91. Input driver의 모듈들

Input device 드라이버의 모듈 중에서 mouse와 관련된 모듈에 대한 것을 보기 위해서 우리는 input.c파일에서 input 디바이스의 등록과 해제를 먼저 보도록 하자.

15.1.1. Input Device의 등록

Input 디바이스 드라이버를 이해하기 위해서는 먼저 input_dev 구조체를 이해해야 한다. 모든 디바이스 드라이버가 그렇듯이 input device를 등록하고 해제하는 함수들은 이 구조체를 사용할 것이다. 정의는 ~/include/linux/input.h에 있으며, 아래와 같이 볼 수 있다.

필드	설명
void *private	Private 데이터 구조체로 디바이스에 고유한 목적으로 사용된다.
int number	Input device의 번호(input device가 추가될 때마다 증가되는 값으로 설정됨)
char *name	Input device의 name
unsigned short idbus	Input device의 bus ID
unsigned short idvendor	Input device의 vendor ID

³¹⁶ 예를 들어서, keyboard의 key를 누르는 것이나 마우스를 움직이는 것이 이에 해당할 것이다.

³¹⁷ keybdev.o와 mousedev.o와 같은 모듈이다.

unsigned short idproduct	Input device의 product ID
unsigned short idversion	Input device의 version ID
unsigned long evbit[NBITS(EV_MAX)]	Event bit 배열
unsigned long keybit[NBITS(KEY_MAX)]	Keyboard bit 배열
unsigned long relbit[NBITS(REL_MAX)]	Relative 좌표 bit의 배열
unsigned long absbit[NBITS(ABS_MAX)]	Absolute 좌표 bit의 배열
unsigned long ledbit[NBITS(LED_MAX)]	LED bit 배열
unsigned long sndbit[NBITS(SND_MAX)]	Sound bit 배열
unsigned int keycodemax	Key code의 최대값
unsigned int keycodesize	Key code의 크기
void *keycode	Key code의 값을 가지는 배열에 대한 포인터
unsigned int repeat_key	Repeat key
struct timer_list timer	Timer
int abs[ABS_MAX+1]	Absolute 좌표의 배열
int rep[REP_MAX+1]	Repeat 배열
unsigned long key[NBITS(KEY_MAX)]	Key값
unsigned long led[NBITS(LED_MAX)]	LED값
unsgined long snd[NBITS(SND_MAX)]	Sound값
int absmax[ABS_MAX+1]	Absolute 좌표의 maximum값을 가지는 배열
int absmin[ABS_MAX+1]	Absolute 좌표의 minimum값을 가지는 배열
int absfuzz[ABS_MAX+1]	Absolute 좌표의 흐림값의 배열
int absflat[ABS_MAX+1]	Absolute 좌표의 내림값의 배열
int (*open)(struct input_dev *dev)	Input device의 open함수
int (*close)(struct input_dev *dev)	Input device의 close함수
int (*event)(struct input_dev *dev, unsigned int type, unsigned int code, int value)	Event handler 함수에 대한 포인터
struct input_handle *handle	Input device의 handle
struct input_dev *next	다음 input_dev구조체에 대한 포인터

표 92. `input_dev` 구조체의 필드 정의

`input_dev` 구조체의 모든 필드들에 대해서 각 디바이스가 다 정의를 둘 필요는 없다. 단지 자신의 디바이스 드라이버가 사용하는 필드에 대한 처리만 해주면 된다. 예를 들어서, USB mouse 디바이스는 event와 key, relative에 대한 필드들을 선별적으로 사용하고 있으며, open/close() 함수들을 정의해서 사용하고 있다.

`input_dev` 구조체를 정의할 때 사용한 각 상수값 및 매크로의 정의는 다음과 같다. `BITS_PER_LONG`은 Intel와 ARM에서 32로 정의되어 있다.³¹⁸

#define EV_MAX	0x1f
#define KEY_MAX	0x1ff
#define REL_MAX	0x0f
#define ABS_MAX	0x1f
#define LED_MAX	0x0f
#define REP_MAX	0x01
#define SND_MAX	0x07
#define NBITS(x)	((((x)-1)/BITS_PER_LONG)+1)

코드 762. MAX값의 정의

³¹⁸ ~/include/asm/types.h를 참고하기 바란다.

Input device를 등록하는 함수는 `input_register_device()`이다. 정의는 `~/drivers/input/input.c`에 있으며, 이하에서 input device의 해제 및 핸들러에 대한 함수들의 정의도 이곳에서 찾을 수 있을 것이다.

```
void input_register_device(struct input_dev *dev)
{
    struct input_handler *handler = input_handler;
    struct input_handle *handle;
/*
 * Initialize repeat timer to default values.
 */
    init_timer(&dev->timer);
    dev->timer.data = (long) dev;
    dev->timer.function = input_repeat_key;
    dev->rep[REP_DELAY] = HZ/4;
    dev->rep[REP_PERIOD] = HZ/33;
/*
 * Add the device.
 */
    if (input_number >= INPUT_DEVICES) {
        printk(KERN_WARNING "input: ran out of input device numbers!\n");
        dev->number = input_number;
    } else {
        dev->number = find_first_zero_bit(input_devices, INPUT_DEVICES);
        set_bit(dev->number, input_devices);
    }
    dev->next = input_dev;
    input_dev = dev;
    input_number++;
/*
 * Notify handlers.
 */
    while (handler) {
        if ((handle = handler->connect(handler, dev)))
            input_link_handle(handle);
        handler = handler->next;
    }
}
```

코드 763. `input_register_device()` 함수의 정의

`input_register_device()` 함수는 `input_dev` 구조체를 넘겨받아서, 이를 전역 변수 `input_dev`에 연결 리스트로 만드는 역할을 한다. 먼저 `input device`의 `timer`를 초기화 한다(`init_timer()`). Time out이 걸릴 때 호출될 함수로서 `input_repeat_key()`를 등록하고, 이 함수가 받을 파라미터로 `input_dev` 구조체를 선언한다. 이것은 repeat가 얼마의 속도로 일어나는지를 알기 위한 것으로 repeat delay를 HZ/4(대략 25ms)로 두고, 간격을 HZ/33(대략 3.3ms)로 두고 있다.

만약 전역 변수 `input_number`가 `INPUT_DEVICES`(=256 : `input device`의 minor 번호가 가지는 한계값)을 넘는다면, 오류가 있을 수 있음을 나타내고, `input_dev` 구조체의 `number` 필드를 `input_number`로 둔다(현재 생성된 `input device`의 개수). 넘지 않는다면, `find_first_zero_bit()` 매크로(혹은 함수)를 호출해서 모든 `input device`에 대한 bit 배열을 가지는 `input_devices`를 살펴서 빈 곳을 찾아 이 번호를 가지고 `number` 필드를 설정한다. 설정이 되었다면, `input_devices` bit 배열에 이것을 반영하기 위해서 `set_bit()`을 호출해서 bit를 1로 설정한다. 즉, 사용되고 있음을 나타내는 것이다.

이젠 `input_dev` 구조체의 배열의 연결하는 일이다. 전역 변수 `input_dev`가 모든 `input_dev` 구조체의 배열을 가지기 위해서 사용된다. 연결 리스트가 만들어지고 나면, `input_number`를 증가시켜서 새로운 디바이스가 추가되었음을 나타낸다.

만약 `handler`가 정의되어 있다면, 핸들러의 `connect` 함수를 호출하도록 하고, 이 값이 올바른 값(0이 아닌 값)을 돌려주면, `input_link_handler()`를 호출해서 `handler`를 추가한다. `input_link_handler()` 함수의 파라미터는

handler의 connect() 함수를 호출한 결과가 넘어간다. handler는 다음 handler의 주소를 가르키도록 하고, 전체 연결된 handler들을 다 한번씩 불러 주도록 한다.

만약 USB mouse 드라이버를 등록한다고 가정한다면, 먼저 input device의 mousedev.o 모듈이 먼저 적재될 것이며, 후에 다시 USB mouse 드라이버를 적재할 것이다. 따라서, mousedev.o 모듈에서 정의한 handler의 mousedev_connect() 함수가 호출 될 것이며, 이 함수의 호출결과로 넘겨받는 handler가 input_link_handle() 함수의 argument로 들어가게 될 것이다.

```
struct input_handler {
    void *private;                                /* Input handler private data */
    /* Input handler function */
    void (*event)(struct input_handle *handle, unsigned int type, unsigned int code, int value);
    struct input_handle* (*connect)(struct input_handler *handler, struct input_dev *dev);
    void (*disconnect)(struct input_handle *handle);
    /* Input handler file operation structure --> make kernel interface */
    struct file_operations *fops;
    int minor;                                     /* Device minor number */
    struct input_handle *handle;                   /* Next input handle */
    struct input_handler *next;                   /* Next input handler */
};
```

코드 764. input_handler 함수의 정의

Input 디바이스의 핸들러는 전역 변수 input_handler에 의해서 유지되는 자료구조체이다. private 포인터는 핸들러에서 사용할 자신의 자료(혹은 데이터)를 위한 포인터이며, event는 해당 event의 핸들러, connect() 함수와 disconnect() 함수, file 연산 구조체에 대한 포인터와 자신이 제어하는 디바이스의 minor 번호 및 input_handle의 포인터와 다음 input_handler 구조체에 대한 포인터로 이루어져 있다.

```
struct input_handle {
    void *private;                                /* Input handle private data */
    int open;                                      /* Open counter */
    struct input_dev *dev;                         /* Input device pointer */
    struct input_handler *handler;                /* Input handler pointer for this handle */
    struct input_handle *dnext;                   /* Next input handle for input device */
    struct input_handle *hnext;                   /* Next input handle for this handle */
};
```

코드 765. input_handle 구조체의 정의

input_handle 구조체는 private data에 대한 포인터와 open counter, input_dev 구조체에 대한 포인터 및 input_handler 구조체의 포인터로 이루어진다. dnext는 다음 input_dev 구조체의 handle을 나타내며, hnext는 input_handler의 handle을 나타내는 값이다. 각각은 input_link_handle() 함수에서 연결이 정해진다.

아래의 그림은 앞에서 등록한 input device의 구조를 전체적으로 보여준다. 전체적인 input device 구조는 input_dev라는 변수가 가르키고 있다.

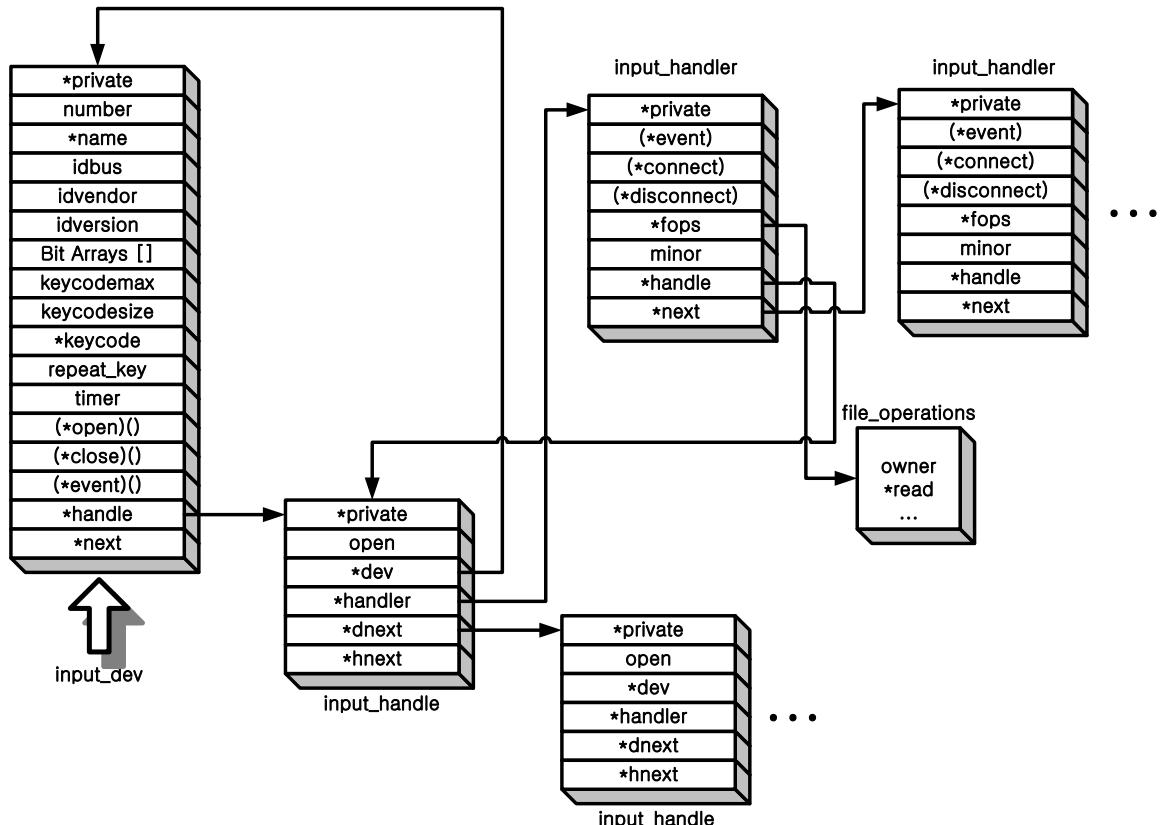


그림 96. Input Device의 Architecture.

[그림 96]에서 보듯이 하나의 `input_device` 구조체는 연결된 `input_handle` 구조체를 가지며, 각각의 `input_handle` 구조체는 또한 연결된 `input_handler` 구조체를 가진다. 따라서, 앞에서 `input device`를 등록할 때는 전체 handler의 함수의 `connect()` 함수가 차례로 수행될 것이다. 여기서 중요한 점은 이러한 handler 각각에 하나의 `file_operations` 구조체를 가지고 있다는 점이다. 즉, 이 핸들러는 하나의 `device node`로 표현될 수 있으며, 커널은 `device node`를 가지는 핸들러를 호출해 줄 수 있게 된다. 따라서, character device driver나 block device driver와 같이 사용될 수 있도록 하고 있음을 알 수 있다.

15.1.2. Input Device의 해제

Input device의 해제를 맡고 있는 함수는 `input_unregister_device()` 함수이다. 이 함수에는 앞에서 등록한 `input device`를 제거하는 일을 한다. 넘겨받는 함수는 등록된 `input_dev` 구조체를 가르키는 포인터이다(`dev`).

```
void input_unregister_device(struct input_dev *dev)
{
    struct input_handle *handle = dev->handle;
    struct input_dev **devptr = &input_dev;
    struct input_handle *dnext;
/*
 * Kill any pending repeat timers.
 */
    del_timer(&dev->timer);
/*
 * Notify handlers.
 */
    while (handle) {
        dnext = handle->dnex;
```

```

        input_unlink_handle(handle);
        handle->handler->disconnect(handle);
        handle = dnext;
    }

/*
 * Remove the device.
 */
    while (*devptr && (*devptr != dev))
        devptr = &((*devptr)->next);
    *devptr = (*devptr)->next;
    input_number--;
    if (dev->number < INPUT_DEVICES)
        clear_bit(dev->number, input_devices);
}

```

코드 766. `input_unregister_device()` 함수의 정의

`input_unregister_device()` 함수는 앞에서 등록한 timer를 먼저 제거한다(`del_timer()`). 그리고나서, `input_dev` 구조체가 가진 핸들러를 하나씩 제거해 나간다(`input_unlink_handle()`). 또한 제거한 핸들의 핸들러에 있는 `disconnect()` 함수를 호출해서 이 디바이스와 관련된 핸들러에게 제거된다는 것을 알려준다. 또한 여기서 주의 할 점은 전체 `input_dev` 구조체와 연결된 핸들러에 대해서 이러한 일이 다 일어난다는 점이다. 여기까지 진행했다면, `input_dev` 구조체가 가진 모든 handle에 대한 제거는 마친 상태이다. 이전 `input_dev` 구조체를 제거할 차례이다. 먼저 해당하는 `input_dev` 구조체를 찾도록 한다(while loop). 찾았다면, 이 `input_dev` 구조체가 가지는 `next` 필드를 이용해서 연결리스트의 다음번에 있는 `input_dev` 구조체를 앞으로 이동 시켜주고, `input_number`를 1 감소 시켜서 현재 시스템에 연결된 `input device`가 감소됨을 알려준다. 만약 현재 `input device`의 수(`dev->number`)가 `INPUT_DEVICES`(=256)보다 작다면, `input device`들의 bitmap을 가지고 있는 `input_devices`에서 이 bit를 지운다(`clear_bit()`). 이것으로 `input_dev` 구조체의 삭제가 끝나게 된다. 나머지 지워진 `input_dev` 구조체는 나중에 디바이스 드라이버와 같은 곳에서 메모리를 해제 시켜주어야 할 것이다.

15.1.3. Input Device Handle의 등록

일반적으로 `handle`이라고 하면, 특정한 연산의 결과 값이나, 혹은 argument로 넘겨지는 것들을 이야기하는 것으로 주로 private 데이터나 함수들에 대한 포인터를 지닌다. Input device의 `handle`은 핸들러에 대한 포인터를 가지며, 각 핸들러들에 대한 연결 구조 및 `input device` 구조체에 대한 포인터를 가지고 있다. 여기서 말하는 등록이란 `handle`을 `input device`의 핸들러 list에 연결 시켜주는 것을 말하며, `input_link_handle()` 함수가 맡고 있다.

```

static void input_link_handle(struct input_handle *handle)
{
    handle->dnext = handle->dev->handle;
    handle->hnext = handle->handler->handle;
    handle->dev->handle = handle;
    handle->handler->handle = handle;
}

```

코드 767. `input_link_handle()` 함수의 정의

`input_link_handle()` 함수는 `input_handle`을 받아서, `handle`의 `dnext`필드는 `handle`의 `input_dev` 구조체의 `handle` 값으로 채우고, `hnext` 필드는 `handle`의 `handler`가 가지는 `handle` 필드로 채운다. `handle`의 `dev`구조체가 가지는 `handle` 필드는 다시 `handle`을 가지도록 만들어서, 현재 디바이스와 연결된 `handle`이 어느것인가를 나타내도록 한다. `handler`의 `handle`역시 `handle`로 주어, 같은 `handle`을 가지도록 만든다. 즉, `input_link_handle`은 `input device`에 대한 `handle`을 linked list로 만드는 역할을 하고 있다.

15.1.4. Input Device Handle의 해제

Input device handle을 해제하는 함수는 `input_unlink_handle()` 함수이며, 앞에서 보았던 `input_link_handle()` 함수의 반대 역할을 한다.

```
static void input_unlink_handle(struct input_handle *handle)
{
    struct input_handle **handleptr;

    handleptr = &handle->dev->handle;
    while (*handleptr && (*handleptr != handle))
        handleptr = &(*handleptr)->dnext;
    *handleptr = (*handleptr)->dnext;
    handleptr = &handle->handler->handle;
    while (*handleptr && (*handleptr != handle))
        handleptr = &(*handleptr)->hnext;
    *handleptr = (*handleptr)->hnext;
}
```

코드 768. `input_unlink_handle()` 함수의 정의

`input_unlink_handle()` 함수는 제거하고자 하는 `input_handle`을 가르키는 포인터를 넘겨받는다. 이 핸들이 가르키는 `input_dev` 구조체(`dev`)의 `handle` 필드를 `handleptr`로 가르키게 하고, 이 값이 같은 `handle`을 `dnext` 필드를 따라가면서 찾는다(while). 찾았다면, `handleptr`이 가르키는 것을 `handleptr`의 `dnext`가 가지는 값으로 대치 시켜서, 핸들의 link를 리스트에서 제거한다. 이젠 `handle`이 가르키는 `handler`의 `handle`필드를 다시 `handleptr`로 두고, `handleptr`이 가르키는 값이 `handle`과 같은 것이 있을 때까지 추적하게 된다. 같은 것을 찾았다면, `handleptr`이 가르키는 내용을 `hnex` 필드 값으로 대치시켜서 리스트에서 제거한다.

앞에서 보았던 `input_link_handle()` 함수와 `input_unlink_handle()` 함수에서 보듯이, 하나의 핸들은 `input_handle` 구조체의 `dnext` 필드와 `hnex` 필드에 동시에 존재할 수 있다는 것을 알았을 것이다. `dnext`는 `device`가 가지는 `handle`의 연결 리스트 구조를 만들어 주고, `hnex`는 핸들러가 가지는 `handle`의 연결 리스트 구조를 만들기에 두 함수는 그 두개의 필드에 대해서 `handle`을 추가하거나 삭제하는 일을 해주고 있다.

15.1.5. Input Device의 Handler의 등록

Input device의 event handler를 등록하는 것은 `input_register_handler()` 함수가 맡고 있다. 이 함수는 해당하는 input device에 핸들러를 추가하는 역할을 하는 함수로서 핸들러 및 file operation에 대한 추가가 있게 된다.

```
void input_register_handler(struct input_handler *handler)
{
    struct input_dev *dev = input_dev;
    struct input_handle *handle;
/*
 * Add minors if needed.
 */
    if (handler->fops != NULL)
        input_table[handler->minor >> 5] = handler;
/*
 * Add the handler.
 */
    handler->next = input_handler;
    input_handler = handler;
/*
 * Notify it about all existing devices.
*/
}
```

```

while (dev) {
    if ((handle = handler->connect(handler, dev)))
        input_link_handle(handle);
    dev = dev->next;
}

```

코드 769. `input_register_handler()` 함수의 정의

`input_register_handler()` 함수가 넘겨받는 값은 `input_handler` 구조체의 포인터이다. 먼저 해당하는 `input_dev` 구조체의 포인터를 구하기 위해서 `dev`에 전역 `input_dev` 구조체의 포인터를 가지는 `input_dev`를 넣어준다. 만약 `handler`의 `fops` 필드가 NULL이 아니라면, `input_table`에 `handler`의 `minor` 필드를 우측으로 5 bit shift시켜서 32로 나누어준 곳에 `handler`를 추가한다. 즉, `input_table` 포인터는 `input_handler` 구조체를 가르키는 전역 구조체로서 8개의 원소를 가지는데, 이곳에 `minor / 32`를 위치를 `handler`로 바꾸어주는 것이다.

이전 핸들러를 실제로 추가해 주어야 할 것이다. 즉, 현재의 전역 변수 `input_handler`의 값으로 넘겨받은 `handler`의 `next` 필드를 초기화하고, 다시 `input_handler`에는 이 핸들러(`handler`)를 주도록 한다. 마지막으로 함수를 마치기 전에 모든 시스템에 있는 `input device`에 새로운 핸들러가 추가되었음을 알려주기 위해서 각 디바이스별로 핸들러의 `connect()` 함수를 호출하도록 한다. 만약 적절한 `handle`값을 `connect()` 함수가 돌려준다면, 이것으로 `input_link_handle()` 함수를 호출해서 `handle`을 `input_dev` 구조체에 연결 시켜주도록 한다.

15.1.6. Input Device의 Handler의 해제

이전 `input device`의 `event handler`를 해제하는 것을 보도록 하자. 이것을 처리하는 함수는 `input_unregister_handler()` 함수이다. 앞에서 `input_register_handler()` 함수가 했던 일을 되돌리는 일이 될 것이다.

```

void input_unregister_handler(struct input_handler *handler)
{
    struct input_handler **handlerptr = &input_handler;
    struct input_handle *handle = handler->handle;
    struct input_handle *hnxt;

/*
 * Tell the handler to disconnect from all devices it keeps open.
 */
    while (handle) {
        hnxt = handle->hnxt;
        input_unlink_handle(handle);
        handler->disconnect(handle);
        handle = hnxt;
    }
/*
 * Remove it.
 */
    while (*handlerptr && (*handlerptr != handler))
        handlerptr = &((*handlerptr)->next);
    *handlerptr = (*handlerptr)->next;
/*
 * Remove minors.
 */
    if (handler->fops != NULL)
        input_table[handler->minor >> 5] = NULL;
}

```

코드 770. `input_unregister_handler()` 함수의 정의

`input_unregister_handler()` 함수가 넘겨받는 값은 제거할 handler에 대한 포인터이다. 모든 핸들러에 대한 것을 가르키고 있는 전역 변수인 `input_handler`의 값을 `handlerptr`이 가지도록 만들고, `handler`가 가지는 `handle`에 대한 포인터를 `handle`이 가르키게 만든다. 이와 같이 했다면, 이젠 해당하는 핸들러의 모든 `handle`을 제거한다(`input_unlink_handle()`). 그리고나서, 제거하려는 핸들러의 `disconnect()` 함수를 호출해서 이 핸들러가 제거되고 있음을 알려주고, `handle`은 다음번 핸들러(`hnext`)를 가르키도록 만든다.

핸들이 제거가 되었으므로 이젠 핸들러를 제거해 주어야 할 것이다. 먼저 해당하는 핸들러를 찾는다(while loop). 그리고나서, 이 핸들러의 `next` 필드로 원래의 핸들러를 대체시켜, 핸들러를 연결리스트에서 제거한다. 이젠 관련된 `input_table[]`에서 앞에서 등록한 minor / 32에 해당하는 entry를 `NULL`로 만들어주도록 한다. 이것을 `input_unregister_handler()` 함수의 수행이 끝나게 된다.

15.1.7. Input Device의 Event 처리

이전 Input device에서 발생하는 event를 처리하는 부분을 보도록 하자. `input_event()` 함수가 이러한 처리를 맡고 있다. 이 함수에서 하는 일은 input device들에서 발생하는 event에 따라서, input_dev 구조체의 해당 event에 관련된 bit를 설정하는 일이다.

```
void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value)
{
    struct input_handle *handle = dev->handle;
/*
 * Filter non-events, and bad input values out.
 */
    if (type > EV_MAX || !test_bit(type, dev->evbit))
        return;
    switch (type) {
        case EV_KEY:
            if (code > KEY_MAX || !test_bit(code, dev->keybit) || !!test_bit(code, dev->key) == value)
                return;
            if (value == 2) break;
            change_bit(code, dev->key);
            if (test_bit(EV_REPEAT, dev->evbit) && dev->timer.function) {
                if (value) {
                    mod_timer(&dev->timer, jiffies + dev->rep[REP_DELAY]);
                    dev->repeat_key = code;
                    break;
                }
                if (dev->repeat_key == code)
                    del_timer(&dev->timer);
            }
            break;
    }
}
```

코드 771. `input_event()` 함수의 정의

처리하고자 하는 `input_dev` 구조체의 핸들을 먼저 `handle` 변수로 둔다. 그리고나서, 넘겨받은 `type`이 `EV_MAX`보다 크거나, 혹은 `type`에 해당하는 `input_dev` 구조체의 `evbit` 필드의 `type`에 해당하는 `bit`이 0인 경우(`!test_bit()`)에는 곧바로 복귀한다. 그렇지 않다면, `switch`문 이하를 `type`에 따라서, 실행하게 된다.

먼저 `type`이 `EV_KEY` 값을 가진다면, 받아 들인 `code`의 값이 `KEY_MAX`보다 크거나, 해당하는 `keybit` 필드가 0이거나, 혹은 해당하는 `key` 필드의 `bit`을 읽어서, 이 값을 넘겨받은 `value`와 같다면, 곧바로 다시 복귀한다. 이것은 `code` 값에 오류가 있다는 말이 되기 때문이다. 또한 만약 `value` 값이 2라고 한다면, `switch`문을 곧바로 나가도록 한다. 이젠 `code`에 해당하는 `key` 필드의 `bit`을 `toggle` 시켜주고(`change_bit()`), 다시 `EV_REPEAT`가 `evbit`에 설정되었고, `timer`가 있다면 `value`에 따라서 `timer`를 갱신하고(`mod_timer()`), `key`가 연속적으로 눌려진 상황을 보여주기 위해서 `repeat_key`에 `code`를 넣어주면서 `switch`문을 빠져나가게 된다. 만약 `repeat_key` 값과 현재의 `code` 값이 같다면, 앞에서 설정한 `timer`를 제거해 주기 위해서 `del_timer()` 함수를 호출한다. 즉, 이미 반복되었다는 것을 `check` 했으므로 더 이상 `timer`가 필요하지 않기 때문이다.

```

case EV_ABS:
    if (code > ABS_MAX || !test_bit(code, dev->absbit))
        return;
    if (dev->absfuzz[code]) {
        if ((value > dev->abs[code] - (dev->absfuzz[code] >> 1)) &&
            (value < dev->abs[code] + (dev->absfuzz[code] >> 1)))
            return;
        if ((value > dev->abs[code] - dev->absfuzz[code]) &&
            (value < dev->abs[code] + dev->absfuzz[code]))
            value = (dev->abs[code] * 3 + value) >> 2;
        if ((value > dev->abs[code] - (dev->absfuzz[code] << 1)) &&
            (value < dev->abs[code] + (dev->absfuzz[code] << 1)))
            value = (dev->abs[code] + value) >> 1;
    }
    if (dev->abs[code] == value)
        return;
    dev->abs[code] = value;
    break;
}

```

코드 772. input_event() 함수의 정의(계속)

type에 EV_ABS값이 주어진 경우이다. 앞에서와 마찬가지로 code값이 ABS_MAX보다 크거나, absbit에 설정이 되지 않은 경우에는 바로 복귀(return) 한다. 만약 absfuzz[]의 code에 대응하는 bit이 설정이 된 경우에는 이것을 최종적으로 abs[]의 code에 들어갈 값(value)을 결정하는데 영향을 주기 위해서 absfuzz[]의 code index에 있는 값을 적절한 연산을 해 주도록 한다.³¹⁹ 만약 계산된 value 값이 이미 abs[]에 있는 값과 같다면 바로 복귀하고, 그렇지 않다면 새로 계산된 값으로 abs[]의 code index 값을 결정해준다(value).

```

case EV_REL:
    if (code > REL_MAX || !test_bit(code, dev->relbit) || (value == 0))
        return;
    break;
case EV_LED:
    if (code > LED_MAX || !test_bit(code, dev->ledbit) || !test_bit(code, dev->led) == value)
        return;
    change_bit(code, dev->led);
    if (dev->event) dev->event(dev, type, code, value);
    break;
case EV SND:
    if (code > SND_MAX || !test_bit(code, dev->sndbit) || !test_bit(code, dev->snd) == value)
        return;
    change_bit(code, dev->snd);
    if (dev->event) dev->event(dev, type, code, value);
    break;
}

```

³¹⁹ 이곳에서 보여주는 연산은 먼저 abs[] 값에 absfuzz[]를 반으로 나눈 값을 빼서, 이 값보다는 value가 크고, 혹은 둘을 더한 값보다 value가 작을 때는 그냥 return해주도록 하는 것이다. 이 경우에는 absfuzz[]에 있는 값의 50%변위 보다 value가 abs[]에 영향을 적게 미친다는 것을 의미한다고 볼 수 있다. 또한 두 번째 연산에는 앞에서 한 absfuzz[]를 반으로 나누는 것을 없앴는데, 이때는 해당 변위(+-absfuzz[])에 value값이 속한다고 보는 것으로(물론 abs[]와 더한 값), abs[]에 3을 곱해서 value값을 더하고, 이를 다시 2로 나누어 새로운 value값을 만들어 준다. 이것은 abs[]의 3/4배를 한 값에 다시 value를 1/4로 해서 더한 값으로 value를 만들어 주는 것으로 abs[]가 갑작스럽게 늘어나는 것을 막아준다. 세 번째 연산으로는 value가 absfuzz[]에 2를 곱한 값을 abs[]에 더하거나 뺀 값의 사이에 존재하는 경우로 이때는 abs[]와 value를 더해서 다시 이를 1/2 해준 값으로 value를 결정하고 있다. 이때도 역시 value값이 너무 크게 증가하는 것을 막기 위해서 둘간이 합을 다시 1/2로 해서 평균 값으로 value를 설정해 준 것이다.

```

        case EV_REL:
            if (code > REL_MAX || dev->rel[code] == value) return;
            dev->rel[code] = value;
            if (dev->event) dev->event(dev, type, code, value);
            break;
    }
/*
 * Add randomness.
 */
#ifndef BUG
    add_input_randomness(((unsigned long) dev) ^ (type << 24) ^ (code << 16) ^ value);
#endif
/*
 * Distribute the event to handler modules.
 */
while (handle) {
    if (handle->open)
        handle->handler->event(handle, type, code, value);
    handle = handle->dnext;
}
}

```

코드 773. input_event() 함수의 정의(계속)

EV_REL의 경우에는 단순히 현재의 REL_MAX보다 크거나, relbit이 설정되 되지 않은 경우이거나, 혹은 value가 0인 경우에는 바로 복귀한다. EV_LED의 경우에도 마찬가지지만, 이 경우에는 설정된 led값이 value와 같은지를 추가적으로 더 확인하고 있으며, 그렇지 않다면 led를 toggle시켜주고(change_bit()), input_dev 구조체의 event() 함수를 호출해 준다. 등록된 함수가 있다면, 이후의 처리를 맡게 될 것이다. EV_SND 및 EV_REL의 경우에도 역시 LED와 마찬가지 일을 하게 된다. 하지만 EV_REL의 경우에는 REP_MAX에 대한 확인과 rep[]에 이미 value가 있는지만을 보게되며, 없다면 rep[]에 value값이 들어가게 되고, 다시 input_dev 구조체의 event() 함수가 호출될 것이다.

“#if 0 ~ #endif”로 처리된 부분은 실행되지는 않지만, 원래의 목적은 input device에서 발생하는 event를 가지고 시스템의 randomness에 영향을 주기 위한 것이다. 즉, 나중에 random number를 생성하는 경우에 이곳에서 중복되지 않는 수를 생성하기 위한 randomness를 더하게 된다. 이제 마지막으로 남은 것은 실제로 input device에 등록된 event handler들을 차례로 호출해 주는 일이다. 이것은 핸들의 dnext 필드를 주적해서 각각의 handler 함수의 event() 함수를 호출하는 일이 될 것이다.

마지막으로 나중에 나올 USB mouse device 드라이버 및 USB keyboard device 드라이버를 위해서 input_event()와 관련된 매크로의 사용에 대해서 잠시 알아보기로 하자.

```
#define input_report_key(a,b,c) input_event(a, EV_KEY, b, !(c))
#define input_report_rel(a,b,c) input_event(a, EV_REL, b, c)
#define input_report_abs(a,b,c) input_event(a, EV_ABS, b, c)
```

코드 774. input_report_XXX() 매크로의 정의

각각의 input_report_XXX()는 위에서 정의한 것과 같이 EV_KEY, EV_REL, EV_ABS라는 event type을 발생시켜주며, key값인 경우에는 value 값을 “!”으로 사용해서 논리적인 값으로 변환하고 있다.

15.1.8. Input Device Driver의 초기화 및 해제

앞에서 우린 이미 input device의 여러 부분들에 대해서 알아보았다. 그렇다면, 이젠 이러한 input device driver가 어떻게 초기화 되는지를 알아볼 차례이다. 아래와 같다.

```
static int __init input_init(void)
{
```

```

if (devfs_register_chrdev(INPUT_MAJOR, "input", &input_fops)) {
    printk(KERN_ERR "input: unable to register char major %d", INPUT_MAJOR);
    return -EBUSY;
}
input_devfs_handle = devfs_mk_dir(NULL, "input", NULL);
return 0;
}

static void __exit input_exit(void)
{
    devfs_unregister(input_devfs_handle);
    if (devfs_unregister_chrdev(INPUT_MAJOR, "input"))
        printk(KERN_ERR "input: can't unregister char major %d", INPUT_MAJOR);
}
module_init(input_init);
module_exit(input_exit);

```

코드 775. `input_init()` 함수 및 `input_exit()` 함수의 정의

`module_init()` 함수와 `module_exit()` 함수는 드라이버가 `module`로서 등록되거나 혹은 kernel과 static하게 link될 경우에 각각 달리 처리된다. 이 함수들은 `module`의 초기화 및 해제에 해당하는 함수를 가르키도록 하는 일종의 macro 역할을 수행하게 된다. 각각은 `input_init()` 함수와 `input_exit()` 함수를 정의하고 있다.

먼저 `input_init()` 함수는 `devfs_register_chrdev()`를 호출해서 input device의 major 번호와 file operation 구조체를 넘겨주고 있다. 이것은 문자 디바이스 드라이버로 device file system에 등록하라는 뜻이다. 예러가 있었다면, `-EBUSY`가 복귀 값이 될 것이다. 이젠 device file system의 한 node를 만들기 위해서 `devfs_mk_dir()`를 호출해서 `input_devfs_handle` 변수가 가르키도록 만든다. 이 handle은 나중에 device file system에서 input device에 해당하는 node를 제거하기 위해서 사용될 것이다. 복귀 값은 0이다.

`input_exit()` 함수는 앞에서 등록한 device file system의 node를 제거하기 위해서 `devfs_unregister()`에 `input_devfs_handle`을 넘겨주어 호출한다. 이것을 마치고나면, 다시 device file system에 등록된 character device를 제거하기 위해서 `devfs_unregister_chrdev()` 함수를 호출하도록 한다. 이것으로 input device driver의 등록과 해제가 끝나게 된다.

```

static int input_open_file(struct inode *inode, struct file *file)
{
    struct input_handler *handler = input_table[MINOR(inode->i_rdev) >> 5];
    struct file_operations *old_fops, *new_fops = NULL;
    int err;

    /* No load-on-demand here? */
    if (!handler || !(new_fops = fops_get(handler->fops)))
        return -ENODEV;
    /*
     * That's _really_ odd. Usually NULL ->open means "nothing special",
     * not "no device". Oh, well...
     */
    if (!new_fops->open) {
        fops_put(new_fops);
        return -ENODEV;
    }
    old_fops = file->f_op;
    file->f_op = new_fops;
    lock_kernel();
    err = new_fops->open(inode, file);
    unlock_kernel();
    if (err) {

```

```

        fops_put(file->f_op);
        file->f_op = fops_get(old_fops);
    }
    fops_put(old_fops);
    return err;
}
static struct file_operations input_fops = {
    owner: THIS_MODULE,
    open: input_open_file,
};

```

코드 776. `input_open_file()` 함수의 정의 및 `input device`의 `file_operations` 구조체 정의

앞에서 `input device driver`를 등록하는 부분에서 `file_operations` 구조체에 대한 정의를 보았다. 즉, 이 함수는 문자 디바이스로 등록된 `input device` 드라이버가 `open`될 때 수행될 것이다. `input_open_file()` 함수는 `device file system`에 등록된 하나의 `device node`에 대한 `open` 연산을 수행한다. 먼저 `input_table[]`에 등록된 핸들러를 `minor`번호를 32로 나눈 값으로 구한다. 핸들러가 없거나, 핸들러의 `file_operations` 구조체 선언이 없다면 `-ENODEV`를 복귀 값으로 줄 것이다.

또한 만약 이렇게 구한 handler의 `file_operations` 구조체의 `open`이 없을 경우에도 예러가 되며, `fops_put()`으로 앞에서 `fops_get()` 연산의 역을 수행한 후, `-ENODEV`를 돌려줄 것이다. 원래의 `file_operations` 구조체를 `old_fops`에 저장하고, 새로운 `file_operations` 구조체를 `file`의 `f_op` 필드에 채워준다. 이후의 `open` 및 `file`에 대한 연산은 모두 이 새로이 적용된 `file_operations` 구조체가 맡아서 처리하게 된다. 이젠 `file`에 대한 실제 `open` 연산을 하기 위해서 커널에 `lock`을 설정하고(`lock_kernel()`), 새로운 `file_operations` 구조체의 `open()` 함수를 호출한다. 이것을 마치고 나면, `unlock_kernel()`로 커널에 대해서 설정한 `lock`을 푼다. 만약 `err`가 `open()` 함수의 호출에서 발생했다면, 이를 처리하기 위해서 다시 `fops_put()`를 호출하게 되며, 원래의 `file_operations()` 구조체를 가져와서 `file`의 `f_op` 필드에 넣어주게 된다. 복귀하기 전에 이전에 있던 `file_operations` 구조체는 `fops_put()`으로 제공하는 `module`의 `reference counter`를 낮추게 된다. 복귀 값은 `open()` 연산의 결과 값인 `err`이다.

```

#define fops_get(fops) \
    (((fops) && (fops)->owner) \
     ? ( try_inc_mod_count((fops)->owner) ? (fops) : NULL ) \
     : (fops))

#define fops_put(fops) \
do { \
    if ((fops) && (fops)->owner) \
        __MOD_DEC_USE_COUNT((fops)->owner); \
} while(0)

```

코드 777. `fops_get()`과 `fops_put()` 매크로의 정의

`fops_get()`과 `fops_put()`은 모두 `~/include/linux/fs.h`에 정의된 것이다. 즉, `fops_get()`은 해당하는 `file_operations` 구조체의 모듈에 대한 `counter`를 증가시키고, `file_operations` 구조체를 돌려주는 매크로이다. 또한 `fops_put()`은 `file_operations` 구조체의 `module`에 대한 `count`를 낮춰주는 역할만을 한다. 이 두개의 매크로는 `file_operations` 구조체를 `export`하고 있는 모듈에 대한 `counter`를 조정해주는 역할을 수행해 주는 매크로로 보면 될 것이다. 해당 모듈이 더 이상 `reference`가 되지 않는다면, 커널의 메모리 공간을 차지할 이유가 없기 때문이다.

15.2. Linux USB Mouse Driver

이번장에서는 앞에서 본 기본적인 USB 디바이스 드라이버를 대신해서, Linux에서의 USB mouse 드라이버를 어떻게 구현하고 있는지를 분석해 보도록 하겠다. Mouse는 시스템에서 사용자의 input을

받는 키보드 디바이스와 가장 기본적인 디바이스이므로 그 이해를 쉽게 할 수 있으리라고 생각한다. 관련된 파일은 ~/drivers/usb/usbmouse.c와 ~/include/linux/usb.h이다.³²⁰

15.2.1. USB Mouse Driver의 등록과 해제

USB mouse 드라이버는 모듈로서 등록되거나, 커널 컴파일에서 static하게 link되어 사용될 수 있도록 되어 있다.

```
static struct usb_driver usb_mouse_driver = {
    name:          "usb_mouse",
    probe:         usb_mouse_probe,
    disconnect:   usb_mouse_disconnect,
    id_table:     usb_mouse_id_table,
};

static int __init usb_mouse_init(void)
{
    usb_register(&usb_mouse_driver);
    info(DRIVER_VERSION ":" DRIVER_DESC);
    return 0;
}

static void __exit usb_mouse_exit(void)
{
    usb_deregister(&usb_mouse_driver);
}

module_init(usb_mouse_init);
module_exit(usb_mouse_exit);
```

코드 778. USB mouse driver의 등록

먼저, USB mouse driver를 등록하기 위해서 사용하는 usb_driver 구조체의 정의부터 보자. name필드로 “usb_mouse”를, probe와 disconnect에 사용할 함수로 각각 usb_mouse_probe와 usb_mouse_disconnect를 사용하며, USB 디바이스의 ID table에 등록하기 위해서 usb_mouse_id_table을 id_table 필드에 선언했다.

```
struct usb_device_id {
    __u16          match_flags;      /* 다음 필드에서 어떤 것이 matching을 위해서 사용하는지를
정의한다. */
    __u16          idVendor;        /* Vendor의 ID */
    __u16          idProduct;       /* Product의 ID */
    __u16          bcdDevice_lo, bcdDevice_hi; /* BCD코드로 된 Device의 ID에 대한 high와
low필드 */
    __u8           bDeviceClass;    /* Device의 Class를 정의한다. */
    __u8           bDeviceSubClass; /* Device의 Subclass를 정의한다. */
    __u8           bDeviceProtocol; /* Device에서 사용하는 protocol을 정의한다. */
    __u8           bInterfaceClass; /* Device의 Interface class를 정의한다. */
    __u8           bInterfaceSubClass; /* Device의 Interface subclass를 정의한다. */
    __u8           bInterfaceProtocol; /* Device의 Interface에서 사용하는 protocol을 정의한다. */
    unsigned long  driver_info;    /* Driver의 정보를 기록하는 부분으로 driver가 사용하며, driver의
matching에서는 사용하지 않는다. */
};
```

코드 779. usb_device_id 구조체의 정의

³²⁰ 이곳에서 사용한 USB mouse driver의 코드는 커널 버전 2.4.6에 해당한다.

이 usb_device_id 구조체를 위해서 USB mouse 디바이스 드라이버에서는 다음과 같이 정의하고 있다. 여기서 USB_INTERFACE_INFO()는 매크로는 앞에서 본 usb_device_id 구조체의 match_flags 필드 및 bInterfaceXXX 필드들에 대한 연산을 수행한다.

```
static struct usb_device_id usb_mouse_id_table [] = {
    { USB_INTERFACE_INFO(3, 1, 2) },
    { }
}; /* Terminating entry */
```

코드 780. usb_mouse_id_table[]의 정의

usb_mouse_id_table[]에 Interface 필드값으로 3, 1, 2를 각각 주고 있다. 이 값들은 앞에서 본 bInterfaceClass, bInterfaceSubClass, bInterfaceProtocol 필드를 각각 차지할 것이다. 여기서 3, 1, 2란 값은 mouse 디바이스가 HID(Human Interface Device) class에 속하기 때문에, USB spec.의 HID spec.을 살펴보면 알 수 있을 것이다. 즉, Class로 3번을 Subclass로 1을 그리고, 사용하는 protocol 번호로 2번을 사용하고 있음을 알 수 있을 것이다.

```
/* 상수값 정의 */
#define USB_DEVICE_ID_MATCH_VENDOR      0x0001
#define USB_DEVICE_ID_MATCH_PRODUCT     0x0002
#define USB_DEVICE_ID_MATCH_DEV_LO      0x0004
#define USB_DEVICE_ID_MATCH_DEV_HI      0x0008
#define USB_DEVICE_ID_MATCH_DEV_CLASS    0x0010
#define USB_DEVICE_ID_MATCH_DEV_SUBCLASS 0x0020
#define USB_DEVICE_ID_MATCH_DEV_PROTOCOL 0x0040
#define USB_DEVICE_ID_MATCH_INT_CLASS    0x0080
#define USB_DEVICE_ID_MATCH_INT_SUBCLASS 0x0100
#define USB_DEVICE_ID_MATCH_INT_PROTOCOL 0x0200

/* 복합적인 상수값들에 대한 정의 */
#define USB_DEVICE_ID_MATCH_DEVICE      (USB_DEVICE_ID_MATCH_VENDOR | USB_DEVICE_ID_MATCH_PRODUCT)
#define USB_DEVICE_ID_MATCH_DEV_RANGE    (USB_DEVICE_ID_MATCH_DEV_LO | USB_DEVICE_ID_MATCH_DEV_HI)
#define USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION (USB_DEVICE_ID_MATCH_DEVICE | USB_DEVICE_ID_MATCH_DEV_RANGE)
#define USB_DEVICE_ID_MATCH_DEV_INFO \
    (USB_DEVICE_ID_MATCH_DEV_CLASS | USB_DEVICE_ID_MATCH_DEV_PROTOCOL)
#define USB_DEVICE_ID_MATCH_INT_INFO \
    (USB_DEVICE_ID_MATCH_INT_CLASS | USB_DEVICE_ID_MATCH_INT_PROTOCOL)

/* match_flags필드에 대해서 사용될 매크로의 정의 */
#define USB_DEVICE(vend,prod) \
    match_flags: USB_DEVICE_ID_MATCH_DEVICE, idVendor: (vend), idProduct: (prod)
#define USB_DEVICE_VER(vend,prod,lo,hi) \
    match_flags: USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION, idVendor: (vend), idProduct: (prod), bcdDevice_lo: (lo), bcdDevice_hi: (hi)
#define USB_DEVICE_INFO(cl,sc,pr) \
    match_flags: USB_DEVICE_ID_MATCH_DEV_INFO, bDeviceClass: (cl), bDeviceSubClass: (sc), bDeviceProtocol: (pr)
#define USB_INTERFACE_INFO(cl,sc,pr) \
    match_flags: USB_DEVICE_ID_MATCH_INT_INFO, bInterfaceClass: (cl), bInterfaceSubClass: (sc), bInterfaceProtocol: (pr)
```

코드 781. 유용한 매크로들

즉, USB_INTERFACE_INFO() 매크로는 Interface Class, Interface Subclass, Interface Protocol을 넘겨받아서, usb_device_id 구조체의 각 필드에 넣어주고, match_flags에는 USB_DEVICE_ID_MATCH_INT_INFO (=0x0380)를 넣어준다. 여기서 USB_DEVICE() 매크로는 idVendor, idProduct 필드를 채우게되며, USB_DEVICE_VER() 매크로는 idVendor, idProduct, bcdDevice_lo, bcdDevice_hi 필드들을 채워줌을 볼 수 있다. 각각에 해당하는 match_flags에는 USB_DEVICE_ID_MATCH_DEVICE (=0x0003)과 USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION (=0x000F)가 들어가게 된다.

USB 디바이스를 동록과 해제는 usb_register()와 usb_unregister()가 처리한다. 넘겨주는 값은 usb_driver 구조체의 포인터이다. 등록되는 동시에 USB driver 리스트에 추가될 것이며, 해당 디바이스가 있는지를 찾게 될 것이다. 찾는 과정에서 앞에서 정의한 probe 필드에 사용된 함수가 호출된다. 따라서, 다음으로 볼 것은 probe 함수이다.

15.2.2. USB Mouse Driver의 Probing

Probe 함수가 하는 역할은 말 그대로 드라이버가 control할 디바이스를 찾는 것이다. 넘겨받은 변수는 usb_device 구조체의 포인터와 Interface 번호, usb_device_id 구조체의 포인터이다.

```
static void *usb_mouse_probe(struct usb_device *dev, unsigned int ifnum,
                           const struct usb_device_id *id)
{
    struct usb_interface *iface;
    struct usb_interface_descriptor *interface;
    struct usb_endpoint_descriptor *endpoint;
    struct usb_mouse *mouse;
    int pipe, maxp;
    char *buf;

    /* Interface의 descriptor를 구한다.*/
    iface = &dev->actconfig->interface[ifnum];
    interface = &iface->altsetting[iface->act_altsetting];
    if (interface->bNumEndpoints != 1) return NULL;

    /* End point의 descriptor를 구한다.*/
    endpoint = interface->endpoint + 0;
    if (!(endpoint->bEndpointAddress & 0x80)) return NULL;
    if ((endpoint->bmAttributes & 3) != 3) return NULL;

    /* 앞에서 구한 end point를 이용해서 receive interrupt를 위한 pipe를 생성한다.*/
    pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
    maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
    usb_set_idle(dev, interface->bInterfaceNumber, 0, 0);

    /* USB mouse driver를 위한 structure를 커널 메모리에서 할당한다.*/
    if (!(mouse = kmalloc(sizeof(struct usb_mouse), GFP_KERNEL))) return NULL;
    memset(mouse, 0, sizeof(struct usb_mouse));
```

코드 782. usb_mouse_probe() 함수의 정의

Probe 함수가 넘겨받는 파라미터 값은 usb_device 구조체의 포인터와, interface 번호, 그리고, usb_device_id 구조체에 대한 포인터이다. 먼저 넘겨받은 usb_device 구조체는 이미 USB subsystem에서 알고 있는 값을 주었기에 이곳에서 해당 interface를 찾는다. 이 interface에서 alternate setting부분을 접근해서 해당 interface값을 얻는다. 만약 interface가 가지는 end point의 값이 10이 아니라면, NULL을 돌려주고 복귀하게 된다. 즉, 초기에는 control을 위한 end point만을 가지고 있기 때문이다. 이전 end point의 디스크립터를 interface 디스크립터로 부터 얻고, 이 end point의 디스크립터가 가지는 end point의 주소를 0x80과 AND 시켜본다. 만약 이 값이 0이 된다면 NULL을 돌려주고 복귀하게 되며, end point가 가지는 attributes 값과 3을 AND 해서 나온것이 3이 아닌 경우에도 NULL을 돌려주고 복귀한다.

이와 관련되어 USB spec.을 잠시 참조하도록 하자. 먼저 mouse 디바이스는 USB spec.의 HID(Human Interface Devices) class에 속한다는 것을 앞에서 이미 보아서 알 것이다. HID class의 spec.에서 mouse와 관련된 spec.을 보면, Endpoint에 대한 주소(bEndpointAddress)의 sample로서 0x80을 사용하고 있으며, Attributes값으로 3을 사용하고 있는 것을 찾을 수 있다.³²¹ 따라서, 여기서는 이러한 값을 비교해서 다른 값이 나오면 return 한다.

`usb_rcvintpipe()`은 upstream에 대한 pipe를 생성하는 함수이다. 넘겨받는 값은 앞에서 찾은 endpoint의 주소와 usb_dev 구조체의 포인터이다. `usb_maxpacket()`은 매크로 정의된 것으로 디바이스에서 데이터의 전송을 위해서 사용할 pipe의 maximum packet의 크기를 구해준다. `usb_pipeout()`은 사용하는 pipe가 OUT(Down Stream)인지 IN(Up Stream)인지를 알려주는 매크로이다.

이전 최대 packet의 수를 정했으며, `usb_set_idle()` 함수는 HID class에서만 사용하는 함수로 새로운 event가 생기거나 명시된 시간 만큼 시간이 지나기전에 Interrupt In pipe에 대해서 특정한 보고를 하지 않도록 만드는, control request를 보내기 위해서 쓴다. 넘겨주는 파라미터 값은 디바이스에 대한 포인터와 interface 번호, 명시된 시간(여기서는 0을 사용했다.), 그리고, report ID 값이다. 여기서 duration을 값이 0이면 endpoint는 report 데이터에서 변화가 감지될 때에만 report를 하며, 다른 report는 무한 시간동안 하지 않는다. Report ID 값이 0이라는 말은 idle rate라는 것이 모든 input의 report에 적용된다는 것을 의미하는 것으로, 0이외의 값은 특정 ID에 대해서 적용된다는 의미이다. Idle rate라는 것은 아무런 일도 일어나지 않는 시간을 말하는 것으로 보통 keyboard의 경우에는 500 milliseconds를, 마우스나 조이스틱에는 무한으로 설정한다. 즉, 이것은 endpoint에서 발생하는 interrupt의 보고 빈도(reporting frequency)를 결정해 주기 위한 것으로서, 마우스에서 데이터의 변화가 감지되는 순간에만 reporting이 되도록 만들어 주는 일을 한다.

```
struct usb_mouse {
    signed char data[8];
    char name[128];
    struct usb_device *usbdev;
    struct input_dev dev;
    struct urb irq;
    int open;
};
```

코드 783. `usb_mouse` 구조체의 정의

`usb_mouse` 구조체는 USB mouse만을 위한 자료 구조이다. `data` 필드는 데이터 전송을 위해서 사용할 buffer로서 8bytes를 할당하고 있으며, 이곳에서 mouse에서 발생한 event들에 대한 내용이 담길 것이다. `name`은 `usb_mouse`의 이름을 보관하는 변수로 “usb_mouse”가 들어간다. `usbdev` 필드는 `usb_device`를 가르키는 포인터이며, `input_dev` 필드는 input 디바이스로 등록하기 위해서 사용한다. Input device에 대해서는 조금 있다가 자세히 알아보기로 하겠다. `irq` 필드는 interrupt의 URB를 가지며, `open`은 `open`된 회수를 기록하는 counter이다. 이 구조체는 probe routine의 끝부분에서 `return` 값으로 돌려주게 되며, 각 디바이스 드라이버에서 사용하게 되는 private한 데이터 구조이다.

```
/* usb_mouse 구조체의 usb_device 구조체 필드와 input_dev 구조체 필드를 만든다.*/
mouse->usbdev = dev;
mouse->dev.evbit[0] = BIT(EV_KEY) | BIT(EV_REL);
mouse->dev.keybit[LONG(BTN_MOUSE)]      =      BIT(BTN_LEFT)      |      BIT(BTN_RIGHT)      |
BIT(BTN_MIDDLE);
mouse->dev.relbit[0] = BIT(REL_X) | BIT(REL_Y);
mouse->dev.keybit[LONG(BTN_MOUSE)] |= BIT(BTN_SIDE) | BIT(BTN_EXTRA);
mouse->dev.relbit[0] |= BIT(REL_WHEEL);
/* input device에서 사용할 private 데이터 구조체와 핸들러(open/close)를 등록한다.
mouse->dev.private = mouse;
mouse->dev.open = usb_mouse_open;
```

³²¹ 물론 HID Class에 대한 spec.을 보면 Appendix E에 Mouse의 endpoint에 대한 descriptor는 하나의 예제이며, 제품마다 다를 가능성도 있다.

```

mouse->dev.close = usb_mouse_close;
/* input 디바이스의 나머지 정보를 기록한다.*/
mouse->dev.name = mouse->name;
mouse->dev.idbus = BUS_USB;
mouse->dev.idvendor = dev->descriptor.idVendor;
mouse->dev.idproduct = dev->descriptor.idProduct;
mouse->dev.idversion = dev->descriptor.bcdDevice;

```

코드 784. **usb_mouse_probe()** 함수의 정의(계속)

자, 이젠 앞에서 생성한 `usb_mouse` 구조체의 각 필드를 채우는 일을 한다. 이 데이터 구조체는 디바이스 구조체의 `private`에 연결되어 각 함수에서 참조하도록 사용할 것이기에 이곳(probe)에서 만들어 주어야 할 것이다.

먼저 `usbdev` 필드를 넘겨받은 `usb_device`의 포인터로 설정하고, `input device`를 나타내는 `dev` 필드를 채워준다. `Input device`에 대한 것은 잠시 후에 알아보기로 하자. 일단 이곳에서 정의하는 값은 `event bit(evbit)`, `key bit(keybit)`, 상대좌표(relbit) 등이며, `input device`의 `private` 데이터 구조체로 앞에서 선언한 `usb_mouse` 구조체를, `open` 함수와 `close` 함수에는 각각 `usb_mouse_open()` 함수와 `usb_mouse_close()` 함수를 주었다. 이외에 디바이스의 이름과 BUS의 ID, vendor ID, product ID, version ID 등은 `usb_device` 구조체의 각 필드 및 `usb_mouse` 구조체에서 사용하는 값으로 넣었다.

```

/* USB의 string descriptor를 얻어와서 기록한다.*/
if (!(buf = kmalloc(63, GFP_KERNEL))) {
    kfree(mouse);
    return NULL;
}
if (dev->descriptor.iManufacturer &&
    usb_string(dev, dev->descriptor.iManufacturer, buf, 63) > 0)
    strcat(mouse->name, buf);
if (dev->descriptor.iProduct &&
    usb_string(dev, dev->descriptor.iProduct, buf, 63) > 0)
    sprintf(mouse->name, "%s %s", mouse->name, buf);
if (!strlen(mouse->name))
    sprintf(mouse->name, "USB HIDBP Mouse %04x:%04x",
           mouse->dev.idvendor, mouse->dev.idproduct);
kfree(buf);
/* Interrupt Transfer type을 위한 URB의 설정 */
FILL_INT_URB(&mouse->irq, dev, pipe, mouse->data, maxp > 8 ? 8 : maxp,
             usb_mouse_irq, mouse, endpoint->bInterval);
/* Input device로 등록한다.*/
input_register_device(&mouse->dev);
printf(KERN_INFO "input%d: %s on usb%d:%d.%d\n",
       mouse->dev.number, mouse->name, dev->bus->busnum, dev->devnum, ifnum);
return mouse;
}

```

코드 785. **usb_mouse_probe()** 함수의 정의(계속)

USB string descriptor를 얻어서 이것을 앞에서 정의한 `usb_mouse` 구조체의 이름으로 만들기 위해서 63bytes의 커널 메모리 공간을 할당 받는다. `usb_device`의 `descriptor`에서 생산자(manufacturer) 값이 존재한다면, `usb_string()` 함수를 이 값을 읽어온다. 읽어온 값은 `usb_mouse` 구조체의 `name` 필드의 뒤에 연결될 것이다. 다시 `usb_device`의 `descriptor`에서 제품(product) 값이 있다면, `usb_string()` 함수를 호출해서 이 값을 읽어오게 되며, 다시 `usb_mouse` 구조체의 `name` 필드에 더해진다. 따라서, 이것으로 USB device name + Manufacturer + Product까지가 정해진 것이다. 이 값을 구할 수 없다면, 앞에서 구한 `idVendor`와 `idproduct`를 가지고 `usb_mouse` 구조체(mouse)의 `name` 필드를 정한다.

`FILL_INT_URB()` 매크로는 interrupt transfer type을 위한 URB(USB Request Block)을 초기화하는 일을 한다. 이전 앞에서 만들어준 `input_dev` 구조체를 이용해서 `input device`로 등록 시키기 위해서 `input_register_device()`를 호출한다. 마지막 복귀값은 `usb_mouse` 구조체이다.

15.2.3. USB Mouse Driver의 Disconnect

`Disconnect`함수는 mouse 디바이스를 probing할 때 해주었던 일을 undo하는 일을 하는 함수이다. 즉, Mouse Device가 USB에서 분리/해제 될 때 호출되는 함수로서, 사용하고 있던 자원들을 반환해야 할 것이다.

```
static void usb_mouse_disconnect(struct usb_device *dev, void *ptr)
{
    struct usb_mouse *mouse = ptr;
    usb_unlink_urb(&mouse->irq);
    input_unregister_device(&mouse->dev);
    kfree(mouse);
}
```

코드 786. `usb_mouse_disconnect()` 함수의 정의

`usb_unlink_urb()` 함수는 현재 진행중인 URB(USB Request Block)을 멈추고 해제시켜줄 것이다. `input_unregister_device()` 함수는 앞에서 `input device`로 등록한 mouse 디바이스의 등록을 해제할 것이며, mouse driver를 위해서 할당했던 `usb_mouse` 구조체의 메모리를 반환한다.

15.2.4. USB Mouse Driver의 Open과 Close

USB mouse 드라이버도 사용하기 위해서는 `open()` 함수를 호출해 주어야 하며, 사용을 마치면, `close()` 함수를 호출해 주어야 한다.

```
static int usb_mouse_open(struct input_dev *dev)
{
    struct usb_mouse *mouse = dev->private;

    if (mouse->open++)
        return 0;
    mouse->irq.dev = mouse->usbdev;
    if (usb_submit_urb(&mouse->irq))
        return -EIO;
    return 0;
}
static void usb_mouse_close(struct input_dev *dev)
{
    struct usb_mouse *mouse = dev->private;

    if (!--mouse->open)
        usb_unlink_urb(&mouse->irq);
}
```

코드 787. `usb_mouse_open()`과 `usb_mouse_close()` 함수의 정의

Input Device로 등록된 USB mouse driver는 앞에서 `input device`로 등록하면서 해준 `open`과 `close`를 위한 함수를 이용해서 `open()`/`close()` 연산을 처리할 것이다. `Open()` 함수는 `open counter`를 유지하기 위해서 `usb_mouse` 구조체의 `open`필드를 증가시켜주며, 만약 이것이 처음으로 `open`을 수행하고 있다면, `interrupt`를 위한 URB 구조체의 `device` 필드를 `usb_device`구조체로 주고, `usb_submit_urb()`는 하위에 있는 USB core 모듈로 앞에서 만든 URB를 전송하라는 명령이 된다. 만약 `return` 값이 0이 아니라면, 에러 값으로 `-EIO`를 돌려준다. 이상 없이 `open`이 수행되었다면 `return` 값은 0이 될 것이다.

Close() 함수는 단순히 앞에서 증가시킨 open의 counter를 감소시킬 것이며, 만약 이 값이 0이 되면, interrupt transfer로 사용하는 URB가 scheduling되어 있는 경우에 취소시킬 목적으로 usb_unlink_urb() 함수를 호출한다.

15.2.5. USB Mouse Interrupt Handler

USB mouse 드라이버는 interrupt에 대해서 이미 USB를 등록 시켜두었다. USB mouse driver의 interrupt handler는 interrupt라는 event의 completion routine으로서 호출되는 함수이다.

```
static void usb_mouse_irq(struct urb *urb)
{
    struct usb_mouse *mouse = urb->context;
    signed char *data = mouse->data;
    struct input_dev *dev = &mouse->dev;

    if (urb->status) return;
    /* Button에 대한 event를 보고한다.*/
    input_report_key(dev, BTN_LEFT,      data[0] & 0x01);
    input_report_key(dev, BTN_RIGHT,     data[0] & 0x02);
    input_report_key(dev, BTN_MIDDLE,   data[0] & 0x04);
    input_report_key(dev, BTN_SIDE,     data[0] & 0x08);
    input_report_key(dev, BTN_EXTRA,   data[0] & 0x10);
    /* Button에 대한 event가 일어난 좌표를 보고한다.*/
    input_report_rel(dev, REL_X,       data[1]);
    input_report_rel(dev, REL_Y,       data[2]);
    input_report_rel(dev, REL_WHEEL,   data[3]);
}
```

코드 788. usb_mouse_irq() 함수의 정의

실제적인 mouse에서 발생하는, input device를 위해서 설정한 event들은 데이터의 변화를 일으킬 것이며, 이것은 다시 앞에서 본 바와 같이 report가 될 것이다. 즉, upstream으로 interrupt transfer가 발생하면, 이것이 reporting이 될 것이며, 앞에서 interrupt transfer type을 위해 정의한 URB의 completion routine인 usb_mouse_irq() 함수가 호출될 것이다. 전달되는 URB의 context는 usb_mouse 구조체이며, 전달 받은 데이터는 이미 usb_mouse 구조체의 data 필드에 있을 것이다. 만약 URB의 status가 0이 아닌 값을 가진다면, USB core 모듈로의 URB의 전달이 잘 못된 경우이다. 따라서, 그냥 return한다. 나머지는 data 필드를 읽어서 해당 event가 일어났는지를 보고하면 input_report_key() 매크로를 이용해서 커널에 보고해주면 될 것이다. input_report_key() 매크로는 앞에서 이미 input device를 설명할 때 보았다.

15.3. Linux USB Keyboard Driver

이번장에서는 USB Keyboard 드라이버의 구현에 대해서 보기로 하겠다. 이미 USB mouse 드라이버의 구현에서 익히 많은 사실들을 알았다고 보고, 이곳에서는 함수들을 위주로 해서 어떻게 구현되고 있는지만을 보기로 하겠다.

15.3.1. USB Keyboard Driver의 등록과 해제

USB keyboard driver를 적재하고 해제하는 함수는 usb_kbd_init()와 usb_kbd_exit() 함수이다. 정의는 ~/drivers/usb/usbkbd.c에 아래와 같이 되어 있다. 이하의 모든 USB keyboard driver와 관련된 함수들을 이곳에서 찾을 수 있을 것이다.

```
static struct usb_device_id usb_kbd_id_table [] = {
    { USB_INTERFACE_INFO(3, 1, 1) },
    { }
};                                     /* Terminating entry */
```

```
MODULE_DEVICE_TABLE(usb, usb_kbd_id_table);

static struct usb_driver usb_kbd_driver = {
    name:          "keyboard",
    probe:         usb_kbd_probe,
    disconnect:   usb_kbd_disconnect,
    id_table:     usb_kbd_id_table,
};

static int __init usb_kbd_init(void)
{
    usb_register(&usb_kbd_driver);
    info(DRIVER_VERSION ":" DRIVER_DESC);
    return 0;
}

static void __exit usb_kbd_exit(void)
{
    usb_deregister(&usb_kbd_driver);
}

module_init(usb_kbd_init);
module_exit(usb_kbd_exit);
```

코드 789

USB keyboard driver를 등록하기 위해서 사용하는 함수는 `usb_register()` 함수이다. 이 함수의 파라미터 값으로 넘어가는 변수는 `usb_device` 구조체로 정의된 `usb_kbd_driver`이다. `usb_kbd_driver`의 각 필드를 보면, `name` 값으로 “`keyboard`”, `probe` 필드로 `usb_kbd_probe()`, `disconnect` 필드로, `usb_kbd_disconnect`를 주고 있다. 또한 사용하는 USB ID table로는 `usb_kbd_id_table`을 주고 있다. 다시 `usb_kbd_id_table`는 HID class로 등록하기 위해서 사용하는 값을 정의해 두었는데, USB class, USB subclass, USB protocol의 값으로 각각 3, 1, 1을 주고 있다. HID class를 위한 3, boot interface를 나타내는 1, 그리고, `keyboard`가 사용하는 protocol값으로 1을 주어서 사용하고 있다.³²² 마지막으로, USB keyboard driver의 해제는 `usb_deregister()` 함수가 처리한다. `usb_register()` 함수는 앞에서와 마찬가지로, `usb_driver` 구조체로 정의된 `usb_kbd_driver`의 `probe()` 함수를 호출 할 것이다.

15.3.2. USB Keyboard Driver의 Probe

USB keyboard driver를 등록할 때 호출되는 함수로 자신이 사용하게될 USB device가 시스템에 있는지를 확인하는 함수이다. 이 함수에서 사용하게될 구조체는 `usb_kbd`이다. 이것을 잠시 살펴보고 넘어가도록 하자.

```
struct usb_kbd {
    struct input_dev dev;           /* Input device로 등록하기 위한 input_dev구조체 */
    struct usb_device *usbdev;      /* Usb device로 등록하기 위한 usb_device구조체 */
    unsigned char new[8];          /* 새로이 전달 받은 keyboard 값 */
    unsigned char old[8];          /* 이미 저장되어 있는 keyboard 값 */
    struct urb irq, led;           /* Interrupt 및 LED를 위한 URB의 정의 */
    devrequest dr;                /* USB device에 대한 요청(request)를 가지는 필드 */
    unsigned char leds, newleds;   /* 원래의 LED값과 새로이 생성된 LED값 */
    char name[128];               /* USB keyboard device의 이름 */
    int open;                      /* Open counter */
};
```

³²² USB HID Class Spec.을 참고하기 바란다.

코드 790. usb_kbd 구조체의 정의

probe() 함수에서는 이와 같이 정의된 usb_kbd 구조체를 메모리를 할당 받아서 생성할 것이며, interrupt나 event시에 이 구조체를 사용하게될 것이다. 또한 input device로의 연결이나, usb device로 등록시킨 값을 저장하는데도 사용된다.

```
typedef struct {
    __u8 requesttype;          /* 요청 type */
    __u8 request;              /* 요청 */
    __u16 value;               /* 요청이 가지는 값 */
    __u16 index;               /* 요청의 HID Class에서는 Interface ID를 나타낸다. */
    __u16 length;              /* 전송될 데이터의 길이 */
} devrequest __attribute__((packed));
```

코드 791. devrequest 구조체의 정의

devrequest 구조체는 ~/include/linux/usb.h에 정의되어 있는 것으로 USB device에 대해서 host controller가 요청(request)를 보내기 위해서 사용하는 구조체이다. 이 구조체는 descriptor를 읽거나, 설정할 때, report를 구하거나, 설정 할 때 및 idle rate에 대한 조정(get/set)등에 사용되며, 또한 protocol 정보를 얻거나 설정할 때도 사용한다.

이제 실제적인 USB keyboard에 대한 probe를 실행하는 usb_kbd_probe() 함수를 보도록 하자. 이 함수에서 하는 일은 자신이 원하는 USB keyboard device가 있는지를 확인하고, 앞에서 정의한 usb_kbd 구조체를 만들어 주는 것과, input device로 등록하는 일이다.

```
static void *usb_kbd_probe(struct usb_device *dev, unsigned int ifnum,
                           const struct usb_device_id *id)
{
    struct usb_interface *iface;
    struct usb_interface_descriptor *interface;
    struct usb_endpoint_descriptor *endpoint;
    struct usb_kbd *kbd;
    int i, pipe, maxp;
    char *buf;

    /* USB device의 active configuration의 interface정보를 가져온다.*/
    iface = &dev->actconfig->interface[ifnum];
    interface = &iface->altsetting[iface->act_altsetting];

    /* Interface를 이용해서 endpoint에 대한 정보를 가져온다.*/
    if (interface->bNumEndpoints != 1) return NULL;
    endpoint = interface->endpoint + 0;
    if (!(endpoint->bEndpointAddress & 0x80)) return NULL;
    if ((endpoint->bmAttributes & 3) != 3) return NULL;

    /* Endpoint를 이용해서 receive를 위한 pipe를 설정한다.*/
    pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
    maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));

    /* USB keyboard의 protocol을 설정한다.*/
    usb_set_protocol(dev, interface->bInterfaceNumber, 0);
    /* Idle rate를 설정한다.*/
    usb_set_idle(dev, interface->bInterfaceNumber, 0, 0);

    /* usb_kbd 구조체를 할당하고, 0으로 초기화 한다.*/
}
```

```

if (!(kbd = kmalloc(sizeof(struct usb_kbd), GFP_KERNEL))) return NULL;
memset(kbd, 0, sizeof(struct usb_kbd));
/* usb_kbd 구조체의 usb_device 포인터 필드를 초기화 한다.*/
kbd->usbdev = dev;

```

코드 792. usb_kbd_probe() 함수의 정의

먼저, 현재 active로 설정된 configuration을 읽어서 interface number(ifnum)에 대한 interface를 찾는다. 이렇게 찾은 interface의 alternate setting을 altsetting[]에 대한 index로 사용해서 우리가 원하는 실제적인 interface를 찾게 된다. 이젠 이렇게 해서 찾은 interface 번호를 이용해서 사용할 endpoint를 찾을 차례이다. 만약 interface가 가진 endpoint의 갯수가 1이 아니라면 NULL을 돌려주도록 한다. 즉, 우리가 USB keyboard를 위해서 사용하고자 하는 endpoint는 하나가 전부라는 말이된다. 이렇게 찾은 인터페이스의 첫번째 endpoint를 endpoint 변수로 두고, 다시 이 endpoint가 가지는 주소에 0x80을 AND 시켜본다. 만약 0이 나오게 되면 NULL을 돌려주게 되며, 또한 endpoint의 attributes 값과 3을 AND한 값이 3과 같지 않았을 때도 NULL을 돌려주게 된다.³²³

usb_rcvintpipe()는 upstream에 대한 pipe를 생성하는 함수이다. 넘겨받는 값은 앞에서 찾은 endpoint의 주소와 usb_dev 구조체의 포인터이다. usb_maxpacket()은 매크로로 정의된 것으로 디바이스에서 데이터의 전송을 위해서 사용할 pipe의 maximum packet의 크기를 구해준다. usb_pipeout()은 사용하는 pipe가 OUT(Down Stream)인지 IN(Up Stream)인지를 알려주는 매크로이다.³²⁴

interrupt를 위한 pipe의 설정이 이제 끝났으며, 이젠 사용할 protocol을 설정하기 위해, usb_set_protocol() 함수를 이용해서 설정하고자 하는 protocol(=0)과 함께 해당 interface에 대해서 호출한다. 역시 idle rate에 대한 설정을 위해서는 usb_set_idle() 함수를 사용하고 있다.³²⁵

이젠 input device driver로 등록하는 절차가 남았다. 일단 usb_kbd 구조체를 커널 메모리로 부터 할당 받도록 한다(kmalloc()). 그리고나서, 할당 받은 메모리를 전부 0으로 초기화하고, 등록된 USB device를 input device를 나타내는 kbd의 usbdev 필드에 넣어 주도록 한다.

```

/* Event bit을 설정한다.*/
kbd->dev.evbit[0] = BIT(EV_KEY) | BIT(EV_LED) | BIT(EV_REP);
kbd->dev.ledbit[0] = BIT(LED_NUML) | BIT(LED_CAPSL) | BIT(LED_SCROLLL) | 
BIT(LED_COMPOSE) | BIT(LED_KANA);
/* Key code를 설정한다.*/
for (i = 0; i < 255; i++)
    set_bit(usb_kbd_keycode[i], kbd->dev.keybit);
clear_bit(0, kbd->dev.keybit);
/* USB keyboard 구조체를 초기화 한다.*/
kbd->dev.private = kbd;
kbd->dev.event = usb_kbd_event;
kbd->dev.open = usb_kbd_open;
kbd->dev.close = usb_kbd_close;
/* 사용할 URB를 설정한다.*/
FILL_INT_URB(&kbd->irq, dev, pipe, kbd->new, maxp > 8 ? 8 : maxp,
            usb_kbd_irq, kbd, endpoint->bInterval);
/* USB keyboard의 request 구조체를 초기화 한다.*/

```

³²³ 이와 관련되어 USB spec.을 잠시 참조하도록 하자. 먼저 keyboard 디바이스는 USB spec.의 HID(Human Interface Devices) class에 속한다는 것을 앞에서 이미 보아서 알 것이다. HID class의 spec.에서 keyboard와 관련된 spec.을 보면, Endpoint에 대한 주소(bEndpointAddress)의 sample로서 0x81을 사용하고 있으며, Attributes 값으로 3을 사용하고 있는 것을 찾을 수 있다. 하지만, 실제 코드에서는 단지 0x80을 사용해서 AND 시켜준 값으로 이것을 검사하고 있다.

³²⁴ 실제로 이것은 pipe를 기술하는 field의 특정 부분에 direction을 나타내는 bit를 설정하는 일을 하는 매크로이다. 즉, 0으로 설정된 경우에는 host에서 device로의 output을 만드는 것이고, 1로 된 경우에는 device에서 host로의 input을 만드는 경우가 된다.

³²⁵ USB mouse device driver를 참고하기 바란다.

```

kbd->dr.requesttype = USB_TYPE_CLASS | USB_RECIP_INTERFACE;
kbd->dr.request = USB_REQ_SET_REPORT;
kbd->dr.value = 0x200;
kbd->dr.index = interface->bInterfaceNumber;
kbd->dr.length = 1;
/* USB keyboard의 input device 구조체 부분의 나머지를 초기화 한다.*/
kbd->dev.name = kbd->name;
kbd->dev.idbus = BUS_USB;
kbd->dev.idvendor = dev->descriptor.idVendor;
kbd->dev.idproduct = dev->descriptor.idProduct;
kbd->dev.idversion = dev->descriptor.bcdDevice;
/* USB descriptor를 가져올 buffer를 할당한다.*/
if (!(buf = kmalloc(63, GFP_KERNEL))) {
    kfree(kbd);
    return NULL;
}

```

코드 793. `usb_kbd_probe()` 함수의 정의(계속)

이전 USB keyboard device driver가 관심을 가지는 event에 대한 bit을 설정할 차례이다. EV_KEY, EV_LED, EV_REP event들에 대해서 설정해 주도록 한다. 또한 LED와 관련된 bit으로는 LED_NUML, LED_CAPSL, LED_SCROLL을 설정하며, 또한 LED_COMPOSE 및 LED_KANA도 설정하도록 한다.³²⁶ 이것을 마치면, USB keyboard의 keycode를 나타내는 255개의 bit을 설정하도록 한다. 여기서, `set_bit()`의 첫번째 인자는 설정하고자 하는 것이 가르키고 있는 데이터 구조체의 offset을 나타낸다고 생각하면 될 것이다. 그것의 반대 작용을 하는 것이 `clear_bit()`이다. 다 채워주고나서 첫번째 bit은 지워주었다.

이전 `usb_kbd` 구조체의 input device와 관련된 부분을 초기 설정한다. Private 데이터로 `kbd` 자체를 가르켜주고, `event`는 `usb_kbd_event()` 함수를, `open`은 `usb_kbd_open()` 함수를, `close()` 함수는 `usb_kbd_close()` 함수로 설정한다. 이것을 마치고 나면, 사용하게 될 URB 구조체를 설정하기 위해서 `FILL_INT_URB()` 매크로를 사용한다. 즉, `interrupt`에 대한 URB를 설정하는 것이다. `interrupt`를 처리해 주는 함수로 `usb_kbd_irq()` 함수를 사용하고 있음을 볼 수 있다.

앞에서 이야기 했던 `devrequest` 구조체는 `~/include/linux/usb.h`에 정의되어 있는 것으로 USB device에 대해서 host controller가 요청(request)를 보내기 위해서 사용하는 구조체이다. `requesttype`으로 `USB_TYPE_CLASS | USB_RECIP_INTERFACE`를 두고 있고, `request`로는 `USB_REQ_SET_REPORT`를, `value`값으로 `0x200`을, `index`로는 `interface`의 `bInterfaceNumber`를, 나머지 `length`에는 `1`을 주고 있다. 이것은 나중에 USB keyboard device에 control information을 전달하고자 할 때 사용할 것이다. 여기서는 일단 Set Report에 대한 `request`를 설정한다고만 생각하도록 하자.³²⁷

나머지는 앞에서 설정을 다 해주지 않은 input device의 `name`, `idbus`, `idvendor`, `idproduct`, `idversion`등의 필드를 USB descriptor의 정보나 상수 값으로 정의해 주는 것이다. 이것을 마치고나면, USB string descriptor를 읽어오기 위한 `buffer`를 할당한다(`buf`).

```

/* USB string descriptor를 읽어오도록 한다.*/
if (dev->descriptor.iManufacturer &&
    usb_string(dev, dev->descriptor.iManufacturer, buf, 63) > 0)
    strcat(kbd->name, buf);
if (dev->descriptor.iProduct &&
    usb_string(dev, dev->descriptor.iProduct, buf, 63) > 0)
    sprintf(kbd->name, "%s %s", kbd->name, buf);
if (!strlen(kbd->name))
    sprintf(kbd->name, "USB HIDBP Keyboard %04x:%04x",
            kbd->dev.idvendor, kbd->dev.idproduct);
kfree(buf);

```

³²⁶ LED_COMPOSE와 LED_KANA는 각각 keyboard의 하위에 있는 compose bit과 한자(kana)를 의미한다고 생각된다. 하지만, 이와 관련된 정확한 정보는 아직 찾을 수 없다.

³²⁷ 자세한 것을 보려면, USB HID Class spec.에 있는 7절의 request부분을 보도록 하라.

```

/* USB set report request에 사용할 URB를 생성한다.*/
FILL_CONTROL_URB(&kbd->led, dev, usb_sndctrlpipe(dev, 0),
                 (void*) &kbd->dr, &kbd->leds, 1, usb_kbd_led, kbd);
/* Input device로 등록한다.*/
input_register_device(&kbd->dev);

printk(KERN_INFO "input%d: %s on on usb%d:%d.%d\n",
       kbd->dev.number, kbd->name, dev->bus->busnum, dev->devnum, ifnum);
return kbd;
}

```

코드 794. **usb_kbd_probe()** 함수의 정의(계속)

앞에서 초기화해준 각종의 string정보에 추가적으로 정보를 덧붙여준다. 바뀌는 부분은 input device의 name이다. 이를 위해서 추가적으로 할당했던 buffer는 해제해 준다(kfree()). 이전 USB keyboard의 set report request를 위한 URB를 만들어준다. 사용되는 complete routine은 `usb_kbd_led()` 함수이다. 마지막으로 복귀하기 전에, `input_register_device()` 함수를 호출해서 앞에서 초기화한 `input_dev` 구조체를 넘겨주어 input device로의 등록을 마치게 된다. 복귀 값은 생성한 `usb_kbd` 구조체의 포인터이다.

15.3.3. USB Keyboard Driver의 Disconnect

USB keyboard device가 제거될 때, 호출되는 함수가 `disconnect` 함수이다. 따라서, 이 함수에서는 진행중인 URB를 cancel하고, input device로 등록된 USB keyboard 디바이스를 해지(unregister) 시켜야 한다. 아래와 같다.

```

static void usb_kbd_disconnect(struct usb_device *dev, void *ptr)
{
    struct usb_kbd *kbd = ptr;
    usb_unlink_urb(&kbd->irq);
    input_unregister_device(&kbd->dev);
    kfree(kbd);
}

```

코드 795. **usb_kbd_disconnect()** 함수의 정의

즉, `usb_unlink_urb()` 함수를 호출해서 현재 진행중인 URB들을 취소(cancel) 시켜주고, `input_unregister_device()` 함수를 호출해서 input device 등록을 해지한다. 마지막으로 할당받았던 구조체를 해제하고 위해서 `kfree()`를 호출한다. 하지만, 이것을 마쳤다고 해서, 완전히 모듈이 해제된 것이 아니므로, 주의하기 바란다.

15.3.4. USB Keyboard Driver의 Open과 Close

모든 디바이스가 그렇듯이 사용되기 위해서는 `open`되어야 하며, 사용을 마치면 `close`되어야 한다. 이것은 Unix와 같은 시스템의 기준에서 보면, 디바이스란 일종의 파일처럼 인식이 되기 때문이다. 일반 파일에 대한 접근과 같이 디바이스를 사용하기 때문이다.

```

static int usb_kbd_open(struct input_dev *dev)
{
    struct usb_kbd *kbd = dev->private;

    if (kbd->open++)
        return 0;
    kbd->irq.dev = kbd->usbdev;
    if (usb_submit_urb(&kbd->irq))
        return -EIO;
    return 0;
}

```

```
static void usb_kbd_close(struct input_dev *dev)
{
    struct usb_kbd *kbd = dev->private;

    if (!--kbd->open)
        usb_unlink_urb(&kbd->irq);
}
```

코드 796. **usb_kbd_open()**와 **usb_kbd_close()** 함수의 정의

usb_kbd_open() 함수는 open된 회수를 가지는 open 필드를 증가시켜주고, 만약 이것이 처음 open되는 것이라면, interrupt를 전송을 위해서 사용할 URB의 device필드를 초기화 시키고, 이 URB를 USR core 모듈로 전달한다(**usb_submit_urb()**). 이 전달에서 오류가 있었다면, -EIO를 돌려준다. 복귀 값은 0이다. **usb_kbd_close()** 함수는 앞에서 했던 **open()** 함수에서 했던 일을 되돌리는 일을 하게되며, 먼저 open counter를 감소시키고, 만약 이 값이 0이 된다면, 스케줄링된 request(URB)가 있다면, 이것이 끝나기 전에 취소시켜준다.

15.3.5. USB Keyboard Driver의 Event 처리

USB keyboard와 관련된 event를 처리하는 함수는 **usb_kbd_event()**이다. 이 함수를 등록 시키는 것은 probe() 함수에서 이미 보았다. LED에 대한 event 처리를 맡고 있다.

```
int usb_kbd_event(struct input_dev *dev, unsigned int type, unsigned int code, int value)
{
    struct usb_kbd *kbd = dev->private;

    if (type != EV_LED) return -1;
    kbd->newleds = (!test_bit(LED_KANA, dev->led) << 3) | (!test_bit(LED_COMPOSE, dev->led) << 3)
    |
    |           (!test_bit(LED_SCROLLL, dev->led) << 2) | (!test_bit(LED_CAPSL, dev->led) << 1)
    |
    |           (!test_bit(LED_NUML, dev->led));
    if (kbd->led.status == -EINPROGRESS)
        return 0;
    if (kbd->leds == kbd->newleds)
        return 0;
    kbd->leds = kbd->newleds;
    kbd->led.dev = kbd->usbdev;
    if (usb_submit_urb(&kbd->led))
        err("usb_submit_urb(leds) failed");
    return 0;
}
```

코드 797. **usb_kbd_event()** 함수의 정의

usb_kbd_event() 함수는 event type이 EV_LED가 아니면, 곧바로 -1을 돌려주고 복귀한다. 새로운 LED값을 결정하기 위해서 전달받은 LED값(dev->led)과 해당 bit이 설정되었는지를 비교한다. 이때 비교되는 bit들은 LED_KANA, LED_COMPOSE, LED_SCROLLL, LED_CAPSL, LED_NUML이 있다. 각각의 bit에 대한 정의는 아래와 같다.

LED bit	Value	설명
LED_NUML	0x00	Numeric Lock
LED_CAPSL	0x01	Capital Lock
LED_SCROLLL	0x02	Scroll Lock
LED_COMPOSE	0x03	Compose Key

LED_KANA	0x04	일본어(Kana) Key ³²⁸
LED_MAX	0x0F	LED 값이 가지는 최대 값(15). 단지 bit단위로 할당 받기 위해서 사용함.

코드 798. LED status bit 정의

만약 현재 LED의 상태(status) -EINPROGRESS³²⁹라면 곧바로 return하게 되며, 현재의 LED 값과 새로운 LED 값이 같다면, 반영을 해줄 필요가 없으므로 복귀한다(return). 다르다면, 현재의 LED 값을 바꾸고(kbd->leds), LED URB를 위한 디바이스를 설정한다음, LED의 URB를 USB core로 전달 하기위해서, usb_submit_urb()를 사용한다. 전송에서 오류가 있었다면, err() 함수를 호출해서 오류를 보고하고, 0을 복귀 값으로 돌려준다.

15.3.6. USB Keyboard Interrupt Handler

USB keyboard에서 발생하는 interrupt의 complete() 함수로 불려지게 되는 함수는 usb_kbd_irq() 함수이다. 이것을 등록하는 부분은 probe() 함수에서 이미 보았다.

```
static void usb_kbd_irq(struct urb *urb)
{
    struct usb_kbd *kbd = urb->context;
    int i;

    if (urb->status) return;
    for (i = 0; i < 8; i++)
        input_report_key(&kbd->dev, usb_kbd_keycode[i + 224], (kbd->new[0] >> i) & 1);
    for (i = 2; i < 8; i++) {
        if (kbd->old[i] > 3 && memscan(kbd->new + 2, kbd->old[i], 6) == kbd->new + 8) {
            if (usb_kbd_keycode[kbd->old[i]])
                input_report_key(&kbd->dev, usb_kbd_keycode[kbd->old[i]], 0);
            else
                info("Unknown key (scancode %#x) released.", kbd->old[i]);
        }
        if (kbd->new[i] > 3 && memscan(kbd->old + 2, kbd->new[i], 6) == kbd->old + 8) {
            if (usb_kbd_keycode[kbd->new[i]])
                input_report_key(&kbd->dev, usb_kbd_keycode[kbd->new[i]], 1);
            else
                info("Unknown key (scancode %#x) pressed.", kbd->new[i]);
        }
    }
    memcpy(kbd->old, kbd->new, 8);
}
```

코드 799

URB의 status를 보고 이 값이 0이외의 값을 가진다면, 아직 processing이되고 있다는 것으로 그냥 복귀하도록 한다. 그렇지 않다면, 제대로 처리를 마쳤다는 것이 되므로 아래로 진행한다. URB의 context에는 probe() 함수에서 이미 usb_kbd 구조체를 넣어주었으므로 이곳에서 넘겨 받을 수 있을 것이다(kbd). usb_kbd 구조체의 new[]에는 새로운 문자가 저장되어 있을 것이므로, keycode와 새로이 입력받은 값을 input_report_key()를 호출해서 event로 알려주도록 한다. 이것은 input device 드라이버 부분에서 input_event() 부분에서 처리가 될 것이다. memscan()은 주어진 특정한 길이의 문자열에서 해당하는 문자를 찾는 inline함수로 architecture에서 지원하는 경우와 그렇지 않은 경우에 대해서 정의를 달리하는 함수이며, 정의는 ~/include/asm/string.h에 있다. 여기서는 새로운 key code 값과 이전에 있던 key code 값이 일치하는지를 찾기 위해서 사용하고 있다. 이곳에서 사용하고 있는 원리는 먼저 이전에

³²⁸ 현재로서는 추측할 뿐이다. 정확한 자료를 아직 찾지 못하고 있다.

³²⁹ ~/include/asm/errno.h에 115로 정의됨. 여기서는 URB가 pending된 상태임을 나타낸다. 즉, 처리중이다.

보관하고 있던 키 값과 같다면, 해당 key가 release되었다는 것을 말하며, 그렇지 않다면 새로운 key가 눌려졌다는 것을 알려준다는 것이다.³³⁰ 복귀하기 전에 새로 입력을 받은 key code값을 old로 복사하도록 한다(memcpy()).

15.3.7. USB Keyboard Driver의 LED 처리

USB Keyboard 드라이버는 keyboard의 LED 상태 변화를 감지해서 이를 처리해 주어야 할 것이다. 이와 같은 역할을 맡고 있는 함수가 usb_kbd_led() 함수이며, 이미 이에 관련된 URB는 probe() 함수에서 생성해 주었다.

```
static void usb_kbd_led(struct urb *urb)
{
    struct usb_kbd *kbd = urb->context;

    if (urb->status)
        warn("led urb status %d received", urb->status);
    if (kbd->leds == kbd->newleds)
        return;
    kbd->leds = kbd->newleds;
    kbd->led.dev = kbd->usbdev;
    if (usb_submit_urb(&kbd->led))
        err("usb_submit_urb(leds) failed");
}
```

코드 800. usb_kbd_led() 함수의 정의

URB의 상태(status)를 먼저 확인해서 이것이 문제가 있다면, 경고 메시지를 쓰게되며(warn()), 새운 LED값과 현재 가지고 있는 LED값을 비교해서 같다면, 곧바로 return 한다. 다르다면, 이를 keyboard에 반영해 주기 위해서 원래의 LED값을 새로운 LED값으로 고치고, LED를 위한 URB전송하기 위해서 usb_kbd 구조체의 led URB 필드에 usb_device구조체를 넣어준 후, urb_submit_urb()를 호출해서 USB core로 URB를 전달한다. 만약 에러가 있다면, 에러 메시지를 출력해준다(err()).

³³⁰ 아직 자세히 어떻게 되는지를 확인하지 못하고 있다. 하지만, 원리상에서는 이렇게 처리한다고 생각하면 간단하게 해결 될 수 있을 것이다.

16. Booting

16.1. Booting이란?

모든 컴퓨터는 실행할 프로그램을 어떤 형식으로든 물리적인 메모리 상으로 가져와야 한다. 즉, secondary storage에 있는 프로그램은 그 이미지 그대로는 실행되지 않고, 물리적인 메모리 상으로 올라와야지만 실행 될 수 있다. 운영 체제 역시 하나의 프로그램이며, 이것도 실행되기 위해서는 물리적인 메모리 상에 올라와야 할 것이다. 하지만 일반적인 프로그램과는 다르게 운영체제가 실행되기 위해서는 어떤 도움도 다른 프로그램으로부터 받을 수가 없다는 것이다. 이러한 운영체제를 메모리상의 적절한 위치에 올리고 실행 하는 것을 booting이라고 하며, computer의 기본적인 기능을 이용해서 가능해진다.

리눅스에서는 Lilo라는 프로그램이 운영체제를 기동 시켜주는 역할을 하고 있으며, lilo는 단순히 운영체제를 물리적인 메모리로 올리고, 운영체제로 control을 넘겨주는 것을 담당하고 있다. 이후에는 이러한 절차에 대해서 살펴볼 것이며, 실제코드의 어떤 부분에서 어떤 처리가 일어나는지를 볼 것이다.

16.2. bootsect.S의 분석

bootsect.S³³¹는 BIOS의 startup routine에 의해서 자동으로 0x7C00에 load된다. Load된 후 이것은 다시 자신을 0x90000으로 옮겨놓게 되며, 이곳으로 Jump해서 실행된다. 그리고 나서 bootsect.S는 setup을 자신의 바로 뒤(0x90200)로 불러드리며, system을 0x10000으로 BIOS interrupt를 이용해서 읽어오게 된다.

```
#include <linux/config.h>
#include <asm/boot.h>

SETUPSECS      = 4           /* default nr of setup-sectors */
BOOTSEG        = 0x07C0      /* original address of boot-sector */
INITSEG        = DEF_INITSEG /* we move boot here - out of the way */
SETUPSEG        = DEF_SETUPSEG /* setup starts here */
SYSSEG         = DEF_SYSSEG /* system loaded at 0x10000 (65536) */
SYSSIZE        = DEF_SYSSIZE /* system size: # of 16-byte clicks */
                           /* to be loaded */
ROOT_DEV       = 0           /* ROOT_DEV is now written by "build" */
SWAP_DEV       = 0           /* SWAP_DEV is now written by "build" */

#ifndef SVGA_MODE
#define SVGA_MODE ASK_VGA
#endif

#ifndef RAMDISK
#define RAMDISK 0
#endif

#ifndef CONFIG_ROOT_RDONLY
#define CONFIG_ROOT_RDONLY 1
#endif
```

코드 801. bootsect.S(1)

³³¹ 현재의 kernel version은 2.4.0 test 9이며, 이 문서에서 참조되는 것은 그때 그때의 version을 맞춰서 진행된다고 생각하기 바란다. 왜냐하면 워낙 빠른 patch가 가해져서, 문서를 작성하는 시점과 최신 버전을 다루고자 하는 노력이 안 맞을 수 있기 때문이다

SETUPSECS는 default setup sector의 개수를 나타낸다. BOOTSEC는 boot sector의 시작주소를, INITSEG는 initialize code가 시작되는 주소로, bootsect.S가 새로이 옮겨가는 주소이다. SETUPSEG은 setup0이 load될 주소이며, SYSSEG은 system0이 load될 주소이다. 또한 SYSSIZE는 16 byte 단위의 system size를 나타낸다.

```
#define DEF_INITSEG      0x9000
#define DEF_SYSSEG        0x1000
#define DEF_SETUPSEG       0x9020
#define DEF_SYSSIZE        0x7F00

/* Internal svga startup constants */
#define NORMAL_VGA          0xfffff      /* 80x25 mode */
#define EXTENDED_VGA         0xffffe      /* 80x50 mode */
#define ASK_VGA              0xffffd      /* ask for it at bootup */
```

코드 802. include/asm/boot.h

상수 값들에 대한 정의는 include/asm/boot.h를 보면 될 것이다. ROOT_DEV 및 SWAP_DEV는 아직 build되지 않은 상황에서는 정의가 되지 않으며, 이는 build하는 과정에서 device를 ROOT 혹은 SWAP device로 사용할 지에 관련되어 있다. 이곳까지의 메모리 이미지는 [그림73]과 같다.

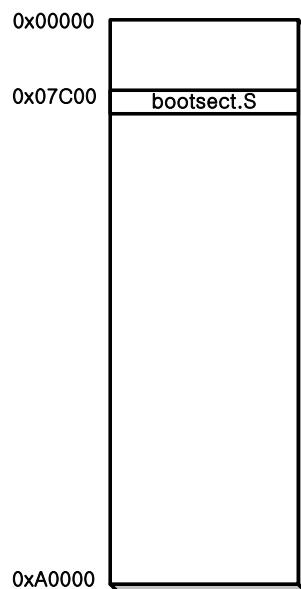


그림 97. bootsect.S의 RAM image(1)

시스템은 아직까지는 x86의 protected mode에서 작동하는 것이 아니라, real mode에서 작동하기에 16bit address만을 할 수 있다. 따라서,.code16이라는 라인이 따라오게 된다.

```
.code16
.text

.global _start
_start:

#if 0 /* hook for debugger, harmless unless BIOS is fussy (old HP) */
    int     $0x3
#endif

    movw   $BOOTSEG, %ax
```

```

movw    %ax, %ds
movw    $INITSEG, %ax
movw    %ax, %es
movw    $256, %cx
subw    %si, %si
subw    %di, %di
cld
rep
movsw
ljmp   $INITSEG, $go

```

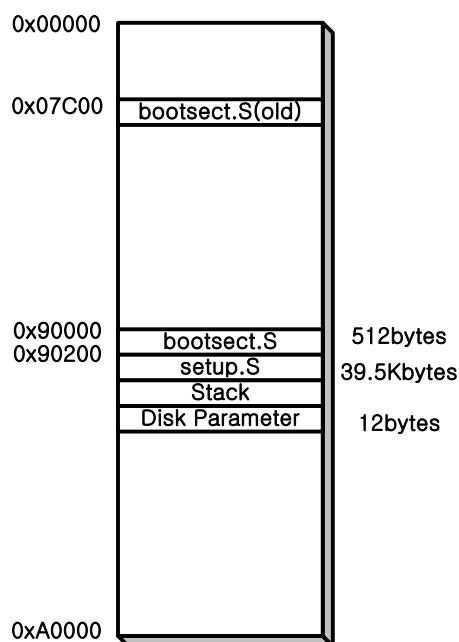
코드 803. bootsect.S(2)

bootsect.S는 최대 512bytes만은 차지 하므로 256번만 word단위(16bit=2byte)로 옮겨주면 된다. 즉, BOOTSEC로부터 INITSEG로 512bytes를 옮기고, INITSEG로 jump를 하게 된다. 이때, label로 go를 주어서 어디서부터 다시 시작하는지를 기록하며 direction flag를 clear한다(CLD).

go:	movw \$0x4000-12, %di	# 0x4000 is an arbitrary value >=
		# length of bootsect + length of
		# setup + room for stack;
		# 12 is disk parm size.
	movw %ax, %ds	# ax and es already contain INITSEG
	movw %ax, %ss	
	movw %di, %sp	# put stack at INITSEG:0x4000-12.

코드 804. bootsect.S(3)

옮겨진 코드의 시작에서 setup의 수행에 필요한 공간을 예약을 하게된다. 여기서 0x4000이란 값은 bootsect와 setup의 크기, 그리고, stack을 위한 공간을 예약한다. 12라는 값을 0x4000에서 빼는 것은 disk parameter를 위한 저장 공간으로 사용하기 위해서이다. AX및 ES는 위에서 이미 INITSEG를 가리키고 있다. [그림74]는 지금까지의 결과로 나타나는 메모리 image를 보여준다.

**그림 98. bootsect.S RAM Image(2)**

현재의 segment register인 ds, es, ss는 cs - INITSEG를 가리키고 있으며, fs = 0을 그리고, gs는 사용되지 않고 있다. 다음으로 하는 일은 multi-sector를 읽기 위한 동작이다. 즉, setup부분을 읽어드리는 과정이 되겠다. 이것은 setup부분이 하나의 sector에 들어가지 않기에, 여러 sector를 읽어야만 하기 때문이다. 먼저 sector count에 대한 patch를 한 다음, FDC를 BIOS interrupt 0x13을 이용해 reset하고, 다시 setup sector를 BIOS service interrupt 0x13을 이용해서 0x90200(\$INITSEG + 0x200)으로 읽어 들인다. 이때 최대 sector count로 36을 사용해서 patch하며, load가 끝나면 disk drive의 parameter들을 구하게 된다. 여기서 알게 되는 정보가 track당 sector의 수가 되겠다..

```

load_setup:
    xorb    %ah, %ah          # reset FDC
    xorb    %dl, %dl
    int     $0x13
    xorw    %dx, %dx          # drive 0, head 0
    movb    $0x02, %cl          # sector 2, track 0
    movw    $0x0200, %bx          # address = 512, in INITSEG
    movb    $0x02, %ah          # service 2, "read sector(s)"
    movb    setup_sects, %al      # (assume all on head 0, track 0)
    int     $0x13          # read it
    jnc    ok_load_setup        # ok - continue

    pushw   %ax          # dump error code
    call    print_nl
    movw    %sp, %bp
    call    print_hex
    popw   %ax
    jmp    load_setup

```

```
ok_load_setup:
```

코드 805. bootsect.S(4)

BIOS에서 제공하는 track당 sector의 크기를 재는 function가 없기에 36, 18, 15, 9의 순서대로 sector를 읽어보는 방법을 썼다. 즉, 이렇게 해서 읽어지는 값으로 track당 sector의 값을 정한다.

```

        movw   $disksizes, %si          # table of sizes to try
probe_loop:
    lodsb
    cbtw          # extend to word
    movw   %ax, sectors
    cmpw   $disksizes+4, %si
    jae    got_sectors          # If all else fails, try 9

    xchgw   %cx, %ax          # cx = track and sector
    xorw    %dx, %dx          # drive 0, head 0
    xorb    %bl, %bl
    movb    setup_sects, %bh
    incb   %bh
    shlb    %bh          # address after setup (es = cs)
    movw    $0x0201, %ax          # service 2, 1 sector
    int     $0x13
    jc     probe_loop          # try next value

```

코드 806. bootsect.S(5)

[코드]에서 보듯이 disksize의 값은 36, 18, 15, 9란 값을 가지게 되며, 직접 읽은 값과의 비교를 통해서 sector의 값을 주게 된다. 이렇게 알려진 disk의 parameter정보들은 나중에 사용된다.

```
got_sectors:
    movw $INITSEG, %ax
    movw %ax, %es          # set up es
    movb $0x03, %ah        # read cursor pos
    xorb %bh, %bh
    int $0x10
    movw $9, %cx
    movw $0x0007, %bx     # page 0, attribute 7 (normal)
    movw $msg1, %bp
    movw $0x1301, %ax     # write string, move cursor
    int $0x10              # tell the user we're loading..
    movw $SYSSEG, %ax      # ok, we've written the message, now
    movw %ax, %es          # we want to load system (at 0x10000)
    call read_it
    call kill_motor
    call print_nl
```

코드 807. bootsect.S(6)

[코드]에서 하는 일은 debugging message에 해당하는 정보를 display하는 것이다. 먼저 cursor의 위치를 받고 난 후, 그곳에 ‘Loading’이라는 메시지를 쓴다. *read_it*함수에서는 이제 실제로 system을 읽어오는 역할을 하게 된다. 여기서 함수 *read_it*를 좀더 자세히 보도록 하자.

```
sread: .word 0                      # sectors read of current track
head:   .word 0                      # current head
track:  .word 0                      # current track

read_it:
    movb setup_sects, %al
    incb %al
    movb %al, sread
    movw %es, %ax
    testw $0x0fff, %ax
die:   jne die                      # es must be at 64kB boundary
       xorw %bx, %bx                  # bx is starting address within segment
rp_read:
#ifdef __BIG_KERNEL__
    bootsect_kludge = 0x220          # 0x200 (size of bootsector) + 0x20 (offset
    lcall bootsect_kludge            # of bootsect_kludge in setup.S)
#else
    movw %es, %ax
    subw $SYSSEG, %ax
#endif
    cmpw syssize, %ax                # have we loaded all yet?
    jbe ok1_read
    ret
```

코드 808. bootsect.S(7)

이 routine에서 하는 일은 system을 0x10000으로 불러들이는 역할을 한다. 주의할 점은 64k boundary를 넘어서지 않는 것이다. 또한 될 수 있으면 전체 track을 한번에 loading한다.

다시 원래의 routine으로 돌아가서, root device가 defined되었는지를 확인하고, 만약 define되지 않았다면, 앞에서 읽어 들여온 parameter정보를 root device로 넘겨준 다음, setup.S로 jump한다.

```

movw    root_dev, %ax
orw    %ax, %ax
jne    root_defined

movw    sectors, %bx
movw    $0x0208, %ax          # /dev/ps0 - 1.2Mb
cmpw    $15, %bx
je     root_defined

movb    $0x1c, %al            # /dev/PS0 - 1.44Mb
cmpw    $18, %bx
je     root_defined

movb    $0x20, %al            # /dev/fd0H2880 - 2.88Mb
cmpw    $36, %bx
je     root_defined

movb    $0, %al               # /dev/fd0 - autodetect
root_defined:
    movw    %ax, root_dev

# After that (everything loaded), we jump to the setup-routine
# loaded directly after the bootblock:

        ljmp   $SETUPSEG, $0

```

코드 809. bootsect.S(8)

추가로 남은 코드들은 `read_it()`함수에서 사용하는 것과, `print`에 관련된 것, 즉, debugging 목적으로 문자열이나 decimal값을 `print`하는 routine들이다. `SETUPSEG`에는 이미 `setup.S`가 올라와 있기에 이곳으로 long jump를 해서 나머지 booting에 필요한 동작들을 차근차근 해 나간다. `boosect.S`의 마지막에 가지는 memory image는 [그림75]와 같다.

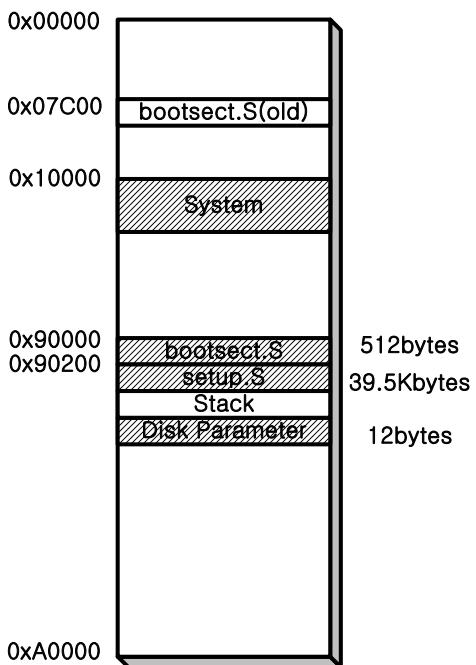


그림 99. bootsect.S의 수행 이후의 memory image

16.3. setup.S의 분석

setup.S가 하는 일은 BIOS로부터 system에서 사용할 data를 읽어오는 일과 그것을 적절하게 system memory에 넣는 것이다. setup.S와 system은 이미 bootsect.S에 의해서 이미 load되어 있는 상황이기에 이러한 일이 가능하다.

BIOS에 memory와 disk등의 parameter들에 대해서 물어보고, 결과를 0x90000~0x901FF사이의 주소에 넣어둔다. 이곳은 bootsect.S에 의해서 사용되었던 부분으로 buffer block들에 의해서 이 영역이 사용되기 전에 protected mode에 있을 system이 나중에 다시 읽어 들여주어야 한다.

```
/* Signature words to ensure LILO loaded us right */
#define SIG1      0xAA55
#define SIG2      0x5A5A

INITSEG = DEF_INITSEG          # 0x9000, we move boot here, out of the way
SYSSEG = DEF_SYSSEG           # 0x1000, system loaded at 0x10000 (65536).
SETUPSEG = DEF_SETUPSEG        # 0x9020, this is the current segment
                                # ... and the former contents of CS

DELTA_INITSEG = SETUPSEG - INITSEG # 0x0020
```

코드 810. setup.S(1)

SIG1과 SIG2는 나중에 lilo에서 setup.S가 읽혀질 경우에 lilo가 제대로 읽어 들였는지를 확인하기 위해서 사용된다. 나머지 INITSEG와 SYSSEG, SETUPSEG들의 값을 보관한다. 그리고, INITSEG와 SETUPSEG사이에 얼마만큼의 공간이 있는지를 미리 계산해 둔다.

```
start_of_setup:
# Bootlin depends on this being done early
    movw $0x01500, %ax
    movb $0x81, %dl
    int $0x13

#endifif SAFE_RESET_DISK_CONTROLLER
# Reset the disk controller.
    movw $0x0000, %ax
    movb $0x80, %dl
    int $0x13
#endif

# Set %ds = %cs, we know that SETUPSEG = %cs at this point
    movw %cs, %ax          # aka SETUPSEG
    movw %ax, %ds
# Check signature at end of setup
    cmpw $SIG1, setup_sig1
    jne bad_sig

    cmpw $SIG2, setup_sig2
    jne bad_sig

    jmp good_sig1

# Routine to print ascii string at ds:si
prtstr:
    lodsb
    andb %al, %al
    jz fin
```

```

call    prtchr
jmp    prtstr

fin:   ret

# Space printing
prtsp2: call    prtspc      # Print double space
prtspc: movb    $0x20, %al    # Print single space (note: fall-thru)

# Part of above routine, this one just prints ascii al
prtchr: pushw   %ax
        pushw   %cx
        xorb    %bh, %bh
        movw    $0x01, %cx
        movb    $0x0e, %ah
        int     $0x10
        popw    %cx
        popw    %ax
        ret

beep:   movb    $0x07, %al
        jmp     prtchr

no_sig_mess: .string      "No setup signature found ..."

good_sig1:
        jmp     good_sig

# We now have to find the rest of the setup code/data
bad_sig:
        movw    %cs, %ax          # SETUPSEG
        subw    $DELTA_INITSEG, %ax # INITSEG
        movw    %ax, %ds
        xorb    %bh, %bh
        movb    (497), %bl        # get setup sect from bootsect
        subw    $4, %bx            # LILO loads 4 sectors of setup
        shlw    $8, %bx            # convert to words (1sect=2^8 words)
        movw    %bx, %cx
        shrw    $3, %bx            # convert to segment
        addw    $SYSSEG, %bx
        movw    %bx, %cs:start_sys_seg

# Move rest of setup code/data to here
        movw    $2048, %di          # four sectors loaded by LILO
        subw    %si, %si
        movw    %cs, %ax          # aka SETUPSEG
        movw    %ax, %es
        movw    $SYSSEG, %ax
        movw    %ax, %ds
        rep
        movsw
        movw    %cs, %ax          # aka SETUPSEG
        movw    %ax, %ds
        cmpw    $SIG1, setup_sig1
        jne     no_sig

        cmpw    $SIG2, setup_sig2
        jne     no_sig

```

```

        jmp      good_sig

no_sig:
        lea      no_sig_mess, %si
        call    prtstr

no_sig_loop:
        jmp     no_sig_loop

good_sig:
        movw   %cs, %ax          # aka SETUPSEG
        subw   $DELTA_INITSEG, %ax # aka INITSEG
        movw   %ax, %ds

# Check if an old loader tries to load a big-kernel
        testb  $LOADED_HIGH, %cs:loadflags # Do we have a big kernel?
        jz     loader_ok          # No, no danger for old loaders.

        cmpb   $0, %cs:type_of_loader # Do we have a loader that
                                      # can deal with us?
        jnz    loader_ok          # Yes, continue.

        pushw  %cs
        popw   %ds          # No, we have an old loader,
        lea     loader_panic_mess, %si # die.
        call    prtstr

        jmp     no_sig_loop

loader_panic_mess: .string "Wrong loader, giving up..."
```

코드 811. setup.S(2)

다음으로는 disk controller를 reset하고 이미 읽어 들여온 setup.S의 마지막 부분에 있는 signature값과 비교를 한다. 나머지는 결과를 print하는 부분이다. 이렇게 읽혀 들여온 값이 맞다면, 이번에는 어떤 loader가 준비되었는지를 살핀다. 올바른 값의 loader가 없다면 그대로 무한 loop를 돌면서 멈춘다. 이후에는 연장(extended) 메모리에 대한 크기를 읽어오는 부분이다. 여기서 [~/include/asm/E820.h](#)를 참고하기 바란다.

```

loader_ok:
# Get memory size (extended mem, kB)

        xorl   %eax, %eax
        movl   %eax, (0x1e0)
#ifndef STANDARD_MEMORY_BIOS_CALL
        movb   %al, (E820NR)
#endif
# Try three different memory detection schemes. First, try
# e820h, which lets us assemble a memory map, then try e801h,
# which returns a 32-bit memory size, and finally 88h, which
# returns 0-64m

# method E820H:
# the memory map from hell. e820h returns memory classified into
# a whole bunch of different types, and allows memory holes and
# everything. We scan through this memory map and build a list
# of the first 32 memory areas, which we return at [E820MAP].
#
```

```

meme820:
    movl $0x534d4150, %edx          # ascii `SMAP'
    xorl %ebx, %ebx                # continuation counter
    movw $E820MAP, %di             # point into the whitelist
                                    # so we can have the bios
                                    # directly write into it.

jmpe820:
    movl $0x0000e820, %eax         # e820, upper word zeroed
    movl $20, %ecx                # size of the e820rec
    pushw %ds                     # data record.
    popw %es
    int $0x15                      # make the call
    jc bail820                    # fall to e801 if it fails

    cmpl $0x534d4150, %eax         # check the return is `SMAP'
    jne bail820                   # fall to e801 if it fails

#   cmpl $1, 16(%di)              # is this usable memory?
#   jne again820

# If this is usable memory, we save it by simply advancing %di by
# sizeof(e820rec).
#
good820:
    movb (E820NR), %al            # up to 32 entries
    cmpb $E820MAX, %al
    jnl bail820

    incb (E820NR)
    movw %di, %ax
    addw $20, %ax
    movw %ax, %di

again820:
    cmpl $0, %ebx                 # check to see if
    jne jmpe820                  # %ebx is set to EOF

bail820:

# method E801H:
# memory size is in 1k chunksizes, to avoid confusing loadlin.
# we store the 0xe801 memory size in a completely different place,
# because it will most likely be longer than 16 bits.
# (use 1e0 because that's what Larry Augustine uses in his
# alternative new memory detection scheme, and it's sensible
# to write everything into the same place.)

meme801:
    movw $0xe801, %ax
    int $0x15
    jc mem88

    andl $0xffff, %edx            # clear sign extend
    shll $6, %edx                # and go from 64k to 1k chunks
    movl %edx, (0x1e0)            # store extended memory size
    andl $0xffff, %ecx            # clear sign extend
    addl %ecx, (0x1e0)            # and add lower memory into
                                    # total size.

```

```
# Ye Olde Traditional Methode. Returns the memory size (up to 16mb or
# 64mb, depending on the bios) in ax.
mem88:

#endif
    movb    $0x88, %ah
    int     $0x15
    movw    %ax, (2)
```

코드 812. setup.S(3)

메모리의 크기를 읽는 방법은 세가지를 사용하게 되며 그 중 한가지라도 만족하면 된다. 세가지 방법에는 0xe820과 0xe801, 그리고 0x88을 이용한 BIOS interrupt 0x15가 있다.

```
# Set the keyboard repeat rate to the max
    movw    $0x0305, %ax
    xorw    %bx, %bx
    int     $0x16

# Check for video adapter and its parameters and allow the
# user to browse video modes.
    call    video                      # NOTE: we need %ds pointing
                                         # to bootsector
```

코드 813. setup.S(4)

Keyboard의 repeat rate를 조정하고, video adapter를 check 및 video mode를 설정한다. 이 부분에서 사용되는 code는 `~/arch/i386/boot/video.S`에 있다.

이후의 과정은 하드디스크의 파라미터(parameter)를 읽어오고, 마이크로 채널 아키텍처(Micro Channel Architecture: MCA) 버스(bus)를 초기화하며, PS/2에 대한 초기화와 고급 전원관리(Advanced Power Management : APM)의 초기화를 한다. 이와 같은 일이 끝나면 이전 리얼모드(real mode)에서 보호모드(protected mode)로의 전환이 있게 된다.³³²

```
...
# Now we want to move to protected mode ...
    cmpw    $0, %cs:realmode_swtch
    jz     rmodeswtch_normal

    lcall   %cs:realmode_swtch
    jmp    rmodeswtch_end

rmodeswtch_normal:
    pushw   %cs
    call    default_switch

rmodeswtch_end:
```

코드 814. setup.S(5)

먼저 `realmode_swtch`값을 0과 비교해서 같은 값이라면 그냥 지나가고, 그렇지 않다면 `cs:realmode_swtch`를 호출한 후에 `default_switch`를 호출하는 것을 생략하고 진행한다. 보호모드로의 전환하기 바로 전에 `default_switch` 함수가 호출되는데, `default_switch`에서 하는 일은 모든 인터럽트를 일어나지 못하게 하는 것이다.

³³² 물론 이것은 x86만의 특징이다. 다른 CPU에 대한 것은 그에 관련된 커널 코드를 다운 받아서 보기 바란다.

```
# we get the code32 start address and modify the below 'jmp'
# (loader may have changed it)
    movl    %cs:code32_start, %eax
    movl    %eax, %cs:code32
```

코드 815.setup.S(6)

이제 우리는 code32의 시작 주소를 알게 되었고, 따라서 어디에 시스템이 로드되어있는지도 알게 되었다. 이를 바탕으로 해서 나중에 제어를 넘겨주기 위한 것이 어디에 있는지를 설정하는 것이다.

```
testb $LOADED_HIGH, %cs:loadflags
jz    do_move0
      # .. then we have a normal low
      # loaded zImage
      # .. or else we have a high
      # loaded bzImage
      # ... and we skip moving

jmp   end_move

do_move0:
      movw $0x100, %ax          # start of destination segment
      movw %cs, %bp             # aka SETUPSEG
      subw $DELTA_INITSEG, %bp  # aka INITSEG
      movw %cs:start_sys_seg, %bx # start of source segment
      cld

do_move:
      movw %ax, %es            # destination segment
      incb %ah                # instead of add ax,#0x100
      movw %bx, %ds            # source segment
      addw $0x100, %bx
      subw %di, %di
      subw %si, %si
      movw $0x800, %cx
      rep
      movsw
      cmpw %bp, %bx           # assume start_sys_seg > 0x200,
                                # so we will perhaps read one
                                # page more than needed, but
                                # never overwrite INITSEG
                                # because destination is a
                                # minimum one page below source

jb    do_move

end_move:
```

코드 816.setup.S(7)

이전 시스템을 적절한 위치로 이동시키는 것이다. 여기서 한가지 주의할 점은 만약 생성된 커널이 bzImage를 바탕으로 한다면, 즉 big kernel image로 되어있다면, 옮기지 말아야 한다는 것이다.

먼저 loadflags의 값을 확인해서 시스템이 상위(high)에 적재(load)되었는지를 확인한다. 그리고나서 만약 상위로 적재되었다면, 옮기는 부분을 실행하지 말고, 그렇지 않다면 시스템을 적절한 위치(0x1000)으로 옮겨놓는다.

```
# then we load the segment descriptors
    movw %cs, %ax              # aka SETUPSEG
    movw %ax, %ds

# Check whether we need to be downward compatible with version <=201
```

```

cmpl    $0, cmd_line_ptr
jne     end_move_self          # loader uses version >=202 features
cmpb    $0x20, type_of_loader
je      end_move_self          # bootsect loader, we know of it

# Boot loader doesn't support boot protocol version 2.02.
# If we have our code not at 0x90000, we need to move it there now.
# We also then need to move the params behind it (commandline)
# Because we would overwrite the code on the current IP, we move
# it in two steps, jumping high after the first one.
        movw  %cs, %ax
        cmpw  $SETUPSEG, %ax
        je    end_move_self

        cli                           # make sure we really have
                                      # interrupts disabled !
                                      # because after this the stack
                                      # should not be used
        subw  $DELTAINITSEG, %ax      # aka INITSEG
        movw  %ss, %dx
        cmpw  %ax, %dx
        jb    move_self_1

        addw  $INITSEG, %dx
        subw  %ax, %dx              # this will go into %ss after
                                      # the move

move_self_1:
        movw  %ax, %ds
        movw  $INITSEG, %ax          # real INITSEG
        movw  %ax, %es
        movw  %cs:setup_move_size, %cx
        std                           # we have to move up, so we use
                                      # direction down because the
                                      # areas may overlap
        movw  %cx, %di
        decw  %di
        movw  %di, %si
        subw  $move_self_here+0x200, %cx
        rep
        movsb
        ljmp  $SETUPSEG, $move_self_here

move_self_here:
        movw  $move_self_here+0x200, %cx
        rep
        movsb
        movw  $SETUPSEG, %ax
        movw  %ax, %ds
        movw  %dx, %ss

end_move_self:                      # now we are at the right place
        lidt  idt_48                # load idt with 0,0
        xorl  %eax, %eax            # Compute gdt_base
        movw  %ds, %ax
        shll  $4, %eax              # (Convert %ds:gdt to a linear ptr)
        addl  $gdt, %eax
        movl  %eax, (gdt_48+2)
        lgdt  gdt_48                # load gdt with whatever is
                                      # appropriate

```

그리고나서는 CS와 DS 레지스터를 같게 만든다. 만약 boot loader가 boot protocol 2.02를 지원하지 않는다면 새로운 위치(0x90000)로 setup.S를 옮겨준다. 나머지는 코드들은 관련된 setup.S를 옮겨주는 부분이다. 옮기기가 다 끝나면 IDT(Interrupt Descriptor Table)와 GDT(Global Descriptor Table)를 설정한다.

```
# that was painless, now we enable a20
    call    empty_8042
    movb   $0xD1, %al
    outb   %al, $0x64
    call    empty_8042
    movb   $0xDF, %al          # A20 on
    outb   %al, $0x60
    call    empty_8042
#
# You must preserve the other bits here. Otherwise embarrassing things
# like laptops powering off on boot happen. Corrected version by Kira
# Brown from Linux 2.2
#
    inb    $0x92, %al          #
    orb    $02, %al           # "fast A20" version
    outb   %al, $0x92          # some chips have only this

# wait until a20 really *is* enabled; it can take a fair amount of
# time on certain systems; Toshiba Tecras are known to have this
# problem. The memory location used here (0x200) is the int 0x80
# vector, which should be safe to use.
    xorw   %ax, %ax          # segment 0x0000
    movw   %ax, %fs
    decw   %ax                # segment 0xffff (HMA)
    movw   %ax, %gs
a20_wait:
    incw   %ax                # unused memory location <0xffff0
    movw   %ax, %fs:(0x200)    # we use the "int 0x80" vector
    cmpw   %gs:(0x210), %ax   # and its corresponding HMA addr
    je     a20_wait           # loop until no longer aliased
```

이전 실제로 real mode에서 protected mode로 들어가는 부분이다. 들어가기 전에 먼저 A20(address line 20)³³³이 enable이 되었는지를 확인한다. 이것은 키보드를 이용해서 확인하게 되는데, 먼저 키보드의 buffer가 비어있는지(empty)를 확인하는 것이다(empty_8042). 이렇게 한 후에 다시 0x64번지에 0xD1을 쓴다. 다시 키보드의 버퍼를 비우고, 0xDF를 0x60으로 쓴다. 과정이 끝났으면, 다시 키보드 버퍼가 비었는지를 확인한다. 0xDF를 0x60에 쓰는 것으로 A20 address line을 켜게(on)된다³³⁴. Fast A20의 경우에는 0x92번지로부터 값을 읽어서, 이전의 0xDF와 or를 시켜 다시 0x92번지에 write한다.

다음과정은 실제로 A20 address line이 실제로 enable되었는지 확인하는 과정이다. fs에 0x0000값을 가져다놓고, gs에는 0xFFFF(0x0000 - 1)을 넣는다. 그렇게 만든후 0에서부터 ax를 하나씩 증가 시키면서, fs:0x200번지에 ax를 집어넣는다. 집어넣은 값과 gs:0x210번지의 값이 같은지를 확인한다. 만약 A20 address line이 enable되지 않았다면, 이 값은 동일할 것이다. 즉, address wraparound가 일어난 것이다. 즉, 리얼 모드에서는 0xFFFF이상이되는 주소에 대해서 0x00000에서부터 새로이 주소가 시작되도록 만들기 때문이다. 만약 다른 값을 가지게 된다면 이는 A20 address line이 enable되었음을 말한다.

³³³ IBM PC-AT Document에서 A20 address line을 조종하는 방법으로 8042 keyboard controller의 spare port를 이용하는 방법이 제시되어 있다.

³³⁴ A20 Address line은 A20MASK#와 같이 AND gate로 연결되어서 1MBytes 이상을 addressing하는데 사용된다. A20MASK#는 Keyboard Controller로부터 오거나, 혹은 keyboard controller에 주어진 명령을 감시하는 logic으로부터 생성될 수 있다.

empty_8042()함수는 키보드의 상황을 보는 것이다. empty_8042()함수의 코드를 보도록 하자.

```
...
# This routine checks that the keyboard command queue is empty
# (after emptying the output buffers)
#
# Some machines have delusions that the keyboard buffer is always full
# with no keyboard attached...
empty_8042:
    pushl    %ecx
    movl    $0x00FFFFFF, %ecx

empty_8042_loop:
    decl    %ecx
    jz    empty_8042_end_loop
    call    delay
    inb    $0x64, %al          # 8042 status port
    testb   $1, %al           # output buffer?
    jz    no_output
    call    delay
    inb    $0x60, %al           # read it
    jmp    empty_8042_loop
no_output:
    testb   $2, %al           # is input buffer full?
    jnz    empty_8042_loop      # yes - loop
empty_8042_end_loop:
    popl    %ecx
    ret
...
```

empty_8042()함수는 ecx레지스터에 0x00FFFFFF값을 넣어준다. 그런 다음에 loop를 돌면서 ecx값이 0가지는지를 확인하고, 만약 0을 가진다면 loop를 마친다. ecx가 0이 아니라면, delay를 둔 다음 al레지스터에 keyboard의 status를 읽어온다. Output buffer에 어떤 값이 있다면, 다시 delay를 둔 다음 그것을 읽도록 한다. Output buffer에 값이 들어있지 않다면, input buffer가 full인지를 확인하게 되며, 만약 full이라면 다시 loop를 돈다. Output buffer에 아무런 값이 없고, input buffer가 full이 아니라면, 그리고, counter값인 ecx가 0이 되면, 원래의 ecx값을 회복한 후에 함수는 복귀한다.

계속해서 이전의 이야기를 진행하면, 이제는 protected mode로의 진입을 시도하게 되는 부분이다. 코드를 보도록 하자.

```
# make sure any possible coprocessor is properly reset..
    xorw    %ax, %ax
    outb    %al, $0xf0
    call    delay

    outb    %al, $0xf1
    call    delay
# well, that went ok, I hope. Now we mask all interrupts - the rest
# is done in init_IRQ().
    movb    $0xFF, %al          # mask all interrupts for now
    outb    %al, $0xA1
    call    delay

    movb    $0xFB, %al          # mask all irq's but irq2 which
    outb    %al, $0x21           # is cascaded
# Well, that certainly wasn't fun :-(. Hopefully it works, and we don't
# need no steenkng BIOS anyway (except for the initial loading :-).
```

```

# The BIOS-routine wants lots of unnecessary data, and it's less
# "interesting" anyway. This is how REAL programmers do it.
#
# Well, now's the time to actually move into protected mode. To make
# things as simple as possible, we do no register set-up or anything,
# we let the gnu-compiled 32-bit programs do that. We just jump to
# absolute address 0x1000 (or the loader supplied one),
# in 32-bit protected mode.
#
# Note that the short jump isn't strictly needed, although there are
# reasons why it might be a good idea. It won't hurt in any case.
    movw    $1, %ax          # protected mode (PE) bit
    lmsw    %ax              # This is it!
    jmp     flush_instr

flush_instr:
    xorw    %bx, %bx        # Flag to indicate a boot
    xorl    %esi, %esi       # Pointer to real-mode code
    movw    %cs, %si
    subw    $DELTA_INITSEG, %si
    shll    $4, %esi         # Convert to 32-bit pointer

# NOTE: For high loaded big kernels we need a
#       jmpi    0x100000,__KERNEL_CS
#
#       but we yet haven't reloaded the CS register, so the default size
#       of the target offset still is 16 bit.
#
#       However, using an operant prefix (0x66), the CPU will properly
#       take our 48 bit far pointer. (INTEL 80386 Programmer's Reference
#       Manual, Mixing 16-bit and 32-bit code, page 16-6)

    .byte 0x66, 0xea        # prefix + jmpi-opcode
code32: .long   0x1000          # will be set to 0x100000
                                # for big kernels
    .word   __KERNEL_CS

```

보조 프로세스가 있다면 보조 프로세스에 대한 reset을 실행한다. 그리고나서 인터럽트가 일어나지 않도록 한 후, protected mode를 나타내는 cr0 레지스터의 PE bit를 설정한다. bx가 booting을 나타내도록 설정하기 위해서 0으로 두고, si를 cs에서 \$DELTA_INITSEG(=0x0020)만큼을 뺀 값을 넣어서, 이를 32bit의 pointer로 확장시킨다.

마지막은 48bit의 far pointer를 이용해서 커널의 CS로 jump를 실행하는 것이다. 이는 직접적인 명령을 이용하는 것이 아니라, 생성될 binary code를 직접적으로 써넣어서, 인위적으로 jump를 야기하는 방법이다. 이렇게 함으로써 이제 드디어 커널로 진입하게 된다.

```

# Here's a bunch of information about your current kernel..
kernel_version: .ascii  UTS_RELEASE
                 .ascii  "("
                 .ascii  LINUX_COMPILE_BY
                 .ascii  "@"
                 .ascii  LINUX_COMPILE_HOST
                 .ascii  ")"
                 .ascii  UTS_VERSION
                 .byte   0

# This routine only gets called, if we get loaded by the simple
# bootsect loader _and_ have a bzImage to load.
# Because there is no place left in the 512 bytes of the boot sector,
# we must emigrate to code space here.

```

```

bootsect_helper:
    cmpw $0, %cs:bootsect_es
    jnz bootsect_second

    movb $0x20, %cs:type_of_loader
    movw %es, %ax
    shrw $4, %ax
    movb %ah, %cs:bootsect_src_base+2
    movw %es, %ax
    movw %ax, %cs:bootsect_es
    subw $SYSSEG, %ax
    lret                                # nothing else to do for now

bootsect_second:
    pushw %cx
    pushw %si
    pushw %bx
    testw %bx, %bx                      # 64K full?
    jne bootsect_ex

    movw $0x8000, %cx                   # full 64K, INT15 moves words
    pushw %cs
    popw %es
    movw $bootsect_gdt, %si
    movw $0x8700, %ax
    int $0x15
    jc bootsect_panic                  # this, if INT15 fails

    movw %cs:bootsect_es, %es          # we reset %es to always point
    incb %cs:bootsect_dst_base+2      # to 0x10000

bootsect_ex:
    movb %cs:bootsect_dst_base+2, %ah
    shlb $4, %ah                      # we now have the number of
                                         # moved frames in %ax
    xorb %al, %al
    popw %bx
    popw %si
    popw %cx
    lret

bootsect_gdt:
    .word 0, 0, 0, 0
    .word 0, 0, 0, 0

bootsect_src:
    .word 0xffff

bootsect_src_base:
    .byte 0x00, 0x00, 0x01            # base = 0x010000
    .byte 0x93                         # typbyte
    .word 0                            # limit16,base24 =0

bootsect_dst:
    .word 0xffff

bootsect_dst_base:
    .byte 0x00, 0x00, 0x10            # base = 0x100000
    .byte 0x93                         # typbyte

```

```

.word    0                                # limit16,base24 =0
.word    0, 0, 0, 0                         # BIOS CS
.word    0, 0, 0, 0                         # BIOS DS

bootsect_es:
.word    0

bootsect_panic:
pushw   %cs
popw   %ds
cld
leaw    bootsect_panic_mess, %si
call    prtstr

bootsect_panic_loop:
jmp     bootsect_panic_loop

bootsect_panic_mess:
.string  "INT15 refuses to access high mem, giving up."
...
# Read the cmos clock. Return the seconds in al
gettime:
pushw   %cx
movb   $0x02, %ah
int    $0x1a
movb   %dh, %al                # %dh contains the seconds
andb   $0x0f, %al
movb   %dh, %ah
movb   $0x04, %cl
shrb   %cl, %ah
aad
popw   %cx
ret

# Delay is needed after doing I/O
delay:
jmp     .+2                                # jmp $+2
ret

# Descriptor tables
gdt:
.word    0, 0, 0, 0                         # dummy
.word    0, 0, 0, 0                         # unused

.word    0xFFFF                            # 4Gb - (0x100000*0x1000 = 4Gb)
.word    0                                # base address = 0
.word    0x9A00                           # code read/exec
.word    0x00CF                            # granularity = 4096, 386
                                         # (+5th nibble of limit)

.word    0xFFFF                            # 4Gb - (0x100000*0x1000 = 4Gb)
.word    0                                # base address = 0
.word    0x9200                           # data read/write
.word    0x00CF                            # granularity = 4096, 386
                                         # (+5th nibble of limit)

idt_48:
.word    0                                # idt limit = 0
.word    0, 0                             # idt base = 0L

```

```

gdt_48:
    .word    0x8000          # gdt limit=2048,
                           # 256 GDT entries

    .word    0, 0            # gdt base (filled in later)

# Include video setup & detection code

#include "video.S"

# Setup signature -- must be last
setup_sig1:    .word    SIG1
setup_sig2:    .word    SIG2

# After this point, there is some free space which is used by the video mode
# handling code to store the temporary mode table (not used by the kernel).

modelist:

.text
endtext:
.data
enddata:
.bss
endbss:

```

나머지는 상수값에 대한 정의와 각종 변수들, bootsect.S에서 helper로 사용하는 함수, GDT(Global Descriptor Table)과 IDT(Interrupt Descriptor Table)들에 대한 정의다. 물론 여기서 GDT와 IDT에 대해서는 적절하게 초기화가 이루어져야 한다.

16.4. head.S의 분석

setup.S가 수행을 넘겨주면 제일 먼저 하는 일은 압축된 커널을 푸는 일일 것이다. 이와 같은 일을 해주는 것이 head.S가 하는 일이다. ~/arch/i386/compressed/head.S는 32bit의 시작(startup) 코드를 가지며 하는 역할은 압축된 커널을 푸는 것과 풀어진 kernel로 실행을 옮기는 역할을 한다. 도와주는 함수로는 misc.c에 있는 압축된 커널을 풀어주는 것이다.

```

.text

#include <linux/linkage.h>
#include <asm/segment.h>

.globl startup_32

startup_32:
    cld
    cli
    movl $__KERNEL_DS,%eax
    movl %eax,%ds
    movl %eax,%es
    movl %eax,%fs
    movl %eax,%gs

    lss SYMBOL_NAME(stack_start),%esp
    xorl %eax,%eax
1:   incl %eax      # check that A20 really IS enabled
    movl %eax,0x000000  # loop forever if it isn't

```

```
cmpl %eax,0x100000
je 1b
```

코드 817. head.S(1)

setup에서 jump를 해서 startup_32가 실행된다. 먼저 direction flag를 지우고, interrupt가 발생하지 않도록 만든다. 그리고 나서 ds와 es, fs, gs가 __KERNEL_DS를 가리키도록 만들어준다. 이때 __KERNEL_DS가 가지는 값은 ~/include/asm/segment.h에 아래와 같이 정의 되어있다.

```
#define __KERNEL_CS      0x10
#define __KERNEL_DS       0x18
#define __USER_CS        0x23
#define __USER_DS        0x2B
```

이와 같이 했으면, 이제는 stack이 어디에 있는지를 가르쳐주도록 하고(esp), A20³³⁵이 enable되었는지를 반복적으로 값을 메모리에 쓰면서 확인한다.

```
...
pushl $0
popfl

xorl %eax,%eax
movl $SYMBOL_NAME(_edata),%edi
movl $SYMBOL_NAME(_end),%ecx
subl %edi,%ecx
cld
rep
stosb
...
```

코드 818. head.S(2)

Flag을 0으로 만들어준 후, 다시 데이터를 가지는 BSS(Block Started by Symbol)영역을 찾아서 0으로 채워준다.

```
...
subl $16,%esp      # place for structure on the stack
movl %esp,%eax
pushl %esi         # real mode pointer as second arg
pushl %eax         # address of structure as first arg
call SYMBOL_NAME(decompress_kernel)
orl   %eax,%eax
jnz   3f
popl %esi# discard address
popl %esi# real mode pointer
xorl %ebx,%ebx
ljmp $(__KERNEL_CS),$0x100000
3:
    movl $move_routine_start,%esi
    movl $0x1000,%edi
    movl $move_routine_end,%ecx
    subl %esi,%ecx
    addl $3,%ecx
    shr $2,%ecx
    cld
```

³³⁵ Fast Interrupt Gate라는 특수한 하드웨어를 말한다.

```

rep
movsl

popl %esi# discard the address
popl %ebx      # real mode pointer
popl %esi# low_buffer_start
popl %ecx      # lcount
popl %edx      # high_buffer_start
popl %eax      # hcount
movl $0x100000,%edi
cli           # make sure we don't get interrupted
...
ljmp ${__KERNEL_CS},$0x1000 # and jump to the move routine

```

코드 819. head.S(3)

압축된 kernel을 메모리에 푸는 일을 하고, 새로이 풀어진 kernel로 jump를 해서 제어를 넘겨준다. 만약 커널이 메모리의 상위에 load가 되었다면, move_routine_start와 move_routine_end 사이에 들어있는 code를 0x1000에 옮겨놓고, 이곳에서 다시 kernel을 0x100000으로 옮기도록 만든다. 그리고 나서 옮겨진 kernel로 jump를 해서 제어를 넘겨준다.

```

...
move_routine_start:
    movl %ecx,%ebp
    shr $2,%ecx
    rep
    movsl
    movl %ebp,%ecx
    andl $3,%ecx
    rep
    movsb
    movl %edx,%esi
    movl %eax,%ecx  # NOTE: rep movsb won't move if %ecx == 0
    addl $3,%ecx
    shr $2,%ecx
    rep
    movsl
    movl %ebx,%esi  # Restore setup pointer
    xorl %ebx,%ebx
    ljmp ${__KERNEL_CS},$0x100000
move_routine_end:

```

코드 820. head.S(4)

위의 코드는 move_routine_start와 move_routine_end 사이의 코드를 보여준다. 이곳에서는 data를 옮겨주는 일과 0x100000으로 jump하는 일을 담당한다.

이로서 대략적인 kernel의 booting 과정을 보았다. 이제 kernel에서 제어를 넘겨받는 일이 남게 되었다. head.S에서 kernel의 code segment로 jump를 하게 되면 가장 먼저 실행되는 것은 ~/arch/i386/kernel/head.S가 된다. 이 파일을 앞에서 본 head.S와 분리시켜서 보기 위해서 kernel의 head.S라고 부르도록 하자.

16.5. Kernel의 head.S의 분석

~/arch/i386/kernel/head.S를 보도록 하자. 이것도 32bit의 시작 코드를 가지고 있으며, 사용하는 CPU가 어떤 타입(type)인지를 찾는 역할을 하며, 커널의 기동시에 필요한 페이지 테이블 및 인터럽트 테이블에 대한 재 설정이 이루어진다. 코드를 보도록 하자.

```
.text
#include <linux/config.h>
#include <linux/threads.h>
#include <linux/linkage.h>
#include <asm/segment.h>
#include <asm/page.h>
#include <asm/pgtable.h>
#include <asm/desc.h>

#define OLD_CL_MAGIC_ADDR 0x90020
#define OLD_CL_MAGIC 0xA33F
#define OLD_CL_BASE_ADDR 0x90000
#define OLD_CL_OFFSET 0x90022
#define NEW_CL_POINTER 0x228 /* Relative to real mode data */

#define CPU_PARAMS SYMBOL_NAME(boot_cpu_data)
#define X86 CPU_PARAMS+0
#define X86_VENDOR CPU_PARAMS+1
#define X86_MODEL CPU_PARAMS+2
#define X86_MASK CPU_PARAMS+3
#define X86_HARD_MATH CPU_PARAMS+6
#define X86_CPUID CPU_PARAMS+8
#define X86_CAPABILITY CPU_PARAMS+12
#define X86_VENDOR_ID CPU_PARAMS+16
```

코드 821. 커널의 head.S

이부분은 사용하게 될 상수 값들에 대한 정의다 `XXX_CL_XXXX_XXXX`이라고 선언된 것은 명령 라인 (command line) 파라미터 값들에 대한 포인터이다. 나머지는 CPU에 대해서 사용하게될 `boot_cpu_data`의 멤버들에 대한 reference이다.

```
ENTRY(stext)
ENTRY(_stext)
startup_32:
/*
 * Set segments to known values
 */
        cld
        movl $(__KERNEL_DS),%eax
        movl %eax,%ds
        movl %eax,%es
        movl %eax,%fs
        movl %eax,%gs
#endif CONFIG_SMP
        orw %bx,%bx
        jz 1f
```

코드 822. 커널의 head.S – continued

이젠 커널을 위한 세그먼트 레지스터들의 내용을 설정한다. 이에 해당하는 레지스터는 `ds`, `es`, `fs`, `gs`이며 설정되는 값은 `__KERNEL_DS`이다.

```
#define cr4_bits mmu_cr4_features-_PAGE_OFFSET
        cmpl $0,cr4_bits
        je 3f
        movl %cr4,%eax          # Turn on paging options (PSE,PAE,...)
```

```

orl cr4_bits,%eax
movl %eax,%cr4
jmp 3f

```

1:
#endif

코드 823. 커널의 head.S – continued

이전 cr4레지스터의 페이징 시스템을 위한 옵션을 켜는 일이다. 486에서는 cr4가 없다는 것을 유의하라. 따라서 cr4_bits가 설정되지 않았다면 3f³³⁶ 부분으로 제어를 옮긴다. 그렇지 않다면, cr4_bits를 cr4레지스터에 설정한 후 3f로 진행한다.

더 진행하기 전에 Intel CPU의 control 레지스터들에 대해서 간단히 보자. [그림76]와 같이 정해진다.

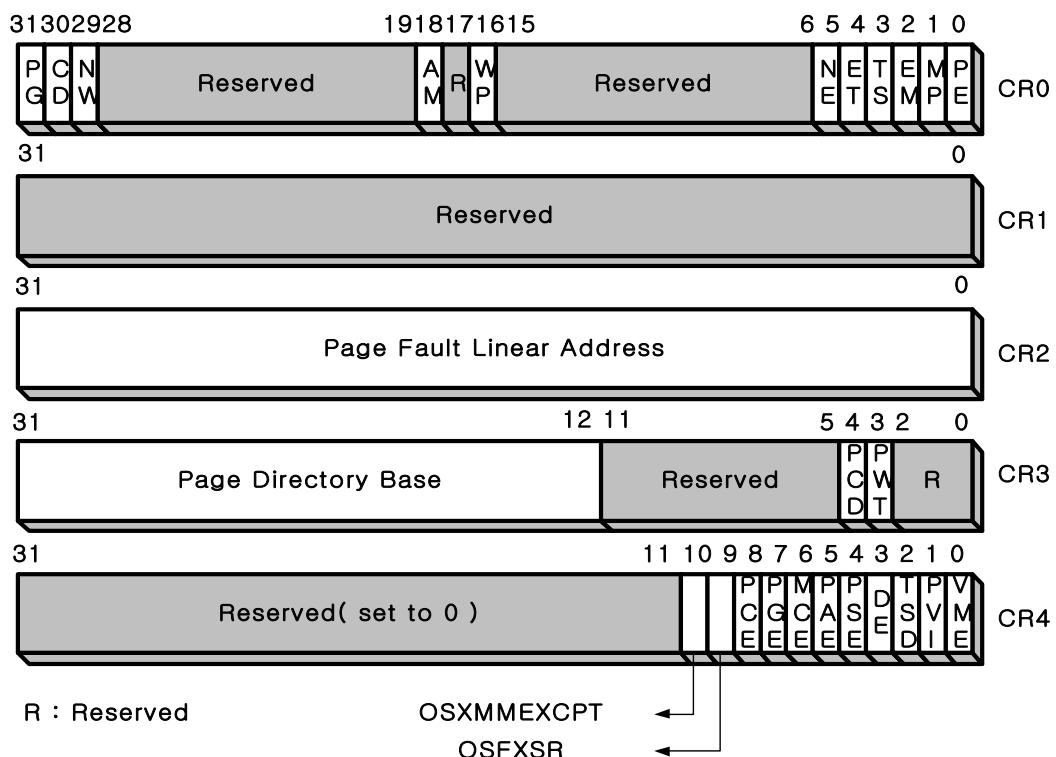


그림 100. Intel의 Control Register

cr0 레지스터는 프로세스의 상태나 운영(operating) 모드를 설정하는 시스템 제어 플랙(flag)을 가진다. cr1 레지스터는 예약된 레지스터이며, cr2레지스터는 페이지 풀트(fault)를 일으킨 선형주소를 가진다. cr3 레지스터는 페이지 디렉토리의 베이스 주소와 두개의 플랙(PCD, PWT)를 가진다. 두개의 플랙은 프로세서의 내부 캐쉬에 페이지 디렉토리를 보관(caching)하는 것을 제어하는 플랙그이다. cr4 레지스터는 여러개의 플랙으로 이루어져 있으며, 각각의 플랙은 여러 CPU 구조상의 확장을 가능하게(enable)하거나, 특정 프로세스에 대해서 운영체제나 기타 실행상의 지원을 나타낸다.

이중에서 논의되고 있는 cr4레지스터의 필드들의 정의는 아래와 같다.

Field	Description
VME	Virtual-8086 Mode Extension
PVI	Protected-Mode Virtual Interrupt

³³⁶ 레이블이 3을 가지는 앞쪽(forward)으로 제어를 옮김을 의미한다.

TSD	Time Stamp Disable
DE	Debugging Extension
PSE	Page Size Extension(1 for 4 Mbytes, 0 for 4 Kbytes)
PAE	Physical Address Extension(1 for 36 bit physical address, 0 for 32 bit physical address)
MCE	Machine-Check Enable
PGE	Page Global Enable(1 for enable global page feature, 0 for disable global page feature)
PCE	Performance-Monitoring Counter Enable
OSXMMEXCPT	Operating System Support for FXSAVE and FXRSTOR instruction
OSFXSR	Operating System Support for Unmasked SIMD Floating-Point Exceptions

표 93. cr4 레지스터의 필드

따라서, 위에서 페이징 시스템을 위해서 PSE, PAE등을 설정했다고 보면된다. 리눅스에서는 4 Kbytes의 페이지 크기만을 지원하며, 32 bit 주소 설정을 설정한다.

```

movl $pg0-__PAGE_OFFSET,%edi /* initialize page tables */
movl $007,%eax           /* "007" doesn't mean with right to kill, but
                           PRESENT+RW+USER */
2:
stosl
add $0x1000,%eax
cmp $empty_zero_page-__PAGE_OFFSET,%edi
jne 2b

```

코드 824. 커널의 head.S – continued

페이지 테이블에 대한 초기화를 수행한다. pg0(0x2000)를 __PAGE_OFFSET(0xC0000000)로 뺀 값³³⁷으로 edi레지스터를 두고, 7(0x07)값을 eax레지스터에 두어서, pg0의 PRESENT+RW+USER로 설정한다. 다시 eax값을 0x1007로 두고, empty_zero_page에서 __PAGE_OFFSET과 뺀 주소를 edi와 비교해보자, 만약 작다면 2³³⁸로 점프한다.

```

3:
movl $swapper_pg_dir-__PAGE_OFFSET,%eax
movl %eax,%cr3           /* set the page table pointer.. */
movl %cr0,%eax
orl $0x80000000,%eax
movl %eax,%cr0           /* ..and set paging (PG) bit */
jmp 1f                   /* flush the prefetch-queue */

1:
movl $1f,%eax
jmp *%eax                /* make sure eip is relocated */

1:
/* Set up the stack pointer */
lss stack_start,%esp

#endif CONFIG_SMP
orw  %bx,%bx
jz  1f                   /* Initial CPU cleans BSS */
pushl $0
popfl
jmp checkCPUtype
1:
#endif CONFIG_SMP

```

³³⁷ 실제 pg0는 커널 영역에 있게 되므로, 0xC0002000값을 가지게 될 것이다. 빼면 pg0의 옵셋 값이 된다.

³³⁸ 레이블이 2를 가지는 뒤쪽(backward)으로 제어를 옮김을 의미한다.

코드 825. 커널의 head.S – continued

이전 페이징 시스템을 사용하게 만드는(enable) 일이다. swapper_pg_dir에서 __PAGE_OFFSET을 뺀 값을 eax에 두고, 이 값을 cr3레지스터에 두자(swapper_page_dir을 cr3가 가르키도록 만드는 것이다.). 만약 cr0를 읽어와서 이 값을 0x80000000과 OR시켜, 다시 cr0에 넣어준후 1로 제어를 옮긴다. Intel 프로세서는 cr0레지스터의 PG bit을 설정하는 것으로 페이징을 사용할 수 있도록 만들어준다. 이전 스택의 포인터를 설정하는 일이다. stack_start의 주소를 esp로 옮겨넣어준다.

이하는 SMP 시스템의 경우에 플랙그 레지스터를 0으로 설정한 후 CPU의 타입(type)을 검사하는 부분으로 제어를 옮기는 것을 보여준다.

```
xorl %eax,%eax
movl $ SYMBOL_NAME(__bss_start),%edi
movl $ SYMBOL_NAME(_end),%ecx
subl %edi,%ecx
rep
stosb

call setup_idt

pushl $0
popfl
```

코드 826. 커널의 head.S – continued

이전 커널의 BSS(Block Started by Symbol)³³⁹ 영역을 0(eax)으로 초기화한다. 영역의 크기는 __bss_start에서 _end까지의 영역이다. 이상의 과정을 마치면 이전 인터럽트 디스크립터 테이블(IDT: Interrupt Descriptor Table)을 설정하는 일이다(setup_idt). 그리고나면 플랙그 레지스터(eflags)를 0으로 초기화 한다.

```
movl $ SYMBOL_NAME(empty_zero_page),%edi
movl $512,%ecx
cld
rep
movsl
xorl %eax,%eax
movl $512,%ecx
rep
stosl
movl SYMBOL_NAME(empty_zero_page)+NEW_CL_POINTER,%esi
andl %esi,%esi
jnz 2f          # New command line protocol
cmpw $(OLD_CL_MAGIC),OLD_CL_MAGIC_ADDR
jne 1f
movzwl OLD_CL_OFFSET,%esi
addl $(OLD_CL_BASE_ADDR),%esi
2:
movl $ SYMBOL_NAME(empty_zero_page)+2048,%edi
movl $512,%ecx
rep
movsl
1:
```

코드 827. 커널의 head.S – continued

³³⁹ 이 영역은 프로그램에서 지역변수를 위해서 사용되거나, 값을 할당하지 않은 변수들을 위한 공간이다. 0으로 설정한다.

이전 bootup parameter들과 command line 옵션을 저장하는 일이다. esi레지스터가 아직도 real mode에서의 데이터를 가르키는 포인터로 사용된다. empty_zero_page영역의 첫 2Kbytes영역에 bootup parameter들을 보관하고, 그 다음의 2Kbytes영역에 command line 옵션을 저장한다.

```
#ifdef CONFIG_SMP
checkCPUtype:
#endif
    movl $-1,X86_CPUID           # -1 for no CPUID initially

    movl $3,X86                  # at least 386
    pushfl                      # push EFLAGS
    popl %eax                   # get EFLAGS
    movl %eax,%ecx              # save original EFLAGS
    xorl $0x40000,%eax          # flip AC bit in EFLAGS
    pushl %eax                   # copy to EFLAGS
    popfl                        # set EFLAGS
    pushfl                      # get new EFLAGS
    popl %eax                   # put it in eax
    xorl %ecx,%eax              # change in flags
    andl $0x40000,%eax          # check if AC bit changed
    je is386

    movl $4,X86                  # at least 486
    movl %ecx,%eax              # check ID flag
    xorl $0x200000,%eax          # if we are on a straight 486DX, SX, or
    pushl %eax                   # 487SX we can't change it
    popl %eax
    xorl %ecx,%eax
    pushl %ecx                   # restore original EFLAGS
    popfl
    andl $0x200000,%eax
    je is486

/* get vendor info */
    xorl %eax,%eax              # call CPUID with 0 -> return vendor ID
    cpuid
    movl %eax,X86_CPUID          # save CPUID level
    movl %ebx,X86_VENDOR_ID      # lo 4 chars
    movl %edx,X86_VENDOR_ID+4    # next 4 chars
    movl %ecx,X86_VENDOR_ID+8    # last 4 chars

    orl %eax,%eax                # do we have processor info as well?
    je is486

    movl $1,%eax                 # Use the CPUID instruction to get CPU type
    cpuid
    movb %al,%cl                 # save reg for future use
    andb $0x0f,%ah               # mask processor family
    movb %ah,X86
    andb $0xf0,%al               # mask model
    shrb $4,%al
    movb %al,X86_MODEL
    andb $0x0f,%cl               # mask mask revision
    movb %cl,X86_MASK
    movl %edx,X86_CAPABILITY
```

코드 828. 커널의 head.S – continued

CPU의 타입을 알아내는 부분이다. 각각의 알아낸 CPU의 타입에 따라서, 최기화가 다르게 진행한다. 먼저 X86_CPUID에 -1값을 지정해서 CPU의 타입을 모른다고 설정한다. 리눅스는 기본적으로 386이상의 시스템을 요구사항으로 가지고 있으므로 X86에는 3(386을 의미)을 넣는다. eflags 레지스터의 AC bit를 flip(0>1, 1>0을 설정)한 후, eflags레지스터를 제설정하고, 이를 다시 읽어온다. 만약 변화가 없었다면 is386(386 CPU)로 분기한다. 변화가 없었다면 계속적으로 진행해서 이전 X86을 4(486)으로 설정한다. 원래의 eflags레지스터의 값이 ecx에 저장되어 있으므로 이것을 다시 읽어와서(eax) ID flag를 XOR시킨다. 이 값을 다시 eflags레지스터로 읽어온후(popfl), 다시 스택에 push한다(pushfl). 스택에 push된 값을 eax로 읽어와서 원래의 eflags값과 비교해서 ID flag가 설정되었다면 is486(486 CPU)로 분기한다. 만약 위의 두가지 연산이 모두 실패한다면, 이전 cpuid 명령을 사용해서 프로세서의 family와 mode, mask버전 및 capability를 읽어온다. 각각 boot_cpu_data의 X86, X86_MODEL, X86_MASK, X86_CAPABILITY필드에 들어간다.

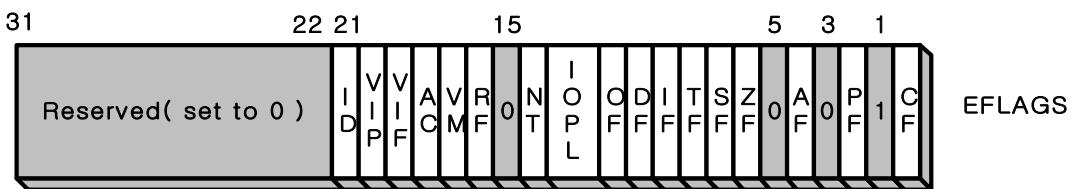


그림 101. Intel CPU의 EFLAGS 레지스터

Eflags레지스터는 [그림77]과 같으며, eflags 레지스터의 각 필드는 아래와 같은 의미를 지닌다.

Field	Description
ID	Identification(support cpuid instruction)
VIP	Virtual Interrupt Pending
VIF	Virtual Interrupt Flag
AC	Alignment Check
VM	Virtual-86 Mode
RF	Resume Flag
NT	Nested Task
IOPL	I/O Privilege Level
IF	Interrupt Enable Flag
TF	Trap(Single step debugging) Flag

표 94. eflags 레지스터의 필드

이곳에서는 ID필드의 내용을 설정해서 cpuid 명령어의 사용할 수 있는지를 물어보는 것으로 사용했었다. 다른 필드들은 각각의 명령어의 처리 결과에 따라서 설정된다.

is486:

```
    movl %cr0,%eax          # 486 or better
    andl $0x80000011,%eax   # Save PG,PE,ET
    orl $0x50022,%eax # set AM, WP, NE and MP
    jmp 2f
```

is386:

```
    pushl %ecx             # restore original EFLAGS
    popfl
    movl %cr0,%eax          # 386
    andl $0x80000011,%eax   # Save PG,PE,ET
    orl $2,%eax             # set MP
```

```
2:    movl %eax,%cr0
      call check_x87
```

```
#ifdef CONFIG_SMP
  incb ready
#endif
```

코드 829. 커널의 head.S – continued

CPU가 486이냐, 아니면 386이냐에 따라서 cr0 레지스터의 내용을 변경시켜주는 부분이다. 486은 PG, PE, ET, AM, WP, NE, MP를 cr0에 설정하고, 386의 경우에는 PG, PE, ET, MP만을 cr0에 설정한다. 이 과정을 다 마치면, 이전 수치 보조 프로세서(X87)가 있는지를 확인하는 일이다. SMP의 경우에는 ready변수를 증가시켜준다. ready변수는 현재 준비된 프로세서의 수를 말해준다.

여기서 cr0레지스터에 대해서 살펴보도록 하자. 아래의 표와 같은 필드로 구성된다.

Field	Description
PG	Paging Enable
PE	Protection Enable
CD	Cache Disable
NW	Write-Transparent Control
AM	Alignment Check Mask
WP	Supervisor Write Protect
NE	Numeric Exception Control
ET	Processor Extention(x87 check)
TS	Task Switch Occured
EM	Emulate Processor Extension(x87 emulation)
MP	Monitor Coprocessor
OE	Enable Segmented Protection

표 95. cr0 레지스터의 필드 정의

이상에서 386이나 486각각에 해당하는 부분만을 설정하고 나머지는 그대로 둔다. 공통적으로 PG, PE, ET는 그래도 사용하고, MP는 공통된 설정이다.

```
lgdt gdt_descr
lidt idt_descr
ljmp $(__KERNEL_CS),$1f
1:   movl $(__KERNEL_DS),%eax      # reload all the segment registers
      movl %eax,%ds            # after changing gdt.
      movl %eax,%es
      movl %eax,%fs
      movl %eax,%gs
#endif CONFIG_SMP
      movl $(__KERNEL_DS), %eax
      movl %eax,%ss            # Reload the stack pointer (segment only)
#else
      lss stack_start,%esp# Load processor stack
#endif
      xorl %eax,%eax
      lldt %ax
      cld                      # gcc2 wants the direction flag cleared at all times
#endif CONFIG_SMP
      movb ready, %cl
      cmpb $1,%cl
      je 1f                  # the first CPU calls start_kernel
                           # all other CPUs call initialize_secondary
      call SYMBOL_NAME(initialize_secondary)
```

```

        jmp L6
1:
#endif
        call SYMBOL_NAME(start_kernel)
L6:
        jmp L6          # main should never return here, but
                      # just in case, we know what happens.
#ifndef CONFIG_SMP
ready: .byte 0
#endif

```

코드 830. 커널의 head.S – continued

GDT(Global Descriptor Table)과 IDT(Interrupt Descriptor Table)을 gdtr과 idtr레지스터에 적재(load)한다. 이렇게 하고나면 이전 페이징이 실행되는 상황이므로 __KERNEL_CS의 레이블 1로 다시 제어를 옮긴다. 프로그램의 다음번 줄(line)이 실행될 것이다. 이전 GDT를 바꾸었으니 ds, es, fs, gs 레지스터의 내용을 재설정 해주어야 할 것이다.

만약 SMP 시스템이라면 ss 레지스터의 내용도 재 설정하고, 그렇지 않다면 esp레지스터에 프로세스의 스택(stack_start)를 다시 적재한다. 이전 실제 커널로 진행하는 부분이다. 진행전에 LDT(Local Descriptor Table)을 ldtr레지스터에 적재한다. SMP시스템의 경우라도 현재 CPU의 번호가 1인 경우에는 1로 진행하고, 그렇지 않은 경우에는 initialize_secondary()함수를 호출하도록 한다. 1로 진행한 경우라면 start_kernel()함수를 실행한다. 여기서부터는 이전 ~/init/main.c의 start_kernel()이 실행된다. initialize_secondary()함수는 ~/arch/i386/kernel/smpboot.c에 정의되어 있으며, 현재 프로세스(커널)의 task_struct로부터 esp를 적재하고, eip로 제어를 옮기는 일을 한다.

```

check_x87:
        movb $0,X86_HARD_MATH
        clts
        fninit
        fstsw %ax
        cmpb $0,%al
        je 1f
        movl %cr0,%eax      /* no coprocessor: have to set bits */
        xorl $4,%eax        /* set EM */
        movl %eax,%cr0
        ret

        ALIGN
1:   movb $1,X86_HARD_MATH
        .byte 0xDB,0xE4      /* fsetpm for 287, ignored by 387 */
        ret

```

코드 831. 커널의 head.S – continued

수치 보조 프로세스(Intel의 경우는 x87)이 있는지를 확인하는 함수이다. 일단 수치 보조 프로세스를 초기화한 후, 상태를 확인한다(fstsw). 만약 없다면 cr0 레지스터의 EM(math emulation)필드를 켜고 복귀하며, 80287을 위한 설정을 해준 후 복귀한다.

이 코드 이하의 부분은 GDT와 IDT, 커널의 스택, swap_page_dir에 대한 구조체 및 기본 인터럽트 핸들러인 ignore_int()함수의 정의가 나온다.

16.6. LILO(Linux Loader)

LILO(Linux Loader)³⁴⁰는 리눅스 커널을 메모리 상으로 옮겨주는 역할을 하는 프로그램이다. 하드디스크가 있는 것을 기본 가정으로 하고 있으며, 다른 운영체제를 가동시켜주기도 한다. LILO는 package 형태로 배포되면 컴파일 및 설치가 가능하다. 또한 이곳에서 설명하는 것은 LILO의 사용법이 아니며, LILO는 자체 다큐먼트도 가지고 있으므로 이후의 설명은 LILO의 다큐먼트를 위주로 해서 설명할 것이다.

먼저 LILO를 다운받아서 설치하게 되면, boot.b 및 os2_d.b, chain.b, dump.b, first.b, second.b 파일이 생성된다. boot.b는 두 부분으로 나누어지며, MBR(Master Boot Record)에 들어갈 내용과 각종 주변 장치를 검사하는 부분으로 나누어진다. 나중에 LILO를 설치하게 될 때 MBR 영역을 전자로 채워주게 된다. os2_d.b는 OS/2를 부팅하기 위해서 사용되며, chain.b는 다른 운영체제를 선택적으로 부팅하기 위해서 사용한다. dump.b는 LILO의 레지스터 상태를 덤프(dump)하기 위해서 사용되며, first.b와 second.b는 boot.b를 만들기 위해서 사용되며, 각각 MBR과 디바이스를 check하는 역할을 한다. 또한, LILO를 설치하면 map이란 파일이 생성되며, 이 파일 속에는 선택할 수 있는 운영체제의 부트 섹터(boot sector)의 위치와 리눅스의 커널 이미지(image)가 들어간다.

16.6.1. Disk의 구성

자, 그럼 booting 과정을 보기위한 첫걸음으로서 disk의 구조에 대해서 알아 보도록 하자. 가장 간단한 disk의 구조로는 floppy disk를 들 수 있겠다. Floppy disk는 크게 boot sector와 data area로 구성되면, 이 경우 data area에 들어가는 내용은 data 및 data 를 관리하기 위한 정보로 구성된다. [그림78]은 floppy disk의 간단한 구성을 보여준다.

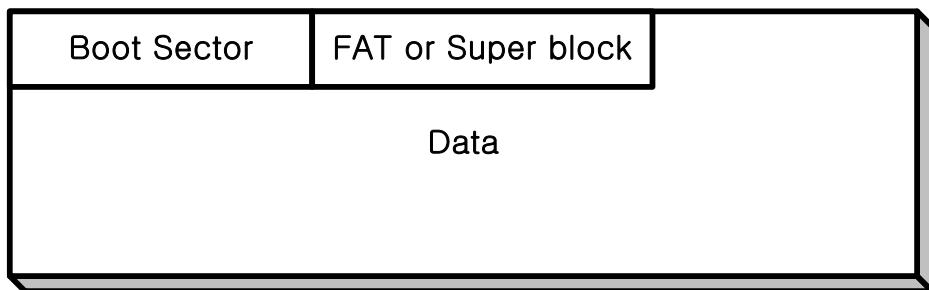


그림 102. Floppy Disk의 구성

Boot sector는 512byte의 영역을 차지하는 작은 영역으로 운영체제를 물리적인 메모리 내로 불러 들이는 역할을 하는 것으로 운영을 달리한다. 따라서, 이 boot sector에 들어가는 내용은 disk의 물리적인 정보를 이용해서 kernel을 물리적인 메모리로 이동시키고, 올라온 kernel로 control을 이동(jump)하는 것만 책임지면 될 것이다. 이러한 boot sector는 컴퓨터가 기동하게 될 때 자동으로 BIOS에 의해서 메모리로 불러 들여지며 control을 넘겨 받는다. [그림79]는 MS-DOS의 경우에 해당하는 boot sector를 간략히 보여준다.

³⁴⁰ LILO버전 21.6.1이 현재의 LILO버전이다. 우린 이것을 기초로해서 코드를 분석할 것이다.

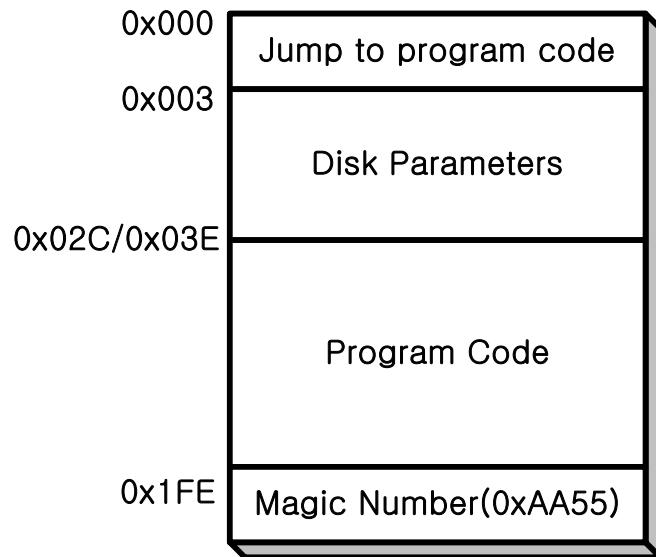


그림 103. MS-DOS의 Boot sector의 구조

MS-DOS의 부트 섹터는 첫부분에 프로그램 코드로 제어를 옮기는 점프(jump)코드와 디스크의 파라미터를 가지는 부분, 프로그램코드, 그리고, MS-DOS 부트 섹터라는 것을 나타내는 매직 넘버(0xAA55)를 가진다.

16.6.2. LILO에서 사용하는 파일

LILO를 사용한 부팅의 경우에 사용되는 파일들 간의 관계는 [그림80]과 같다. 먼저 부트 섹터는 제어를 제일 처음으로 받는 부분으로 주 부트 로더(primary boot loader)를 포함하고 있으며, default 명령어 라인, 두개의 디스크립터 테이블 섹터들에 대한 주소, 두번째 부트 로더의 섹터들에 대한 주소를 가진다. 일반적인 부트 섹터는 **boot.b**³⁴¹가 들어간다. 주 부트 로더는 두번째 부트 로더의 섹터들에 대한 주소를 최대 8개까지만 가질 수 있다.

³⁴¹ 다른 운영체제의 커널을 로드하는 경우에는 chain.b가 사용된다. /boot이하의 디렉토리를 참조하자.

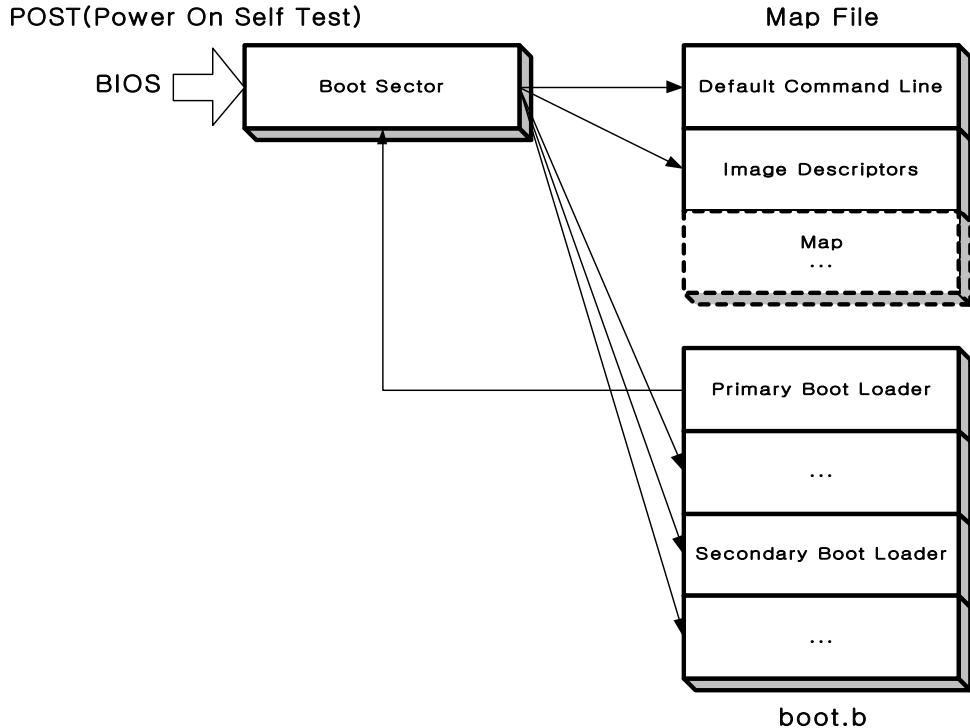


그림 104. LILO에서 사용하는 파일들

먼저 시스템에 파워가 들어오면 POST(Power On Self Test)과정을 수행하게 되고, BIOS는 즉시 부팅 디바이스로 부터 부트 섹터를 읽어들이고, 시스템의 제어를 넘긴다. 부트 섹터는 자신이 찾아서 접근하고자 하는 파일들에 대한 섹터의 주소를 가지고 있어서, 필요한 경우에 각각의 파일을 접근하게 된다.

16.6.3. MAP File

맵(map) 파일은 섹션(section)이라고 불리는 부분과 특수한 데이터 섹터로 구성된다. 각각의 섹션은 디스크의 여러 섹터를 차지할 수 있으며, 다른 파일의 섹터 주소를 가질 수 있다. 다음과 같은 세 가지의 예외를 가진다.

- 만약 hole이 있거나, 혹은 unstripped³⁴² 커널의 플로피 부트 섹터가 없다면, 0 섹터의 주소가 사용된다. 이 섹터는 맵 파일의 일부이다.
- 다른 운영체제를 부팅할 경우, 첫번째 섹터는 merge된 chain loader³⁴³이다. 이 chain loader는 맵 파일의 섹션보다 먼저 나온다.
- 각각의 맵 섹션은 이미지(image)를 기술하며, 이미지에 대한 옵션 라인(line)을 포함하는 섹터가 뒤따른다.

맵 섹터의 마지막 주소부분은 사용되지 않거나 혹은 섹션의 다음 맵 섹터의 주소를 가진다. 또한 맵 파일의 첫 다섯 섹터는 특별한 의미를 가지는데, 첫번째 섹터는 default 명령어 라인을, 다음의 두 섹터는 boot image descriptor 테이블을, 4번째 섹터는 0으로 채워져서 파일이 hole을 가지게 될 때 mapping되는데 사용되며, 다섯번째 섹터는 keyboard translation table을 가진다. 아래의 [그림81]과 같은 구조를 지닌다.

³⁴² 커널의 심벌 정보가 완전히 제거되지 않은 것이 unstripped 커널이다. 완전히 다 제거가 되었다면, stripped 커널이 될 것이다.

³⁴³ FreeBSD와 같은 운영체제를 부팅하기 위해서는 chain.b가 필요하다. 보통은 boot.b와 같은 형식이다.

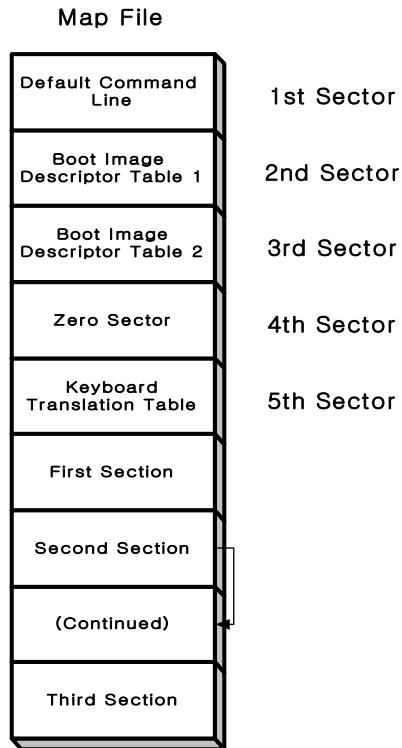


그림 105. Map 파일의 구조

커널의 이미지 부분은 다시 로드(load)될 섹터의 연속으로 구성된다. 또한 맵의 섹션은 예비의(fallback) 명령어 라인과 파라미터(parameter) 라인(line) 옵션도 포함한다. 옵션사항으로 두번째의 맵 섹션에 RAM disk image가 들어가서 로드될 수 있다.

Boot image descriptor는 [그림82]와 같이 구성된다. [그림82]에서는 커널이 이미지와 로드된 RAM disk image에 대한 기술을 볼 수 있다.

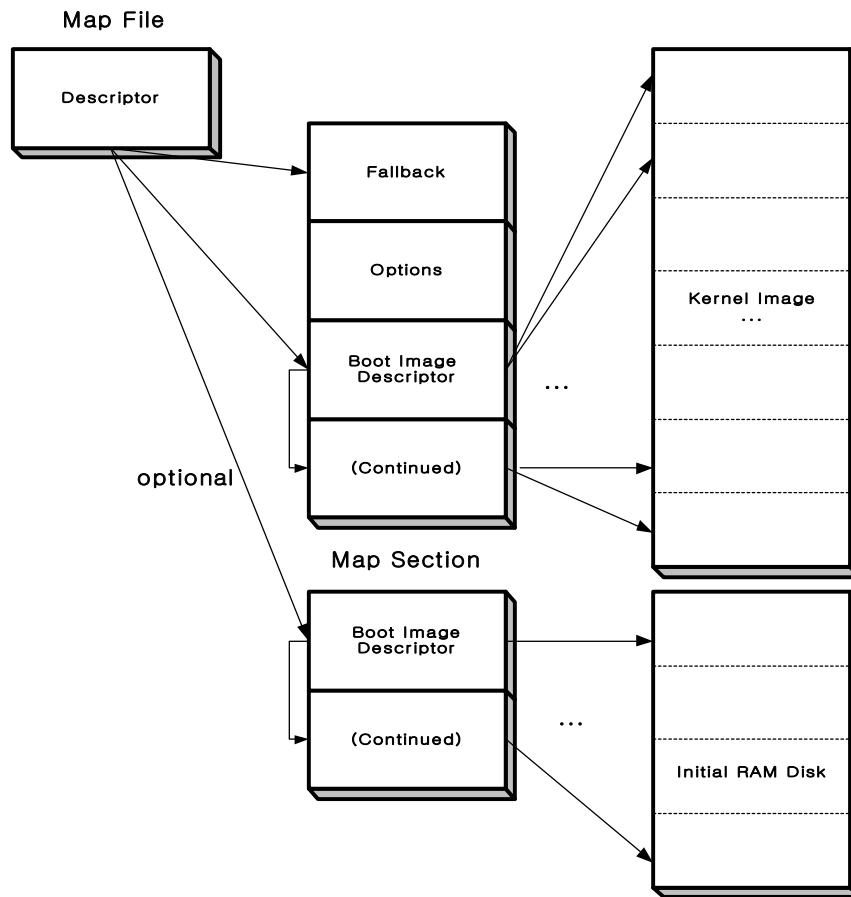


그림 106. Map File의 Image 디스크립터의 구조

만약 다른 운영체제를 부팅하고자 한다면, chain loader(chain.b)가 patch된 파티션 테이블과 함께 합쳐(megre)져서 맵파일에 들어가야 한다. Boot image의 맵 파일의 섹션은 chain loader가 들어갈 섹터의 다음에 나오며, 가짜(dummy) 플로피 부트 섹터³⁴⁴의 주소와 로더의 섹터, 다른 운영체제의 부트 섹터만을 가진다. 맵 섹션은 여기서도 Fallback 섹터와 옵션을 위한 불필요한 섹터는 가진다. 이를 나타내면 [그림83]과 같다.

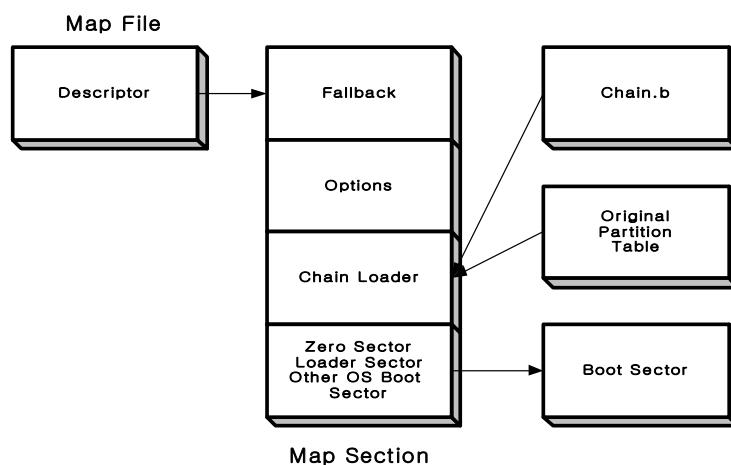


그림 107. 다른 운영체제를 위한 Map File의 Descriptor

³⁴⁴ Zero sector를 말한다. 이것의 내용은 상관하지 않는다.

이전 LILO의 로드 순서를 보도록 하자. LILO로 하는 역할은 이전에 부팅에서 살펴본 부분에서 bootsect.S가 하는 역할과 동일하다. 즉, 전체 압축된 커널의 이미지를 가져와서, 초기화를 실행하고, setup.S로 점프하는 일을 한다. 이후의 과정은 setup.S가 알아서 진행한다.

다시 커널의 부팅을 요약하면, boot sector는 ROM-BIOS에 의해서 0x07C00으로 로드된다. 이것은 다시 0x9A000으로 자신을 이동시키게 되며, 스택을 0x9B000에서 0x9A200으로 설정한다. 스택 영역은 0x9A200에서 하위 방향으로 사용될 것이다. 다시 두번째(secondary) 부트 로더를 0x9B000으로 로드한 후, 제어를 그곳으로 옮긴다. 이때 “L”자를 보여주고³⁴⁵, 두번째 부트 로더를 시작하기 전에 “I”자를 보여준다. 만약 두번째 부트 로더를 로딩하는 과정에서 에러가 있었다면, “L”자 다음에 두개의 숫자가 표시될 것이다. 만약 에러가 영구적이라고 할 경우에는 계속 에러를 발생시킬 것이다. LILO를 컴파일 할 때 NO1STDIAG를 설정했다면, 이러한 에러는 보여지지 않을 것이다.

두번째 부트 로더는 디스크립터 테이블을 0x9D200으로 로드하고, default 명령어 라인을 0x9D600으로 로드한다. 만약 default 명령어 라인이 설정(enable)되었다면, 해당하는 매직 넘버(magic number)는 유효한 값을 가지지 않은 것으로 처리되며, 섹터는 디스크에 다시 기록(write)된다. second.S를 cpp³⁴⁶로 처리할 때, LCF_READONLY를 설정함으로써 이 부분에서 생길 수 있는 치명적인 오류를 막을 수 있다. 두번째 부트 로더는 사용자 input을 검사한 후, 만약 default가 사용되었거나, 사용자가 다른 image를 명시했을 경우는 옵션을 가리키는 섹터가 0x9D600으로 로드되며, 파라미터 라인은 0x9D800에서 구성된다. 만약 결과적으로 만들어진 파라미터 라인에 “lock”을 옵션으로 가질 때는, 사용자에 의해서 입력된 명령어 라인이 새로운 default command line으로 디스크에 저장된다. 또한 fallback 명령어 라인이 설정되었다면, default 명령어 라인으로 복사될 것이다.

만약 사용자가 initial RAM disk 이미지를 주었다면, 16Mbytes나 물리적인 메모리의 끝부분중에서 더 낮은 부분에 로드된다. 시작 주소는 page boundary로 맞춰져서 initial RAM disk가 차지하고 있던 부분이 시스템의 free 메모리 pool로 쉽게 돌려질 수 있도록 한다. 여기서 16Mbytes라는 한계는 BIOS에서 데이터를 이동하는데 사용할 수 있는 최대 주소지정이 24bit에 의해서 나타내지기 때문이다. 다음으로는 이미지의 플로피 부트 섹터가 0x90000으로 로드되며, setup.S 부분은 0x90200으로, 커널 부분은 0x10000으로 로드된다. 만약 커널이 high(big image)로 컴파일 되었다면, 커널은 0x100000으로 로드될 것이다. 이러한 로드 연산중에 맵 파일은 0x9D000으로 로드된다.

만약 로드된 이미지가 커널의 이미지라면, 제어는 setup.S의 코드로 넘어가게 되며, 만약 다른 운영체제가 부팅 되었다면, chain 로더가 0x90200으로 로드되고, 다른 OS의 부트 섹터는 0x90400으로 로드된다. Chain 로더는 파티션테이블³⁴⁷을 0x903BE에서 0x00600으로 옮기며, 부트 섹터는 다시 0x7C00으로 옮긴다. 제어는 이제 부트 섹터로 넘어가게 된다.

또한 두번째(second) 드라이버³⁴⁸에서도 부팅하는 것을 허락하는 chain loader의 경우에는, BIOS에 대한 호출(call)을 가로채서 사용가능한 메모리의 상위(top)에 있는 드라이버 번호와 교체(swap)할 수 있는 작은 함수(small function)을 설치할 수 있다.

두번째 부트 로더는 시작된 후에, 다시 “L”자가 화면에 표시되며, 디스크립터 테이블과 default 명령어 라인을 로드한 후에 “O”자를 표시한다. 디스크립터 테이블은 로드되기전에 정확한 위치에 로드되었는지 확인하게 되며, 만약 그렇지 못할 경우에는 “?”가 표시된다. 만약 디스크립터 테이블이 정확한 checksum을 가지고 있지 않다면 “-“가 표시될 것이다.

아래의 [그림84]는 두번째 로더까지 진행된 이후에 가지는 메모리의 모습이다. 최상위에 drive swapper가 존재한다. 이것은 위에서 설명한 작은 함수(small function)이다. 커널의 setup.S는 0x90200에, 커널은 big image가 아닐 경우에는 0x10000에 로드된다. 주 부트로더 이후에 스택 영역이 있으며, 두번째 부트 로더 이하에 맵이 로드되는 영역이다.

³⁴⁵ 이미지를 옮긴 후.

³⁴⁶ C Pre-Processor: C 전처리자를 말한다. GCC의 경우 cpp가 같이 설치된다.

³⁴⁷ 파티션 테이블은 chain 로더의 일부로서 0x903BE로 로드된다.

³⁴⁸ 플로피나 하드디스크 드라이브.

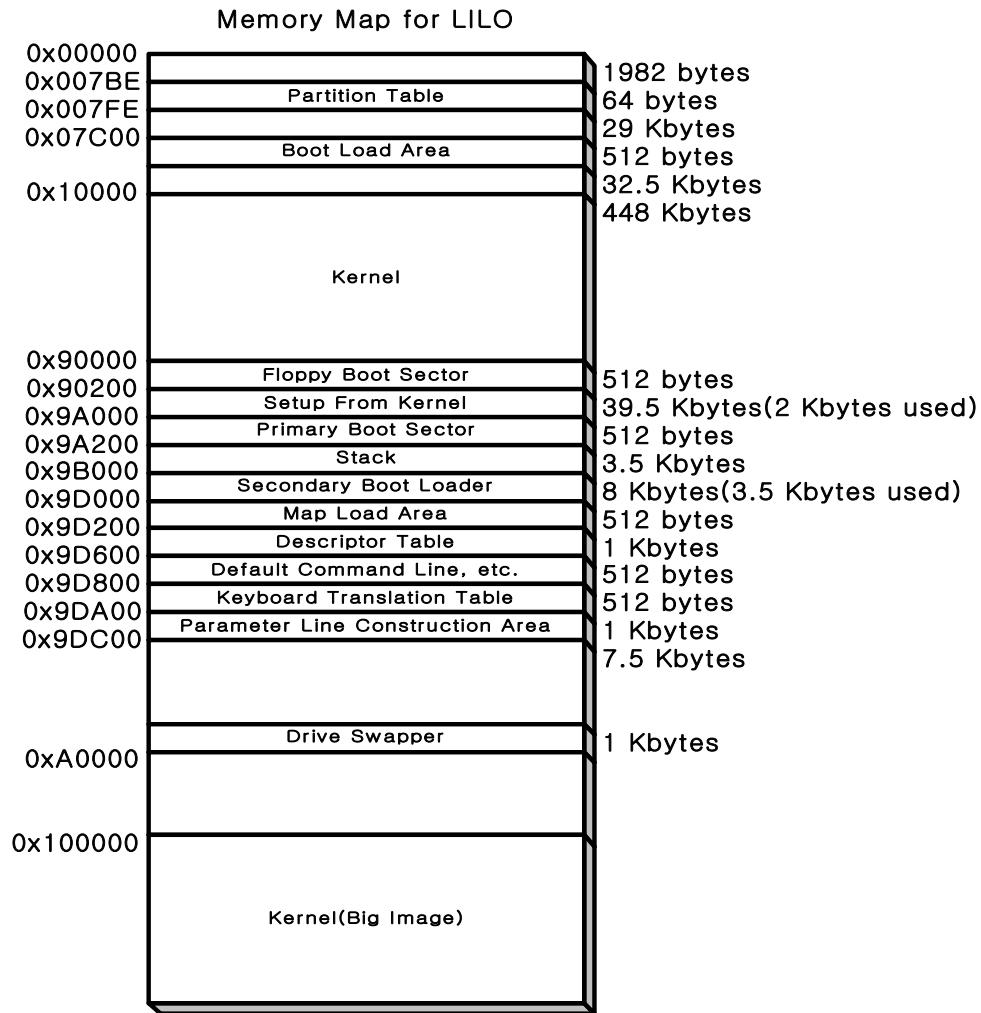


그림 108. LILO의 메모리 구성

메모리 0x90020에서 0x90023부분은 두번째 부트 로더에 의해서 명령어 라인에 의해서 덮혀쓰지게 되며, LILO의 빌드시에 LARGE_EBDA 옵션이 있는 경우에는 0x90000-0x9FFFF는 0x80000에서 0x8FFFF로 바뀌게 된다. 물론 drive swapper의 주소는 사용가능한 메모리의 끝부분을 차지한다.

16.6.4. 파라미터의 설정

각각의 커널의 부트 섹터는 커널이 파일 시스템을 접근하기전, 부팅 시에 사용할 수 있는 설정(configuration) 파라미터를 포함한다. 이러한 파라미터 값은 커널이 컴파일 될 때나 혹은, 나중에 “rdev”와 같은 명령으로 바꿀 수 있는데, LILO에서는 부팅시 메모리에 이러한 파라미터들이 올라오면, 커널로 전달되는 파라미터 라인에 해당 item에 올려놓아서 바꿀 수 있다. 파라미터들은 아래와 같은 옵셋에 저장된다.

Offset	Description
497	섹터 단위(512bytes)의 setup 코드의 크기. 이전 버전의 커널일 경우에는 0을 둔다.
498 ~ 499	루트 파일 시스템을 read-only(001 아닌 경우)나 read-write(0인 경우)로 마운트 한다.
500 ~ 501	커널의 크기를 패러그래프(paragraph = 16 bytes)단위로 나타낸다.
502 ~ 503	unused
504 ~ 505	RAM disk의 크기를 Kbytes단위로 나타낸다. 0으로 설정될 경우에는 RAM disk가 없음.
506 ~ 507	VGA의 텍스트 모드 설정으로 아래의 값을 가진다. 0xFFFFD : VGA모드를 사용자가 부팅시에 명시하도록 한다.

	0xFFFFE : 80x50("extended") 모드로 설정. 0xFFFF : 80x25("normal") 모드로 설정. 이외의 값은 인터액티브한 VGA 모드 선택 메뉴에서 보여지는 해당 모드를 선택한다. 이 부분의 옵션은 LILO에 의해서 boot sector부분을 패치(patch)하는 것으로 설정되는 것으로 파라미터 라인으로 커널에 전달되지는 않는다.
508	루트 파일 시스템으로 마운트될 디바이스의 부번호(minor number).
509	루트 파일 시스템으로 마운트될 디바이스의 주번호(major number).

표 96. 파라미터 옵셋의 정의

또한, 커널은 부트 로더에 의해서 제공되는 파라미터의 처리를 지원한다. 파라미터의 스트링은 NULL로 끝나는 ASCII 스트링(string)으로 되어 있으며, 각각의 스트링은 공란(space)로 구분되는 단어(word)를 가지거나 변수(variable)=값(value)의 쌍으로 되어 있다. 아래와 같은 디스크립터가 커널로 파라미터의 스트링을 보내기 위해서 설정되어 있어야 한다.

- 0x90020 : 매직 넘버로 0xA33F가 온다.
- 0x90022 : 0x90000에 상대적인, 파라미터 라인의 첫번째 byte의 옵셋을 가리킨다.

부트 로더는 명령어 라인과, 옵션들을 가지는 섹터, 내부적으로 생성된 접두어(prefix)를³⁴⁹ 부터 파라미터 라인을 만들며, 아래의 [그림85]와 같이 생성된다.

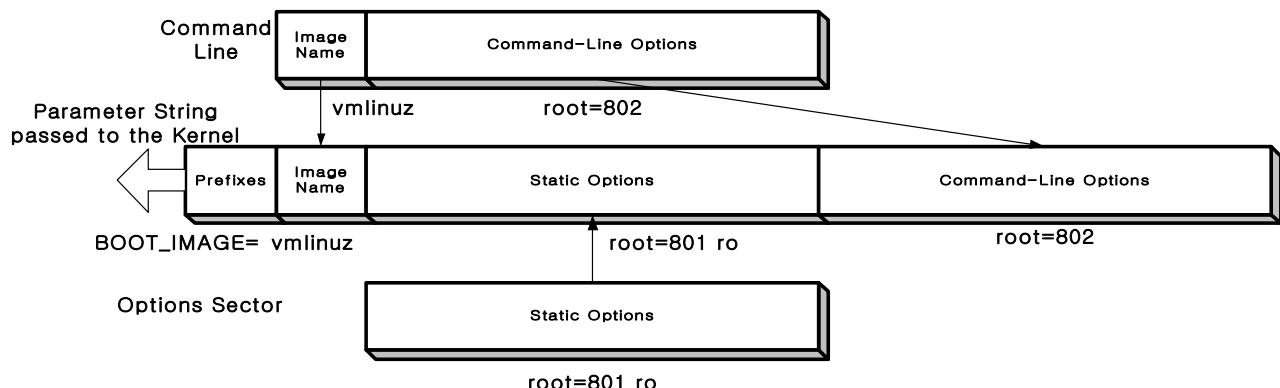


그림 109. 커널 파라미터 라인의 생성

이곳에서 사용된 예는 명령어 라인에서 “vmlinuz root=802”, 옵션들의 섹터에서 “root=801 ro”를 준것이며 생성된 것은 “BOOT_IMAGE=vmlinuz root=801 ro root=802”이다. 파라미터 라인에서 준 “root=801”은 “root=802”로 덮어써지기 override 때문에 커널에서 무시된다.

16.6.5. 외부 인터페이스

LILO는 자신이 실행되기전에 실행된 프로그램으로부터 명령어 라인을 받을 수 있다. 외부에서 제공하는 명령어 라인 방법은 사용자가 일반적인 방법으로 부트 프롬프트(prompt)를 불러내지 않았을 경우에만 사용된다. 다음과 같은 레지스터 값들이 설정되어 있다는 것을 가정한다.

1. DL : 0xFE값을 가진다.
2. ES:SI : “LILO” 스트링을 가리키는 포인터로 스트링은 반드시 대문자로 되어야 하며, 스트링의 끝을 나타내는 문자는 필요없다. 또한 스트링은 세그먼트의 경계를 지나선 안되기에, SI 레지스터의 값을 반드시 0xFFFFD이하여야 한다.

³⁴⁹ “auto”나 “BOOT_IMAGE=”와 같은 것들이 해당한다.

3. ES:BX : NULL로 끝나는 스트링을 가르키는 포인터이다. 스트링의 최대 길이는 78자이며(물론 NULL을 제외하고), 세그먼트 경계를 지나지 않아야 한다. 만약 ES:BX가 NULL을 가리킨다면, boot prompt로 들어가서 keyboard input을 받아들이라는 요청으로 해석되며, 스트링이 공백(blank)으로 구성되어 있다면, default boot image로 부팅한다는 것으로 해석된다.

LILO는 default 명령어 라인을 맵 파일에서 가져올 수 있으며, 이것은 외부에서 제공 되는 명령어 라인이 없을 경우에만 사용된다.

맵 파일에 들어있는 default 명령어 라인은 맵 파일의 첫번째 섹터를 차지한다. 만약 사용자가 shift키로 부트 프롬프트를 불러내지 않았거나, 외부에서 주어지는 명령어 라인이 없을 경우에는 맵 파일에 들어 있는 명령어 라인이 마치 사용자가 키보드로 입력한 값처럼 해석될 것이다.

맵 파일의 첫번째 섹터의 첫 2 bytes는 반드시 매직 넘버로 DC_MAGIC(0xF4F2)값을 little-endian 형식으로 가져야 한다. 그 뒤에는 바로 NULL로 문자로 끝나는 스트링이, NULL문자까지 포함해서 최대 510 bytes까지 올 수 있다. 하지만 부트 로더에서는 공백을 지운 후 78자까지만을 명령어 라인으로 한정한다. 무시하고자 한다면 명령어 라인의 시작 두 바이트의 매직 넘버를 지우던가, 아니면 명령어 라인에 NULL문자만을 남겨놓으면 된다.³⁵⁰

16.6.6. bootsect.S의 분석

bootsect.S는 커널에 있는 bootsect.S와 동일한 역할을 수행한다. 이곳에 있는 코드들은 전부 AS86 assembler를 사용해서 컴파일되기 때문에, 이전에 있던 커널의 bootsect.S의 코드와는 다르다. 즉, 이전에 있던 커널의 bootsect.S의 경우에는 GCC를 이용해서 컴파일을 수행한다. bootsect.S는 커널의 bootsect.S와 같은 역할을 하며, 코드만 달리 작성되었기에 이곳에서는 더 이상 논하지 않기로 한다.

16.6.7. first.S의 분석

first.S는 boot.b의 한 영역을 차지하며, 512bytes의 크기를 가진다. boot sector영역에 들어가기 위해서 그 정도의 크기를 가지게 되며, second.S를 로드(load)해 주는 역할을 한다. 소스 코드³⁵¹를 보도록 하자.

```

start:    mov    ax,#BOOTSEG      ! set DS
          mov    ds,ax
          mov    ext_es,es ! copy possible external parameters
          mov    ext_si,si
          mov    ext_bx,bx
          mov    ext_dl,dl
          mov    ax,#FIRSTSEG    ! beam us up ...
          mov    es,ax
          mov    cx,#256
          sub    si,si
          sub    di,di
          cld
          rep
          movsw
          jmpi   go,FIRSTSEG

```

먼저 유의할 점은 GNU assembler와는 달리 이곳에서의 연산은 뒤에 있는 연산자를 앞으로 보내는 차이가 있다. 가령 예를 들어서,

```
        mov ax, #BOOTSEG
```

라고 할 경우 ax에 BOOTSEG(0x07C0)를 옮기라는 명령이 된다. 컴파일러로는 GAS가 아니라 as86을 사용한다.

³⁵⁰ 이상에서 언급한 부분은 LILO의 Technical Document의 내용을 저자가 생각하는 방향으로 새로이 정렬해 놓은 것일 뿐이다. 더 보고자 한다면, LILO의 source코드와 함께 제공되는 문서를 참고하라.

³⁵¹ 이곳에서 참조하는 모든 코드는 LILO 버전 21.6.1이다. 또한 코드는 AS86문법으로 작성되었다.

코드를 보면 먼저 DS(Data Segment) 레지스터를 설정하고, 있을지도 모를 외부 인터페이스(interface)를 위한 세그먼트 레지스터를 설정한 후, FIRSTSEG(0x9A00) 세그먼트로 256bytes를 옮긴다. 그리고 나서

```
jmpi go,FIRSTSEG
```

와 같이해서 제어를 옮겨진 세그먼트로 전달한다.

go:	cli	! no interrupts
	mov ds,ax	! AX is already set
	mov es,ax	! (ES may be wrong when restarting)
	mov sp,#STACK	! set the stack
	mov ax,#STACKSEG	
	mov ss,ax	
	sti	! now it is safe
	mov al,#0x0d ! gimme a CR ...	
	call display	
	mov al,#0x0a ! ... an LF ...	
	call display	
	mov al,#0x4c ! ... an 'L' ...	
	call display	

새로이 옮겨진 주소에서 먼저 인터럽트가 발생하지 못하도록 한 후(cli), DS및 ES를 옮겨진 영역으로 설정하고, 스택영역을 설정해 준다. 다시 인터럽트를 쓸 수 있도록 만들어주고, CR(Carriage Return)과 LF(Line Feed)및 ‘L’자를 화면에 찍어준다.

lagain:	mov si,#d_addr	! ready to load the second stage loader
	mov bx,#SECOND	
	cld	
sload:	lodsw	! get CX
	mov cx,ax	
	lodsw	! get DX
	mov dx,ax	
	or ax,cx	! at EOF ?
	jz done	! yes -> start it
	inc si	! skip the length byte
	call cread	
	jc error	! error -> start over again
	add bx,#512	! next sector
	jmp sload	
...		
done:	mov al,#0x49 ! display an 'T'	
	call display	
	jmpi 0,SECONDSEG	! start the second stage loader

이제는 second.S를 옮길 준비가 된 상태이다. bx에 SECOND(0x1000)를 넣어주고, 옮기는 방향에 대한 설정을 한 후(cld-Clear Direction), second.S를 읽어온다. 만약 다 읽었다면, done으로 제어를 옮겨서 ‘I’를 표시한 후, 곧바로 SECONDSEG(0x9B00)로 제어를 옮겨준다. (jmpi 0, SECONDSEG)

다 읽은 것이 아니라면, 다시 cread() 함수를 호출해서 다음의 한 섹터(sector)를 디스크로부터 읽어드린다.

cread:	test dl,#LINEAR_FLAG	! linear address ?
	jz readsect	! no -> go on
	and dl,#0xff-LINEAR_FLAG	! remove flag

`cread()` 함수는 디스크로부터 한 섹터를 읽어오는 역할을 한다. 먼저 LINEAR_FLAG(0x40)으로 dl 레지스터와 비교한다. 이곳에서 현재 선형적(linear)으로 주소를 사용하고 있는지 확인하고, 만약 그렇지 않다면 바로 한 섹터를 읽어 들인다. 그렇다면, dl 레지스터에 0xBF(0xFF-0x40)값과 AND를 한 값을 저장한다.

```

push    bx          ! BX is used as scratch
push    cx          ! LSW
push    dx          ! MSW with drive
mov     ah,#8        ! get drive geometry (do not clobber ES:DI)
int     0x13
jc     linerr       ! error -> quit
mov     al,dh        ! AL <- #heads-1
pop     dx          ! get MSW
mov     t_drive,dl ! save drive
mov     dl,dh        ! linear address (high) into DX
xor     dh,dh
#endif CYL_CHECK
push    cx          ! compute #cyls-1
xchg   ch,cl
rol    ch,1
rol    ch,1
and    ch,#3
mov     n_cyl,cx ! save #cyls-1
pop     cx
#endif
and    cx,#0x3f      ! CX <- #secs
mul    cl          ! AX <- #secs/cyl
add    ax,cx
xchg   ax,bx
pop    ax          ! linear address (low) into AX
div    bx          ! DX <- cylinder, AX <- remaining secs
xchg   ax,dx
div    cl          ! AL <- track, AH <- sector
inc    ah
mov    t_sector,ah
xchg   ax,dx        ! AX <- cylinder, DL <- track
mov    dh,dl        ! set up DX (head:drive)
mov    dl,t_drive
#endif CYL_CHECK
cmp    ax,n_cyl      ! valid cylinder number ?
ja     linerr3       ! no -> error
#endif
xchg   ah,al        ! build cylinder number
ror    al,1
ror    al,1
or    al,t_sector
mov    cx,ax
pop    bx          ! restore BX
readsect:
    mov    ax,#0x201    ! read one sector
    int    0x13
    ret             ! quit, possibly with errors

#endif CYL_CHECK
linerr3:pop    bx          ! pop BX and linear address
                xor     ax,ax        ! zero indicates internal error
                stc

```

```
ret
```

나머지는 새로이 하나의 섹터를 디스크로부터 읽기 위해서 섹터와 트랙, 실린더 등을 결정하는 절차다. 다 결정되고 나면 ax 레지스터에 0x201을 넣고, 인터럽트 0x13을 호출해서 하나의 섹터를 읽어 들인다. 다 읽었다면 이전 수행하던 곳으로 다시 돌아가서 수행을 계속하게 된다. 즉, 원하는 second.S의 섹터를 다 읽어 들였는지를 확인하고 그곳으로 제어를 옮기는 것이다.

여기까지 제대로 수행되었다면, 화면에는 “LI”라는 글자가 나오게 될 것이다.

16.6.8. second.S의 분석

second.S의 역할은 실제적으로 커널을 읽어드리고, 그곳으로 점프(jump)하는 일이다. 몇 개의 sector를 읽어드리고 나서, (리눅스의 경우에는 setup sector를) 읽어드린 곳으로 제어를 옮기는 것으로 자신의 일을 마친다. 이때, 만약 LILO의 명령 행(command line)에 어떤 부팅 옵션(option)이 있는지를 확인하고, 그것을 적절히 다음 프로그램에서 사용할 수 있도록 메모리에 보관한다. 이것을 위해서 second.S에서는 명령 행을 읽고 저장하는 함수를 가지고 있다.

second.S의 일은 압축된 커널의 이미지에서 setup.S를 실행시켜주는 부분까지다. 이하의 부팅 과정은 앞에서 설명한 과정과 같다.

17. Embedded Linux Implementation

앞에서 우린 일반적인 PC상에서의 Linux 커널이 어떻게 구현되어 있는지를 살펴보았다. 여기서부터는 Embedded 환경에서 Linux를 어떻게 적용시켜 나가는지를 살펴보게 될 것이다. 참고로 하는 것은 Intel의 SA1100 마이크로 프로세스를 기준으로 할 것이며, 실제 적용하는 것은 삼성전자의 Web Video Phone이 될 것이다. Intel의 SA1100은 ARM7TDMI를 기본 코어(core)로 가지고 구현된 Embedded용 microprocessor로서 널리 사용되고 있기에 이곳에서 다루는 기본이 될 것이다.

17.1. What is embedded system?

Embedded Linux에 대한 구현에 대해서 알아보기 전에, 우리는 embedded system이 무엇인 지부터 이해해야 한다. Embedded system이란 특정 application에 적합하도록 구현된 시스템을 의미한다. 될 수 있으면, cost-effective하게 구성되기 때문에 각종 요구 사항에서 minimize하는 것을 목표로 만들어진다. Embedded system을 정의하는 말로는 여러 가지가 있지만, 그 중에서 가장 잘 기술하고 있다고 생각되는 것은 바로 “사용자의 프로그래밍이 필요로 하지 않는 시스템”이 적절한 것으로 보인다. 즉, 사용자는 단지 몇 개의 키 조작(혹은 전혀 그러한 조작이 필요 없기도 하다.)으로 시스템을 운용할 수 있다는 말이 된다. 즉, 모든 개발 및 프로그래밍에 관련된 것은 제품을 만드는 vendor의 책임이며, 이를 service하고 update하는 것도 개발사의 몫이 된다. 사용자는 다만 그 제품을 적절한 목적으로 사용하기만 하면 되는 것이다. 따라서, embedded system은 일종의 전제 제품처럼만 사용자에게 보여질 것이며, 내부에는 무엇이 있는지를 모르게 된다. 최대한 이러한 rule을 잘 지키고, 사용자의 조작이 쉽게 만들어질 수록 좋은 제품이라고 할 수 있을 것이다. 너무 많은 기능과 사용자의 간섭이 필요하다면, 제품에 대한 manual만 익히는데도 시간이 많이 들 것이다. 하지만, 요즘에는 전문가적인 사용자들인 경우에는 이러한 것을 원하기도 한다. 어쨌든, 이번 장에서는 많은 단말기와 PDA에 사용되고 있는 ARM core를 사용한 embedded system을 어떻게 만드는가를 보기로 하겠다.

17.2. Cross Compiler

Embedded Linux 구현하는 과정에서 가장 먼저 해야 할 과정은 Cross Compiler를 만드는 일이다. 이것은 나중에 커널을 컴파일하고, 각종 프로그램들을 컴파일하는데 사용된다. Cross Compile을 하는 이유는 개발환경에서 직접적으로 컴파일을 할 수 없기에, 개발하는 환경을 다른 Architecture를 사용하는 컴퓨터에 구성하고, 여기서 모든 개발과정을 마칠 수 있도록 하기 위해서이다. 이때 사용되는 것이 바로 호스트(host)와 타깃(target)이라는 용어이다. 즉, 호스트는 개발환경이 갖추어진 컴퓨터를 말하며, 타깃은 개발한 프로그램들이 올라가는 환경을 말한다. Cross compile을 하기 위해서 필요한 파일들은 아래와 같다.

- Binutils : 이것은 binary utility의 약자로서 ar, nm, ld, ranlib, objdump등등의 이진 파일을 다루는데, 필요한 utility들을 가지고 있다. GNU 사이트로부터 download받을 수 있을 것이다.
- GCC : 이것은 GNU cc의 약자로서 컴파일러이다. C compiler 및 Fortran, C++등등의 컴파일러를 가지고 있다. 역시 GNU 사이트로부터 download받을 수 있을 것이다.
- GLIBC : 이것은 GNU C Library이다. C 프로그램을 컴파일 하기 위해서 필요한 Library를 가지고 있다. GCC와 GLIBC는 컴파일 하기 위해서 Architecture에 의존적인 커널을 필요로 한다. 역시 GNU site로부터 download받을 수 있다.

위와 같은 Tool들은 전부 GPL License를 따르기에 개발과정에서 source의 형태로 다운 받아서, 새로이 컴파일 해서 설치해야 할 것이다. 아래에서 이들을 컴파일하고 설치하는 과정에 대해서 보도록 하겠다.

17.2.1. Binutil Compile

가장 먼저 compile할 것은 binutil³⁵²이다. 보통 대개의 경우 이 프로그램을 다운 받아서 설치하는 경우에는 별로 어려움이 없을 것이다. 아래와 같은 명령의 순서로 수행하도록 한다.

³⁵² 이 글을 쓰고 있는 현재 Binutil의 버전은 2.10.1까지 ftp.gnu.org에 올라와 있었다. 따라서, 이것을 사용하도록 하겠다.

```
#tar binutils-2.10.1.tar.gz
#cd binutils-2.10.1
#./configure --target=arm-linux --prefix=/usr/local/arm
#make
#make install
```

먼저, tar는 압축된 파일을 푸는 과정이다. 풀고 나면 디렉토리가 생성되므로 이곳으로 이동한다(cd). 그리고 나서, configure를 수행하는데, 이때 주는 옵션이 여러 가지가 있을 수 있겠으나, 우린 target을 사용해서 어떤 시스템용으로 컴파일 할 것인지와 prefix를 사용해서 어디에 설치할 것인가를 결정하는 일만 해준다. 별로 에러가 발생하지는 않을 것이다. 그럼, 다음으로 넘어가기 전에, 잠시 아래와 같이 binutils를 설치한 디렉토리의 실행 파일들이 있는 bin 디렉토리를 자신이 사용하는 환경의 path로 넣어주도록 하자. 사용하는 shell에 맞춰서(bash와 csh에 맞게), PATH 환경변수를 설정하면 될 것이다.

```
#vi ~/.bashrc    <bash shell을 사용하는 경우>
#vi ~/.cshrc    <csh shell을 사용하는 경우>
export PATH=/usr/local/arm/bin:$PATH    <bash shell을 사용하는 경우>
set PATH=(/usr/local/arm/bin $PATH)      <csh shell을 사용하는 경우>
```

이렇게 해주어야지 나중에 다른 프로그램들을 컴파일 할 경우에도 아무런 문제가 없을 것이다. 설정이 끝났다면, 자신의 현재 실행 환경을 바꾸기 위해서 source ~/.bash(혹은 source ~/.cshrc)를 해주도록 하자. 자, 이제는 GCC를 컴파일 할 차례이다. 하지만, GCC는 커널에 의존적인 부분이 있기 때문에, 먼저 커널을 가져와서 적당한 디렉토리에 설치해 주어야 할 것이다.

17.2.2. Kernel Installation

커널을 설치하기에 적당한 디렉토리로 좋은 곳이 /usr/src 부분이 될 것이다. 물론 여러 개의 커널이 이곳에 있을 수 있다. /usr/src를 보면 이전 버전의 커널이 linux라는 link를 통해서 연결되어 사용되고 있을 것이다. 아래와 같이 제거하도록 하자.

```
#cd /usr/src
#rm linux
```

이젠 새로운 커널을 받아와야 할 것이다. 먼저 모든 Linux 커널³⁵³을 받을 수 있는 곳으로 좋은 곳은 www.kernel.org이므로 이곳에서 최신 버전의 커널을 받도록 하자. 그리고 나서, ARM용 patch³⁵⁴ 및 Intel StrongARM용 patch³⁵⁵을 받도록 하자. 받아왔다면, 이젠 아래와 같은 과정으로 진행하도록 한다.

#cp linux-2.4.2.tar.gz /usr/src	-> /usr/src로 커널 소스를 복사한다.
#cp patch-2.4.2-rmk1.gz /usr/src	-> ARM patch level 1을 /usr/src로 복사한다.
#cp diff-2.4.2-rmk1-np3.gz /usr/src	-> Nico StrongARM patch level 3을 /usr/src로 복사한다.
#cd /usr/src	-> /usr/src로 디렉토리를 바꾼다.
#tar xvfz linux-2.4.2.tar.gz	-> 압축된 커널을 푸다.
#gunzip patch-2.4.2-rmk1.gz	-> 압축된 ARM patch를 푸다.
#gunzip diff-2.4.2-rmk1-np3.gz	-> 압축된 Nico StrongARM patch를 푸다.
#patch -p0 < patch-2.4.2-rmk1	-> ARM patch를 한다.
#patch -p0 < diff-2.4.2-rmk1-np3	-> Nico StrongARM patch를 한다.
#mv linux linux-rmk1-np3	-> 디렉토리의 이름을 바꾼다.

³⁵³ 현재 이 글을 작성하는 중에 나와 있는 최신버전의 커널은 2.4.3버전이다. 하지만, 잘 컴파일에 조금 문제가 있어서, 커널 2.4.2버전을 기준으로 작성했다.

³⁵⁴ [ftp://ftp.arm.linux.org.uk/pub/armlinux/source/kernel-patches/v2.4/](http://ftp.arm.linux.org.uk/pub/armlinux/source/kernel-patches/v2.4/)에서 구할 수 있을 것이다. 현재는 2.4.3에 대한 patch를 구할 수 있을 것이다.

³⁵⁵ [ftp://ftp.arm.linux.org.uk/pub/linux/arm/people/nico/](http://ftp.arm.linux.org.uk/pub/linux/arm/people/nico/)에서 구할 수 있을 것이다. 역시 2.4.3에 대한 patch를 구할 수 있을 것이다.

```
#ln -s linux-rmk1-np3 linux          -> 디렉토리에 symbolic link를 설정한다.
```

위와 같은 과정에서 중요한 것은 ARM용 patch와 StrongARM용 patch를 해주어야 한다는 점이다. 빼먹지 않도록 주의하자. 이전 커널을 설정(configuration)하도록 하자. 이 과정을 마치면 일부 커널의 디렉토리 설정이 만들어질 것이다. 즉, /usr/src/linux/include/asm link가 asm-arm으로 설정되며, /usr/src/linux/include/linux/version.h가 생성될 것이다. 아래와 같이 하도록 하자.

```
#cd /usr/src/linux
#make menuconfig
... < System type>을 선택해서 제대로 만들기를 원하는 것이 선택되어 있는지 확인한다. 우리의 경우에는 (SA1100-based) ARM system type을 선택하면 될 것이다.
... < Configuration>을 저장하고 나온다.
#vi Makefile
... < ARCH := arm & CROSS_COMPILE = arm-linux-> 있는지 확인한다. 없다면, 써주어도 될 것이다.
#make dep <- version.h를 생성함.
```

이 과정을 마치면, 나머지 cross compile환경을 구성하기 위한 Linux 커널의 header가 적절하게 설정되었을 것이다. 이전 GCC를 컴파일 할 준비가 다 되었다.

17.2.3. GCC Compile

GCC는 프로그램을 컴파일 하기위해서 사용된다. 컴파일 전에 몇 가지의 설정할 디렉토리가 있다. 이는 참조하기 위한 header 파일의 디렉토리이다. 아래와 같이 하도록 하자.

```
#cd /usr/local/arm/arm-linux
#mkdir include
#cd include
#ln -s /usr/src/linux/include/asm-arm asm      -> asm과 linux include 디렉토리 link를 생성한다.
#ln -s /usr/src/linux/include/linux linux
#ls -al asm                                     -> 제대로 link가 생성되었는지 확인한다.
#ls -al linux
```

위의 과정을 마치면, GCC와 GLIBC를 컴파일 하기위한 header 파일을 찾는 곳에 대한 지정이 끝나게 된다. GCC³⁵⁶를 download 받아서 설치하는 과정으로 들어가자. GCC이외에 역시 patch가 또 필요하다. ARM용 패치³⁵⁷를 download 받도록 하자.

처음 하는 GCC에 대한 컴파일은 우리가 사용하게 될 Glibc를 가지지 못하였기에, Glibc를 컴파일 하기위한 GCC이다. 이를 native compiler라는 말을 사용해서 나타내기도 한다. 이후에 Glibc를 컴파일 한 후 다시 GCC를 컴파일 하게 될 것이다. 아래와 같은 과정으로 하도록 하자.

```
#tar xvfz gcc-2.95.3.tar.gz           -> GCC의 압축을 푼다.
#bzip2 -d gcc-2.95.3-diff-010218.bz2 -> GCC의 patch 압축을 푼다.
#cd gcc-2.95.3
#patch -p1 < ../gcc-2.95.3-diff-010218 -> GCC의 patch를 한다.
```

이것을 마쳤다면, Dinhbit_libc hack을 고쳐주어야 한다. 이것은 처음으로 GCC를 컴파일 할 때 발생할 수 있는 문제를 해결하기 위한 것이다. 아래와 같이 하도록 하자.

```
#cd gcc/config/arm
#vi t-linux
```

³⁵⁶ GCC는 [ftp://ftp.gnu.org/gnu/gcc/](http://ftp.gnu.org/gnu/gcc/)에서 받을 수 있을 것이다. 현재 이 글을 쓰는 시점에서는 2.95.3버전이 있어서 그것을 사용했다.

³⁵⁷ ARM용 GCC patch를 받을 수 있는 곳은 [ftp://ftp.armlinux.org/pub/toolchain](http://ftp.armlinux.org/pub/toolchain) 이다. 이곳에는 현재 2.95.3버전에 대한 patch를 제공하고 있다.

... <아래와 같은 줄을 찾아서 고치도록 하자.>

TARGET_LIBGCC2_CFLAGS = -fomit-frame-pointer -fPIC를

TARGET_LIBGCC2_CFLAGS = -fomit-frame-pointer -fPIC -Dinhibit_libc -D__gthr_posix_h

이젠 compile하는 단계이다.

```
#cd gcc-2.95.3                                > GCC의 암축을 해제한 디렉토리로 바꾼다.
#./configure --target=arm-linux --prefix=/usr/local/arm --disable-threads --with-cpu=strongarm110 --enable-languages=c
#make
#cd gcc
#./xgcc --dumpmachine                         -> GCC가 제대로 만들어 졌는지 확인한다.
arm-linux                                         -> 이와 같은 것을 출력하면 올바른 것이다.
#cd ..
#make install
#cd /usr/local/arm/bin
#./arm-linux-gcc -v                            -> 만들어진 GCC의 버전 정보를 확인한다.
Reading specs from /usr/local/arm/lib/gcc-lib/arm-linux/2.95.3/specs  -> 이와 같은 정보가 나올 것이다.
gcc version 2.95.3 20010315 (release)
```

--enable-languages=c와 같이 해준 것은 단지 C 컴파일러만을 build하기 위한 것이다. 또한 --disable-threads를 해준 것은 아직 thread에 대한 기능을 제공하는 library가 없기 때문이다. 여기서 한가지 확인할 것은 /usr/local/arm/lib/gcc-lib/arm-linux/2.95.3/specs가 아래와 같은 것을 가지고 있는지를 보는 일이 남았다.

```
...
*link:
%{h*} %{version:-v} %{b} %{WI,*:%*} %{static:-Bstatic} %{shared:-shared} %{symbolic:-Bsymbolic}
%{rdynamic:-export-dynamic}%{!dynamic-linker:-dynamic-linker /lib/ld-linux.so.2}-X %{mbig-endian:-EB}
%{mapcs-26:-m armelf_linux26} %{!mapcs-26:-m armelf_linux} -p
...
```

이와 같이 되어 있다면, GCC의 컴파일은 무사히 마친 것이다. 하지만, 나중에 Glibc를 설치한 후에, 위와 비슷한 과정을 한번 더 해주어야 한다. 이 과정 중에서 잘 되지 않는다면, 아래같이 해 주도록 하자.

<앞에서 configure까지를 마쳤다고 가정하자.>

```
#cd gcc-2.95.3/libiberty
#make install
#cd ../../gcc
#make
#./xgcc --dumpmachine                         > GCC가 제대로 만들어 졌는지 확인한다.
arm-linux
#make install
#cd /usr/local/arm/bin
#./arm-linux-gcc -v                            -> 만들어진 GCC의 버전 정보를 확인한다.
Reading specs from /usr/local/arm/lib/gcc-lib/arm-linux/2.95.3/specs  -> 이와 같은 정보가 나올 것이다.
gcc version 2.95.3 20010315 (release)
```

즉, 여기서는 개별적인 디렉토리에 접근해서 GCC를 컴파일하고 설치하는 것이다. 접근하는 디렉토리는 위에서와 같이 libiberty와 gcc이다.

17.2.4. Glibc Compile

Glibc³⁵⁸ 도 GNU Tool이므로 ftp.gnu.org에서 download받을 수 있다. 필요한 것은 crypt³⁵⁹ 와 linuxthreads, localeata, glibc이다. 아래와 같이 하도록 하자.

```
#tar xvfz glibc-2.2.2.tar.gz
#cd glibc-2.2.2
#tar xvfz ./glibc-linuxthreads-2.2.2.tar.gz          -> 현재 디렉토리 위에 glibc-linuxthreads-2.2.2.tar.gz가
                                                               있다고 가정한다.
-> 만약 이전 버전의 Glibc를 컴파일 하고 있다면, 추가적으로 아래의 두 가지를 더 압축을 해제하도록 한다.
#tar xvfz ..../glibc-crypt-X.X.X.tar.gz
#tar xvfz ..../glibc-locatedata.X.X.X.tar.gz          -> 이것은 대체로 필요하지 않을 수도 있으니, 꼭
                                                               해줄 필요가 없다.
-> 추가적으로 만약 LD_LIBRARY_PATH가 현재의 디렉토리를 가지지 않아야 한다는 메시지가 나오면
~/.bashrc 파일에 있을지도 모르는 LD_LIBRARY_PATH를 변수를 없애기 위해서 아래와 같이 한다.
#vi ~/.bashrc
export LD_LIBRARY_PATH=

#source ~/.bashrc

#CC=arm-linux-gcc AR=arm-linux-ar ./configure arm-linux --prefix=/usr/local/arm/arm-linux
--enable-add-ons=linuxthreads
-> 만약 이전 버전의 Glibc를 컴파일 하고 있다면, 아래와 같이 하도록 하자.
#CC=arm-linux-gcc ./configure arm-linux --prefix=/usr/local/arm/arm-linux
--enable-add-ons=linuxthreads,crypt,locatedata

-> 마지막으로 위와 같이 해서 잘 안될 경우에는 --with-headers=/usr/src/linux/include360 를 덧붙여서
해보도록 하자. 또한, AR이 잘못 설정된 경우에는(arm-linux-ar로 config.make 파일에 설정이 안된 경우)
config.make 파일에서 “AR = arm-linux-ar”로 바꾸어 준다.
-> 앞의 과정에서 크기가 큰 파일들을 설치하기 때문에 종종 파일 시스템이 full이 되는 경우가 있기에
주의하도록 한다.
```

위의 과정에서 중요한 것은 CC=arm-linux-gcc를 적는 부분과 ./configure 다음에 바로 --target를 쓰지 않고, arm-linux가 온다는 것이다. 주의해야 할 것이다. 이것이 잘되었다면, 현재 디렉토리의 config.make 파일에 CC에는 arm-linux-gcc가 있을 것이며, AR에는 /usr/local/arm/arm-linux/bin/ar0이 설정되어 있을 것이다. 확인하도록 하자.

```
#make
#make install
```

이제 설정이 다 끝났으므로, compile하고 설치하는 단계이다. make, make install을 순차적으로 실행하면 될 것이다. Glibc 말고 다른 public하게 사용할 수 있는 library로는 newlib³⁶¹ 가 있다. 이것은 <http://sources.redhat.com/newlib/>에서 download받을 수 있다. 이것에 대한 것은 이 문서에서는 설명하지 않도록 하겠다. 관심이 있는 사람은 그곳에서 직접 받아서 설치해 보기 바란다. 과정은 별로 다른 점이

³⁵⁸ Glibc 버전 2.2.2와 linuxthreads 버전 2.2.2, locatedata 버전 2.0.6, crypt 버전 1.09를 ftp.gnu.org에서 찾을 수 있을 것이다. 물론 이 글을 쓰는 현재 그렇다는 말이다.

³⁵⁹ 2.2.2 버전의 Glibc를 configure를 하면, 이미 crypt가 포함되어 있다는 말이 나올 것이다. 이럴 경우에는 crypt를 사용하지 않아도 될 것이다. 하지만, 2.2.2버전 아래의 Glibc를 컴파일 하기 위해서 필요하므로 적어두도록 하겠다. locatedata 역시 같은 이유에서 적어두었다.

³⁶⁰ 물론 이것은 우리가 앞에서 압축을 해제한 커널이 들어있는 곳의 include 디렉토리이다.

³⁶¹ 이 글을 쓰고 있는 현재 1.9.0 버전까지 나와있다.

없다. Glibc를 컴파일하는 시간은 상당히 길 것이므로 잠시 커피라도 한잔하는 것이 좋을 것이다. 여기까지 문제없이 되었다면 새로이 GCC를 컴파일 해주는 일이 남았다.³⁶²

17.2.5. GCC re-compile

GCC를 다시 컴파일 하는 것은 앞에서 했던 `-Dinhibit_libc` hack을 지우는 것이다. 또한 앞에서 `-disable`과 같이 만들었던 부분을 `configure`를 실행할 때 지우도록 하고, `language`로 C만을 지원하도록 했는데, 이젠 C++및 다른 언어도 사용할 수 있으므로 이것도 없애도록 하자. 이 시점에서 중요한 것은 이미 Glibc를 우린 가지고 있다는 사실이다. 아래와 같이 해서 다시 컴파일 한 후 설치하도록 하자.

```
#cd gcc-2.95.3/gcc/config/arm
#vi t-linux
<앞에서 -Dinhibit_libc -D__gthr_posix_h를 추가했던 부분을 찾아서 아래와 같이 원래대로 복구한다.>
...
TARGET_LIBGCC2_CFLAGS=-fomit-frame-pointer -fPIC
#cd gcc-2.95.3/libchill
#vi basicio.c
<#define PATH_MAX를 추가해 주어야 한다. 이것은 나중에 컴파일 하는 도중에 PATH_MAX가 정의되지
않아서 발생하는 error를 막기 위함이다. #ifndef PATH_MAX 앞에 #define PATH_MAX 4095를 넣어 두도록
하자. 대략 40 line정도에 있다.>
#cd gcc-2.95.3
#./configure --target=arm-linux --prefix=/usr/local/arm --with-cpu=strongarm110
#make
#make install
```

여기까지 다 되었다면, 기본적인 ARM용 application을 컴파일 할 준비를 마친 것이다. 간단한 테스트용 프로그램을 만들어서 시험해 보도록 하자. 만약 만들고자 하는 프로그램이 Hello World!!!를 화면에 프린트 하는 프로그램이라고 가정하자.

```
#vi hello.c
<간단히 include에 stdio.h를 넣고, printf() 함수를 사용해서 Hello World!!!를 출력하는 프로그램을
만들자.>
#arm-linux-gcc -o hello hello.c
#ls -al hello
<여기서 제대로 컴파일이 되었다면, hello라는 프로그램이 실행 파일 형태로 만들어졌을 것이다.
#file hello
-> 파일이 어떤 특성을 가지고 있는지를 보여준다. 제대로
되었다면 아래와 같은 것이 화면에 보일 것이다.
hello:ELF 32-bit LSB executable, Advanced RISC Machines ARM, version 1, dynamically linked (uses shared libs),
not stripped
```

여기까지 했으면, 일단 커널은 compile하는 것이 가능하다. 다음으로 넘어가기 전에, `/usr/src/linux`에 가서, 아래와 같이 해보자.

```
#cd /usr/src/linux
#make menuconfig
<커널의 설정 사항을 정하고, configuration을 저장한 후, 나온다.>
#make dep
#make
-> 커널을 컴파일 해서 object파일을 만든다.
#cd /usr/src/linux
#ls -al
```

³⁶² Glibc를 컴파일하는 도중에 에러가 생긴다면, 상당히 난해할 것이다. 이럴 경우에는 다시 Glibc를 컴파일 하는 단계의 처음으로 돌아가서 압축을 풀고, `configure`를 실행한 다음 하는 것이 좋다. 그렇게 하지 않으면, 이전에 했던 `configure`의 영향으로 `config.cache`와 같은 것들이 그대로 남아서 여러모로 문제를 일으킬 소지가 많다.

```

...
<여기서 커널이 제대로 만들어졌다면, vmlinux가 생성되어 있을 것이다.>
#cd /usr/src/linux
#make zImage
-> 커널의 압축 이미지와 loader를 더해서 object 파일을
   만든다363.
#cd /usr/src/linux/arch/arm/boot
#ls -al
...
<여기서 제대로 압축된 커널의 이미지가 생성되었다면, zImage가 보이게 될 것이다.>
```

마지막으로 위에서 각종 프로그램을 컴파일하는 도중에 프로그램의 소스가 너무 커서 디스크가 가득차는 경우가 있을 수 있으므로 항상 자신의 Linux partition이 여유가 있는지 확인해야 할 것이다. 또한 컴파일 도중에 에러가 많이 있을 수 있는데, 이때는 뭘 수 있으면, 처음부터 새로 해주는 것이 좋을 것이다. 즉, 한 프로그램의 압축을 해제하고, patch한 후 다시 configure해서 설치하도록 하자.

17.2.6. Reference Web Site

참고할 만한 Web Site로는 다음과 같은 것들이 있으니, Cross Compiler를 만드는 도중에 문제가 생기면 새로운 버전이나 patch가 존재하는지 알아보도록 하자.

- www.armlinux.org/docs/toolchain : 이곳은 ARM Toolchain을 만드는 방법에 대한 문서를 구할 수 있는 곳이다.
- ftp.arm.linux.org.uk/pub/armlinux/source/kernel-patches/ : 이곳에서는 ARM용 kernel patch를 구할 수 있는 곳이다.
- ftp.arm.linux.org.uk/pub/linux/arm/people/nico/ : 이곳은 StrongARM용 nico라는 사람이 만든 kernel patch를 받을 수 있는 곳이다.
- www.arm.linux.org.uk : 이곳은 Linux와 관련된 ARM 정보를 찾을 수 있는 곳이다.
- www.netwinder.org : 이곳 역시 Linux와 관련된 ARM용 정보를 찾을 수 있으며, 여러 가지의 Tool들을 미리 만들어서 RPM package로 만들어두고 있으므로 Cross Compiler를 직접 만들기 귀찮은 사람은 여기서 다운 받아서 사용할 수 있을 것이다.
- www.cs.cmu.edu/~wearable/software/assabet.html : 이곳에서는 Intel의 SA1100 Assabet evaluation board에 대한 것을 다루고 있다.
- ftp.handhelds.org/pub/linux/arm/toolchain/source : 이곳에서도 많은 Binary utility source를 받을 수 있을 것이다. Patch도 물론 받을 수 있다.
- www.handhelds.org : 위의 사이트에 같이 포함시켜서 알아두자.
- www.lart.tudelft.nl : ARM용 Linux에서 대한 많은 정보를 가지고 있다.
- www.kesl.org : 한국의 Embedded Linux에 관련된 장소로 게시판과 자료집 들이 있다.
- www.uclinux.org : uClinux 사이트로 Embedded Linux에 대한 정보를 찾을 수 있을 것이다.

이상에서 우린 간단하게 남아, 응용 프로그램과 커널을 개발할 수 있는 환경의 구축해 보았다. 처음으로 Cross Compiler를 만드는 일은 그렇게 쉬운 일이 아니며, 많은 시행 착오를 해서 만들어 지기도 한다. 이전 본격적으로 ARM에서 프로그램을 개발하기 위해서 가장 기본적으로 필요한 Assembly Language에 대해서 보도록 하자. Intel x86기반의 PC보다는 Assembly Language가 상당히 쉬울 것이라고 생각되지만, 처음이니 힘들기는 마찬가지 일 것이다.

17.3. SA1100의 GPIO(General Purpose Input/Output)

GPIO(General Purpose Input/Output)란 28개로 이루어진 port pins으로서 application의 특수 목적에 따라, input이나 혹은 output의 신호를 발생하거나 잡기를 원할 목적으로 사용되는 것이다. 전체 28개의 GPIO를 위한 GPIO port pin이 존재하며, 각각의 pin은 input이나 혹은 output으로 프로그램되거나 혹은 인터럽트

³⁶³ 이 과정에서 에러가 있을 수 있다. /usr/src/linux/arch/arm/boot/compressed에 들어가서 head-sa1100.S를 확인해서 에러가 나는 곳을 보자. 아마 “:q”가 보일 것이다. 편집하다가 잘못한 것으로 생각된다.

source로 사용될 수 있다. 모든 28개의 pin은 reset이 발생하면 모두 input으로 설정되며, 바꿔기 전에는 그대로 input으로 남을 것이다.

GPIO pin은 **GPIO pin direction register(GPDR)**을 프로그램하므로써 input이나 혹은 output으로 사용될 수 있다. 만약 output으로 프로그램이 된다면, **GPIO pin output set register(GPSR)**와 **GPIO pin output clear register(GPCR)**에 어떤 값을 넣어서 제어될 수 있으며, 이러한 레지스터에 값을 쓴다는 말은 직접적으로 읽기와 쓰기가 되지 않는 output data 레지스터를 콘트롤하게 된다. 물론 이와 같은 레지스터들은 pin이 input이나 output이건 간에 상관없이 write가 가능하지만, 프로그램된 output 상태는 pin이 output으로 설정될 경우에만 효력이 나타난다.

GPIO pin이 input으로 프로그램될 경우에는 현재의 각 GPIO pin의 상태는 **GPIO pin-level register(GPLR)**을 통해서 읽을 수 있다. 이 register는 output으로 pin이 설정될 때, 현재의 pin의 상태를 확인하기 위해서 언제든지 읽혀질 수 있다. 덧붙여서 **GPIO rising-edge detect register(GRER)**와 **GPIO falling-edge register(GFER)**를 사용해서 각각의 GPIO pin은 rising-edge나 falling-edge를 감지하도록 프로그램될 수 있다. 각각의 pin의 감지된 상태는 다시 **GPIO edge detect status register(GEDR)**를 읽어서 알 수 있다. 이러한 edge는 인터럽트의 발생을 감지한다거나, SA1100을 sleep mode에서 깨어나게 하는 event로 사용할 수 있다. 또한 어떤 GPIO pin들은 다른 목적으로 사용할 수 있는데(alternate function), LCD나 serial controller와 같이 데이터의 입출력을 위해서 부가적인 pin을 요구하는 경우에 GPIO pin을 할당해서 쓰도록 만든다. 이러한 목적을 위해서 **GPIO alternate function register(GAFR)**가 있다. 이하에서는 위에서 언급한 레지스터들을 좀더 자세히 보도록 하겠다³⁶⁴.

종합적으로 이야기 하자면, GPIO를 위한 제어 블록에 들어가는 레지스터는 총 8개가 있으며, 하나는 pin의 상태를 보는데 사용하며, 다른 pin의 상태를 제어하는데, pin의 방향을 설정하는데 하나의 pin이, pin의 edge 타입을 설정하는데 2개의 pin과 상태를 보고하기 위한 플랙으로 사용하는 하나의 pin, alternate function을 나타내기 위해서 다시 하나의 pin을 사용한다. 주의할 점은 GPDR만이 reset에 의해서 초기화된다는 점이다. 나머지 pin들은 reset후에 값이 결정되지 않으며, 반드시 소프트웨어 적으로 초기화를 해주어야 할 것이다.

17.3.1. GPIO Pin-Level Register(GPLR)

각각의 GPIO port pin들의 상태를 보여주는 레지스터이며, 0에서 27번까지의 bit이 port pin번호에 일치한다. 이 레지스터는 read-only이며, 특정 pin의 현재 레벨을 결정하기 위해서 사용한다. 즉, 이 레지스터는 프로그램된 pin의 상태와는 상관이 없다. 각 bit의 포맷과 해당하는 의미는 아래와 같다.

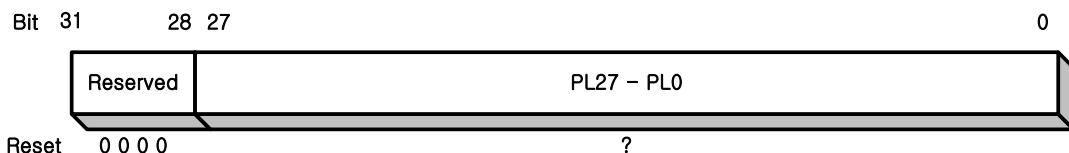


그림 110. GPLR

[그림 110. GPLR]에서 reset후에 읽어오는 내용은 “Reset”이라고 되어있는 부분이며, 이곳에 “?”라고 되어있다면, unknown 상태임을 말한다. 따라서, reset 후에 예약된 부분(Reserved)을 제외하면 전부 unknown상태가 된다.

Bit	Name	Description
n	PL(n)	GPIO port pin level bit n(where n = 0 through 27) 0 - pin state가 low이다. 1 - pin state가 high이다.
31..28	--	예약된 부분이다.

표 97. GPLR 레지스터의 bit 정의

³⁶⁴ 이곳에서 나오는 내용은 Intel의 SA1100의 메뉴얼에서 가져온 내용이므로 직접 참고하기 바란다.

17.3.2. GPIO Pin Direction Register(GPDR)

GPDR 레지스터는 pin의 방향성(direction)을 설정하는 역할을 한다. 28개의 port pin들에 대해서 각각 한 bit를 씩 가지고 있으며, 만약 이 bit이 1로 설정된다면, out으로 port가 사용된다는 것을 의미하며, 0으로 설정되었다면, input을 의미한다. 하드웨어적인 reset이 발생하면, 이 레지스터에 있는 모든 bit은 지워지게 되며, soft reset이나 기타 sleep reset과 같은 상황은 이 레지스터에 영향을 미치지 않는다. 28에서 31 bit까지의 reserved bit에 대해서는 write는 무시되며, read는 0을 돌려줄 것이다.

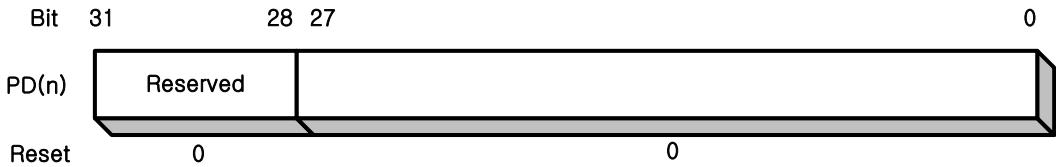


그림 111. GPDR

[그림 111]은 GPDR레지스터의 bit 정의를 보여준다. 이 그림에서 reset후에는 전체 register의 bit이 0이 됨을 알 수 있다. 각각의 pin에 대한 정의는 아래와 같다.

Bit	Name	Description
n	PD(n)	GPIO port pin direction n(where n = 0 through 27) 0 - pin이 input으로 설정되었다(configure). 1 - pin이 output으로 설정되었다(configure).
31..28	--	예약된 부분이다.

표 98. GPDR 레지스터의 bit정의

17.3.3. GPIO Pin Output Set Register(GPSR)과 Pin Output Clear Register(GPCR)

만약 port가 output으로 설정되면, 사용자³⁶⁵는 GPSR레지스터나 GPCR레지스터에 write를 함으로써, pin의 상태를 제어할 수 있게된다. GPSR의 해당 bit을 1로 write함으로써, output pin을 설정할 수 있으며, 다시 이 output pin을 clear하기 위해서 GPCR의 해당 bit에 1을 write한다. 즉, 이 레지스터들은 write-only 레지스터인 것이다. 이 레지스터들에 대한 read는 예측할 수 없는 값을 돌려줄 것이며, GPSR이나 GPCR에 0을 write하는 것은 아무런 효과가 없다. 또한 input으로 설정된 pin에 대해서, 해당 bit에 1을 write하는 것 역시 아무런 효과가 없다. 예약된 bit에 대한 write는 무시될 것이며, reset후에 가질 수 있는 값은 write-only이므로 사용할 이유가 없다.

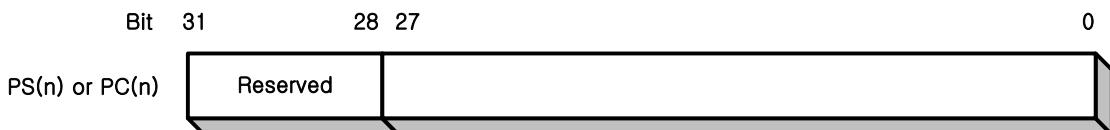


그림 112. GPSR or GPCR

[그림 112]은 GPSR과 GPCR레지스터의 bit정의를 보여준다. 각각의 bit에 대한 정의는 아래의 표와 같으며, reset후에 가질 수 있는 bit의 값은 의미가 없다.

Bit	Name	Description
n	PS(n)	GPIO output pin set n(where n = 0 through 27) 0 - pin level은 영향받지 않는다. 1 - 만약 pin이 output으로 설정되었다면, pin level을 high(1)으로 설정한다.
31..28	--	예약된 부분이다.

³⁶⁵ 물론 여기서 말하는 사용자(user)란 디바이스 드라이버나 커널 코드가 될 것이다.

표 99. GPSR 레지스터의 bit정의

Bit	Name	Description
n	PC(n)	GPIO output pin clear n(where n = 0 through 27) 0 - pin level은 영향받지 않는다. 1 - 만약 pin이 output으로 설정되었다면, pin level을 low(0)로 설정한다.
31..28	--	예약된 부분이다.

표 100. GPCR 레지스터의 bit 정의

사용자는 pin에 대해서 외부의 충돌(conflict)가 있는지를 결정하기 위해서, pin의 상태를 설정했거나 지운(clear) 후에 output으로 설정된 pin에 해당하는 GPLR 레지스터의 bit를 검사할 수 있다. 예를 들어서, GPIO output pin을 high(1)로 만드는 off-chip 디바이스³⁶⁶가 있고, 사용자가 pin의 상태를 GPCR에 1을 write함으로써 clear시켰다면, 사용자는 GPLR을 읽어서, write한 값과 실제값과의 비교를 통해서 충돌(conflict)을 감지할 수 있게된다. 즉, 지운 결과가 반영되는지를 확인하기 위해서 GPLR을 읽는 것이다.

17.3.4. GPIO Rising-Edge Detect Register(GRER)과 Falling-Edge Detect Register(GFER)

각각의 GPIO port는 rising-edge나 falling-edge 및 둘 다를 감지할 수 있도록 프로그램될 수 있다. 만약 각각의 pin에 프로그램된 edge가 감지(detect)되면, 해당 bit에 이와 같은 상태가 보고가 될 것이다(1로 설정될 것이다.). 이와 같은 특성은 interrupt controller가 해당 bit의 설정에 따라서 CPU로 signal을 보내도록 프로그램될 수 있도록 만들어주며, 또한 SA1100이 sleep mode에서 깨어날 수 있도록 해줄 수 있다. 이러한 목적으로 사용하기 위해서 GRER과 GFER 레지스터가 있는 것이다. 즉, GRER과 GFER은 GPIO pin의 상태변화에 대한 형(type)을 가지고 있어서, 만약 어떤 형(type)의 edge가 감지되면, 이를 GEDR(GPIO Edge Detect Register)를 통해서 보고하게 된다. GRER은 논리적인 값이 0에서 1로 pin의 상태가 변하게될 때 감지하도록 설정하며, GFER은 논리적인 값이 1에서 0으로 바뀔 때, 감지하도록 설정하는 레지스터이다. 이렇게 감지된 상태는 앞에서 말했듯이 GEDR레지스터의 해당 bit을 설정(1)할 것이다.

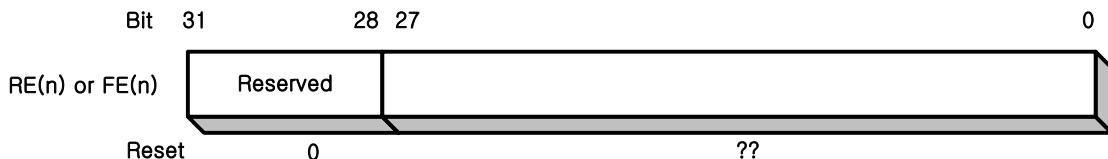


그림 113. GRER or GFER

[그림 113]은 GRER 레지스터와 GFER 레지스터의 bit정의를 보여준다. 여기서 reset후의 값은 reserved 영역에 대해서는 0으로 설정되며, 나머지 영역의 값은 정해지지 않는다는 것을 알 수 있다. 각 bit에 대한 정의는 아래의 표를 보기 바란다.

Bit	Name	Description
n	RE(n)	GPIO pin n rising-edge detect (where n = 0 through 27) 0 - disable rising-edge detect. 1 - GPIO pin에서 rising-edge가 감지되면, 해당 GEDR bit을 1로 설정하라.
31..28	--	예약된 부분이다.

표 101. GRER 레지스터의 bit정의

Bit	Name	Description
-----	------	-------------

³⁶⁶ CPU core이외의 외부장치를 말하는 것으로, 일반 디바이스로 생각하면 될 것이다.

n	FE(n)	GPIO pin falling-edge detect (where n = 0 through 27) 0 - disable falling-edge detect. 1 - GPIO pin에서 falling-edge가 감지되면, 해당 GEDR bit을 1로 설정하라.
31..28	--	예약된 부분이다.

표 102. GFER 레지스터의 bit정의

예약된 부분의 bit에 대한 write는 무시되며, read는 0을 돌려줄 것이다. Reset에서는 예약된 영역은 0을 가지며, 나머지 부분의 bit들은 값이 정해지지 않았다.

17.3.5. GPIO Edge Detect Status Register(GEDR)

GRER이나 GFER 레지스터에 설정된 값에 따른 edge가 GPIO pin에서 감지되면, 이 감지된 결과는 GEDR 레지스터의 각 GPIO pin에 해당하는 bit이 설정된다. GEDR 레지스터 bit은 한번 설정이 되고나면, CPU는 반드시 이것을 지워주어야 한다. 지우는 것은 해당 GEDR 레지스터의 bit에 1을 쓰는 것으로 가능하다. 0을 GEDR 레지스터에 쓰는 것은 아무런 효과가 없다.

각각의 edge 감지는 GEDR 레지스터의 bit를 설정하게되며, GPIO pin 0에서 27까지에 대해서 GEDR 레지스터가 상태를 유지해준다. 또한 이것은 CPU에 대해서 interrupt 요청을 발생시키게 되며, 결과적으로 CPU는 이 인터럽트를 처리했다는 것을 알려주기 위해서 GEDR 레지스터의 해당 bit에 1을 write하는 것으로 지우기(clear)를 할 것이다. 여기서 한가지 주의 할 점은 GPIO pin 11에서 27까지가 하나의 group을 이루어서 하나의 interrupt 요청을 발생시킨다는 점인데, 이때 GEDR 레지스터의 11에서 27까지의 bit이 사용된다. 하지만, 0에서 10번까지의 GPIO pin은 독립적인 각각의 인터럽트 요청을 발생하도록 되어 있다. 따라서, 정확한 인터럽트 요청을 처리하기 위해서는 인터럽트가 발생했을 때, 해당 인터럽트가 어떤 것인가를 파악하기 위해서 GEDR 레지스터를 읽고 이를 처리해 주도록 해야 할 것이다.

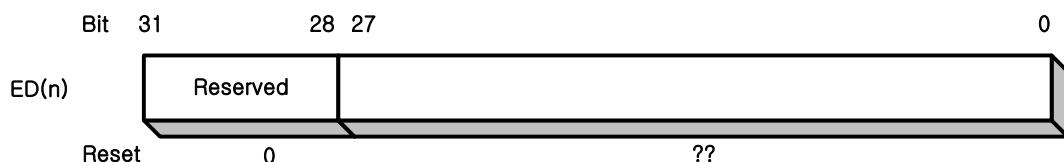


그림 114. GEDR

[그림 114]는 GEDR 레지스터의 bit정의를 보여준다. 이곳에서 reset 후에 가질 수 있는 값은 예약 영역에 대해서는 0이며, 나머지 영역은 정의되지 않았다. 각 bit의 역할은 아래의 표와 같다.

Bit	Name	Description
n	ED(n)	GPIO edge detect status n (where n = 0 through 27) 0 - GRER이나 GFER에 명시된 edge-detect가 발생하지 않았다. 1 - GRER이나 GFER에 명시된 edge-detect가 발생했다.
31..28	--	예약된 부분이다.

표 103. GEDR 레지스터의 bit정의

17.3.6. GPIO Alternate Function Register(GAFR)

만약 CPU가 GAFR 레지스터의 한 bit을 설정하면, 해당 GPIO pin은 alternate function으로 전환된다. 이 pin은 나중에 다른 목적으로 사용하기 위해서 설정된 것이다. Reset이 발생하면, 모든 bit은 0으로 지워지게 된다. GAFR을 사용하는 이유는 GPIO pin을 디바이스에 대한 input이나 output으로 사용하기 위한 것으로 나중에 LCD와 같은 디바이스에 사용되고 있다.

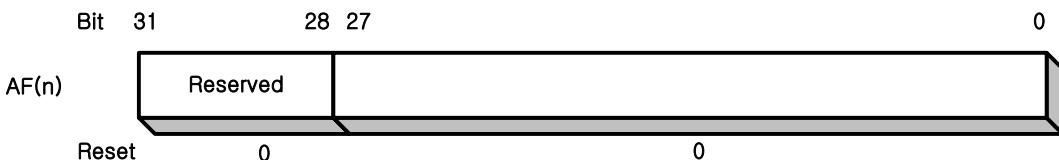


그림 115. GAFR

[그림 115]은 GAFR 레지스터의 bit정의를 보여준다. Reset후에는 전체 레지스터의 bit들이 전부 0값으로 초기화 됨을 알 수 있다. 각각의 bit에 대한 정의는 아래의 표와 같다.

Bit	Name	Description
n	ED(n)	GPIO alternate function bits (where n = 0 through 27) 레지스터에 있는 bit들은 해당하는 GPIO pin이 alternate function으로 사용됨을 1로 설정해서 나타낸다. 0은 해당하는 GPIO pin이 normal하게 사용되고 있음을 나타낸다.
31..28	--	예약된 부분이다.

표 104. GAFR 레지스터의 bit정의

SA1100 보드에서는 GPIO pin들이 alternate function으로 사용되도록 GAFR 레지스터를 프로그래밍 할 수 있다. 만약 GPIO가 alternate function으로 사용된다면, GPIO로서는 동시에 사용되지 못한다. Pin 0와 pin 1은 sleep mode에서 특수한 목적으로 사용되도록 예약되어 있기에 alternate function으로 사용하지 못한다. 다음에 보여주는 표는 SA1100에서 GPIO pin을 alternate function으로 어떻게 사용할 수 있는가를 보여준다.

Pin	Alternate Function	Direction	Unit	Signal Description
27	32KHZ_OUT	Output	Clocks	Raw 32.768-kHz oscillator output.
26	RCLK_OUT	Output	Clocks	Internal clock/2
25	RTC clock	Output	RTC	Trimmed 1-Hz clock
24	Reserved	--	--	--
23	TREQB	Input	Test controller	TIC request B
22	TREQA/MBREQ	Input	Test controller	Either TIC request A or MBREQ
21	TIC_ACK/MBGNT	Output	Test controller	Either TIC acknowledge or MBGNT
21	MCP_CLK	Input	Serial port 4	MCP clock in
20	UART_SCLK3	Input	Serial port 3:UART	Sample clock input
19	SSP_CLK	Input	Serial port 2: SSP	Sample clock input
18	UART_SCLK1	Input	Serial port 1:UART	Sample clock input
17	SDLC_AAF	Output	Serial port 1:SDLC	Abort after frame control
16	SDLC_SCLK	I/O	Serial port 1:SDLC	Geoport clock out
15	UART_RXD	Input	Serial port 1:UART	UART receive
14	UART_TXD	Output	Serial port 1:UART	UART transmit
13	SSP_SFRM	Output	Serial port 4: SSP	SSP frame clock
12	SSP_SCLK	Output	Serial port 4: SSP	SSP serial clock
11	SSP_RXD	Input	Serial port 4: SSP	SSP receive
10	SSP_TXD	Output	Serial port 4: SSP	SSP transmit
2..9	LDD<8..15>	Output	LCD controller	High-order data pins for split-screen color LCD support
1	Reserved	--	--	No alternate function
0	Reserved	--	--	No alternate function

표 105. GPIO alternate function의 정의

표에서 볼 수 있는 것들 중에서 디바이스 드라이버를 만들 때 유념하고 보아야 할 부분은 바로, LCD에 대한 부분이 될 것이다. 즉, LCD에 대한 데이터를 보내고자 할 때 사용하는 것이 LDD<8..15>이다. 이것은 screen상에 뿌려지게될 데이터의 output을 위해서 사용하는 GPIO pin이다.

17.3.7. GPIO Register Location

이제 GPIO에 대한 설명 중에서 남은 것은 과연 어디에 GPIO 레지스터들이 존재하는가 일 것이다. 커널이나 디바이스 드라이버는 이 위치에 존재하는 GPIO 레지스터들에 대한 연산을 수행할 것이기 때문이다. 아래의 표와 같다.

Location	Name	Description
0x9004 0000	GPLR	GPIO pin level register
0x9004 0004	GPDR	GPIO pin direction register
0x9004 0008	GPSR	GPIO pin output set register
0x9004 000C	GPCR	GPIO pin output clear register
0x9004 0010	GRER	GPIO rising-edge detect register
0x9004 0014	GFER	GPIO falling-edge detect register
0x9004 0018	GEDR	GPIO edge detect status register
0x9004 001C	GAFR	GPIO alternate function register

표 106. GPIO 레지스터의 위치 정의

위와 같은 GPIO 레지스터들은 Linux 커널의 ~/include/asm-arm/arch-sa1100/SA-1100.h에 아래와 같이 정의되어 있다.

```
#define _GPLR 0x90040000 /* GPIO Pin Level Reg. */
#define _GPDR 0x90040004 /* GPIO Pin Direction Reg. */
#define _GPSR 0x90040008 /* GPIO Pin output Set Reg. */
#define _GPCR 0x9004000C /* GPIO Pin output Clear Reg. */
#define _GRER 0x90040010 /* GPIO Rising-Edge detect Reg. */
#define _GFER 0x90040014 /* GPIO Falling-Edge detect Reg. */
#define _GEDR 0x90040018 /* GPIO Edge Detect status Reg. */
#define _GAFR 0x9004001C /* GPIO Alternate Function Reg. */

#if LANGUAGE == C

#define GPLR (*((volatile Word *) io_p2v (_GPLR)))
#define GPDR (*((volatile Word *) io_p2v (_GPDR)))
#define GPSR (*((volatile Word *) io_p2v (_GPSR)))
#define GPCR (*((volatile Word *) io_p2v (_GPCR)))
#define GRER (*((volatile Word *) io_p2v (_GRER)))
#define GFER (*((volatile Word *) io_p2v (_GFER)))
#define GEDR (*((volatile Word *) io_p2v (_GEDR)))
#define GAFR (*((volatile Word *) io_p2v (_GAFR)))

#elif LANGUAGE == Assembly

#define GPLR (io_p2v (_GPLR))
#define GPDR (io_p2v (_GPDR))
#define GPSR (io_p2v (_GPSR))
#define GPCR (io_p2v (_GPCR))
#define GRER (io_p2v (_GRER))
#define GFER (io_p2v (_GFER))
#define GEDR (io_p2v (_GEDR))
#define GAFR (io_p2v (_GAFR))

#endif /* LANGUAGE == C */
```

코드 832. Linux에서의 GPIO 레지스터 위치

여기서 io_p2v() 매크로는 Physical Address를 가상 주소로 변환하는데 사용하며, 각 레지스터들은 이 매크로를 통해서 가상 주소 공간에서의 위치를 알 수 있게 된다. 이 매크로의 정의는 ~/include/asm-arm/arch-sa1100/hardware.h에 아래와 같이 되어 있다.

```
/*
 * SA1100 internal I/O mappings
 *
 * We have the following mapping:
 *   phys           virt
 *   80000000      f8000000
 *   90000000      fa000000
 *   a0000000      fc000000
 *   b0000000      fe000000
 */
#define VIO_BASE      0xf8000000      /* virtual start of IO space */
#define VIO_SHIFT     3                /* x = IO space shrink power */
#define PIO_START     0x80000000      /* physical start of IO space */

#define io_p2v( x )    \
((((x)&0x00ffff) | (((x)&0x30000000)>>VIO_SHIFT)) + VIO_BASE )
#define io_v2p( x )    \
((((x)&0x00ffff) | (((x)&(0x30000000>>VIO_SHIFT))<<VIO_SHIFT)) + PIO_START )
```

코드 833. 물리적인 주소와 가상 주소 공간 사이의 변환

즉, 물리적인 주소 0x90000000은 가상 주소 0xFA000000으로 내부적인 I/O mapping에 주소 공간의 할당이 일어난다. 물리 주소를 가상 주소로 고치는데는 io_p2v() 매크로를 사용하며, 이를 역으로 수행하는 것이 io_v2p() 매크로이다.

17.4. SA1100의 Interrupt Controller

Intel SA1100의 interrupt controller는 모든 interrupt의 source에 대해서 masking할 수 있도록 하고 있으며, 또한 그들을 둑어서 CPU interrupt인 **FIQ**(Fast Interrupt)나 **IRQ**(Normal Interrupt)로 만들수 있도 있다. 즉, SA1100은 앞에서 본 GPIO와 경우와 마찬가지로 2 level interrupt 구조를 가지고 있다.

첫번째 level에 해당하는 인터럽트 구조는 **ICIP**(Interrupt Controller IRQ Pending) 레지스터와 **ICFP**(Interrupt Controller FIQ Pending) 레지스터로 대표되는 것으로서, 모든 enabled되고 masking이 되지 않은 인터럽트의 source를 가진다. 인터럽트를 masking하는 레지스터로는 **ICMR**(Interrupt Controller Mask Register)가 있으며, **ICIP**는 IRQ interrupt를 발생하도록 할 때 프로그램되는 레지스터이다. **ICFP**는 모든 유효한(valid) 인터럽트들을 포함하는 것으로, FIQ interrupt를 발생시키도록 프로그램 된 것들이다. 위에서 사용되는 interrupt routing 방법을 결정하는 것이 바로 **ICLR**(Interrupt Controller Level Register)이다.

두번째 level에 해당하는 인터럽트 구조는 source device에 포함된 레지스터들로서, 디바이스는 단지 첫번째 level의 인터럽트를 발생시키기만 하고, 인터럽트의 원인을 알기 위해서는 반드시 이러한 레지스터들의 status를 파악해야 한다. 즉, 두번째 level의 interrupt status는 인터럽트에 대한 부가적인 정보를 제공하기에, **ISR**(Interrupt Service Routine)내에서 사용될 것이다. 일반적으로 여러개의 두번째 level interrupt들은 OR가 되어서, 첫번째 level의 interrupt의 bit를 설정하는 일을 할 것이다. 또한 source device 내부의 인터럽트를 enable시키기 위해서는, 적어도 이러한 device의 interrupt와 관련된 레지스터들을 접근해서 내용을 조작해 주어야 할 것이다.

대부분의 경우, interrupt의 근간이 되는 source는 ICIP와 ICFP 레지스터를 읽는 것으로 결정된다. 정확한 인터럽트의 원인(source)를 파악하기 위해서는 디바이스 내의 레지스터를 본다. SA1100이 idle mode에 있다면, 모든 enabled된 interrupt는 SA1100이 연산을 다시 수행(resume)하도록 만드는 역할을 해줄 것이며, ICMR은 idle mode에서는 무시된다.

SA1100의 interrupt controller는 4개의 레지스터를 가지고 있다. 각각은 ICIP(Interrupt Controller IRQ Pending Register), ICFP(Interrupt Controller FIQ Pending Register), ICMR(Interrupt Controller Mask Register), ICLR(Interrupt Controller Level Register)이다.

Controller Level Register)이다. Reset이 발생한 후에, FIQ와 IRQ 인터럽트는 CPU내에서는 disable이 되며, 다른 interrupt controller 레지스터들은 unknown상태가 된다. 따라서, 반드시 CPU내의 interrupt를 enable시키기 전에 소프트웨어적으로 초기화되어야 한다. 이젠 각각의 interrupt controller 레지스터들에 대해서 알아보기로 하자.

17.4.1. Interrupt Controller Pending Register(ICPR)

ICPR 레지스터는 32 bit의 read-only이며, 현재 시스템에서 활성화된(active) 인터럽트들을 보여준다. 또한 이 레지스터에 들어있는 bit들은 ICMR 레지스터의 상태(즉, mask 상태)에 영향을 받지 않는다. 다음의 표는 ICPR의 각 bit위치에 할당된 인터럽트의 source에 대한 것을 보여준다. 이 표는 인터럽트에 대한 source가 무엇인지와 각각에 관련된 second-level 인터럽트로는 몇가지가 있는지를 요약하고 있다.

Bit Position	Unit	Source Module	# of Level 2 Sources	Bit Field Description
31	System	Real-time clock	1	RTC equals alarm register.
30			1	One Hz clock TIC occurred.
29		Operating system timer	1	OS timer equals match register 3.
28			1	OS timer equals match register 2.
27			1	OS timer equals match register 1.
26			1	OS timer equals match register 0.
25	Peripheral	DMA controller	3	Channel 5 service request.
24			3	Channel 4 service request.
23			3	Channel 3 service request.
22			3	Channel 2 service request.
21			3	Channel 1 service request.
20			3	Channel 0 service request.
19		Serial port 4b	3	SSP service request.
18		Serial port 4a	8	MCP service request.
17		Serial port 3	6	UART service request.
16		Serial port 2	6 + 6	UART/HSSP service request.
15		Serial port 1b	6	UART service request.
14		Serial port 1a	5	SDLC service request.
13		Serial port 0	6	UDC service request.
12		LCD controller	12	LCD controller service request.
11	System	General-purpose I/O	17	“OR” of GPIO edge detects 27-11
10			1	GPIO 10 edge detect.
9			1	GPIO 9 edge detect.
8			1	GPIO 8 edge detect.
7			1	GPIO 7 edge detect.
6			1	GPIO 6 edge detect.
5			1	GPIO 5 edge detect.
4			1	GPIO 4 edge detect.
3			1	GPIO 3 edge detect.
2			1	GPIO 2 edge detect.
1			1	GPIO 1 edge detect.
0			1	GPIO 0 edge detect.

표 107. ICPR 레지스터의 bit정의

위의 표에서 보듯이 여러개의 unit이 하나의 인터럽트 signal에 대해서 하나 이상의 interrupt source를 가질 수 있다. 따라서, 만약 인터럽트 signal을 이와같은 unit들로 부터 받았다면, ISR(Interrupt Service Routine)은 interrupt controller의 flag register를 이용해서 어느 인터럽트가 발생했는지를 정확히 알아야

하며³⁶⁷, 그리고나서 interrupt를 발생시킨 unit의 status register를 읽어서 어떤 원인으로 interrupt가 발생했는지를 파악하고 서비스 한다. 하지만, 하나의 interrupt source만을 가지는 모든 인터럽트들에 대해서는 인터럽트 핸들러(ISR)는 인터럽트의 원인을 알기 위해서 interrupt controller의 레지스터 만을 사용하면 될 것이다.

Linux는 ICPR레지스터의 각 bit에 대한 것을 ~/include/asm-arm/arch-sa1100/irqs.h에 아래와 같이 정의하고 있다.

#define SA1100_IRQ(x)	(0 + (x))
#define IRQ_GPIO0	SA1100_IRQ(0)
#define IRQ_GPIO1	SA1100_IRQ(1)
#define IRQ_GPIO2	SA1100_IRQ(2)
#define IRQ_GPIO3	SA1100_IRQ(3)
#define IRQ_GPIO4	SA1100_IRQ(4)
#define IRQ_GPIO5	SA1100_IRQ(5)
#define IRQ_GPIO6	SA1100_IRQ(6)
#define IRQ_GPIO7	SA1100_IRQ(7)
#define IRQ_GPIO8	SA1100_IRQ(8)
#define IRQ_GPIO9	SA1100_IRQ(9)
#define IRQ_GPIO10	SA1100_IRQ(10)
#define IRQ_GPIO11_27	SA1100_IRQ(11)
#define IRQ_LCD	SA1100_IRQ(12) /* LCD controller */
#define IRQ_Ser0UDC	SA1100_IRQ(13) /* Ser. port 0 UDC */
#define IRQ_Ser1SDLC	SA1100_IRQ(14) /* Ser. port 1 SDLC */
#define IRQ_Ser1UART	SA1100_IRQ(15) /* Ser. port 1 UART */
#define IRQ_Ser2ICP	SA1100_IRQ(16) /* Ser. port 2 ICP */
#define IRQ_Ser3UART	SA1100_IRQ(17) /* Ser. port 3 UART */
#define IRQ_Ser4MCP	SA1100_IRQ(18) /* Ser. port 4 MCP */
#define IRQ_Ser4SSP	SA1100_IRQ(19) /* Ser. port 4 SSP */
#define IRQ_DMA0	SA1100_IRQ(20) /* DMA controller channel 0 */
#define IRQ_DMA1	SA1100_IRQ(21) /* DMA controller channel 1 */
#define IRQ_DMA2	SA1100_IRQ(22) /* DMA controller channel 2 */
#define IRQ_DMA3	SA1100_IRQ(23) /* DMA controller channel 3 */
#define IRQ_DMA4	SA1100_IRQ(24) /* DMA controller channel 4 */
#define IRQ_DMA5	SA1100_IRQ(25) /* DMA controller channel 5 */
#define IRQ_OST0	SA1100_IRQ(26) /* OS Timer match 0 */
#define IRQ_OST1	SA1100_IRQ(27) /* OS Timer match 1 */
#define IRQ_OST2	SA1100_IRQ(28) /* OS Timer match 2 */
#define IRQ_OST3	SA1100_IRQ(29) /* OS Timer match 3 */
#define IRQ_RTC1Hz	SA1100_IRQ(30) /* RTC 1 Hz clock */
#define IRQ_RTCAlrm	SA1100_IRQ(31) /* RTC Alarm */

코드 834. Linux의 ICPR bit에 대한 정의

여기서, 우리가 관심을 가지고 볼 부분은 GPIO에 대한 것이며, 이는 나중에 다시 Booting에 다루게 될 것이다. 현재로서는 이와 같은 정의를 가진다는 사실만을 기억하기 바란다. 또한 주의해서 볼 것은 GPIO 11에서 27까지는 하나의 인터럽트를 공유해서 사용한다는 것이다(IRQ_GPIO11_27). 예를 들어, IRQ_GPIO11_27이 발생했다면, 인터럽트 핸들러는 인터럽트를 공유하는 모든 ISR들을 호출해서 해당 인터럽트가 발생했는지를 각각의 디바이스의 레지스터를 직접 접근해서 알아야 한다.

³⁶⁷ 이것은 interrupt를 발생시킨 unit을 결정하는 것이다. 정확한 interrupt의 source는 각 unit의 status register를 읽어서 결정해야 한다.

17.4.2. Interrupt Controller IRQ Pending Register(ICIP) and FIQ Pending Register(ICFP)

ICIP와 ICFP는 각각이 전체 32개의 인터럽트당 하나씩의 flag bit을 두고 있으며, 이 bit들은 인터럽트 요구가 각 unit에서 만들어 졌는지를 나타낸다. ISR내에서는 ICIP와 ICFP를 읽어서 인터럽트 source가 무엇인지를 파악하게 된다. 일반적으로 ISR은 다시 interrupt를 일으킨 디바이스내의 레지스터를 읽어서 어떻게 인터럽트를 service할지를 결정할 것이다.

ICPR의 레지스터의 bit들은 read-only이며, interrupt source의 unit내에서 주어진 interrupt status 레지스터의 논리적으로 OR가 된 것이다. 일단 인터럽트가 service가 되고나면, ISR은 source unit의 status 레지스터에 처리했다고 표시하게 되며, 이것은 pending된 인터럽트를 지우는 효과가 있다. Source의 인터럽트 status bit을 지우는(clear) 것은, 만약 interrupt source unit내에 다른 interrupt status bit이 설정되지 않았다면, 자동적으로 해당하는 ICIP와 ICFP의 flag를 지우게 될 것이다. 이것은 해당 interrupt source status bit에 1을 것으로 clear시킨다. 0을 쓰는 것은 아무런 효과가 없을 것이다. 다음은 ICIP레지스터와 ICFP레지스터의 bit 정의를 보여주는 그림이다.

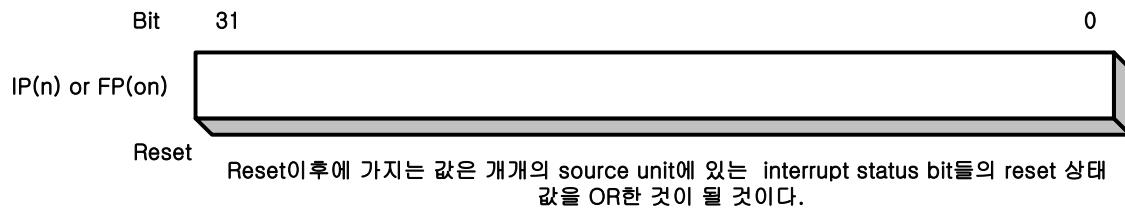


그림 116. ICIP or ICFP

Reset 이후에는 각 source unit의 interrupt status에 대한 bit들의 reset후에 가지는 값을 ICIP와 ICFP 레지스터가 가지게 될 것이다.

17.4.3. Interrupt Controller Mask Register(ICMR)

ICMR 레지스터는 32개의 pending interrupt들에 대해서 각각 하나의 씩의 mask bit를 가지고 있다. 이 mask bit은 pending interrupt bit이 CPU(Processor)의 인터럽트를 만들어 낼 것인지를 제어하는 일을 한다. 이렇게 제어되는 인터럽트로는 IRQ(normal)과 FIQ(fast IRQ)가 있다. 만약 pending IRQ가 active가 되면, 해당하는 ICMR 레지스터의 bit이 1로 설정된 경우에만 CPU로 인터럽트가 전달된다. 여기서 주의할 점은 SA1100이 idle mode에 있다면, mask bit들은 무시된다는 점이다. Idle mode에서는 인터럽트가 발생하면, 해당하는 pending bit이 설정되며, mask bit의 상태와 관계없이 인터럽트는 자동적으로 active된다³⁶⁸.

Mask bit은 크게 두가지 목적으로 사용한다. 즉, 첫번째가 실제로 인터럽트를 유발하는 것을 막으면서, 주기적으로 소프트웨어가 인터럽트 source를 polling하는 것을 허가하는 것이며, 두번째가 인터럽트 핸들러 routine이 이미 발생한 pending 인터럽트의 리스트를 관리하는 동안에 우선 순위가 낮은 인터럽트가 발생하는 것을 막기 위한 것이다. 즉, 인터럽트를 처리하는 동안에 발생할 수 있는 인터럽트를 막는 것이다. ICMR은 reset 후에 초기화가 되지 않으므로, 이를 초기화 하고 사용하는 것도 모두 소프트웨어의 몫이다. 다음의 그림은 ICMR 레지스터의 format과 bit정의를 보여준다.

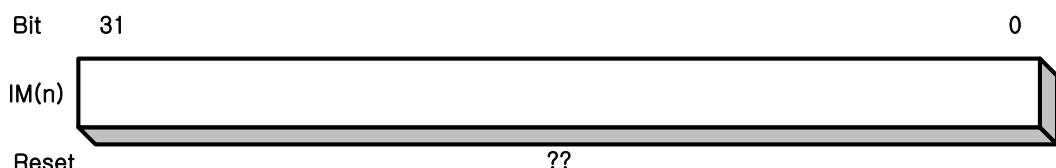


그림 117. ICMR

Bit	Name	Description
n	IM(n)	Interrupt Mask n (where n = 0 through 31).

³⁶⁸ 나중에 보게 될 ICCR을 참조하기 바란다.

		0 - Pending interrupt는 mask되어 active되지 않는다. 즉, CPU나 Power Manager로 인터럽트가 전달되지 않는다. 1 - Pending interrupt는 active될 수 있다. 즉, CPU나 Power Manager로 인터럽트가 전달 될 수 있다. 주의: IM bit은 idle mode에서는 무시된다.
--	--	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

표 108. ICMR 레지스터의 bit 정의

17.4.4. Interrupt Controller Level Register(ICLR)

ICLR 레지스터는 pending interrupt가 CPU의 FIQ나 IRQ 인터럽트를 발생시킬지를 제어한다(control). 만약 pending interrupt가 mask되지 않았다면(unmask), 해당 ICLR의 bit 필드가 어떤 CPU interrupt가 요청(assert)되어야 하는지를 선택하기 위해서 decoding된다. 만약 인터럽트가 mask되었다면, 해당하는 ICLR 레지스터의 bit은 아무런 의미가 없다.

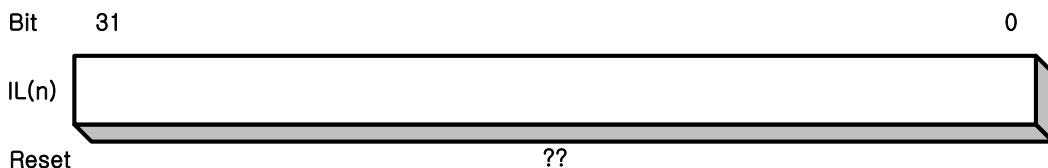


그림 118. ICLR

Bit	Name	Description
n	IL(n)	Interrupt Level n (where n = 0 through 31). 0 - 인터럽트는 CPU의 IRQ 인터럽트의 input으로 들어간다. 1 - 인터럽트는 CPU의 FIQ 인터럽트의 input으로 들어간다.

표 109. ICLR 레지스터의 bit 정의

ICLR 레지스터는 reset에서 값이 정해지지 않으므로 unknown이 될 것이다.

17.4.5. Interrupt Controller Control Register(ICCR)

이 레지스터는 interrupt controller의 레지스터에는 속하지 않으며, 단지 SA1100을 idle mode에서 복귀하는 역할을 수행하기 위한 한 bit(DIM: Disable Idle Mask)을 가지고 있다. 만약 이 bit이 1로 설정되어 있다면, SA1100을 idle mode에서 복귀시킬 수 있는 모든 interrupt는 ICMR의 내용(contents)에 의해서 정의된다. 만약 이 bit이 0으로 설정된다면, 모든 enable된 interrupt는 SA1100을 idle mode에서 복귀 시키게 될 것이다. ICCR 레지스터의 포맷(format)은 아래의 그림과 같다.

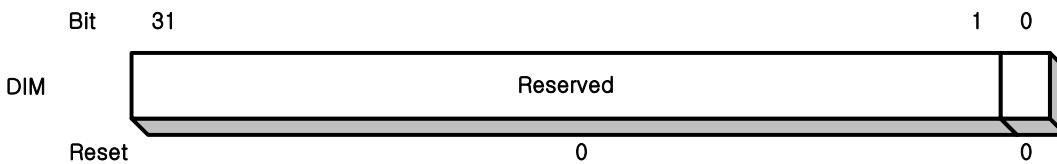


그림 119. ICCR

ICCR 레지스터는 reset이후에 가지는 값이 모든 bit에 대해서 0이다. 아래의 표는 각 bit이 하는 역할을 설명하고 있다.

Bit	Name	Description
0	DIM	Disable Idle Mask 0 - 모든 enable된 interrupt는 SA1100을 idle mode에서 복귀시킬 것이다. 1 - 단지 enable되고 mask되지 않은(즉, ICMR에 의해서 정의된)

		interrupt만이 SA1100을 idle mode에서 복귀시켜줄 것이다. 이 bit은 reset이후에 0으로 지워진다.
1..31	--	예약된 부분이다.

표 110. ICCR 레지스터의 bit 정의

이전 위에서 설명한 레지스터들 간의 관계를 좀더 명확히 하기 위해서 다음에 나오는 그림을 보도록 하자.

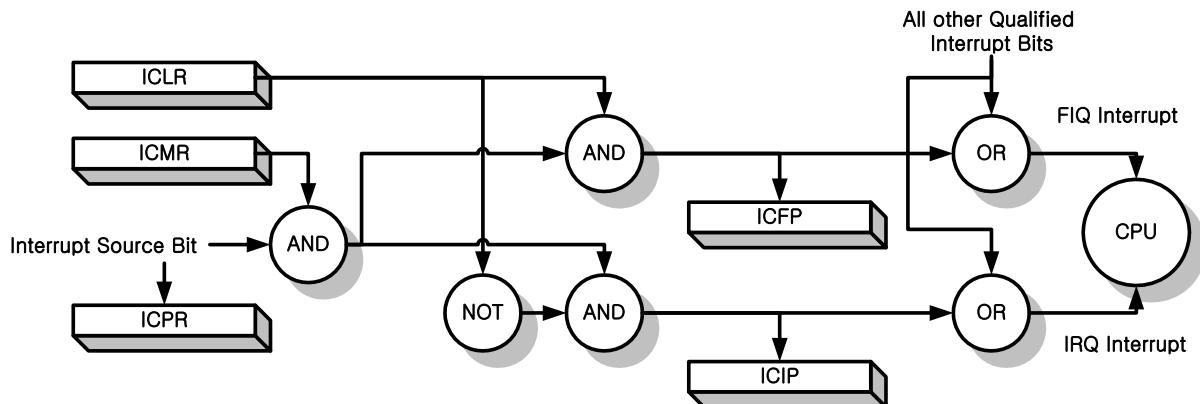


그림 120. Interrupt Controller의 블록 diagram

즉, Interrupt Source bit0이 들어와서 이 값이 ICPR을 설정하게 되며, ICMR과 AND된 후, 다시 ICLR과 AND되어 ICFP를 설정하던가, 아니면, ICMR의 NOT값과 AND가 되어 ICIP를 설정한다. 다시 이렇게 생성된 값은 다른 공인된(qualified) 인터럽트 bit과 OR가 되어서 CPU로 전달되며, CPU는 다시 ISR을 수행해서 인터럽트를 처리하게 될 것이다.

17.4.6. Interrupt Controller Register Location

다음의 표는 interrupt controller 레지스터의 블록이 위치하는 물리적인 주소를 알려주는 것이다. 이 주소들에 대한 연산(read나 write)는 해당 레지스터의 내용을 바꿀 것이며, 시스템에 발생하는 인터럽트에 영향을 줄 것이다.

Location	Name	Description
0x9005 0000	ICIP	Interrupt Controller IRQ Pending Register
0x9005 0004	ICMR	Interrupt Controller Mask Register
0x9005 0008	ICLR	Interrupt Controller Level Register
0x9005 0010	ICFP	Interrupt Controller FIQ Pending Register
0x9005 0020	ICPR	Interrupt Controller Pending Register
0x9005 000C	ICCR	Interrupt Controller Control Register

표 111. Interrupt Controller Register의 위치 정의

이와 같이 정의된 Interrupt Controller Register들은 Linux에서는 다음과 같이 정의하고 있다. 정의는 ~/include/asm-arm/arch-sa1100/SA-1100.h에 있다.

```

#define _ICIP          0x90050000 /* IC IRQ Pending reg. */
#define _ICMR         0x90050004 /* IC Mask Reg. */
#define _ICLR         0x90050008 /* IC Level Reg. */
#define _ICCR         0x9005000C /* IC Control Reg. */
#define _ICFP         0x90050010 /* IC FIQ Pending reg. */
#define _ICPR         0x90050020 /* IC Pending Reg. */

```

```
#if LANGUAGE == C
#define ICIP      (*((volatile Word *) io_p2v (_ICIP)))
#define ICMR     (*((volatile Word *) io_p2v (_ICMR)))
#define ICLR     (*((volatile Word *) io_p2v (_ICLR)))
#define ICCR     (*((volatile Word *) io_p2v (_ICCR)))
#define ICFP     (*((volatile Word *) io_p2v (_ICFP)))
#define ICPR     (*((volatile Word *) io_p2v (_ICPR)))
#endif /* LANGUAGE == C */
```

코드 835. Linux의 Interrupt Controller Register의 위치 정의

나중에 이렇게 정의된 레지스터들에 대한 접근이 있을 경우에 우리는 단지 “ICIP = XXX”와 같은 형태로 값을 넣어주게 될 것이며, 또한 내용을 읽어오기 위해서는 “XXX = ICIP”와 같은 형태가 될 것이다.

17.5. SA1100의 Memory Map

Embedded System을 디자인하고 운영체제를 올리는데 있어서 가장 중요한 아마 메모리 맵(memory map)일 것이다. 즉, 마이크로 프로세서와 주변기기들이 어떻게 연결되어 있으며, 어떤 주소 공간을 사용하는가 일 것이다. 이것은 입출력에 필요한 모든 것에 관련되기에 반드시 잘 읽혀야 한다.

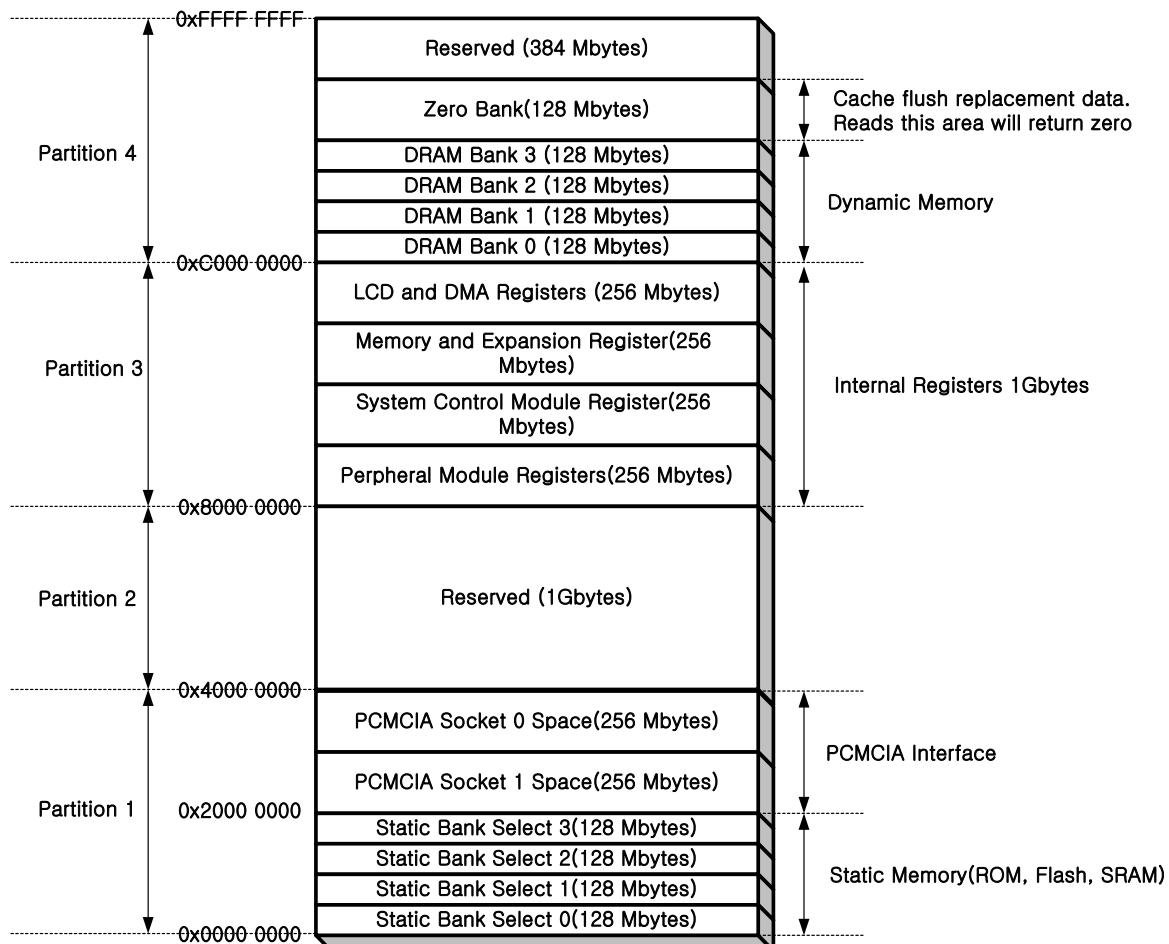


그림 121. SA1100의 메모리 맵

[그림 121]은 SA1100의 메모리 맵을 나타낸다. 만약 다른 보드를 디자인 했다면, 여기서 보여주는 그림은 달라질 수 있을 것이다. 가령 예를 들어서 PCMCIA를 사용하지 않는다면, 이 영역은 다른

목적으로 메모리를 맵을 설정해 줄 수 있을 것이다. 이곳에서는 단지 SA1100에 대한 것만을 이야기하고 있기에, SA1100에 한정된 메모리 맵의 정의만을 보여준 것이다.

SA1100의 메모리 맵은 크게 4개의 partition³⁶⁹으로 나누어진다. 각각은 1Gbytes씩을 차지하며, 하위의 partition은 ROM, SRAM, Flash와 같은 static memory device를 위한 공간과 PCMCIA expansion bus area를 위한 공간이다. 이 영역은 0x00000000에서 0x3FFFFFFF를 차지한다. 이 영역을 다시 4개의 128Mbytes 블록으로 나누어서 static memory device들을 위한 공간으로 주고, 두개의 512 Mbytes 블록을 PCMCIA를 위한 공간으로 사용하고 있다. Static memory 공간은 ROM, SRAM, Flash 메모리를 위한 공간으로, 하위에는 boot시에 ROM이 있다고 가정하며, boot ROM의 크기는 ROMSEL CPU pin의 상태로 결정된다. PCMCIA공간은 다시 Socket³⁷⁰ 0와 Socket 1공간으로 나누어지며, 다시 이 공간은 I/O, memory, attribute 공간으로 구분된다.

0x40000000에서 0x7FFFFFFF까지의 메모리 맵은 예약된 부분이며, 이러한 공간에 대한 접근(access)는 data abort exception을 발생시키게 될 것이다. 0x80000000에서 0xBFFFFFFF까지의 공간은 on-chip register들을 위한 것으로 ARM V4 architecture에서 명시된 레지스터들을 제외한 레지스터들에 대한 공간이다. 이 블록은 다시 4개의 256 Mbytes로 각각이 나누어지며, 프로세서내의 주요 functional block들에 대한 control register들을 포함한다. 예를 들어서 MECM, SCM, PCM등이 있을 수 있겠다. LCD와 DMA controller들은 PCM의 나머지들과는 분리된 공간을 가지며, 상위의 256 Mbytes를 차지한다.

마지막 4번째 partition은 0xC0000000에서 0xFFFFFFFF까지를 차지하며, DRAM 메모리를 포함한다. Bank의 크기는 각각 128 Mbytes로 고정되어 있으며, 여러 bank들이 구현되어 있는 관계로, 메모리 관리 unit(MMU: Memory Management Unit)을 통해서 mapping되어야 하는 맵에는 빈 공간(gap)이 있을 수 있다. DRAM bank 바로 위에 있는 128 Mbytes의 블록은 메모리 controller내에 mapping되며, 항상 read연산에서 0을 돌려주도록 되어있다. 이것은 cache flushing을 빠르게 하기 위한 것으로, 외부 메모리에서 cache로 데이터를 읽어드리는 수고를 덜어준다. 또한 이 영역에 대한 write는 아무 효력이 없다. 마지막 상위의 384 Mbytes의 영역은 예약된 부분으로 이 영역에 대한 접근은 data abort exception을 유발할 것이다.

앞에서 본 영역들에 대해서 Linux가 어떻게 정의하는지를 보고자 한다면, ~/include/asm-arm/arch-sa1100/SA-1100.h를 참고하기 바란다. 우리는 이 파일에서 정의된 것 중에서 DRAM에 대한 것만 예로서 보도록 하자.

...			
#define MemBnkSp	0x08000000	/* Memory Bank Space [byte]	*/
#define StMemBnkSp	MemBnkSp	/* Static Memory Bank Space [byte] */	
#define StMemBnk0Sp	StMemBnkSp	/* Static Memory Bank 0 Space */	
		/* [byte]	*/
#define StMemBnk1Sp	StMemBnkSp	/* Static Memory Bank 1 Space */	
		/* [byte]	*/
#define StMemBnk2Sp	StMemBnkSp	/* Static Memory Bank 2 Space */	
		/* [byte]	*/
#define StMemBnk3Sp	StMemBnkSp	/* Static Memory Bank 3 Space */	
		/* [byte]	*/
#define DRAMBnkSp	MemBnkSp	/* DRAM Bank Space [byte]	*/
#define DRAMBnk0Sp	DRAMBnkSp	/* DRAM Bank 0 Space [byte]	*/
#define DRAMBnk1Sp	DRAMBnkSp	/* DRAM Bank 1 Space [byte]	*/
#define DRAMBnk2Sp	DRAMBnkSp	/* DRAM Bank 2 Space [byte]	*/
#define DRAMBnk3Sp	DRAMBnkSp	/* DRAM Bank 3 Space [byte]	*/
#define ZeroMemSp	MemBnkSp	/* Zero Memory bank Space [byte]	*/
#define _StMemBnk(Nb)		/* Static Memory Bank [0..3]	*/\

³⁶⁹ 여기서 말하는 partition은 메모리에 한정된 것이다. 다른 것과 혼동하지 않도록 하자.

³⁷⁰ 물론 여기서 말하는 Socket도 network device에서 사용하는 socket interface와 혼동하지 않도록 해야 할 것이다.

```

(0x00000000 + (Nb)*StMemBnkSp)           /* Static RAM의 시작 주소 */
#define _StMemBnk0   _StMemBnk (0)    /* Static Memory Bank 0      */
#define _StMemBnk1   _StMemBnk (1)    /* Static Memory Bank 1      */
#define _StMemBnk2   _StMemBnk (2)    /* Static Memory Bank 2      */
#define _StMemBnk3   _StMemBnk (3)    /* Static Memory Bank 3      */

#if LANGUAGE == C
typedef Quad     StMemBnkType [StMemBnkSp/sizeof (Quad)];
#define StMemBnk          /* Static Memory Bank [0..3] */ \ 
((StMemBnkType *) io_p2v (_StMemBnk (0)))
#define StMemBnk0  (StMemBnk [0])  /* Static Memory Bank 0      */
#define StMemBnk1  (StMemBnk [1])  /* Static Memory Bank 1      */
#define StMemBnk2  (StMemBnk [2])  /* Static Memory Bank 2      */
#define StMemBnk3  (StMemBnk [3])  /* Static Memory Bank 3      */
#endif /* LANGUAGE == C */

#define _DRAMBnk(Nb)          /* DRAM Bank [0..3] */ \ 
(0xC0000000 + (Nb)*DRAMBnkSp)           /* DRAM의 시작 주소 */
#define _DRAMBnk0   _DRAMBnk (0)    /* DRAM Bank 0      */
#define _DRAMBnk1   _DRAMBnk (1)    /* DRAM Bank 1      */
#define _DRAMBnk2   _DRAMBnk (2)    /* DRAM Bank 2      */
#define _DRAMBnk3   _DRAMBnk (3)    /* DRAM Bank 3      */

#if LANGUAGE == C
typedef Quad     DRAMBnkType [DRAMBnkSp/sizeof (Quad)];
#define DRAMBnk          /* DRAM Bank [0..3] */ \ 
((DRAMBnkType *) io_p2v (_DRAMBnk (0)))
#define DRAMBnk0  (DRAMBnk [0])  /* DRAM Bank 0      */
#define DRAMBnk1  (DRAMBnk [1])  /* DRAM Bank 1      */
#define DRAMBnk2  (DRAMBnk [2])  /* DRAM Bank 2      */
#define DRAMBnk3  (DRAMBnk [3])  /* DRAM Bank 3      */
#endif /* LANGUAGE == C */
/* Zero Bank의 시작 주소 */
#define _ZeroMem 0xE0000000      /* Zero Memory bank */
...

```

코드 836. Static Memory와 DRAM Memory에 대한 맵 정의

코드에서 보듯이 Static Memory는 0x00000000에서 시작해서 4개의 128Mbytes 블록으로 이루어져 있으며, DRAM Memory는 0xC0000000에서 시작해서 4개의 128 Mbytes블록으로 이루어져 있다. 마지막으로 Zero Memory는 0xE0000000에서 시작함을 또한 볼 수 있을 것이다.

이전 대략적인 SA1100의 구성에 대해서 알았을 것이다. 물론 이곳에서 다루지 않은 부분 들에 대해서는 Intel의 SA1100 메뉴얼을 읽어서 익혀야 할 것이다. 하지만, 이 정도 충분하다고 생각한다. 나중에 Linux의 실제 코드를 보면서 관련된 사항들을 차례로 보기로 하겠다. 자, 그럼 Linux를 SA1100으로 porting하는 과정의 첫번째로 Booting하는 과정에 대해서 알아보기로 하자.

17.6. Boot Loader(BLOB)

Boot loader가 하는 일을 보드의 초기화와 저장 매체로부터(혹은 serial이나 ethernet을 통해서) 커널을 읽어서 메모리의 적절한 위치로 적재(load)한 다음, 커널로 제어를 옮기는 역할을 수행한다. 또한 개발중인 커널의 경우에는 네트워크를 통한 커널 image의 다운로드도 제공하며, 현재로서는 일반적으로 serial을 통한 커널의 다운로딩을 제공하는 것들이 많다. 각각의 CPU architecture마다 boot loader가 각기 존재하기 때문에, 커널의 개발자는 자신이 사용하는 CPU의 architecture가 어떤 것인가에 따라서, 선택할 수 있는 boot loader의 종류를 알 수 있을 것이다. 이것도 못마땅하다면, 직접 개발해서 사용할 수도 있지만, 개발기간이 좀더 추가되어야 할 것이다.

우리가 이곳에서 보게될 boot loader로는 **blob**이며, 각각의 target마다 적절한 boot loader를 선택해 주거나, 혹은 스스로 개발해서 사용해야 할 것이다. ARM에서 사용하는 boot loader의 대표적인 예로 blob를 택했으며, 관련된 site로는 <http://www.lart.tudelft.nl/lartware/blob>³⁷¹를 보기 바란다. 이하에서는 blob를 기본 boot loader로 사용해서 설명하는 내용이다.

먼저, 본격적으로 source 코드를 보면서 진행하기에 앞서 전체적인 BLOB의 진행 절차에 대해서 간단히 알아보도록 하자.

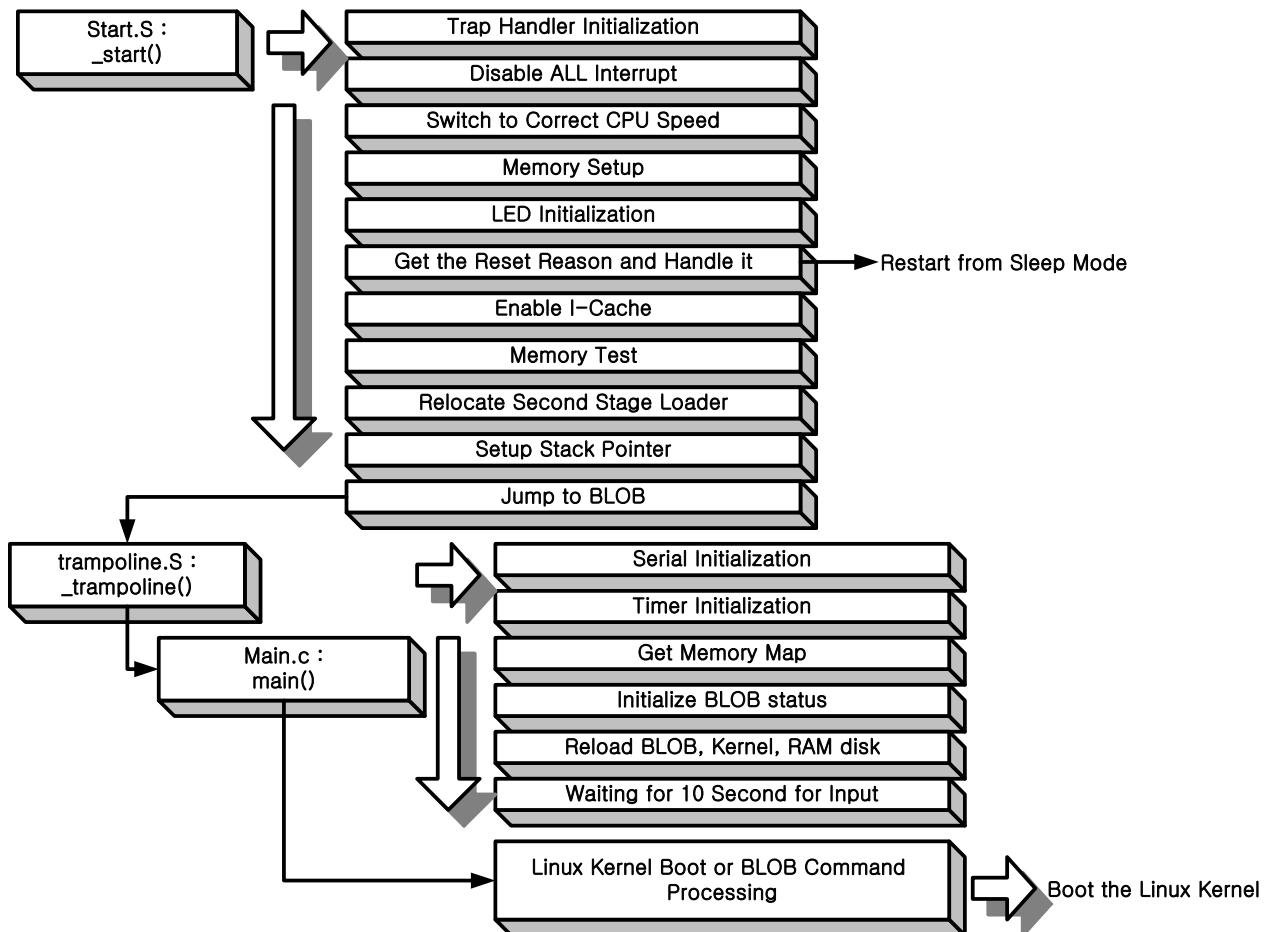


그림 122. BLOB의 실행 순서

[그림 122]은 BLOB의 실행 순서 별로 추적하면서 정리한 것이다. 큰 분류별로 본 것으로 최종적으로 Linux Kernel을 booting하는 것을 목적으로 한다. 먼저 시작하는 곳은 start.S파일에서 시작하게 되고, trampoline.S를 통해서 BLOB의 main() 함수가 정의된 main.c를 실행하게 된다.

이전 이러한 구조를 가진 BLOB를 앞에서 보여준 순서에 의해서 차례로 설명을 진행하도록 하겠다. 그리고, BLOB의 설명이 끝나면, SA1100 Linux Kernel의 booting으로 넘어가서 실제 Linux 커널의 boot sequence를 추적해 보겠다.

³⁷¹ 이 site에서는 ARM toolchain 및 그것들에 대한 patch와 Linux 커널에 대한 patch도 찾을 수 있을 것이다. Cross compiler를 직접 compile하기 싫은 사람들을 위한 binary 파일도 제공하므로, 그냥 가져다가 쓰면 될 것이다. 이 글을 쓰고 있는 현재 BLOB은 2.0.3 버전을 찾을 수 있었기에, 이하의 코드는 이것을 기준으로 하겠다.

17.6.1. start.S의 분석

먼저, blob의 source tree를 보면, tools와 src, include를 디렉토리를 찾을 수 있을 것이다. 각각은 사용하는 tool 및 utility에 대한 디렉토리와, blob의 source, 그리고, head 파일을 가지고 있다. 여기서는 compile과 설치는 설명하지 않기로하고, 일단 먼저 start-ld-script라는 파일을 보도록 하자. 이것은 LD(Loader & Linker)의 input으로 주어져서, object 파일을 생성하는데 규칙을 제공할 것이다. 아래와 같은 부분이 있을 것이다.

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;
    . = ALIGN(4);
    .text : { *(.text) }
    . = ALIGN(4);
    .rodata : { *(.rodata) }
    . = ALIGN(4);
    .data : { *(.data) }
    . = ALIGN(4);
    .got : { *(.got) }
    . = ALIGN(4);
    .bss : { *(.bss) }
}
```

코드 837. start-ld-script 파일

파일에서 OUTPUT_FORMAT은 ELF32의 little endian으로 코드를 생성하겠다는 말이며, OUTPUT_ARCH는 0이 binary를 실행할 수 있는 CPU architecture로 ARM을 주겠다는 뜻이다. Entry point가 되는 함수는 _start이다. 또한 SECTIONS의 정의를 보면, text, rodata, data, got, bss라는 section들이 정의되어 있음을 볼 수 있을 것이다. 이중에서 실행할 코드가 들어가는 부분은 text이다. “.”은 현재의 address를 가르키는 일종의 포인터로 생각하면 될 것이다. 따라서, 코드는 0x00000000에서 시작해서 4 byte 단위로 정렬된 text section이 제일 먼저 나올 것이다. 이전 blob가 어디서부터 실행될지를 알게되었다(_start). _start는 start.S에 다시 아래와 같이 정의되어 있으며, 이곳에서부터 설명을 진행하도록 하겠다.

```
.text
/* Jump vector table as in table 3.1 in [1] */
.globl _start
_start:   b      reset
          b      undefined_instruction
          b      software_interrupt
          b      prefetch_abort
          b      data_abort
          b      not_used
          b      irq
          b      fiq
```

코드 838. start.S 파일

.text로 이곳이 text section에 속함을 알 수 있다. 또한 _start가 .global로 정의되어 export된 symbol임을 짐작할 수 있으며, 앞에서 _start가 entry point가 된다는 것을 알았다. _start의 첫부분은 reset으로 branch하는 연산이다. 따라서, reset으로 제어가 옮겨질 것이다. 나머지는 ARM에서 발생할 수 있는 exception 상황에 대한 처리 routine들로 되어 있으며, 각각은 undefined instruction, software interrupt, prefetch abort, data abort, irq, fiq의 핸들러를 가르킨다.

```
/* the actual reset code */
reset:
    /* First, mask **ALL** interrupts */
    ldr      r0, IC_BASE
    mov      r1, #0x00
    str      r1, [r0, #ICMR]

    /* switch CPU to correct speed */
    ldr      r0, PWR_BASE
    LDR      r1, cpuspeed
    str      r1, [r0, #PPCR]
```

코드 839. start.S 파일(계속)

IC_BASE(=0x900050000)는 interrupt controller register의 기본(base) 주소를 가진다. 이 값을 r0에 가져오고, r1에는 0을 둔다. 다시 r0가 가르키는 값에 ICMR의 offset(=0x04)을 더해서, 이곳에 r1(=0)을 넣는다. 이것으로 모든 interrupt를 masking해서 interrupt가 발생하지 않도록 만든다.

PWR_BASE(=0x900200000)는 power manager register의 기본 기본 주소를 가진다. r0에 PWR_BASE를 가져오고, r1에는 cpuspeed 값을 가져온후 r0에 PPCR(=0x14)을 더한 주소에 r1을 넣는다. 이것으로 CPU의 clock speed를 조정해준다. 이때 사용되는 cpuspeed의 값은 아래와 같이 정의된다.

```
/* The initial CPU speed. Note that the SA11x0 CPUs can be safely overclocked:
 * 190 MHz CPUs are able to run at 221 MHz, 133 MHz CPUs can do 206 Mhz.
 */
#endif (defined ASSABET) || (defined CLART) || (defined LART) \
    || (defined NESA) || (defined NESA)
cpuspeed:.long 0x0b      /* 221 MHz */
#endif defined SHANNON
cpuspeed:.long 0x09      /* 191.7 MHz */
#else
#warning "FIXME: Include code to use the correct clock speed for your board"
cpuspeed:.long 0x05      /* safe 133 MHz speed */
#endif
```

코드 840. start.S 파일(계속)

즉, ASSABET과 CLART, LART, NESA가 정의된 cpuspeed가 0x0B를 사용하며, SHANNON이 정의된 경우에는 0x09를, 나머지는 0x05로 둔다. 이곳에서 정의된 상수값들은 실제로 PPL(Phase-Locked Loop) 설정하는 PPCR fpwltxjdm1 gkdn1 5 bit(CCF<0..5> : Core Clock Configuration Field)에 반영되는 값이다. 이 레지스터에 정의된 값은 reset후에는 남아있지 않기 때문에 이곳에서 새로 설정해 주게된다. 상수값이 의미하는 바는 CPU 메뉴얼을 찾아보기 바란다. 각각의 값에 따라서, 3.6864 MHz의 crystal oscillator를 설정하는 것으로, 실제 211.2 MHz, 191.7 MHz, 132.7 MHz라는 동작 속도를 정해준다.³⁷²

```
/* setup memory */
bl      memsetup

/* init LED */
bl      ledinit
```

코드 841. start.S 파일(계속)

앞에서 이미 interrupt를 masking했고, CPU의 동작 speed를 결정해 주었다. 이젠 초기 booting에서 사용할 메모리를 설정하는 부분이다. BL(Branch with Link)을 사용해서 subroutine으로 memsetup()이라는 함수를

³⁷² 133MHz는 대략적인 속도를 나타낸 값이다.

호출한다. 이것을 마치고나면, 다시 ledinit()라는 함수를 호출해서 LED(Light Emitted Device)를 초기화한다. 각각의 함수는 memsetup.S와 ledasm.S에 있으며, 이 파일을 분석하고 계속 진행하도록 하겠다.

```
.globl memsetup
memsetup:
#endif defined USE_SA1100
    /* Setup the flash memory */
    ldr      r0, MEM_BASE
    ldr      r1, mcs0
    str      r1, [r0, #MCS0]
    /* Set up the DRAM */
    /* MDCAS0 */
    ldr      r1, mdcas0
    str      r1, [r0, #MDCAS0]
    /* MDCAS1 */
    ldr      r1, mdcas1
    str      r1, [r0, #MDCAS1]
    /* MDCAS2 */
    ldr      r1, mdcas2
    str      r1, [r0, #MDCAS2]
    /* MDCNFG */
    ldr      r1, mdcnfg
    str      r1, [r0, #MDCNFG]
```

코드 842. memsetup.S 파일

memsetup.S 파일에서 하는 일은 초기 메모리를 설정하는 일이다. 프로그램은 주석이 잘 처리가 되어 있는 관계로 보기 가 편할 것이다. 먼저 SA1100 board를 사용하고 있다면, #if defined USE_SA1100이하에서 실행이 시작될 것이다.

MEM_BASE(=0xA0000000)는 메모리 설정(memory configuration) 레지스터들의 기본(base)주소를 담고 있다. 이것을 r0로 가져온다. mcs0(=0xffff8fff8)³⁷³를 다시 r1에 가져오고, MCS0(=0x10)과 r0를 더한 값이 가르키는 곳에 r1을 넣도록 한다. MCS0은 static memory control register 0을 의미한다. 이 레지스터와 MCS1은 read/write가 가능한 레지스터들로서 static memory에 대한 설정을 제어한다. Reset후에 가질 수 있는 값은 가능한 가장 늦은 speed의 ROM에 맞춰지며, 각각의 레지스터는 동일한 두개 16 bit의 field를 가진다. 따라서, 전체로는 4개의 필드로 구성된다.

여기서 잠시 SA1100의 메모리와 관련된 register들을 조금 정리해보도록 하겠다. 먼저 메모리 configuration register가 있는 위치는 다음과 같다.

위치	심벌	이름
0xA000 0000	MDCNFG	DRAM Configuration Register
0xA000 0004	MDCAS0	DRAM CAS Waveform Shift Register 0
0xA000 0008	MDCAS1	DRAM CAS Waveform Shift Register 1
0xA000 000C	MDCAS2	DRAM CAS Waveform Shift Register 2
0xA000 0010	MSC0	Static Memory Control Register 0
0xA000 0014	MSC1	Static Memory Control Register 1
0xA000 0018	MECR	Expansion Bus Configuration Register

표 112. SA1100의 메모리 설정 레지스터

MDCNFG 레지스터는 32 bit read/write 레지스터로서 DRAM을 설정하는 control bit을 가진다. 모든 DRAM bank들은 같은 종류의 DRAM device들로 구성되어야 한다. MDCASX의 심벌을 가지는 레지스터는 32

³⁷³ SA1100을 사용한다면 Brutus가 정의된 곳을 사용할 것이다.

bit의 read/write가 가능하며, CAS(Column Address Selection)의 waveform을 결정해주는 것으로 DRAM을 접근하는데 필요한 CPU cycle을 나타내준다. MSCX의 심벌을 가지는 레지스터는 32 bit의 read/write가 가능하며, ROM이나 flash와 같은 Static 메모리에 대한 제어를 담당한다. MECR 레지스터는 expansion 메모리의 설정을 관리하며, PCMCIA와 같은 것을 위해서 사용되어 진다. 좀더 자세한 것을 알고자 한다면, SA1100의 개발자 메뉴얼을 참고하기 바란다. 현재로서는 이와 같은 레지스터에 특정한 값을 설정해서 BLOB을 load하거나 C 함수를 수행하기 위해서 reset 이후에 올바른 값으로 정해 주어야 한다는 것만 기억하기 바란다.

```

ldr      r1, MEM_START
.rept    8
        ldr      r0, [r1]
.endr
#endif defined USE_SA1110
...
# Note : 이 부분은 SA1110을 위한 메모리 설정을 다룬다. 거의 비슷한 과정을 거치는 것이므로 생략하도록 하겠다.
...
#else
#error "Configuration error: CPU not defined!" /* 만약 SA11x0을 사용하지 않는다면 에러로 처리한다. */
#endif
        mov      pc, lr           /* Sub routine을 복귀한다. (앞에서 BL로 jump했음)*/

```

코드 843. memsetup.S 파일

앞에서는 이미 메모리와 관련된 register들에 대한 설정을 마쳤다. r1에 MEM_START(=0xC0000000)의 값을 읽어오고, r0에 r1이 가르키고 있는 곳에서 값을 읽어 넣는 일을 8번 반복한다. 이것은 disable된 bank가 refresh하지 못하도록 막는 역할을 한다. 이젠 메모리에 대한 설정을 마쳤으므로 pc(program counter register)에 앞에서 branch시에 저장해 두었던 lr(link register)의 내용을 가져오도록 한다(mov). 이것으로 subroutine의 return이 이루어진다.

```

.globl ledinit
ledinit:
        ldr      r0, GPIO_BASE
        ldr      r1, LED
        str      r1, [r0, #GPDR]      /* LED GPIO is output */
        str      r1, [r0, #GCSR]      /* turn LED on */
        mov      pc, lr
.globl led_on
        /* turn LED on. clobbers r0 and r1 */
led_on:
        ldr      r0, GPIO_BASE
        ldr      r1, LED
        str      r1, [r0, #GCSR]
        mov      pc, lr
.globl led_off
        /* turn LED off. clobbers r0 and r1 */
led_off:
        ldr      r0, GPIO_BASE
        ldr      r1, LED
        str      r1, [r0, #GPCR]
        mov      pc, lr

```

코드 844. ledasm.S 파일

이젠, ledasm.S를 보기로 하자. 이곳에선 단순히 LED에 대한 초기화만을 담당한다. LED의 GPIO를 초기화하고, LED를 켜준다. 물론, 여기서 정의된 led_on()과 led_off() 함수는 LED를 켜고 끄는 역할을

하는 함수로 나중에 사용될 것이다. ledinit() 함수를 먼저 보도록 하자. r0에 GPIO_BASE(=0x90040000)를 넣고, r1에는 LED를 넣는다. LED값은 led.h 파일에 아래와 같은 정의를 가진다.

```
/* define the GPIO pin for the LED */
#ifndef ASSABET
#define LED_GPIO 0x00020000 /* GPIO 17 */
#elif (defined CLART) || (defined LART) || (defined NESA)
#define LED_GPIO 0x00800000 /* GPIO 23 */
#elif defined PLEB
#define LED_GPIO 0x00010000 /* GPIO 16 */
#else
#warning "FIXME: Include code to turn on one of the LEDs on your board"
#define LED_GPIO 0x00000000 /* safe mode: no GPIO, so no LED */
#endif
```

코드 845. led.h 파일

즉, ASSABET으로 정의된 경우에는 0x00020000(=GPIO17)을, LART나 CLART 및 NESA가 정의된 경우에는 0x00800000(=GPIO23)을, PLEB가 정의된 경우에는 0x00010000(=GPIO16)을 각각 사용한다. 정의된 값이 없다면, 0x00000000이 될 것이다. 우리가 사용하는 정의는 ASSABET인 경우이므로, LED_GPIO는 0x00020000이 될 것이다. 이 값으로 r1을 초기화하고, GPDR에 output으로 GPIO의 pin direction을 설정하게 되며, GPSR에 해당 bit를 ON시켜서 LED가 켜지도록 만든다. 나머지 LED를 켜는 함수와 끄는 함수는 GPSR과 GPCR 레지스터의 해당 bit를 ON/OFF시키는 동작만 하도록 되어 있다. 이제 메모리와 LED에 대한 설정을 끝냈으므로 다시 start.S로 돌아가서 계속 이야기를 진행하도록 하자.

```
/* check if this is a wake-up from sleep */
ldr    r0, RST_BASE
ldr    r1, [r0, #RCSR]
and    r1, r1, #0x0f
teq    r1, #0x08
bne    normal_boot      /* no, continue booting */

/* yes, a wake-up. clear RCSR by writing a 1 (see 9.6.2.1 from [1]) */
mov    r1, #0x08
str    r1, [r0, #RCSR]    ;

/* get the value from the PSPR and jump to it */
ldr    r0, PWR_BASE
ldr    r1, [r0, #PSPR]
mov    pc, r1
```

코드 846. start.S 파일(계속)

RST_BASE(=0x90030000)는 reset controller의 base주소를 가르킨다. SA1100의 reset controller는 다양한 reset을 관리할 목적으로 사용되며, 프로그래머의 관점에서 보면, 크게 두개의 register로 구성된다. 하나는 software reset을 시키는 일을 하며, 다른 하나는 프로세서의 reset 이유가 무엇인가를 나타내는 목적으로 사용된다. 각각은 RSRR(Reset Controller Software Reset Register)와 RCSR(Reset Controller Status Register)로 나뉜다.

먼저 RCSR의 레지스터를 읽어서 r1에 둔다. 이 값을 0x0F로 AND시켜서 하위 4bit만을 살펴보게되며, 이 값이 0x08과 같지 않다면, hardware reset이나, software reset, 혹은 watchdog reset이 들어간 상태임을 보여주게 된다. 나머지 하나는 sleep mode reset이다. 즉, sleep mode에서 깨어나게 되는 reset이 걸린다면, RCSR에 있는 sleep mode reset을 지우기 위해서 RCSR에 0x08을 넣어주며, PSPR(Power Manager Scratch Pad

Register)³⁷⁴를 읽어서 sleep mode 이전의 프로세서의 설정을 읽어 r1에 넣어준 후, 이 값으로 jump를 한다. 즉, sleep mode에서 깨어나서 계속 연산을 진행해 나가는 것이 된다.

만약 sleep mode에서 reset이 걸린 것이 아니라면, normal_boot으로 제어가 이동해서 진행될 것이다. 우리가 현재 다루고 있는 것은 hardware reset이후의 상황이므로 normal_boot이 될 것이다.

```
normal_boot:
    /* enable I-cache */
    mrc      p15, 0, r1, c1, c0, 0 @ read control reg
    orr      r1, r1, #0x1000          @ set Icache
    mcr      p15, 0, r1, c1, c0, 0 @ write it back

    /* check the first 1MB  in increments of 4k */
    mov      r7, #0x1000
    mov      r6, r7, lsl #8          /* 4k << 2^8 = 1MB */
    ldr      r5, MEM_START

mem_test_loop:
    mov      r0, r5
    bl       testram
    teq      r0, #1
    beq      badram

    add      r5, r5, r7
    subs     r6, r6, r7
    bne      mem_test_loop
```

코드 847. start.S 파일(계속)

이전 I-cache(Instruction Cache)를 enable시켜준다. 이것은 control register를 읽어서, I cache enable을 나타내는 bit을 설정한 후, 다시 이것으로 원래의 값을 대치시켜주면 된다. 이것을 마치고나면, 이전 첫 1MBytes 영역에 대한 test에 들어간다. 먼저 r7에 4K(=0x1000)을 넣고, 다시 이를 좌측으로 8 bit shift해서 r6에 넣는다. 이렇게 하고나면, r6는 1 M을 가질 것이다. r5에는 MEM_START(=0xC0000000)로 두어 메인 메모리의 시작 주소를 가르키도록 한다. 메모리 테스트가 일어나는 곳은 mem_test_loop이다. 이곳에서 loop를 돌면서 확인하게 된다. 확인하는 단위는 4KBytes로 1MBytes길이에 대해서 일어난다. 먼저, r0에 메모리의 시작 주소를 가져온다(r5). 테스트는 testram() 함수에서 일어나게되며(bl), 만약 테스트의 결과 값(r0)이 1이라면, badram으로 제어를 옮기게 되며, 그렇지 않다면, r5에 r7(=4K)를 더하고, r6에서는 r7을 빼서 다시 loop를 돌게된다.

```
.text
.globl testram
    @ r0 = address to test
    @ returns r0 = 0 - ram present, r0 = 1 - no ram
    @ clobbers r1 - r4

testram:
    ldmia  r0, {r1, r2}      @ store current value in r1 and r2
    mov    r3, #0x55 @ write 0x55 to first word
    mov    r4, #0xaa@ 0xaa to second
    stmia r0, {r3, r4}
    ldmia  r0, {r3, r4}      @ read it back
    teq    r3, #0x55 @ do the values match
    teqeq r4, #0xaa
    bne    bad              @ oops, no
    mov    r3, #0xaa@ write 0xaa to first word
    mov    r4, #0x55          @ 0x55 to second
```

³⁷⁴ PSPR에 들어있는 값은 프로세서의 설정을 나타내는 어떤 값이거나, ROM과 같은 곳에 있는 routine에 대한 pointer가 될 수 있을 것이다.

```

stmia    r0, {r3, r4}
ldmia    r0, {r3, r4}      @ read it back
teq     r3, #0xaa @ do the values match
teqeq   r4, #0x55
bad:    stmia    r0, {r1, r2} @ in any case, restore old data
        moveq    r0, #0          @ ok - all values matched
        movne    r0, #1          @ no ram at this location
        mov      pc, lr

```

코드 848. testmem.S 파일

실제 source code를 보면, testmem.S와 testmem2.S 두개의 파일이 있다. 각 파일 하는 일은 동일하며, 차이가 나는 것은 stack을 이용해서 변동이 있게 되는 register의 내용을 저장할 것인가, 아니면, 그냥 수행할 것인가만이 차이가 나며, 나머지는 동일하다. 여기서는 testmem.S를 보도록 하겠다. r0에는 test의 시작 주소가, 그리고, 나중에 test의 결과 같이 r0로 돌아가게 된다. 결과값이 0이라면, RAM이 존재한다는 것을 나타내고, 그렇지 않다면, RAM이 없음을 나타낸다. r1에서 r4까지는 이 subroutine에서 사용되는 register들이다³⁷⁵.

r0 가 가르키는 곳에 원래 저장된 두개의 32 bit 값을 r1과 r2로 읽어드린다(ldmia). r3에는 0x55를 r4에는 0xAA를 두고, 이 값을 r0가 가르키는 곳에 적는다(stmia). 다시 r0가 가르키는 곳에서 r3와 r4를 읽어서, 이 값들디 앞에서 적었던 값과 일치하는지 검사한다(teq, teqeq). 만약 같지 않다면, 쓴 값과 읽은 값이 다르므로 RAM이 존재하지 않는다는 말이 되므로, bad로 제어를 옮기게 될 것이다. r3에는 다시 0xAA를 넣고, r4에 0x55를 넣은 후, r0가 가르키는 곳에 다시 r3와 r4를 저장한다(stmia). 다시 같은 위치에서 r3와 r4를 가져와서(ldmia)이 값이 앞서 쓴 값과 같은지를 비교한다. r0에는 앞에서 저장해 두었던 r1과 r2를 다시 쓰게되고, r0에는 RAM이 존재하는지 그렇지 않은지에 따라서, 0이나 1값을 주게된다(moveq, movne). 마지막으로 (mov pc, lr)로 subroutine은 복귀한다. 따라서, 결과값은 r0에 들어갈 것이다.

```

badram:
    b      blinky
...
blinky:
/* This is test code to blink the LED
   very useful if nothing else works */
    bl      led_on
    bl      wait_loop
    bl      led_off
    bl      wait_loop
    b      blinky
wait_loop:
/* busy wait loop*/
    mov    r2, #0x1000000
wait_loop1:
    subs   r2, r2, #1
    bne    wait_loop1
    mov    pc, lr

```

코드 849. start.S 파일(계속)

앞에서 제대로 RAM을 찾을 수 없었다면, 제어는 badram으로 옮겨지게 될 것이며, badram에서는 blinky로 branch(b)하게 된다. blinky에서는 단순히 LED를 깜빡이는 일을 할 것이다. led_on과 led_off는 앞에서 이미 보았지만, LED를 켜고 끄는 일을 하게되며, wait_loop는 delay를 주기 위해서 사용하고 있다.

```

/* the first megabyte is OK, so let's clear it */
    mov    r0, #((1024 * 1024) / (8 * 4)) /* 1MB in steps of 32 bytes */
    ldr    r1, MEM_START

```

³⁷⁵ 이와 같은 이유로 앞에서 r1에서 r4까지는 사용하지 않았었다. r0와 r5, r6, r7만이 사용되었다.

```

    mov    r2, #0
    mov    r3, #0
    mov    r4, #0
    mov    r5, #0
    mov    r6, #0
    mov    r7, #0
    mov    r8, #0
    mov    r9, #0
clear_loop:
    stmia  r1!, {r2-r9}
    subs   r0, r0, #(8 * 4)
    bne   clear_loop

```

코드 850. start.S 파일(계속)

이전 첫 1 MBytes에 대한 확인이 끝났다. 이젠 확인된 1MBytes부분을 0으로 clear시키는 일을 하게된다. 먼저 r0에 32 bytes단위로 1 Mbytes를 나눈 값으로 초기화한 후, r1에는 시작 메모리의 주소(MEM_START)를 가르키도록 만든다. r2에서 r9까지는 0으로 초기화 한다. stmia(Store and Increment after)를 사용해서 r1이 가르키는 주소에 r2에서 r9까지를 저장한 후, r1을 증가시킨다. r0는 원래의 값에서 32를 감소시키게되며, 1 Mbytes를 clear했는지를 알아본다. Clear를 다하지 못했다면, 계속 clear_loop를 돌면서 수행할 것이다.

```

/* get a clue where we are running, so we know what to copy */
and    r0, pc, #0xff000000/* we don't care about the low bits */

/* relocate the second stage loader */
add    r2, r0, #(128 * 1024)          /* blob is 128kB */
add    r0, r0, #0x400                 /* skip first 1024 bytes */
ldr    r1, MEM_START
add    r1, r1, #0x400                 /* skip over here as well */

```

코드 851. start.S 파일(계속)

이전 현재 우리가 어디서 진행하고 있는지를 알기 위해서 pc(Program Counter)의 상위 1bytes만을 제외하고 나머지를 0으로 clear시켜서 r0에 넣는다. 이 값에 128K($=128 \times 1024$)를 곱해서 r2로 두고, r0에는 다시 0x400만큼을 더한다. r1에는 메모리 시작주소(MEM_START)를 가르키도록 만들고, 여기에 0x400을 더해서 첫 1024($=0x400$)을 skip하도록 한다. 이 절차는 rest-ld-script이라는 곳에서 시작 주소를 0xC0000400으로 준 것에 관련되어 있는 것으로 blob가 첫 1024 Bytes이후의 번지로 copy가 될 것이기 때문이다.

```

/* r0 = source address
 * r1 = target address
 * r2 = source end address
 */
copy_loop:
    ldmia  r0!, {r3-r10}
    stmia  r1!, {r3-r10}
    cmp    r0, r2
    ble    copy_loop

    /* turn off the LED. if it stays off it is an indication that
     * we didn't make it into the C code
     */
    bl led_off

    /* set up the stack pointer */
    ldr    r0, MEM_START

```

```

add    r1, r0, #(1024 * 1024)
sub    sp, r1, #0x04

/* blob is copied to ram, so jump to it */
add    r0, r0, #0x400
mov    pc, r0

```

코드 852. start.S 파일 (계속)

r0에는 앞에서 copy할 source의 시작 주소가 들어가도록 하고, r1에는 target이 되는 주소를, r2에는 copy의 대상이 되는 것의 마지막 주소(128Kbytes의 크기)가 들어가도록 해주었다. 이젠 r0가 가르키는 곳에선 r1이 가르키는 곳으로 copy를 한다. 이것은 ldmia와 stmia의 연속적인 것으로 가능하며, r0가 r2와 같게 되었을 때, copy가 끝난다. 그렇지 않다면, copy_loop로 제어를 옮겨서 계속 copy를 할 것이다.

copy가 끝나면, LED를 off시키고(led_off), r0은 다시 메모리의 시작 번지로 만들어주게 되며(MEM_START), r1은 메모리의 시작주소에서 1Mbytes 떨어진 곳을, sp는 다시 r1보다 4작은 곳을 가르키도록 만든다. 이젠 stack pointer를 설정해 주었으므로 C routine을 실행할 준비가 되었다. 이미 앞에서 1024Bytes 정도 MEM_START에서 떨어진 곳에 copy를 해두었으므로, r0에 다시 0x400을 더한 후 pc를 그곳으로 옮겨서 실행을 계속 진행하도록 한다(mov pc, r0). 실제 수행이 옮겨지는 곳은 trampoline.S에 정의된 second stage boot loader로서 다음과 같은 정의를 가진다.

```

.text

.globl _trampoline
_trampoline:
    bl    main
    /* if main ever returns we just call it again */
    b     _trampoline

```

코드 853. trampoline.S 파일

trampoline.S 파일은 단지 main() 함수로 branch with link하는 연산만을 수행할 뿐이다. 만약 main에서 return하게 된다면, 다시 한번 _trampoline을 호출하도록 해서 계속 loop를 돈다. 이젠 main.c로 진행할 준비가 다 끝난 것이며, main.c에 있는 main() 함수가 수행될 것이다.

17.6.2. main.c의 분석

main.c 파일이 BLOB의 핵심 routine이 들어있는 부분이다. 이곳에서 BLOB가 하는 일은 다음과 같이 요약해 볼 수 있다. 먼저, serial을 초기화하고, timer를 동작시키며, 시스템의 메모로 map을 구한 후, BLOB의 상태를 초기화 한다. 이것을 마치고나면, 커널과 RAM disk를 Flash로부터 읽어서 RAM으로 가져다 두게되고, 사용자의 입력을 기다리게 된다. main.c에 있는 entry point가 되는 routine은 main() 함수이다.

```

int main(void)
{
    u32 blockSize = 0x00800000;
    int numRead = 0;
    char commandline[128];
    int i;
    int retval = 0;

    /* Turn the LED on again, so we can see that we safely made it
     * into C code.
     */
    led_on();

    /* We really want to be able to communicate, so initialise the

```

```

/* serial port at 9k6 (which works good for terminals)
 */
SerialInit(baud9k6);
TimerInit();

/* Print the required GPL string */
SerialOutputString("\nConsider yourself LARTed!\n\n");
SerialOutputString(PACKAGE " version " VERSION "\n"
                  "Copyright (C) 1999 2000 2001 "
                  "Jan-Derk Bakker and Erik Mouw\n"
                  "Copyright (C) 2000 "
                  "Johan Pouwelse\n");
SerialOutputString(PACKAGE " comes with ABSOLUTELY NO WARRANTY; "
                  "read the GNU GPL for details.\n");
SerialOutputString("This is free software, and you are welcome "
                  "to redistribute it\n");
SerialOutputString("under certain conditions; "
                  "read the GNU GPL for details.\n");

```

코드 854. main.c 파일

먼저, main() 함수는 LED를 켜기 위해서 led_on()함수를 호출한다. 또한 serial과 timer를 각각 초기화 하기 위해서 SerialInit()와 TimerInit()함수를 호출한다. SerialInit() 함수를 호출할 때 넘겨주는 값은 baud9k6(=23)으로 serial.h에 아래와 같은 정의를 가진다.

```

typedef enum { /* Some useful SA-1100 baud rates */
    baud1k2 = 191,
    baud9k6 = 23,
    baud19k2 = 11,
    baud38k4 = 5,
    baud57k6 = 3,
    baud115k2 = 1
} eBauds;

```

코드 855. SA1100의 Serial을 위한 baud rate의 정의

위와 같이 정의된 값은 아래의 공식을 이용해서 baud rate를 정하는데 사용되는 값이다. 정의된 값은 BRD에 해당한다.

$$BaudRate = \frac{3.6864 \times 10^6}{16 \times (BRD + 1)}$$

이 공식에서 사용될 BRD 값은 나중에 사용되는 serial port의 UART control register에 들어가서, serial의 baud rate를 결정한다. 여기서 잠시 serial의 초기화를 보고 넘어가도록 하자. 코드는 serial.c에 아래와 같다.

```

void SerialInit(eBauds baudrate)
{
#if defined USE_SERIAL1
    while(Ser1UTSR1 & UTSR1_TBY) {
    }
    Ser1UTCR3 = 0x00;
    Ser1UTSR0 = 0xff;
    Ser1UTCR0 = ( UTCR0_1StopBit | UTCR0_8BitData );
    Ser1UTCR1 = 0;
    Ser1UTCR2 = (u32)baudrate;
    Ser1UTCR3 = ( UTCR3_RXE | UTCR3_TXE );

```

```
#elif defined USE_SERIAL3
    while(Ser3UTSR1 & UTSR1_TBY) {
    }
    Ser3UTCR3 = 0x00;
    Ser3UTSR0 = 0xff;
    Ser3UTCR0 = ( UTCR0_1StopBit | UTCR0_8BitData );
    Ser3UTCR1 = 0;
    Ser3UTCR2 = (u32)baudrate;
    Ser3UTCR3 = ( UTCR3_RXE | UTCR3_TXE );
#else
#error "Configuration error: No serial port used at all!"
#endif
}
```

코드 856. SerialInit() 함수의 정의

즉, serial port 1을 사용할지 아니면, serial port 3을 사용할지에 따라서, 각각이 사용하는 register set이 달라지지만 근본적으로 동일한 register set을 serial port 1과 3이 같이 가지고 있으므로, 들어가는 내용은 동일하다. 먼저, 각각의 serial port의 상태가 busy이라면, 상태가 바뀔 때까지 기다린다. 그리고나서, UTCR3(UART Control Register 3)에 0x00을 써서 Rx와 Tx를 OFF시킨 후, UTSR0(UART Status Register 0)에 0xFF를 써서 지우도록(clear) 한다. 다시 UTCR0에 1 stop bit과 8 bit data로 설정을 변경한 후, UTCR1에는 0을 UTCR2에는 전달받은 baud rate 정보를 기입해서 어떤 baud rate으로 동작할지를 결정해 준다. 이것을 다 마치면, 이젠 UTCR3에 Rx Enable가 Tx Enable bit을 설정(set)해서 Rx와 Tx가 일어나도록 만들어준다. 여기서 UTCR1은 앞의 공식에서 사용한 BRD의 상위 4 bit을, UTCR2는 하위 8 bit을 가지도록 만들어 주는 것이다³⁷⁶.

TimerInit() 함수의 정의는 time.c에 있으며, 아래와 같이 되어 있다. 이곳에서 하는 일은 시스템에 있는 timer interrupt의 발생빈도를 설정하는 것이다.

```
void TimerInit(void)
{
    /* clear counter */
    OSCR = 0;
    /* we don't want to be interrupted */
    OIER = 0;
    /* wait until OSCR > 0 */
    while(OSCR == 0)
        ;
    /* clear match register 0 */
    OSMR0 = 0;
    /* clear match bit for OSMR0 */
    OSSR = OSSR_M0;
    numOverflows = 0;
}
```

코드 857. TimerInit() 함수의 정의

SA1100에는 크게 OSCR(Operating System Counter Register)라는 up-counter register와 4개의 OSMR(Operating System Match Register)가 존재한다. OSCR은 reset의 영향을 받지 않으며, OSMR은 사용자가 read와 write를 할 수 있다. 만약 OSCR이 OSMR의 특정 레지스터와 일치하고, interrupt enable bit이 설정된 경우에는 OSSR(Operating System Status Register)에 해당 bit이 설정되게 되며, 이러한 bit은 interrupt controller bit에 route되어 인터럽트를 발생시키도록 할 수 있다. 또한 OSMR3는 watchdog match register의 역할을 수행할

³⁷⁶ 여기서 정의된 register들에 대한 것은 ~/include/asm-arm/arch-sa1100/SA-1100.h에 그 정의가 나온다. 따라서, BLOB을 compile하기 위해서는 적절한 SA1100의 patch가 된 커널의 source를 가지고 있어야 할 것이다. 이하 부분에서 나오는 각종 레지스터의 정의도 마찬가지다.

수 있도록 되어 있어서, 만약 OSCR이 OSMR3와 같아진다면, SA1100을 reset시킬 수 있다. Reset이후에 known state로 갈 수 있는 관련된 레지스터로는 또한 WMER(Watchdog Match Enable Register)가 있으며, 사용자는 FIQ(Fast Interrupt)와 IRQ(Interrupt) interrupt를 CPU에 enable 시키기 전에, 레지스터들을 초기화 시켜야 하며, status register는 clear시켜주어야 한다.

위에서는 OSCR을 먼저 0으로 지우고(clear), 다시 OIER(Operating System Timer Enable Register)를 다시 0으로 지웠다. 이것으로 counter와 interrupt를 다 clear 시켜주었다. 이젠 OSCR에 countering이 일어날 때까지 기다렸다가, OSMR0를 다시 0으로 만들어 주었으며, OSSR의 OSMR0와 일치하는 bit을 clear 시켜주었다. 즉, OSSR의 특정 bit에 1을 write하는 것으로 clear 시킨다. 또한 numOverflows는 Overflow가 일어난 회수를 가지는 변수로 0으로 초기화 시킨다. 이것을 timer에 대한 초기화를 마친다.

나마지 BLOB의 main() 함수에 대한 것은 serial을 통해서 string을 쓰는 것으로, SerialOutputString() 함수³⁷⁷를 호출해서 처리하고 있다. 앞으로도 이 함수를 많이 사용하게 될 것이므로 여기서 보도록 하자. 정의된 곳은 serial.c 파일이며 아래와 같다.

```
/*
 * Output a single byte to the serial port.
 */
void SerialOutputByte(const char c)
{
#if defined USE_SERIAL1
    /* wait for room in the tx FIFO */
    while((Ser1UTSR0 & UTSR0_TFS) == 0);
    Ser1UTDR = c;
#elif defined USE_SERIAL3
    /* wait for room in the tx FIFO */
    while((Ser3UTSR0 & UTSR0_TFS) == 0);
    Ser3UTDR = c;
#else
#error "Configuration error: No serial port used at all!"
#endif
/* If \n, also do \r */
if(c == '\n')
    SerialOutputByte('\r');
}
/*
 * Write a null terminated string to the serial port.
 */
void SerialOutputString(const char *s) {
    while(*s != 0)
        SerialOutputByte(*s++);
}
/* SerialOutputString */
```

코드 858. SerialOutputString() 함수의 정의

SerialOutputString()은 단순히 character string을 입력으로 받아서, 이것을 한 byte 단위로 SerialOutputByte() 함수를 호출해서 출력하는 일을 한다. 실제적인 출력은 SerialOutputByte()에서 일어난다. SerialOutputByte() 함수는 사용하게 되는 serial port에 따라 Ser1UTDR과 Ser3UTDR 레지스터에 하나의 character를 적어 넣는 일을 한다. 먼저 쓰기전에 UTSR 레지스터를 읽어서, Tx FIFO가 full인지 아닌지를 검증하게 된다. 만약 Tx FIFO가 half-full이하이게 되면, UTDR 레지스터에 데이터를 쓴다. 만약 쓰려고한 데이터가 “\n” character일 경우에는 “\r”을 추가해서 line feed가 되도록 만든다. 보시다시피 이 함수는 재귀적인(recursive)한 호출로 사용되고 있음을 알 수 있을 것이다. 이젠 계속 main() 함수를 보도록 하자.

³⁷⁷ 앞에서 나온 PACKAGE와 VERSION과 같은 문자열을 나타내는 macro는 BLOB를 compile하는 과정에서 생성된다.

```

/* get the amount of memory */
get_memory_map();

/* initialise status */
blob_status.kernelSize = 0; /* 커널의 크기를 0으로 초기화 시킨다.*/
blob_status.kernelType = fromFlash; /* fromFlash로 kernelType을 초기화 시킨다.*/
blob_status.ramdiskSize = 0; /* RAM disk의 크기를 0으로 초기화 시킨다.*/
blob_status.ramdiskType = fromFlash; /* fromFlash로 ramdiskType을 초기화 시킨다.*/
blob_status.blockSize = blockSize; /* blockSize필드를 blockSize(=0x00800000)으로
둔다.*/
blob_status.downloadSpeed = baud115k2; /* Baud rate을 11500BPS로 설정한다.*/

/* Load kernel and ramdisk from flash to RAM */
Reload("blob");
Reload("kernel");
Reload("ramdisk");

#endif BLOB_DEBUG
/* print some information */
SerialOutputString("Running from ");
if(RunningFromInternal())
    SerialOutputString("internal");
else
    SerialOutputString("external");
SerialOutputString(" flash, blockSize = 0x");
SerialOutputHex(blockSize);
SerialOutputByte('\n');
#endif

```

코드 859. main.c 파일(계속)

`get_memory_map()` 함수는 시스템에 있는 메모리의 양을 구하는 함수이다. 즉, 이후에 커널 및 RAM disk를 load하기 위해서, 현재 시스템에 얼마나 많은 메모리가 있는지를 알아야 할 것이다. 또한 `blob_status` 변수에 대한 초기화도 이곳에서 일어나게 된다. 이 변수는 `main.h`에 있는 `blob_status_t` 구조체로 정의되며, 아래와 같다.

```

typedef struct {
    int kernelSize; /* 커널의 크기 */
    block_source_t kernelType; /* 커널을 어디서 가져오는지를 정의한다.*/
    int ramdiskSize; /* RAM disk의 크기 */
    block_source_t ramdiskType; /* RAM disk를 어디서 가져오는지를 정의한다.*/
    int blobSize; /* BLOB 자체의 크기 */
    block_source_t blobType; /* BLOB 자체를 어디서 가져오는지를 정의한다.*/
    u32 blockSize; /* 하나의 Block의 크기를 정의한다.*/
    eBauds downloadSpeed; /* Downloading Speed를 정의한다.*/
} blob_status_t;

```

코드 860. `blob_status_t` 구조체의 정의

이 구조체에서 사용하고 있는 `block_source_t` 구조체와 `eBauds`는 다시 아래와 같이 `main.h`와 앞에서 본 `serial.h`에 정의되어 있다.

```

typedef enum {
    fromFlash = 0,
    fromDownload = 1
}

```

```
} block_source_t;
```

코드 861. block_source_t 구조체의 정의

block_source_t 구조체에는 두개의 필드가 있으며, 어디서 image를 가져오는지를 정하고 있다. 각각을 enumeration으로 설정해서 fromFlash와 fromDownload에 0과 1을 주고 있다.

앞에서 blob_status_t 구조체를 초기화 시키는 것은 이미 보았으며, 이것을 마치고나면, Reload() 함수를 호출해서 blob와 kernel 및 ramdisk를 reload한다. BLOB_DEBUG이 선언된 경우에는 debugging을 위한 메시지를 화면에 나타내기 위해서 SerialOutputXXX()라는 함수를 사용하게 되는데, 그 중간에 BLOB가 internal이거나 external이거나를 가리기 위해서 RunningFromInternal()이란 inline 함수를 사용하게 된다. 이 inline 함수는 flash.h에 아래와 같은 정의를 가진다.

```
static inline int RunningFromInternal(void) {
    if(((*(u32 *)0xA0000010) & 0x04) == 0)
        return 1;
    else
        return 0;
}
```

코드 862. RunningFromInternal() inline 함수의 정의

이 함수는 0xA0000010에 있는 값을 0x04와 AND시켜서 이 값이 0이면 1을 돌려주고, 그렇지 않으면 0을 돌려주는 함수이다. 실제 0xA0000010은 Static Memory Control Register(MSC0)을 나타내는 레지스터로 이곳의 RBw0에 있는 bit과 0x04가 AND된다. 즉, 이 bit은 ROM의 bus width를 나타내는 값으로 1인 경우에는 16 bit를, 0인 경우에는 32 bit bus width를 가진다는 뜻이다. Reset 이후에는 이 값은 ROM_SEL pin의 inverse값을 가지도록 설정되며, 즉, 현재 ROM에서 실행되고 있는지 아니면, RAM에서 수행되고 있는지를 알려준다.

더 진행하기에 전에, 앞에서본 get_memory_map() 함수와 Reload() 함수를 보도록 하자. 각각의 함수는 memory.c와 main.c에 정의되어 있다.

```
void get_memory_map(void)
{
    u32 addr;
    int i;

    /* init */
    for(i = 0; i < NUM_MEM_AREAS; i++)
        memory_map[i].used = 0;
    /* first write a 0 to all memory locations */
    for(addr = MEMORY_START; addr < MEMORY_END; addr += TEST_BLOCK_SIZE)
        *(u32 *)addr = 0;
    /* scan memory in blocks */
    i = 0;
    for(addr = MEMORY_START; addr < MEMORY_END; addr += TEST_BLOCK_SIZE) {
        if(testram(addr) == 0) {
            /* yes, memory */
            if(* (u32 *)addr != 0) { /* alias? */
#endif BLOB_DEBUG
                /* Debugging을 위한 목적으로 serial을 통해서 output을 내보낸다. */
#endif
                if(memory_map[i].used)
                    i++;
                continue;
            }
        }
    }
}
```

```

/* not an alias, write the current address */
* (u32 *)addr = addr;

#ifndef BLOB_DEBUG
    /* Debugging을 위한 목적으로 serial을 통해서 output을 내보낸다.*/
#endif

        /* does this start a new block? */
        if(memory_map[i].used == 0) {
            memory_map[i].start = addr;
            memory_map[i].len = TEST_BLOCK_SIZE;
            memory_map[i].used = 1;
        } else {
            memory_map[i].len += TEST_BLOCK_SIZE;
        }
    } else {
        /* no memory here */
        if(memory_map[i].used == 1)
            i++;
    }
}

```

코드 863. get_memory_map() 함수의 정의

먼저 사용하게 될 메모리의 메모리 맵(map)을 나타내는 memory_map[] 변수의 used 필드를 전부 0으로 clear시킨다. NUM_MEM AREAS는 전체 메모리 맵의 영역이 얼마나 있는가를 가르키는 상수 값으로 32를 가지며, memory.h에 정의되어 있다. 이 32라는 숫자는 alias로 연결되어 있는 부분을 찾는데 충분한 값으로 설정되어야 한다. memory_map[] 변수는 아래와 같은 memory_area_t 구조체로 정의된다. 이젠 이렇게 적어놓은 메모리에 대해서 다시 loop를 돌면서 실제로 메모리가 있는지를 확인하는 작업이 되며, 앞에서 본 것과 유사한 testram() 함수가 사용된다. 단지 앞에서 본 함수와 차이가 나는 점은 이 함수는 C에서 호출이 되는 함수이기 때문에, 사용하는 레지스터들에 대해서 save하는 것이 따르게 된다. return값은 앞에서와 마찬가지로 0인 경우에는 존재한다는 것을 나타내며, 그렇지 않은 경우에는 1이 돌려받게 된다. 정의된 곳은 testmem2.S 파일이다. 함수의 앞과 뒤 부분을 유심히 보도록 하자.³⁷⁸

```

typedef struct {
    u32 start;           /* 메모리 영역의 시작 주소 */
    u32 len;             /* 메모리 영역의 길이 */
    int used;            /* 사용되었는지를 나타내는 flag */
} memory_area_t;

```

코드 864. memory_area_t 구조체의 정의

메모리를 1MBytes 단위(=TEST_BLOCK_SIZE)씩 증가 시키면서 각 메모리 block의 시작 위치에 loop를 돌면서 0을 쓰도록 한다. 이때 메모리의 시작(MEMORY_START)와 메모리의 마지막 주소(MEMORY_END)는 각각 0xC0000000과 0xE0000000이다. 이 영역에 있는 메모리는 SA1100의 메뉴얼에서 찾을 수 있듯이 DRAM BANK 0, 1, 3, 4 된다.

testram() 함수의 호출 결과가 0이 아니라면, RAM이 존재하지 않는다는 말이 되므로, memory_map[]에 해당 block의 used 필드가 1인 경우에만 블록의 갯수를 나타내는 변수인 i를 증가시켜주고, 그렇지 않다면 다음 loop로 진행한다. 만약, 메모리가 있지만 앞에서 설정한 block의 시작 주소에 쓴 내용이 0과 같아 않다면, 이 메모리가 alias되어서 사용되고 있다는 것을 의미할 수도 있다. 이때는 memory_map[]의 해당 block에 대한 used 필드를 봄에서 1로 설정된 경우에만 블록의 수를 증가 시켜주기 위해서 i를 증가시킨다. 그렇지 않다면, 현재의 주소 값(addr)을 그대로 addr변수가 가르키는 곳에 적어주어서, alias를 판단할 있는 근거를 만들어준다.

만약, 현재의 block이 사용중이지 않다면(memory_map[i].used == 0), memory_map[]의 element를 설정한다. 즉, 시작 주소에는 addr를 기입하고, 크기는 1 Mbytes(=TEST_BLOCK_SIZE), used 필드에는 1로 준다.

³⁷⁸ 여기서는 추가적인 설명을 덧붙이지 않겠다. 앞에서 본 testram() 함수를 참고하기 바란다.

사용중이라면(memory_map[i].used == 1), 단지 크기만을 증가시켜준다(TEST_BLOCK_SIZE를 더해줌). 즉, 이 block은 새로운 block이 아니며, 원래 있던 block의 일부로 생각해 준다는 것이다.

```
SerialOutputString("Memory map:\n");
for(i = 0; i < NUM_MEM_AREAS; i++) {
    if(memory_map[i].used) {
        SerialOutputString(" 0x");
        SerialOutputHex(memory_map[i].len);
        SerialOutputString(" @ 0x");
        SerialOutputHex(memory_map[i].start);
        SerialOutputString(" (");
        SerialOutputDec(memory_map[i].len / (1024 * 1024));
        SerialOutputString(" MB)\n");
    }
}
```

코드 865. get_memory_map() 함수의 정의(계속)

이전 debugging과 각종 정보를 사용자에게 보여줄 차례이다. 각각의 memory_map[] element에 대해서 loop를 돌면서 메모리의 시작과 길이, 그리고, 크기를 써주게 된다. 이때 이용되는 함수가 SerialOutputHex()와 SerialOutputDec()이다. 각각 아래와 같이 serial.c에 정의되어 있다.

```
void SerialOutputHex(const u32 h)
{
    char c;
    int i;

    for(i = NIBBLES_PER_WORD - 1; i >= 0; i--) { /* NIBBLES_PER_WORD = 8 */
        c = (char)((h >> (i * 4)) & 0x0f);
        if(c > 9)
            c += ('A' - 10);
        else
            c += '0';
        SerialOutputByte(c);
    }
}
void SerialOutputDec(const u32 d)
{
    int leading_zero = 1;
    u32 divisor, result, remainder;

    remainder = d;
    for(divisor = 1000000000;
        divisor > 0;
        divisor /= 10) {
        result = remainder / divisor;
        remainder %= divisor;
        if(result != 0 || divisor == 1)
            leading_zero = 0;
        if(leading_zero == 0)
            SerialOutputByte((char)(result) + '0');
    }
}
```

코드 866. SerialOutputHex()와 SerialOutputDec() 함수의 정의

각각의 함수는 넘겨받은 32 bit크기의 변수를 hexadecimal 값과 decimal 값으로 serial을 통해서 내보내는 일을 하며, 결과적으로 앞에서 본 SerialOutputByte() 함수를 사용해서 한 byte단위로 처리한다. 이것은 단순히 data를 알아보기 쉽게 conversion하는 함수이다.

```
void Reload(char *commandline)
{
    u32 *src = 0;
    u32 *dst = 0;
    int numWords;

    if(MyStrNCmp(commandline, "blob", 4) == 0) {
        src = (u32 *)BLOB_RAM_BASE;
        dst = (u32 *)BLOB_START;
        numWords = BLOB_LEN / 4;
        blob_status.blobSize = 0;
        blob_status.blobType = fromFlash;
        SerialOutputString("Loading blob from flash ");
    } else if(MyStrNCmp(commandline, "kernel", 6) == 0) {
        src = (u32 *)KERNEL_RAM_BASE;
        dst = (u32 *)KERNEL_START;
        numWords = KERNEL_LEN / 4;
        blob_status.kernelSize = 0;
        blob_status.kernelType = fromFlash;
        SerialOutputString("Loading kernel from flash ");
    } else if(MyStrNCmp(commandline, "ramdisk", 7) == 0) {
        src = (u32 *)RAMDISK_RAM_BASE;
        dst = (u32 *)INITRD_START;
        numWords = INITRD_LEN / 4;
        blob_status.ramdiskSize = 0;
        blob_status.ramdiskType = fromFlash;
        SerialOutputString("Loading ramdisk from flash ");
    } else {
        SerialOutputString("*** Don't know how to reload \\"");
        SerialOutputString(commandline);
        SerialOutputString("\n");
        return;
    }
    MyMemcpy(src, dst, numWords);
    SerialOutputString(" done\n");
}
```

코드 867. Reload() 함수의 정의

Reload() 함수는 commandline이라는 변수로 string을 넘겨받는다. 이 값을 MyStrCmp()에 해당하는 것이 있는지를 알 수 있도록, blob나 kernel, ramdisk와 같은 문자열과, 비교하는 문자열의 길이를 나타내는 것을 같이 준다. MyStrCmp() 함수는 넘겨받은 문자열을 비교해서 같은지 다른지 만을 보게된다. 아래와 같은 정의를 가진다. MyStrCmp()의 정의는 util.c에 있다. 이 파일에서 MyMemcpy() 함수도 같이 보기로 하자.

```
int MyStrNCmp(const char *s1, const char *s2, int maxlen)
{
    int i;

    for(i = 0; i < maxlen; i++) {
        if(s1[i] != s2[i])
            return ((int) s1[i]) - ((int) s2[i]);
        if(s1[i] == 0)
            break;
    }
}
```

```

        return 0;
    }
    return 0;
} /* MyStrNCmp */

```

코드 868. MyStrCmp() 함수의 정의

MyStrNCmp() 함수는 단지 넘겨받은 두개의 문자열이 같은지 만을 보는 함수이다. 넘겨받은 문자열의 길이 만큼을 비교한다. 같다면 0을 돌려줄 것이며, 다르다면, 첫번째 문자열에서 두번째 문자열의 다른 부분에 대한 string index를 뺀 값을 돌려줄 것이다.

```

void MyMemCpy(u32 *dest, const u32 *src, int numWords)
{
#ifndef BLOB_DEBUG
    SerialOutputString("\n### Now copying 0x");
    SerialOutputHex(numWords);
    SerialOutputString(" words from 0x");
    SerialOutputHex((int)src);
    SerialOutputString(" to 0x");
    SerialOutputHex((int)dest);
    SerialOutputByte('\n');
#endif
    while(numWords--) {
        if((numWords & 0xffff) == 0x0)
            SerialOutputByte('.');
        *dest++ = *src++;
    }
#ifndef BLOB_DEBUG
    SerialOutputString(" done\n");
#endif
} /* MyMemCpy */

```

코드 869. MyMemCpy() 함수의 정의

MyMemCpy() 함수는 목적지(dest)와 원래의 주소(src) 및 복사할 크기를 넘겨받는 함수로 단순히 한번에 32 bit 만큼씩을 loop를 돌면서 복사(copy)하는 일을 해준다. 이때, 복사하려는 크기가 0xFFFF와 AND시켜서 0인 경우에는 “.”를 serial로 출력해서 얼마나 복사가 진행되고 있는지를 나타낸다.

앞에서 하던 이야기로 돌아가서, 이젠 각 if 절에 있는 MyStrNCmp()를 보도록 하자. 가장 먼저 비교하는 것은 “blob”이다. 비교해서 같다면, src와 dst에 각각 BLOB_RAM_BASE(=0xC1000000)와 BLOB_START(=0x00000000)을 가져온다.³⁷⁹ 또한 numWords에는 BLOB_LEN(=0x10000 or 64K)을 4로 나눈 값을 넣고, blob_status.blobSize에 0을 blobType에는 fromFlash(=0 or download from flash)를 두도록 한다. 그리고나서, 이제 이것을 loading한다는 것을 보여주기 위해서 serial에 “loading blob from flash”를 쓴다. 나머지는 MyMemCpy()에서 dst에 src가 가르키는 메모리를 copy해 줄 것이다.

commandline 변수에 “kernel”을 받았을 경우에는 src에 KERNEL_RAM_BASE(=0xC0008000)³⁸⁰을 두고, dst에는 KERNEL_START(=0x10000 or 64K)를 준다. numWords에는 커널의 길이를 나타내는 KERNEL_LEN(=0xC0000 or 3 x 256K)을 4로 나눈 값을 주어 옮길 값이 4 byte 단위로 얼마나 되는지를 표시하며, blob_status.kernelSize에는 0을 blob_status.kernelType에는 fromFlash를 두어서 커널을 flash로부터 loading함을 표시한다. 마지막으로 커널을 loading한다는 것을 나타내기 위해서 SerialOutputString()을 써서 serial로 output을 내보낸다. 역시 실제적인 copy는 MyMemCpy()가 하게 된다.

³⁷⁹ 이것은 Assabet의 경우를 가정해서 이렇게 해주었다. 자세한 것은 flash.h를 참조하기 바란다.

³⁸⁰ 이 값은 나중에 커널을 compile할 때 적재할 주소를 알려주는 부분에서 0xC0008000으로 설정해 주기 때문이다.

이전 마지막으로 남은 RAM disk에 대한 것이다. src에는 RAMDISK_RAM_BASE(=0xC0800000)을 주고, dst에는 INITRD_START(Initial RAM Disk Start = KERNEL_START + KERNEL_LEN)를 준다. numWords에는 INITRD_LEN(=0x280000 or 5 x 512K, 대략 2.5 Mbytes정도)를 4로 나눈 값을 주고, blob_status.ramdiskSize에는 0을 blob_status.ramdiskType에는 fromFlash를 두어서, flash부터 loading함을 설정한다. 역시 SerialOutputString() 함수를 사용해서 ramdisk를 flash로부터 loading한다는 것을 알려준다. 그 외의 경우에는 잘못된 Reload() 함수의 호출이므로 debugging 메시지를 출력한 후 복귀하도록 한다. 따라서, 최종적으로 RAM에 loading된 flash image는 다음과 같게 나타내지게 된다.

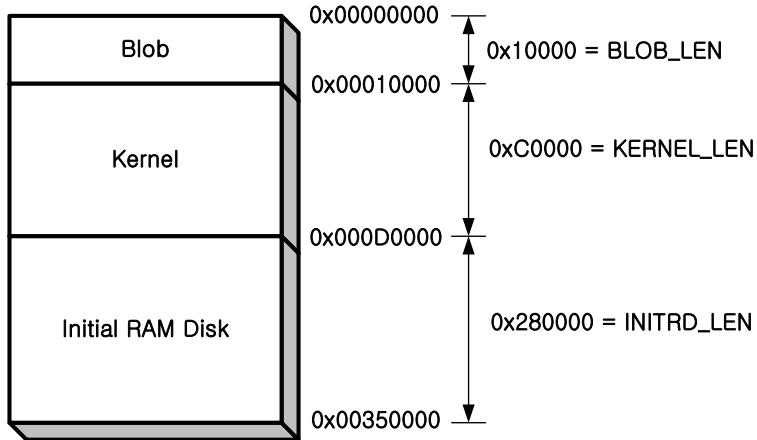


그림 123. Reload() 함수 수행이후의 메모리 상태

여기까지 해서, flash에 있는 메모리를 시스템의 DRAM 영역으로 loading하는 것을 마쳤다. 다시 main() 함수로 돌아가서 계속 진행하도록 하자.

```

/* wait 10 seconds before starting autoboot */
SerialOutputString("Autoboot in progress, press any key to stop ");
for(i = 0; i < 10; i++) {
    SerialOutputByte('.');
    retval = SerialInputBlock(commandline, 1, 1);
    if(retval > 0)
        break;
}
/* no key was pressed, so proceed booting the kernel */
if(retval == 0) {
    commandline[0] = '\0';
    boot_linux(commandline);
}
SerialOutputString("\nAutoboot aborted\n");
SerialOutputString("Type \"help\" to get a list of commands\n");
    
```

코드 870. main() 함수의 정의(계속)

SerialOutputString()을 호출해서 autoboot이 진행할 것임을 나타내게된다. 이때 for loop를 돌면서, SerialInputBlock()을 호출해서 입력을 받도록한다. 만약 특정 입력이 있다면, retval은 0이 아닌 값을 가지게 될 것이며, SerialOutputString()을 호출해서 autoboot이 abort되었음을 표시하고, BLOB는 command를 받을 준비를 하게된다. 만약 아무런 input을 받지 않았다면, retval은 0이 될 것이며, commandline에는 '\0'을 첫번째 argument로 주고, boot_linux() 함수를 호출할 것이다.

먼저 SerialInputBlock() 함수를 보도록 하자. SerialInputBlock() 함수는 결과적으로는 SerialInputByte() 함수를 호출하므로 SerialInputByte() 함수와 같이 보도록 하겠다. 정의는 serial.c에 있다.

```

int SerialInputByte(char *c)
{
    
```

```
#if defined USE_SERIAL1
    if(Ser1UTSR1 & UTSR1_RNE) {
        int err = Ser1UTSR1 & (UTSR1_PRE | UTSR1_FRE | UTSR1_ROR);
        *c = (char)Ser1UTDR;
    }#elif defined USE_SERIAL3
    if(Ser3UTSR1 & UTSR1_RNE) {
        int err = Ser3UTSR1 & (UTSR1_PRE | UTSR1_FRE | UTSR1_ROR);
        *c = (char)Ser3UTDR;
    }#else
#error "Configuration error: No serial port at all"
#endif
/* If you're lucky, you should be able to use this as
 * debug information ;-) -- Erik
 */
if(err & UTSR1_PRE)
    SerialOutputByte('@');
else if(err & UTSR1_FRE)
    SerialOutputByte('#');
else if(err & UTSR1_ROR)
    SerialOutputByte('$');
/* We currently only care about framing and parity errors */
if((err & (UTSR1_PRE | UTSR1_FRE)) != 0) {
    return SerialInputByte(c);
} else {
    led_toggle();
    return(1);
}
} /* no bit ready */
return(0);
}
} /* SerialInputByte */
```

코드 871. SerialInputByte() 함수의 정의

SerialInputByte() 함수는 serial port로부터 input을 받는 함수이다. 성공했을 때는 1을 돌려줄 것이며, 실패시에는 0을 돌려준다. 성공적인 경우에는 읽은 character는 넘겨받은 c 변수에 저장된다. 먼저 사용할 serial port가 무엇인가에 따라서 Ser1UTSR1이나, Ser3UTSR1을 보도록 할 것이며, UTSR1_RNE는 Receive FIFO Not Empty를 나타낸다. 따라서, receiver FIFO가 비어있지 않다는 말은 입력을 받을 수 있다는 말이되므로, input을 받기 위해서 Ser1UTDR이나 Ser3UTDR을 접근하게 된다. 읽은 값은 c에 저장된다. 예러가 있었다면, Ser1UTSR1이나 Ser3UTSR1을 확인해서 이 에러값을 err에 저장한다.

각각의 에러가 의미하는 바는 다음과 같다. PRE의 경우는 Parity Error를 나타내며, FRE는 Framing Error를 나타내고, ROR은 Receive Overrun을 나타낸다. Parity error는 받은 데이터의 parity가 문제가 있음을 나타내며, frame error는 stop bit이 1이 아니라, 0이 되었음을 나타내며, receive overrun은 receiver쪽의 FIFO가 이미 full이 되었음을 의미한다. 각 에러 상황에 대해서 SerialOutputByte() 함수를 호출해서 적절한 문자를 display해준다. 만약 error가 PRE나 FRE라면, 다시 한번 SerialInputByte() 함수를 호출해서 읽게되며, 그렇지 않다면, led_toggle() 함수를 호출해서 led_off()와 led_on()을 이전의 LED 상태(led_state)에 맞게 호출해서 깜빡여 준 후, 1을 돌려줄 것이다. Receive FIFO가 비어있다면, 당연히 0을 돌려준다.

```
int SerialInputBlock(char *buf, int bufsize, const int timeout)
{
    u32 startTime, currentTime;
    char c;
    int i;
    int numRead;
```

```

int maxRead = bufsize;

startTime = TimerGetTime();
for(numRead = 0, i = 0; numRead < maxRead;) {
    /* try to get a byte from the serial port */
    while(!SerialInputByte(&c)) {
        currentTime = TimerGetTime();
        /* check timeout value */
        if((currentTime - startTime) >
           (timeout * TICKS_PER_SECOND)) {
            /* timeout! */
            return(numRead);
        }
    }
    buf[i++] = c;
    numRead++;
}
return(numRead);
}

```

코드 872. SerialInputBlock() 함수의 정의

SerialInputBlock()함수는 특정한 문자(character)가 들어왔는지를 보지 않고, 단순히 읽은 문자들의 block을 돌려주는 일을 한다. 먼저 startTime에 현재의 시간(TimerGetTimer())을 보관하고, for loop를 돌면서 문자들을 읽어 들인다. 읽은 문자의 개수가 return값으로 사용된다(numRead). Timeout값은 넘겨준 값에 의존하며, 초단위로 나타낸다.³⁸¹ 만약 SerialInputByte()를 while loop를 돌면서 계속 호출해서 특정한 수의 input 문자를 읽어들이면, for loop를 끝낸다. 그렇지 않고, 입력이 없이 timeout이 된다면, 현재의 읽은 문자의 개수 만큼을 돌려주게 된다. 이곳에서 timeout을 검사하기 위해서 사용하는 TimerGetTime() 함수는 time.c에 아래와 같이 정의된다.

```

/* returns the time in 1/TICKS_PER_SECOND seconds */
u32 TimerGetTime(void)
{
    /* turn LED always on after one second so the user knows that
     * the board is on
     */
    if((OSCR % TICKS_PER_SECOND) < (TICKS_PER_SECOND >>7))
        led_on();
    return((u32)OSCR);
}

```

코드 873. TimerGetTime() 함수의 정의

즉, OSCR 레지스터를 읽어서 바로 돌려준다. 중간에 시스템이 살아있다는 것을 보여주기 위해서 led_on()을 호출해서 LED를 켜주는 것이 전부이다. TICKS_PER_SECOND는 3686400을 가지는 값으로 OS timer의 초당 clock frequency를 나타내는 값이다.

```

void boot_linux(char *commandline)
{
    register u32 i;
    void (*theKernel)(int zero, int arch) = (void (*)(int, int))KERNEL_RAM_BASE;

    setup_start_tag();
    setup_memory_tags();
    setup_commandline_tag(commandline);
}

```

³⁸¹ 앞에서 이미 1로 설정해서 호출해 주었다(main()함수).

```

setup_initrd_tag();
setup_ramdisk_tag();
setup_end_tag();

/* we assume that the kernel is in place */
SerialOutputString("\nStarting kernel ...\\n\\n");

/* turn off I-cache */
asm ("mrc p15, 0, %0, c1, c0, 0": =r" (i));
i &= ~0x1000;
asm ("mcr p15, 0, %0, c1, c0, 0": : "r" (i));

/* flush I-cache */
asm ("mcr p15, 0, %0, c7, c5, 0": : "r" (i));
theKernel(0, ARCH_NUMBER);
SerialOutputString("Hey, the kernel returned! This should not happen.\\n");
}

```

코드 874. boot_linux() 함수의 정의

boot_linux() 함수는 linux.c에 정의된 함수이다. 여기서 하는 일은 커널에 전달될 parameter를 만드는 일과 I-cache를 OFF시키고, flush시키는 일이며, 최종적으로 압축된 Linux를 실행시키는 일이다. 따라서, 압축된 커널의 시작부분에는 uncompression하는 code가 들어가 있을 것이다. 리눅스 커널의 실행시에 중요한 넘겨주어야 할 변수로는 첫번째 argument에는 0을, 두번째 argument에는 architecture 번호를 주어야 한다는 것이다. Kernel이 return을 하지 않을 것이므로, 마지막 line에 있는 SerialOutputString()은 실행되지 않을 것이다.

넘겨주는 architecture 번호에 대한 정의는 linux.h에 아래와 같이 되어 있다. 같은 것을 커널 source의 ~/arch/arm/tools/mach_types과 같은 곳에서 찾을 수 있다.

```

#ifndef ASSABET
#define ARCH_NUMBER (25)
#elif defined BRUTUS
#define ARCH_NUMBER (16)
#elif defined CLART
#define ARCH_NUMBER (68)
#elif defined LART
#define ARCH_NUMBER (27)
#elif defined NESA
#define ARCH_NUMBER (75)
#elif defined PLEB
#define ARCH_NUMBER (20)
#elif defined SHANNON
#define ARCH_NUMBER (97)
#else
#warning "FIXME: Calling the kernel with a generic SA1100 architecture code. YMMV!"
#define ARCH_NUMBER (18)
#endif

```

코드 875. 각 CPU architecture에 따른 ARCH_NUMBER의 정의

또한, 넘겨준 architecture 번호 값은 커널 source의 ~/arch/arm/boot/compressed/head.S에서 다음과 같이 사용하게 된다.

```

...
.align
start:
.type start,#function

```

```

.rept    8
mov     r0, r0
.endr

b      1f
.word   0x016f2818      @ Magic numbers to help the loader
.word   start            @ absolute load/run zImage address
.word   _edata           @ zImage end address
1:      mov     r7, r1            @ save architecture ID
       mov     r8, #0           @ save r0
...

```

코드 876. 커널의 head.S의 시작부분

head.S에서 보듯이 넘겨받은 argument값은 r0에는 001, r1에는 ARCH_NUMBER가 들어가게 되며, 이 값은 다시 r7에 들어가서, architecture ID로서의 구실을 하게된다. 더 자세한 것은 나중에 SA1100의 booting에 대해서 설명하면서 보게 될 것이므로, 여기서는 이 정도에서 마치기로 하고, setup_XXX() 함수들에 대해서 보기로 하겠다. setup_XXX() 함수는 커널에 넘겨줄 parameter를 생성해주는 함수이다. 이것은 마치 x86에서 LILO를 사용하게 되는 경우와 동일한 것으로 커널에 적절한 argument를 넘겨주어서, 커널에서 사용하는 변수를 설정하는 일을 한다.

먼저 setup_XXX() 함수를 보기 전에 setup_XXX() 함수에서 사용하게 되는 tag 구조체에 대한 정의부터 보기로 하자. 이 구조체는 커널 source의 ~/include/asm-arm/setup.h에 아래와 같은 정의를 가진다.

```

struct tag {
    struct tag_header hdr;                      /* tag의 크기와 magic number */
    union {
        struct tag_core          core;           /* 메모리의 크기와 시작 주소 */
        struct tag_mem32         mem;            /* 비디오 RAM의 위치와 resolution */
        struct tag_videotext videotext;          /* RAM disk의 옵션, 크기 및 시작 주소 */
        struct tag_ramdisk ramdisk;             /* Initial RAM disk의 크기와 시작 주소 */
        struct tag_initrd initrd;              /* Serial의 개수 */
        struct tag_serialnr serialnr;           /* Revision 번호 */
        struct tag_revision revision;           /* Video Fram Buffer의 설정 사항 */
        struct tag_videoflb videoflb;
        struct tag_cmdline cmdline;
        /* */
        /* Acorn specific */
        /* */
        struct tag_acorn acorn;                /* ACORN specific 정보 */
        /* */
        /* DC21285 specific */
        /* */
        struct tag_memclk memclk;             /* 메모리 clock 정보 */
    } u;
};

```

코드 877. tag 구조체의 정의

즉, 위와 같이 정의된 tag 구조체를 적절한 메모리 위치에 두고, 커널에서는 나중에 이곳에 정의된 정보를 다시 읽어서 사용하도록 하는 것이다. 이것이 정의된 위치는 main.h에 있는 BOOT_PARAMS(=0xc0000100)에 들어가게 된다.

```

static void setup_start_tag(void)
{

```

```

params = (struct tag *)BOOT_PARAMS;

params->hdr.tag = ATAG_CORE;
params->hdr.size = tag_size(tag_core);
params->u.core.flags = 0;
params->u.core.pagesize = 0;
params->u.core.rootdev = 0;
params = tag_next(params);
}

```

코드 878. setup_start_tag() 함수의 정의

이 함수는 BOOT_PARAMS에 가지고 있는 tag구조체를 초기화 하는 역할을 한다. ATAG_CORE는 커널의 ~/include/arm/setup.h에 정의된 것으로 0x54410001라는 값을 가진다.³⁸² 크기는 tag_core의 크기를 주도록 하고(tag_size()), union 구조체로 선언된 부분의 core에 대한 초기화를 위해서 flags에 0, pagesize에 0, rootdev에도 0을 준다. 다음번의 tag의 위치를 주기 위해서 tag_next()를 사용하고 있다.³⁸³

```

#define tag_next(t) ((struct tag *)((u32 *)(t) + (t)->hdr.size))
#define tag_size(type)      ((sizeof(struct tag_header) + sizeof(type)) >> 2)

```

코드 879. tag관련 매크로의 정의

tag_next() 매크로는 다음 tag구조체의 위치를 나타내는데 사용하며, tag_size() 매크로는 tag header의 길이와 해당하는 type의 크기를 합한 값을 4로 나누어 4 byte 단위로 나타낸 크기를 돌려준다.

```

static void setup_memory_tags(void)
{
    int i;

    for(i = 0; i < NUM_MEM_AREAS; i++) {
        if(memory_map[i].used) {
            params->hdr.tag = ATAG_MEM;
            params->hdr.size = tag_size(tag_mem32);
            params->u.mem.start = memory_map[i].start;
            params->u.mem.size = memory_map[i].len;
            params = tag_next(params);
        }
    }
}

```

코드 880. setup_memory_tags() 함수의 정의

setup_memory_tags() 함수는 현재 시스템 메모리 설정에 대한 정보를 가르쳐줄 목적으로 사용한다. Tag의 header에 ATAG_MEM을 넣고 tag의 크기를 알려준 다음, 각각의 메모리 map(memory_map[])의 entry에 대해서 시작과 크기를 넣어준다.

```

static void setup_commandline_tag(char *commandline)
{
    char *p;

    /* eat leading white space */
    for(p = commandline; *p == ' '; p++)
}

```

³⁸² 현재로서는 단순히 magic number를 가지는 것으로 보인다.

³⁸³ tag_size()나 tag_next()은 새로운 버전의 커널에 들어 있는 것을 찾을 수 있을 것이다. 커널 버전 2.4.9에서는 ~/include/arm/setup.h에서 찾을 수 있다. 다른 버전의 커널도 확인해 보기 바란다. 이것을 정의하고 있지 않다면, BLOB의 build가 제대로 되지 않을 것이다.

```

        ;
/* skip non-existent command lines so the kernel will still
 * use its default command line.
 */
if(*p == '\0')
    return;
params->hdr.tag = ATAG_CMDLINE;
params->hdr.size = (sizeof(struct tag_header) + strlen(p) + 1 + 4) >> 2;
strcpy(params->u.cmdline.cmdline, p);
params = tag_next(params);
}

```

코드 881. setup_commandline_tag() 함수의 정의

이전 넘겨받은 command line에 해당하는 tag를 생성할 차례이다. 먼저, 주어진 command line에서 blank를 제거하기 위해서 문자단위로 검사한다. 만약 이러한 검사를 진행하는 도중에 '\0'를 만난다면, 문자열의 끝을 나타내므로 그냥 return한다. 그렇지 않다면, header의 tag필드에는 command line이라는 것을 알려주는 ATAG_CMDLINE을 넣어주고, command line의 크기를 계산해 주고, 넘겨받은 실제 command line에서 blank를 제외한 부분을 copy해서 넣어준다.

```

static void setup_initrd_tag(void)
{
    /* an ATAG_INITRD node tells the kernel where the compressed
     * ramdisk can be found. ATAG_RDIMG is a better name, actually.
     */
    params->hdr.tag = ATAG_INITRD;
    params->hdr.size = tag_size(tag_initrd);

    params->u.initrd.start = RAMDISK_RAM_BASE;
    params->u.initrd.size = INITRD_LEN;

    params = tag_next(params);
}

```

코드 882. setup_initrd_tag() 함수의 정의

setup_initrd_tag()는 initial RAM disk에 대한 tag를 설정하는 것이다. ATAG_INITRD를 tag head에 넣어주고, RAMDISK_RAM_BASE(=0xC0800000)를 initial RAM disk의 시작 주소로 주고, 크기를 INITRD_LEN(=0x280000)으로 설정했다.

```

static void setup_ramdisk_tag(void)
{
    /* an ATAG_RAMDISK node tells the kernel how large the
     * decompressed ramdisk will become.
     */
    params->hdr.tag = ATAG_RAMDISK;
    params->hdr.size = tag_size(tag_ramdisk);

    params->u.ramdisk.start = 0;
    params->u.ramdisk.size = RAMDISK_SIZE;
    params->u.ramdisk.flags = 1; /* automatically load ramdisk */

    params = tag_next(params);
}

```

코드 883. setup_ramdisk_tag() 함수의 정의

`setup_ramdisk_tag()` 함수는 RAM disk에 대한 tag를 설정하는 함수이다. 시작 주소로 0을, 크기로는 `RAMDISK_SIZE(=8 x 1024 x 1024 or 8Mbytes)`를 주었으며, 자동으로 RAM disk를 load하라는 뜻으로 `flags`에 1을 설정해 주었다.

```
static void setup_end_tag(void)
{
    params->hdr.tag = ATAG_NONE;
    params->hdr.size = 0;
}
```

코드 884. `setup_end_tag()` 함수의 정의

이젠 tag의 마지막임을 나타내기 위해서 `setup_end_tag()` 함수를 사용한다. `ATAG_NONE`을 tag header에 넣었고, tag의 크기가 0임을 알려주었다. 이것으로 커널에 넘겨질 tag에 대한 모든 설정은 끝났다.

실제로 커널에서 넘겨받은 tag에 대한 처리가 일어나는 곳은 `~/arch/arm/kernel/setup.c`의 `parse_tag_XXX()`와 같은 함수이다. 이 함수에서는 넘겨준 tag들을 tag이 의미하는 뜻에 따라, 앞에서 이미 보았듯이, core, memory, commandline, initrd, RAM disk등의 순으로 처리해 나가는 것을 볼 수 있을 것이다.

```
/* the command loop. endless, of course */
for(;;) {
    DisplayPrompt(NULL);
    /* wait 10 minutes for a command */
    numRead = GetCommand(commandline, 128, 600);
    if(numRead > 0) {
        if(MyStrNCmp(commandline, "boot", 4) == 0) {
            boot_linux(commandline + 4);
        } else if(MyStrNCmp(commandline, "clock", 5) == 0) {
            SetClock(commandline + 5);
        } else if(MyStrNCmp(commandline, "download ", 9) == 0) {
            Download(commandline + 9);
        } else if(MyStrNCmp(commandline, "flash ", 6) == 0) {
            Flash(commandline + 6);
        } else if(MyStrNCmp(commandline, "help", 4) == 0) {
            PrintHelp();
        } else if(MyStrNCmp(commandline, "reblob", 6) == 0) {
            Reblob();
        } else if(MyStrNCmp(commandline, "reboot", 6) == 0) {
            Reboot();
        } else if(MyStrNCmp(commandline, "reload ", 7) == 0) {
            Reload(commandline + 7);
        } else if(MyStrNCmp(commandline, "reset", 5) == 0) {
            ResetTerminal();
        } else if(MyStrNCmp(commandline, "speed ", 6) == 0) {
            SetDownloadSpeed(commandline + 6);
        }
        else if(MyStrNCmp(commandline, "status", 6) == 0) {
            PrintStatus();
        } else {
            SerialOutputString("**** Unknown command: ");
            SerialOutputString(commandline);
            SerialOutputByte('\n');
        }
    }
    return 0;
} /* main */
```

코드 885. main.c 파일(계속)

다시 main.c로 돌아와서, 계속 진행하도록 하겠다. 만약 autoreboot이 취소 되었다면, main() 함수는 for loop를 돌면서 사용자의 입력에 따른 반응을 보일 것이다. 먼저 DisplayPrompt() 함수를 호출해서 사용자가 입력을 하기를 기다리게 된다. 여기서부터 명령어와 관련된 처리는 command.c 파일을 보기 바란다. GetCommand() 함수를 호출해서 내려진 명령을 commandline 변수로 읽어서, 이 명령이 어떤 것인가를 비교해서(MyStrNCmp()), 해당하는 명령을 수행하는 loop를 돌게된다. 명령의 종류로는 boot, clock, download, flash, help, reblob, reboot, reload, reset, speed, status 등이 있으며, 그외의 명령에 대해서는 unknown으로 생각한다. 먼저 DisplayPrompt() 함수와 GetCommand() 함수를 본 후, 각각의 command에 대해서 보도록 하자. main() 함수는 이것으로 마무리한다.

```
/* display a prompt, or the standard prompt if prompt == NULL */
void DisplayPrompt(char *prompt)
{
    if(prompt == NULL) {
        SerialOutputString(PACKAGE "> ");
    } else {
        SerialOutputString(prompt);
    }
}
```

코드 886. DisplayPrompt() 함수의 정의

DisplayPrompt() 함수는 단순히 “>”문자를 표시하는 역할을 한다. 만약 넘겨받은 prompt변수의 값이 NULL인 경우에는 “>”를 serial을 통해서 쓰게되고, 그렇지 않고 특정한 prompt를 요구하면, 그것을 그대로 보여준다.

```
/* more or less like SerialInputString(), but with echo and backspace */
int GetCommand(char *command, int len, int timeout)
{
    u32 startTime, currentTime;
    char c;
    int i;
    int numRead;
    int maxRead = len - 1;

    TimerClearOverflow();
    startTime = TimerGetTime();
    for(numRead = 0, i = 0; numRead < maxRead;) {
        /* try to get a byte from the serial port */
        while(!SerialInputByte(&c)) {
            currentTime = TimerGetTime();
            /* check timeout value */
            if((currentTime - startTime) >
                (timeout * TICKS_PER_SECOND)) {
                /* timeout */
                command[i++] = '\0';
                return(numRead);
            }
        }
        if((c == '\r') || (c == '\n')) {
            command[i++] = '\0';
            /* print newline */
            SerialOutputByte('\n');
            return(numRead);
        } else if(c == '\b') { /* FIXME: is this backspace? */
    }
```

```

        if(i > 0) {
            i--;
            numRead--;
            /* cursor one position back.*/
            SerialOutputString("\b \b");
        }
    } else {
        command[i++] = c;
        numRead++;
        /* print character */
        SerialOutputByte(c);
    }
}
return(numRead);
}

```

코드 887. GetCommand() 함수의 정의

GetCommand()함수는 특정 시간동안 명령 input을 받는 함수이다. 먼저 TimerClearOverflow() 함수를 호출해서, timer가 overflow가 되었다면 이를 지워준다. 정의는 time.c에 아래와 같이 되어있다.

```

int TimerDetectOverflow(void)
{
    return(OSSR & OSSR_M0);
}
void TimerClearOverflow(void)
{
    if(TimerDetectOverflow())
        numOverflows++;
    OSSR = OSSR_M0;
}

```

코드 888. TimerClearOverflow() 함수의 정의

TimerClearOverflow()함수는 TimerDetectOverflow() 함수를 다시 호출해서 overflow가 일어났는지를 확인하게 되고, 만약 그렇다면, numOverflows 변수를 하나 증가 시키게 되며, OSSR(Operating System Status Register)은 OSMR(Operating System Match Register) 4개의 레지스터 중에서 어떤 레지스터와 OSCR(Operating System Counter Register)이 일치하였는지를 나타낸다. 따라서, 이 값이 OSMR0와 일치가 된다면(OSSR_M0 = 0x00000001), overflow가 일어났다고 생각하도록 한다. 즉, 현재 우리가 사용하는 timer와 관련된 match register로 앞에서 OSMR0를 사용하기에 이것만을 기준으로 삼고 있다(TimerInit()). For loop를 돌기전에 미리 현재의 시간을 읽어서(TimerGetTime()) startTime 변수에 저장하고, 이 값을 기준으로 얼마나 시간이 흘렀느냐에 따라서 timeout을 시켜준다.

For loop는 최대 읽기 허용 byte를 넘어서거나, timeout이 일어난다면 현재까지 읽은 byte를 돌려주고 복귀할 것이며, 만약 '\r'이나 '\n'과 같은 문자를 받더라도, 같은 일을 할 것이다. 문자열을 읽는 함수는 SerialInputByte()를 사용한다. 한 문자를 읽을 때마다, timeout이 계산되며, 허용된 timeout을 넘어서게 되면, 현재까지 읽은 command line에 '\0'을 덧붙인 문자열을 넘겨줄 것이며, 현재까지 읽은 byte수를 복귀 값으로 전달한다. 만약 읽은 문자가 '\b'(=back space)인 경우에는 읽은 문자의 수를 감소시켜주게 되며, 표시하는 문자에는 '\b'를 두번 써서, 지우는 효과를 보이도록 할 것이다. 그 외의 문자에 대해서는 읽은 값을 그대로 echo시켜준다.

17.6.3. BLOB의 command 분석

여기서 부터는 main()함수에서 BLOB가 command를 읽었을 때, 어떤 일이 일어나는지를 분석할 것이다. 앞에서 이미 보았듯이 분석할 command가 되는 것은 boot, clock, download, flash, help, reblob, reboot, reload, reset, speed, status 등이다. 각각의 command를 차례로 볼 것이다.

먼저 간략하게 BLOB의 command가 어떤 역할을 하는지를 source로 들어가기 전에 보도록 하자. 여기서 command를 입력하기 위해서는 10초정도의 대기 시간내에 해야 할 것이다. 그렇지 않다면, 바로 Linux 커널을 booting시키는 곳으로 진행할 것이다.

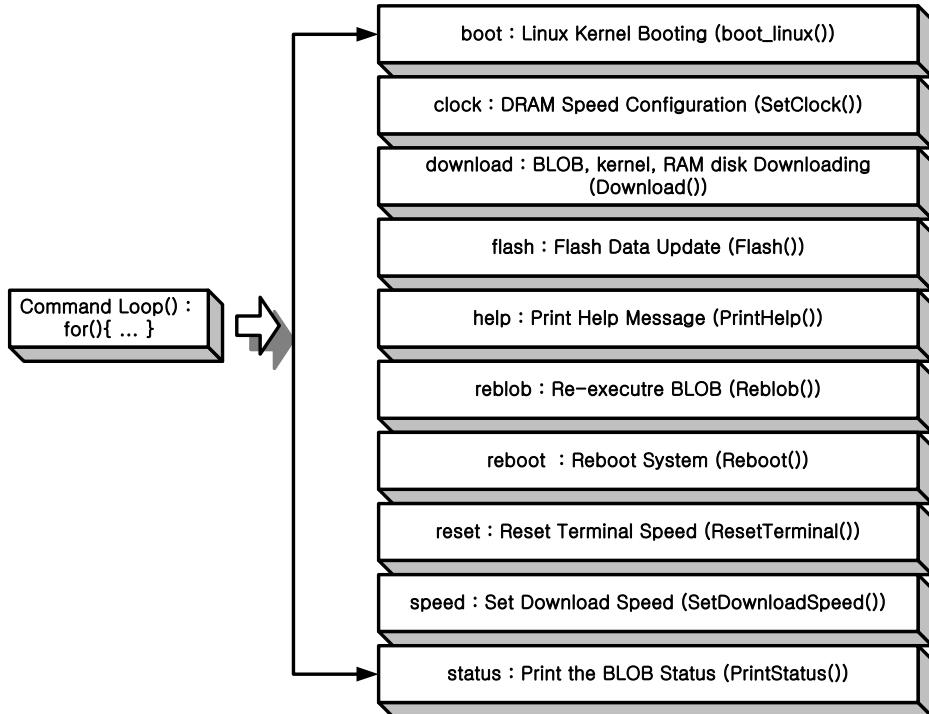


그림 124. BLOB의 Command Loop

[그림 124]은 BLOB에서 사용하고 있는 command와 그것이 해주는 일 및 처리하는 함수를 보여준다. 그림에서 알 수 있듯이, 사용되는 command는 boot, clock, download, flash, help, reblob, reboot, reset, speed, status가 있으며, 각각은 BLOB의 상태를 바꾸는 일을 한다. 즉, BLOB과 커널, RAM disk 등을 downloading을 한다던가, 혹은 reboot 및 RAM 설정을 바꾸는 등의 일이 될 것이다. 이러한 command loop를 통해서 사용자는 serial을 이용해 BLOB의 동작을 조정해 줄 수 있다. 자, 이제 그럼 위에서부터 순서대로 실제적인 코드를 보기로 하자.

17.6.3.1. Boot Command

Boot command는 앞에서 이미 보았다. 즉, boot 명령을 받았을 때 boot_linux() 함수를 호출해서, 현재까지 가지고 있는 commandline에 4를 더해서 넘겨주는 일을 한다. 즉, 이것은 boot 명령어 다음에 커널의 command line에 넘겨줄 데이터를 명시할 수 있다는 의미이다. 관련된 것은 앞에서 이미 설명을 했으므로 다음으로 넘어가도록 하자.

17.6.3.2. Clock Command

Clock command 역시 clock이라는 명령어와 같이 option과 같은 것을 명시해서, 이를 SetClock() 함수로 전달할 수 있도록 하고 있다. SetClock() 함수는 DRAM의 파라미터를 설정하거나, speed를 조정하기 위해서 사용하는 함수로서 clock.c에 아래와 같이 정의되어 있다.

```

/* Struct with the SA-1100 PLL + DRAM parameter registers */
enum {
    SA_PPCR,
    SA_MDCNFG,
    SA_MDCAS0,
    SA_MDCAS1,
    SA_MDCAS2
}

```

```

};

...
void SetClock(char *commandline)
{
int i;
u32 regs[5];
u32 startTime, currentTime;

for(i = 0; i < 5; i++) {
    commandline = GetHexValue(commandline, &regs[i]);
#warning "FIXME: GetHexValue() returns NULL after the fifth call..."
    if((commandline == NULL) && (i != 4)) {
        SerialOutputString(__FUNCTION__ ": can't get hex values\n");
        return;
    }
}
/* we go slower, so first set PLL register */
PPCR = regs[SA_PPCR];
MDCNFG = regs[SA_MDCNFG];
MDCAS0 = regs[SA_MDCAS0];
MDCAS1 = regs[SA_MDCAS1];
MDCAS2 = regs[SA_MDCAS2];
/* sleep for a second */
startTime = TimerGetTime();
for(;;) {
    currentTime = TimerGetTime();
    if((currentTime - startTime) > (u32)TICKS_PER_SECOND)
        return;
}
} /* SetClock */

```

코드 889. SetClock() 함수의 정의

SetClock() 함수는 먼저 받은 commandline을 가지고 GetHexValue() 함수를 호출한다. Command line이 변환된 값은 regs[] 배열에 각각의 register에 따라 채워지게 된다. GetHexValue() 함수의 return값이 NULL이고, 다섯번의 loop를 다 돌지 못했다면, 넘겨받은 commandline을 제대로 다 해석하지 못했다는 것을 의미하므로, SerialOutputString()을 호출해서 error가 생겼다는 것을 나타낸 후, 바로 return한다. 그렇지 않다면, 이 값을 반영하기 위해서 PPCR, MDCNFG, MDCAS0, MDCAS1, MDCAS2 레지스터에 해석한 값을 넣어준다. 또한 이것이 효력을 발생시킬때 까지 가디리기 위해서, 1초 정도의 delay를 준다(for).

```

int MyIsXDigit(char isdigit) {
    if ((isdigit >= '0' && isdigit <= '9') ||
        (isdigit >= 'a' && isdigit <= 'f'))
        return 1;
    else
        return 0;
} /* MyIsXDigit */

int MyXDigitValue(char isdigit) {
    if (isdigit >= '0' && isdigit <= '9')
        return isdigit - '0';
    if (isdigit >= 'a' && isdigit <= 'f')
        return 10 + isdigit - 'a';
    return -1;
} /* MyXDigitValue */

```

코드 890. MyIsXDigit()과 MyXDigitValue() 함수의 정의

MyIsXDigit()과 MyXDigitValue() 함수는 GetHexValue() 함수에서 사용하는 함수로 먼저 잠시보도록 하겠다. 이 함수가 하는 역할은 단순히 hexadecimal인지를 확인하는 것과 hexadecimal로 변형하는 일을 한다. 먼저 MyIsXDigit() 함수는 하나의 문자(isdigit)를 넘겨받아서, 이 값이 ‘0’과 ‘9’사이에 있거나, ‘a’에서 ‘f’사이에 있다면 1을 돌려주고, 그렇지 않다면 0을 돌려준다. MyXDigitValue() 함수는 하나의 문자(isdigit)를 받아들여서, 문자가 ‘0’에서 ‘9’사이에 있다면, 원래의 문자가 가지는 ASCII값에서 ‘0’을 뺀 값을 돌려주고, ‘a’에서 ‘f’사이에 있다면, 원래의 문자가 가지는 ASCII값에서 ‘a’를 뺀다음 이것에 10을 더해서 돌려준다. 즉, 문자가 가지는 hexadecimal로 표현된 의미를 10진수로 바꾼다.

```
/* Converts 8 characters 0-9a-f into a 4 byte hexadecimal value */
char *GetHexValue(char *commandline,u32 *value)
{
    int i;

    if(commandline[0] == '\0') {
        SerialOutputString(__FUNCTION__ ": zero length string\n");
        return(NULL);
    }
    *value=0x00;
    if (MyStrNCmp(commandline, " 0x", 3) == 0) {
        commandline += 3;
    } else if(commandline[0] == ' ') {
        commandline += 1;
    } else {
        SerialOutputString(__FUNCTION__ ":value not hexdecimal\n");
        return NULL;
    }
    SerialOutputString("char: ");
    SerialOutputString(commandline);
    SerialOutputByte(' ');
    for (i=0; i<8;i++) {
        if (*commandline == '\0') {
            SerialOutputString(__FUNCTION__ ":hex value not 32 bits\n");
            return NULL;
        } else {
            if (MyIsXDigit(*commandline) == 0) {
                SerialOutputString("hex value contains invalid characters\n");
                return NULL;
            } else {
                *value |= (u32) MyXDigitValue(*commandline) << ((7-i)*4);
            }
        }
        commandline++;
    }
    SerialOutputString("hex 0x");
    SerialOutputHex(*value);
    SerialOutputByte('\r');
    return commandline;
} /* GetHexValue */
```

코드 891. GetHexValue() 함수의 정의

먼저 넘겨받은 commandline이 올바른 문자열을 가지는지 확인한다. 그렇지 않다면, 에러 메시지를 보인 후 NULL을 돌려준다. 바꾼 값을 가지는 변수 *value를 초기화시키고, commandline 변수가 ‘0x’³⁸⁴를

³⁸⁴ 제일 앞쪽에 space를 포함하고 있으므로, 비교하고자 하는 문자열의 길이는 3이다.

가지는지 검사한다. 만약 같다면, commandline 변수를 3증가시켜서, ‘0x’이후의 문자를 가르키도록 만든다. 그렇지 않다면, commandline[0]이 spece(=“ ”)와 같은지 비교해서, 같다면 commandline 변수를 하나 증가 시켜준다. 앞의 두가지 사항이 없다면, 에러를 출력하고 NULL을 돌려준다.

앞에서 넘겨받은 commandline을 출력하고, for loop를 돌면서 8자리의 문자로 표현된 값을 decimal 값으로 바꾼다. 먼저 commandline[]이 ‘\0’으로 이미 끝이 났는지를 검사한다. 그렇다면, 에러 메시지를 출력하고 NULL을 돌려준다. 그렇지 않다면, commandline[]이 hexadecimal을 가지는지를 검사한다. 그렇지 않다면, 역시 에러 메시지를 보여준 후 NULL을 돌려준다. 위의 두가지 조건이 다 아니라면, 이전 hexadecimal을 decimal 값으로 변형하기 위해서 MyXDigitValue() 함수를 사용해서 값을 변환한다. 변환된 값은 각 bit position에 맞도록 shift되어 value가 가르키는 값에 OR된다. 한 문자의 처리를 마쳤다면, 다음 commandline 문자의 처리를 위해서 commandline은 증가시킨다. 다 끝났다면, SerialOutputXXX()와 같은 함수를 호출해서 연산의 결과를 화면에 나타내고, 다음번 처리를 위해서 commandline을 돌려주고 복귀한다.

위에서 보았듯이 commandline 상에서 준 문자열이 메모리를 설정하는 레지스터의 내용에 정해진 순서대로 들어가기 때문에, 사용자는 clock이라는 명령을 사용하게 될 때는 순서에 유념해야 할 것이다.

17.6.3.3. Download Command

Download command는 MyStrNCmp()를 호출할 때, ‘download’라고 명시해서 뒤에 하나의 space까지를 포함하고 있다. 따라서, download 이후에 주어지는 파라미터 값에 의해서 호출의 결과가 달라질 것이다. Download() 함수의 정의는 main.c에 아래와 같이 되어 있다.

```
void Download(char *commandline)
{
    u32 startAddress = 0;
    int bufLen;
    int *numRead = 0;

    if(MyStrNCmp(commandline, "blob", 4) == 0) {
        /* download blob */
        startAddress = BLOB_RAM_BASE;
        bufLen = blob_status.blockSize - BLOB_BLOCK_OFFSET;
        numRead = &blob_status.blobSize;
        blob_status.blobType = fromDownload;
    } else if(MyStrNCmp(commandline, "kernel", 6) == 0) {
        /* download kernel */
        startAddress = KERNEL_RAM_BASE;
        bufLen = blob_status.blockSize - KERNEL_BLOCK_OFFSET;
        numRead = &blob_status.kernelSize;
        blob_status.kernelType = fromDownload;
    } else if(MyStrNCmp(commandline, "ramdisk", 7) == 0) {
        /* download ramdisk */
        startAddress = RAMDISK_RAM_BASE;
        bufLen = blob_status.blockSize - RAMDISK_BLOCK_OFFSET;
        numRead = &blob_status.ramdiskSize;
        blob_status.ramdiskType = fromDownload;
    } else {
        SerialOutputString("*** Don't know how to download \n");
        SerialOutputString(commandline);
        SerialOutputString("\n");
        return;
    }
}
```

코드 892. Download() 함수의 정의

Download() 함수는 BLOB과 kernel 및 RAM disk를 download받는 역할을 수행한다. 먼저, cmdline 변수에 blob이 있는 경우에는 startAddress에 BLOB_RAM_BASE(=0xc1000000)로 두고, bufLen을 blob_status.blockSize - BLOB_BLOCK_OFFSET(=0x00000000)으로 둔다. Serial을 통해서 읽어야 할 byte의 수를 numRead에 두기 위해서 blob_status.blobSize를 가르키도록 만들고, blob_status.blobType에는 serial을 통해서 downloading한다는 것을 나타내기 위해서 fromDownload로 만든다. 나머지 downloading에 관련된 것은 아래에서 처리할 것이다. 여기서는 일단 필요한 파라미터 값만 설정한다.

Kernel을 downloading하기 위해서는 startAddress에 KERNEL_RAM_BASE(=0xC0008000)를 주고, bufLen에는 blob_status.blockSize - KERNEL_BLOCK_OFFSET(=0x00008000)을 둔다. numRead에는 blob_status.kernelSize의 주소를 주고, blob_status.kernelType은 fromDownload로 둔다.

RAM disk를 download하기 위해서는 startAddress에 RAMDISK_RAM_BASE(=0xC0800000)를 주고, bufLen에는 blob_status.blockSize - RAMDISK_BLOCK_OFFSET(=0x00800000)을 둔다. numRead에는 blob_status.ramdiskSize의 주소를 주고, blob_status.ramdiskType에는 fromDownload로 둔다. 나머지 경우에 대해서는 잘못된 download 명령으로 간주한다.

```

SerialOutputString("Switching to ");
PrintSerialSpeed(blob_status.downloadSpeed);
SerialOutputString(" baud\n");
SerialOutputString("You have 60 seconds to switch your terminal emulator to the same speed and\n");
SerialOutputString("start downloading. After that \" PACKAGE \" will switch back to 9600 baud.\n");
SerialInit(blob_status.downloadSpeed);
*numRead = UUDecode((char *)startAddress, bufLen);
SerialOutputString("\n(Please switch your terminal emulator back to 9600 baud)\n");
if(*numRead < 0) {
    /* something went wrong */
    SerialOutputString("**** Uudecode receive failed\n");

    /* reload the correct memory */
    Reload(commandline);
    SerialInit(baud9k6);
    return;
}
SerialOutputString("Received ");
SerialOutputDec(*numRead);
SerialOutputString(" (0x");
SerialOutputHex(*numRead);
SerialOutputString(") bytes.\n");
SerialInit(baud9k6);
}

```

코드 893. Download() 함수의 정의(계속)

이전 실제적으로 downloading을 할 차례이다. 먼저 현재의 serial speed를 보여주도록 한다. Download speed에는 앞에서 이미 설정한 115200 bps가 있으므로, 이것을 이용해서 serial을 다시 설정하도록 한다(SerialInit()). 설정이 끝났다면, UUDecode() 함수를 사용해서 원하는 만큼을 downloading 받도록 한다. UUDecode() 함수는 startAddress에서 시작해서 bufLen만큼의 크기를 downloading해 줄 것이다. 이것을 마치고나면, 다시 원래의 serial speed(9600 bps)로 바꾸기 위해서 다시 한번 SerialInit() 함수가 호출될 것이다. 만약 downloading중에 문제가 발생했다면, 원래의 이미지를 다시 load하기 위해서 Reload() 함수가 호출될 것이다. 성공적으로 downloading이 되었다면, downloading된 실제의 크기를 화면상에 보여줄 것이다.

```

int UUDecode(char *bufBase, int bufLen)
{
/* Receives and decodes an incoming uuencoded stream. Returns the number of
bytes put in the buffer on success, or -1 otherwise. */

```

```

int n, linesReceived = 0;
char ch, *p;
int bytesWritten = 0, retries = 0;
char buf[INT_BUF_SIZE];

/* Search for header line. We don't care about the mode or filename */
retries = 0;
do {
    SerialInputString(buf, sizeof(buf), 6);
    TEST_MAX_RETRIES;
} while (MyStrNCmp(buf, "begin ", 6) != 0);

```

코드 894. UUDecode() 함수의 정의

UUDecode() 함수는 uucodec.c에 정의된 함수이다. 원래 UUDecode와 UUEncode를 합쳐서 Unix-to-Unix copy와 같은 곳에서 사용하기 위한 것으로 여기서는 binary file을 전송하는데 사용하고 있다. 먼저 input을 계속 읽어서, begin이라는 단어가 나오는지를 확인한다. 사용하는 buffer의 크기는 INT_BUF_SIZE (=1024 bytes)이다.

```

/* for each input line */
for (;;) {
    if (SerialInputString(p = buf, sizeof(buf), 2) == 0) {
        SerialOutputString("\n*** Short file. Aborting\n");
        return -1;
    }
    /* Status print to show where we are at right now */
    if((linesReceived++ & 0x007F) == 0) {
        SerialOutputByte('.');
    }
    /*
     * `n' is used to avoid writing out all the characters
     * at the end of the file.
     */
    if ((n = DEC(*p)) <= 0)
        break;
}

```

코드 895. UUDecode() 함수의 정의(계속)

For loop를 무한번 돌면서 계속 serial을 통해서 downloading을 받도록 한다. 이미 앞에서 header를 읽었으므로, 여기서는 데이터가 될 것이다. 만약 download할 것이 없다면, 잘못된 file을 명시한 것인으로 에러 메시지를 보여주고 -1을 돌려준다. 읽어들인 line수가 특정 값(=0x007F)과 같다면, 진행 상황을 보여주기 위해서 “.”을 serial을 통해서 보여준다. DEC() 매크로는 하나의 문자를 decoding하는 것으로, 아래와 같이 구현된다. 관련된 매크로의 정의를 다 둑어서 보여주도록 하겠다.

```

#define INT_BUF_SIZE 1024
#define MAX_RETRIES 10

#define TEST_MAX_RETRIES do { \
    if(retries++ > MAX_RETRIES) { \
        SerialOutputString("\n*** Timeout exceeded. Aborting.\n"); \
        return -1; \
    } \
} while(0)

#define DEC(c) (((c) - ' ') & 077)           /* single character decode */
#define IS_DEC(c) (((c) - ' ') >= 0) && ((c) - ' ') <= 077 + 1)
/* #define IS_DEC(c) (1) */

```

```
#define OUT_OF_RANGE do {      \
    SerialOutputByte('\n');   \
    SerialOutputString(buf); \
    SerialOutputString("\n*** Received character out of range. Aborting.\n"); \
    return -1; \
} while(0)

#define PUT_CHAR(x) do {      \
    if(bytesWritten < bufLen) \
        bufBase[bytesWritten++] = x; \
} while(0)
```

코드 896. uucodec.c의 매크로 및 상수 정의

즉, DEC() 매크로는 문자(c)에서 ‘·사이의 값을 구한 후, 이 값을 다시 07³⁸⁵과 AND 시킨 값을 만든다. IS_DEC() 매크로는 제대로 한 문자(c)가 decoding이 되었는지를 확인하기 위해서, 문자(c)와 ‘·사이의 값이 0에서 077 + 1까지의 값을 가지는지를 확인하게 된다. OUT_OF_RANGE 매크로는 단순히 받은 문자가 범위를 벗어났다는 에러 메시지를 보여주기 위한 것이며, PUT_CHAR() 매크로는 buffer에 하나의 문자를 끝에 채워넣는 역할을 한다. 따라서, 앞에서와 같이 DEC() 매크로를 사용한 것은 현재 decoding하고 있는 문자가 제대로 된 값인지를 확인하는 부분이다.

```
for (++p; n > 0; p += 4, n -= 3)
    if (n >= 3) {
        if (!(IS_DEC(*p) && IS_DEC(*(p + 1)) &&
              IS_DEC(*(p + 2)) && IS_DEC(*(p + 3))))
            OUT_OF_RANGE;
        ch = DEC(p[0]) << 2 | DEC(p[1]) >> 4;
        PUT_CHAR(ch);
        ch = DEC(p[1]) << 4 | DEC(p[2]) >> 2;
        PUT_CHAR(ch);
        ch = DEC(p[2]) << 6 | DEC(p[3]);
        PUT_CHAR(ch);

    }
    else {
        if (n >= 1) {
            if (!(IS_DEC(*p) && IS_DEC(*(p + 1))))
                OUT_OF_RANGE;
            ch = DEC(p[0]) << 2 | DEC(p[1]) >> 4;
            PUT_CHAR(ch);
        }
        if (n >= 2) {
            if (!(IS_DEC(*(p + 1)) &&
                  IS_DEC(*(p + 2))))
                OUT_OF_RANGE;
            ch = DEC(p[1]) << 4 | DEC(p[2]) >> 2;
            PUT_CHAR(ch);
        }
        if (n >= 3) {
            if (!(IS_DEC(*(p + 2)) &&
                  IS_DEC(*(p + 3))))
                OUT_OF_RANGE;
            ch = DEC(p[2]) << 6 | DEC(p[3]);
            PUT_CHAR(ch);
        }
    }
}
```

³⁸⁵ 이것은 C언어에서 8진수를 나타내는 format이다. 따라서, binay로는 01110111(b)가 될 것이며, hexadecimal로는 0x770I 될 것이다. 즉, 한 문자 길이가 된다(8bit).

```
        PUT_CHAR(ch);
    }
}
```

코드 897. UUDecode() 함수의 정의(계속)

이 부분은 UUDecode() 함수에 있는 두개의 for loop중에서 내부에 있는 for loop에 대한 것을 보여준다. 여기서 사용하는 n이라는 변수는 앞에서 for loop를 진행하기 전에 한 문자를 읽어서, 얼마나 많이 decoding을 해야할 부분이 있는지를 알려주는 값이다. 실제 decoding하는 절차는 한번에 4개의 byte씩을 진행하게 된다. n이 3이상이라면, 3개의 문자가 모두 올바른 decode값을 가지는지 확인하고, 이 문자들을 decoding(DEC())해서 각각을 차례로 buffer에 넣게 된다. 나머지는 n이 각각 1보다 크거나 같은 경우와 2보다 크거나 같은 경우, 3보다 크거나 같은 경우로 나누어서 각각의 문자를 decoding해서 buffer에 저장하게 된다.³⁸⁶

```
}
SerialOutputByte('\n');
if (SerialInputString(p = buf, sizeof(buf), 2) == 0 || (MyStrNCmp(buf, "end", 3))) {
    SerialOutputString("*** No \"end\" line. Aborting.\n");
    return(-1);
}
return(bytesWritten);
} /* UUDecode */
```

코드 898. UUDecode() 함수의 정의(계속)

이전 loop를 돌면서 encoding된 data부분에 대한 처리는 마쳤다. 마지막으로, “end”가 끝에 위치하는지를 판단한 후, 그렇지 못하다면 제대로 파일을 전달 받지 못한 것이 되므로, 에러 메시지를 출력한 후 -1을 돌려준다. 성공적으로 수행을 마쳤을 경우, 최종적으로 돌려주는 값은 버퍼에 쓴 문자의 수가 될 것이다.

```
#if 0
/* ENC is the basic 1 character encoding function to make a char printing */
#define ENC(c) ((c) ? ((c) & 077) + ':' : '^')
void UUEncode(char *bufBase, int bufLen) {
    register int ch, n;
    register char *p;
    char buf[80];

    SerialOutputString("begin 644 testme.jdb\n");
    while (bufLen > 0) {
        n = (bufLen > 45) ? 45 : bufLen;
        MyMemCpyChar(buf, bufBase, n);
        bufBase += n;
        bufLen -= n;
        ch = ENC(n);
        SerialOutputByte(ch);
        for (p = buf; n > 0; n -= 3, p += 3) {
            ch = *p >> 2;
            ch = ENC(ch);
            SerialOutputByte(ch);
            ch = ((*p << 4) & 060) | ((p[1] >> 4) & 017);
            ch = ENC(ch);
            SerialOutputByte(ch);
            ch = ((p[1] << 2) & 074) | ((p[2] >> 6) & 03);
        }
    }
}
```

³⁸⁶ UUDecoding의 자세한 algorithm을 아직은 다 이해하고 있지 못하기에 여기까지만 보는 것으로 일단은 만족하길 바란다.

```

        ch = ENC(ch);
        SerialOutputByte(ch);
        ch = p[2] & 077;
        ch = ENC(ch);
        SerialOutputByte(ch);

    }

    SerialOutputByte('\n');

}

ch = ENC('\0');
SerialOutputByte(ch);
SerialOutputByte('\n');
SerialOutputString("end\n");
} /* UUEncode */
#endif

```

코드 899. UUEncode() 함수의 정의

UUEncode()는 파일을 upload하기 위해서 사용하는 UUDecode()의 반대 역할을 수행하는 함수이다. 현재는 사용될 이유가 없기 때문에, 일단 #if 0 ~ #endif 절로 선언되어 compile 시에 빠지게 되지만, 이해의 목적으로 덧붙여 보았다.

ENC()는 문자 단위로 encoding을 수행하는 매크로이다. 앞에서 decoding 시에 했던 반대 연산을 수행하게 된다. 먼저 문자가 있다면, 이 문자를 077과 AND 시킨 값에 다시 ‘.’를 더해 주고, 그렇지 않다면 “.”를 두도록 한다. 먼저 “begin ..”으로 시작하는 문자열을 먼저 serial을 통해서 내보내고, buffer를 45 문자의 크기 단위로 encoding하도록 만든다. 먼저 주어진 문자 만큼 씩(45나 혹은 남은 문자열의 크기)을 encoding을 할 buffer(buf)에 저장하고, 길이를 나타내는 n을 ENC() 매크로를 사용해서 encoding한 다음, serial을 통해서 전달한다. 나머지 복사된 buffer에 있는 문자는 3문자씩 encoding되어, serial을 통해서 전달된다. 이렇게 전달된 문자열은 ‘\n’에 의해서 분리되며, 다음번 loop가 남은 문자열에 대해서 수행된다. 데이터에 대한 encoding이 끝나면, ‘\0’을 encoding하게 되며, 다시 ‘\n’을 보낸 후, 마지막으로 ‘end\n’으로 encoding을 마치도록 한다.

더 진행하기에 앞서서 잠시 UUDecoding과 UUEncoding에 대해서 이론적인 이야기를 조금 더 하도록 하겠다. 원래 UUdecode와 UUencode는 네트워크 상에서 시스템들 간에 파일을 전송하기 위해서 encoding이나 decoding하는 목적으로 사용되는 일종의 utility 프로그램이다. 예전의 Unix 시스템에서 시스템간에 파일을 전송하기 위해서 사용한다고 해서 UU(Unix-to-Unix)라는 이름을 붙였다는 것은 이미 보았다. 그러나, 현재는 대부분의 운영체제에서 사용될 수 있도록 porting이 되어 있으며, 또한 쉽게 구할 수도 있다. 대부분의 전자우편에 대해서 첨부물을 전송하기 위한 방편으로도 사용되고 있으며, 기본적으로 text(여기서는 영어문자라는 의미로 사용한다.)가 아닌 binary로 데이터를 보내고자 할 경우에 이것을 사용할 수 있다. 즉, binary로 된 데이터 파일을 7 bit으로 표기되는 ASCII로 만들어서 전달해 준다.

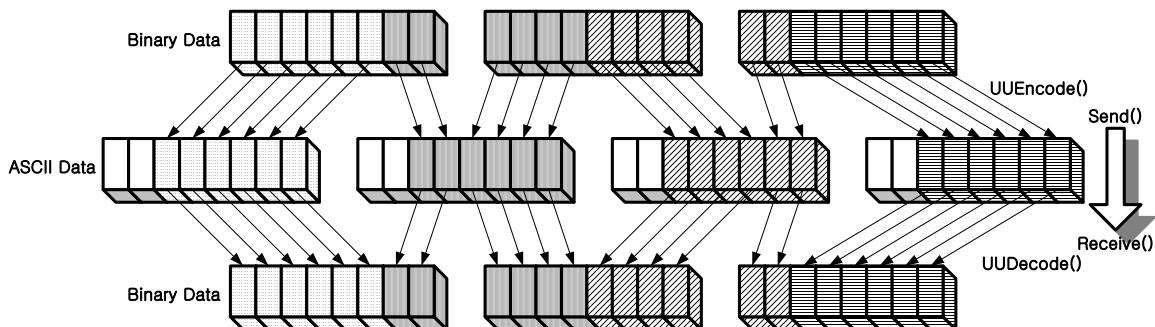


그림 125. UUEncode와 UUDecode

[그림 125]에서 보듯이 binary 형태로 된 data를 먼저 UUencode를 사용해서 ASCII 데이터로 만들어주고, 이를 다른 시스템으로 전달하게 되며, 받는 측에서는 받은 데이터를 다시 UUdecode를 사용해서 decoding해서 binar 형태로 복원해 주게 되는 것이다.

17.6.3.4. Flash Command

Flash command의 처리는 Flash() 함수가 맡고 있다. 정의는 main.c에 있으며, 메모리에 있는 데이터 block을 다시 flash 메모리에 write하는 일을 한다. write되는 내용에는 BLOB, 커널, RAM 디스크 등이다. 아래와 같다.

```
void Flash(char *commandline)
{
    u32 startAddress = 0;
    tBlockType block;
    int numBytes = 0;
    int maxSize = 0;

    if(MyStrNCmp(commandline, "blob", 4) == 0) {
        startAddress = BLOB_RAM_BASE;
        block = blBlob;
        numBytes = blob_status.blobSize;
        maxSize = BLOB_LEN;
        if(blob_status.blobType == fromFlash) {
            SerialOutputString("**** No blob downloaded\n");
            return;
        }
        SerialOutputString("Saving blob to flash ");
    } else if(MyStrNCmp(commandline, "kernel", 6) == 0) {
        startAddress = KERNEL_RAM_BASE;
        block = blKernel;
        numBytes = blob_status.kernelSize;
        maxSize = KERNEL_LEN;
        if(blob_status.kernelType == fromFlash) {
            SerialOutputString("**** No kernel downloaded\n");
            return;
        }
        SerialOutputString("Saving kernel to flash ");
    } else if(MyStrNCmp(commandline, "ramdisk", 7) == 0) {
        startAddress = RAMDISK_RAM_BASE;
        block = blRamdisk;
        numBytes = blob_status.ramdiskSize;
        maxSize = INITRD_LEN;
        if(blob_status.ramdiskType == fromFlash) {
            SerialOutputString("**** No ramdisk downloaded\n");
            return;
        }
        SerialOutputString("Saving ramdisk to flash ");
    } else {
        SerialOutputString("**** Don't know how to flash \n");
        SerialOutputString(commandline);
        SerialOutputString("\n");
        return;
    }
    if(numBytes > maxSize) {
        SerialOutputString("**** Downloaded image too large for flash area\n");
        SerialOutputString("**** (0x");
        SerialOutputHex(numBytes);
        SerialOutputString(" downloaded, maximum size is 0x");
    }
}
```

```

        SerialOutputHex(maxSize);
        SerialOutputString(" bytes)\n");
        return;
    }
    EraseBlocks(block);
    SerialOutputByte(' ');
    WriteBlocksFromMem(block, (u32 *)startAddress, numBytes);
    SerialOutputString(" done\n");
}

```

코드 900. Flash() 함수의 정의

Flash() 함수가 받는 commandline은 앞에서 이미 “flash”가 제거된 문자열이다. 따라서, 받아들은 문자열이 “blob”, “kernel”, “ramdisk”인지에 따라서, flash에 write할 내용이 달라지게 된다. Write를 하기 위해서는 먼저 flash 메모리에서 해당 데이터 block을 지우고(EraseBlocks()), 새로운 block을 쓰게 된다(WriteBlocksFromMem()). EraseBlocks() 함수가 필요로 하는 block 변수는 각각에 대해서 blBlob, blKernel, blRamdisk로 정해지며, flash.h에 enum 변수로 정해져 있다. EraseBlocks() 함수 자체는 flash.c에 있다. 또한, WriteBlocksFromMem() 함수가 필요로 하는 argument로는 어떤 종류의 데이터 block에 대한 것인가를 나타내는 block 변수와 그 block의 시작 주소(startAddress) 및, 써야하는 크기(numbytes)가 된다. 그외의 다른 경우에 대해서는 잘못된 flash 명령이라는 것을 보여주게 되며, 지우거나 쓰기전에 항상 최대로 flash에 write를 해야 하는 크기가 올바른 값을 가지는지 확인하게 된다. WriteBlocksFromMem() 함수도 역시 flash.c에 정의되어 있다.

각각의 경우에 대해서 startAddress가 가지는 값과, 써야하는 크기, 그리고 최대 값은 다음과 같이 볼 수 있다. 즉, BLOB의 경우에는 BLOB_RAM_BASE(=0xC1000000), blob_status.blobSize, BLOB_LEN(=0x10000)이 되며, 커널의 경우에는 KERNEL_RAM_BASE(=0xC0008000), blob_status.kernelSize, KENREL_LEN(=0xc0000) 가 되며, RAM 디스크의 경우에는 RAMDISK_RAM_BASE(=0xC0800000), blob_status.ramdiskSize, INITRD_LEN(=0x280000)이 된다. 즉, 이것은 이미 download command에서 downloading을 수행해서 각각의 image를 가지고 있는 주소가 된다.

```

void EraseBlocks(tBlockType which)
{
    char      *thisBlock;
    int       numBlocks, i;

    switch(which) {
    case blBlob:
        thisBlock = (char *)BLOB_START;
        numBlocks = NUM_BLOB_BLOCKS;
        break;
    case blKernel:
        thisBlock = (char *)KERNEL_START;
        numBlocks = NUM_KERNEL_BLOCKS;
        break;
    case blRamdisk:
        thisBlock = (char *)INITRD_START;
        numBlocks = NUM_INITRD_BLOCKS;
        break;
    default:
        /* this should not happen */
        return;
    }
    for(i = 0; i < numBlocks; i++, thisBlock += MAIN_BLOCK_SIZE) {
        SerialOutputByte('.');
        led_toggle();
        if(EraseOne(thisBlock) & STATUS_ERASE_ERR != 0) {
            SerialOutputString("\n*** Erase error at address 0x");

```

```

        SerialOutputHex((u32)thisBlock);
        SerialOutputByte('\n');
        return;
    }
}
} /* EraseBlocks */

```

코드 901. EraseBlocks() 함수의 정의

EraseBlocks() 함수는 넘어오는 bBlockType의 which 값에 따라서 switch문으로 분기한다. 즉, blBlob인 경우에는 BLOB 자체의 이미지를 지우라는 것이며, blKernel인 경우에는 kernel을, blRamdisk인 경우에는 RAM disk가 가진 block을 지우라는 뜻이다. 각각에 대해서 시작 주소와 지워야 할 block의 수를 thisBlock과 numBlocks 변수로 나타낸다. 이 변수들에 들어가는 내용은 코드에서 보듯이 XXX_START와 NUM_XXX_BLOCKS로 나타내진다. 실질적인 flash 메모리에 대한 지우기는 EraseOne() 함수가 처리한다. 즉, 지워야 할 block의 수 만큼 시작주소부터 한번에 한 block씩 지워 나가게 된다. 여기서 한번에 지우는 단위가 되는 MAIN_BLOCK_SIZE의 값은 $32768 * 4 (= 32K * 4 = 128K)$ Bytes가 된다.

```

static u32 EraseOne(const char *whichOne)
{
/* Routine to erase one block of flash */
    volatile u32 *writeMe = (u32 *)whichOne;
    u32      result;
#ifndef SHANNON || defined NESA
/* SHANNON이나 NESA가 정의된 경우에 실행되는 코드가 이곳에 있다.*/
    ...
    return 0;
#else
    *writeMe = data_to_flash(ERASE_SETUP);
    *writeMe = data_to_flash(ERASE_CONFIRM);
    do {
        *writeMe = data_to_flash(STATUS_READ);
        result = data_from_flash(*writeMe);
    } while((~result & STATUS_BUSY) != 0);
    *writeMe = data_to_flash(READ_ARRAY);
    return result;
#endif
} /* EraseOne */

```

코드 902. EraseOne() 함수의 정의

EraseOne() 함수가 넘겨받는 값은 지우기를 할 block의 시작 주소이다. Flash를 지우기 위한 절차는 다음과 같다. 먼저 setup 명령을 내려서(ERASE_SETUP) 지우기를 시작할 준비를 하도록 만들고, confirm으로 ERASE_CONFIRM을 flash에 지우라는 명령으로 내린다. 이젠 flash가 지우기를 끝내기를 기다리면 될 것이다. 이 것은 do{} while() loop를 돌면서 다음과 같은 일을 해주면 될 것이다. 즉, flash가 준비 되었는지를 상태를 읽는 명령으로 STATUS_READ를 하고, flash로 부터 상태를 읽어서, result에 보관한다. 이 값을 NOT한 것에 STATUS_BUSY를 AND 시켜서, 이 값이 0이 아닌동안 대기하면 된다. 즉, flash가 명령을 처리하는데 걸리는 시간적인 delay를 주고자 하는 것이다. 여기서 사용하고 있는 data_to_flash()와 data_from_flash()는 모두 flashasm.S에 정의되어 있다.³⁸⁷

```

void WriteBlocksFromMem(tBlockType type, const u32 *source, int length)
{
    volatile u32      *flashBase;

```

³⁸⁷ LART board에서 사용하는 data_to_flash()와 data_from_flash()만이 현재는 정의된 상태이다. 나머지에 대해서는 확인된 것이 없으며, 그냥 subroutine을 return할 뿐이다. 이 함수들에 대한 분석은 다음으로 미룬다. 아직 코드에 대해서 명확히 파악하지 못했기 때문이다.

```

u32          result;
int           maxLength, i;

...
if((u32)source & 0x03) {
    SerialOutputString("*** Source is not on a word boundary: 0x");
    SerialOutputHex((u32)source);
    SerialOutputByte('\n');
    return;
}
if(length & 0x03)
    length += 0x04;
length &= ~((u32) 0x03);
switch(type) {
case blBlob:
    flashBase = (u32 *)BLOB_START;
    maxLength = BLOB_LEN;
    break;
case blKernel:
    flashBase = (u32 *)KERNEL_START;
    maxLength = KERNEL_LEN;
    break;
case blRamdisk:
    flashBase = (u32 *)INITRD_START;
    maxLength = INITRD_LEN;
    break;
default:
    /* this should not happen */
    return;
}
if(length > maxLength)
    length = maxLength;

```

코드 903. WriteBlocksFromMem() 함수의 정의

WriteBlocksFromMem() 함수가 넘겨받는 값은 쓰려고 하는 block의 type(blBlob, blKernel, blRamdisk)와 쓰려고하는 데이터의 주소(source), 그리고 길이(length)이다. 만약 length가 4byte단위로 정렬이 되지 않은 경우에는 정렬해(alignment) 준다. 각각의 쓰려고 하는 block의 type에 따라, flash의 기본 주소(flashBase)는 BLOB_START, KERNEL_START, INITRD_START의 값을 가질 것이며, 이 block이 가지는 최대 길이는 각각 BLOB_LEN, KERNEL_LEN, INITRD_LEN이 될 것이다. 만약 최대 길이보다 큰 크기를 가지는 데이터를 쓰고자 할 때는 쓰려고 하는 데이터의 길이를 허용하는 최대 길이(maxLength)로 맞춘다.

```

...
for(i = 0; i < length; i+= 4, flashBase++, source++) {
    if((i % MAIN_BLOCK_SIZE) == 0) {
        SerialOutputByte('.');
        led_toggle();
    }
    *flashBase = data_to_flash(PGM_SETUP);
    *flashBase = *source;
...
do {
    *flashBase = data_to_flash(STATUS_READ);
    result = data_from_flash(*flashBase);
} while((~result & STATUS_BUSY) != 0);

*flashBase = data_to_flash(READ_ARRAY);

```

```

        if((result & STATUS_PGM_ERR) != 0 || *flashBase != *source) {
            SerialOutputString("\n*** Write error at address 0x");
            SerialOutputHex((u32)flashBase);
            SerialOutputByte('\n');
            return;
        }
    }
...
} /* WriteBlocksFromMem */

```

코드 904.. WriteBlocksFromMem() 함수의 정의(계속)

이전 한번에 4 bytes 단위로 flash에 쓰는 과정이다. MAIN_BLOCK_SIZE 만큼 쓰게 되면, ':' 가 나타나며 LED가 점멸할 것이다(led_toggle()). 먼저 PGM_SETUP을 명령을 flash에 내린다. 이때, 내려준 명령에 대해서 주소를 data_to_flash()가 return해 주면, 이곳에 다시 source가 가르키는 내용을 넣어주도록 한다. 이전 do{} while() loop를 돌면서 flash의 상태를 읽어 BUSY상태가 아닌 경우가 될 때까지 기다려주고, 다시 READ_ARRAY 명령을 flash에 내린다. 만약 쓰기기에 오류가 있거나, 쓴 내용(*flashBase)과 source가 가르키고 있는 내용이 다르다면 error 메시지를 출력한다. 이 for loop는 해당하는 길이 만큼을 다 쓸 때까지 반복적으로 수행한다.

Flash에 대한 명령 및 status는 다음과 같이 요약해서 볼 수 있다. 앞에서 flash를 지울 때도 사용했던 명령과 status를 같이 보도록 하자.

Command	Value	Description
READ_ARRAY	0x00FF00FF	Flash가 준비된 상태(즉, 다른 명령을 읽기 위해서 enable)가 되도록 만든다..
ERASE_SETUP	0x00200020	하나의 block에 대한 erase를 준비하라. 뒤에 ERASE_CONFIRM이 따라오게 된다.
ERASE_CONFIRM	0x00D000D0	Erase가 되도록 요청한다.
PGM_SETUP	0x00400040	다음의 flash에 대한 프로그램(program) 동작을 위해서 반복하라.
STATUS_READ	0x00700070	성공적으로 동작을 완료한 경우에 상태를 읽도록 한다. 즉, block에 대한 erase나, program, 혹은 lock bit에 대한 설정을 한 후에 상태 값을 읽어오기 위해서 사용한다.
STATUS_CLEAR	0x00500050	상태값을 clear시키기 위해서 사용한다.
STATUS_BUSY	0x00800080	현재 flash가 BUSY상태임을 나타낸다.
STATUS_ERASE_ERR	0x00200020	앞서 실행했던 flash에 대한 erase에 에러가 있었다.
STATUS_PGM_ERR	0x00100010	앞서 실행했던 flash에 대한 program에 에러가 있었다.

표 113. Flash에 대한 명령 및 status 값에 대한 정의

위의 도표에서 보여준 것 이외에도 보드의 종류에 따라서 다른 flash 명령들이 존재하지만 그것은 여기서 생략하기로 하겠다. 직접 보고자 하는 사람은 flash.c 파일을 참조하기 바란다.

17.6.3.5. Help Command & Status Command

Help command는 BLOB를 사용하는 방법을 묻는 명령어이다. 단순히 PrintHelp() 함수만 호출해서 간략하게 사용하는 방법에 대한 것을 보여주는 역할을 할 뿐이다. 사용하는 방법만을 잠시 보기 위해서 여기에 인용해 보기로 한다.

```

void PrintHelp(void)
{
    SerialOutputString("Help for " PACKAGE " " VERSION " the LART bootloader\n");
    SerialOutputString("The following commands are supported:\n");
    SerialOutputString("* boot [kernel options]           Boot Linux with optional kernel options\n");
    SerialOutputString("* clock PPCR MDCNFG MDCAS0 MDCAS1 MDCAS2\n");
    SerialOutputString("Set the SA1100 core clock and DRAM timings\n");
}

```

```

SerialOutputString("                                     (WARNING: dangerous command!)\n");
SerialOutputString("* download {blob|kernel|ramdisk} Download blob/kernel/ramdisk image to RAM\n");
SerialOutputString("* flash {blob|kernel|ramdisk}   Copy blob/kernel/ramdisk from RAM to flash\n");
SerialOutputString("* help                           Get this help\n");
SerialOutputString("* reblob                         Restart blob from RAM\n");
SerialOutputString("* reboot                         Reboot system\n");
SerialOutputString("* reload {blob|kernel|ramdisk} Reload blob/kernel/ramdisk from flash to RAM\n");
SerialOutputString("* reset                           Reset terminal\n");
SerialOutputString("* speed                            Set download speed\n");
SerialOutputString("* status                           Display current status\n");
}

```

코드 905. PrintHelp() 함수의 정의

즉, LART가 제공하는 BLOB에 대한 package정보와 버전 정보를 보여주며, 각각의 명령어를 어떻게 사용하는지와 이 명령어가 어떠한 역할을 하는지를 보여주고 있다. 따로이 설명할 부분은 없다.

Status command 역시 현재의 BLOB의 상태를 보여주는 역할만 하고 있다. 즉, PrintStatus() 함수를 호출해서 화면에 display하는 일이다. 정의는 PrintHelp() 함수와 마찬가지로 main.c에 아래와 같이 되어 있다.

```

void PrintStatus(void)
{
    /* BLOB의 package정보와 버전 정보를 보여준다.*/
    SerialOutputString("Bootloader      : PACKAGE "\n");
    SerialOutputString("Version        : VERSION "\n");
    SerialOutputString("Running from  : ");
    /* BLOB의 현재 실행 모드를 보여준다.*/
    if(RunningFromInternal())
        SerialOutputString("internal");
    else
        SerialOutputString("external");
    /* Flash에 대한 상태 정보를 보여준다.*/
    SerialOutputString(" flash\nBlocksize     : 0x");
    SerialOutputHex(blob_status.blockSize);           /* Flash의 block size */
    SerialOutputString("\nDownload speed: ");                /* Download speed */
    PrintSerialSpeed(blob_status.downloadSpeed);
    SerialOutputString(" baud\n");
    /* 메모리에 있는 Blob의 상태를 보여준다.*/
    SerialOutputString("Blob          : ");
    if(blob_status.blobType == fromFlash) {
        SerialOutputString("from flash\n");
    } else {
        SerialOutputString("downloaded, ");
        SerialOutputDec(blob_status.blobSize);
        SerialOutputString(" bytes\n");
    }
    /* 메모리에 있는 Kernel의 상태를 보여준다.*/
    SerialOutputString("Kernel        : ");
    if(blob_status.kernelType == fromFlash) {
        SerialOutputString("from flash\n");
    } else {
        SerialOutputString("downloaded, ");
        SerialOutputDec(blob_status.kernelSize);
        SerialOutputString(" bytes\n");
    }
}

```

```

/* 메모리에 있는 RAM disk의 상태를 보여준다.*/
SerialOutputString("Ramdisk      : ");
if(blob_status.ramdiskType == fromFlash) {
    SerialOutputString("from flash\n");
} else {
    SerialOutputString("downloaded, ");
    SerialOutputDec(blob_status.ramdiskSize);
    SerialOutputString(" bytes\n");
}
}

```

코드 906. PrintStatus() 함수의 정의

PrintStatus() 함수에서 볼 수 있는 것은 blob_status 구조체가 중심이 되어, 현재의 모든 BLOB의 상태를 구성하고 있다는 점이며, 각각의 모듈(BLOB, kernel, RAM disk)별로 이러한 정보가 유지 된다는 점이다. 또한 이러한 정보 중에서 현재 메모리에 가지고 있는 image가 download된 것인지, 아니면 flash로부터 읽어온 것인가도 보여주고 있다는 것이다.

17.6.3.6. Reblob Command

Reblob 명령은 RAM에 download하거나 flash로부터 가져온 BLOB을 새로이 시작하기 위한 것이다. 처리는 Reblob() 함수가 하고 있으며, 정의는 main.c에 아래와 같이 되어 있다.

```

void Reblob(void)
{
    void (*blob)(void) = (void (*)(void))BLOB_RAM_BASE;

    SerialOutputString("Restarting blob from RAM...\\n\\n");
    msleep(500);
    blob();
}

```

코드 907. Reblob() 함수의 정의

Reblob() 함수는 잠시간의 delay를 준다음, 메모리에 있는 BLOB의 이미지를 새로 수행하는 함수이다. 단지 download되거나 flash로 부터 read한 BLOB으로 jump만 한다. 여기서 사용되는 msleep() 함수는 time.c에 아래와 같이 정의되어 있다.

```

void msleep(unsigned int msec)
{
    u32 ticks, start, end;
    int will_overflow = 0;
    int has_overflow = 0;
    int reached = 0;

    if(msec == 0)
        return;
    ticks = (TICKS_PER_SECOND * msec) / 1000;
    start = TimerGetTime();
    /* this could overflow, but it nicely wraps around which is
     * exactly what we want
     */
    end = start + ticks;
    /* detect the overflow */
    if(end < start) {
        TimerClearOverflow();
        will_overflow = 1;
    }
}

```

```

do {
    if(will_overflow && !has_overflow) {
        if(TimerDetectOverflow())
            has_overflow = 1;
        continue;
    }
    if(TimerGetTime() >= end)
        reached = 1;
} while(!reached);
}

```

코드 908. msleep() 함수의 정의

msleep() 함수가 넘겨받는 값은 얼마동안 sleep을 할 것인가를 결정하는 값으로 millisecond단위 값이다. Millisecond단위로 tick의 개수를 계산해서 ticks에 둔다. 그리고, 시작 시간을 구해서(TimerGetTime()) start에 두도록 한다. 이전 앞에서 계산된 시간과 현재 시간을 더해서 end로 만들고, 만약 end가 start보다 작다면 overflow가 있었다는 말이되므로 overflow를 지우기 위해서 TimerClearOverflow() 함수를 호출한다. 또한 overflow가 발생할 것이라는 것을 나타내기 위해서 will_overflow변수를 1로 설정한다. 나머지는 do{} while() loop를 돌면서 현재 시간이 앞에서 계산된 시간(end)보다 크거나 같을 때까지 수행하는 것이다. Loop내에서 overflow가 발생할지도 모르는 경우에는 TimerDetectOverflow()를 호출해서 실제로 overflow가 있었는지를 검출하게 되며, 이때 has_overflow를 1로 설정한다. 따라서, 이 loop에서 주어진 시간 만큼 대기하면서 수행을 delay시켜주는 것이다.

17.6.3.7. Reboot Command

SA1100의 Reset Controller Register들로는 RSRR(Reset Controller Software Reset Register)과 RCSR(Reset Controller Status Register)이 있다. RSRR은 최하위에 하나의 software reset bit을 가지고 있으며, 이 bit이 설정이 되면, SA1100을 reset시켜주는 역할을 한다. RCSR은 CPU에 의해서 reset의 마지막 원인을 결정하기 위해서 사용되는 값을 가지는 레지스터로 크게 다음과 같은 reset 상황을 SA1100은 지원한다.

- Hardware reset : hardware적으로 reset이 걸렸다.
- Software reset : software적으로 reset이 걸렸다.
- Watchdog reset : watchdog reset이 걸렸다.
- Sleep mode reset : sleep mode reset이 걸렸다.

각각의 reset 상황에 대해서 하위에서 부터 hardware, software, watchdog, sleep mode의 순으로 한 bit씩 할당되어 있으며, reset후에는 hardware reset bit만이 1로 남으며, 나머지는 전부 0으로 clear된다. 이 register는 각각의 bit에 1을 write하는 것으로 bit를 clear할 수 있다. 각 bit에 있는 0이 의미하는 바는 해당하는 일이 일어나지 않았다는 것이며, 1은 해당하는 일이 발생했다는 말이다. 기타 다른 reserved된 bit은 의미가 없으며, 읽었을 때 0을 돌려줄 것이다.

```

void Reboot(void)
{
    SerialOutputString("Rebooting...\n\n");
    msleep(500);
    RCSR = 0;
    RSRR = 1;
}

```

코드 909. Reboot() 함수의 정의

Reboot command는 새로 booting을 시키는 명령이다. 잠시 시간을 delay시킨 후에 RCSR(=0x90030004 : Reset Controller Status Register)에는 0을 주고, RSRR(=0x90030000 : Reset Software Reset Register)에는 1을 써서, software reset을 걸어준다. 각각의 register에 대한 정의는 ~/include/asm-arm/arch-sa1100/SA-1100.h 있다.

17.6.3.8. Reload Command

Reload command는 앞에서 이미본 Reload() 함수를 수행한다. 넘겨받은 argument값에 따라서, BLOB, kernel, RAM disk를 새로 load하는 역할을 수행한다.

17.6.3.9. Reset Command

Reset command는 terminal의 speed를 원래의 상태로 복귀하는 명령이다. ResetTerminal() 함수를 호출해서 처리한다. 정의는 main.c에 있으며,

```
void ResetTerminal(void)
{
    int i;

    SerialInit(baud9k6);
    SerialOutputString("~-c");
    for(i = 0; i < 100; i++)
        SerialOutputByte('\n');
    SerialOutputString("~c");
}
```

코드 910. ResetTerminal() 함수의 정의

ResetTerminal() 함수는 SerialInit()를 호출해서 9600bps로 serial의 baud rate를 맞춰주는 역할만 한다. 나머지는 일종의 debugging에 해당하는 문자열을 보여주는 것 뿐이다.

17.6.3.10. Speed Command

Speed command는 download 속도를 설정해 주는 함수이다. SetDownloadSpeed() 함수를 호출해서 처리해 준다. SetDownloadSpeed() 함수는 main.c에 아래와 같이 정의되어 있다.

```
void SetDownloadSpeed(char *commandline)
{
    if(MyStrNCmp(commandline, "1200", 4) == 0) {
        blob_status.downloadSpeed = baud1k2;
    } else if(MyStrNCmp(commandline, "1k2", 3) == 0) {
        blob_status.downloadSpeed = baud1k2;
    } else if(MyStrNCmp(commandline, "9600", 4) == 0) {
        blob_status.downloadSpeed = baud9k6;
    } else if(MyStrNCmp(commandline, "9k6", 3) == 0) {
        blob_status.downloadSpeed = baud9k6;
    } else if(MyStrNCmp(commandline, "19200", 5) == 0) {
        blob_status.downloadSpeed = baud19k2;
    } else if(MyStrNCmp(commandline, "19k2", 4) == 0) {
        blob_status.downloadSpeed = baud19k2;
    } else if(MyStrNCmp(commandline, "38400", 5) == 0) {
        blob_status.downloadSpeed = baud38k4;
    } else if(MyStrNCmp(commandline, "38k4", 4) == 0) {
        blob_status.downloadSpeed = baud38k4;
    } else if(MyStrNCmp(commandline, "57600", 5) == 0) {
        blob_status.downloadSpeed = baud57k6;
    } else if(MyStrNCmp(commandline, "57k6", 4) == 0) {
        blob_status.downloadSpeed = baud57k6;
    } else if(MyStrNCmp(commandline, "115200", 6) == 0) {
        blob_status.downloadSpeed = baud115k2;
    } else if(MyStrNCmp(commandline, "115k2", 5) == 0) {
        blob_status.downloadSpeed = baud115k2;
    } else {
        SerialOutputString("*** Invalid download speed value \\"");
    }
}
```

```

        SerialOutputString(commandline);
        SerialOutputString("\n*** Valid values are:\n");
        SerialOutputString("**** 1200, 9600, 19200, 38400, 57600, 115200,\n");
        SerialOutputString("**** 1k2, 9k6, 19k2, 38k4, 57k6, and 115k2\n");
    }
    SerialOutputString("Download speed set to ");
    PrintSerialSpeed(blob_status.downloadSpeed);
    SerialOutputString(" baud\n");
}

```

코드 911. SetDownloadSpeed() 함수의 정의

넘겨받은 문자열을 비교해서(MyStrNCmp()) 이 값에 따라 blob_status.downloadSpeed를 결정해 준다. 들어갈 수 있는 download speed로는 1200, 9600, 19200, 38400, 57600, 115200 bps가 있으며, 해당하는 값이 없다면 에러 메시지를 보여줄 것이다. 또한 조정된 download speed가 얼마인지도 보여준다. 여기서 설정한 download speed는 나중에 download를 하게 될 때 사용될 것이므로, 실제적인 hardware의 설정은 이곳에서 찾을 수 없다. 단지 blob_status 구조체의 downloadSpeed 필드를 갱신하는 일만 해준다. 여기서 사용되는 PrintSerialSpeed() 함수도 main.c에 아래와 같은 정의를 가지고 있으며, 단순히 현재 설정한 serial의 속도를 보여주는 일만 한다.

```

void PrintSerialSpeed(eBauds speed)
{
    switch(speed) {
        case baud1k2:
            SerialOutputDec(1200);
            break;
        case baud9k6:
            SerialOutputDec(9600);
            break;
        case baud19k2:
            SerialOutputDec(19200);
            break;
        case baud38k4:
            SerialOutputDec(38400);
            break;
        case baud57k6:
            SerialOutputDec(57600);
            break;
        case baud115k2:
            SerialOutputDec(115200);
            break;
        default:
            SerialOutputString("(unknown)");
            break;
    }
}

```

코드 912. PrintSerialSpeed() 함수의 정의

넘겨받은 serial의 속도에 따라서 switch문을 통해 해당하는 speed별로 SerialOutputDec() 함수를 호출해서 화면에 나타내 준다.

이상에서 우린 SA1100의 boot loader로 사용된 BLOB에 대한 전체 과정을 살펴보았다. 즉, blob을 통해서 커널을 loading할 준비가 다 된 것이다. 커널은 flash에 저장되어 있거나, 혹은 serial을 통해서 downloading되어야 하며, 사용하게 될 RAM disk도 같은 방법을 통해서 사용할 수 있을 것이다. 초기에 BLOB를 flash에 쓰는 것은 Jflash와 같은 것을 사용하면 될 것이다. 이와 관련된 것은 여기서는 더 이상 논의 하지 않을 것이며, 이젠 커널로 들어가서 어떤 과정들이 일어나는지를 보도록 하자.

17.7. SA1100의 Booting

삼성전자의 WVP(Web Video Phone)의 Booting 순서를 추적하면서, 어떤 부분을 고쳐주어야 하는지를 실제로 코드를 보면서 설명하도록 하겠다. 먼저 커널의 전체 구조는 아래의 [그림]과 같다.

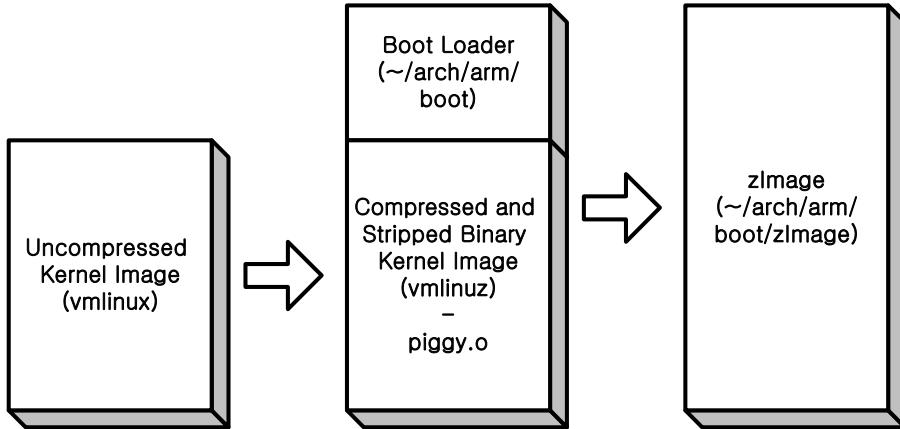


그림 126. 압축된 커널 이미지

따라서, 우리는 ~/arch/arm/boot이하의 부분에서 커널의 boot loader가 있음을 알 수 있다. 실제적으로 커널 boot loader의 역할을 해주는 것은 ~/arch/arm/boot/compressed/head.S이다. 시스템은 처음 부팅하면, ROM 영역이 있는 0x00000000을 접근해서, 우리가 만들어준 zImage를 찾게 된다. 일반적으로 부트 로더가 있어야 하겠지만, 우선은 ROM emulator같은 것을 이용해서 ROM에 직접 생성된 압축 커널 이미지를 그대로 write하여 사용했다. 찾은 커널을 적절한 위치(0xC0008000)로 loading을 해서 이곳에서부터 실행을 시작한다.

head.S를 바로 보기 전에 \${KERNEL_SRC}/arch/arm/boot/compressed/Makefile와 바로 위의 디렉토리에 생성될 vmlinux.lds 파일³⁸⁸을 먼저 보도록 하자. 두 파일은 boot이하에서 어떻게 boot loader가 생성될지와 커널과 생성된 boot loader가 어떤 형식으로 link될지를 결정해준다.

Makefile을 먼저 보면, 아래와 같은 코드로 되어 있음을 알 수 있다. 즉, 이 파일이 하는 역할은 커널의 컴파일 도중에 make zImage를 실행할게 될 때 실행될 make 명령의 script를 가진다.

```

...
HEAD          = head.o
...
ifeq ($(CONFIG_ARCH_SA1100),y)
OBJS          += head-sa1100.o setup-sa1100.o
ifeq ($(CONFIG_SA1100_NANOENGINE),y)
  OBJS += hw-bse.o
endif
endif
...
SEDFLAGS      = s/TEXT_START/$(ZTEXTADDR)/;s/LOAD_ADDR/$(ZRELADDR)/
...
all:          vmlinux

vmlinux: $(HEAD) $(OBJS) piggy.o vmlinux.lds
           $(LD) $(ZLDFLAGS) $(HEAD) $(OBJS) piggy.o $(LIBGCC) -o vmlinux
  
```

³⁸⁸ 이 파일은 vmlinux.lds.in을 통해서 생성된다.

```

$(HEAD):      $(HEAD:.o=.S)
              $(CC) $(AFLAGS) -traditional -c $(HEAD:.o=.S)

piggy.o: $(SYSTEM)
          $(OBJCOPY) -O binary -R .note -R .comment -S $(SYSTEM) piggy
          gzip $(GZFLAGS) < piggy > piggy.gz
          $(LD) -r -o $@ -b binary piggy.gz
          rm -f piggy piggy.gz
...
vmlinux.lds: vmlinux.lds.in
              @sed "$(SEDFLAGS)" < vmlinux.lds.in > $@
...

```

코드 913. \${KERNEL_SRC}/arch/arm/boot/compressed/Makefile

중요한 부분만을 따왔다. vmlinux를 만들기 위해서 \$(HEAD)와 \$(OBJS) 및 piggy.o와 vmlinux.lds가 필요하다. \$(HEAD)는 \$(HEAD)는 먼저 head.S와 CONFIG_ARCH_SA1100이 “y”로 선택되었을 때 필요한 head-sa1100.o, setup-sa1100.o가 필요하게 된다. 따라서, 이 파일들 역시 우리의 분석 대상이 될 것이다. piggy.o는 예전에 만들어진 vmlinux를 strip시켜서 symbol 정보 등을 제거해서 압축시킴으로써 만들어진다. vmlinux.lds는 vmlinux.lds.in에 의존(depend)하고 있으며, vmlinux.lds.in을 sed를 사용해서 “\$(SEDFLAGS)”을 거쳐서 만들어진다. 여기서 사용되는 \$(SEDFLAGS) 변수는 TEXT_START를 찾아서 \$(ZTEXTADDR)로 바꾸고, LOAD_ADDR를 찾아서 \$(ZRELADDR)로 바꾸라는 말이다. 즉, text의 시작주소와 커널의 초기 load되는 주소를 정해주는 것과 관련이 있다.

vmlinux.lds.in을 보도록 하자. 여기서 중요한 것은 SEDFLAGS와 ZTEXTADDR이다. ~/arch/boot/compressed에 아래와 같이 나와 있다.

```

OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = LOAD_ADDR;                                /* 처음에 커널이 load되는 주소 */
    _load_addr = .;
    . = TEXT_START;                               /* 커널의 Text(Code)영역이 있는 주소 */
    _text = .;
    .text : {
        _start = .;
        *(.start)
        *(.text)
        *(.fixup)
        *(.gnu.warning)
        *(.rodata)
        *(.glue_7)
        *(.glue_7t)
        input_data = .;
        piggy.o                                     /* 압축된 커널이 들어있는 장소 */
        input_data_end = .;
        . = ALIGN(4);
    }
    _etext = .;
    .data : {
        *(.data)
    }
    _edata = .;
    . = BSS_START;                                /* BSS 영역의 시작 */
    __bss_start = .;
    .bss : {

```

```

*(.bss)
}
_end = .;
.stack : {
    *(.stack)                                /* Stack 영역의 시작 */
}
.stab 0 : { *(.stab) }                      /* String Table의 주소 */
.stabstr 0 : { *(.stabstr) }
.stab.excl 0 : { *(.stab.excl) }
.stab.exclstr 0 : { *(.stab.exclstr) }
.stab.index 0 : { *(.stab.index) }
.stab.indexstr 0 : { *(.stab.indexstr) }
.comment 0 : { *(.comment) }                  /* Comment */
}

```

코드 914. vmlinux.lds.in

여기서 “.”이 나타내는 뜻은 현재의 주소를 의미한다. 즉, 첫번째 `=LOAD_ADDR`은 section의 시작의 첫 주소 부분을 LOAD_ADDR로 시작하라는 의미이다. “`_load_addr=.`”는 그 주소에서 `_load_addr`이 결정됨을 의미한다. 그리고, entry point로는 `_start`를 가지도록 ENTRY(`_start`)로 나타내 주었으며, 나머지는 각각의 section별로 어떤 것들을 가지고 있는지 보여준다. 이 Makefile의 LOAD_ADDR과 TEXT_ADDR의 input으로 들어가는 값은 `~/arch/arm/boot`에 있는 Makefile이다. 아래와 같은 값이 중요한 의미를 지닌다.

```

...
ifeq ($(CONFIG_ARCH_SA1100),y)
ZTEXTADDR      = 0xC0008000          /* WVP : 0xC0F00000 */
ZRELADDR       = 0xC0008000          /* WVP : 0xC0008000 */
...
export SYSTEM ZTEXTADDR ZBSSADDR ZRELADDR INITRD_PHYS INITRD_VIRT PARAMS_PHYS
...

```

즉, `CONFIG_ARCH_SA1100` “y”로 설정되었다면, ZTEXTADDR은 `0xC0008000`이 ZRELADDR에는 `0xC0008000` 들어간다. 삼성전자의 WVP의 경우에는 Load되는 주소를 `0xC0008000`으로 Text의 시작주소를 `0xC0F00000`으로 주어야 하므로 이곳을 수정해 주어야 할 것이다. 위에서 comment로 되어 있는 부분을 쓰도록 하자. 즉, 일단은 `0xC0008000`으로 첫번째 load가 이루어 질 것이며, 나중에 다시 `0xC0F00000`으로 압축이 해제된 Text가 들어가게 될 것이다. 위에서 필요한 변수들은 “export”를 써서 다른 Makefile에서 참조할 수 있도록 했다.

여기서 위에서 보여준 그림을 다시 조금 더 확장해 본다면, [그림 127]와 같이 될 것이다. 즉, 생성된 압축 커널은 초기에 `0xC0008000`으로 load되고, 이를 다시 `0xC0F00000`으로 커널을 압축을 해제한 후 옮겨준다.

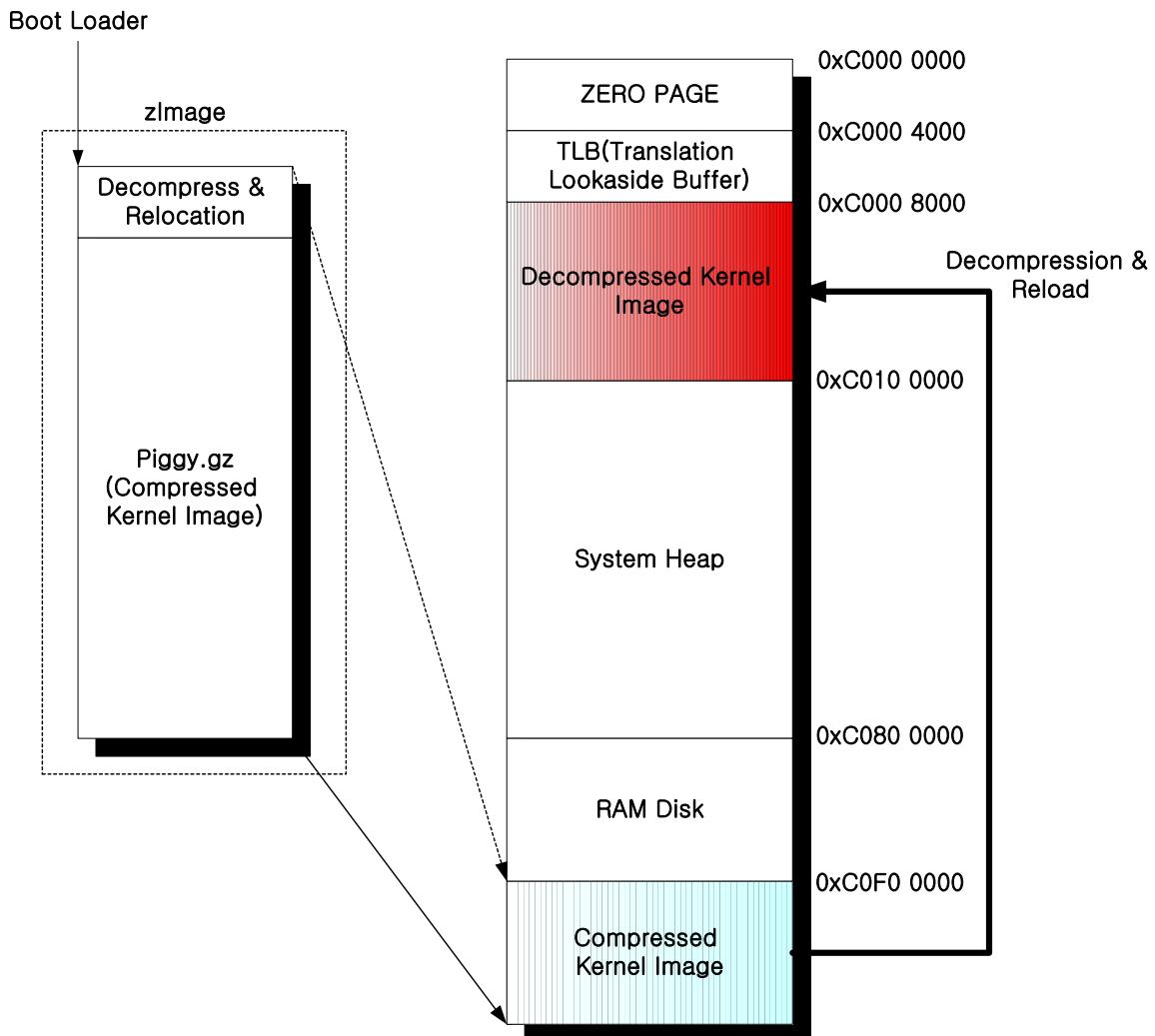


그림 127. Kernel Image의 loading과 decompression

이하에서 이러한 압축된 커널을 load하고, 다시 압축을 해제하는 부분에 대해서 찾을 수 있을 것이므로, 여기서는 이 그림만을 머리에 담도록 하자.

17.7.1. head.S의 분석

처음으로 booting이 되어서 실행되는 부분은 역시 ~/arch/boot/compressed/head.S이다. 코드는 아래와 같다. 이곳에서 하는 일은 적절히 압축된 커널의 압축을 해제해서 적당한 위치(0xC0F00000)에 두고 그곳으로 제어를 옮기는 일이다. 필요 없는 부분은 제거하고, 중요한 부분들을 중심으로 보도록 하겠다. 먼저 위에서 ENTRY()로 준 값이 _start이기에 부팅 시에 제거는 그곳으로 넘어갈 것이다.

```
/*
 * linux/arch/arm/boot/compressed/head.S
 */
#include <linux/config.h>
#include <linux/linkage.h>
...
.section ".start", #alloc, #execinstr
/*
 * sort out different calling conventions
 */
.align
```

```

start:
    .type    start,#function
    .rept    8                      /* 8번 반복하라. */
    mov     r0, r0
    .endr    /* 여기까지만 반복하라. */

    b      1f
    .word   0x016f2818            @ Magic numbers to help the loader
    .word   start                 @ absolute load/run zImage address
    .word   _edata                @ zImage end address
1:   mov     r7, r1                @ save architecture ID
    mov     r8, #0                 @ save r0
#endif CONFIG_ANGELBOOT
    mov     r0, #0x17              @ angel_SWIreason_EnterSVC
    swi     0x123456              @ angel_SWI_ARM
    mrs     r0, cpsr               @ turn off interrupts to
    orr     r0, r0, #0xc0           @ prevent angel from running
    msr     cpsr_c, r0
#endif

```

코드 915. head.S

section으로 “.start”를 주고 있으며, 할당되어야 하며, 실행 명령을 가진다는 것을 표시하고 있다. 시작점은 start가 될 것이며, 이것은 4bytes 단위로 정렬된다. 별다르게 하는 일은 없으며, mov r0, r0를 8번 반복적으로 실행해서 Instruction Cache를 비운다. b(Branch)명령을 사용해서 1f(Forward)로 제어를 옮긴다. 1f와 b사이에 있는 값은 단순히 loader가 이것이 올바른 값을 가지는 것인지를 나타내주는 Magic Number값과 zImage의 로드 및 실행 절대 주소 값과 마지막 주소 값을 가지는 start, _edata를 정의하고 있다. 제어는 이제 “1:”로 옮겨오고, r7에 r1을 넣어서, architecture ID를 저장하고, r8에는 0을 넣는다. 하지만, 현재로서는 r1에 아무 값도 없으므로 r7도 적절한 값을 가지지 못한다.

CONFIG_ANGELBOOT가 선택된 경우에는 #ifdef 이하의 부분을 실행한다. Angel boot을 실행하기 위해서는 SVC(Supervisor Mode)에서만 가능하며, FIQ(Fast Interrupt)와 IRQ(Interrupt)가 disable된 상태로만 가능하다. 사용하지 않는다면, 이 부분을 필요로 하지 않는다. 먼저 r0에 0x17을 넣고, swi(software interrupt) 0x123456을 준다. mrs 명령은 cpsr이나 spsr_<mode>의 내용이 general register로 불러드리는데 사용한 것으로 mrs r0, cpsr은 cpsr를 읽어서 r0에 둔다는 의미이다. 여기서 다시 0xC0(=11000000b)를 OR시켜서 cpsr_c에 다시 저장하면, FIQ와 IRQ가 disable된다.

이하의 부분은 .text의 section에 저장되는 부분이다. 따라서, 만약 .start가 아직 남은 것이 있다면, 그곳을 먼저 수행하고, .text가 수행될 것이다. 이에 해당하는 부분이 바로 ~/arch/arm/boot/compressed/head-sa1100.S이다. 아래와 같다.

```

#include <linux/config.h>
#include <linux/linkage.h>
#include <asm/mach-types.h>

.section      ".start", #alloc, #execinstr
_SA1100_start:
    @ Preserve r8/r7 i.e. kernel entry values
#endif(CONFIG_SA1100_GRAPHICSCLIENT) && !defined(CONFIG_ANGELBOOT)
    mov     r7, #MACH_TYPE_GRAPHICSCLIENT
    mov     r8, #0
#endif
#endif CONFIG_SA1100_VICTOR
    teq     r7, #MACH_TYPE_VICTOR
    bne     10f

    @ Copy cmdline to 0xc0000000
    mov     r1, #0xc0000000

```

```

        cmp    r0, #0
        moveq r2, #0
1:      ldrneb r2, [r0], #1
        cmpne r2, #0
        strb   r2, [r1], #1
        bne    1b

10:
#endif

@ Data cache might be active.
@ Be sure to flush kernel binary out of the cache,
@ whatever state it is, before it is turned off.
@ This is done by fetching through currently executed
@ memory to be sure we hit the same cache.
bic    r2, pc, #0x1f
add   r3, r2, #0x4000          @ 16 kb is quite enough...
1:      ldr    r0, [r2], #32
        teq   r2, r3
        bne    1b
        mcr   p15, 0, r0, c7, c10, 4      @ drain WB
        mcr   p15, 0, r0, c7, c7, 0 @ flush I & D caches

@ disabling MMU and caches
mrc   p15, 0, r0, c1, c0, 0 @ read control reg
bic    r0, r0, #0x0d          @ clear WB, DC, MMU
bic    r0, r0, #0x1000         @ clear Icache
mcr   p15, 0, r0, c1, c0, 0

#endif CONFIG_ANGELBOOT
/*
 * Pause for a short time so that we give enough time
 * for the host to start a terminal up.
 */
1:      mov    r0, #0x00200000
        subs   r0, r0, #1
        bne    1b
#endif

```

코드 916. head-sa1100.S

우리가 관심이 있는 것은 SA1100자체로서 CONFIG_SA1100이외에는 다 무시한다. 즉, Data cache를 비우는 과정부터 수행될 것이다. bic r2, pc, #0x1f는 pc(=r15)에서 0x1f(00011111b)값을 지워서 r2에 넣는 일을 한다. 즉, 32bytes단위로 정렬하는 것이 된다. r2에 0x4000을 더해서 r3로 둔다(add r3, r2, #0x4000). 이것은 16kbytes정도를 사용하겠다는 말이 되며, 이하에서는 r2와 r3를 비교해서 같지 않다면, 계속 r0값에 [r2]가 가르키는 값으로 넣어주고, r2는 32만큼씩 이동한다. 이것은 현재 실행되고 있는 명령이 확실히 Instruction cache에서 지워지도록 만들어주는 것으로 그냥 읽기만 해도 data cache가 비워지는 것을 이용하는 것이다. 한번에 32byte가 지워지기 때문에, 32씩 r2를 옮기고 있다.

이하의 mcr명령은 Write Back과 Instruction cache, Data cache를 비우는 명령이다. 아직 MMU를 사용하지 않으므로 MMU및 cache는 disable시키도록 하자. 이것은 순서적으로 먼저 control register의 내용을 읽어오고(mcr p15, 0, r0, c1, c0, 0), 읽어온 값에 0x0D(=1101)의 값에 대응하는 bit을 지운 후(Write Back, Data cache, MMU disable), 다시 0x1000(=000100000000)에 해당하는 bit을 지운다(Instruction Cache). 그런 후에 마지막으로 이 값을 다시 control register에 넣는다(mcr p15, 0, r0, c1, c0, 0). 이것이 끝나면, 다시 원래의 head.S로 돌아가서 .text에서 수행을 계속해 나갈 것이다.

```

.text
/* 이곳에 Samsung WVP를 위한 코드를 추가 한다.- 추가되는 코드는 다음에서
설명한다.*/
1:      adr    r2, LC0

```

```

ldmia r2, {r2, r3, r4, r5, sp}
/* BSS를 0으로 채운다. BSS의 시작주소는 r2가 가지며, 마지막 주소는 r3가 가진다.*/
1:    mov    r0, #0
      str    r0, [r2], #4          @ clear bss    /* 0으로 r2가 가리키는 곳을 채운다.
 */
      str    r0, [r2], #4
      str    r0, [r2], #4
      str    r0, [r2], #4
      cmp    r2, r3
      blt    1b                  /* r2와 r3의 값을 비교해서 r2가 작다면, 다시 1b로 */

      mrc    p15, 0, r6, c0, c0  @ get processor ID
      bl     cache_on

      mov    r1, sp          @ malloc space above stack
      add    r2, sp, #0x10000  @ 64k max

      teq    r4, r5          @ will we overwrite ourselves?
      moveq   r5, r2          @ decompress after image
      movne   r5, r4          @ decompress to final location

      mov    r0, r5
      mov    r3, r7

/* decompress_kernel의 parameter로
   output_start : r0,
   free_mem_ptr_p : r1,
   free_mem_ptr_end_p : r2,
   arch_id : r3
으로 설정된다.*/
      bl     SYMBOL_NAME(decompress_kernel)

/* r0에는 압축이 해제된 커널의 크기가 넘어온다.*/
      teq    r4, r5          @ do we need to relocate
      beq    call_kernel      @ the kernel?

      add    r0, r0, #127
      bic    r0, r0, #127      @ align the kernel length

```

코드 917. head.S의 계속

.text0이라는 text(code)영역임을 표시한다. 이곳에서 앞으로 더 이상 진행하기 전에 Samsung WVP를 위한 초기화 해주어야 할 일들을 수행한다. 첨가된 부분이므로 SA1100에 기초한 다른 것을 구현하고자 한다면, 이곳에서 해당하는 일을 해주는 것이 좋을 것이다. 설명은 옆에 달아놓은 주석을 참조하도록 하자.

```

#endif CONFIG_SA1100_WVP
/* LCD를 OFF하고 PPL을 조정한다.*/
ldr    r1, =WVP_PORT2        /* WVP의 port2 주소를 r1으로 가져온다.*/
ldr    r0, =0x0               /* WVP의 port2 주소는 0x22000000이다.*/
str    r0, [r1]              /* LDC를 OFF한다.*/

ldr    r0, =0x90020014       /* PLL(Phase Locked Loop)의 control register주소 */
ldr    r1, =0x08              /* PLL을 176 MHz로 맞춘다.*/
str    r1, [r0]

```

```

/* 이곳에 Supervisor mode로 전환하고, Interrupt를 불가능하게 만드는 것이
   있을 수 있다. */

/* Instruction Cache를 enable시키고, 나머지는 disable 시킨다.*/
ldr    r0, =0x1000          /* I cache만 enable, 나머지는 전부 OFF */
mcr    p15, 0, r0, c7, c7, 0  /* I cache와 D cache를 flush */
mcr    p15, 0, r0, c8, c7, 0  /* I 와 D TLB(Translation Lookaside Buffer)를 flush */

mcr    p15, 0, r0, c1, c0, 0  /* I cache는 Enable, MMU는 disable */
mcr    p15, 0, r0, c15, c1, 2  /* Clock switching을 enable */

/* GPIO에 대한 초기화를 한다.
   이곳에서 사용하는 Symbol들을 위해서 ~/asm/hardware.h가 필요하다. */
/* IRQ related init – GRER과 GFER를 0으로 초기화, GEDR을 전부 1로 설정 */
ldr    r1, =_GRER           /* GPIO Rising Edge Register */
ldr    r0, =0x0
str   r0, [r1]
ldr    r1, =_GFER           /* GPIO Falling Edge Register */
str   r0, [r1]
ldr    r1, =_GEDR           /* GPIO Edge Detect Register */
ldr    r0, =0xFFFFFFFF
str   r0, [r1]

/* GPIO Alternate function과 direction register 설정 */
ldr    r1, =_GAFR           /* GPIO Alternate Function Register */
ldr    r0, =0x0800003fc      /* 0000 1000 0000 0000 0000 0011 1111 1100b */
ldr    r0, [r1]               /* 2에서 9번 bit까지와 27번 bit을 ON 시킨다.
                                27 번 bit는 나중에 watchdog을 위한 output이다.
                                2에서 9번 bit는 LCD에서 output으로 사용한다.
*/
ldr    r1, =_GPDR           /* GPIO Pin Direction Register */
ldr    r0, =0x095993fc       /* 0000 1001 0101 1001 1001 0011 1111 1100b
                                27번 bit은 watchdog을 위한 output이다. */
str   r0, [r1]               /* Input : 0, Output : 1으로 설정한다. */

ldr    r1, =_GPSR           /* GPIO Pin output Set Register */
ldr    r0, =0x00100000       /* 0000 0000 0001 0000 0000 0000 0000 0000b */
str   r0, [r1]

ldr    r1, =_GPCR           /* GPIO Pin output Clear Register */
ldr    r0, =0x01400000       /* 0000 0001 0100 0000 0000 0000 0000 0000b */
str   r0, [r1]

/* 메모리 관련 설정 */
bl     setupdram            /* r14에 DramConfigData의 주소를 저장 */

DramConfigData:
.word 0x02f1a1b3
.word 0x9999998F
.word 0x0000F999
.word 0xFFFFFFF0
.word 0x47fc99f4
.word 0x92a12464
.word 0x0c650c62

setupdram:
ldr    r0, =0xA0000000       /* DRAM 설정 Register의 위치를 가져온다. */

```

```

ldmia r14, {r1-r7}          /* 앞에서 저장한 r14를 r1~r7으로 읽어 들인다.*/
stmia r0, {r1-r7}           /* r0이 가르키고 있는 DRAM 설정 레지스터에
                           r1~r7을 쓴다.*/
loop1:
    mov r0, #0x00000200      /* r0에 0x00000200을 넣는다.*/
loop*/:
    subs r0, r0, #0x1        /* 일정 시간동안 delay를 준다. 예로서 200번의
                           */
    bne loop1
    mov r7, #63               /* Machine 의 Architecture ID번호를 인위적으로 준다.
*/
/* zImage를 예약된(reserved) RAM 공간으로 복사한다.*/
    ldr r0, =0x00000000      /* 메모리의 최하 번지 */
    ldr r1, =0xC0F00000      /* zImage가 옮겨갈 위치 */
    ldr r2, =0x00100000      /* 1MByte의 영역에 zImage가 올라가 있다고 가정한다.
*/
/* MoveMe:
    ldr r3, [r0], #4          /* Block 메모리 복사 : 4bytes단위 */
    str r3, [r1], #4
    subs r2, r2, #4
    bne MoveMe
    ldr pc, =JumpToRam       /* Block copy후 옮겨진 메모리로 jump한다.
*/
JumpToRam:
#endif

```

코드 918. head.S에 대한 Samsung WVP용의 코드 추가

이곳에서 한가지 짊고 넘어가야 하는 것은 booting의 초기시에는 ROM 영역에서 시작하지만, 상대적인 번지를 접근하고자 한다면, 적절한 위치로 커널이 옮겨져야 한다는 것이다. 즉, 커널의 compile시에 적재될 위치로 옮기는 것이 필요한 것이다. 이것을 마치고 나면, 이제부터는 Label에 대한 접근이 가능하다. 이 앞에서는 상대적인 주소 지정에 의한 Label의 참조만이 가능하다.

head.S의 설명으로 돌아가서, adr r2, LC0(LC0는 아래부분에 정의되어 있다.) r2에 LC0의 주소를 가져와서 사용하겠다는 것이다. 이하의 ldmia r2, {r2, r3, r4, r5, sp}는 r2가 가지는 주소에서 r2, r3, r4, r5, sp(stack pointer)를 가져오는 것으로 아래의 내용들이 각각의 레지스터에 들어갈 것이다. 즉, r2에는 __bss_start가, r3에는 __end, r4에는 __load_addr, r5에는 __start, sp에는 user_stack+4096이 각각 들어갈 것이다.

```

LC0:
.type LC0, #object
.word __bss_start             /* r2 */
.word __end                   /* r3 */
.word __load_addr              /* r4 */
.word __start                  /* r5 */
.word user_stack+4096          /* sp */
.size LC0, . - LC0
...
.align
.section ".stack"
user_stack:
.space 4096

```

이하의 부분은 r2를 BSS의 영역으로 설정하고, r3를 마지막 BSS의 주소로 설정한 다음 전체 BSS를 0으로 만드는 부분이다. BSS를 전부 0으로 채웠다면, 아래의 mrc p15, 0, r6, c0, c0를 수행해서 processor ID를 받아온다. mrc는 coprocessor를 사용하는데 쓰이는 명령이다.

이전 sp를 r1으로 읽고, sp에 0x10000을 더해서 r2에 넣는다. 만약 r4와 r5가 같다면, __load_addr과 __start가 같은 값을 가지므로 r2를 r5에 넣어서, __bss_start에서 __start가 있도록 만들고, 그렇지 않다면, __load_addr과

`_start`이 다른 값을 가지므로 `r4`를 `r5`에 넣도록 한다. 즉, `_load_addr`를 `_start`가 가지도록 만든다. 이렇게 `r5`가 결정되면, `r5`를 다시 `r0`에 넣고, `r7`에는 이전에 구한 architecture ID값을 가지고 있으므로 이를 `r3`에 넣은 후, `bl SYMBOL_NAME(decompress_kernel)`을 사용해서 제어를 옮긴다.

여기서 주의할 점은 우리는 `decompress_kernel`로 제어를 옮기기 전에 C 함수를 수행하기 위한 BSS영역을 만들어주었다는 점이다. 즉, `decompress_kernel`은 C로 되어있는 함수이므로 이것을 수행하기 위해서는 반드시 BSS 및 stack을 만들어주어야 한다. 넘겨주는 parameter값은 각각 `r0, r1, r2, r3`가 된다. `decompress_kernel`을 보도록 하자.

```
ulg decompress_kernel(ulg output_start, ulg free_mem_ptr_p, ulg free_mem_ptr_end_p, int arch_id)
{
    /* output_start : r0, free_mem_ptr_p : r1, free_mem_ptr_end_p : r2, arch_id : r3 으로 설정된다.*/
    output_data          = (uch *)output_start;           /* Points to kernel start */
    free_mem_ptr         = free_mem_ptr_p;
    free_mem_ptr_end     = free_mem_ptr_end_p;
    __machine_arch_type = arch_id;

    proc_decomp_setup();
    arch_decomp_setup();

    makecrc();
    puts("Uncompressing Linux...");
    gunzip();
    puts(" done, booting the kernel.\n");
    return output_ptr;
}
```

코드 919. `decompress_kernel()`함수의 정의

`r0`가 `output_start`를 가지므로 이 값은 커널이 `decompress`가 된 후에 어디서 시작하는지를 나타내는 `output_data`가 될 것이다. `r1`은 stack 영역의 위에 있는 `malloc`을 위한 주소공간을 가지며, `free_mem_ptr`가 나타난다. `r2`는 `malloc`을 위한 주소공간의 끝을 가지며, `r2`에서 `r1`을 빼면 64kbytes의 공간이 되며, `free_mem_ptr_end`가 `r2`의 값을 가진다. `arch_id`는 `r3`값을 취하며, `__machine_arch_type`가 진다. 압축을 해제하기 전에 `proc_decomp_setup()`과 `arch_decomp_setup()`을 먼저 실행하고, `makecrc()`함수를 호출한다. 만약 serial cable을 살려두었다면, `puts()`는 “Uncompressing Linux...”을 serial로 보낼 것이며, `gunzip()`을 호출한 후, 다시 “done, booting the kernel”을 serial로 보낼 것이다. `return`되는 값은 `output_ptr`이다. 이 값은 `r0`에 넣어서 돌려질 것이다. 이 값은 압축이 해제된 kernel의 크기를 나타낸다.

더 booting에 대해서 진행하기 전에 Samsung의 WVP를 위해서 첨가되는 부분을 보도록 하자. 즉, `puts`와 같은 함수가 serial로 데이터를 내 보내기 위해서 필요한 일등이 될 것이다.

먼저 `proc_decomp_setup()`부터 보도록 하자. SA1100인 경우에는 `~/include/asm-arm/proc-armv/uncompress.h`에 아래와 같이 나와있다.

```
static inline void proc_decomp_setup (void)
{
    __asm__ __volatile__ ("
        mrc    p15, 0, r0, c0, c0          /* coprocessor 15에서 데이터를 읽어서 r0에 넣는다.*/
        eor    r0, r0, #0x44 << 24       /* r0에서 0x44(=01000100b)를 24bit 원쪽으로 shift시켜서
                                         exclusive OR시킨다.*/
        eor    r0, r0, #0x01 << 16       /* 다시 r0에 0x01(=00000001b)를 16bit 만큼 원쪽으로
                                         시켜서 exclusive OR시킨다.*/
        eor    r0, r0, #0xA1 << 8        /* 다시 r0에 0xA1(=10100001b)를 8bit 만큼 원쪽으로
                                         시켜서 exclusive OR시킨다.*/
        movs   r0, r0, lsr #5            /* 마지막으로 r0를 오른쪽으로 5만큼 shift시켜서 r0로 넣*/
    " :: "" : "" : "r0" );
}
```

```

    는다. 이때 CPSR의 S, N, Z, C bit이 영향을 받는다.*/
    mcreq p15, 0, r0, c7, c5, 0      @ flush I cache /* Instruction Cache를 비운다.*/
    mrceq p15, 0, r0, c1, c0          /* coprocessor 15에서 데이터를 읽어서 r0에 넣는다.*/
    orreq r0, r0, #1 << 12           /* r0을 0x1(=0001b)를 왼쪽으로 12bit shift한다.*/
    mcreq p15, 0, r0, c1, c0          @ enable I cache /* Instruction Cache를 enable시킨다.*/
    mov    r0, #0                      /* r0에 0을 넣는다.*/
    mcreq p15, 0, r0, c15, c1, 2     @ enable clock switching /* Clock switching을
enable시킨다.*/
    " :: : "r0", "cc", "memory");
}

```

코드 920. proc_decomp_setup()

이 코드는 inline C함수로서 inline assembly명령을 포함한다. 이 함수가 하는 일은 coprocessor 15를 통해서 시스템의 설정을 변경하는 것으로 Cache를 비우거나 enable/disable하는 역할을 한다. 이곳에서는 I cache를 비우고 나서, 다시 I cache를 enable하고, clock switching을 enable하는 일을 한다. XXXeq라고 된 것은 마지막 Instruction의 수행결과 CPSR의 Z bit이 설정되었을 경우에 실행하라는 말이다. 즉, 위에서 0x4401A1을 p15의 c0에서 읽어온 r0의 내용과 exclusive OR시켜서, 이 값을 right로 5bit shift한 것이 0이라면 실행될 것이다.

arch_decomp_setup()함수는 architecture 의존적인 설정을 하는 곳이다. ~/include/asm/arch/uncompress.h에 아래와 같이 정의되어 있다. 이 헤더 파일을 보면, puts()함수에 대한 것도 찾을 수 있을 것이다.

```

extern void sa1100_setup( int arch_id );
#define arch_decomp_setup() sa1100_setup(arch_id)

static void puts( const char *s )
{
    volatile unsigned long *serial_port;

    if (machine_is_assabet()) {
        if( machine_has_neponset() )
            serial_port = (unsigned long *)_Ser3UTCR0;
        else
            serial_port = (unsigned long *)_Ser1UTCR0;
    } else if (machine_is_brutus()||machine_is_nanoengine() ||
               machine_is_pangolin() || machine_is_freebird())
        serial_port = (unsigned long *)_Ser1UTCR0;
    else if (machine_is_empeg() || machine_is_bitsy() ||
              machine_is_victor() || machine_is_lart() ||
              machine_is_sherman() || machine_is_yopy() ||
              machine_is_huw_webpanel())
        serial_port = (unsigned long *)_Ser3UTCR0;
    /* 자신의 machine type에 맞는 UART의 port로 설정한다.*/
    else if(machine_is_wvp())           /* 아래에서 이 값을 정하는 부분을 보게 될 것이다.*/
        serial_port = (unsigned long *)_Ser1UTCR0;           /* Serial Port 1을 사용한다.*/
    else
        return;

    for ( ; *s; s++) {
        /* wait for space in the UART's transmitter */
        while (!(serial_port[UTSR1] & UTSR1_TNF));
        /* send the character out.*/
        serial_port[UART_TX] = *s;

        /* if a LF, also do CR... */
    }
}

```

```

    if (*s == 10) {
        while (!(serial_port[UTSR1] & UTSR1_TNF));
        serial_port[UART_TX] = 13;
    }
}

```

코드 921. puts()함수 및 arch_decomp_setup()함수의 정의

puts()함수는 boot loader에서 이미 설정된 serial port를 이용해서 메시지를 보내기 위해서 사용하는 함수이다. 각각의 board에 따라서 다른 serial port를 사용하기에 machine_is_xxx()라는 함수를 사용하고 있다. 이 함수에 대해서는 조금 있다가 확인해보도록 하고, 이렇게 machine type에 따른 serial port에 따라서, 우리가 사용하고자 하는 보드의 type을 정했다면, machine_is_wvp()라는 형식을 사용하면 될 것이다. _SerXUTCRX의 값은 ~/include/asm-arm/arch-sa1100/SA-1100.h에 정해져 있으므로 참조하기 바란다. 기본적으로 UART의 control register의 주소를 가지고 있다. 나머지는 이렇게 설정된 serial port에 그대로 넘겨받은 문자를 적어넣는 일이다.

먼저 serial_port[UTSR1]을 살펴서 UART의 transmitter쪽에 보낼 수 있는 여분의 공간이 있는지 확인한다. 이것은 while() loop를 돌면서 UTSR1_TNF와 UTSR1가르키는 serial_port[]부분을 AND시켜서 0일 때까지 수행하는 것으로 확인이 가능하다. 그리고, 나서 serial_port의 UART_TX에 글자를 옮겨서 집어 넣는 것이다. 한번에 한자씩 적어넣는다. 만약 LF(Line Feed)나 CR(Carrige Return)을 만나면 serial port에 공간이 이 있는지 확인하고 New Line 문자인 13을 적어넣는다.

machine_is_xxx() 매크로는 ~/arch/arm/tools/mach_types화일을 참조해서 자동적으로 생성된다. 이 파일에서 아래와 같은 것을 찾을 수 있을 것이다.

# machine_is_xxx	CONFIG_xxxx	MACH_TYPE_xxx	number
psionw	ARCH_PSIONW	PSIONW	60
aln	ARCH_ALN	ALN	61
camelot	ARCH_CAMELOT	CAMELOT	62
# 아래의 라인을 추가하자.			
wvp	ARCH_WVP	WVP	63

이렇게 해주면, 나중에 machine_is_xxx()와 같은 매크로나, CONFIG_xxxx, MACH_TYPE_xxx와 같은 상수값을 사용하게 될 때, 여기서 정의된 것이 xxx에 대치되어서 사용될 것이다. ~/arch/arm/tools/Makefile을 보면 여기서 정의해준 내용이 어디에 들어가는지 볼 수 있다. 아래를 참고하자.

```

$(TOPDIR)/include/asm-arm/mach-types.h: mach-types gen-mach-types
awk -f gen-mach-types mach-types > $@

```

즉, gen-mach-types를 이용해서 mach-types를 awk가 \$(TOPDIR)/include/asm-arm/mach-types.h로 생성해 냄을 볼 수 있다. 우리가 추가한 내용이 잘 동작하고 있다면, ~/include/asm-arm/mach-types.h에는 아래와 같은 내용이 추가될 것이다.

```

...
#define MACH_TYPE_WVP 63
...
#endif CONFIG_SA1100_WVP
#ifndef machine_arch_type
# undef machine_arch_type
# define machine_arch_type __machine_arch_type
#else
# define machine_arch_type MACH_TYPE_WVP
#endif
#define machine_is_wvp() (machine_arch_type == MACH_TYPE_WVP)

```

```
#else
# define machine_is_wvp() (0)
#endif
...
```

따라서, 코드에서 나오는 `machine_is_wvp()`은 (`machine_arch_type == MACH_TYPE_WVP`)로 대치되거나, 아니면 항상 0인 값을 가진다.

`arch_decomp_setup()`은 `sa1100_setup()`으로 정해져 있다. `sa1100_setup()`이 넘겨받는 값은 `arch_id`로 앞에서 `r3` 레지스터의 값이다. `sa1100_setup()`함수는 `~/arch/arm/boot/compressed/setup_sa1100.S`에 정의되어 있다. 넘겨받는 값은 아무 것도 없다. C에서 함수의 호출에 넘겨주는 parameter값은 순서대로 `r0`부터 정해지므로 `r0`에 들어가게 된다. 이 함수가 하는 역할은 UART(Universal Asynchronous Receiver & Transmitter)에 대한 board에 특정한 설정을 해주는 것이다. 아래와 같이 정해진다.

```
.text
GPIO_BASE: .long 0x90040000
#define GPLR 0x00
#define GPDR 0x04
...
#define GPSR 0x08
#define GAFR 0x1c
PPC_BASE: .long 0x90060000
#define PPAR 0x08
IC_BASE: .long 0x90050000
#define ICMR 0x04
UART1_BASE: .long 0x80010000
UART3_BASE: .long 0x80050000
#define UTCR0 0x00
#define UTCR1 0x04
#define UTCR2 0x08
#define UTCR3 0x0c
#define UTSR0 0x1c
#define UTSR1 0x20
#ifndef CONFIG_SA1100_DEFAULT_BAUDRATE
#define CONFIG_SA1100_DEFAULT_BAUDRATE 9600
#endif
#define BAUD_DIV ((230400/CONFIG_SA1100_DEFAULT_BAUDRATE)-1)
SCR_loc:.long SYMBOL_NAME(SCR_value)
#define GPIO_2_9 0x3fc
...
ENTRY(sa1100_setup)
    mov r3, r0          @ keep machine type in r3

    @ Clear all interrupt sources
    ldr r0, IC_BASE     /* Interrupt Controller Register의 주소를 구한다.*/
    mov r1, #0
    str r1, [r0, #ICMR] /* ICMR(Interrupt Clear Mask Register)에 0을 넣어서 interrupt가
                           생기지 않도록 한다. */

    teq r3, #MACH_TYPE_ASSABET /* MACH_TYPE_ASSABET과 r3를 비교한다.*/
    bne skip_SCR         /* 같지 않다면 skip_SCR로 제어를 옮긴다. */

    /* MACH_TYPE_ASSABET 인 경우에 실행한다.*/
    ldr r0, GPIO_BASE    /* GPIO Control Register의 Base Address를 구한다
*/
    ldr r1, [r0, #GPDR]  /* GPDR를 읽어서 r1에 넣는다. */
```

	and	r1, r1, #GPIO_2_9	/* GPIO_2_9 (= 0x3FC = 00111111100b)를 r1과 AND
			*/
	str	r1, [r0, #GPDR]	/* r1을 다시 원래의 GPDR에 넣는다. */
	mov	r1, #GPIO_2_9	/* 0x3F를 r1에 넣는다. */
	str	r1, [r0, #GPSR]	/* GPSR에 r1을 넣는다. */
	ldr	r1, [r0, #GPDR]	/* GPDR을 다시 읽어서 r1에 넣는다. */
	bic	r1, r1, #GPIO_2_9	/* r1에서 0x3FC를 bit clear(bic)한다. */
	str	r1, [r0, #GPDR]	/* r1값을 GPDR에 넣는다. */
	/* 100번 순환하면서 r1으로 GPLR값을 넣는 것을 반복한다. */		
1:	mov	r2, #100	/* r2에 100을 넣는다. */
	ldr	r1, [r0, #GPLR]	/* GPLR을 읽어서 r1에 넣는다. */
	subs	r2, r2, #1	/* r2에서 1을 뺀 후, 다시 r2에 넣는다. */
	bne	1b	/* 같지 않다면, 1b로 제어를 옮긴다. */
	and	r2, r1, #GPIO_2_9	/* r1값과 GPIO_2_9(0x3FC)와 AND한다. */
	ldr	r1, SCR_loc	/* r1에 SRC_loc의 주소를 넣는다. */
	str	r2, [r1]	/* r2를 r1이 가르키고 있는 주소에 넣는다. */
	ldr	r1, [r0, #GPDR]	/* r1에 GPDR의 값을 읽어 온다. */
	and	r1, r1, #GPIO_2_9	/* r1과 0x3FC를 AND한다 */
	str	r1, [r0, #GPDR]	/* GPDR에 r1을 넣는다. */

코드 922. sa1100_setup()함수의 정의

r0로 넘겨받은 arch_id를 r3에 저장한다. 다시 r0에는 IC_BASE(Interrupt Controller Base)를 가지도록 만들고, r1에 0을 넣은후 IC_BASE를 가진 r0에 #ICMR(Interrupt Controller Mask Register) 읍셋을 더해서, r1을 넣는다. 즉, Interrupt Controller의 Mask값을 0으로 설정해서 Interrupt가 발생하지 못하도록 만든다. 만약 r3에 저장된 arch_id값이 MACH_TYPE_ASSABET과 같다면 이하를 진행하고, 그렇지 않다면, skip_SCR로 제어를 옮긴다.

여기서부터는 이전 GPIO의 설정에 관련된 것이다. 즉, Serial Port를 사용하기 위한 GPIO설정을 해준다. 설정해 주어야 하는 GPIO 레지스터 들에는 GPDR, GPSR, GPLR, GARF등이 있다. 각각의 GPIO 레지스터들에 대한 것은 앞에서 이미 논의 했으므로, 이곳에서는 어떤 설정만이 적용되는지 보도록 하겠다.

MACH_TYPE_ASSABET의 경우에는 먼저 r0에 GPIO_BASE의 주소를 가져온다. 이후의 연산은 이것을 이용해서 load와 store를 사용할 것이다. GPDR의 현재 설정을 읽어서, 먼저 GPDR에서 값을 읽어서 이를 r1에 저장하고, 여기서 r1을 다시 0x3F값으로 AND 시켜서 r1에 도로 집어 넣는다. 즉, 2 bit에서 9 bit까지의 내용에 대해서 기존의 설정을 유지하고, 나머지는 전부 0으로 clear시켰다. 1로 설정한다는 말은 output으로 사용하겠다는 말이며, 0은 input으로 사용하겠다는 말이다. 이것을 원래 GPDR이 가르키는 값으로 다시 넣는다.

이전 output으로 설정된 GPDR에 맞게 GPSR을 설정해 주어야 할 것이다. 먼저 앞에서 설정한 GPIO_2_9를 r1에 넣고, 이 값으로 GPSR을 설정한다. GPSR의 한 bit을 1로 설정한다는 말은 output으로 설정된 pin들을 enable한다는 것이다. 따라서, 2 bit에서 9 bit까지의 output pin을 enable시켰다. 다시 r1에 GPDR를 읽어와서, 앞에서 AND한 GPIO_2_9까지의 bit를 지운다(bic). 이렇게 지운 값을 다시 GPDR에 넣어 줌으로써, 2 bit에서 9 bit까지의 내용을 지우게 된다. 전체 GPDR은 이제 0으로 설정된다. 위와 같은 연산이 제대로 수행되기를 잠시 대기하는 것으로 GPLR을 100번 읽는 연산을 반복한다. GPLR은 각 GPIO pin의 상태를 알려주는 레지스터이다.

이전 GPLR에서 읽은 값(r1에 들어있다.)에 GPIO_2_9를 AND시켜서 r2에 넣는다. 이것은 GPLR의 값을 잠시 보관할 목적으로 사용된다. 그리고나서 SCR_loc를 r1에 넣는데, 이때 SCR_loc는 SCR_value의 주소를 가지고 있다. 따라서, SCR_value의 주소를 r1이 가르키도록 만들고, 이곳에 r2에 저장했던 GPIO_2_9값을 집어 넣는다. SCR_value는 System Control Register의 값을 가진다. 다시 r1에 GPDR을

읽어서 가져오고, 이 값에 2 bit에서 9 bit까지를 GPIO_2_9로 AND시켜서 GPDR에 넣는다. 이것으로 ASSABET board의 경우에 필요한 절차를 수행했다.

위에서 한 일과 똑 같은 과정을 ~/arch/arm/mach-sa1100/hw.c의 get_assabet_scr()함수에서 다시 수행하기 때문에 참고적으로 보도록 하자. 아래와 같다. 이것은 위에서 행한 것에 대해서 이해를 돋기 위함이다.

```
#ifdef CONFIG_SA1100_ASSABET
unsigned long BCR_value = BCR_DB1110;
unsigned long SCR_value = SCR_INIT;
EXPORT_SYMBOL(BCR_value);
EXPORT_SYMBOL(SCR_value);

void __init get_assabet_scr(void)
{
    unsigned long flags, scr, i;

    local_irq_save(flags);           /* Interrupt가 발생하지 않도록 한다.*/
    GPDR |= 0x3fc;                 /* GPIO 9:2를 output으로 설정한다.*/
    GPSR = 0x3fc;                  /* GPIO output으로 설정된 것을 enable 시킨다.*/
    GPDR &= ~0x3fc;                /* 다시 GPIO 9:2를 input으로 설정한다.*/
    for(i = 100; i--; scr = GPLR); /* GPIO 9:2 까지의 상태를 읽는다.*/
    GPDR |= 0x3fc;                 /* 원래의 상태로 GPIO pin의 direction을 설정한다.*/
    local_irq_restore(flags);        /* 원래의 Interrupt flag값으로 되돌린다.*/
    scr &= 0x3fc;                  /* 시스템의 configuration 값으로 저장한다.*/
    SCR_value = scr;
}
#endif /* CONFIG_SA1100_ASSABET */
```

코드 923. get_assabet_scr()함수의 정의(계속)

이와같이 두번에 걸쳐서 같은 일을 하는 것은 커널이 zImage형태로 올라오지 않을 수 있다는 사실을 대비하기 위한 것이다. 즉, setup-sa1100.S는 압축된 커널 이미지를 생성할 때 사용되기 때문이다. 또한, 압축을 해제하는 부분으로부터 SCR 값을 전달하는 것도 문제가 있기 때문이다.

skip_SCR:

```
@ Initialize UART (if bootloader has not done it yet)...
teq      r3, #MACH_TYPE_BRUTUS
teqne   r3, #MACH_TYPE_ASSABET
teqne   r3, #MACH_TYPE_GRAPHICSCLIENT
/* 이곳을 수정한다.*/
teqne   r3, #MACH_TYPE_WVP
/* 수정된 부분의 끝 */
bne     skip_uart

@ UART3 if Assabet is used with Neponset
teq      r3, #MACH_TYPE_ASSABET      @ if Assabet
tstreq  r2, #(1 << 9)             @ ... and Neponset present
ldreq   r0, UART3_BASE
beq     uart_init

@ UART3 on GraphicsClient
teq      r3, #MACH_TYPE_GRAPHICSCLIENT
ldreq   r0, UART3_BASE
beq     uart_init

/* 이곳을 추가해서 넣는다.*/
@ At least for SAMSUNG_WVP, the UART3 is used through
```

```

@ the alternate GPIO function...
teq      r3, #MACH_TYPE_WVP
ldreq    r0, UART3_BASE
beq     uart_init
/* 추가한 부분의 끝 */

@ At least for Brutus, the UART1 is used through
@ the alternate GPIO function...
teq      r3, #MACH_TYPE_BRUTUS
bne     uart1

```

코드 924. sa1100_setup()함수의 정의(계속)

skip_SCR로 제어를 옮겨왔다면, UART를 초기화하는 부분이다. 먼저 r3에 넣어둔 arch_id값과 같은 것은 것이 있는지 확인하고, 만약 같다면, xxxne부분은 수행되지 않으므로, 바로 다음 번의 비교(teq)로 넘어간다. 따라서, 해당하는 MACH_TYPE_XXX를 찾지 못하면, 곧장 skip_uart로 제어를 옮긴다. Brutus와 Assabet, Graphicsclient 중 하나라도 있다면, 아래로 UART3에 대한 setup을 실행한다. 가장 먼저 MACH_TYPE_ASSABET에 대해서 설정한다. 따라서, 우리가 필요한 설정으로 바꾸기 위해서는 위에서와 같은 수정이 필요하다.

수정을 가한 부분은 r3와 MACH_TYPE_WVP(=63)을 비교해서 만약 같다면, UART3_BASE를 r0에 가져오고 uart_init로 제어를 옮긴다.

```

alt_GPIO_uart: ldr    r0, GPIO_BASE
                ldr    r1, [r0, #GPDR]
                bic    r1, r1, #1<<15
                orr    r1, r1, #1<<14
                str    r1, [r0, #GPDR]
                ldr    r1, [r0, #GAFR]
                orr    r1, r1, #(1<<15)|(1<<14)
                str    r1, [r0, #GAFR]
                ldr    r0, PPC_BASE
                ldr    r1, [r0, #PPAR]
                orr    r1, r1, #1<<12
                str    r1, [r0, #PPAR]

uart1:         ldr    r0, UART1_BASE
uart_init:
1:             ldr    r1, [r0, #UTSR1]
                tst    r1, #1<<0          @ TBY
                bne    1b
                mov    r1, #0
                str    r1, [r0, #UTCR3]
                mov    r1, #0x08          @ 8N1
                str    r1, [r0, #UTCR0]
                mov    r1, #BAUD_DIV
                str    r1, [r0, #UTCR2]
                mov    r1, r1, lsr #8
                str    r1, [r0, #UTCR1]
                mov    r1, #0x03          @ RXE + TXE
                str    r1, [r0, #UTCR3]
                mov    r1, #0xff          @ flush status reg
                str    r1, [r0, #UTSR0]

skip_uart:
                @ Extra specific setup calls
                @ The machine type is passed in r0
                mov    r0, r3
#endif CONFIG_SA1100_NANOENGINE

```

```

teq      r0, #MACH_TYPE_NANOENGINE
beq      SYMBOL_NAME(bse_setup)
#endif
out:     mov      pc, lr

```

코드 925. sa1100_setup()함수의 정의(계속)

`alt_GPIO_uart`는 Brutus에 대해서 GPIO의 alternate function을 사용하기 위한 설정을 하는 곳이다. 해당 사항이 없다면, 기본적으로 사용하는 것은 `uart1`에서 보이는 `UART1_BASE`이다. 즉, `UART1`을 사용해서 `serial`을 설정한다는 말이다. `alt_GPIO_uart`를 보면, `r0`에 `GPIO_BASE`를 가져와서, `GPDR`를 읽어 `r1`에 둔다. 이 값에서 15bit에 있는 한 bit를 clear한 후 14 bit를 설정해서, 다시 `GPDR`에 설정한다. 그리고나서, `GPIO pin`을 alternate function에서 사용하기 위해서 `GAFR` 레지스터를 읽어, `r1`에 두고, 15 bit과 14bit를 설정한 후, 이 값으로 `GAFR` 레지스터에 새로 적어 넣는다. `PPC_BASE`는 Peripheral Pin Controller 레지스터의 기본 주소를 가진 값으로, 이를 이용해서 LCD나 serial port 1~4까지의 pin을 이용할 수 있도록 한다. 여기서는 `PPAR`(`PPC pin assignment register`)를 접근하기 위한 값이며, `PPAR`에서 값을 읽어서, `r1`에 저장한 후, 12 bit를 설정해서 다시 `PPAR`에 넣는다.

이전 `uart1`에 도달했다. 여기서는 `r0`에 단순히 `UART1_BASE`주소를 넣어주고, 이를 위한 설정을 아래에서 할 것이다. `uart_init`에서는 `UART`와 관련된 레지스터들에 대한 설정을 실재적으로 해주는 부분이다. `UTSR1`(`UART status register 1`)의 값을 읽어와서 1과 같은지 비교한다. 같지 않다면, 1b로 branch해서 다시 한번 살펴보도록 한다. 같다면, 아래로 계속 진행해서 `UTCR3`(`UART control register 3`)에 0을 넣는다. `r1`에 다시 8을 넣고, `UTRC0`에 이 값을 집어 넣는다. 이것으로 8 bit no parity로 설정한다. `BAUD_DIV`는 baud rate divisor를 값으로 baud rate를 결정하기 위한 값이다. 이 값을 `UTCR2`에 넣는다. 다시 `r0`를 8로 right shift하고, 이것을 `UTCR1`에 넣는다. 그리고나서 3을 `r1`에 넣고, 이 값을 `UTCR3`에 넣어서 Rx와 Tx를 enable시킨다. `Status` 레지스터를 비우기 위해서 `r1`에 0xFF를 넣은 다음 이 값을 `UTSR0`를 채운다. 이것으로 `UART`의 설정을 마치게 된다.

`skip_uart`는 `UART`의 사용을 하지 않는다는 말이다. `r3`에 있는 `MACH_TYPE_XXX`의 값을 다시 `r0`에 넣고, 이 값을 `MACH_TYPE_NANOENGINE`³⁸⁹ 과 비교해서 같다면, `bse_setup`으로 branch하고, 그렇지 않다면 마지막으로 함수를 복귀하기 위해서 `pc`(program counter)값에 `lr`(Link Register)를 넣어준다. 이 값은 앞에서 함수를 호출하는 과정에서 `lr`에 수행할 다음번 주소를 보관하기 때문에, 단순히 `pc`에 `lr`를 넣어주는 것으로 함수의 복귀가 가능하기 때문이다. `UART` 및 `PPC`에 대해서는 나중에 다시 살펴볼 것이다. 각각의 레지스터가 하는 역할도 그때 다시 설명하도록 하겠다.

이전 `sa1100_setup()`함수를 수행하였다고 보고, `serial port`를 이용해서 `output`을 보낼 수 있는 환경이 되었다. 앞에서 하던 이야기로 돌아가서 `decompress_kernel()` 함수를 계속 보도록 하자. 나머지의 코드는 아래와 같다.

```

ulg decompress_kernel(ulg output_start, ulg free_mem_ptr_p, ulg free_mem_ptr_end_p, int arch_id)
{
    ...
    /* 여기서부터 */
    makecrc();
    puts("Uncompressing Linux...");
    gunzip();
    puts(" done, booting the kernel.\n");
    return output_ptr;
}

```

코드 926. decompress_kernel()함수(계속)

`makecrc()`함수는 gzip-1.0.3의 `makecrc.c`에서 가져온 함수로 정의는 `~/lib/inflat.c`에 나와 있다. 이 함수가 하는 역할은 CRC-32 table을 만드는 일을 한다. `puts()`는 앞에서 보았듯이 `serial`로 `output`을 쓰는 역할을 하게 되며, 실제적인 커널의 압축을 해제하는 것은 `gunzip`이 한다. 끝나면, 다시 `puts()`함수로 압축

³⁸⁹ 물론 이것은 `CONFIG_SA1100_NANOENGINE`이 설정된 경우에 해당된다. 여기서는 필요하지 않기에 설명은 빼겠다.

해제를 마쳤다는 것을 보여주며, 압축이 해제된 커널의 크기를 나타내는 output_ptr을 돌려준다. 앞에서도 말했듯이 이 값은 r0에 넣어서 복귀코드로 전달된다. gunzip()함수 역시 makecrc()와 같은 ~/lib/inflate.c에 정의되어 있으니 참고하기 바란다. 이곳에서는 다루지 않는다.

	teq	r4, r5	@ do we need to relocate
	beq	call_kernel	@ the kernel?
	add	r0, r0, #127	
	bic	r0, r0, #127	@ align the kernel length
	add	r1, r5, r0	@ end of decompressed kernel
	adr	r2, reloc_start	
	adr	r3, reloc_end	
1:	ldmia	r2!, {r8 - r13}	@ copy relocation code
	stmia	r1!, {r8 - r13}	
	ldmia	r2!, {r8 - r13}	
	stmia	r1!, {r8 - r13}	
	cmp	r2, r3	
	blt	1b	
	bl	cache_clean_flush	
	add	pc, r5, r0	@ call relocation code
LC0:	.type	LC0, #object	
	.word	__bss_start	
	.word	_end	
	.word	_load_addr	
	.word	_start	
	.word	user_stack+4096	
	.size	LC0, . - LC0	

코드 927. head.S (계속)

이젠 커널의 압축이 해제가 되었다. 만약 r4와 r5가 같은 값을 가지는지 확인한다. 현재 r4는 커널의 실행 시작 주소를 가지고, r5는 압축이 해제된 커널의 시작 주소를 가지고 있다. r4의 경우 우리는 _load_addr를 0xC0008000을 가지고 있고, r5에는 _start 주소로 0xC0F00000을 가지고 있으므로 같지 않다. 따라서, call_kernel은 수행되지 않을 것이다. 이젠 r0에 127을 더하고, 다시 r0에서 127에 해당하는 bit을 지우는 것으로 압축이 해제된 커널의 크기를 정렬한다.

최종적으로 r0에는 압축이 해제된 커널의 크기를, r1에서 r3까지는 사용되지 않으며, r4에는 커널이 load된 주소를 r5에는 압축이 해제된 커널의 start위치, r6에는 processor ID를, r7에는 architecture ID를 가지는 레지스터 상태가 되었다. 나머지 r8에서 r14까지는 사용하지 않는다.

r5에 r0를 더해서 r1으로 둔다. r2에는 reloc_start를 r3에는 reloc_end의 주소를 넣는다. 즉, 커널을 relocation하기 위해서 준비하는 절차이다. reloc_start와 reloc_end는 나중에 나오는 내용이다. ldmia에서부터 blt 1b까지가 reloc_start에서 reloc_end까지의 내용을 압축이 해제된 커널의 뒤 쪽으로 block copy하는 부분이다. r8에서 r13까지의 6개의 레지스터에 reloc_start부터 시작하는 메모리 데이터를 넣고, r1이 가르키는 주소로 복사한다. 한번에 $4 \times 6 = 24$ (bytes)가 복사된다. 이 block copy 연산은 r2가 r3와 같거나 크게 되는 시점에서 끝난다.

bl은 return address를 r14에 기록하고 branch를 하는 instruction이다. 따라서, cache_clean_flush를 호출하고, 다시 이곳에서 수행이 계속될 것이다. Cache에 대한 함수로 주어진 것으로는 cache_on(), cache_off(), cache_clean_flush()이 있다. 먼저 cache_on() 함수는 cache를 enable하는 함수이며 enable하기 전에 일부 page table들의 내용을 정해줄 필요가 있다. 즉, page table이 어느정도 설정이 되면, instruction cache(I cache)와 data cache(D cache)를 사용할 수 있게 된다. cache_off()는 StrongARM의 MMU와 cache를 끄는 역할을 한다. 이 상태에서 Instruction cache를 on으로 두는 것은 안전하다. 마지막으로 cache_clean_flush()함수는 메모리와 cache의 일관성(일치성: consistency)을 유지하기 위해서 cache를 지우고, cache의 내용을 지우는 일을 한다. 따라서, cache_clean_flush로 제어를 옮겨서 캐쉬의 내용을 비우고 다음으로 진행해 나간다.

이전 pc에 r0와 r5의 값을 더해서 넣는다. 이것은 reloc_start와 reloc_end를 아까 위에서 압축이 해제된 위치에 풀었기에 그것으로 제어를 옮기기 위한 것이다. 이제 다음 번 연산은 pc에서 새로이 fetch될 것이다³⁹⁰

```
/* cache_on()함수의 정의가 있다.*/
...
/* 이 부분은 이야기의 중심이 되는 부분이기에 그대로 살려 두었다.
 * This code is relocatable. It is relocated by the above code to the end
 * of the kernel and executed there. During this time, we have no stacks.
 *
 * r0      = decompressed kernel length
 * r1-r3  = unused
 * r4      = kernel execution address
 * r5      = decompressed kernel start
 * r6      = processor ID
 * r7      = architecture ID
 * r8-r14 = unused
*/
        .align 5                  /* 왜 5 bytes 단위의 align을 하는가? */
reloc_start:    add    r8, r5, r0
                mov    r1, r4
1:           .rept 4
                ldmia r5!, {r0, r2, r3, r9 - r13}    @ relocate kernel
                stmia r1!, {r0, r2, r3, r9 - r13}
                .endr

                cmp    r5, r8
                blt   1b
call_kernel:   bl     cache_clean_flush
                bl     cache_off
                mov    r0, #0
                mov    r1, r7                  @ restore architecture number
                mov    pc, r4                  @ call kernel

/* 여러 SA11XX 프로세서를 위한 재배치 가능한 cache 지원 함수들의 있다.*/
...
/* cache_off()함수와 cache_clean_flush()함수의 정의가 있다.*/
...
/* 여러 debugging과 관련된 routine들이 정의 되어 있다. */
...
reloc_end:
        .align
        .section ".stack"
user_stack:    .space 4096          /* 커널이 임시적으로 사용하게 되는 stack */
```

코드 928. head.S

이전 relocation이 수행하고, 커널로 제어를 옮기는 일만 남았다. 재배치를 하는 것은 역시 block 단위로 메모리를 원하는 위치로 복사하는 것이다. 이 코드를 수행하기 전의 레지스터들의 값을 항상 잘 기억하도록 하자.

r8에 r0와 r5를 더해서 넣는다. 즉, r8는 현재 압축이 수행된 커널의 이미지의 끝 주소를 가지고 있다. r1에는 r4를 넣어서, 커널의 수행이 있어야 하는 번지를 가르킨다. 아까 우리는 r4의 경우 _load_addr을

³⁹⁰ reloc_start로 제어가 옮겨간다.

가지며, 값은 0xC0008000 임을 보았다. r5에는 _start 주소로 0xC0F00000을 가지고 있었다. 따라서, r4의 0xC008000으로 커널을 재배치하게 될 것이다.

먼저 r5가 가르키는 메모리 주소에서 r0, r2, r3, r9 ~ r13까지의 레지스터로 데이터를 가져오고, 이를 다시 r1이 가르키는 주소로 적어 넣는다. 따라서, 한번에 8개의 4 bytes 단위로 데이터를 이동시켜준다. 즉, 32 bytes의 이동이 일어난다. 이것을 다시 4번 반복(rept)하게 되므로, $32 \times 4 = 128$ bytes의 이동이 loop를 한번 수행하게 될 때마다 일어나게 된다. 128이란 수는 2의 7승에 해당한다. 전체 압축이 해제된 커널의 실제 이미지가 128 bytes씩 r4가 가리키는 주소로 복사된다. 즉, 커널의 재배치가 수행되는 것이다.

재배치가 다 되었다면, 이전 cache를 다시 한번 더 비우고(cache_clean_flush()), cache를 끄고나서(cache_off()), r0에는 0을, r1에는 architecture ID를 넣은 후, r4로 제어를 옮긴다(mov pc, r4).

여기서 중요한 점은 cache_on() 함수는 reloc_start와 reloc_end 사이에 들어가지 않는다는 점이다. 즉, 이 함수는 나중에 캐시를 enable할 때 다시 사용할 수 있음을 말한다. 이 과정을 그림으로 정리하면 [그림]와 같다.

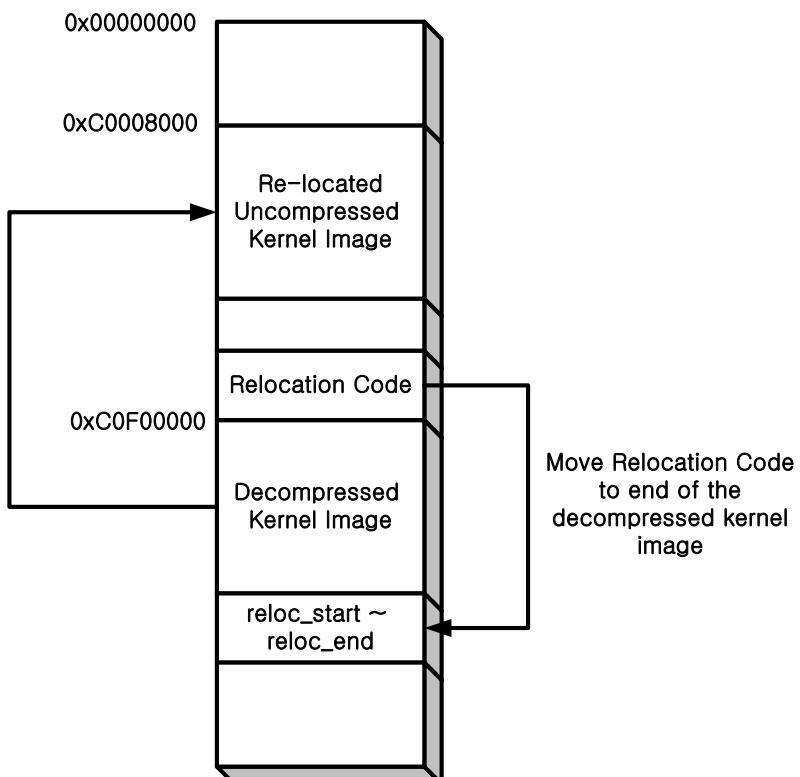


그림 128. 커널의 재배치

물론 이것은 전형적인 SA1100의 커널 재배치와는 다른 Samsung WVP용의 메모리에 기초한 커널의 재배치이다. 달라지는 것은 초기의 커널의 적재 주소와 재적재 주소, 그리고, 커널의 압축이 풀리는 위치이다. 이것은 시스템의 메모리 map이 달라질 수 있기 때문에, porting하는 각자가 알아서 주어야 할 것이다.

이것으로 ~/arch/arm/boot에 대한 이야기는 마쳤다. 물론 이곳에서 보여주지 않은 많은 코드들이 있으므로 그것은 따로이 공부하는 것이 좋을 것이다. 이전 ~/arch/arm/kernel에서 이야기를 진행한다. 가장 먼저 보게될 코드는 head-armv.S이다. 즉, 실제 커널의 제일 앞부분에 위치하는 코드 부분이다. 앞에서 call_kernel에서 커널로 제어를 옮기면 가장 먼저 수행된다.

17.7.2. head-armv.S의 분석

head-armv.S는 ARM Architecture 4에 해당하는 32 bit 커널 startup 코드를 가지고 있다. 여기서 부터는 이제 완전한 32 bit 커널이 구동되기 때문에 이를 위한 메모리 관리 부분의 설정이 필요할 것이다.

```

...
#define K(a,b,c) ((a) << 24 | (b) << 12 | (c))

/*
 * 이 파일이 커널의 컴파일에 설정되었다면, TEXTADDR은 0xXXXX8000의 형태를 가지고 있어야 할 것이다. 우리의 경우에는 0xC0008000이 TEXTADDR의 주소이다.*/
#ifndef TEXTADDR & 0xffff != 0x8000
#error TEXTADDR must start at 0xXXXX8000
#endif

.globl SYMBOL_NAME(swapper_pg_dir)
.equ SYMBOL_NAME(swapper_pg_dir), TEXTADDR - 0x4000

.macro pgtbl, reg, rambase
    adr \reg, stext
    sub \reg, \reg, #0x4000
.endm

/*
 * Since the page table is closely related to the kernel start address, we
 * can convert the page table base address to the base address of the section
 * containing both.
*/
.macro krnladr, rd, pgtable, rambase
    bic \rd, \pgtable, #0x000ff000
.endm

```

코드 929. head-armv.S

이곳에서는 Page Table을 위한 16K정보를 TEXTADDR(=0xC0008000)이하의 주소에 배치한다. 이를 위해서 0x4000만큼의 공간이 필요하며, 여기에 PAGE_OFFSET을 더한 값보다 TEXTADDR을 설정하면 되기에, TEXTADDR을 위해서 0xXXXX8000이란 값은 충분할 것이다.

이 영역에 들어가는 것이 바로 swapper_pg_dir이다. 이 변수는 페이지 테이블의 가상 주소(Virtual Address)를 알려주며, 아래에 정의된 매크로 pgtbl은 이러한 테이블의 위치에 의존적이 아닌 주소를 알려주기 위해서 사용하는 매크로이다. 즉, stext를 TEXTADDR로 설정하면, 이곳에서 0x4000을 뺀 번지(예를 들어서, 0x4000)를 돌려줄 것이다. 또한 krnladr 역시 매크로이며, rd에 pgtable에서 0x000FF000을 bit clear시킨 내용을 넣어서 전달한다. 이것은 나중에 페이지 테이블을 생성하는데 사용하는 매크로들이다.

```

.section ".text.init",#alloc,#execinstr
.type stext, #function
ENTRY(stext)
    mov r12, r0
#ifndef CONFIG_ARCH_NETWINDER || defined(CONFIG_ARCH_INTEGRATOR)
    .rept 8
    mov r0, r0
    .endr

    adr r2, 1f
    ldmdb r2, {r7, r8}
    and r3, r2, #0xc000
    teq r3, #0x8000
    beq __entry
    bic r3, r2, #0xc000
    orr r3, r3, #0x8000
    mov r0, r3
    mov r4, #64
    sub r5, r8, r7
    b 1f

```

```

.word    _stext
.word    __bss_start
1:
.rept    4
ldmia   r2!, {r6, r7, r8, r9}
stmia   r3!, {r6, r7, r8, r9}
.endr
subs    r4, r4, #64
bcs     1b
movs    r4, r5
mov     r5, #0
movne   pc, r0

mov     r1, #MACH_TYPE_NETWINDER      @ (will go in 2.5)
mov     r12, #2 << 24                 @ scheduled for removal in 2.5.xx
or     r12, r12, #5 << 12

__entry:
#endif
#ifndef CONFIG_ARCH_L7200
/*
 * FIXME - No bootloader, so manually set 'r1' with our architecture number.
 */
        mov     r1, #MACH_TYPE_L7200
#endif defined(CONFIG_ARCH_INTEGRATOR)
        mov     r1, #MACH_TYPE_INTEGRATOR
#endif

```

코드 930. head-armv.S (계속)

이전 커널의 실제적인 startup 시작점(entry point)이다. 앞에서 head.S의 마지막 부분을 보면, r0은 0으로 설정되어 있음을 알 수 있다. 또한 r1의 경우는 architecture ID를 가진다. 현재는 MMU를 앞에서 disable한 상태이며, I cache는 ON상태이거나 OFF상태, D cache는 OFF상태이다. 우리는 앞의 proc_decomp_setup() 함수에서 I cache를 enable 시켰기에 ON 상태로 보는 것이 좋을 것이다. 나머지는 Netwinder나 Integrator가 설정된 경우에 해당하므로 직접적인 상관은 없다. 이것은 예전에 있었던 boot firmware와의 호환성 문제를 해결할 목적으로 존재하는 코드이며, 커널 2.5에서는 없어질 계획이다. 또한 boot firmware가 없는 경우도 해결되는데, 이 경우에는 단지 r0를 0으로 r1을 적절한 architecture ID값으로 두면 된다. 참조할 것은 앞에서 본 ~/arch/arm/tools/mach_types이다.

mov	r0, #F_BIT I_BIT MODE_SVC	@ make sure svc mode
msr	cpsr_c, r0	@ and all irqs disabled
bl	__lookup_processor_type	

코드 931. head-armv.S(계속)

r0에 F_BIT과 I_BIT, MODE_SVC bit을 OR시켜서 넣는다. 이는 FIRQ와 IRQ, Supervisor Mode로 cpsr_c를 설정하기 위함이다. 즉, FIRQ와 IRQ가 발생하지 않으며, 현재의 모드를 Supervisor Mode(or Kernel Mode, System Mode)로 두기 위함이다.

이것을 마치면 가장 먼저, processor의 type을 찾는 것이다. __lookup_processor_type() 함수가 호출된다. 이 함수가 하는 일은 processor type을 찾아서 적절한 위치에 정보를 저장하는 것이다. 코드는 아래와 같다.

__lookup_processor_type:		
adr	r5, 2f	
ldmia	r5, {r7, r9, r10}	/* r7 = __proc_info_end r9 = __proc_info_begin r10 = 2b (이것은 아래에서 2 label의 주소가 될 것이다.)

```

        */
sub    r5, r5, r10      @ convert addresses
add    r7, r7, r5          @ to our address space
add    r10, r9, r5
mrc    p15, 0, r9, c0, c0      @ get processor id
1:   ldmia  r10, {r5, r6, r8}      @ value, mask, mmuflags
    and    r6, r6, r9      @ mask wanted bits
    teq    r5, r6
    moveq   pc, lr
    add    r10, r10, #36      @ sizeof(proc_info_list)
    cmp    r10, r7
    blt    1b
    mov    r10, #0          @ unknown processor
    mov    pc, lr

/* 여기에 적힌 곳을 참조하기 바란다.
 * Look in include/armv/procinfo.h and arch/arm/kernel/arch.[ch] for
 * more information about the __proc_info and __arch_info structures.
 */
2:   .long   __proc_info_end
        .long   __proc_info_begin
        long    2b
        .long   __arch_info_begin
        .long   __arch_info_end

/* 이 곳에서 2:로 정의된 곳에 저장된 정보를 이용하는 __lookup_architecture_type() 함수가 온다.*/

```

코드 932. head-armv.S의 __lookup_processor_type() 함수 정의

다시 앞으로 돌아가서, 먼저 r5에는 __proc_info_end부터 시작하는 주소가 있는 2f의 위치를 가져온다. 그리고나서, r5를 기준으로 해서 r7에는 __proc_info_end를, r9에는 __proc_info_begin을, r10에는 “2:”의 위치를 가져온다. r5는 연산이 일어난 후에 증가되어 다음번의 주소를 가르키게 될 것이다. 따라서, r5는 __arch_info_begin을 가르키고 있을 것이다. r5에서 r10을 빼서 r5에 집어 넣는 것으로 상대적으로 어느 위치에 있는지를 찾게 되며, 이 값을 r7에 더해서 __proc_info_end의 주소 더하기 “2:”에서 __arch_info_begin 사이의 공간 크기(sizeof(long)x3)를 하게된다. 이렇게 하고 나면, 또한 r9에 r5를 더하는 것은 __proc_info_begin의 주소 더하기 앞에 나온 __arch_info_begin 사이의 공간 크기를 더해서 r10으로 준다. 이렇게 하는 이유는 현재 MMU가 ON 상태가 아니므로 우리는 절대주소 공간에 대해서 접근하지 않고, 상대적인 offset으로 접근하기 위함이다. 이와 같은 계산을 마치고 나면, r10은 아래에서 보게될 __XXX_proc_info의 처음을 가르키게 될 것이다³⁹¹.

이젠 비교할 대상들에 대한 주소가 어디에 놓여 있는지를 찾았으므로, 실제적으로 processor의 ID를 읽어올 차례이다. r9를 프로세스의 ID를 읽는 곳에 사용해서 mrc p15, 0, r9, c0, c0에 준다. 이렇게 하고나면, r9에는 프로세스의 정보가 들어간다. 또한 r10에서부터 시작하는 주소에서 값을 읽어와서 차례로 r5, r6, r8에 저장한다. 읽어온 프로세스의 정보가 r9에 들어 있으므로, r6와 AND시켜서 다시 r6에 넣는다. 즉, r6는 mask값을 가지고 있다. 이 값을 가지고 r5와 비교한다. 즉, r5에는 프로세스의 정보가 들어 있는데, r9와 비교해서 어떤 프로세스인지를 확인할 수 있다. 참고로 coprocessor 15의 register 0에 있는 정보는 아래와 같다.

³⁹¹ 다음에 나올 __sa110_proc_info를 보면 된다.

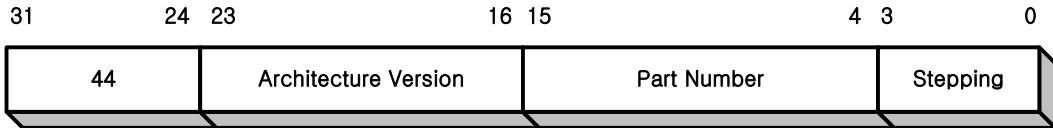


그림 129. ARM Coprocessor 15 – register 0 : Processor ID register

Field	Description
Architecture Version	ARM architecture version 01 = Version 4
Part Number	Part number A11 = SA1100
Stepping	Stepping revision of SA1100 1 = B stepping 2 = C stepping 8 = D stepping 9 = E stepping 11 = G stepping

표 114. SA1100의 Coprocessor Register 0의 필드 정의

따라서, r6에는 0xFFFFFFFF0값을 넣어서, stepping값을 지우며, r5에는 SA1100의 경우 0x4401A110가 들어가 것이다. 그리고, r8에는 0x00000C0E로 MMU(Memory Management Unit)의 flag이 들어간다. 이것은 나중에 함수가 복귀했을 때 사용한다.

계속 설명하자면, r10이 가르키는 곳에서 r5와 r6, r8로 데이터를 가져온다. r6와 r9를 AND시켜서 필요한 bit만을 골라내고, 이 값을 r5와 비교한다. 만약 같은 값을 가진다면, pc에 lr을 가지고와서 함수를 return하고, 그렇지 않다면, r10에 36을 더해서 이동시킨다. r10을 r7과 비교해서 r10이 r7보다 작은 값을 가진다면, 계속 위의 비교를 진행하고, 그렇지 않다면, r10에 0을 넣은 후, 함수를 복귀한다. 따라서, 만약 r10에 0값이 들어가 있다면 우리는 원하는 프로세스를 찾지 못한 것이다.

복귀시에 의미를 가지는 레지스터는 r8이 MMU flag으로 page table의 flag들로 사용되는 값을 가지며, r9에는 processor ID register에서 읽은 내용이, r10에는 __XXX_proc_info 구조에를 가르키는 값이다.

참고로, “2:”아래에 있는 __proc_info_XXX와 __arch_info_XXX는 ~/arch/arm/vmlinux-armv.lds.in에 정의되어 있다. 이것은 역시 ld 프로그램의 input으로 사용될 script파일이다. 아래와 같은 부분을 눈여겨 보도록 하자.

```
OUTPUT_ARCH(arm)
ENTRY(stext)
SECTIONS
{
    . = TEXTADDR;
    .init : { /* Init code and data */ _stext = .; __init_begin = .; *(.text.init) __proc_info_begin = .; *(.proc.info) __proc_info_end = .; __arch_info_begin = .; *(.arch.info) __arch_info_end = .; *(.data.init) . = ALIGN(16); __setup_start = .;
```

```

        *(.setup.init)
__setup_end = .;
__initcall_start = .;
        *(.initcall.init)
__initcall_end = .;
.= ALIGN(4096);
__init_end = .;
}
...

```

즉, TEXTADDR이 가르키는 부분에서 부터 시작하는 .init 섹션의 _stext와 __init_begin의 .text.init 아래에 바로 __proc_info_begin이 오게 되며, .proc.info, __proc_info_end, __arch_info_begin, .arch.info, __arch_info_end의 순으로 배열된다. 현재 우리가 원하는 것은 .proc.info 부분에 있는 것이다. .proc.info section의 경우 SA1100에 대한 것은 ~/arch/arm/mm/proc-sa110.S에 아래와 같이 정의되어 있다. 이곳에서 정의된 정보로 __proc_info_begin에서 __proc_info_end부분이 채워질 것이다.

```

...
.section ".proc.info", #alloc, #execinstr
.type    __sa110_proc_info,#object
__sa110_proc_info:
.long    0x4401a100
.long    0xfffffffff0
.long    0x00000c0e
b       __sa110_setup
.long    cpu_arch_name
.long    cpu_elf_name
.long    HWCAP_SWP | HWCAP_HALF | HWCAP_26BIT
.long    cpu_sa110_info
.long    sa110_processor_functions
.size    __sa110_proc_info, . - __sa110_proc_info
/* 0] 부분이 SA1100 __proc_info_XXX에서 우리가 찾는 부분이 될 것이다. */
.type    __sa1100_proc_info,#object
__sa1100_proc_info:
.long    0x4401a110
.long    0xfffffffff0
.long    0x00000c0e
b       __sa1100_setup
.long    cpu_arch_name
.long    cpu_elf_name
.long    HWCAP_SWP | HWCAP_HALF | HWCAP_26BIT
.long    cpu_sa1100_info
.long    sa1100_processor_functions
.size    __sa1100_proc_info, . - __sa1100_proc_info
/* 여기가 끝이다. */
.type    __sa1110_proc_info,#object
__sa1110_proc_info:
.long    0x6901b110
.long    0xfffffffff0
.long    0x00000c0e
b       __sa1100_setup
.long    cpu_arch_name
.long    cpu_elf_name
.long    HWCAP_SWP | HWCAP_HALF | HWCAP_26BIT
.long    cpu_sa1110_info
.long    sa1100_processor_functions
.size    __sa1110_proc_info, . - __sa1110_proc_info

```

이중에서 우리가 사용하게 될 것은 `_sa1100_proc_info`에 들어있는 내용이다. 이곳에 processor에 대한 각종 정보와 ID값 및 해당 processor에 맞는 설정 함수들이 정의되어 있다. 나중에 이것에 대해서 좀더 살펴볼 기회가 있을 것이다.

```
/* __lookup_processor_type()이 수행되고 난 이후이다.*/
    teq      r10, #0          @ invalid processor?
    moveq    r0, #'p'        @ yes, error 'p'
    beq     __error
    bl      __lookup_architecture_type
```

코드 933. head-armv.S(계속)

앞에서 설명했듯이 해당하는 프로세스를 찾지 못하면, r10에는 0이 들어있을 것이다. 따라서, 이것을 비교해서 0인 경우에는 r0에 p를 넣고 `__error`로 제어를 이동한다. 에러가 없다면, 이젠 `__lookup_architecture_type()`을 호출한다.

```
__error:
#endif CONFIG_DEBUG_LL
    mov      r8, r0          @ preserve r0
    adr      r0, err_str
    bl       printascii
    mov      r0, r8
    bl       printch

#endif
#ifndef CONFIG_ARCH_RPC
/*
 * Turn the screen red on a error - RiscPC only.
 */
    mov      r0, #0x02000000
    mov      r3, #0x11
    orr      r3, r3, r3, lsl #8
    orr      r3, r3, r3, lsl #16
    str     r3, [r0], #4
    str     r3, [r0], #4
    str     r3, [r0], #4
    str     r3, [r0], #4

#endif
1:      mov      r0, r0
        b       1b
#endif CONFIG_DEBUG_LL
err_str: .asciz   "\nError: "
        .align
#endif
```

코드 934. head-armv.S의 `__error()`함수의 정의

`__error()`함수가 하는 역할은 단순히 debugging 메시지를 보내고, 무한 루프를 도는 것 뿐이다. r0가 가질 수 있는 값은 예러의 종류를 가지는 문자값으로 “a”인 경우에는 Invalid Architecture, “p”인 경우에는 Invalid Processor, “i”인 경우에는 Invalid Calling Convention이다. 에러가 있는 경우에는 더 이상 진행하지 못하고 여기서 멈출 것이다.

```
/* 이곳은 이미 __lookup_processor_type에서 보았다.*/
2:      .long    __proc_info_end
        .long    __proc_info_begin
        .long    2b
        .long    __arch_info_begin
        .long    __arch_info_end
```

```
/* 아래에 __lookup_architecture_type도 위의 데이터를 접근한다.*/
__lookup_architecture_type:
    adr    r4, 2b
    ldmia  r4, {r2, r3, r5, r6, r7}      @ throw away r2, r3
    sub    r5, r4, r5                  @ convert addresses
    add    r4, r6, r5                  @ to our address space
    add    r7, r7, r5
1:   ldr    r5, [r4]                 @ get machine type
    teq    r5, r1
    beq    2f
    add    r4, r4, #SIZEOF_MACHINE_DESC
    cmp    r4, r7
    blt    1b
    mov    r7, #0                   @ unknown architecture
    mov    pc, lr
2:   ldmib  r4, {r5, r6, r7}        @ found, get results
    mov    r7, r7, lsr #18          @ pagetable byte offset
    mov    pc, lr
```

코드 935. head-armv.S의 __lookup_architecture_type() 함수의 정의

r4에 위에 있는 label 2의 주소를 가져온다. 이곳에서 r2, r3, r5, r6, r7을 읽어와서 이중에서 r2와 r3는 쓰지 않고 버린다. 즉, r5에는 label 2의 주소를, r6에는 __arch_info_begin을 r7에는 __arch_info_end의 주소를 가져온다. r4는 연산 이후에 다음번의 주소를 가지므로, __arch_info_end의 바로 다음 주소를 가질 것이다. r4에서 r5를 빼서 r5에 다시 넣는다. 즉, r5에는 label 2의 전체 크기가 들어갈 것이다. r5에 r6를 더해서 r4로 준다. 이렇게해서 r4에는 __arch_info_begin의 상대적인 시작주소가 들어간다. r7도 같은 식으로 계산해서, __arch_info_end의 상대적인 주소가 들어간다. 이곳에 들어갈 수 있는 내용을 정의하기 위해서는 ~/include/asm-arm/mach/arch.h에 있는 매크로가 사용된다. 간단히 들어갈 수 있는 내용은 아래에 보이는 machine_desc 구조체이다.

```
struct machine_desc {
    /* 앞의 첫 4개의 필드는 head-armv.S에서 사용한다.*/
    unsigned int          nr;           /* architecture number      */
    unsigned int          phys_ram;     /* start of physical ram   */
    unsigned int          phys_io;      /* start of physical io    */
    unsigned int          virt_io;      /* start of virtual io     */

    const char            *name;         /* architecture name        */
    unsigned int          param_offset; /* parameter page          */

    unsigned int          video_start;  /* start of video RAM      */
    unsigned int          video_end;    /* end of video RAM        */

    unsigned int          reserve_lp0 :1; /* never has lp0           */
    unsigned int          reserve_lp1 :1; /* never has lp1           */
    unsigned int          reserve_lp2 :1; /* never has lp2           */
    unsigned int          soft_reboot :1; /* soft reboot              */
    const struct tagtable *tagtable;    /* tag table                */
    int                  tagsize;       /* tag table size           */
    void                (*fixup)(struct machine_desc *,
                                struct param_struct *, char **,
                                struct meminfo *);

    void                (*map_io)(void); /* IO mapping function      */
    void                (*init_irq)(void); /* interrupt initialization */
};
```

코드 936. machine_desc 구조체의 정의

machine_desc 구조체의 정의 중에서 첫 4개의 필드가 head-armv.S에서 사용하는 부분이다. r1은 여전히 machine의 architecture ID번호를 가지고 있으므로 이 값과 r4가 가리키는 부분에서 읽어온 r5의 내용과 비교한다. 같다면, 2f로 진행하고, 그렇지 않다면, machine_desc 구조체의 크기만큼을 증가시킨다. 만약 이렇게 증가 시킨 값이 r70이 가지고 있는 __arch_info_end보다 크다면, r7에 0을 넣은 다음 복귀한다. 즉, r70이 0을 가지고 있다면, 에러가 발생한 것이다. 그렇지 않다면, 다시 1으로 돌아가서 다음에 있는 machine_desc구조체를 읽어와서 비교를 계속한다. 만약 해당하는 architecture type을 찾았다면(2f), r4를 기준으로 해서 다음에 있는 3개의 필드를 r5와 r6, r7에 각각 읽어온다. 즉, physical RAM의 시작주소, physical I/O의 시작주소, virtual I/O의 시작주소가 해당한다. 이중에서 r7에 저장된 virtual I/O의 시작주소는 18 bit를 오른쪽으로 shift해서 page table의 byte offset으로 활용한다. 이것을 마치면 함수는 복귀한다. 복귀시에 의미를 가진 레지스터들의 값을 다음과 같다. 즉, r5에는 RAM의 물리적인 시작주소를, r6에는 I/O space의 물리적인 주소를, r7에는 I/O를 위한 Page table내의 byte offset 값이 들어간다.

machine_desc구조체가 __arch_info_begin과 __arch_info_end에 들어간다고 했는데, 그럼 어떤 것들을 우리가 어떻게 정의해서 넣는지를 보아야 할 것이다. 먼저 사용하는 매크로가 정의된 ~/include/arm/mach/arch.h 파일을 보도록 하자.

```
#define MACHINE_START(_type,_name) \
const struct machine_desc __mach_desc_##_type\ \
__attribute__((__section__(".arch.info")))= { \
    nr:          MACH_TYPE_##_type, \
    name:        _name, \
#define MAINTAINER(n) \
#define BOOT_MEM(_pram,_pio,_vio) \
    phys_ram:    _pram, \
    phys_io:    _pio, \
    virt_io:    _vio, \
#define BOOT_PARAMS(_params) \
    param_offset: _params, \
#define VIDEO(_start,_end) \
    video_start: _start, \
    video_end:  _end, \
#define DISABLE_PARPORT(_n) \
    reserve_lp##_n: 1, \
#define BROKEN_HLT /* unused */ \
#define SOFT_REBOOT \
    soft_reboot: 1, \
#define FIXUP(_func) \
    fixup:      _func, \
#define MPIO(_func) \
    map_io:     _func, \
#define INITIRQ(_func) \
    init_irq:   _func, \
#define MACHINE_END \
}; \
#endif
```

코드 937. machine_desc구조체 정의를 위해서 사용하는 매크로들

먼저, 제일 앞에는 MACHINE_START()라는 매크로가 와야 한다. 이것으로 __mach_desc_XXX가 .arch.info section으로 정해지게 될 것이다. nr에는 MACH_TYPE_XXX가 들어가게되며, name으로는 넘겨준 _name이 들어간다. 나머지로 우리가 현재 관심을 두고 있는 것은 BOOT_MEM(_param, _pio, _vio)이다. 이것이 machine_desc구조체의 physical RAM의 시작주소와 physical I/O의 시작주소, virtual I/O의 시작주소를 표시한다.

그럼 위에서 정의한 매크로가 사용되는 것을 볼 차례이다. 이것을 위해서 ~/arch/arm/mach-sa1100/arch.c를 보도록 하자.

```

...
#ifndef CONFIG_SA1100_ASSABET
MACHINE_START(ASSABET, "Intel-Assabet")
    BOOT_MEM(0xc0000000, 0x80000000, 0xf8000000)
    FIXUP(fixup_sa1100)
    MAPIO(sa1100_map_io)
    INITIRQ(genarch_init_irq)
MACHINE_END
#endif
...

```

코드 938. machine_desc 구조체 정의를 위한 매크로의 사용 예

예를 들어서 보는 것은 SA1100를 바탕으로한 Intel의 Assabet 보드이다. MACHINE_START()에 ASSABET와 “Intel-Assabet”를 주었으므로, __mach_desc_ASSABET라는 machine_desc 구조체가 생성될 것이며, 이것의 name 필드에 “Intel-Assabet”가 들어갈 것이다. 또한 BOOT_MEM(0xC0000000, 0x80000000, 0xF8000000)을 주었으므로, physical RAM의 주소는 0xC0000000에 위치하며, 0x80000000이 physical I/O의 시작 주소가 되며, 0xF8000000이 virtual I/O의 시작 주소가 될 것이다. 이곳에서 설명하지 않은, FIXUP()과 MAPIO(), INITIRQ()는 각각이 시스템의 초기화 설정을 위해서 필요한 함수들로 나중에 다시 볼려질 것이다.

Samsung의 WVP를 위해서는 아래와 같은 부분을 추가해야 할 것이다. 즉, physical RAM 영역과 physical I/O 및 virtual I/O의 주소를 설정하는 일이다.

```

#ifndef CONFIG_SA1100_WVP
MACHINE_START(WVP, "Samsung-Wvp")
    BOOT_MEM(0xc0000000, 0x80000000, 0xf8000000)
    FIXUP(fixup_sa1100)
    MAPIO(sa1100_map_io)
    INITIRQ(genarch_init_irq)
MACHINE_END
#endif

```

코드 939. Samsung WVP용 machine_desc 구조체를 위한 매크로 사용 예

보면 알수 있듯이, 이것은 Assabet 보드와 동일하다. 기본적으로 Assabet 보드의 메모리 맵과 Samsung WVP의 메모리 맵이 비슷하기 때문이다. 물론 차이점이 있지만, 이곳에서는 이야기 하지 않기로 한다. FIXUP(), MAPIO(), INITIRQ()로 각각 fixup_sa1100(), sa1100_map_io(), genarch_init_irq() 함수를 사용한다.

teq	r7, #0	@ invalid architecture?
moveq	r0, #'a'	@ yes, error 'a'
beq	__error	
bl	__create_page_tables	
adr	lr, __ret	@ return address
add	pc, r10, #12	@ initialise processor @ (return control reg)
.type __switch_data, %object		
.long __mmap_switched		
.long SYMBOL_NAME(compat)		
.long SYMBOL_NAME(__bss_start)		
.long SYMBOL_NAME(_end)		
.long SYMBOL_NAME(processor_id)		
.long SYMBOL_NAME(__machine_arch_type)		
.long SYMBOL_NAME(cr_alignment)		
.long SYMBOL_NAME(init_task_union)+8192		

```

__ret:
    .type    __ret, %function
    ldr      lr, __switch_data
    mcr      p15, 0, r0, c1, c0
    mov      r0, r0
    mov      r0, r0
    mov      r0, r0
    mov      pc, lr

```

코드 940. head-armv.S(계속)

이전 processor의 정보와 architecture에 의존적인 정보를 대부분 다 구해왔다. 만약 r7이 0을 가진다면, r0에 ‘a’를 넣고, __error() 함수를 호출한다. 그렇지 않다면, __create_page_table() 함수를 호출해서 virtual address를 사용하기 위한 페이지 테이블의 생성을 시작한다. 페이지 테이블의 생성이 끝나면, lr에 __ret의 주소를 넣고, pc에 r100이 가르키는 곳에 12를 더한 값을 넣는다. 이것은 복귀해서 어디서 부터 실행할 것인가를 지정한 후 제어를 옮기는 것이다. __sa1100_setup() 함수가 호출될 것이며, 복귀해서는 __ret가 실행될 것이다.

__ret는 다시 lr에 __switch_data의 주소를 넣고, r0 값을 coprocessor 15의 control register 1의 c0에 쓸 것이다. 이렇게 하면, 드리어 MMU가 완전히 기동하게 되며, MMU control register에 완전히 기동될 때 까지는 세개의 instruction은 효력이 없다. 따라서, mov r0, r0가 3번이 반복되는 것은 아무런 의미가 없고, 단지 MMU가 완전히 기동되는 것을 기다리는 것 뿐이다. 이것을 마치면, 앞서 전장했던 lr를 다시 pc에 넣는다. 즉, __mmap_swapped로 제어가 옮겨간다. 이 함수는 이장의 마지막 부분에서 다시 보게 될 것이다.

```

__create_page_tables:
    pgtbl    r4, r5                                @ page table address
    /*
     * Clear the 16K level 1 swapper page table
     */
    mov      r0, r4
    mov      r3, #0
    add      r2, r0, #0x4000
1:   str      r3, [r0], #4
    str      r3, [r0], #4
    str      r3, [r0], #4
    str      r3, [r0], #4
    teq      r0, r2
    bne      1b
    /*
     * Create identity mapping for first MB of kernel to
     * cater for the MMU enable. This identity mapping
     * will be removed by paging_init()
     */
    krladr  r2, r4, r5                                @ start of kernel
    add      r3, r8, r2                                @ flags + kernel base
    str      r3, [r4, r2, lsr #18] @ identity mapping
    /*
     * Now setup the pagetables for our kernel direct
     * mapped region. We round TEXTADDR down to the
     * nearest megabyte boundary.
     */
    add      r0, r4, #(TEXTADDR & 0xff000000) >> 18 @ start of kernel
    add      r0, r0, #(TEXTADDR & 0x00f00000) >> 18
    sub      r2, r2, r5                                @ kernel ram offset
    sub      r0, r0, r2, lsr #18                          @ virtual ram start
    add      r3, r3, #4 << 20                            @ phys kernel + 4 MB

```

```

1:      add    r2, r8, r5          @ phys ram + flags
        str    r2, [r0], #4        @ fill pagetable
        add    r2, r2, #1 << 20
        cmp    r2, r3
        bcc    1b
        /*
         * Ensure that the first section of RAM is present.
         * we assume that:
         *   1. the RAM is aligned to a 256MB boundary
         *   2. the kernel is executing in the same 256MB chunk
         *       as the start of RAM.
         */
        bic    r0, r0, #0x0ff00000 >> 18  @ round down
        and    r2, r5, #0xf0000000@ round down
        add    r3, r8, r2          @ flags + rambase
        str    r3, [r0]
        bic    r8, r8, #0x0c        @ turn off cacheable
                                         @ and bufferable bits

```

코드 941. head-armv.S(계속) - __create_page_tables() 함수

먼저, r5가 물리적인 RAM의 주소를 가지고 있으므로 여기서 페이지 테이블의 물리적인 주소를 가져와서 r4에 넣는다. 사용하는 것은 macro로 정의된 pgtbl이다. r4를 r0에 넣고, r3에 0을 넣은 후, r2는 페이지 테이블의 크기인 0x4000을 더해서 넣는다. 이젠 r0부터 시작해서 r2까지의 영역에 있는 페이지 테이블을 전부 0으로 채운다. 즉, 페이지 테이블은 전부 0으로 초기화 된다.

이젠 r2에 커널의 시작주소를 가져온다. 매크로 krnladr를 사용했다. r8에는 현재 __lookup_processor_type에서 넘겨받은 페이지 테이블에 대한 flag를 가지고 있으므로 이것과 r2를 더해서 r3에 넣는다. r3를 다시 r2를 오른쪽으로 18bit shift한 값을 r4에 더해서, 그 위치에 r3를 저장한다. 따라서, 만약 커널이 0xC0000000에 있다고 하면, 이를 18bit 우측으로 shift하면, 0x3000이 되며, 이 값을 r4에 더하면, 0xC0003000이 된다. 즉, 이곳에 kernel에 대한 메모리 flag과 커널의 base주소가 들어간다. 18bit을 우측으로 shift하므로, 256KBytes를 하나의 페이지 테이블 엔트리가 가르킨다고 본다.

r4에 다시 TEXTADDR과 0xFF000000을 우측으로 18bit shift한 값을 AND시켜서 더해서 r0에 넣고, 다시 r0에 TEXTADDR과 0x00F00000을 우측으로 18bit shift한 값을 AND시켜서 더한다. 이렇게 하면 r0에는 실제적인 커널의 실행 시작주소에 대한 페이지 테이블의 옵셋이 들어간다. r2에는 현재 커널의 RAM offset을 가리키도록 만들기 위해서 r2에서 물리적인 RAM의 주소를 가지는 r5를 뺀다. 그리고나서 페이지 테이블 내에서의 위치를 알기 위해서 r2를 우측으로 18bit shift한 다음 r0에서 r2를 뺀다. r3은 4 MBytes의 끝을 나타내기 위해서 4를 왼쪽으로 20bit shift한다. 이젠 위에서 구한 r2를 r0가 가르키는 페이지 테이블의 한 엔트리로 채우는 일이 남았따. 하나의 entry가 1Mbytes의 공간을 가르키므로 r2는 loop를 돌기전에 1Mbytes만큼을 증가시켜준다. r2와 r3를 비교해서 r2가 r3보다 작은 동안 r2를 r0가 가르키는 page table entry로 계속 써넣는다.

r0는 이제 커널이 사용할 4Mbytes에 대한 공간에 대한 페이지 테이블을 다 설정했다. 이젠 RAM의 section부분이 실제로 존재하는지를 확인하는 일이 남았다. 다음과 같은 가정을 한다. RAM은 256MB 경계로 정렬되어 있으며, 커널은 RAM의 시작 부분과 같은 동일한 256Mbytes 덩이에서 실행된다고 본다. 먼저 r0에서 0x0FF00000을 우측으로 18bit shift한 값(=0x03FC0000)에 해당하는 bit를 지운다. 이것은 r0를 1Mbytes단위로 round down하는 효과가 있다. 다시 r5에 0xF0000000을 AND시켜서 r2에 넣는다. r2는 RAM의 base주소를 가지게 되며, 이 값에 r8이 가르키는 페이지 테이블 flag를 더해서 r3에 둔다. r3를 r0가 가르키는 페이지 테이블의 entry에 집어넣는다. r8이 가지는 페이지 테이블 flag에서 0x0C를 지운다. 이것은 cachability와 bufferability bit을 지워서 이 부분에 대해서는 caching도 buffering도 하지 말것을 알려주는 것이다.

```

#endif CONFIG_DEBUG_LL
/*
 * Map in IO space for serial debugging.
 * This allows debug messages to be output

```

```

        * via a serial console before paging_init.
    */
    add    r0, r4, r7
    rsb    r3, r7, #0x4000      @ PTRS_PER_PGD*sizeof(long)
    cmp    r3, #0x0800
    addge  r2, r0, #0x0800
    addlt  r2, r0, r3
    orr    r3, r6, r8
1:     str    r3, [r0], #4
    add    r3, r3, #1 << 20
    teq    r0, r2
    bne    1b
#endif(CONFIG_ARCH_NETWINDER) || defined(CONFIG_ARCH_CATS)
/*
 * If we're using the NetWinder, we need to map in
 * the 16550-type serial port for the debug messages
 */
    teq    r1, #MACH_TYPE_NETWINDER
    teqne  r1, #MACH_TYPE_CATS
    bne    1f
    add    r0, r4, #0x3fc0
    mov    r3, #0x7c000000
    orr    r3, r3, r8
    str    r3, [r0], #4
    add    r3, r3, #1 << 20
    str    r3, [r0], #4
1:
#endif
#endif
#endif CONFIG_ARCH_RPC
/*
 * Map in screen at 0x02000000 & SCREEN2_BASE
 * Similar reasons here - for debug. This is
 * only for Acorn RiscPC architectures.
 */
    add    r0, r4, #0x80          @ 02000000
    mov    r3, #0x02000000
    orr    r3, r3, r8
    str    r3, [r0]
    add    r0, r4, #0x3600        @ d8000000
    str    r3, [r0]
#endif
    mov    pc, lr                /* 함수의 복귀 */

```

코드 942. head-armv.S(계속) - __create_page_tables()함수(계속)

나머지 `__create_page_table()` 함수 부분은 `CONFIG_ARCH_XXX`와 `CONFIG_DEBUG_LL`에 의존적인 부분이며, 현재 우리가 진행하고 있는 부분에 대해서는 관련이 없다. `__create_page_table()` 함수의 수행을 끝내면 다시 호출된 위치로 복귀한다(`mov pc, lr`). 복귀한 이후에는 `__sa1100_setup()` 함수가 수행될 것이며, `_ret`가 이 함수의 복귀후에 수행된다. 그리고나서, 다시 아래에 나와 있는 `__mmap_switched()` 함수가 수행될 것이다.

```

/*
 * This code follows on after the page
 * table switch and jump above.
 *
 * r0 = processor control register

```

```

* r1 = machine ID
* r9 = processor ID
*/
.align 5
__mmap_switched:
    adr    r3, __switch_data + 4
    ldmia r3, {r2, r4, r5, r6, r7, r8, sp}@ r2 = compat
                           @ sp = stack pointer
    str    r12, [r2]

1:   mov    fp, #0          @ Clear BSS (and zero fp)
    cmp    r4, r5
    strcc fp, [r4],#4
    bcc    1b

    str    r9, [r6]          @ Save processor ID
    str    r1, [r7]          @ Save machine type
    orr    r0, r0, #2        @ .......A.
    bic    r2, r0, #2        @ Clear 'A' bit
    stmia r8, {r0, r2}       @ Save control register values
    b      SYMBOL_NAME(start_kernel)

```

코드 943. head-armv.S(계속) - __mmap_switched() 함수

이전 마지막으로 실제 커널의 시작부분인 start_kernel()함수로 제어를 옮기는 일이 남았다. 이것을 담당하는 것이 __mmap_switched()함수가 하는 일이다. __switch_data+4의 주소를 r3에 두고, r2, r4, r5, r6, r7, r8, sp를 가져온다. 즉, r2에는 compat를, r4에는 __bss_start, r5에는 _end, r6에는 processor_id, r7에는 __machine_arch_type을, r8에는 cr_alignment을, sp에는 init_task_union을 가져온다.

r12는 앞에서 #ifdef() #endif 절을 수행할 때 선택된 값이 저장된 것으로 compatibility값을 가진다. 이 값을 r2가 가리키는 compat부분에 넣어준다. fp에 0을 주고, r4와 r5를 비교해서 r4가 더 작은 값을 가진다면, r4가 가르키는 곳에 fp값을 넣어준다. r4는 __bss_start를 가리키며, r5는 _end를 가리키므로, 이 사에 들어가는 BSS영역에 대해서 0으로 초기화를 시켜주는 부분이다. r4에 4씩 증가시켜주며, r4가 r5보다 작을때까지 계속 수행한다.

r9에는 현재 processor의 ID를 가지고 있으므로, 이 값을 r6가 가르키는 processor_id값에 넣어준다. 다시 r1에는 machine type을 가지고 있으므로, 이 값을 r7이 가르키는 __machine_arch_type에 넣어준다. r0에는 현재 processor의 control register의 bit 설정을 가지고 있으므로 여기에 0x02(=0000 0010b)를 OR시켜서 r0에 넣고, 다시 r0에 0x02를 clear시켜서 r2에 넣는다. 0x02는 Coprocessor 15의 control register 1의 1bit 중에서 "A" bit를 제어하는 것으로 enable이 되며, alignment fault가 enable된다. 그렇지 않다면, disable이다. 이 r0와 r2값을 각각 r80이 가르키는 cr_alignment부분에 저장한다. 마지막으로 b SYMBOL_NAME(start_kernel)을 써서, 커널의 main entry routine인 start_kernel()함수로 제어를 옮긴다.

아직까지 다루지 않은 것은 __sa1100_setup()함수이다. 페이지 테이블의 생성이 끝나면, __ret의 주소를 lr로 이동시켜서 함수의 호출 후에 실행될 주소를 저장해 놓고, r10에 12를 더한 값을 pc에 준다. r10은 앞에서 processor의 정보를 구할 때 사용했는데, 구한 프로세스의 정보 구조체에서 12 bytes를 더하면, 우리의 경우에는 앞에서 본 __sa1100_proc_info의 4번째에 있는 __sa1100_setup() 함수가 해당될 것이다. 즉, 이곳으로 제어가 옮겨진다. __sa1100_setup() 함수의 수행을 마치면, __ret에서 수행이 진행된다. __sa1100_setup() 함수는 ~/arch/arm/mm/proc-sa110.S에 다음과 같이 정의되어 있다.

```

.align
.section ".text.init", #alloc, #execinstr
__sa1100_setup:  @ Allow read-buffer operations from userland
    mcr    p15, 0, r0, c9, c0, 5
__sa110_setup:
    mov    r0, #F_BIT | I_BIT | SVC_MODE

```

msr	cpsr_c, r0
mov	r0, #0
mcr	p15, 0, r0, c7, c7 @ invalidate I,D caches on v4
mcr	p15, 0, r0, c7, c10, 4 @ drain write buffer on v4
mcr	p15, 0, r0, c8, c7 @ invalidate I,D TLBs on v4
mcr	p15, 0, r4, c2, c0 @ load page table pointer
mov	r0, #0x1f @ Domains 0, 1 = client
mcr	p15, 0, r0, c3, c0 @ load domain access register
mrc	p15, 0, r0, c1, c0 @ get control register v4
bic	r0, r0, #0x0e00 @??r.....
bic	r0, r0, #0x0002 @a.
orr	r0, r0, #0x003d @DPWC.M
orr	r0, r0, #0x1100 @ ...I...S.....
mov	pc, lr

코드 944. __sa1100_setup() 함수의 정의

__sa1100_setup() 함수는 .text.init 섹션으로 정의되어 있다. 먼저 coprocessor 15에서 c9 control register에 opcode로 5를 적어서 사용자 모드에서 MCR access를 가능하도록 만든다. __sa1100_setup()이 하는 __sa1100_setup()과 같이 사용된다. 먼저 r0에 F_BIT과 L_BIT, SVC_MODE bit을 설정해서 cpsr_c에 넣는다. 즉, FIQ, IRQ를 disable시키고, supervisor mode로 진행하도록 한다. r0에 다시 0을 넣은 후, 이젠 coprocessor 15의 c7에 넣는다. 이것은 I cache와 D cache를 flush하는 역할을 한다. 이것을 마치면 다시 r0를 coprocessor 15의 c7 register에 opcode를 4를 주어 c10과 같이 써준다. 이것으로 write buffer를 소모(drain) 시킨다. 다시 instruction과 data를 위한 TLB(translation lookaside buffer)를 비우기 위해서 coprocessor 15의 c8 control register의 c7에 넣어준다. r4는 현재 page table에 대한 pointer를 가지고 있으므로, 이것으로 페이지 테이블의 주소를 초기화 해주기 위해서 coprocessor 15의 c2의 c0에 넣어준다. Coprocessor 15의 register 2는 아래와 같은 포맷을 가지고 있다.

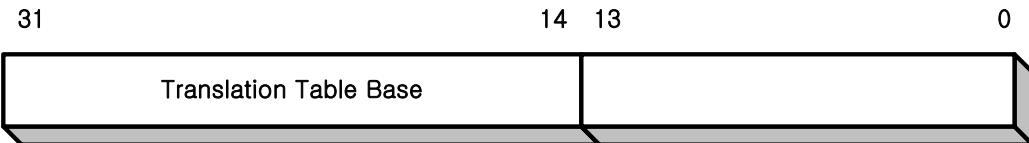


그림 130. Coprocessor 15의 register 2의 format.

Coprocessor 15의 register 2는 현재 active중인 level 1 page table의 base주소를 가진다. 이젠 페이지 테이블의 위치까지를 결정해 주었다.

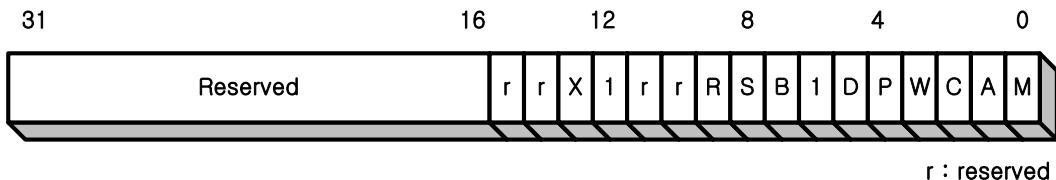
다음으로 할 일은 domain의 access register를 설정하는 일이다. 먼저 r0에 0x1F(=0001 1111b) 값을 넣고, 이것을 coprocessor 15의 register 1의 c0에 쓴다. 먼저 domain0이란 section이나 large page 혹은 small page들로 구성된 모음이며, ARM에서는 총 16개의 이러한 domain을 지원한다. 또한 각각의 domain에 대해서 domain access control register와 연계된 접근 제어를 하게 되는데, 2 bit로 이러한 접근 제어를 명시한다. 요약하면와 같은 접근 제어가 이루어진다.

Value	Access Types	Description
00	No access	모든 access는 domain fault를 유발한다.
01	Client	모든 access는 section이나 page의 descriptor에 있는 접근허가 bit에 대해서 검사가 이루어진다.
10	Reserved	예약된 값이다.
11	Manager	Section이나 page descriptor에 있는 접근허가 bit과 검사가 이루어지지 않으며, permission fault는 생기지 않는다.

표 115. Domain Access의 값

따라서, 32bit 값은 2bit씩 16등분으로 나누어지며, 각각의 16개의 domain에 대해서 위에서 정의한 접근허가 검사가 이루어진다. 앞에서는 하위 3개의 domain에 대해서만 접근 제어를 하며, 나머지에 대해서는 전부 domain fault를 일으키도록 설정하고 있다. 즉, 하나는 client로의 접근 제어를, 나머지는 manager 접근 검사가 이루어진다.

이것을 마치면 이전 coprocessor 15의 c0를 r0로 읽어온다. c1 control register는 MMU와 cache의 전체적인 control을 담당하는 레지스터이다. 이 값을 읽어서, 먼저 0x0E00(=0000 1110 0000 0000b) bit를 지우고, 다시 0x0002(=0000 0000 0000 0010b) bit를 지운다. 이 두개의 bit를 지우는 것은 R(ROM : bit 9)에 대한 access permission을 MMU가 간섭하지 않는다는 말이며, 또한 A(Alignment : bit 1) fault를 enable한다는 말이다. 이전 여기에 0x003D(=0000 0000 0011 1101)를 OR시킨다. 즉, On-chip MMU를 enable시키고, Data cache 및 Write buffer를 enable시킨다는 말이다. D를 enable(=1)시킨다는 말은 26-bit address exception 검사를 disable시킨다는 것이며, P를 1로 설정하면, Exception handler가 32-bit mode로 진입한다는 말이다. 설정되지 않으면, 26-bit mode로 진입한다. 다시 이곳에 0x1100(=0001 0001 0000 0000b)를 OR 시켜서 I bit과 S bit을 설정하는데, I bit은 Instruction cache를 enable시키고, S bit은 MMU에 의해서 access 검사가 이루어짐을 나타낸다. Coprocessor 15의 control register 1의 포맷은 [그림]와 같다.



r : reserved

그림 131. Coprocessor 15의 control register 1의 bit 필드 정의

이와 같은 설정이 다 끝나면, r0에는 MMU의 설정에 대한 bit이 다 정해져 있다. 이전 함수를 복귀하도록 한다(mov pc, lr).

```
...
/* __error() 함수가 정의되어 있다.*/
/* __lookup_processor_type() 함수가 정의되어 있다.*/
/* __proc_info_XXX와 __arch_info_XXX의 주소가 이곳에서 언급된다.*/
/* __lookup_architecture_type() 함수가 정의되어 있다.*/

```

코드 945. head-armv.S(계속)

head-armv.S의 나머지는 앞에서 설명한 코드들이 위치한다. 각각이 함수의 형태로 앞의 코드에서 이미 호출되었다. 이제 우리는 ~/init/main.c에 있는 start_kernel()을 보기로 하겠다.

17.7.3. start_kernel() 함수의 분석

start_kernel() 함수는 ~/init/main.c에 정의된 함수이다. 이 함수에서 하는 일은 크게 다음과 같이 나누어 볼 수 있다.

1. Architecture에 특수한(혹은 의존적인) 설정
2. Paging 시스템에 대한 초기화
3. Exception(혹은 trap)에 대한 초기화
4. Interrupt에 대한 초기화
5. Scheduler의 초기화
6. Timer의 초기화
7. Console의 초기화
8. Module들의 초기화
9. Cache와 buffer의 초기화
10. 메모리 시스템의 초기화

11. 파일 시스템의 초기화
12. init 시스템 프로세스의 생성

그럼 이제 부터는 `start_kernel()` 함수의 코드를 보면서 차례로 설명해 나가기로 하겠다. 앞에서 보여준 x86의 경우에도 시스템의 부팅이 마무리되면, 커널의 실제적인 시작을 알리는 `start_kernel()` 함수로 제어가 옮기기 때문에 참고하면 될 것이다.

```
asmlinkage void __init start_kernel(void)
{
    char * command_line;
    unsigned long mempages;
    extern char saved_command_line[];

/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
    lock_kernel();
    printk(linux_banner);
    setup_arch(&command_line);
    printk("Kernel command line: %s\n", saved_command_line);
```

코드 946. `start_kernel()` 함수의 정의

`start_kernel()` 함수는 가장 먼저 커널에 `lock`을 설정한다(`lock_kernel()`). 즉, 다른 부분에서 커널로의 진입을 막는다는 말이다. 그리고나서, `printk()` 함수로 `linux_banner`을 넘겨주어 호출한다. `linux_banner`은 `~/init/version.c`에 정의되어 있다. 이것을 마치면, 이전 `setup_arch()` 함수에 `command_line argument`을 넘겨주어서 호출한다. 성공적으로 `setup_arch()` 함수를 수행했다면, 다시 `printk()` 함수를 호출해서 `saved_command_line`을 쓸 것이다.

17.7.3.1. `setup_arch()` 함수의 분석

`setup_arch()` 함수는 `~/arch/arm/kernel/setup.c`에 정의된 함수로 `architecture`에 의존적인 설정을 하는 함수이다. 아래와 같다.

```
void __init setup_arch(char **cmdline_p)
{
    struct param_struct *params = NULL;
    struct machine_desc *mdesc;
    char *from = default_command_line;

    ROOT_DEV = MKDEV(0, 255);
    setup_processor();
    mdesc = setup_architecture(machine_arch_type);
    machine_name = mdesc->name;
    if (mdesc->soft_reboot)
        reboot_setup("s");
    if (mdesc->param_offset)
        params = phys_to_virt(mdesc->param_offset);
    /*
     * Do the machine-specific fixups before we parse the
     * parameters or tags.
     */
    if (mdesc->fixup)
        mdesc->fixup(mdesc, params, &from, &meminfo);
    if (params) {
        struct tag *tag = (struct tag *)params;
        /*
```

```

        * Is the first tag the CORE tag? This differentiates
        * between the tag list and the parameter table.
        */
if (tag->hdr.tag == ATAG_CORE)
    parse_tags(mdesc->tagtable, mdesc->tagsize, tag);
else
    parse_params(params);
}

```

코드 947. setup_arch()함수의 정의

setup_arch()함수는 먼저 root device를 생성한다. 전역변수로 ~/fs/super.c에 kdev_t으로 정의된 ROOT_DEV를 사용하며, major번호로는 0을 minor 번호로는 255를 주어서 MKDEV() 매크로를 사용해서 만든다. 그리고나서 setup_processor()를 호출해서 프로세서에 대한 설정을 해준다. 이것을 마치면 setup_architecture()함수를 호출해서 architecture에 의존적인 설정을 해주며, 여기서 machine에 대한 descriptor를 얻어온다. 만약 soft_reboot이 설정되어 있다면, reboot_setup()함수를 "s"를 파라미터로 넘겨주어서 호출하고, machine descriptor 필드에 param_offset(파라미터의 offset)이 있다면, 이것은 현재 물리적인 주소를 나타내므로 가상주소로 변화해서 params로 설정한다(phys_to_virt()). 또한, 만약 fixup routine이 설정되어 있다면, fixup() 함수를 수행하고, 이곳에서 meminfo값이 결정된다. meminfo는 ~/include/asm-arm/setup.h의 meminfo구조체로 정되어 있어 아래와 같다.

```

#define NR_BANKS 8
struct meminfo {
    int nr_banks;
    unsigned long end;
    struct {
        unsigned long start;
        unsigned long size;
        int node;
    } bank[NR_BANKS];
};
extern struct meminfo meminfo;

```

코드 948. meminfo구조체의 정의

meminfo구조체는 메모리 bank의 수와 어디가 몇개가 들어있는지, 그리고, 각각의 bank의 시작주소와 크기를 가지는 데이터 구조체이다. 현재로는 최대 8개의 메모리 bank가 올 수 있다. reboot_setup()함수는 ~/arch/arm/kenrel/process.c에 아래와 같이 간단히 reboot_mode에 하나의 character를 집어넣는 것으로 수행을 마친다.

```

static char reboot_mode = 'h';
int __init reboot_setup(char *str)
{
    reboot_mode = str[0];
    return 1;
}

```

코드 949. reboot_setup()함수의 정의

이젠 앞에서 본 machine descriptor에 대해서 fixup()을 어떻게 수행하는지 보기위해서 다시 보도록 하자. 아래와 같이 machine descriptor를 SA1100은 사용했다. 여기서는 Assabet인 경우와 Samsung WVP인 경우 두가지를 같이 보여주겠다.

```

...
#ifndef CONFIG_SA1100_ASSABET
MACHINE_START(ASSABET, "Intel-Assabet")

```

```

BOOT_MEM(0xc0000000, 0x80000000, 0xf8000000)
FIXUP(fixup_sa1100)
MAPIO(sa1100_map_io)
INITIRQ(genarch_init_irq)
MACHINE_END
#endif
#ifndef CONFIG_SA1100_WVP
MACHINE_START(WVP,"Samsung-Wvp")
    BOOT_MEM(0xc0000000, 0x80000000, 0xf8000000)
    FIXUP(fixup_sa1100)
    MAPIO(sa1100_map_io)
    INITIRQ(genarch_init_irq)
MACHINE_END
#endif
...

```

코드 950. Machine descriptor의 정의

즉, SA1100을 기반으로 한 Assabet과 Samsung WVP는 같은 machine_descriptor 구조체를 사용하고 있다. 따라서, fixup() 함수로는 fixup_sa1100() 함수가 호출될 것이며, I/O mapping하는 함수로는 sa1100_map_io() 함수를 IRQ를 initialize하는 함수로는 genarch_init_irq() 함수가 호출될 것이다.

setup_processor()함수 역시 setup.c에 정의된 함수로 아래와 같이 정의되며, 이전에 보았던 __proc_info_XXX를 검사해서 해당하는 CPU를 찾는 역할을 한다.

```

static void __init setup_processor(void)
{
    extern struct proc_info_list __proc_info_begin, __proc_info_end;
    struct proc_info_list *list;

    /*
     * locate processor in the list of supported processor
     * types. The linker builds this table for us from the
     * entries in arch/arm/mm/proc-*.S
     */
    for (list = &__proc_info_begin; list < &__proc_info_end ; list++)
        if ((processor_id & list->cpu_mask) == list->cpu_val)
            break;
    /*
     * If processor type is unrecognised, then we
     * can do nothing...
     */
    if (list >= &__proc_info_end) {
        printk("CPU configuration botched (ID %08x), unable "
              "to continue.\n", processor_id);
        while (1);
    }
    proc_info = *list->info;
#endif
#ifdef MULTI_CPU
    processor = *list->proc;
#endif
    printk("Processor: %s %s revision %d\n",
          proc_info.manufacturer, proc_info.cpu_name,
          (int)processor_id & 15);
    sprintf(system_utsname.machine, "%s%c", list->arch_name, ENDIANNESS);
    sprintf(elf_platform, "%s%c", list->elf_name, ENDIANNESS);
    elf_hwcap = list->elf_hwcap;
}

```

```
    cpu_proc_init();
}
```

코드 951. setup_processor() 함수의 정의

먼저 for loop를 돌면서 __proc_info_begin과 __proc_info_end사이에 processor_id와 list의 cpu_mask값을 AND시켜서 list의 cpu_val과 같은 것이 있을 때까지 진행한다. 만약 list가 proc_info_end보다 크다면, 잘못된 값을 가지는 processor이므로 while() loop를 무한번 수행한다. 즉, 시스템이 멈출 것이다. processor_id는 앞에서 head-armv.S에서 start_kernel()함수를 호출하기전에 저장된 값이다.

해당하는 프로세스를 찾았다면, 이젠 proc_info를 이용해서 이 프로세스의 정보를 가지는 자료구조를 가르키도록 설정한다. 만약 MULTI_CPU를 선택했다면, processor 변수를 사용해서 이것을 가르키도록 한다. 나머지는 해당하는 CPU의 정보등을 print하고, elf_hwcap에는 list의 elf_hwcap를 넣은 후, cpu_proc_init()함수를 호출한다. proc_info_list 구조체는 ~/include/asm-arm/procinfo.h에 아래와 같이 정의되어 있다.

```
...
struct proc_info_list {
    unsigned int      cpu_val;
    unsigned int      cpu_mask;
    unsigned long     __cpu_mmu_flags;      /* used by head-armv.S */
    unsigned long     __cpu_flush;          /* used by head-armv.S */
    const char        *arch_name;
    const char        *elf_name;
    unsigned int      elf_hwcap;
    struct proc_info_item *info;
#endif MULTI_CPU
    struct processor *proc;
#else
    void             *unused;
#endif
};

#endif /* __ASSEMBLY__ */
#define HWCAP_SWP           1      /* SWP를 지원한다. */
#define HWCAP_HALF          2      /* Half word를 지원한다. */
#define HWCAP_THUMB         4      /* Thumb mode를 지원한다. */
#define HWCAP_26BIT         8      /* 26 bit addressing을 지원한다. */
#endif
```

코드 952. proc_info_list 구조체의 정의

이 구조체는 앞에서 본 __sa1100_proc_info를 각각의 필드로 가지고 있을 것이다. 따라서, elf_hwcap 필드는 SA1100의 경우, HWCAP_SWP | HWCAP_HALF | HWCAP_26BIT으로 설정되어 있다. 마지막으로 수행되는 cpu_proc_init()함수는 single CPU를 사용하는 경우에는 ~/include/asm-arm/cpu-single.h에 아래와 같이 정의된다.

```
#ifdef __STDC__
#define __cpu_fn(name,x)  cpu_##name##x
#else
#define __cpu_fn(name,x)  cpu_/**/name/**/x
#endif
#define cpu_fn(name,x)     __cpu_fn(name,x)
...
#define cpu_proc_init       cpu_fn(CPU_NAME,_proc_init)
```

코드 953. cpu_proc_init() 함수의 정의

즉, CPU_NAME에 따라서 cpu_XXX_proc_init()함수가 호출됨을 알 수 있다. 우리가 사용하는 CPU_NAME 값은 ~/include/asm-arm/proc-fns.h에 sa1100으로 정해질 것이므로, 찾으려는 함수도 cpu_sa1100_proc_init() 함수가 될 것이다. 이 함수는 앞에서 본 proc-sa110.S 파일에 아래와 같이 정의되어 있다.

```
ENTRY(cpu_sa110_proc_init)
ENTRY(cpu_sa1100_proc_init)
    mov    r0, #0
    mcr    p15, 0, r0, c15, c1, 2          @ Enable clock switching
    mov    pc, lr
```

코드 954. cpu_sa1100_proc_init()함수의 정의

즉, r0에 0을 넣어서, coprocessor 15의 c15 control register에 opcode값으로 2를 주고, 보조 control register로는 c1을 사용해서 쓴다. 이것은 Clock switching을 enable하는 일을 한다³⁹².

setup_architecture()함수는 앞에서 보았던 __arch_info_begin과 __arch_info_end사이에 있는 데이터를 검색해서 해당하는 architecture를 가지는 자료구조를 얻기위해서 사용하는 함수이다. 아래같이 정의된다.

```
static struct machine_desc * __init setup_architecture(unsigned int nr)
{
    extern struct machine_desc __arch_info_begin, __arch_info_end;
    struct machine_desc *list;

    /*
     * locate architecture in the list of supported architectures.
     */
    for (list = &__arch_info_begin; list < &__arch_info_end; list++)
        if (list->nr == nr)
            break;
    /*
     * If the architecture type is not recognised, then we
     * can do nothing...
     */
    if (list >= &__arch_info_end) {
        printk("Architecture configuration botched (nr %d), unable "
              "to continue.\n", nr);
        while (1);
    }
    printk("Architecture: %s\n", list->name);
    if (compat)
        printk(KERN_WARNING "Using compatibility code "
              "scheduled for removal in v%d.%d.%d\n",
              compat >> 24, (compat >> 12) & 0x3ff,
              compat & 0x3ff);
    return list;
}
```

코드 955. setup_architecture()함수의 정의

즉, 넘겨받은 값과 일치하는 데이터 구조체에 대한 것을 __arch_info_begin에서 찾기를 시작해서 __arch_info_end까지를 살펴본다. 여기서 데이터 구조라는 것은 해당하는 machine에 대한 descriptor가 된다. 발견된다면, list가 그 machine_desc의 자료구조에 대한 포인터를 가지게 될 것이며, 이 값이 복귀값으로 주어진다.

```
mdesc = setup_architecture(machine_arch_type);
```

³⁹² Intel의 SA1100의 manual을 참고하기 바란다.

```

machine_name = mdesc->name;
if (mdesc->soft_reboot)
    reboot_setup("s");
if (mdesc->param_offset)
    params = phys_to_virt(mdesc->param_offset);
/*
 * Do the machine-specific fixups before we parse the
 * parameters or tags.
 */
if (mdesc->fixup)
    mdesc->fixup(mdesc, params, &from, &meminfo);

```

코드 956. setup_arch()함수의 정의(계속)

다시 설명을 앞으로 돌아가서 setup_architecture()를 호출해서 machine에 대한 descriptor까지를 얻을 수 있었다. machine_name을 machine_desc자료구조의 name필드로 채워준다. 만약 machine_desc의 soft_reboot필드가 설정되어 있다면, reboot_setup()에 “s”를 넘겨주어서 호출한다.

```

static char reboot_mode = 'h';
int __init reboot_setup(char *str)
{
    reboot_mode = str[0];
    return 1;
}
__setup("reboot=", reboot_setup);

```

코드 957. reboot_setup()함수의 정의

reboot_setup()함수의 정의는 ~/arch/arm/kernel/process.c에 나와 있으며, 단순히 reboot_mode에 전달받은 string을 설정하는 역할을 할 뿐이다.

machine_desc구조체에 param_offset이 설정된 경우에는 params에 param_offset이 가지는 값을 가상주소로 변환해서 넣는다. params는 parameter에 대한 포인터를 가지게 될 것이다. machine_desc구조체에 fixup이 설정된 경우에는 다시 fixup() 함수를 호출한다. 우리는 앞에서 이미 보았지만, 이곳에 fixup_sa1100() 함수를 선언해 두었기에 이것이 호출될 것이다. 정의는 ~/arch/arm/mach-sa1100/arch.c에 나와 있다.

```

#define SET_BANK(__nr,__start,__size) \
    mi->bank[__nr].start = (__start), \
    mi->bank[__nr].size = (__size), \
    mi->bank[__nr].node = (((unsigned)(__start) - PHYS_OFFSET) >> 27)

static void __init fixup_sa1100(struct machine_desc *desc, struct param_struct *params,
                                char **cmdline, struct meminfo *mi)
{
    pm_power_off = sa1100_power_off;

    if (machine_is_assabet()) {
        /*
         * On Assabet, we must probe for the Neponset board *before*
         * paging_init() has occurred to actually determine the amount
         * of RAM available.
         */
        extern void map_sa1100_gpio_regs(void);
        extern void get_assabet_scr(void);
        map_sa1100_gpio_regs();
        get_assabet_scr();
        SET_BANK( 0, 0xc0000000, 32*1024*1024 );
    }
}

```

```

        mi->nr_banks = 1;
        if (machine_has_neponset()) {
            printk("Neponset expansion board detected\n");
            /*
             * Note that Neponset RAM is slower...
             * and still untested.
             * This would be a candidate for
             * _real_ NUMA support.
             */
            //SET_BANK( 1, 0xd0000000, 32*1024*1024 );
            //mi->nr_banks = 2;
        }
        ...
        else if (machine_is_brutus()) {
            SET_BANK( 0, 0xc0000000, 4*1024*1024 );
            SET_BANK( 1, 0xc8000000, 4*1024*1024 );
            SET_BANK( 2, 0xd0000000, 4*1024*1024 );
            SET_BANK( 3, 0xd8000000, 4*1024*1024 );
            mi->nr_banks = 4;
            ROOT_DEV = MKDEV(RAMDISK_MAJOR,0);
            setup_ramdisk( 1, 0, 0, 8192 );
            setup_initrd( __phys_to_virt(0xd8000000), 3*1024*1024 );
        }
        ...
        /* 이곳에 Samsung WVP를 위한 부분을 추가하면 될 것이다.*/
        ...
    }
}

```

코드 958. fixup_sa1100()함수의 정의

fixup_sa1100()함수가 하는 일은 메모리 BANK가 mapping된 주소에 대한 페이지 테이블에 들어갈 내용을 결정하는 것이다. 또한 메모리 BANK의 수와 root device를 결정하고, ramdisk에 대한 설정 및 initial ramdisk에 대한 설정을 한다. 필요하다면 power management함수도 설정한다. 이곳에서는 전형적인 것과 차이가 보이는 부분만을 담아 두었다. 이곳에 필요하다면 자신이 가지고 있는 board에 대한 설정을 넣어주어야 할 것이다. 예를 들어서 Samsung WVP의 경우에는 아래와 같이 하면 될 것이다.

```

...
} else if (machine_is_wvp()) {
    SET_BANK( 0, 0xc0000000, 16*1024*1024 );
    SET_BANK( 1, 0xc8000000, 16*1024*1024 );
    mi->nr_banks = 2;
    ROOT_DEV = MKDEV(RAMDISK_MAJOR,0);
    setup_ramdisk( 1, 0, 0, 8192 );
    setup_initrd( 0xc0200000, 4*1024*1024 );
}
...

```

코드 959. Samsung WVP를 위한 fixup_sa1100()함수의 수정 부분

즉, 메모리 BANK1은 0xC0000000에 mapping되며, 16Mbytes를 가지고 있으며, 메모리 BANK2는 0xC8000000에 mapping되며, 똑같이 16Mbytes를 가진다. 두개의 메모리 BANK를 가지므로 nr_banks에는 2를 주고, ROOT_DEV의 값으로는 RAMDISK_MAJOR를 major번호로, 그리고 minor번호로는 0을 주었다. RAM disk는 8192bytes를 가진다. initial RAM disk는 0xC0200000에 mapping이 되며, 4Mbytes를 사용한다.

`setup_ramdisk()`함수와 `setup_initrd()`함수는 `~/arch/arm/kernel/setup.c`에 아래와 같이 정의되어 있으며, 각각 RAM disk에 대한 초기화와 initial RAM disk에 대한 설정을 수행한다.

```
void __init setup_ramdisk(int doload, int prompt, int image_start, unsigned int rd_sz)
{
#ifdef CONFIG_BLK_DEV_RAM
    extern int rd_doload, rd_prompt, rd_image_start, rd_size;

    rd_image_start = image_start;
    rd_prompt = prompt;
    rd_doload = doload;
    if (rd_sz)
        rd_size = rd_sz;
#endif
}
/*
 * initial ram disk
 */
void __init setup_initrd(unsigned int start, unsigned int size)
{
#ifdef CONFIG_BLK_DEV_INITRD
    if (start == 0)
        size = 0;
    initrd_start = start;
    initrd_end = start + size;
#endif
}
```

코드 960. `setup_ramdisk()`함수와 `setup_initrd()`함수의 정의

먼저 `setup_ramdisk()`는 `CONFIG_BLK_DEV_RAM`이 커널 컴파일에서 설정된 경우에 실행되는 함수로, 앞에서 block device를 볼 때, RAM disk에 관련된 곳에서 정의된 `rd_image_start`, `rd_prompt`, `rd_doload`, `rd_size` 변수를 설정하는 일을 한다. 각각의 변수가 가지는 의미는 `rd_image_start`는 image의 시작 block의 번호를, `rd_prompt`는 RAM disk를 위해서 prompt를 할 것인지 말것인지를, `rd_doload`는 RAM disk를 load할 것인지 말 것인지를, `rd_size`는 RAM disk의 크기를 나타낸다. 앞에서 넘겨주는 parameter에 의해서 RAM disk는 0번째³⁹³ block에 들어가게 되며, load할 것이고, prompt는 보이지 않게 만들고, 크기는 8192를 가질 것이다.

`setup_initrd()`함수 역시 `CONFIG_BLK_DEV_INITRD`가 설정된 경우에 실행되는 함수로 `initrd_start`와 `initrd_end` 변수를 초기화하는 역할을 한다. 앞에서 넘겨준 parameter에 의해서 initial RAM disk의 시작주소는 `0xC02000000`이 되며, 끝번지는 `0xC0200000+0x00400000(0xC0600000)`가 될 것이다.

```
if (params) {
    struct tag *tag = (struct tag *)params;
    /*
     * Is the first tag the CORE tag? This differentiates
     * between the tag list and the parameter table.
     */
    if (tag->hdr.tag == ATAG_CORE)
        parse_tags(mdesc->tagtable, mdesc->tagsize, tag);
    else
        parse_params(params);
}
if (meminfo.nr_banks == 0) {
```

³⁹³ 0부터 시작해서 번호를 가진다.

```

meminfo.nr_banks    = 1;
meminfo.bank[0].start = PHYS_OFFSET;
meminfo.bank[0].size = MEM_SIZE;
}
init_mm.start_code = (unsigned long) &_text;
init_mm.end_code = (unsigned long) &_etext;
init_mm.end_data = (unsigned long) &_edata;
init_mm.brk       = (unsigned long) &_end;

memcpy(saved_command_line, from, COMMAND_LINE_SIZE);
saved_command_line[COMMAND_LINE_SIZE-1] = '\0';
parse_cmdline(&meminfo, cmdline_p, from);

```

코드 961. setup_arch()함수의 정의(계속)

앞에서 계산한 parameter들을 가지는 주소가 있다면, tag으로 이것을 가르키도록 만든다. 먼저 param_struct 구조체부터 보기로 하자. 정의는 ~/include/asm-arm/setup.h에 있다.

```

#define COMMAND_LINE_SIZE 1024
struct param_struct {
    union {
        struct {
            unsigned long page_size;           /* 0 - 페이지 크기 */
            unsigned long nr_pages;          /* 4 - 페이지의 수 */
            unsigned long ramdisk_size;       /* 8 - RAM disk의 크기 */
            unsigned long flags;              /* 12 - flags */

#define FLAG_READONLY 1
#define FLAG_RDLOAD 4
#define FLAG_RDPROMPT 8
            unsigned long rootdev;           /* 16 - Root device */
            unsigned long video_num_cols;   /* 20 - 비디오 모드에서 column의 수 */
            unsigned long video_num_rows;   /* 24 - 비디오 모드에서 row의 수 */
            unsigned long video_x;          /* 28 - video의 x축 resolution */
            unsigned long video_y;          /* 32 - video의 y축 resolution */
            unsigned long memc_control_reg; /* 36 - 메모리 control register */
            unsigned char sounddefault;     /* 40 - 사운드 디폴트 값 */
            unsigned char adfsdrives;       /* 41 - ADFS 드라이버 */
            unsigned char bytes_per_char_h; /* 42 - Horizontal 문자당 byte수 */
            unsigned char bytes_per_char_v; /* 43 - Vertical 문자당 byte수 */
            unsigned long pages_in_bank[4]; /* 44 - BANK당 페이지의 수 */
            unsigned long pages_in_vram;    /* 60 - Video RAM에 있는 페이지의 수 */
            unsigned long initrd_start;     /* 64 - INITRD의 시작주소 */
            unsigned long initrd_size;      /* 68 - INITRD의 크기 */
            unsigned long rd_start;         /* 72 - RAM disk의 시작주소 */
            unsigned long system_rev;       /* 76 - System revision 번호 */
            unsigned long system_serial_low; /* 80 - 시스템의 serial low speed */
            unsigned long system_serial_high; /* 84 - 시스템의 serial high speed */
            unsigned long mem_fclk_21285;    /* 88 - 메모리 frequency clock */
        } s;
        char unused[256];
    } u1;
    union {
        char paths[8][128];             /* 압축된 커널의 path name */
        struct {
            unsigned long magic;          /* Magic number */
        }
    }
}
```

```

        char n[1024 - sizeof(unsigned long)]; /* 1024 bytes padding for magic number */
    } s;
} u2;
char cmdline[COMMAND_LINE_SIZE]; /* Command line option */
};

```

코드 962. param_struct구조체의 정의

tag구조체는 위에서 정의한 param_struct구조체의 각 부분을 더 자세히 구분 짓기 위해서 사용하는 구조체로 아래와 같은 정의를 가지며, param_struct구조체와 같이 정의되어 있다.

```

struct tag {
    struct tag_header hdr;
    union {
        struct tag_core          core;
        struct tag_mem32         mem;
        struct tag_videotext     videotext;
        struct tag_ramdisk        ramdisk;
        struct tag_initrd         initrd;
        struct tag_serialnr      serialnr;
        struct tag_revision       revision;
        struct tag_videofb        videofb;
        struct tag_cmdline        cmdline;
        /*
         * Acorn specific
         */
        struct tag_acorn          acorn;
        /*
         * DC21285 specific
         */
        struct tag_memclk         memclk;
    } u;
};

```

코드 963. tag구조체의 정의

따라서, 앞에서 param_struct로 정의된 포인터를 tag구조체를 가르키는 포인터로 casting해서 이 값을 tag으로 주었다. 만약 tag의 hdr의 tag필드가 ATAG_CORE인 경우에는 parse_tags()에 machine_desc구조체의 tagtable과 tagsize, tag를 넘겨서 호출하고, 그렇지 않을 경우에는 parse_params()에 params를 넘겨서 호출한다. param_tags()함수와 parse_params()함수는 command line에서 주어진 parameter를 parsing해서 위에서 정의한 param_struct의 항목을 정하는 일을 한다.

이전 메모리 정보를 구할 차례이다. meminfo구조체를 사용한다. meminfo구조체는 ~/include/arm-setup.h에 아래와 같이 정의되어 있다.

```

/*
 * Memory map description
 */
#define NR_BANKS 8

struct meminfo {
    int nr_banks;           /* Memory BANK의 수 */
    unsigned long end;       /* 아래에 있는 BANK에 대한 array의 크기 */
    struct {
        unsigned long start;   /* 메모리 BANK의 start 주소 */
        unsigned long size;    /* 메모리 BANK의 크기 */
        int node;              /* node번호 */
    }
};

```

```

    } bank[NR_BANKS];
};

extern struct meminfo meminfo;
/* setup.c에 정의되어 있다.*/
static struct meminfo meminfo __initdata = { 0, };

```

코드 964. meminfo구조체의 정의

meminfo의 BANK의 수가 0이라며, nr_banks에는 1을 주고, 시작주소와 크기를 PHYS_OFFSET으로, MEM_SIZE(=16x1024x1024=16Mbytes)으로 둔다. init_mm.start_code에는 _text를 두도록 하자. 이곳이 init 메모리의 code의 시작부분을 가르킨다. end_code와 end_data, brk필드는 각각 _etext와 _edata, _end를 둔다. init 메모리의 코드 데이터에 대한 부분이 얼마나 차지 하는지를 저장하는 것이다.

이젠 command line이 있는 곳에서 saved_command_line으로 command line을 복사해서 저장한다(memcpy()). 마지막을 나타내기 위해서 “\0”을 마지막에 채운다. 이젠 넘겨진 command line을 해석(parsing)하는 parse_cmdline()함수를 호출한다. meminfo와 commandline_p 및 원래의 command line이 있는 from(default_command_line을 가르킨다.)을 넘겨주도록 한다.

parse_cmdline()함수가 하는 일은 메모리의 크기와 관련된 것이다. 따라서, 주어진 command line parameter값들로 앞에서 간략화 설정한 meminfo부분을 채워줄 것이다.

```

bootmem_init(&meminfo);
paging_init(&meminfo, mdesc);
request_standard_resources(&meminfo, mdesc);
...

```

코드 965. setup_arch()함수의 정의(계속)

이제 남은 것은 bootmem_init() 함수와 paging_init() 함수, 그리고, machine_desc구조체로 앞에서 구한 mdesc를 이용해서 request_standard_resource() 함수를 호출하는 것이다. 또한, architecture에 의존적인 인터럽트 초기화 루틴을 init_arch_irq에 두도록 한다. 우리의 경우는 genarch_init_irq()이 해당할 것이다³⁹⁴. 먼저 bootmem_init()함수부터 보기로 하자.

bootmem_init()함수는 모든 노드³⁹⁵를 위한 부트 메모리의 할당자를 초기화 시키는 것으로 architecture에 의존적인 초기화시에 호출되는 함수이다. 정의된 곳은 ~/arch/arm/mm/init.c이다.

```

void __init bootmem_init(struct meminfo *mi)
{
    struct node_info node_info[NR_NODES], *np = node_info;
    unsigned int bootmap_pages, bootmap_pfn, map_pg;
    int node, initrd_node;

    bootmap_pages = find_memend_and_nodes(mi, np);
    bootmap_pfn = find_bootmap_pfn(0, mi, bootmap_pages);
    initrd_node = check_initrd(mi);
    map_pg = bootmap_pfn;
    np += numnodes - 1;
    for (node = numnodes - 1; node >= 0; node--, np--) {
        if (np->end == 0) {
            if (node == 0)
                BUG();
            continue;
        }
        init_bootmem_node(NODE_DATA(node), map_pg, np->start, np->end);
    }
}

```

³⁹⁴ machine_desc 구조체의 INITIRQ에 해당한다.

³⁹⁵ 여기서 노드(node)란 특정 RAM의 bank에서 덩어리로 존재하는(start와 end 값을 가지는) 메모리 영역을 말한다.

```

        free_bootmem_node_bank(node, mi);
        map_pg += np->bootmap_pages;
        if (node == 0)
            reserve_node_zero(bootmap_pfn, bootmap_pages);
    }
#endif CONFIG_BLK_DEV_INITRD
    if (initrd_node >= 0)
        reserve_bootmem_node(NODE_DATA(initrd_node), __pa(initrd_start),
                             initrd_end - initrd_start);
#endif
    if (map_pg != bootmap_pfn + bootmap_pages)
        BUG();
}

```

코드 966. bootmem_init()함수의 정의

find_memend_and_nodes()함수는 메모리 정보를 검사해서, 메모리의 마지막과, node의 수, 각각의 node의 PFN(Page Frame Number), boot memory의 bitmap pages의 수를 알려주는 역할을 한다. find_bootmap_pfn()함수는 boot memory가 mapping된 PFN을 돌려주는 함수이다. 이를 각각의 함수의 return값을 bootmap_pages와 bootmap_pfn으로 둔다. check_initrd()함수는 initrd의 node번호를 결정하는 것으로 만약 물리적인 메모리를 벗어난다면, 에러로 처리될 것이다. 넘겨받은 값은 다시 initrd_node에 저장한다. map_pg는 앞에서 구한 bootmap_pfn을 넣는다. np는 앞에서 node_info변수를 가르키는 포인터이며, node의 개수에 -1을 한 값을 원래의 값에 더해서 제일 마지막의 위치를 가지도록 만든다. 참고로 node_info구조체는 필드로 start와 end, 그리고 boot memory map을 가진다.

for loop를 돌면서 boot memory에 대한 초기화를 한다. 이것은 node의 마지막에서부터 시작해서 줄어드는 방향으로 진행하도록 한다. 마지막의 node번호 0은 나중에 다른 node들을 설정(setup)하기 위해서 예약해 둔다(reserve_node_zero()). 먼저, node의 end필드가 0이라면, node가 0인지를 확인하고 맞다면, BUG()를 호출해서 bug있음을 알리고 다음번 loop로 진행한다(continue). init_bootmem_node()함수는 boot memory에 대한 page data를 초기화 하는 역할을 하며, free_bootmem_node_bank()함수는 node에서 모든 사용 가능한 RAM을 boot memory allocator와 같이 등록하는 역할을 한다. 이렇게 처리된 node의 bootmap_pages 필드는 map_pg에 더해진다. 만약 node번호가 0이라면, 해당하는 boot memory의 PFN과 page수가 예약된다.

만약 COFIG_BLK_DEV_INITRD가 선언되었다면, initrd_node를 확인해서 해당하는 부분의 boot memory의 node를 예약한다(reserve_bootmem_node()). 마지막으로 앞에서 구한 map_pg와 boot memory의 PFG과 boot memory의 page수를 더한 값을 비교해서 다르다면, BUG()를 호출해서 에러가 있음을 나타낸다.

이곳까지 진행했다면, boot memory에 대한 초기화가 다 이루어졌다. 이젠 나머지들에 대해서 페이징 테이블을 생성하고, 초기화 하는 일을 해주어야 할 것이다. 이것을 해주는 것이 paging_init()함수이다.

17.7.3.2. paging_init() 함수의 분석

paging_init()함수가 정의된 곳도 앞의 bootmem_init()와 같은 파일이다. 즉, ~/arch/arm/mm/init.c를 참조하기 바란다. 이곳에서 해주는 일은 페이징을 위한 초기화를 하는 것으로 아래와 같다.

```

void __init paging_init(struct meminfo *mi, struct machine_desc *mdesc)
{
    void *zero_page, *bad_page, *bad_table;
    int node;

    memcpy(&meminfo, mi, sizeof(meminfo));
    /*
     * allocate what we need for the bad pages.
     * note that we count on this going ok.
     */
    zero_page = alloc_bootmem_low_pages(PAGE_SIZE);
    bad_page = alloc_bootmem_low_pages(PAGE_SIZE);
    bad_table = alloc_bootmem_low_pages(TABLE_SIZE);
}

```

```

/*
 * initialise the page tables.
 */
memtable_init(mi);
if (mdesc->map_io)
    mdesc->map_io();
flush_tlb_all();

```

코드 967. paging_init() 함수의 정의

paging_init() 함수가 하는 일은 page table에 대한 설정과, zone memory map에 대한 초기화 및 zero page와 bad page, bad page table들에 대한 설정을 한다. 이에 대해서 차근차근 알아보도록 하겠다.

먼저 앞에서 얻어온 meminfo 구조체를 mi에 복사한다. 그리고, zero page와 bad page, bad page의 table을 위한 것을 boot memory의 하위번지에서 PAGE_SIZE 만큼을 할당받는다(alloc_bootmem_low_pages()). memtable_init() 함수는 initial mapping에 대해서 설정한다. 이러한 mapping을 유지하는데는 zero page로 할당한 page를 사용할 것이며, traps_init()에 의해서 나중에 mapping이 overwrite될 것이다. 이러한 mapping은 반드시 가상 주소 순으로 있어야 한다. 만약 memory descriptor(mdesc)가 I/O에 대한 mapping을 가지고 있다면, mdesc->map_io()를 호출해서 I/O mapping을 처리하도록 한다. 이것을 마치고 나면, TLB(Translation Lookaside Buffer)를 완전히 비우기 위해서 flush_tlb_all() 함수를 호출한다.

memtable_init() 함수를 잠시 보도록 하자. ~/arch/arm/mm/mm-armv.c에 아래와 같이 정의되어 있다. 이 함수는 map_desc 구조체를 할당받아서, 이를 초기화 하는 일을 한다.

```

void __init memtable_init(struct meminfo *mi)
{
    struct map_desc *init_maps, *p, *q;
    unsigned long address = 0;
    int i;

    init_maps = p = alloc_bootmem_low_pages(PAGE_SIZE);
    p->physical    = virt_to_phys(init_maps);
    p->virtual     = 0;
    p->length      = PAGE_SIZE;
    p->domain      = DOMAIN_USER;
    p->prot_read   = 0;
    p->prot_write  = 0;
    p->cacheable   = 1;
    p->bufferable  = 0;
    p++;

    for (i = 0; i < mi->nr_banks; i++) {
        if (mi->bank[i].size == 0)
            continue;
        p->physical    = mi->bank[i].start;
        p->virtual     = __phys_to_virt(p->physical);
        p->length      = mi->bank[i].size;
        p->domain      = DOMAIN_KERNEL;
        p->prot_read   = 0;
        p->prot_write  = 1;
        p->cacheable   = 1;
        p->bufferable  = 1;
        p++;
    }
}

```

코드 968. memtable_init() 함수의 정의

`init_maps` 변수와 `p`을 `PAGE_SIZE`만큼을 `boot memory`의 하위 `page`부분에서 할당 받는다. 각각은 `map_desc` 구조체에 대한 포인터로 메모리 mapping에 대한 descriptor의 역할을 하는 구조체를 가르킨다. `map_desc` 구조체의 정의는 `~/include/asm-arm/mach/map.h`에 아래와 같이 정의되어 있다.

```
struct map_desc {
    unsigned long virtual;           /* 가상 주소공간의 시작 주소 */
    unsigned long physical;          /* 물리적인 주소공간의 시작 주소 */
    unsigned long length;            /* 주소 공간의 길이 */
    int domain:4;                   /* Domain type */
    prot_read:1;                    /* Read protection bit */
    prot_write:1;                   /* Write Protection bit */
    cacheable:1;                    /* Cacheable bit */
    bufferable:1;                   /* Bufferable bit */
    last:1;                         /* Last bit */
};

#define LAST_DESC \
{ last: 1 }
```

코드 969. `map_desc` 구조체의 정의

즉, 메모리 공간의 가상 주소와 시작주소, 그리고, 그 길이를 가지고 있으며, 이 주소공간이 어떤 목적으로 사용되는지를 나타내는 domain type과 read/write, cacheable/bufferable을 나타내는 필드를 가진다. `Last`는 이 구조체의 마지막임을 표시한다.

따라서, `init_maps`는 메모리 mapping의 초기번지를 가르키고, 이것을 이용해서 mapping의 시작주소의 메모리 descriptor를 초기화한다. Domain type은 DOMAIN_USER로 설정하고, read/write에 0을, caching은 가능하며, buffering은 하지 않도록 설정했다. 나머지는 각각의 메모리 information 구조체로부터 가져온 것을 map descriptor에 초기화 시키는 부분이다. 각 모모리 bank마다 차례로 설정하다. 중요한 것은 domain을 DOMAIN_KERNEL로 설정하 있으며, read/write에 각각 0과 1을 주었으며, cacheable/bufferable에 1을 주었다.

Domain은 1 Mbytes로 된 여러 section들로 만들어진 메모리 공간으로서 다음과 같은 뜻을 가지고 있다. 먼저 Domain 번호는 해당 메모리 영역이 사용자/커널/I/O 영역 중에서 어떤 영역으로 사용되는지를 나타내며, Domain 타입(type)은 해당 메모리 공간에 대한 접근 권한을 말한다. 정의는 다음과 같이 `~/inlcude/asm-arm/proc-armv/domain.h`에 있다.

```
/*
 * Domain numbers
 *
 * DOMAIN_IO      - domain 2 includes all IO only
 * DOMAIN_KERNEL - domain 1 includes all kernel memory only
 * DOMAIN_USER    - domain 0 includes all user memory only
 */

#define DOMAIN_USER          0 /* 사용자 영역 */
#define DOMAIN_KERNEL        1 /* 커널 영역 */
#define DOMAIN_TABLE         1 /* 커널에서 사용하는 페이지 테이블 영역 */
#define DOMAIN_IO             2 /* I/O 영역 */

/*
 * Domain types
 */
#define DOMAIN_NOACCESS      0 /* 접근을 허가하지 않는 영역 */
#define DOMAIN_CLIENT         1 /* Client 영역 : 페이지나 section에 있는 접근 허가 bits를 검사 */
#define DOMAIN_MANAGER        3 /* Manager 영역 : 접근 허가 bit를 검사하지 않음 */
```

코드 970. Domain 번호 및 타입에 대한 정의

따라서, domain이라는 것을 이용해서 어떤 영역이 어떤 역할로서 사용될지와 어떤 접근 허가 방법이 사용될지를 결정한다고 보면 된다.

```
#ifdef FLUSH_BASE
    p->physical = FLUSH_BASE_PHYS;
    p->virtual = FLUSH_BASE;
    p->length = PGDIR_SIZE;
    p->domain = DOMAIN_KERNEL;
    p->prot_read = 1;
    p->prot_write = 0;
    p->cacheable = 1;
    p->bufferable = 1;
    p++;
#endif
#endif
#ifndef FLUSH_BASE_MINICACHE
    p->physical = FLUSH_BASE_PHYS + PGDIR_SIZE;
    p->virtual = FLUSH_BASE_MINICACHE;
    p->length = PGDIR_SIZE;
    p->domain = DOMAIN_KERNEL;
    p->prot_read = 1;
    p->prot_write = 0;
    p->cacheable = 1;
    p->bufferable = 0;
    p++;
#endif
```

코드 971. memtable_init() 함수의 정의(계속)

이전 data cache의 flushing을 목적으로 하는 페이지 테이블을 만드는 과정이다. 각각은 FLUSH_BASE와 FLUSH_BASE_MINICACHE가 설정된 경우에만 수행된다. 이곳은 SA1100의 ZERO bank에 해당하는 영역이다. 이곳에서 사용하고 있는 상수값들은 아래와 같다.

```
/* ~/include/asm-arm/arch-sa1100/hardware.h에서 */
/* Flushing areas */
#define FLUSH_BASE_PHYS      0xe0000000      /* SA1100 zero bank */
#define FLUSH_BASE           0xf5000000
#define FLUSH_BASE_MINICACHE 0xf5800000
/* ~/include/asm-arm/pgtable.h에서 */
#define PMD_SHIFT            20
#define PGDIR_SHIFT          20
...
#define PMD_SIZE             (1UL << PMD_SHIFT)
#define PMD_MASK              (~(PMD_SIZE-1))
#define PGDIR_SIZE            (1UL << PGDIR_SHIFT)
#define PGDIR_MASK              (~(PGDIR_SIZE-1))
```

코드 972. SA1100의 zero bank 위치 정의 및 page와 관련된 상수의 정의

PGDIR_SIZE는 1을 20 bit 왼쪽으로 shift한 값이므로, 1Mbytes를 나타낸다고 보면 될 것이다. Flush 부분들의 domain은 DOMAIN_KERNEL로 설정하고 있으며, read/write는 1, 0으로, cacheable/bufferable에 대해서는 각각이 따로 1, 1과 1, 0을 주고 있다. 즉, 첫번째 부분에 대해서는 caching과 buffering이 가능하지만, 두번째 부분은 caching만이 가능하다. 또한 두 부분은 read만이 가능하며, write는 불가이다.

```
clear_mapping(0);
i = 0;
q = init_maps;
do {
```

```

        if (address < q->virtual || q == p) {
            clear_mapping(address);
            address += PGDIR_SIZE;
        } else {
            create_mapping(q);
            address = q->virtual + q->length;
            address = (address + PGDIR_SIZE - 1) & PGDIR_MASK;
            q++;
        }
    } while (address != 0);
    flush_cache_all();
}

```

코드 973. memtable_init() 함수의 정의(계속)

clear_mapping() 함수는 0을 넘겨받아서, 가상 주소 공간 0에서 시작하는 PGD(Page Global Directory)에 대한 mapping을 지우는 일을 한다. 나머지는 메모리 descriptor에 있지 않은 PGD들을 삭제하는 일을 한다. do{} while() loop를 돌면서 initial mapping을 일일이 살펴서, clear_mapping() 함수를 호출해서 페이지 descriptor에 있지 않는 PGD의 entry를 하나씩 지운다. clear_mapping() 함수는 ~/arch/arm/mm/mm-armv.c에 아래와 같은 정의를 가진다.

```

static inline void clear_mapping(unsigned long virt)
{
    pmd_clear(pmd_offset(pgd_offset_k(virt), virt));
}

```

코드 974. clear_mapping() 함수의 정의

clear_mapping() 함수는 단순히 pmd_clear()를 호출해서 해당 entry를 지우는 일을 한다. 다시 pmd_clear()는 각각의 CPU별로 cpu_XXX_set_pmd()와 같이 assembly로 정의된 함수이며, 하나의 페이지 딕렉토리 entry를 지우는 역할을 한다. 아래와 같은 코드를 가진다³⁹⁶.

```

ENTRY(cpu_sa110_set_pmd)
ENTRY(cpu_sa1100_set_pmd)
    str    r1, [r0]
    mcr   p15, 0, r0, c7, c10, 1           @ clean D entry
    mcr   p15, 0, r0, c7, c10, 4           @ drain WB
    mov    pc, lr

```

코드 975. cpu_sa110(or sa1100)_set_pmd() 함수의 정의

먼저, 넘겨받은 argument의 첫번째가 가르키는 메모리 번지에 두번째 argument를 저장한다(str). 이것은 해당 entry가 virtual address를 어디에서 사용하고 있는지를 나타내는 것이다. 다음으로 coprocessor 15(=p15)의 register 7(=c7)에 CRm으로 c10과 OPC_2로 1을 주어 D cache의 entry를 지운다. 여기서 넘겨주는 데이터(c0)는 가상 주소를 나타내는 ARM register이다. 다시 p15의 c7레지스터에 c10과 4를 주어서, write buffer를 비우도록(drain)한다. 이곳에서 사용하는 c0 레지스터는 무시된다. 즉, data cache에 저장된 내용이 있다면, 이것을 비우도록 만드는 것이다. 이것을 마치면, sub routine을 복귀하기 위해서 pc에 lr을 넣도록 한다.

create_mapping() 함수는 맵 디스크립터(map descriptor)에 의해서 명시된 mapping에 필요한 모든 페이지 딕렉토리의 엔트리(entry) 및 페이지 테이블들을 생성한다.

```
/*
```

³⁹⁶ 이것은 single CPU이 경우에 해당하는 코드이다. 나중에 관련된 부분을 볼 때, pmd_clear()와 같이 정의되는 함수들을 볼 수 있을 것이다.

```

* Create the page directory entries and any necessary
* page tables for the mapping specified by `md'.  We
* are able to cope here with varying sizes and address
* offsets, and we take full advantage of sections.
*/
static void __init create_mapping(struct map_desc *md)
{
    unsigned long virt, length;
    int prot_sect, prot_pte;
    long off;

    prot_pte = L_PTE_PRESENT | L_PTE_YOUNG | L_PTE_DIRTY |
        (md->prot_read ? L_PTE_USER : 0) |
        (md->prot_write ? L_PTE_WRITE : 0) |
        (md->cacheable ? L_PTE_CACHEABLE : 0) |
        (md->bufferable ? L_PTE_BUFFERABLE : 0);

    prot_sect = PMD_TYPE_SECT | PMD_DOMAIN(md->domain) |
        (md->prot_read ? PMD_SECT_AP_READ : 0) |
        (md->prot_write ? PMD_SECT_AP_WRITE : 0) |
        (md->cacheable ? PMD_SECT_CACHEABLE : 0) |
        (md->bufferable ? PMD_SECT_BUFFERABLE : 0);

```

코드 976. create_mapping() 함수의 정의

이미 앞에서 맵 디스크립터에 의해서 정의된 영역에 대해서 이전 해당 페이지 테이블의 엔트리(prot_pte)와 섹션에 각 섹션들에 해당하는 엔트리(prot_sect)를 만들려고 변수를 초기화 한다. Read/Write 및 cacheable/bufferable에 따라서, 생성되는 엔트리의 정보가 이곳에서 선택된다.

```

virt    = md->virtual;
off     = md->physical - virt;
length = md->length;

while ((virt & 0xfffffff || (virt + off) & 0xfffffff) && length >= PAGE_SIZE) {
    alloc_init_page(virt, virt + off, md->domain, prot_pte);
    virt    += PAGE_SIZE;
    length -= PAGE_SIZE;
}
while (length >= PGDIR_SIZE) {
    alloc_init_section(virt, virt + off, prot_sect);
    virt    += PGDIR_SIZE;
    length -= PGDIR_SIZE;
}
while (length >= PAGE_SIZE) {
    alloc_init_page(virt, virt + off, md->domain, prot_pte);
    virt    += PAGE_SIZE;
    length -= PAGE_SIZE;
}
}

```

코드 977. create_mapping() 함수의 정의(계속)

alloc_init_page() 함수는 새로운 페이지 테이블의 엔트리를 생성하는 함수이다. 필요하다면, 새로운 페이지 테이블을 위한 공간도 같이 할당한다. 그리고, alloc_init_section()은 섹션을 위한 페이지 글로벌 딕토리(Page Global Directory)를 할당하고 초기화 시킨다. 따라서, PAGE_SIZE(=4096 Bytes)단위로 1 Mbytes영역에 대해서 새로운 페이지 테이블에 대한 entry를 첫번째 while() loop에서 생성하고, 만약 이렇게 생성했다고 하더라도, 남은 영역에 대한 크기가(PGDIR_SIZE = 1 Mbytes)보다 큰 경우에는 1

Mbytes 단위로 다시 section에 대한 페이지 글로벌 딕렉토리를 두 번째 while() loop에서 생성해준다. 남은 1 Mbytes 이하의 공간에 대해서는 마지막 loop에서 페이지 테이블의 엔트리를 생성한다.

```
/*
 * Add a PAGE mapping between VIRT and PHYS in domain
 * DOMAIN with protection PROT. Note that due to the
 * way we map the PTEs, we must allocate two PTE_SIZE'd
 * blocks - one for the Linux pte table, and one for
 * the hardware pte table.
 */
static inline void
alloc_init_page(unsigned long virt, unsigned long phys, int domain, int prot)
{
    pmd_t *pmdp;
    pte_t *ptep;

    pmdp = pmd_offset(pgd_offset_k(virt), virt);
    if (pmd_none(*pmdp)) {
        pte_t *ptep = alloc_bootmem_low_pages(2 * PTRS_PER_PTE *
                                              sizeof(pte_t));
        ptep += PTRS_PER_PTE;
        set_pmd(pmdp, __mk_pmd(ptep, PMD_TYPE_TABLE | PMD_DOMAIN(domain)));
    }
    ptep = pte_offset(pmdp, virt);
    set_pte(ptep, mk_pte_phys(phys, __pgprot(prot)));
}
```

코드 978. alloc_init_page() 함수의 정의

alloc_init_page() 함수는 새로운 페이지를 할당해서 이것을 페이지 테이블 entry를 가지도록 만드는 일을 한다. 여기서 중요한 점은 2개의 PTE_SIZE(= PTRS_PER_PTE x sizeof(pte_t))를 할당해서 하나는 Linux를 위한 PTE 테이블로 사용하고, 나머지는 하드웨어의 PTE 테이블로 사용한다는 것이다. 먼저 가상 주소에 대한 페이지 미들 딕렉토리(PMD : Page Middle Directory)가 있는지를 확인한다. 없다면, 새로 PGD를 할당해서 Linux를 위한 PMD 영역을 설정하고(set_pmd()), 해당 엔트리(PTE : Page Table Entry)에 할당되었음을 표시한다(set_pte()).

```
/*
 * Create a SECTION PGD between VIRT and PHYS in domain
 * DOMAIN with protection PROT
 */
static inline void
alloc_init_section(unsigned long virt, unsigned long phys, int prot)
{
    pmd_t pmd;

    pmd_val(pmd) = phys | prot;
    set_pmd(pmd_offset(pgd_offset_k(virt), virt), pmd);
}
```

코드 979. alloc_init_section() 함수의 정의

alloc_init_section() 함수는 section에 해당하는 PMD를 생성하는 역할을 한다. 간단히 가상 주소에 해당하는 PMD를 찾아서, 해당 영역의 물리적인 주소와 접근 권한(protection or access permission)을 설정한다.

```
#define pgd_index(addr)          ((addr) >> PGDIR_SHIFT)
#define __pgd_offset(addr) pgd_index(addr)
```

```
#define pgd_offset(mm, addr) ((mm)->pgd+pgd_index(addr))
```

```
/* to find an entry in a kernel page-table-directory */
#define pgd_offset_k(addr) pgd_offset(&init_mm, addr)
```

코드 980. 페이지에 관련된 매크로 정의

`alloc_init_page()` 함수와 `alloc_init_section()` 함수는 모두 `pgd_offset_k()`라는 매크로를 사용하고 있다. 이 매크로는 `~/include/asm-arm/pgtable.h`에 정의된 것으로 다시 `pgd_offset(&init_mm, addr)`로 되어 있다. 여기서 관심을 가지고자 하는 부분이 바로 `init_mm`이다. 즉, `init_mm`에 대해서 어느 위치에 있는 PGD를 접근할 것인가를 결정하기 때문이다. `init_mm`은 커널의 초기화시에 최초로 생성할 task의 `mm_struct` 구조체를 가지는 변수이다. 아래와 같이 `~/arch/arm/kernel/init_task.c`에 정의되어 있다.

```
struct mm_struct init_mm = INIT_MM(init_mm);
```

`INIT_MM()` 매크로는 다시 다음과 같이 `~/include/linux/sched.h`에 정의되어 있으며, 단순히 `mm_struct` 구조체를 정해진 변수 값으로 초기화 하는 일을 수행한다.

```
#define INIT_MM(name) \
{ \
    mmap: &init_mmap, \
    mmap_avl: NULL, \
    mmap_cache: NULL, \
    pgd: swapper_pg_dir, \
    mm_users: ATOMIC_INIT(2), \
    mm_count: ATOMIC_INIT(1), \
    map_count: 1, \
    mmap_sem: __MUTEX_INITIALIZER(name.mmap_sem), \
    page_table_lock: SPIN_LOCK_UNLOCKED, \
    mmlist: LIST_HEAD_INIT(name.mmlist), \
}
```

코드 981. INIT_MM 매크로의 정의

즉, `vm_area_struct` 구조체를 가르키는 `mmap` 필드에는 `init_mmap` 변수의 주소가 들어가고, PGD에는 `swapper_pg_dir`를 사용한다. 나머지는 전체 시스템의 `mm_struct` 구조체를 유지하기 위해서 필요한 것을 초기화 시켜준다.

```
static struct vm_area_struct init_mmap = INIT_MMAP;
```

`init_mmap` 변수는 다시 `~/arch/arm/kernel/init_task..c`에 위와 같이 정의되어 있으며, init task의 `vm_area_struct` 구조체를 만들어주기 위한 것이다. `INIT_MMAP`은 `~/include/asm-arm/processor.h`에 아래와 같이 정의되어 있다.

```
#define INIT_MMAP { \
    vm_mm: &init_mm, \
    vm_page_prot: PAGE_SHARED, \
    vm_flags: VM_READ | VM_WRITE | VM_EXEC, \
    vm_avl_height: 1, \
}
```

코드 982. INIT_MMAP 매크로의 정의

mm_struct를 가르키기 위해서 init_mm의 주소를 가지며, 해당 가상 페이지의 공유 권한(protection)은 나중에 생성될 프로세스들을 위해서 PAGE_SHARED로 정의되며, READ/WRITE/EXEC 권한을 준다. 초기에 생성될 AVL tree의 높이는 10이므로 1로 초기화 시켜주었다.

다시 원래의 paging_init() 함수로 돌아가서, mdesc->map_io() 부분을 보도록 하자. 즉, I/O 메모리 영역에 대한 설정을 하는 부분이다. 따라서, 이것은 machine descriptor에서 초기화 시켜준 I/O 영역의 초기화를 위한 함수를 호출하는 일이 될 것이다. 따라서, 결국 앞에서 설정한 sa1100_map_io() 함수가 실행된다. 정의는 ~/arch/arm/mm/mm-sa1100.c에 아래와 같이 되어 있다.

```
void __init sa1100_map_io(void)
{
    struct map_desc *desc = NULL;

    iotable_init(standard_io_desc);

    if (machine_is_assabet())
        desc = assabet_io_desc;
    else if (machine_is_bitsy())
        desc = bitsy_io_desc;
    else if (machine_is_freebird())
        desc = freebird_io_desc;
    else if (machine_is_cerf())
        desc = cerf_io_desc;
    else if (machine_is_empeg())
        desc = empeg_io_desc;
    else if (machine_is_graphicsclient())
        desc = graphicsclient_io_desc;
    else if (machine_is_lart())
        desc = lart_io_desc;
    else if (machine_is_pleb())
        desc = pleb_io_desc;
    else if (machine_is.nanoengine())
        desc = nanoengine_io_desc;
    else if (machine_is_sherman())
        desc = sherman_io_desc;
    else if (machine_is_victor())
        desc = victor_io_desc;
    else if (machine_is_xp860())
        desc = xp860_io_desc;
    else if (machine_is_yopy())
        desc = yopy_io_desc;
    else if (machine_is_pangolin())
        desc = pangolin_io_desc;
    else if (machine_is_huw_webpanel())
        desc = huw_webpanel_io_desc;
    /* 다음과 같이 추가하면, 자신의 board에 맞는 설정을 해줄 것이다.*/
    else if( machine_is_wvp())
        desc = wvp_io_desc;      /* wvp_io_desc는 나중에 정의 */
    /* 여기까지다.*/
    if (desc)
        iotable_init(desc);
}
```

코드 983. sa1100_map_io() 함수의 정의

sa1100_map_io() 함수는 먼저 기본적인 I/O 영역에 대한 mapping을 한 후, 각각의 board에 맞는 I/O mapping을 다시 설정한다(iotable_init()). 각각의 board에 대한 모든 descriptor를 보기 보다는 간단히 표준

I/O 디스크립터만 보고 넘어가도록 하겠다. 또한 참고적으로 새로운 I/O descriptor를 추가 하려고 하는 경우도 참가했다.

```
static struct map_desc standard_io_desc[] __initdata = {
    /* virtual      physical      length      domain      r w c b */ \
    { 0xf6000000, 0x20000000, 0x01000000, DOMAIN_IO, 1, 1, 0, 0 }, /* PCMCIA0 IO */
    { 0x7f000000, 0x30000000, 0x01000000, DOMAIN_IO, 1, 1, 0, 0 }, /* PCMCIA1 IO */
    { 0x80000000, 0x80000000, 0x02000000, DOMAIN_IO, 0, 1, 0, 0 }, /* PCM */
    { 0xfa000000, 0x90000000, 0x02000000, DOMAIN_IO, 0, 1, 0, 0 }, /* SCM */
    { 0xfc000000, 0xa0000000, 0x02000000, DOMAIN_IO, 0, 1, 0, 0 }, /* MER */
    { 0xfe000000, 0xb0000000, 0x02000000, DOMAIN_IO, 0, 1, 0, 0 }, /* LCD + DMA */
    LAST_DESC
};

...
/* wvp_io_desc를 정의한다.*/
static struct map_desc wvp_io_desc[] __initdata = {
#ifndef CONFIG_SA1100_WVP
    { 0xe8000000, 0x22000000, 0x00100000, DOMAIN_IO, 1, 1, 0, 0 }, /* LED */
#endif
    LAST_DESC
};
```

코드 984. I/O Descriptor의 정의

I/O descriptor의 정의는 가상주소와 이것이 mapping될 물리적인 주소, I/O 영역의 길이 및 DOMAIN 번호, read/write, cacheable/bufferable이 순서대로 들어가게 된다. I/O를 위한 공간이므로 cacheable/bufferable은 항상 0으로 설정된다는 것을 볼 수 있을 것이다.

```
void __init iotable_init(struct map_desc *io_desc)
{
    int i;

    for (i = 0; io_desc[i].last == 0; i++)
        create_mapping(io_desc + i);
}
```

코드 985. iotable_init() 함수의 정의

iotable_init() 함수는 앞에서 본 I/O descriptor에 해당하는 mapping table을 만들기 위해서 각 descriptor의 엔트리마다 한번씩 create_mapping() 함수를 호출해 준다.

```
for (node = 0; node < numnodes; node++) {
    unsigned long zone_size[MAX_NR_ZONES];
    unsigned long zhole_size[MAX_NR_ZONES];
    struct bootmem_data *bdata;
    pg_data_t *pgdat;
    int i;

    for (i = 0; i < MAX_NR_ZONES; i++) {
        zone_size[i] = 0;
        zhole_size[i] = 0;
    }
    pgdat = NODE_DATA(node);
    bdata = pgdat->bdata;
    zone_size[0] = bdata->node_low_pfn - (bdata->node_boot_start >> PAGE_SHIFT);
#endif CONFIG_SA1111
    /* 256 pages in the DMA zone for SA1111 */
```

```

        zone_size[1] = zone_size[0] - 256;
        zone_size[0] = 256;
#endif
{
    /*
     * For each bank in this node, calculate the size of the
     * holes.  holes = node_size - sum(bank_sizes_in_node)
     */
    zhole_size[0] = zone_size[0];
    for (i = 0; i < mi->nr_banks; i++) {
        if (mi->bank[i].node != node)
            continue;
        zhole_size[0] -= mi->bank[i].size >> PAGE_SHIFT;
    }
#endif
    free_area_init_node(node, pgdat, 0, zone_size, bdata->node_boot_start, zhole_size);
}

```

코드 986. paging_init()함수의 정의(계속)

이전 각각의 node내에 있는 zone을 초기화 할 차례이다. for loop는 노드의 갯수만큼 반복되면서 수행된다. zone의 크기를 가지는 zone_size와 zone에 hole이 존재할 때 그 크기를 가지는 zhole_size를 MAX_NR_ZONES의 크기를 가지는 배열로 정의한다. bootmem_data를 가르키는 것으로 bdata 포인터 및 pg_data_t 구조체의 포인터인 pgdat를 정의한다.

일단 전체 zone을 나타내는 배열에 대해서 초기화 한다. 전부 0을 각각의 배열에 채워 넣는다. NODE_DATA() 매크로는 연속적인 페이지(contig_page_data)의 데이터의 포인터를 가져온다. 이를 pgdat 변수가 가지도록 만들고, 다시 이로부터 boot memory에 대한 정보를 가지는 bdata를 구해온다. zone_size[]의 첫번째 index에 해당하는 내용은 이곳에서 구한 페이지 프레임 번호에서 node의 boot memory start번지를 PAGE_SHIFT해서 얼마나 많은 페이지 프레임을 사용했는지를 빼준 값으로 넣는다. 만약 SA1111로 설정된 경우에는 이곳에서 구한 값에서 256을 빼서 zone_size[] 배열의 두번째 원소를 초기화 하며, 첫번째 원소는 256으로 놓는다.

설정된 값이 SA1111이 아니라면, 이젠 이 node에 속한 메모리의 bank로부터 hole(메모리의 주소 공간중에서 사용하지 않는 부분이다.)를 계산할 차례이다. 먼저 zone_size[0]을 zhole_size[0]로 둔다. 그리고나서, 각각의 메모리 뱅크가 가진 크기를 zhole_size[0]에서 계속적으로 빼준다. 남은 것이 zone hole의 크기가 될 것이다. 다른 노드에 대한 위의 연산을 반복하기 전에, 노드의 free area에 대한 초기화를 하기 위해서 free_area_init_node()함수를 호출한다. 이 함수는 유효한 페이지들에 대한 초기화를 수행할 것이다.

```

memzero(zero_page, PAGE_SIZE);
memzero(bad_page, PAGE_SIZE);
empty_zero_page = virt_to_page(zero_page);
empty_bad_page = virt_to_page(bad_page);
empty_bad_pte_table = ((pte_t *)bad_table) + TABLE_OFFSET;
}

```

코드 987. paging_init()함수의 정의(계속)

zero_page와 bad_page에 대해서 PAGE_SIZE만큼을 0으로 채운다(memzero()). empty_zero_page는 zero_page의 페이지 주소를 가지며, empty_bad_page는 bad_page의 페이지 주소, empty_bad_pte_table은 bad_table(Bad 페이지 테이블 entry)에 TABLE_OFFSET을 더한 값을 가진다.

request_standard_resources()함수는 ~/arch/arm/kernel/setup.c에 정의되어 있으며, 함수의 이름에서도 알듯이 표준 자원들에 대한 요청을 하는 함수이다. 아래와 같다.

```

static void __init request_standard_resources(struct meminfo *mi, struct machine_desc *mdesc)
{

```

```

struct resource *res;
int i;

kernel_code.start = __virt_to_bus(init_mm.start_code);
kernel_code.end = __virt_to_bus(init_mm.end_code - 1);
kernel_data.start = __virt_to_bus(init_mm.end_code);
kernel_data.end = __virt_to_bus(init_mm.brk - 1);
for (i = 0; i < mi->nr_banks; i++) {
    unsigned long virt_start, virt_end;

    if (mi->bank[i].size == 0)
        continue;
    virt_start = __phys_to_virt(mi->bank[i].start);
    virt_end = virt_start + mi->bank[i].size - 1;
    res = alloc_bootmem_low(sizeof(*res));
    res->name = "System RAM";
    res->start = __virt_to_bus(virt_start);
    res->end = __virt_to_bus(virt_end);
    res->flags = IORESOURCE_MEM | IORESOURCE_BUSY;
    request_resource(&iomem_resource, res);
    if (kernel_code.start >= res->start &&
        kernel_code.end <= res->end)
        request_resource(res, &kernel_code);
    if (kernel_data.start >= res->start &&
        kernel_data.end <= res->end)
        request_resource(res, &kernel_data);
}

```

코드 988. request_standard_resources()함수의 정의

커널의 코드와 데이터에 대한 영역을 init_mm³⁹⁷(initial memory management)구조체에서 얻어온다. init_mm은 ~/arch/arm/kernel/init_task.c에 정의된 것으로 초기의 mm_struct구조체를 가지며, 이곳에서 사용된 __virt_to_bus() 매크로는 가상주소를 물리적인 주소로 변환하는데 사용한다. Memory info구조체에서 각 메모리 bank에 대해서 for loop를 돌면서 아래와 같은 일을 한다.

만약 bank의 크기가 0이라면 다음 bank로 넘어가고, 그렇지 않다면 가상주소의 시작과 끝을 구해서 각각 virt_start와 virt_end로 둔다. resource 구조체 만큼 메모리를 할당받아서(alloc_bootmem_low()), 이를 res로 나타내고, 이 자원의 이름으로는 System RAM을 시작과 끝주는 앞에서 구한 virt_start와 virt_end의 물리적인 주소값을 준다. flag은 자원의 현재 상태를 나타내는 값으로 IORESOURCE_MEM와 IORESOURCE_BUSY를 OR시킨 값으로 설정해서 MEM자원이며, 사용됨을 나타낸다.

request_resource()함수는 해당 메모리 공간에 대한 할당을 하는 함수이다. 전체 I/O를 위한 메모리 자원에 대한 포인터(&iomem_resource)와 앞에서 할당받은 자원에 대한 자료구조의 포인터를 넘겨주어서 할당하도록 한다. 만약 커널 코드와 데이터가 메모리 자원의 공간상에 위치한다면, 이를 위한 자원도 할당한다(request_resource()). 참고로 코드에서 보인 kernel_code와 kernel_data는 아래와 같이 정의된다.

```

/* ~/include/linux/ioport.h 에서 */
struct resource {
    const char *name;                      /* 자원을 할당받는 객체에 대한 이름 */
    unsigned long start, end;                /* 자원의 시작과 마지막 주소 */
    unsigned long flags;                    /* 자원에 대한 설정 사항 */
    struct resource *parent, *sibling, *child; /* 자원에 대한 Tree 연결 구조 */
};

/* ~/arch/arm/kernel/setup.c에서 */
static struct resource mem_res[] = {

```

³⁹⁷ init_mm은 setup_arch에서 이미 _text, _etext, _edata, _end등으로 이미 설정되어 있다.

```

{ "Video RAM", 0, 0, IORESOURCE_MEM },
{ "Kernel code", 0, 0, IORESOURCE_MEM },
{ "Kernel data", 0, 0, IORESOURCE_MEM }
};

#define video_ram    mem_res[0]
#define kernel_code mem_res[1]
#define kernel_data mem_res[2]

```

코드 989. 표준 메모리 자원의 정의

커널의 코드와 데이터를 나타내는 resource 구조체는 초기에 둘다 시작과 끝의 값으로 0으로 가지며, IORESOURCE_MEM³⁹⁸ flag으로 설정한다. 앞에서 request_standard_resources() 함수의 초기에, 이들 각각에 대한 값이 init_mm에서 읽혀들어 오게 된다.

```

if (mdesc->video_start) {
    video_ram.start = mdesc->video_start;
    video_ram.end   = mdesc->video_end;
    request_resource(&iomem_resource, &video_ram);
}
if (mdesc->reserve_lp0)
    request_resource(&ioport_resource, &lp0);
if (mdesc->reserve_lp1)
    request_resource(&ioport_resource, &lp1);
if (mdesc->reserve_lp2)
    request_resource(&ioport_resource, &lp2);
}

```

코드 990. request_standard_resouces()함수의 정의(계속)

아직까지 할당하지 않고 남은 것은 video RAM에 대한 것과 lp(Line Printer: Parallel Port)에 대한 것이다. Machine descriptor 구조체에서 해당하는 field를 찾아서, 각각 할당 받도록 한다(request_resource()).

```

/*
 * Set up various architecture-specific pointers
 */
init_arch_irq = mdesc->init_irq;
#endif CONFIG_VT
#ifdef CONFIG_VGA_CONSOLE
    conswitchp = &vga_con;
#elif defined(CONFIG_DUMMY_CONSOLE)
    conswitchp = &dummy_con;
#endif
#endif
}
```

코드 991. setup_arch()함수의 정의(계속)

이것으로 request_standard_resources() 함수의 설명이 끝났다. 다시 setup_arch()의 이야기로 돌아가서, 인터럽트에 대해서 초기화를 담당할 함수를 설정하는 부분이다. 즉, machine descriptor에서 init_irq에 해당한다. 따라서, 우리의 경우에는 machine descriptor에 genarch_init_irq()로 설정을 했기 때문에, 이것이 나중에 인터럽트를 초기화 하는 곳에서 불려질 것이다. 만약 커널이 CONFIG_VT(virtual terminal)을 사용하도록 되어있고, CONFIG_VGA_CONSOLE로 설정되어 있다면, conswitchp에 vga_con의 주소를 넣도록 한다. 만약 CONFIG_DUMMY_CONSOLE로 설정된 경우에는 conswitchp에 dummy_con의 주소를 넣도록 한다.

³⁹⁸ 이 flag 값 이외에도 DMA나 IRQ등과 같은 자원들을 나타내는 많은 flag값이 ~/include/linux/ioport.h에 정해져 있음을 알 수 있다.

우린 지금 `start_kernel()`함수를 분석하는 중이다. `start_kernel()`함수에서 `setup_arch()`이후에 오는 것은 `printk()`를 호출해서 `saved_command_line`을 출력하는 부분이 있다. 그 다음에 바로 `parse_options()`함수가 온다. `parse_option()`함수는 `command_line`상에서 주어진 input에 대해서 해석(parsing)하는 일을 한다. 이 함수는 커널에 대한 옵션을 받아들여서 이를 환경 변수(environment variable)이나 argument로 저장하는 하는 역할을 하고 있다.

17.7.3.3. `trap_init()` 함수의 분석

`trap_init()`함수는 커널이 사용하게 될 예외상황(exception)을 처리하는 부분에 대해서 초기화를 하는 역할을 한다. Intel StrongARM SA1100 보드에서 사용하는 예외 상황으로는 다음과 같은 것이 있다.

- Reset
- Undefined Instruction
- Software Interrupt
- Prefetch Abort
- Data Abort
- Reserved
- IRQ(Normal Interrupt Request)
- FIQ(Fast Interrupt Request)

일단 예외 상황이 발생하면, 기본적으로 PC(Program Counter or Instruction Pointer)를 0x00000000에서 시작하는 적절한 exception vector로 바꾸게 된다. 이렇게 바뀐 exception vector에 따라서, 처리는 각각의 exception handler가 맡는다. `trap_init()`함수는 이러한 vector를 Linux에서 사용하는 trap handler로 재설치하는 일을 한다. 정의는 `~/arch/arm/kernel/trap.c`에 아래와 같이 되어 있다.

```
void __init trap_init(void)
{
    extern void __trap_init(void);

    __trap_init();
#endif CONFIG_CPU_32
    modify_domain(DOMAIN_USER, DOMAIN_CLIENT);
#endif
}
```

코드 992. `trap_init()`함수의 정의

`trap_init()`함수는 단지 `__trap_init()`를 호출하는 일만 담당하고 있다. 만약 `CONFIG_CPU_32`가 설정된 경우에는 `modify_domain()`함수를 호출해서 메모리에 대한 access 권한을 변경한다. 따라서, 실제적인 예외 상황의 처리에 대한 초기화는 `__trap_init()`에서 수행된다. `__trap_init()`는 `~/arch/arm/kernel/entry-armv.S`에 아래와 같이 정의되어 있다.

```
ENTRY(__trap_init)
    stmdfd    sp!, {r4 - r6, lr}
    adr       r1, .LCvectors
    mov       r0, #0
    ldmia    r1, {r1, r2, r3, r4, r5, r6, ip, lr}
    stmia    r0, {r1, r2, r3, r4, r5, r6, ip, lr}
    add       r2, r0, #0x200
    adr       r0, __stubs_start
    adr       r1, __stubs_end
1:   ldr       r3, [r0], #4
    str       r3, [r2], #4
    cmp       r0, r1
    blt      1b
    LOADREGS(fd, sp!, {r4 - r6, pc})
```

코드 993. __trap_init()함수의 정의

stmfd는 sp(Stack Pointer)가 가르키는 곳으로 r4에서 r6까지의 register와 lr(=r14)를 저장하라는 말이다. 즉, 현재의 register들을 저장하고 이하에서 그것을 다시 사용하게 된다. 물론 나중에 이 routine을 빠져나가게 될 때, 원상태로 복구하게 될 것이다(LOADREGS()). 이때 lr은 pc(=r15)에 들어가게 된다.

이전 r1에 .LCvectors의 주소를 가져온다. r0에는 0을 넣고, r1이 가르키는 곳에서 r1, r2, r3, r4, r6, ip, lr을 load해서 다시 r0가 가르키는 곳에 저장한다. r2는 r0에 0x200을 더한 값으로 준다. r0에는 다시 __stubs_start의 주소를 가져오고, r1에는 __stubs_end의 주소를 가져온다. 이전 4bytes씩 r0에서 읽어서, r2로 복사한다. r0와 r1을 비교해서 작다면, 계속 4bytes씩 옮기는 일을 수행한다. 즉, 0x00000000의 첫부분에는 .LCvectors의 내용이 들어가게 되며, 다시 0x00000200에는 __stubs_start에서 __stubs_end까지의 내용이 들어가게 된다. 연산을 마치면 원래의 레지스터 값을 복원하고 복귀 한다(LOADREGS()).

.LCvectors:	.equ	__real_stubs_start, .LCvectors + 0x200	
	swi	SYS_ERROR0	/* Reset */
	b	__real_stubs_start + (vector_undefinstr - __stubs_start)	/* Undefined Instruction */
	ldr	pc, __real_stubs_start + (.LCvswi - __stubs_start)	/* Software Interrupt */
	b	__real_stubs_start + (vector_prefetch - __stubs_start)	/* Prefetch Abort */
	b	__real_stubs_start + (vector_data - __stubs_start)	/* Data Abort */
	b	__real_stubs_start + (vector_addrexcptn - __stubs_start)	/* Address exception */
	b	__real_stubs_start + (vector_IRQ - __stubs_start)	/* Normal Interrupt Request */
	b	__real_stubs_start + (vector_FIQ - __stubs_start)	/* Fast Interrupt Request */

코드 994. .LCvectors의 정의

.LCvectors에는 각각의 exception handler의 주소가 들어있다. __real_stubs_start는 .LCvectors에 0x200을 더한 값을 가진다. .LCvectors가 가지는 내용은 프로그램의 주석을 참조하면 될 것이다. 각각은 해당하는 exception handler로의 branch instruction을 상대적인 주소로 수행하고 있음을 알 수 있을 것이다. 즉, 실제 stub의 시작주소(start address)에 상대적으로 얼마정도에 위치해 있는지를 나타내는 offset을 더해서 정해진다. 시작주소(__real_stubs_start) 역시도 재배치가 가능하도록 되어 있다.

__stubs_start와 __stubs_end사이에 들어있는 부분들에 대해서 좀더 보도록 하자. __stubs_start와 __stubs_end사이에 들어있는 코드들은 각각이 해당하는 예외 상황을 처리하는 핸들러를 불러주는 코드들이이다. 이곳에 들어있는 코드 중에서 인터럽트의 초기화와 관련된 처리를 하는 vector_IRQ를 보기로 하자.

.LCsirq:	.word	__temp_irq	
...			
vector_IRQ:	@		
	@ save mode specific registers		
	@		
ldr	r13, .LCsirq		
sub	lr, lr, #4		
str	lr, [r13]	@ save lr_IRQ	
mrs	lr, spsr		
str	lr, [r13, #4]	@ save spsr_IRQ	
@			
@ now branch to the relevant MODE handling routine			
@			
mov	r13, #I_BIT MODE_SVC		
msr	spsr_c, r13	@ switch to SVC_32 mode	
and	lr, lr, #15		
ldr	lr, [pc, lr, lsl #2]		
movs	pc, lr	@ Changes mode and branches	
.LCtab_irq:	.word	__irq_usr	@ 0 (USR_26 / USR_32)

.word	__irq_invalid	@ 1 (FIQ_26 / FIQ_32)
.word	__irq_invalid	@ 2 (IRQ_26 / IRQ_32)
.word	__irq_svc	@ 3 (SVC_26 / SVC_32)
.word	__irq_invalid	@ 4
.word	__irq_invalid	@ 5
.word	__irq_invalid	@ 6
.word	__irq_invalid	@ 7
.word	__irq_invalid	@ 8
.word	__irq_invalid	@ 9
.word	__irq_invalid	@ a
.word	__irq_invalid	@ b
.word	__irq_invalid	@ c
.word	__irq_invalid	@ d
.word	__irq_invalid	@ e
.word	__irq_invalid	@ f
...		
__temp_irq:	.word 0	@ saved lr_irq
	.word 0	@ saved spsr_irq
	.word -1	@ old_r0

코드 995. vector_IRQ의 정의

vector_IRQ는 인터럽트를 dispatch하는 코드이다. 인터럽트 모드로 들어가면, spsr(Saved Program Status Register)에 사용자 모드의 cpsr(Current Program Status Register)을 저장하고, lr(Link Register)에는 사용자 모드의 PC(Program Counter)를 저장한다.

먼저 r13에 .LCsirq를 가져오고, lr에는 원래의 lr에서 4를 빼서 넣는다. 이것은 나중에 돌아갈 주소를 지정하는 것으로, ARM에서는 3 stage pipeline을 사용하기에 미리 현재 decoding되고 있는 instruction보다 PC가 이미 2 단계를 더 진행해 있다. 따라서, 바로 다음번에 실행할 번지를 구하기 위해서는 이곳에서 4 bytes³⁹⁹ 만큼을 빼주어야 한다. r13이 가리키는 .LCsirq의 __temp_irq에 이렇게 계산한 lr를 보관해 둔다. 나중에 이 값은 복구 되어서 사용할 것이다. spsr의 읽어서 이를 lr에 넣는다(mrs). 읽어온 spsr은 다시 r13이 가르키는 곳에 4 bytes만큼을 증가시킨후에 저장한다.

r13에 I bit부분과 MODE_SVC bit부분을 설정해서 넣고, 이를 spsr_c 레지스터에 넣는다. 이것은 인터럽트를 disable시키고, 시스템 service모드⁴⁰⁰로 진행하도록 만든다. 다시 이전에 읽어온 spsr값을 가지고 있는 lr에 15를 AND 시켜서 하위 4 bit만 남도록 만든다. 이 값을 index로 해서 해당 모드로 branch하는 주소를 찾게 되는데, spsr의 모드를 나타내는 하위 5 bit중에서 4 bit만을 사용한다. 이 index값에 4를 곱해서(lsl #2) 해당하는 곳의 핸들러의 주소를 찾는다. 이젠 pc에 찾은 핸들러의 주소를 넣어주면 될 것이다.

나머지 vector_XXX에 관련된 것들도 위와 비슷한 과정을 전부 거칠도록 되어 있다. 즉, 위에서 본 각각의 예외 상황에 대한 처리를 담당하는 핸들러로 dispatching하는 일을 한다.

```
asmlinkage void __init start_kernel(void)
{
    char * command_line;
    unsigned long mempages;
    extern char saved_command_line[];

/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
    lock_kernel();
    printk(linux_banner);
    setup_arch(&command_line);
    printk("Kernel command line: %s\n", saved_command_line);
```

³⁹⁹ 하나의 Instruction이 32bit를 차지하므로 4 bytes이다.

⁴⁰⁰ Supervisor mode.

```

parse_options(command_line);
trap_init();
init_IRQ();
...

```

코드 996. start_kernel()함수의 정의 계속)

앞에서 했던 이야기로 다시 돌아가기 위해서, 잠시 정리하도록 하자. 현재 우리는 start_kernel() 함수의 분석에 대해서 이야기를 하고 있으며, 커널에 lock을 설정해서 다른 부분이 커널로 진입하지 못하도록 만든 후(lock_kernel()), architecture에 의존적인 설정(setup)을 하는 setup_arch()함수를 호출했으며, command line에서 주어진 option들에 대한 처리를 위해서 parse_options() 함수도 호출했다. 이와 같은 일을 수행하고 나서, 예외 상황에 대한 처리를 위한 진입점(entry point)를 설정해 주었다. 이젠 실제적인 인터럽트를 처리하는 인터럽트 핸들러들에 대한 초기화를 하는 init_IRQ()를 보도록 하겠다.

17.7.3.4. init_IRQ() 함수의 분석

```

void __init init_IRQ(void)
{
    extern void init_dma(void);
    int irq;

    for (irq = 0; irq < NR_IRQS; irq++) {
        irq_desc[irq].probe_ok = 0;
        irq_desc[irq].valid = 0;
        irq_desc[irq].noautoenable = 0;
        irq_desc[irq].mask_ack = dummy_mask_unmask_irq;
        irq_desc[irq].mask = dummy_mask_unmask_irq;
        irq_desc[irq].unmask = dummy_mask_unmask_irq;
    }
    init_arch_irq();
    init_dma();
}

```

코드 997. init_IRQ()함수의 정의

init_IRQ함수는 인터럽트에 대한 기술(descriptor)를 나타내는 데이터 구조체인 irq_desc[]를 초기화 시키고, architecture에 의존적인 인터럽트의 초기화를 위해서 init_arch_irq()함수를 호출하며, DMA를 사용하는 디바이스 들에 대한 초기화를 위해서 init_dma()함수를 부른다. irq_desc[]데이터 구조체는 irqdesc 구조체 타입으로 정의되며, 아래와 같이 ~/include/asm-arm/mach/irq.h에 정의되어 있다.

```

struct irqdesc {
    unsigned int nomask : 1; /* IRQ는 IRQ가 진행중일 때는 mask를 하지 않는다.*/
    unsigned int enabled : 1; /* IRQ가 현재 enable되었다.*/
    unsigned int triggered : 1; /* IRQ가 triggering되었다.*/
    unsigned int probing : 1; /* IRQ가 probing하는데 사용되고 있다.*/
    unsigned int probe_ok : 1; /* IRQ가 probing하는데 사용될 수 있다.*/
    unsigned int valid : 1; /* IRQ가 유효하다.*/
    unsigned int noautoenable : 1; /* 자동적으로 IRQ를 enable시키지 않는다.*/
    unsigned int unused : 25; /* 사용하지 않는 필드*/
    void (*mask_ack)(unsigned int irq); /* IRQ를 masking하고 ACK를 보내는 함수 포인터*/
    void (*mask)(unsigned int irq); /* IRQ를 masking하는 함수 포인터*/
    void (*unmask)(unsigned int irq); /* IRQ를 unmasking하는 함수 포인터*/
    struct irqaction *action; /* IRQ와 관련된 처리를 담당하는 함수 포인터 */
}
/* IRQ lock detection

```

```

/*
unsigned int      lck_cnt;           /* IRQ lock counter */
unsigned int      lck_pc;            /* Program Counter의 값 */
unsigned int      lck_jif;           /* Jiffies 값 */
};

```

코드 998. irqdesc구조체의 정의

irqdesc구조체는 전체 시스템에서 사용할 인터럽트에 대한 기술자 역할을 하는 구조체를 정의한다. 이곳에서 중요한 부분으로 볼 수 있는 곳은 irqaction구조체로 정의된 action필드이다. 이 필드는 아래와 같은 정의를 다시 가지고 있다.

```

struct irqaction {
    void (*handler)(int, void *, struct pt_regs *); /* IRQ handler */
    unsigned long flags;                            /* IRQ flags */
    unsigned long mask;                            /* IRQ mask값 */
    const char *name;                             /* IRQ를 사용하는 디바이스의 이름 */
    void *dev_id;                               /* IRQ를 사용하는 디바이스의 ID */
    struct irqaction *next;                        /* 다음에 연결된 irqaction구조체에 대한 포인터 */
};

```

코드 999. irqaction구조체의 정의

irqaction구조체는 IRQ를 핸들링 하기 위한 핸들러를 가지고 있다. 이것은 나중에 자주 보겠지만, 디바이스 드라이버를 작성하는 프로그램마가 인터럽트 핸들러를 설치할 때 정해지는 부분이다. 따라서, 이곳은 디바이스가 커널에 비동기적인 event를 발생시킬 경우 그 처리를 담당하는 부분이 된다. 디바이스 드라이버는 request_irq()함수와 같은 것으로 인터럽트 핸들러를 설치하고, 인터럽트 번호를 할당받는데, 내부적으로는 커널은 irqaction구조체를 하나 할당해서, irq_desc[]구조체에 해당하는 인터럽트 번호의 action필드에 attach해 준다.

앞의 이야기로 돌아가서, init_arch_irq()함수는 앞에서 machine descriptor를 볼 때, genarch_init_irq()함수를 가진다는 것을 보았다. 따라서, genarch_init_irq() 함수를 보도록 하자. 아래와 같이 ~/arch/arm/kernel/irq-arch.c에 나와 있다.

```

void __init genarch_init_irq(void)
{
    irq_init_irq();
}

```

코드 1000. genarch_init_irq()함수의 정의

genarch_init_irq()함수는 단순히 irq_init_irq()함수를 불러주는 wrapper의 구실 밖에는 하지 않는다. irq_init_irq()함수는 ~/include/asm-arm/arch-sa1100/irq.h에 아래와 같이 inline함수로 정의되어 있다.

```

static __inline__ void irq_init_irq(void)
{
    int irq;

    /* disable all IRQs */
    ICMR = 0;
    /* all IRQs are IRQ, not FIQ */
    ICLR = 0;
    /* clear all GPIO edge detects */
    GFER = 0;
    GRER = 0;
    GEDR = -1;
}

```

```

/*
 * Whatever the doc says, this has to be set for the wait-on-irq
 * instruction to work... on a SA1100 rev 9 at least.
 */
ICCR = 1;

```

코드 1001. irq_init_irq()함수의 정의

먼저, SA1100의 인터럽트에 대해서 알아야지만 이곳에서 하는 일을 이해할 수 있을 것이다. 간단히 ICMR은 Interrupt Controller Mask Register로서 이 레지스터에 인터럽트 번호에 해당하는 bit이 설정된 경우에만 인터럽트가 발생된다. 따라서, ICMR에 0을 넣게 되면, 모든 인터럽트의 발생이 금지된다. 다시 ICLR은 Interrupt Controller Level Register로서 FIQ 인터럽트를 발생시킬지, 아니면 IRQ를 발생시킬지를 결정하는 레지스터로서 이곳에 0을 넣는다면, 모든 인터럽트를 IRQ로 처리하겠다는 말이다. 따라서, FIQ를 사용하지 않는다는 말이된다.

이와 같은 설정을 마치면, 이젠 응용 프로그램에 맞도록 GPIO(General Purpose I/O) pin들에 대한 설정을 해줄 차례이다. 총 28개의 GPIO pin은 특수한 input/output signal을 전송하기 위해서 사용하게 되는데, GFER(GPIO Falling Edge Register)와 GRER(GPIO Rising Edge Register)를 0으로 두어서 초기화를 한다. GEDR(GPIO Edge Detect Register)는 GPIO pin의 edge detect 상태를 보관하는 레지스터로서 -1값을 넣는다는 말은 레지스터의 각 bit을 전부 1로 설정한다는 말이된다. 이것으로 GPIO에 대한 설정은 끝났다.

ICCR(Interrupt Controller Control Register)는 하나의 control bit을 가지는 레지스터로 DIM(Disable Idle Mask) bit을 가진다. 0으로 설정되어 있다면, 모든 enable된 인터럽트는 SA1100을 idle mode에서 깨어나도록 만들 것이다. 1로 설정된다면, 단지 enable되고, mask되지 않은 인터럽트만이(즉, ICMR에서 설정된 인터럽트 만이) SA1100을 idle mode에서 깨워줄 것이다. 이 bit은 reset이 일어난 후에는 지워져 있게 된다. 따라서, 우린 이 레지스터에 1을 넣어주었다.

```

for (irq = 0; irq <= 10; irq++) {
    irq_desc[irq].valid = 1;
    irq_desc[irq].probe_ok = 1;
    irq_desc[irq].mask_ack = sa1100_mask_and_ack_GPIO0_10_irq;
    irq_desc[irq].mask = sa1100_mask_GPIO0_10_irq;
    irq_desc[irq].unmask = sa1100_unmask_GPIO0_10_irq;
}
for (irq = 11; irq <= 31; irq++) {
    irq_desc[irq].valid = 1;
    irq_desc[irq].probe_ok = 0;
    irq_desc[irq].mask_ack = sa1100_mask_irq;
    irq_desc[irq].mask = sa1100_mask_irq;
    irq_desc[irq].unmask = sa1100_unmask_irq;
}
for (irq = 32; irq <= 48; irq++) {
    irq_desc[irq].valid = 1;
    irq_desc[irq].probe_ok = 1;
    irq_desc[irq].mask_ack = sa1100_mask_and_ack_GPIO11_27_irq;
    irq_desc[irq].mask = sa1100_mask_GPIO11_27_irq;
    irq_desc[irq].unmask = sa1100_unmask_GPIO11_27_irq;
}
setup_arm_irq(IRQ_GPIO11_27, &GPIO11_27_irq );

```

코드 1002. irq_init_irq()함수의 정의(계속)

이전 각각의 인터럽트를 나타내는 irq_desc[] 배열의 각 필드들을 초기화해 주도록 한다. 0에서 10번까지의 irq_desc[] 배열의 원소는 probe_ok 필드를 1로 설정하고 있으며, mask_ack, mask, unmask에 해당하는 함수로 sa1100_mask_and_ack_GPIO0_10_irq 함수와, sa1100_mask_GPIO0_10_irq 함수 및 sa1100_unmask_GPIO0_10_irq 함수를 각각 사용하고 있다. 11에서 31까지의 인터럽트 번호에는 probe_ok를 0으로 설정해 주었으며, 앞에서와 같은 함수의 포인터를 나타내는 부분에 sa1100_mask_irq 함수와,

sa1100_unmask_irq 함수를 주었다. 32에서 48까지의 인터럽트 번호에는 다시 probe_ok 필드에 1을 주었으며, sa1100_mask_and_ack_GPIO11_27_irq 함수와 sa1100_mask_GPIO11_27_irq 함수 및 sa1100_unmask_GPIO11_27_irq 함수를 사용했다⁴⁰¹. 여기서 유의할 점은 인터럽트 번호 0에서 10까지는 GPIO 0에서 10번까지의 pin이 해당하고, 나머지 32에서 48까지는 GPIO 11에서 27까지의 pin이 사용하고 있다는 점과, GPIO의 경우에는 항상 ACK를 해주는 함수가 따로이 존재해서 GPIO를 사용한 interrupt의 경우에는 반드시 ACK를 해주어야 한다는 점이다. setup_arm_irq()함수는 아래의 코드에서도 나오고 있으니 조금 이따가 보기로 하자. 넘겨주는 값으로는 IRQ_GPIO11_27과 GPIO11_27_irq가 있으며, 각각은 다시 아래와 같은 정의를 가지고 있다.

```
/* ~/include/asm-arm/arch-sa1100/irqs.h에서 */
#define SA1100_IRQ(x)          (0 + (x))
...
#define IRQ_GPIO11_27           SA1100_IRQ(11)
/* ~/include/asm-arm/arch-sa1100/irq.h에서 */
static struct irqaction GPIO11_27_irq = {
    name:                 "GPIO 11-27",
    handler:              sa1100_GPIO11_27_demux,
    flags:                SA_INTERRUPT
};
```

코드 1003. IRQ_GPIO11_27과 GPIO11_27_irq의 정의

IRQ_GPIO11_27은 SA1100_IRQ(11)을 사용해서 정의되며, 다시 SA1100_IRQ(x)는 0에, x를 더한 값으로 정의한다. 따라서, IRQ_GPIO11_27은 11이라는 값을 가지게 된다. GPIO11_27_irq는 해당하는 인터럽트에 대해서 취해줄 action을 나타내는 필드로 이름을 “GPIO 11-27”을 가지며, 인터럽트 핸들러로는 sa1100_GPIO11_27_demux() 함수를 가진다. 인터럽트의 flag값으로는 SA_INTERRUPT로 설정된다. 여기서 중요한 점은 GPIO 11에서 27까지의 인터럽트는 인터럽트 번호 11번을 사용하게 되며, 이것을 처리하는 것은 sa1100_GPIO11_27_demux() 함수가 맡을 것이라는 것이다. 이 함수는 다시 ~/include/asm-arm/arch-sa1100/irq.h에 아래와 같이 정의되어 있다.

```
static int GPIO_11_27_enabled;           /* enabled i.e. unmasked GPIO IRQs */
static int GPIO_11_27_spurious;         /* GPIOs that triggered when masked */

static void sa1100_GPIO11_27_demux(int irq, void *dev_id,
                                    struct pt_regs *regs)
{
    int i, spurious;

    while( (irq = (GEDR & 0xfffff800)) ){
        spurious = irq & ~GPIO_11_27_enabled;
        if (spurious) {
            GEDR = spurious;
            GRER &= ~(spurious & GPIO_11_27_spurious);
            GFER &= ~(spurious & GPIO_11_27_spurious);
            GPIO_11_27_spurious |= spurious;
            irq ^= spurious;
            if (!irq) continue;
        }
        for (i = 11; i <= 27; ++i) {
            if (irq & (1<<i)) {
                do_IRQ( IRQ_GPIO_11_27(i), regs );
            }
        }
    }
}
```

⁴⁰¹이 함수들은 나중에 다시 설명하도록 하겠다.

{}

코드 1004. sa1100_GPIO11_27_demux()함수의 정의

sa1100_GPIO11_27_demux() 함수가 넘겨받는 파라미터 값은 디바이스의 id와 인터럽트가 발생했을 때의 레지스터 값이다. while loop를 돌면서 GEDR(GPIO Edge Detect Register)에 0xFFFFF800을 AND시켜서 값이 있을 동안 계속 수행한다. 즉, 감지된(detected) 인터럽트에 대해서 계속적으로 서비스를 하겠다는 말이다. 0에서 10번 bit에 대해서는 처리하지 않는다는 것을 0xFFFFF800과 AND 시킨다는 것을 보면 알 수 있을 것이다.

GPIO_11_27_enabled를 NOT한 값과 앞에서 AND 시킨 연산의 결과를 가지는 irq를 다시 AND시켜서 enable되지 않은 인터럽트를 걸러낸다. 이렇게 계산된 값을 spurious 변수에 넣고, GEDR에 쓴다. 이것은 GEDR을 지우는 일을 한다. 또한 GRER과 GFER에는, masking된 때에 triggering된 GPIO를 가지는 GPIO_11_27_spurious와 spurious 변수를 AND 시켜 나온 값을, 다시 NOT한 후에 넣는다. GRER과 GFER에 이 값을 넣는다는 말은 더 이상 disable된 interrupt에 대해서 edge detection이 일어나지 않도록 만든다. 다시 GPIO_11_27_spurious에는 원래의 값에 spurious를 OR시켜서 넣는다. irq에는 spurious와 EXCLUSIVE OR시킨 값을 넣어주도록 하고, 만약 irq값이 0이라면 계속 loop를 돈다.

이전 irq에 정말 service를 받아야 하는 인터럽트들을 담고 있으므로, 이를 서비스할 차례이다. 11bit에서 27bit까지를 처리하게 되므로, for loop를 돌면서 해당하는 bit이 설정된 경우에만 do_IRQ() 함수를 호출한다. 따라서, 실제적인 인터럽트 번호와 관련된 처리는 do_IRQ()가 맡을 것이며, 넘겨주는 값은 IRQ_GPIO_11_27(x)가 된다. IRQ_GPIO_11_27(x)는 다시 아래와 같이 정의된다.

```
#define IRQ_GPIO_11_27(x) (32 + (x) - 11)
```

이는 32번 이상의 인터럽트 번호를 할당받아서 사용하기 위해서이다. 따라서, 11번 인터럽트는 32를 사용할 것이며, 계속적으로 12번은 33번을, 마지막으로 27은 인터럽트 번호 48을 사용할 것이다.

참고로, SA1100에 대한 할당된 인터럽트들로는 어떤 것이 있는지를 보려면, ~/include/asm-arm/arch-sa1100/irqs.h를 참고하기 바란다. 아래와 같이 되어있다.

#define SA1100_IRQ(x)	(0 + (x))
...	
#define IRQ_LCD	SA1100_IRQ(12) /* LCD controller */
#define IRQ_Ser0UDC	SA1100_IRQ(13) /* Ser. port 0 UDC */
#define IRQ_Ser1SDLC	SA1100_IRQ(14) /* Ser. port 1 SDLC */
#define IRQ_Ser1UART	SA1100_IRQ(15) /* Ser. port 1 UART */
#define IRQ_Ser2ICP	SA1100_IRQ(16) /* Ser. port 2 ICP */
#define IRQ_Ser3UART	SA1100_IRQ(17) /* Ser. port 3 UART */
#define IRQ_Ser4MCP	SA1100_IRQ(18) /* Ser. port 4 MCP */
#define IRQ_Ser4SSP	SA1100_IRQ(19) /* Ser. port 4 SSP */
#define IRQ_DMA0	SA1100_IRQ(20) /* DMA controller channel 0 */
#define IRQ_DMA1	SA1100_IRQ(21) /* DMA controller channel 1 */
#define IRQ_DMA2	SA1100_IRQ(22) /* DMA controller channel 2 */
#define IRQ_DMA3	SA1100_IRQ(23) /* DMA controller channel 3 */
#define IRQ_DMA4	SA1100_IRQ(24) /* DMA controller channel 4 */
#define IRQ_DMA5	SA1100_IRQ(25) /* DMA controller channel 5 */
#define IRQ_OST0	SA1100_IRQ(26) /* OS Timer match 0 */
#define IRQ_OST1	SA1100_IRQ(27) /* OS Timer match 1 */
#define IRQ_OST2	SA1100_IRQ(28) /* OS Timer match 2 */
#define IRQ_OST3	SA1100_IRQ(29) /* OS Timer match 3 */
#define IRQ_RTC1Hz	SA1100_IRQ(30) /* RTC 1 Hz clock */
#define IRQ_RTCAlrm	SA1100_IRQ(31) /* RTC Alarm */

코드 1005. SA1100의 인터럽트 번호 할당

코드에서 보듯이 LCD와 UART 및 DMA, RTC(Real Time Clock)등의 인터럽트 번호가 12번에서 31번까지로 정해져 있다.

```
#ifdef CONFIG_SA1111
    if( machine_is_assabet() && machine_has_neponset() ){
        /* disable all IRQs */
        INTEN0 = 0;
        INTEN1 = 0;
        /* detect on rising edge */
        INTPOL0 = 0;
        INTPOL1 = 0;
        /* clear all IRQs */
        INTSTATCLR0 = -1;
        INTSTATCLR1 = -1;
        for (irq = SA1111_IRQ(0); irq <= SA1111_IRQ(26); irq++) {
            irq_desc[irq].valid = 1;
            irq_desc[irq].probe_ok = 1;
            irq_desc[irq].mask_ack = sa1111_mask_and_ack_lowirq;
            irq_desc[irq].mask = sa1111_mask_lowirq;
            irq_desc[irq].unmask = sa1111_unmask_lowirq;
        }
        for (irq = SA1111_IRQ(32); irq <= SA1111_IRQ(54); irq++) {
            irq_desc[irq].valid = 1;
            irq_desc[irq].probe_ok = 1;
            irq_desc[irq].mask_ack = sa1111_mask_and_ack_highirq;
            irq_desc[irq].mask = sa1111_mask_highirq;
            irq_desc[irq].unmask = sa1111_unmask_highirq;
        }
    }
    if( machine_has_neponset() ){
        /* setup extra Neponset IRQs */
        irq = NEPONSET_ETHERNET_IRQ;
        irq_desc[irq].valid = 1;
        irq_desc[irq].probe_ok = 1;
        irq = NEPONSET_USAR_IRQ;
        irq_desc[irq].valid = 1;
        irq_desc[irq].probe_ok = 1;
        set_GPIO_IRQ_edge( GPIO_NEPE_IRQ, GPIO_RISING_EDGE );
        setup_arm_irq( IRQ_GPIO_NEPE_IRQ, &neponset_irq );
    }else{
        /* for pure SA1111 designs to come (currently unused) */
        set_GPIO_IRQ_edge( 0, GPIO_RISING_EDGE );
        setup_arm_irq( -1, &sa1111_irq );
    }
}
#endif

#if defined(CONFIG_SA1100_GRAPHICSCLIENT)
    if( machine_is_graphicsclient() ){
        /* disable all IRQs */
        ADS_INT_EN1 = 0;
        ADS_INT_EN2 = 0;
        /* clear all IRQs */
        ADS_INT_ST1 = 0xff;
        ADS_INT_ST2 = 0xff;
        for (irq = ADS_EXT_IRQ(0); irq <= ADS_EXT_IRQ(7); irq++) {
            irq_desc[irq].valid = 1;
            irq_desc[irq].probe_ok = 1;
            irq_desc[irq].mask_ack = ADS_mask_and_ack_irq0;
            irq_desc[irq].mask = ADS_mask_irq0;
            irq_desc[irq].unmask = ADS_unmask_irq0;
        }
}
```

```

        for (irq = ADS_EXT_IRQ(8); irq <= ADS_EXT_IRQ(15); irq++) {
            irq_desc[irq].valid = 1;
            irq_desc[irq].probe_ok = 1;
            irq_desc[irq].mask_ack = ADS_mask_and_ack_irq1;
            irq_desc[irq].mask = ADS_mask_irq1;
            irq_desc[irq].unmask = ADS_unmask_irq1;
        }
        GPDR &= ~GPIO_GPIO0;
        set_GPIO_IRQ_edge(GPIO_GPIO0, GPIO_FALLING_EDGE);
        setup_arm_irq(IRQ_GPIO0, &ADS_ext_irq );
    }
#endif
}

```

코드 1006. irq_init_irq() 함수의 정의(계속)

나마지는 각각의 board에 특수한 인터럽트의 설정을 다루고 있는 부분이다. 여기서 중요한 것은 모든 인터럽트에 대해서 초기화시에는 disable시키고, clear한다는 점과 어떤 edge에서 인터럽트가 생길지를 결정한다는 점이다. 마지막으로 호출하는 함수는 setup_arm_irq()이다. 이 함수는 ~/arch/arm/kernel/irq.c에 아래와 같이 정의되어 있다.

```

int setup_arm_irq(int irq, struct irqaction * new)
{
    int shared = 0;
    struct irqaction *old, **p;
    unsigned long flags;
    struct irqdesc *desc;

    if (new->flags & SA_SAMPLE_RANDOM) {
        rand_initialize_irq(irq);
    }
    desc = irq_desc + irq;
    spin_lock_irqsave(&irq_controller_lock, flags);
    p = &desc->action;
    if ((old = *p) != NULL) {
        /* Can't share interrupts unless both agree to */
        if (!(old->flags & new->flags & SA_SHIRQ)) {
            spin_unlock_irqrestore(&irq_controller_lock, flags);
            return -EBUSY;
        }
        /* add new interrupt at end of irq queue */
        do {
            p = &old->next;
            old = *p;
        } while (old);
        shared = 1;
    }
    *p = new;
    if (!shared) {
        desc->nomask = (new->flags & SA_IRQNOMASK) ? 1 : 0;
        desc->probing = 0;
        if (!desc->noautoenable) {
            desc->enabled = 1;
            desc->unmask(irq);
        }
    }
    spin_unlock_irqrestore(&irq_controller_lock, flags);
    return 0;
}

```

{}

코드 1007. setup_arm_irq()함수의 정의

`setup_arm_irq()` 함수는 ARM의 인터럽트를 초기화 하는 역할을 하는 것으로, 넘겨받는 값은 인터럽트 번호화 관련된 인터럽트 핸들러에 대한 기술을 가지는 `irqaction`구조체에 대한 포인터이다. 만약 설정하고자 하는 인터럽트 핸들러를 나타내는 기술자가 `flags` 필드에 `SA_SAMPLE_RANDOM`을 가진다면, 시스템의 `random seed generation`에 관련된 것으로 `rand_initialize_irq()`함수를 호출해서 처리하도록 만든다. 일반적인 디바이스 드라이버의 `interrupt action`은 이런 값을 가지지 않을 것이다.

이전 해당하는 인터럽트 디스크립터에 대한 주소를 얻는다(`desc`). 먼저 해당 디스크립터에 대한 수정을 하기 전에, `spin_lock_irqsave()`를 사용해서 `lock`을 설정하고, 디스크립터의 `action`필드에 대한 포인터를 얻어온다(`p`). 이것을 `old`로 두고, `NULL`이 아닌 경우에 다음과 같은 일을 한다(즉, 둘 이상의 인터럽트와 관련된 `action`을 설정하려고 하고 있다.)

이전에 설정된 `action`의 `flags` 필드가 인터럽트 공유를 지원하고(`SA_SHIRQ`), 새로이 설치될 `action`도 인터럽트 공유를 지원한다면, 계속 진행할 것이다. 그렇지 않다면, 둘 이상의 `action`을 설정하려고 하지만, 공유가 안된다는 말이므로, `lock`을 해제하고 에러값으로 `-EBUSY`를 돌려준다. 인터럽트의 공유를 두 `action`구조체가 모두 지원한다면, 새로운 인터럽트의 `action`구조체를 인터럽트 디스크립터의 `action`필드를 나타내는 곳의 `next`를 따라가서 제일 마지막에 덧붙인다. 이때 `shared`는 1이 설정될 것이다.

만약 `shared`가 0인 경우(즉, 공유 인터럽트를 사용하지 않는 것이 된다. 앞에서 해당하는 인터럽트 디스크립터의 `irqaction` 필드가 `NULL`인 경우일 것이다.) 이때는 그냥 새로이 `irqaction`필드를 단순히 채워주기만 하면 될 것이다. 디스크립터의 `nomask`를 `new->flags`과 `SA_IRQNOMASK` 값과 AND한 값에 따라서, 1이나 0으로 설정하고, `probing` 필드는 0으로 설정해서 현재 `probing`하고 있지 않다고 설정한다. 또한 `noautoenable`이 설정되지 않은 경우에는 `enable`상태로 만들어주기 위해서 `enabled`필드에 1을 넣고, `unmask()`함수에 넘겨받은 `irq`값을 주어서 해당하는 인터럽트의 `masking`을 풀어주게 된다. 마지막으로 복귀전에 `lock`을 풀어주게 되며, 0을 넘겨준다.

다시 더 이상 진행하기 전에, 앞에서 보았던 함수들에 대해서 조금더 알아보기로 하자. 이 함수들은 각각 `masking`과 `unmasking`, `masking` 및 `acknowledging`을 하는 함수들로 인터럽트 디스크립터를 설정할 때 사용했었다.

```
static void sa1100_mask_irq(unsigned int irq)
{
    ICMR &= ~(1 << irq);
}

static void sa1100_unmask_irq(unsigned int irq)
{
    ICMR |= (1 << irq);
}
```

코드 1008. sa1100_mask/unmask_irq()함수의 정의

`sa1100_mask_irq()` 함수와 `sa1100_unmask_irq()` 함수는 단지 넘겨받는 인터럽트 번호를 `ICMR`(Interrupt Controller Mask Register)에 반영하는 일을 한다. `masking`은 `ICMR`에 해당하는 인터럽트 번호의 `bit`을 clear하고, `unmasking`은 해당 인터럽트 번호의 `bit`을 set한다.

```
/*
 * SA1100 GPIO edge detection for IRQs.
 */
extern int GPIO_IRQ_rising_edge;
extern int GPIO_IRQ_falling_edge;
/*
 * GPIO IRQs must be acknowledged. This is for IRQs from 0 to 10.
 */
static void sa1100_mask_and_ack_GPIO0_10_irq(unsigned int irq)
```

```
{
    ICMR &= ~(1 << irq);
    GEDR = (1 << irq);
}
static void sa1100_mask_GPIO0_10_irq(unsigned int irq)
{
    ICMR &= ~(1 << irq);
}
static void sa1100_unmask_GPIO0_10_irq(unsigned int irq)
{
    GRER = (GRER & ~(1 << irq)) | (GPIO_IRQ_rising_edge & (1 << irq));
    GFER = (GFER & ~(1 << irq)) | (GPIO_IRQ_falling_edge & (1 << irq));
    ICMR |= (1 << irq);
}
```

코드 1009. sa1100_mask/unmask/mask_and_ack_GPIO0_10_irq() 함수의 정의

SA1100_mask./unmask/mask_and_ack_GPIO0_10_irq() 함수들은 ICMR에 대한 조작과 GPIO에 대한 조작을 수반한다. 먼저 masking과 acknowledging하기 위해서는 ICMR에 해당 인터럽트 bit를 clear하고 GEDR에는 1을 써서 ACK해준다. Masking을 위해서는 ICMR에 해당 인터럽트 bit를 clear하는 일을 하며, unmasking하는 것은 GRER과 GFER에는 각각의 edge detection상황을 나타내고 있는 GPIO_IRQ_rising(falling)_edge에 해당 인터럽트 bit를 AND 시켜고, 원래의 GRER과 GFER값에서 해당 인터럽트 bit를 clear한 값을 OR시켜서 넣는다. ICMR에는 당연히 해당 인터럽트 bit이 set되어야 할 것이다.

```
static int GPIO_11_27_enabled;           /* enabled i.e. unmasked GPIO IRQs */
static int GPIO_11_27_spurious;         /* GPIOs that triggered when masked */
...
static void sa1100_mask_and_ack_GPIO11_27_irq(unsigned int irq)
{
    int mask = (1 << GPIO_11_27_IRQ(irq));
    GPIO_11_27_spurious &= ~mask;
    GPIO_11_27_enabled &= ~mask;
    GEDR = mask;
}
static void sa1100_mask_GPIO11_27_irq(unsigned int irq)
{
    int mask = (1 << GPIO_11_27_IRQ(irq));
    GPIO_11_27_spurious &= ~mask;
    GPIO_11_27_enabled &= ~mask;
}
```

코드 1010. sa1100_mask/mask_and_ack_GPIO11_27_irq() 함수의 정의

GPIO 11에서 27까지는 mutiplexing해서 사용하고 있음을 앞에서 보았다. 이들을 위한 masking과 masking 및 ACK를 처리하는 함수들이다. 먼저 masking가 ACK를 처리하기 위해서 GPIO_11_27_IRQ(x)를 불러서 mask 변수의 값을 구한다. GPIO_11_27_IRQ(x)는 아래와 같다.

#define GPIO_11_27_IRQ(i) (11 + (i) - 32)

이렇게 해서 구하는 값은 원래의 IRQ(32번 부터 시작해서 주어진 인터럽트 번호)에서 해당하는 GPIO pin의 번호를 구해준다. 이 값을 GPIO_11_27_spurious 및 GPIO_11_27_enabled와 NOT시켜서 AND한다. 이렇게 해서 일단 해당하는 인터럽트를 masking(disable) 시켜준다. 그리고나서, GEDR을 clear시키기 위해서 mask 변수 값을 쓴다. 즉, ACK 해주는 것이다. 단순히 masking만을 하려한다면, 앞에서 해준 GEDR에 쓰는 과정을 생략하면 될 것이다.

```

static void sa1100_unmask_GPIO11_27_irq(unsigned int irq)
{
    int mask = (1 << GPIO_11_27_IRQ(irq));
    if (GPIO_11_27_spurious & mask) {
        /*
         * We don't want to miss an interrupt that would have occurred
         * while it was masked. Simulate it if it is the case.
         */
        int state = GPLR;
        if (((state & GPIO_IRQ_rising_edge) |
             (~state & GPIO_IRQ_falling_edge)) & mask) {
            do_IRQ(irq, NULL);
            /* we are being called again from do_IRQ() so ... */
            return;
        }
    }
    GPIO_11_27_enabled |= mask;
    GRER = (GRER & ~mask) | (GPIO_IRQ_rising_edge & mask);
    GFER = (GFER & ~mask) | (GPIO_IRQ_falling_edge & mask);
}

```

코드 1011. sa1100_unmask_GPIO11_27_irq()함수의 정의

sa1100_unmask_GPIO11_27_irq()함수는 앞에서 했던 masking연산을 꺼꾸로 돌리는 일을 한다. 마찬가지로 GPIO_11_27_IRQ() 매크로를 사용해서 mask값을 구해오고, 만약 mask된 동안에 발생했을지도 모를 인터럽트를 처리하기 위해서 GPIO_11_27_spurious와 mask값을 AND시켜본다. state에는 GPLR(GPIO Pin Level Register)를 읽어와서, 현재 pin의 상태를 가져온다. 만약 state와 GPIO_IRQ_rising_edge와 AND 시켜서 나온 값과, GPIO_IRQ_falling_edge와 state값을 NOT시킨 값을 AND 시키면서 나온 값을 OR 시켰을 때 어떤 값이 있고, mask가 값을 가지고 있다면, do_IRQ()를 호출해서 인터럽트를 처리해주고 바로 복귀(return)하도록 한다. GPLR에 있는 각 bit의 값은 0이때는 pin이 low인 것을 나타내며, 1인 경우에는 pin이 high인 것을 나타낸다. 따라서, state를 그냥 AND시키거나 NOT을 한 후에 AND시키는 것은 rising edge와 falling edge를 감지하기 위한 것이다.

이전 GPIO_11_27_enabled에 unmasking값을 더해주고(mask), GRER과 GFER에는 각각 이전에 있던 GRER과 GFER값에서 mask값을 clear 한 후에, 다시 이 값에 GPIO_IRQ_rising(falling)_edge값과 mask값을 AND시킨 값에 OR시켜서 넣어주도록 한다. 이는 원래의 GRER과 GFER의 설정된 값에 현재의 mask설정을 해주는 일을 한다.

17.7.3.5. do_IRQ() 함수의 분석

이 함수는 앞에서 인터럽트를 demultiplexing하는 곳에서 잠시 보았다. 실제적인 인터럽트를 처리하는 부분이기 때문에, 나중에 인터럽트에 대한 이야기를 더 하려고 한다면, 이곳에서 잠시 지켜보고 넘어가야 할 것이다. 정의는 ~/arch/arm/kernel/irq.c에 아래와 같이 나와있다.

```

asmlinkage void do_IRQ(int irq, struct pt_regs * regs)
{
    struct irqdesc * desc;
    struct irqaction * action;
    int cpu;

    irq = fixup_irq(irq);
    /*
     * Some hardware gives randomly wrong interrupts. Rather
     * than crashing, do something sensible.
     */
    if (irq >= NR_IRQS)
        goto bad_irq;
    desc = irq_desc + irq;

```

```

spin_lock(&irq_controller_lock);
desc->mask_ack(irq);
spin_unlock(&irq_controller_lock);
cpu = smp_processor_id();
irq_enter(cpu, irq);
kstat.irqs[cpu][irq]++;
desc->triggered = 1;
/* Return with this interrupt masked if no action */
action = desc->action;

```

코드 1012. do_IRQ()함수의 정의

do_IRQ()함수는 디바이스의 모든 일반적인 인터럽트를 다루는 함수이다. 넘겨받는 값으로는 인터럽트 번호와 레지스터의 상태를 보관하는 값이다. 먼저 넘겨받은 인터럽트 번호를 가지고 fixup_irq()함수⁴⁰²를 호출해서 수정된 인터럽트 번호를 얻는다. 이렇게 계산된 irq값이 NR_IRQS보다 크다면, 즉, 시스템에서 사용하는 전체 인터럽트 번호보다 크다면, 당연히 잘못된 인터럽트 번호이므로 bad_irq로 제어를 옮긴다. 모든 인터럽트에 대한 디스크립터를 가지는 irq_desc에 계산된 irq값을 더해서, 인터럽트 번호에 해당하는 인터럽트 디스크립터에 대한 포인터를 얻어서 desc 변수에 준다. 이젠 인터럽트에 대한 핸들링을 할 차례로 먼저 인터럽트 콘트롤러에 대한 lock을 설정한다(spin_lock()). 디스크립터에 설정된 mask_ack 필드에 들어 있을 함수를 호출해서 인터럽트에 대한 mask의 설정과 ACK를 해준다. 이것을 마치면, 앞에서 설정했던 lock을 풀게 된다(spin_unlock()).

커널에서 관리정보를 유지하기 위해서 인터럽트를 처리하는 CPU가 어느 것인지를 구하고(smp_processor_id()), 이 CPU가 현재 interrupt를 처리한다는 것을 나타낸다(irq_enter()⁴⁰³). 커널의 인터럽트 정보를 업데이트하기 위해서 kstat.irq[][]를 증가 시켜준다. 이젠 인터럽트가 triggering되었다는 것을 나타내기위해서 인터럽트 디스크립터의 triggered flag를 1로 설정한다. 이것을 마치고나면, 인터럽트를 실질적으로 처리하는 설치된 인터럽트 action에 대한 구조체의 포인터를 얻어서, 이를 action변수로 나타낸다.

```

if (action) {
    int status = 0;
    if (desc->nmask) {
        spin_lock(&irq_controller_lock);
        desc->unmask(irq);
        spin_unlock(&irq_controller_lock);
    }
    if (!(action->flags & SA_INTERRUPT))
        __sti();
    do {
        status |= action->flags;
        action->handler(irq, action->dev_id, regs);
        action = action->next;
    } while (action);
    if (status & SA_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);
    __cli();
    if (!desc->nmask && desc->enabled) {
        spin_lock(&irq_controller_lock);
        desc->unmask(irq);
        spin_unlock(&irq_controller_lock);
    }
}

```

⁴⁰² fixup_irq() 함수는 SA1100에서는 단순히 원래의 irq값 만을 돌려주는 매크로로 ~/include/asm-arm/arch-sa1100/irq.h에 정의되어 있다.

⁴⁰³ 이 매크로는 단순히 현재 CPU의 인터럽트 카운트를 증가시키는 일만을 해준다. 반대되는 것으로는 아래에서 보게될 irq_exit() 매크로가 있다. 인터럽트의 처리로 진입하기 전에 irq_enter()를 해주고, 나중에 처리를 마치고 나올 때, irq_exit()을 해주면 될 것이다.

```

    }
}

/*
 * Debug measure - hopefully we can continue if an
 * IRQ lockup problem occurs...
 */
check_irq_lock(desc, irq, regs);
irq_exit(cpu, irq);
if (softirq_active(cpu) & softirq_mask(cpu))
    do_softirq();
return;
bad_irq:
irq_err_count += 1;
printk(KERN_ERR "IRQ: spurious interrupt %d\n", irq);
return;
}

```

코드 1013. do_IRQ()함수의 정의(계속)

action변수가 NULL이 아니라면, if절 이하를 수행할 것이다. 디스크립터의 nomask필드가 설정되었다면, 이를 수행해야 할 것이다. 즉, unmask필드에 들어있는 함수를 인터럽트 번호를 넘겨주어서 호출한다. 호출전과 호출후에는 반드시 lock을 설정하고, 해제하는 것이 들어있어야 할 것이다. 처리가 끝나면, 인터럽트 action구조체의 flags에 SA_INTERRUPT⁴⁰⁴ 가 들어있는지 살펴본다. 만약 없다면, __sti()를 호출해서 모든 인터럽트가 발생하지 못하도록 만든다. do while() loop는 해당하는 인터럽트 핸들러들을 차례로 호출하는 역할을 한다. action의 next가 NULL이 아닌동안 진행된다.

만약 SA_SAMPLE_RANDOM이 설정된 경우에는 randomness를 반영하기 위해서 add_interrupt_randomness() 함수를 호출해준다. 이것을 마치면 인터럽트 핸들링이 끝났으므로, __cli()를 호출해서 인터럽트를 다시 enable시켜주도록 한다. 만약 인터럽트 디스크립터의 nomask필드가 없고(NULL), enabled필드가 설정된 경우에는 디스크립터의 unmask필드에 설정된 함수를 수행해서 인터럽트를 unmasking해 준다. 역시 lock을 설정하고 해제하는 일이 필요하다.

check_irq_lock()함수⁴⁰⁵는 IRQ lock을 detection하는 역할을 하는 함수이다. 이 함수를 사용하는 이유는 source에서도 볼 수 있듯이 IRQ lockup 문제가 발생하더라도 계속 진행하기 위함이다. irq_exit()는 앞에서 나온 irq_enter() 매크로의 반대 역할을 하게 되며, 만약 현재 CPU에 인터럽트 처리를 하는 도중에 소프트웨어 인터럽트가 설정되었는지를 확인해서, 설정된 소프트웨어 인터럽트가 있다면, do_softirq()함수를 호출해서 이를 처리하도록 한다. 이것은 디바이스에서 발생한 인터럽트를 빨리 처리해주고, 관련되어 발생할지도 모르는 소프트웨어 인터럽트도 함께 처리해 주기 위함이다. 복귀 값은 없다. bad_irq는 잘못된 인터럽트 번호를 가질 때 처리할 목적으로 단순히 irq_err_count를 증가시키고, 에러 메시지를 출력해줄 뿐이다. 복귀값은 역시 없다.

이하에서는 인터럽트와 관련해서 조금 더 추가적으로 볼 것들을 정리해 보았다. Booting에서만 볼 것이 아니라, 일반적으로 발생하는 인터럽트의 처리와 관련된 것이니 참고하기 바란다.

⁴⁰⁴ SA_INTERRUPT는 인터럽트 핸들러가 반드시 모든 interrupt가 disable된 상황에서 실행되어야 한다는 것을 나타낸다. 참고로, SA_SHIRQ는 다른 디바이스와 인터럽트 라인이 공유되는 것을 허락한다는 말이며, SA_SAMPLE_RANDOM은 디바이스가 random하게 발생하는 사건의 source로서 생각해도 된다는 의미이다. 따라서, 커널은 random number generator로 디바이스를 사용할 수 있다. 사용자 프로세스는 /dev/random이나 /dev/urandom 디바이스 파일로 부터 random number를 취함으로써 이러한 특징을 접근할 수 있게 된다. 또한, SA_PROBE는 커널이 hardware device를 probe하는 일을 하는데, 인터럽트 라인을 사용하고 있음을 나타낸다.

⁴⁰⁵ 정확히 이 함수를 사용하는 이유를 지금으로서는 파악할 수 없다. 다만 이 함수는 인터럽트 디스크립터의 lock count필드와 lock program count, lock jiffies값에 변화를 준다는 사실만을 기억하기 바란다.

17.7.3.6. Interrupt Service의 구현

SA1100을 사용하는 경우에 발생할 수 있는 인터럽트는 사용자 모드인 경우에 생기는 인터럽트와 시스템모드(커널 모드)에서 발생하는 인터럽트로 크게 구분할 수 있다. 각각을 처리하는 함수(혹은, 어셈블리 언어로 되어 있는 경우에는 export된 레이블(label)이 될 것이다.)는 앞에서 trap_init()함수를 설명하면서 본 “.LCtab_irq”에 있는 irq_svc와 irq_usr이 될 것이다. 각각은 supervisor 모드에서의 IRQ 처리와 user 모드에서의 IRQ처리를 담당하고 있다. 각 함수의 정의는 ~/arch/arm/kernel/entry-armv.S에 아래와 같이 되어 있다.

```

.align 5
__irq_svc:
    sub    sp, sp, #S_FRAME_SIZE
    stmia sp, {r0 - r12}           @ save r0 - r12
    ldr    r7, .LCirq
    add    r5, sp, #S_FRAME_SIZE
    ldmia r7, {r7 - r9}
    add    r4, sp, #S_SP
    mov    r6, lr
    stmia r4, {r5, r6, r7, r8, r9}   @ save sp_SVC, lr_SVC, pc, cpsr, old_ro
1:   get_irqnr_and_base r0, r6, r5, lr
    movne r1, sp
    @
    @ routine called with r0 = irq number, r1 = struct pt_regs *
    @
    adrsvc ne, lr, 1b
    bne   do_IRQ
    ldr    r0, [sp, #S_PSR]        @ irqs are already disabled
    msr    spsr, r0
    ldmia sp, {r0 - pc}^          @ load r0 - pc, cpsr

```

코드 1014. __irq_svc()함수의 정의

먼저 보는 것은 __irq_svc() 함수이다. 이 함수는 현재 시스템의 모드가 supervisor모드에서 인터럽트가 걸린 것이다. 먼저 sp(stack pointer)에서 S_FRAME_SIZE(=72)만큼을 빼도록 한다. sp가 가르키는 곳에 r0에서 r12까지를 레지스터를 저장한다. 그리고나서, “.LCirq”로부터 값을 읽어서 r7에 넣는다. 이 값은 __temp_irq를 나타내는 것으로 보관된 IRQ 모드에서의 lr(link register) 역할을 한다. 다시 sp에 S_FRAME_SIZE를 더해서 r5에 놓고, r7이 가르키는 곳에서 r7, r8, r9으로 데이터를 가져온다. 이 값은 보관된 lr, 보관된 spsr, 이전의 r0가 된다. 이전 sp에 S_SP(=52)를 더해서 r4로 놓고, r6에는 lr값을 가져온다. 그리고나서, r4가 가르키는 곳에 r5, r6, r7, r8, r9를 저장하도록 한다. 즉, sp에서 52를 더한 곳에 이전의 sp값과 현재의 lr, 이전에 있던 pc, 이전의 cpsr, 이전의 r0값을 저장하도록 하는 것이다.

여기서 get_irqnr_and_base()는 SA1100 architecture로 정의된 경우에는 아래와 같이 ~/arch/arm/kernel/entry-armv.S에 정의되어 있다.

```

...
#endif defined(CONFIG_ARCH_SA1100)
.macro disable_fiq
.endm

.macro get_irqnr_and_base, irqnr, irqstat, base, tmp /* 매크로 파라미터 : r0, r6, r5, lr */
    mov    r4, #0xfa000000           @ ICIP = 0xfa050000
    add    r4, r4, #0x00050000
    ldr    \irqstat, [r4]
    ldr    \irqnr, [r4, #4]          @ get irqs
    ands   \irqstat, \irqstat, \irqnr
    mov    \irqnr, #0
    beq    1001f
    tst    \irqstat, #0xff

```

```

moveq  \irqstat, \irqstat, lsr #8
addeq  \irqnr, \irqnr, #8
tsteq  \irqstat, #0xff
moveq  \irqstat, \irqstat, lsr #8
addeq  \irqnr, \irqnr, #8
tsteq  \irqstat, #0xff
moveq  \irqstat, \irqstat, lsr #8
addeq  \irqnr, \irqnr, #8
tst    \irqstat, #0x0f
moveq  \irqstat, \irqstat, lsr #4
addeq  \irqnr, \irqnr, #4
tst    \irqstat, #0x03
moveq  \irqstat, \irqstat, lsr #2
addeq  \irqnr, \irqnr, #2
tst    \irqstat, #0x01
addeqs \irqnr, \irqnr, #1
1001:
.endm

.macro  irq_prio_table
.endm

```

코드 1015. get_irqnr_and_base() 매크로의 정의

get_irqnr_and_base() 매크로는 넘겨받는 파라미터 갓으로 인터럽트 번호를 나타내는 irqnr과, 인터럽트의 상태를 나타내는 irqstat 및 base주소(기본주소)와 일시적으로 사용할 tmp가 있다. 코드의 전후에 disable_fiq() 매크로와 irq_prio_table() 매크로는 아무런 정의도 가지고 있지 않음을 알 수 있다.

먼저 r4에 0xFA000000을 가져온다. 이 값에 0x00050000을 더해서 0xFA050000⁴⁰⁶의 값을 만들어 준다. 이 값은 ICIP(Interrupt Controller IRQ Pending Register)⁴⁰⁷의 주소값이다. 이 값에서 irqstat로 어떤 IRQ가 pending되어 있는지를 읽어온다. 다시 이 값에 4를 더해서 ICMR(Interrupt Controller Mask Register)를 찾아서 그 값을 irqnr에 넣는다. 이렇게 구한 iqstat에서 masking된 인터럽트를 제외하기 위해서, 방금 앞에서 구한 두개의 값을 AND시켜서 irqstat에 넣는다. irqnr에는 0의 값을 주고, 앞에서 AND연산이 CPSR의 Z bit를 설정했다면(equal), 1001 레이블로 제어를 옮긴다. 그렇지 않다면, 계속 진행된다.

irqstat와 0xFF를 test해서(비교해서), 같다면, XXXeq라고 되어있는 연산들을 수행한다. 먼저 같다면, irqstat를 우측으로 8bit shift한다. irqnr에는 8을 더해주고, 다시 irqstat와 0xFF를 비교한다. 또 같다면, irqstat를 다시 8bit 우측으로 shift하고, irqnr에는 8을 더해준다. 이런 식으로 해서 irqstat에 들어있는 32bit값에 대해서 어떤 인터럽트가 생성되었는지를 찾는다. 여기서 중요한 것은 인터럽트가 우선 순위가 있다는 점이다. 즉, 높은 bit에서 걸린 인터럽트를 먼저 처리하기 위해서 계속적으로 irqnr에 해당하는 인터럽트가 있는지를 보고, 값을 더해준다는 것으로 유추할 수 있을 것이다. 따라서, 우리가 구한 값은 irqnr이다. 이 변수는 발생한 인터럽트중 가장 우선 순위가 높은 인터럽트의 번호를 가진다.

참고로 매크로의 정의는 “.macro”로 시작해서, 매크로의 이름과 파라미터들의 리스트로 구성되며, 내부에서 파라미터들은 “parameter”的 형식으로 사용된다. 매크로의 끝은 “.endm”이 나타낸다.

⁴⁰⁶ ICIP는 원래 물리적인 주소값으로 0x90050000(_ICIP)를 가진다. 하지만 ~/include/asm-arm/arch-sa1100/SA-1100.h에 정의된 내용을 보면, io_p2v(_ICIP)로 되어 있음을 볼 수 있으며, 다시 ~/include/asm-arm/arch-sa1100/hardware.h에 io_p2v() 매크로는 물리적인 주소 0x80000000을 0xFA000000으로, 0x90000000을 0xFA000000으로, 0xA0000000을 0xFC000000으로, 0xB0000000을 0xFE000000으로 매핑하고 있음을 찾을 수 있다. 따라서, 현재 MMU가 enable된 상황에서는 물리적인 주소로 곧바로 접근하기보다는, 이렇게 새롭게 mapping된 논리적인 주소 0xFA050000으로 접근을 해야할 것이다.

⁴⁰⁷ ICFP(Interrupt Controller FIQ Pending Register)는 FIQ에서 발생한 인터럽트 request의 상태를 가지고 있는 레지스터이다. 현재 Linux에서는 FIQ를 사용하지 않기에 이곳을 읽어줄 필요는 없다.

다시 앞의 이야기로 돌아가서 `get_irqnr_and_baae()` 매크로의 실행 후에, r0가 `irqnr`을 가짐을 알 수 있을 것이다. 그리고, 매크로 연산에서 제일 마지막에 “s”와 같은 것을 붙여서 CPSR의 변화를 주었고, 이것을 통해서 CPSR의 Z bit이 clear가 되었다면, `movne`가 수행되면서, r1에는 `sp`를 위치시킬 것이다.

```
.macro adrsvc cond, reg, label
adr\cond          \reg, \label
.endm
```

코드 1016. `adrsvc()` 매크로의 정의

`adrsvc`는 `cond`, `reg`, `label`을 받아드려서 `adr[cond]`를 만들고, `reg`에 `label`을 불러오는 역할을 한다. 따라서, “`adrsvc ne, lr, 1b`”와 같이 사용한다면, “`adrne lr, 1b`”와 같이 변환이 일어나게 된다. 이것은 나중에 다시 `return` 상황이 발생했을 때, 어디서부터 수행을 계속할 것인가를 결정하는 것이되며, 인터럽트가 있다면, 계속적으로 수행될 것이다. `adr`명령은 ARM instruction으로 정해진 것은 아니지만, pseudo instruction으로 `ldr`과 같은 일을 하지만, literal pool⁴⁰⁸을 사용하지 않는 명령이다. 따라서, 여기서는 `label`의 주소를 `register`로 가져오는 일을 한다.

이전 인터럽트 번호를 알고 있으며(`irqnr=r0`), r1에는 인터럽트가 발생했을 때의 stack에 저장된 레지스터들에 대한 포인터도 알고 있다. `do_IRQ()`함수를 호출해서 해당하는 인터럽트 service를 받도록 한다. 인터럽트 service가 끝나면, r0에는 `sp(r14)`에 `S_PSR(=64)`를 더한 값을 index로 사용해서 값을 가져온다. 가져온 값을 `spsr`에 채워넣고, `sp`가 가르키는 곳에서 r0부터 `pc(r15)`까지의 원래 저장된 값을 다 가져온다. 마지막에 있는 “^”은 인터럽트에서 복귀할 때 사용자 모드에서 사용하던 레지스터들과 `spsr`에 저장된 `cpsr`값을 복구하기 위함이다.

```
.align 5
__irq_usr:
    sub    sp, sp, #S_FRAME_SIZE
    stmia sp, {r0 - r12}           @ save r0 - r12
    ldr    r4, .LCirq
    add    r8, sp, #S_PC
    ldmia r4, {r5 - r7}           @ get saved PC, SPSR
    stmia r8, {r5 - r7}           @ save pc, psr, old_r0
    stmdb r8, {sp, lr}^
    alignment_trap r4, r7, __temp_irq
    zero_fp
1:   get_irqnr_and_base r0, r6, r5, lr
    movne r1, sp
    adrsvc ne, lr, 1b
    @
    @ routine called with r0 = irq number, r1 = struct pt_regs *
    @
    bne    do_IRQ
    mov    r4, #0
    get_current_task r5
    b      ret_with_reschedule
```

코드 1017. `__irq_usr()` 함수의 정의

`irq_usr()` 함수는 사용자 모드에서 발생한 인터럽트의 처리를 담당하는 함수이다. 앞에서 본 `irq_svc()`와 거의 비슷하지만, 한가지 다른점은 사용자 모드에서 진입한 인터럽트라는 것이다. 먼저 `sp`는 `S_FRAME_SIZE`를 빼주고, `sp`가 가르키는 주소에 r0에서 r12까지의 레지스터의 내용을 저장한다. r4에는 `.LCirq`를 가져오고, `sp`에 `S_PC(=52)`를 빼서, r8에 넣는다. r4에서 시작하는 곳에 있는 값을 각각 r5,

⁴⁰⁸ Literal Pool이란 어떤 값에 대한 정의를 가지는 문장으로, 예를 들어서 `ldr`의 레지스터에 “`=const`”와 같은 명령어를 사용할 때, `const`가 `ldr` instruction에서 사용할 수 있는 연산자의 범위를 벗어날 때, 참조해서 가져올 수 있도록 상수를 정의하는 부분을 말한다. 이곳에서는 그냥 `adr(address)` pseudo instruction을 사용해서 literal pool의 사용을 없앴다.

r6, r7에 읽어오도록 한다. 이것은 앞에서 인터럽트로 진행할 때 저장된 pc와 spsr 값을 읽어오는 일을 한다. 이렇게 읽어온 값을 앞에서 계산된 r8(stack)에 차례로 저장하고(pc, cpsr, r0), sp와 lr을 다시 pc와 cpsr0이 저장된 영역에 넣는다. 마지막에 있는 “^”은 현재 모드의 레지스터 대신에 사용자 모드에서의 레지스터 bank를 사용하도록 만든다.

```
.macro alignment_trap, rbase, rtemp, sym
    ldr    \rtemp, [rbase, #OFF_CR_ALIGNMENT(\sym)]
    mcr    p15, 0, \rtemp, c1, c0
.endm
```

코드 1018. alignment_trap() 매크로의 정의

alignment_trap() 매크로는 파라미터 값으로 3개를 넘겨받는다. 이 값을 이용해서 ldr 연산을 만들어 내게 되는데, 넘겨주는 값으로 base, temp, symbol 값이 있다. 매크로를 expand시켜서 보면, 아래와 같은 코드가 된다.

```
ldr    r7, [ r4, #OFF_CR_ALIGNMENT(__temp_irq) ]
mcr    p15, 0, r7, c1, c0
...
SYMBOL_NAME(cr_alignment):
.space 4
```

OFF_CR_ALIGNMENT은 단순히 넘겨받은 값을 cr_alignment 변수에서 빼는 역할을 한다. cr_alignment은 일단 4 bytes를 가지는 공간을 할당 받고 있으며, mcr 명령을 통해서 coprocessor 15(=p15)에 r7에 있는 내용을 c1레지스터에 적는 일을 한다.

```
.macro zero_fp
#ifndef CONFIG_NO_FRAME_POINTER
    mov    fp, #0
#endif
.endm
```

코드 1019. zero_fp() 매크로의 정의

zero_fp() 매크로는 단순히 fp(frame pointer) 레지스터를 설정하는 역할을 한다. 코드에서 CONFIG_NO_FRAME_POINTER가 설정되지 않았다면, fp에 0을 넣는다.

앞에서와 마찬가지로 get_irqnr_and_base() 매크로는 r0에 interrupt 번호를 가져올 것이다. r1에는 사용하는 stack의 pointer를 놓고, adrsvc역이 앞에서 마찬가지로 lr를 1b로 설정한다. 이젠 do_IRQ()를 호출해서 인터럽트를 처리하는 일만 남았다. do_IRQ()를 호출해서 인터럽트의 처리를 마쳤다면, get_current_task() 매크로를 호출한다. 넘겨주는 값은 r5 레지스터이다.

```
.macro get_current_task, rd
    mov    \rd, sp, lsr #13
    mov    \rd, \rd, lsl #13
.endm
```

코드 1020. get_current_task() 매크로의 정의

get_current_task() 매크로는 sp를 우측으로 13bit shift해서 rd 레지스터에 두고, 이 값을 다시 좌측으로 13 bit shift하는 역할을 하는 함수이다. 이렇게 함으로써, 하위 13bit를 clear한다. 이는 Linux에서 현재 task의 task_struct 구조체를 stack에 보존하기 때문에, 이것을 수행해서 r5에 task_struct 구조체의 포인터를 가져오기 위함이다.

이전 사용자 모드로 다시 돌아가기 전에, ret_with_reschedule로 제어를 옮겨서, 혹시 새로이 스케줄링 해줄 일이 생겼는지를 확인한다. 스케줄링 요청이 있다면, 새로운 프로세스를 선택해서 수행해 줄

것이다. 즉, 커널이 시스템 콜⁴⁰⁹의 return과 인터럽트의 복귀시에 새로운 스케줄링 요구가 있는지를 확인하는 부분이다.

앞에서 우린 두가지의 인터럽트 처리에 대한 것을 보았다. 주로 entry-XXX.S라고 불어 있는 부분은 시스템 콜이나, 인터럽트의 진입 및 복귀 부분에 해당하는 것이며, 인터럽트의 처리는 __irq_svc/usr() 함수가 맡고 있다. 인터럽트의 실제적인 처리는 do_IRQ() 함수가 맡고 있으며, 이 함수는 __irq_svc/usr() 함수에서 호출된다. 만약 사용자 모드에서 인터럽트 요구가 있었다면, 복귀하는 부분은 ret_with_reschedule에 해당하며, 이것은 ~/arch/arm/kernel/entry-common.S에 정의되어 있다.

ret_with_reschedule:	@ external entry (r5 must be set) (__irq_usr)	
ldr	r0, [r5, #TSK_NEED_RESCHED]	
ldr	r1, [r5, #TSK_SIGPENDING]	
teq	r0, #0	
bne	ret_reschedule	
teq	r1, #0	@ check for signals
blne	ret_signal	

코드 1021. ret_with_reschedule() 함수의 정의

ret_with_reschedule() 함수는 r5가 현재의 task의 task_struct 구조체를 가지는 포인터 역할을 한다고 가정한다. 이곳에서 TSK_NEED_RESCHED (= offset of need_resched 필드)를 더한 값을 index로 사용해서 r0의 값을 읽어온다. r1에는 TSK_SIGPENDING (= offset of sigpending 필드)를 더한 값을 index로 사용해서 값을 읽어온다. 만약 r0이 0을 가지지 않는다면, 다시 스케줄링 될 필요가 있다는 말이 되므로 ret_reschedule로 제어를 옮긴다. 만약 r1이 0을 가지지 않는다면, pending된 시그널이 있으므로 이를 처리해 주어야 할 것이다. blne를 사용해서 link를 저장하고 branch한다.

ret_reschedule:	adrsvc al, lr, ret_with_reschedule @ internal
b	SYMBOL_NAME(schedule)

코드 1022. ret_reschedule의 정의

ret_reschedule은 단지 lr에 ret_with_reschedule의 주소를 기록하고, schedule() 함수를 호출하는 역할을 한다. schedule() 함수는 커널의 스케줄링을 담당하는 핵심 함수이며, 여기서는 다루지 않는다. 다만, 스케줄링 요구를 처리한다는 것만 기억하기 바란다.

ret_signal:	mov r1, sp @ internal
mov	r2, r4
b	SYMBOL_NAME(do_signal) @ note the bl above sets lr

코드 1023. ret_signal의 정의

ret_signal은 r1에 sp(stack pointer)를 놓고, r2에는 r4를 놓은 후, do_signal() 함수를 호출하는 역할을 한다. 앞에서 이미 lr를 설정해 놓은 상태이므로 do_signal을 수행한 이후에는 ret_with_reschedule의 제일 마지막 다음의 주소에 있는 instruction을 수행할 것이다. 이것은 아래와 같다.

⁴⁰⁹ 이곳에서는 단순히 사용자 모드에서 진행하다가 인터럽트가 걸려서 서비스를 했으며, 이전 복귀하기 위한 절차를 밟는 도중이다. 즉, 시스템 콜과는 무관한다. 이는 단순히 시스템에서 rescheduling이 일어나는 것을 설명해줄 목적으로 적은 부분이다.

```

/ 이 부분은 ret_with_reschedule 이후에 놓인다.*/
ret_from_all:      restore_user_regs      @ internal
...
/* ~/arch/arm/kernel/entry-armv.S에서 */
.macro  restore_user_regs
    ldr    r0, [sp, #S_PSR]           @ Get calling cpsr
    mov    ip, #I_BIT | MODE_SVC
    msr    cpsr_c, ip              @ disable IRQs
    msr    spsr, r0                @ save in spsr_svc
    ldmia  sp, {r0 - lr}^          @ Get calling r0 - lr
    mov    r0, r0
    ldr    lr, [sp, #S_PC]          @ Get PC
    add    sp, sp, #S_FRAME_SIZE
    movs   pc, lr                @ return & move spsr_svc into cpsr
.endm

```

코드 1024. ret_from_all의 정의

ret_from_all은 단순히 restore_user_regs() 매크로를 호출한다. restore_user_regs() 매크로는 단순히 사용자 모드에서 시스템 모드로 변화되는 순간에 저장했던 레지스터들을 다시 다 복구하는 일을 한다. sp에서 S_PSR(=64)를 더한 값의 내용을 가져와서 r0로 놓고, ip(=r7)는 I_BIT과 MODE_SVC bit들을 OR시키도록 한다. 이 값으로 cpsr_c(현재 모드의 cpsr 레지스터)를 설정한다. 즉, Interrupt를 disable시키고, supervisor 모드로 만든다. 또한 spsr에는 앞에서 가져온 spsr의 값으로 다시 채워두고, sp에서 r0에서 r15(=lr)까지의 내용을 가져온다. “^”를 설정함으로써 cpsr 값도 불러온다. 이젠 저장된 PC(Program Counter) 값을 읽어올 차례이다. sp에 S_PC(=60)을 더해서 lr 값을 취한다(PC). sp는 원래의 값을 가지도록 S_FRAME_SIZE(=72)를 더해주게 되며, 마지막으로 얻어온 lr 값을 pc를 채워넣는다. 이것으로 사용자 모드로 복귀하게 된다.

이젠 사용자 모드에서의 모든 레지스터의 내용이 복구가 되었으므로 계속적으로 수행이 가능할 것이다. 이것으로 ARM(SA1100)에서의 interrupt service에 대한 간단한 설명을 마치도록 하고, 앞에서 진행하던 이야기로 돌아가서 start_kernel() 함수를 계속 보도록 하겠다.

17.7.3.7. sched_init() 함수의 분석

sched_init() 함수는 리눅스에서 사용하는 스케줄러에 대한 초기화를 목적으로 하는 함수이다. 주로, bottom half에 대한 초기화를 담당한다.

```

void __init sched_init(void)
{
    /*
     * We have to do a little magic to get the first
     * process right in SMP mode.
     */
    int cpu = smp_processor_id();
    int nr;

    init_task.processor = cpu;
    for(nr = 0; nr < PIDHASH_SZ; nr++)
        pidhash[nr] = NULL;
    init_timervecs();
    init_bh(TIMER_BH, timer_bh);
    init_bh(TQUEUE_BH, tqueue_bh);
    init_bh(IMMEDIATE_BH, immediate_bh);
    /*
     * The boot idle thread does lazy MMU switching as well:
     */
    atomic_inc(&init_mm.mm_count);
}

```

```
    enter_lazy_tlb(&init_mm, current, cpu);
}
```

코드 1025. sched_init() 함수의 정의

smp_processor_id() 함수는 현재 사용하고 있는 CPU의 번호를 얻어오는 역할을 한다. 이렇게 얻어온 CPU번호는 cpu 변수가 나타내도록 한다. init_task는 시스템에서 제일먼저 생성되는 kernel thread이다. 이 프로세스가 사용하는 프로세서 값으로 cpu로 한다. PIDHASH_SZ는 pid를 가지고 찾는 hash 함수를 위한 것인데, 생성된 프로세스들은 hash함수에 의해서 생성된 테이블의 인덱스를 이용해서 그 위치가 정해진다. 초기에는 테이블의 전체 entry가 NULL을 가르키도록 만든다.

init_timervecs() 함수는 시스템의 타이머를 초기화 하는 함수이다. 시스템에서 사용하는 타이머 들은 전부 list의 형태로 관리되는데, 크게 연결리스트로 tv1에서 tv5까지가 여기서 초기화된다. init_bh는 bottom half에 대한 초기화를 담당하는 함수이다. 각각의 bottom half에 대해서 그것을 다루는 핸들러 함수를 등록하는 하게 된다. TIMER_BH는 timer_bh가, TQUEUE_BH는 tqueue_bh가, IMMEDIATE_BH는 immediate_bh가 담당하게 된다. 이젠 init_mm은 init kernel thread의 메모리 맵(map)을 나타낸다. 이 메모리 맵에 대해서 사용카운트(mm_count)를 하나 증가시켜주고, enter_lazy_tlb()함수를 호출해서 lazy tlb(translation lookaside buffer)를 시키도록 만드는 inline으로 정의된 함수이다. 하지만, ARM에서는 아무런 일도 하지 않는다.

계속 진행하기에 앞서 init_bh()함수를 좀더 보도록 하자. 이 함수는 앞에서 말했듯이 bottom half에 대한 초기화를 담당하며, 정의는 ~/kernel/softirq.c에 아래와 같이 되어 있다.

```
void init_bh(int nr, void (*routine)(void))
{
    bh_base[nr] = routine;
    mb();
}
```

코드 1026. init_bh() 함수의 정의

보다시피, 넘겨받은 nr값으로 bh_base[] 배열의 인덱스로 사용하고, 해당 bottom half의 핸들러로 routine을 이용한다. mb()는 매크로로 정의된 것으로 현재로서는 아무런 instruction을 제공하지 않지만, compiler에 cache를 하지 못하도록 만드는 일을 한다. 따라서, 앞에서 보았던 각각의 핸들러들이 설정됨을 이해할 수 있을 것이다.

17.7.3.8.time_init() 함수의 분석

time_init() 함수는 시스템의 timer 인터럽트를 초기화 하는 역할을 한다. 정의는 ~/arch/arm/kernel/time.c에 아래와 같이 되어 있다.

```
/*
 * This must cause the timer to start ticking.
 * It doesn't have to set the current time though
 * from an RTC - it can be done later once we have
 * some buses initialised.
 */
void __init time_init(void)
{
    xtime.tv_usec = 0;
    xtime.tv_sec   = 0;

    setup_timer();
}
```

코드 1027. time_init() 함수의 정의

xtime변수는 현재 시스템의 시간을 기록하는 일을 한다. 초(second)단위와 마이크로 초(micro second)단위의 두부분으로 나누어져 있다. setup_timer()는 다시 ~/include/asm-arm/arch-sa1100/time.h에 inline로 아래와 같이 정의되어 있다.

```
extern inline void setup_timer (void)
{
    gettimeoffset = sa1100_gettimeoffset;
    timer_irq.handler = sa1100_timer_interrupt;
    OSMR0 = 0;           /* set initial match at 0 */
    OSSR = 0xf;          /* clear status on all timers */
    setup_arm_irq(IRQ_OST0, &timer_irq);
    OIER |= OIER_E0; /* enable match on timer 0 to cause interrupts */
    OSCR = 0;           /* initialize free-running timer, force first match */
}
```

코드 1028. setup_timer() 함수의 정의

gettimeoffset변수는 현재의 time offset을 구하는 함수에 대한 포인터(sa1100_gettimeoffset())를 가진다. timer_irq.handler에는 sa1100_timer_interrupt() 함수를 두어서 timer 인터럽트에 대한 처리를 맡긴다. OSMR0(Operating System Timer Match Register 0)와 OSSR(Operating System Timer Status Register)은 시스템의 timer와 관련된 register들이다. 먼저 OSMR0에 0을 설정해서 timer match에 대한 초기화를 해주고, OSSR에는 0xF를 넣어서, 모든 타임머에 대한 상태(status)를 지워주었다. setup_arm_irq() 함수는 IRQ_OST0(=SA1100_IRQ(26)) 인터럽트 번호에 해당하는 인터럽트 핸들러를 연결지음을 앞에서 이미 보았다. OIER(Operating System Timer Interrupt Enable Register)에 OIER_E0(=0x00000001)을 OR시켜서 enable시켜준다. OSCR(Operating System Timer Counter Register)는 timer의 counter를 를 유지하는 레지스터이며, 이것은 0으로 초기화 한다.

```
static unsigned long sa1100_gettimeoffset (void)
{
    unsigned long ticks_to_match, elapsed, usec;

    /* Get ticks before next timer match */
    ticks_to_match = OSMR0 - OSCR;
    /* We need elapsed ticks since last match */
    elapsed = LATCH - ticks_to_match;
    /* Now convert them to usec */
    usec = (unsigned long)(elapsed*tick)/LATCH;
    return usec;
}
```

코드 1029. sa1100_gettimeoffset() 함수의 정의

sa1100_gettimeoffset() 함수는 microsecond단위로 time을 돌려주는 함수이다. LATCH는 매크로로 정의된 값으로 다음과 같이 제공된다. 정의는 ~/include/linux/timex.h에 있다.

```
/* LATCH is used in the interval timer and ftape setup. */
#define LATCH ((CLOCK_TICK_RATE + HZ/2) / HZ) /* For divider */

...
/* ~/kernel/timer.c에서 가져왔다.*/
long tick = (1000000 + HZ/2) / HZ; /* timer interrupt period */
```

코드 1030. LATCH의 정의

즉, LATCH는 CLOCK_TICK_RATE(=3686400)에 HZ(=100)을 반으로 나눈 값을 더하고, 다시 이를 HZ로 나눈 값이다. 따라서, LATCH는 36864.5값을 가진다.

`sa1100_gettimeoffset()` 함수에서는 OSMR0의 값에서 OSCR를 뺀 값으로 ticks_to_match를 나타내고, 이 값이 의미하는 바는 timer의 counter와 matching되는 값에서 얼마나 많은 tick이 지나갔는지를 알려준다는 것이다. 이 값을 다시 LATCH에서 빼주어 흘러간 tick을 구하고, 이를 다시 elapsed로 놓고 tick을 곱해서 timer 인터럽트 사이의 시간이 흘러간 시간을 구하게 된다. 최종적으로 이 값을 LATCH로 나누어서 우리가 원하는 microsecond 단위의 시간을 구한다.

```
static void sa1100_timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    long flags;
    int next_match;

    /* Loop until we get ahead of the free running timer.
     * This ensures an exact clock tick count and time accuracy.
     * IRQs are disabled inside the loop to ensure coherence between
     * lost_ticks (updated in do_timer()) and the match reg value, so we
     * can use do_gettimeofday() from interrupt handlers.
     */
    do {
        do_leds();
        save_flags_cli( flags );
        do_timer(regs);
        OSSR = OSSR_M0; /* Clear match on timer 0 */
        next_match = (OSMR0 += LATCH);
        restore_flags( flags );
    } while( (signed long)(next_match - OSCR) <= 0 );
}
```

코드 1031. `sa1100_timer_interrupt()` 함수의 정의

`sa1100_timer_interrupt()` 함수는 타이머 인터럽트에 대한 처리를 맡는다. 넘겨 받는 변수로는 timer 인터럽트에 대한 번호와 디바이스의 ID, 레지스터 값에 대한 포인터이다. `do_leds()` 함수는 LED(Light Emitted Device)에 대한 조절을 하는 함수이다. 단순히 끄고 켜는 일을 한다고 생각하면 될 것이다. 인터럽트의 발생을 막기 위해서 `save_flags_cli()`를 호출하고, `do_timer()`를 불러서 타이머 인터럽트를 처리한다. OSSR은 OSSR_M0fmf 넣어서 timer0에 대한 matching을 지워주고 다음번의 match가 생겨서 인터럽트가 발생하도록 `next_match`에는 OSMR0에 LATCH를 더한 값을 넣는다. 이것을 마치면 다시 인터럽트가 생기도록 `restore_flags()`를 호출한다. `do while()` loop는 `next_match`가 OSCR보다 작거나 같은 동안 계속해주도록 한다. 이렇게 해서 누적된 timer tick 인터럽트를 한번에 처리한다.

```
void do_timer(struct pt_regs *regs)
{
    (*(unsigned long *)&jiffies)++;
#ifndef CONFIG_SMP
    /* SMP process accounting uses the local APIC timer */
    update_process_times(user_mode(regs));
#endif
    mark_bh(TIMER_BH);
    if (TQ_ACTIVE(tq_timer))
        mark_bh(TQUEUE_BH);
}
```

코드 1032. `do_timer()` 함수의 정의

`do_timer()` 함수는 타이머 인터럽트의 처리를 맡고 있다. 정의는 `~/kernel/timer.c`에 나오며, 앞에서 이미 한번 본 적이 있는 함수이다. 이 함수가 하는 역할은 단순히 jiffies값을 증가시키고, process의 시간 사용량을 증가시키며, TIMER_BH를 설정하며, timer와 관련된 queue(tq_timer)가 active 상태로 있는 task가 있다면, 이를 수행하기 위해서 TQUEUE_BH도 bottom half를 설정하도록 한다(`mark_bh()`). 이는 나중에

software interrupt를 처리하는 부분에서 하나씩 처리가 일어날 것이다. 간단히 이러한 일만을 하고 다시 인터럽트 핸들러는 복귀하게된다. 너무 많은 인터럽트 처리의 overhead는 시스템의 성능을 약화시키는 일을 하므로, bottom half의 사용은 중요한 역할을 한다.

17.7.3.9. softirq_init() 함수의 분석

softirq_init()함수는 software적으로 커널내에서 발생시킨 모든 인터럽트에 대한 초기화를 담당하는 함수이다. 이를 처리하기 위한 자료구조를 만드는 것을 중점적으로 처리해준다.

```
void __init softirq_init()
{
    int i;

    for (i=0; i<32; i++)
        tasklet_init(bh_task_vec+i, bh_action, i);
    open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
    open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
}
```

코드 1033. softirq_init() 함수의 정의

예전 버전과는 달리 커널 버전 2.4.X에서 부터는 bottom half라는 개념이 tasklet(작은 태스크)이라는 개념으로 달리 사용되고 있다. softirq_init()함수의 중요한 역할은 바로 이러한 tasklet에 대한 초기화를 담당한다는 것이다. 먼저 전체 32개의 tasklet을 나타내는 벡터(bh_task_vec)에 대한 처리를 맡고 있는 핸들러(bh_action)을 초기화하는 tasklet_init()함수를 호출한다. 정의는 아래와 같다.

```
void tasklet_init(struct tasklet_struct *t,
                  void (*func)(unsigned long), unsigned long data)
{
    t->func = func;
    t->data = data;
    t->state = 0;
    atomic_set(&t->count, 0);
}
```

코드 1034. tasklet_init() 함수의 정의

tasklet_init()함수는 넘겨받은 tasklet_struct구조체의 func필드에는 넘겨받은 action을 연결시켜주고, data에는 다시 넘겨받은 데이터 값을(여기서는 index값이 될 것이다.), 또한 상태와 count는 각각 0으로 초기화 시켰다. bh_action()함수는 다시 아래와 같이 정의된다.

```
spinlock_t global_bh_lock = SPIN_LOCK_UNLOCKED;

static void bh_action(unsigned long nr)
{
    int cpu = smp_processor_id();

    if (!spin_trylock(&global_bh_lock))
        goto resched;
    if (!hardirq_trylock(cpu))
        goto resched_unlock;
    if (bh_base[nr])
        bh_base[nr]();
    hardirq_endlock(cpu);
    spin_unlock(&global_bh_lock);
    return;
resched_unlock:
```

```

    spin_unlock(&global_bh_lock);
resched:
    mark_bh(nr);
}

```

코드 1035. bh_action() 함수의 정의

bh_action()함수는 현재 코드를 수행중인 CPU에 대한 ID를 읽어서(smp_processor_id()), 이를 cpu에 저장한다. 전역 bottom half에 대한 lock을 설정하기 위해서 spin_trylock()을 호출하고, 만약 얻지 못한다면, 다시 bottom half를 수행하고 표시한다(mark_bh()). 또한, 현재의 수행중인 CPU에 대한 lock도 구하기 위해서 hardirq_trylock()을 호출한다. 이것이 실패하면, 이전에 설정한 lock을 해제하고 bottom half를 marking한 후 복귀한다. 여기까지 진행했다면, 이젠 드디어 관련된 action을 수행할 차례이다. 단지 전역 변수 bh_base[] 배열의 nr번째 있는 함수를 수행해 주면 될 것이다. bottom half의 수행이 끝났다면, 앞에서 얻었던 lock들을 해제하고 복귀하면 될 것이다.

```

static spinlock_t softirq_mask_lock = SPIN_LOCK_UNLOCKED;

void open_softirq(int nr, void (*action)(struct softirq_action*), void *data)
{
    unsigned long flags;
    int i;

    spin_lock_irqsave(&softirq_mask_lock, flags);
    softirq_vec[nr].data = data;
    softirq_vec[nr].action = action;
    for (i=0; i<NR_CPUS; i++)
        softirq_mask(i) |= (1<<nr);
    spin_unlock_irqrestore(&softirq_mask_lock, flags);
}

```

코드 1036. open_softirq() 함수의 정의

open_softirq()함수는 software IRQ에 대한 vector를 초기화 하는 일을 한다. 즉, 앞에서 본 tasklet(or bottom half)의 action에 대한 vector를 가진다. 즉, 이 vector가 하나 수행되면, 관련된 tasklet들이 수행되며, 각각의 tasklet에 있는 action들이 전부 수행될 기회를 가지게 되는것이다. 일종의 이차원 배열 형식으로 생각하면 될 것이다. 다만, 배열의 원소가 action들로 연결되어서 반응이 일어난다는 점이 다르다.

먼저 IRQ가 걸리지 않도록 lock을 설정하고(spin_lock_irqsave()), softirq_vec[] 배열의 data필드와 action필드에 넘겨받은 action과 데이터를 각각에 맞게 넣어준다. softirq_mask() 매크로는 현재 CPU의 software IRQ mask를 가지는 값으로, 전 CPU들에 대해서 이 software IRQ가 허락된다는 것을 bit으로 나타내준다. 이것을 마치면, 앞에서 설정한 IRQ lock을 해제한다(spin_unlock_irqrestore()).

앞에서 softirq_init()에서는 TASKLET_SOFTIRQ와 HI_SOFTIRQ에 대해서 software IRQ를 설정해 주었다. 참고로 이 둘에 대한 정의는 ~/include/linux/interrupt.h에 아래와 같이 나와 있다.

```

enum
{
    HI_SOFTIRQ=0,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    TASKLET_SOFTIRQ
};

```

코드 1037. software IRQ의 종류

HI_SOFTIRQ를 0으로 NET_TX_SOFTIRQ와 NET_RX_SOFTIRQ, TASKLET_SOFTIRQ의 순서로 각각의 software IRQ들의 값이 정해진다. 아직 network에 대해서는 초기화가 되지 않았기에, 중간에 있는

NET_XXX_SOFTIRQ에 대해서는 open_softirq() 함수를 사용하지 않았다. 따라서, 이곳에서 유추해 볼 수 있는 것은 일반적인 bottom half와 관련된 TASKLET_SOFTIRQ보다는 NET에 관련된 것이 더 우선 순위가 높다는 것을 알 수 있을 것이다. 따라서, 네트워크 디바이스 드라이버와 같은 것을 구현할 때, network bottom half에 대한 처리보다는 TX와 RX에 대한 처리가 먼저 일어나게 될 것이다.

17.7.3.10.console_init() 함수의 분석

이곳에서 console에 대한 초기화를 하는 것은 아직 bus에 대한 초기화를 하지 않았기에 위험할 수도 있다. 하지만, 시스템으로부터 어떤 메시지를 출력 받기를 원한다면, 이곳에서 console 대해서 초기화를 해주어야 할 것이다. 이것은 디버깅(debugging)을 위한 것일 수도 있으며, 단순히 시스템의 정보를 출력하기 위한 것일 수도 있을 것이다. 정의는 ~/drivers/char/tty_io.c에 아래와 같이 되어 있다.

```
/*
 * Initialize the console device. This is called *early*, so
 * we can't necessarily depend on lots of kernel help here.
 * Just do some early initializations, and do the complex setup
 * later.
 */
void __init console_init(void)
{
    /* Setup the default TTY line discipline. */
    memset(ldiscs, 0, sizeof(ldiscs));
    (void) tty_register_ldisc(N_TTY, &tty_ldisc_N_TTY);
    /*
     * Set up the standard termios. Individual tty drivers may
     * deviate from this; this is used as a template.
     */
    memset(&tty_std_termios, 0, sizeof(struct termios));
    memcpy(tty_std_termios.c_cc, INIT_C_CC, NCCS);
    tty_std_termios.c_iflag = ICRNL | IXON;
    tty_std_termios.c_oflag = OPOST | ONLCR;
    tty_std_termios.c_cflag = B38400 | CS8 | CREAD | HUPCL;
    tty_std_termios.c_lflag = ISIG | ICANON | ECHO | ECHOE | ECHOK |
        ECHOCTL | ECHOKE | IEXTEN;
```

코드 1038. console_init() 함수의 분석

console_init() 함수는 크게 두 가지 부분으로 나누어진다. 먼저 첫 번째 부분이 TTY line discipline에 대한 설정이 있고, 두 번째 부분이 console device에 대한 초기화이다. 그럼 먼저 첫 부분부터 보기로 하자. ldiscs 변수는 tty_ldisc 구조체로 정의된 변수로서 아래와 같이 정의된다.

```
/*~/include/linux/tty_ldisc.h에서 */
struct tty_ldisc {
    int      magic;           /* magic number : TTY_LDISC_MAGIC */
    char    *name;            /* name */
    int      num;             /* 번호 : Line Identifier */
    int      flags;           /* 상태 flag : Line Type - LDISC_FLAG_DEFINED */
    /*
     * The following routines are called from above.
     */
    int      (*open)(struct tty_struct *);          /* TTY에 대한 open함수 */
    void    (*close)(struct tty_struct *);           /* TTY에 대한 close함수 */
    void    (*flush_buffer)(struct tty_struct *tty); /* TTY에 대한 buffer의 flush함수 */
    ssize_t  (*chars_in_buffer)(struct tty_struct *tty); /* 특정 character를 buffer에서 찾는 함수 */
    ssize_t  (*read)(struct tty_struct * tty, struct file * file,
                    unsigned char * buf, size_t nr);      /* TTY에 대한 read함수 */
```

```

ssize_t (*write)(struct tty_struct * tty, struct file * file,
                 const unsigned char * buf, size_t nr); /* TTY에 대한 write함수 */
int     (*ioctl)(struct tty_struct * tty, struct file * file,
                 unsigned int cmd, unsigned long arg); /* TTY에 대한 ioctl함수 */
void    (*set_termios)(struct tty_struct *tty, struct termios * old); /* TTY의 terminal I/O structure를 설정
                                                                     하는 함수 */
unsigned int (*poll)(struct tty_struct * , struct file * ,
                     struct poll_table_struct *); /* TTY에 대한 polling함수 */

/*
 * The following routines are called from below.
 */
void    (*receive_buf)(struct tty_struct *, const unsigned char *cp,
                      char *fp, int count); /* 받은 데이터 buffer에 대한 포인터를 전달한다.
*/
int     (*receive_room)(struct tty_struct *); /* Input buffer에 있는 character들의 수를 구한다. */
void    (*write_wakeup)(struct tty_struct *); /* Output buffer에 있는 character들의 수를 구한다.
*/
};

/* ~/include/linux/tty_io.c에서 */
struct tty_ldisc ldiscs[NR_LDISCS]; /* line discipline dispatch table */

```

코드 1039. tty_ldisc구조체의 정의 및 ldiscs 변수의 정의

tty_ldisc의 정의 중에서 comment로 들어간 부분을 보면 above(상위)에서 호출된다와 below(하위)에서 불린다로 추정 할 수 있는 것은 tty_ldisc구조체가 어떤 시스템의 구성요소들 간의 연결하는 역할을 한다는 것을 알 수 있다. 이것은 실제로 리눅스에서 터미널을 구현하는 곳에서 사용되며, 하위의 드라이버를 연결한다던가 아니면, 상위에 가상 터미널과 같은 것을 둘 경우에 상요될 수 있을 것이다.

먼저 ldisc전역 변수를 전부 0으로 놓는다(memset()). 그리고나서, tty_register_ldisc() 함수를 호출해서 하나의 ldisc로 tty_ldisc_N_TTY를 등록시켜준다. tty_register_ldisc() 함수는 아래와 같이 정의되는 함수이다.

```

int tty_register_ldisc(int disc, struct tty_ldisc *new_ldisc)
{
    if (disc < N_TTY || disc >= NR_LDISCS)
        return -EINVAL;

    if (new_ldisc) {
        ldiscs[disc] = *new_ldisc;
        ldiscs[disc].flags |= LDISC_FLAG_DEFINED;
        ldiscs[disc].num = disc;
    } else
        memset(&ldiscs[disc], 0, sizeof(struct tty_ldisc));
    return 0;
}

```

코드 1040. tty_register_ldisc() 함수의 정의

tty_register_ldisc() 함수는 새로운 하나의 ldisc구조체를 가리키는 포인터를 넘겨받아서, 이것으로 전역 변수 ldiscs[] 배열의 한 entry를 만들어주는 역할을 한다. NULL pointer를 넘겨준다면, 해당하는 인덱스의 element값이 0로 채워질 것이다. 만약 넘겨받은 ldiscs[] 배열에 대한 index가 범위를 벗어나게 되면, -EINVAL이 복귀값이 될 것이다.

console_init() 함수에서는 tty_register_ldisc()에 대한 호출 파라미터로 N_TTY와 tty_ldisc_N_TTY를 주었다. N_TTY는 터미널에서 사용할 line discipline의 종류를 나타내는 것으로 0에서 14까지가 정해져 있다.

```

/* ~/include/asm-arm/termios.h에서 */
/* line disciplines */
#define N_TTY          0
#define N_SLIP         1
#define N_MOUSE2       2
#define N_PPP          3
#define N_STRIP        4
#define N_AX25         5
#define N_X25          6      /* X.25 async */
#define N_6PACK        7
#define N_MASC         8      /* Reserved for Mobitex module <kaz@cafe.net> */
#define N_R3964        9      /* Reserved for Simatic R3964 module */
#define N_PROFIBUS_FDL 10     /* Reserved for Profibus <Dave@mvhi.com> */
#define N_IRDA         11     /* Linux IrDa - http://irda.sourceforge.net/ */
#define N_SMSBLOCK     12     /* SMS block mode - for talking to GSM data cards about SMS messages */
#define N_HDLC          13     /* synchronous HDLC */
#define N_SYNC_PPP     14

...
/* ~/drivers/char/n_tty.c에서 */
struct tty_ldisc n_tty_ldisc_N_TTY = {
    TTY_LDISC_MAGIC,           /* magic */
    "n_tty",                  /* name */
    0,                        /* num */
    0,                        /* flags */
    n_tty_open,               /* open */
    n_tty_close,              /* close */
    n_tty_flush_buffer,       /* flush_buffer */
    n_tty_chars_in_buffer,    /* chars_in_buffer */
    read_chan,                /* read */
    write_chan,               /* write */
    n_tty_ioctl,              /* ioctl */
    n_tty_set_termios,        /* set_termios */
    normal_poll,              /* poll */
    n_tty_receive_buf,        /* receive_buf */
    n_tty_receive_room,       /* receive_room */
    0                         /* write_wakeup */
};

```

코드 1041. tty_ldisc_N_TTY 구조체의 정의

코드를 보면 각각의 N_XXX가 뜻하는 바를 어느정도는 이해 할 수 있을 것이다. 대표적인 것으로 N_TTY는 console을 나타내며, N_SLIP는 SLIP line에, N_MOUSE는 mouse에, N_PPP는 PPP line에, N_STRIP는 Starmode Radio IP에 각각 해당한다. tty_ldisc_N_TTY는 N_TTY에 대한 ldisc구조체를 정의한다. TTY_LDISC_MAGIC을 magic number로 가지며, 이름은 n_tty이다. 각각의 필드가 의미하는 바는 앞에서 이미 보아서 알 것이다. tty_ldisc_N_TTY 구조체에서 나머지 함수들에 대한 것은 분석하지 않기로 하겠다. 다만 나중에 tty_ldis_N_TTY를 커널에서 사용하기 위해서 호출하는 함수라는 사실만을 기억하기 바란다.

console_init() 함수는 이제 표준(standard) termios 구조체를 설정한다. 이 표준 termios구조체는 template과 같은 역할을 하며, 개개의 tty driver들은 자신만의 고유한 특성을 가질 것이다. 먼저 표준 termios구조체를 나타내는 tty_std_termios를 0으로 초기화 한다. 그리고나서, 이 구조체의 c_cc필드에 INIT_C_CC를 NCCS만큼을 복사해서 넣는다. INIT_C_CC는 초기 제어 문자들을 가지는 변수로서 그 길이가 NCCS이다. 일단 termios구조체부터 먼저 보기로 하자. 정의는 ~/include/asm-arm/termbits.h에 아래와 같이 되어 있다.

```
#define NCCS 19
```

```
struct termios {
    tcflag_t c_iflag;          /* input mode flags */
    tcflag_t c_oflag;          /* output mode flags */
    tcflag_t c_cflag;          /* control mode flags */
    tcflag_t c_lflag;          /* local mode flags */
    cc_t c_line;               /* line discipline or line protocol */
    cc_t c_cc[NCCS];           /* control characters */
};
```

코드 1042. termios 구조체의 정의

POSIX⁴¹⁰ 터미널의 설정은 일반적으로 명확하지 않다. 다양한 많은 것들이 존재하기 때문이며, 전송하는 데이터 레이트(rate)에서부터 사용하는 코딩 방식이 각각 다 다르기 때문이다. 이러한 다양한 기능들을 사용가능하게 하기 위해서 POSIX 터미널의 구현에서 사용되는 명령이나, 구조체들, 그리고, 상수값들을 termios.h에 정의해 두고 있다. 이 파일은 다시 architecture에 의존적인 파일들로 ~/include/asm/termios.h와 ~/include/asm/termbits.h 파일을 포함하고 있다. 터미널에 대한 설정은 termios구조체를 사용해서 할 수 있다. 이 구조체는 또한 설정 사항에 대한 검사와 파라미터 값들에 변동이 있는지도 검사하는데도 사용될 수 있다.

이야기를 더 진행하기에 앞서 이곳에서 간단히 어떻게 통신이 일어나는지를 살펴보기로 하자. 주로 살펴보는 것은 serial과 같은 디바이스가 어떻게 통신에 사용되는지를 보는 것이다. 프로세스가 POSIX 터미널과 데이터를 주고 받기 위해서는 두개의 버퍼를 사용한다.

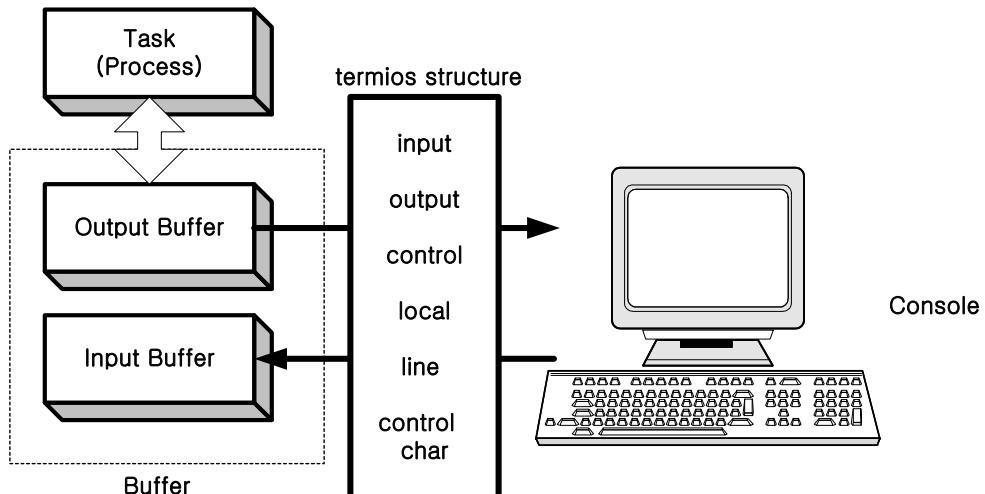


그림 132. termios 구조체의 사용 방식

데이터를 보내는 것과 받는 것은 buffer를 사용함으로써 더 빠르게 일어날 수 있으며(즉, buffer의 사용으로 효율이 높아지는 것이다.) 또한 전달하고자 하는 문자들은 buffer와 터미널 사이에서 터미널의 특성에 따라서 변형(transformation)될 수 있다. 즉, 데이터를 전달하고자 할 때, 상대방이 알아들을 수 있도록 만드는 과정이 더해지게 되는 것이다.

이젠 앞으로 다시 돌아가서, tty_std_termios구조체의 flag들에 대한 설정을 보기로 하자. 먼저 c_iflag는 input mode를 나타내는 flag으로 ICRNL과 IXON의 OR값으로 결정되었다. ICRNL은 IGNR⁴¹¹이 설정되지 않았다면, CR(carriage return) 문자를 받으면 NL(new line)으로 변환하라는 것이다. IXON은 output에 XON/OFF flow control를 활성화(activate)시키라는 말이다. c_oflag은 OPOST와 ONLCR을 OR시켜서 설정했다. OPOST는 문자들에 대한 처리(processing)을 활성화(activate)시키라는 뜻이며, ONLCR은 CR문자를 column 0에 나타내지는 말라는 의미이다. c_cflag에 설정된 B38400과 CS8, CREAD, HUPCL을

⁴¹⁰ UNIX와 같은 운영체제에 대한 표준(standard)이다.

⁴¹¹ Ignore the CR character.

OR시켜주었는데, B38400은 BSDLY⁴¹²를 위한 매스크 값이며, CS8은 CSIZE⁴¹³에 대한 매스크 값이고, CREAD는 receiver를 활성화 시켜주며, HUPCL은 제어되고 있는 마지막 프로세스가 끝나면 자동적으로 연결을 끊어주라는 뜻이다. c_lflag에는 ISIG, ICANON, ECHO, ECHOE, ECHOK, ECHOCTL, ECHOKE, IEXTEN을 OR시켜주었는데, 각각이 하는 역할은 아래의 표와 같다⁴¹⁴.

이름	설명
ISIG	INTR, QUIT, SUSP(혹은 DSUSP)와 같은 문자를 받으며, SIGINT, SIGQUIT, SIGTSP와 같은 시그널이 각각 전달된다.
ICANON	Canonic mode를 활성화 시킨다. 즉, 터미널을 interactive mode로 사용한다.
ECHO	타이핑된 문자들을 보이도록 만든다.
ECHOE	만약 ICANON이 설정되었다면, ERASE 문자가 앞에 있는 문자를 지우며, WERASE문자는 앞에 있는 word를 지운다.
ECHOK	만약 ICANON이 설정되었다면, KILL 문자는 현재의 라인을 지운다.
ECHOCTL	만약 IEXTEN일 설정되었다면, TAB, NL, START, STOP, CR, BS와 다른 모든 control 문자들은 자신이 가진 ASCII 값에 8진수로 100을 더한 값을 가진다.
ECHOKE	만약 ICANON이 설정되었다면, KILL문자가 ECHOE와 ECHOPRT에 의해서 명시된 것과 같이 라인의 각 문자들을 지우기 위해서 전달된다.
IEXTEN	장치(device)의 구현(implementation)에 의해서 정의된 처리 시스템을 활성화 시킨다.

표 116. local mode의 설정 요약

여기서 보이는 모든 것들을 이해할 필요는 없다. 단지 중요한 것은 일반적으로 사용할 표준 termios 구조체의 input, output, control, local mode 필드를 설정해 준다는 사실만 이해하면 될 것이다.

```
#ifdef CONFIG_VT
    con_init();
#endif
#ifndef CONFIG_SERIAL_CONSOLE
#if (defined(CONFIG_8xx) || defined(CONFIG_8260))
    console_8xx_init();
#elif defined(CONFIG SERIAL)
    serial_console_init();
#endif /* CONFIG_8xx */
#endif CONFIG_SGI_SERIAL
    sgi_serial_console_init();
#endif
#if defined(CONFIG_MVME162_SCC) || defined(CONFIG_BVME6000_SCC) || defined(CONFIG_MVME147_SCC)
    vme_scc_console_init();
#endif
#if defined(CONFIG SERIAL167)
    serial167_console_init();
#endif
#if defined(CONFIG_SH_SCI)
    sci_console_init();
#endif
#endif
#endif CONFIG_3215
    con3215_init();
#endif
```

⁴¹² Backspace 문자후에 delay를 주라.

⁴¹³ 문자 코딩(character coding)의 크기를 나타낸다.

⁴¹⁴ 물론 이것만이 정의된 것이 아니라, 일단 이곳에서는 필요한 것만 보도록 하겠다. 나중에 터미널에 대한 이야기를 하면, 그곳에서 좀더 자세히 보도록 할 것이다.

```
#ifdef CONFIG_HWC
    hwc_console_init();
#endif
#ifndef CONFIG_SERIAL_21285_CONSOLE
    rs285_console_init();
#endif
#ifndef CONFIG_SERIAL_SA1100_CONSOLE
    sa1100_rs_console_init();
#endif
#ifndef CONFIG_SERIAL_AMBA_CONSOLE
    ambauart_console_init();
#endif
#endif
}
```

코드 1043. console_init() 함수의 정의

이제 남은 것은 커널의 컴파일 시에 설정된 값에 따라서, console device로 사용되는 것을 초기화 시켜주는 부분이다. 이 설정을 마치고 나면, 디버깅 정보를 console을 통해서 볼 수 있다. 이중에서 우리가 보고자 하는 것은 SA1100 보드에서 사용하는 serial console에 대한 초기화를 담당하는 함수로 커널 설정에서 CONFIG_SERIAL_SA1100_CONSOLE가 있을 경우에 컴파일되는 sa1100_rs_console_init() 함수이다. 정의는 ~/drivers/char/serial_sa1100.c에 아래와 같이 나와 있다.

```
/* ~/include/linux/console.h에서 */
struct console
{
    char      name[8];
    void     (*write)(struct console *, const char *, unsigned);
    int      (*read)(struct console *, const char *, unsigned);
    kdev_t   (*device)(struct console *);
    int      (*wait_key)(struct console *);
    void     (*unblank)(void);
    int      (*setup)(struct console *, char *);
    short    flags;
    short    index;
    int      cflag;
    struct   console *next;
};

...
/* ~/drivers/char/serial_sa1100.c에서 */
static struct console sercons = {
    name:          TTY_NAME,           /* tty의 이름 */
    write:         serial_console_write, /* write에 사용하는 함수 */
    device:        serial_console_device, /* 해당 device에 대한 포인터를 돌려주는 함수 */
    wait_key:     serial_console_wait_key, /* key의 입력을 기다리는 함수 */
    setup:         serial_console_setup, /* serial console를 설정(setup)하는 함수 */
    flags:         CON_PRINTBUFFER,   /* 현재 serial console의 상태를 나타내는 flag */
    index:        -1                  /* console의 초기 index값: 64에서 시작해서 빼나감 */
};

...
/*
 * Register console.
 */
void __init sa1100_rs_console_init(void)
{
    register_console(&sercons);
}
```

코드 1044. console구조체의 정의 및 sa1100_rs_console_init() 함수의 정의

sa1100_rs_console_init() 함수는 하나의 console을 등록시켜주는 역할을 한다. 등록에 사용하는 함수는 register_console()이며, 등록되는 console은 sercons이다. sercons는 console구조체를 가지고 있으며, 이 구조체는 ~/include/linux/console.h에 정의되어 있다.

register_console() 함수를 보도록 하자. 정의는 ~/kernel/printk.c에 아래와 같이 나와 있다. Console driver는 커널을 초기화 하는 동안에 이 함수를 호출해서, printk() 함수와 함께 console을 통한 printing을 한다. 또는 console driver가 초기화 되기 전에 커널에 의해서 print된 메시지를 printing하기 위해서 사용할 수도 있다.

```
void register_console(struct console * console)
{
    int      i, j, len;
    int      p;
    char     buf[16];
    signed char msg_level = -1;
    char     *q;
    unsigned long flags;

    /*
     *      See if we want to use this console driver. If we
     *      didn't select a console we take the first one
     *      that registers here.
     */
    if (preferred_console < 0) {
        if (console->index < 0)
            console->index = 0;
        if (console->setup == NULL ||
            console->setup(console, NULL) == 0) {
            console->flags |= CON_ENABLED | CON_CONSDEV;
            preferred_console = 0;
        }
    }
    /*
     *      See if this console matches one we selected on
     *      the command line.
     */
    for(i = 0; i < MAX_CMDLINECONSOLES && console_cmdline[i].name[0]; i++) {
        if (strcmp(console_cmdline[i].name, console->name) != 0)
            continue;
        if (console->index >= 0 &&
            console->index != console_cmdline[i].index)
            continue;
        if (console->index < 0)
            console->index = console_cmdline[i].index;
        if (console->setup &&
            console->setup(console, console_cmdline[i].options) != 0)
            break;
        console->flags |= CON_ENABLED;
        console->index = console_cmdline[i].index;
        if (i == preferred_console)
            console->flags |= CON_CONSDEV;
        break;
    }
    if (!(console->flags & CON_ENABLED))
        return;
}
```

코드 1045. register_console() 함수의 정의

preferred_console이 0보다 작다면, 아직 아무런 console이 없다는 말이된다. 또한 넘겨받은 console 구조체의 index값이 0보다 작다면, 초기화가 되지 않은 것이다. console구조체의 setup필드가 NULL이거나, console구조체의 setup필드에 있는 함수를 수행한 결과가 0이라면, console구조체의 flags필드에 console이 enable되었음을 나타내고, device와 연결되어 있음을 나타낸다. 여기서 주의할 점은 if절에서 OR가 될 경우에는 첫번째 expression을 먼저보고, 이것이 참이 아닐 경우에 두번째 것이 수행된다는 것이다. 따라서, NULL이 아니라면, console->setup()이 수행될 것이다. 이것이 또한 0값을 return하지 않는다면, console은 enable되지 않는다.

이전 command line에서 전달한 console인지를 확인하는 절차이다. 즉, command line에서 주어진 console에 대한 option을 반영하려는 것이다. 이것은 for loop를 돌면서 console_cmdline[] 배열의 이름을 살펴보는 것이다. console_cmdline[]배열은 console_cmdline구조체로 정의되어 있다. 필드로는 이름을 나타내는 name과 사용하게될 minor device 번호를 나타내는 index, driver를 위한 option을 기록하는 character pointer인 option필드로 구성된다. 먼저 이름이 같은지를 비교한다. 같지 않다면, for loop를 계속 돈다. 그리고나서, console의 index를 비교한다. 역시 같지 않다면, 계속 다음 loop를 진행하고, 만약 console의 index값이 0보다 작다면, console구조체의 index값을 command line에서 받은 index값으로 정한다. 만약 설정함수가 있다면, 이를 수행한다(setup()). 이때 설정이 실패한다면 loop를 빠져나온다. console구조체의 flags필드의 값은 CON_ENABLED가 되고, index필드는 command line에서 주어진 것을 사용하며, 만약 command line에서 받은 console에 대해서 preferred_console이 일치하는 값이 있다면(preferred_console), 이를 device와 관련지어준다(flags |= CON_CONDEV). 여기까지 진행했다면, 더이상 진행할 이유가 없으므로 loop를 빠져나온다. console구조체의 flags에 CON_ENABLE가 설정되지 않았다면, 해당하는 console에 대한 설정에 문제가 있다는 말이되기에 바로 return한다⁴¹⁵.

```

/*
 *      Put this console in the list - keep the
 *      preferred driver at the head of the list.
 */
spin_lock_irqsave(&console_lock, flags);
if ((console->flags & CON_CONSDEV) || console_drivers == NULL) {
    console->next = console_drivers;
    console_drivers = console;
} else {
    console->next = console_drivers->next;
    console_drivers->next = console;
}
if ((console->flags & CON_PRINTBUFFER) == 0)
    goto done;
/*
 *      Print out buffered log messages.
 */
p = log_start & LOG_BUF_MASK;
for (i=0,j=0; i < log_size; i++) {
    buf[j++] = log_buf[p];
    p = (p+1) & LOG_BUF_MASK;
    if (buf[j-1] != '\n' && i < log_size - 1 && j < sizeof(buf)-1)
        continue;
    buf[j] = 0;
    q = buf;
    len = j;
    if (msg_level < 0) {
        if(buf[0] == '<' &&
           buf[1] >= '0' &&

```

⁴¹⁵ 실제로 정의된 것을 보면, console_drivers구조체는 console구조체에 대한 포인터이다. 따라서, 사용중인 console 구조체를 console_driver가 나타낸다고 보면 될 것이다.

```

        buf[1] <= '7' &&
        buf[2] == '>') {
            msg_level = buf[1] - '0';
            q = buf + 3;
            len -= 3;
        } else
        {
            msg_level = default_message_loglevel;
        }
    }
    if (msg_level < console_loglevel)
        console->write(console, q, len);
    if (buf[j-1] == '\n')
        msg_level = -1;
    j = 0;
}
done:
    spin_unlock_irqrestore(&console_lock, flags);
}

```

코드 1046. register_console() 함수의 정의(계속)

이전 연결 리스트에 대한 연산을 수행해야 하므로 lock을 설정한다(spin_lock_irqsave()). console구조체의 flags필드가 CON_CONDEV로 설정되어 있거나, console_drivers가 NULL을 가진다면, console로 사용할 드라이버를 설정하지 않은 것이므로 console구조체를 console의 연결리스트에 넣어주고, console구조체로 console_drivers를 초기화 시킨다. 그렇지 않은 경우라면, 이미 console드라이버가 있는 경우므로, 단지 console구조체의 연결리스트에 console_drivers구조체의 next필드를 넣어주고, console_driver구조체의 next필드에 넘겨받은 console구조체를 넣어준다. 이렇게 함으로써, console구조체의 연결 리스트와 console_drivers구조체의 연결 리스트 간의 연결고리를 만들어준다.

만약 console 구조체의 flags필드에 CON_PRINTBUFFER가 설정된 경우에는 곧바로 done으로 제어를 옮기고, 그렇지 않을 경우에는 프린트할 메시지가 있다는 말이므로 해당 console에 대한 write 연산을 실행한다. 먼저 프린트할 메시지의 시작번지를 구한다(log_start). buf는 프린트할 내용을 담은 변수이며, 그 시작은 q로 나타낸다. 여기서 중요한 점은 log를 가지는 buffer가 환형(circular) buffer로 구성된다는 점이며, 메시지에는 log level을 나타내는 것이 들어있을 수 있다는 것이다. log level이 없다면, 단순히 default_message_loglevel로 msg_level을 설정한다. 만약 msg_level이 console_loglevel보다 작다면, console구조체의 write연산을 수행한다. buf의 마지막에는 '\n'을 넣어서 다음 라인(line)의 시작을 나타내주고, msg_level은 한번 출력력을 마쳤으므로 -1값을 주어서 다시 초기화 시킨다. 이 과정을 마치면, 앞에서 설정했던 lock을 해제한다(spin_unlock_irqrestore()).

참고적으로 등록된 console을 제거하는 함수는 unregister_console()이다. 정의는 아래와 같다. 이 함수는 단순히 console을 나타내는 구조체의 연결 리스트에서 해당하는 console구조체를 삭제하는 일을 한다.

```

int unregister_console(struct console * console)
{
    struct console *a,*b;
    unsigned long flags;
    int res = 1;

    spin_lock_irqsave(&console_lock, flags);
    if (console_drivers == console) {
        console_drivers=console->next;
        res = 0;
    } else
    {
        for (a=console_drivers->next, b=console_drivers ;
             a; b=a, a=b->next) {

```

```

        if (a == console) {
            b->next = a->next;
            res = 0;
            break;
        }
    }

/* If last console is removed, we re-enable picking the first
 * one that gets registered. Without that, pmac early boot console
 * would prevent fbcon from taking over.
 */
if (console_drivers == NULL)
    preferred_console = -1;
spin_unlock_irqrestore(&console_lock, flags);
return res;
}

```

코드 1047. unregister_console() 함수의 정의

해당 console 구조체를 빼어내기 위해서는 일단 lock을 설정해야 할 것이다(spin_lock_irqsave()). 만약 현재의 console_drivers가 넘겨받은 console과 같다면, console_drivers는 console의 next필드로 다시 설정해 준다. 복귀 값은 0이 될 것이다.

만약 console_drivers와 넘겨받은 console구조체가 같지 않다면, console 구조체의 연결리스트를 뒤져서 이를 제거해 주어야 할 것이다. 즉, console_drivers에서 시작해서 next필드를 찾아나가면 될 것이다. 만약 이렇게 저거해 나가다가, console_drivers가 NULL이 될 경우에는 preferred_console에 -1을 주어서 현재 console로 아무것도 없음을 나타낸다. 마지막은 당연히 앞에서 설정한 lock을 해제하는 것이다(spin_unlock_irqrestore()). 복귀값은 앞에서 설정된 값이 될 것이다.

17.7.3.11. printk() 함수의 분석

console에 대한 설정을 마치고 나면, 이젠 드디어 시스템의 각종 정보를 printk() 함수를 통해서 볼 수 있게 되었다. printk() 함수는 ~/kernel/printk.c에 아래와 같이 정의되어 있다.

```

static char buf[1024];
...
asmlinkage int printk(const char *fmt, ...)
{
    va_list args;
    int i;
    char *msg, *p, *buf_end;
    int line_feed;
    static signed char msg_level = -1;
    long flags;

    spin_lock_irqsave(&console_lock, flags);
    va_start(args, fmt);
    i = vsprintf(buf + 3, fmt, args); /* hopefully i < sizeof(buf)-4 */
    buf_end = buf + 3 + i;
    va_end(args);
}

```

코드 1048. printk() 함수의 정의

printk()함수가 넘겨받는 파라미터 값은 format을 가지는 string과 그 format에 맞춰서 나타낼 값들이다. 이곳에서 중요한 것은 msg_level인데, 각각의 메시지 레벨에 따라서, 출력할 것인가를 알려주는 값이다. 나중에 이 값을 현재 console의 log level과 비교한다. 먼저 console에 대한 출력을 관리하므로, spin_lock_irqsave()를 호출해서 console_lock을 얻는다. va_start() 매크로는 variable argument에 대한 처리를

담당하는 매크로로서 args가 fmt를 가르키는 포인터뒤를 가르키도록 만들어준다⁴¹⁶. 이전 vsprintf()를 사용해서 buf에 fmt포맷으로 args를 프린트 해 준다. buf_end는 버퍼의 마지막으로 가르키는 것으로 앞에서 프린트한 곳의 끝부분을 나타내도록 만들어준다.

```

for (p = buf + 3; p < buf_end; p++) {
    msg = p;
    if (msg_level < 0) {
        if (p[0] != '<' || p[1] < '0' || p[1] > '7' || p[2] != '>') {
            p -= 3;
            p[0] = '<';
            p[1] = default_message_loglevel + '0';
            p[2] = '>';
        } else
            msg += 3;
        msg_level = p[1] - '0';
    }
    line_feed = 0;
    for (; p < buf_end; p++) {
        log_buf[(log_start+log_size) & LOG_BUF_MASK] = *p;
        if (log_size < LOG_BUF_LEN)
            log_size++;
        else
            log_start++;
        logged_chars++;
        if (*p == '\n') {
            line_feed = 1;
            break;
        }
    }
    if (msg_level < console_loglevel && console_drivers) {
        struct console *c = console_drivers;
        while(c) {
            if ((c->flags & CON_ENABLED) && c->write)
                c->write(c, msg, p - msg + line_feed);
            c = c->next;
        }
    }
    if (line_feed)
        msg_level = -1;
}
spin_unlock_irqrestore(&console_lock, flags);
wake_up_interruptible(&log_wait);
return i;
}

```

코드 1049. printk() 함수의 정의(계속)

이젠 buf에 들어있는 내용을 추적하는 것이다. 먼저 메시지의 log레벨을 알려고 한다. 만약 정해진 log레벨이 없다면, default message log level을 적용하도록 한다(msg_level). 이젠 buf의 끝까지를 계속 추적하는데, log_buf는 console로 데이터를 내보내기 전에 데이터를 일시적으로 log를 저장하는 환형(circular) buffer이다. 따라서, 일단 p-프린트할 내용을 담고 있는 버퍼의 포인터-를 log_buf에 하나씩 차례로 넣어준다. 만약 p의 내용에 new line을 나타내는 문자가 있다면, line_feed를 설정하고, log size가 LOG_BUF_LEN을 넘어서면, log_start값을 증가시켜서 앞서서 저장했던 log를 지운다. 전체 log 메시지의 길이는 logged_chars가 될 것이다.

⁴¹⁶ 즉, 이것은 argument가 stack에 넘어올 것이라는 가정하에, fmt에 들어갈 실제 argument에 대한 포인터를 생성하기 위한 것이다.

이전 msg_level이 console_loglevel보다 작은지를 비교한다. 물론 console driver가 있어야 할 것이다. 즉, console driver가 있으며, enable되었고, write함수를 정의하고 있다면, 이를 이용해서 메시지를 각각의 console로 write할 것이다. 다시 loop를 돌기전에 line_feed가 설정되었는지를 확인하고, 설정이 되었다면, msg_level을 -1초기화 시킨다. 나머지 부분은 앞에서 설정했던 console에 대한 lock을 해제하고 위해서 spin_unlock_irqrestore()를 호출하고, log_wait에서 대기하고 있는 job을 깨우기 위해서 wake_up_interruptible() 함수를 실행시켜주게 되며⁴¹⁷, write한 byte수를 복귀 값으로 전달한다.

17.7.3.12. init_modules() 함수의 분석

커널의 설정에서 CONFIG_MODULES이 정의되어 있다면, init_modules() 함수가 start_kernel() 함수에서 호출될 것이다. init_modules() 함수는 커널의 모듈 설정과 관련된 초기화를 수행하는 함수이다. 정의는 ~/kernel/module.c에 아래와 같이 되어 있다.

```
void __init init_modules(void)
{
    kernel_module.nsyms = __stop__ksymtab - __start__ksymtab;

#ifndef __alpha__
    __asm__("stq $29,%0" : "=m"(kernel_module_gp));
#endif
}
```

코드 1050. init_modules() 함수의 정의

이 함수는 단지, kernel_module구조체의 nsyms필드를 __stop__ksymtab에서 __start__ksymtab값을 빼서 넣어주는 일만 한다. 만약 __alpha__가 정의된 경우에는(alpha architecture), kernel_module의 gp필드에 29를 넣어둔다. 이것은 alpha에 architecture에 의존적인 것으로 주소의 boundary를 검사하는 곳에서 사용된다.⁴¹⁸ __stop__ksymtab와 __start__ksymtab는 및 exception table에 대한 정의는 ~/arch/arm/vmlinux-armv.lds.in에 이미 되어 있다. 따라서, 이곳에 들어가는 크기가 얼마나 되는지를 nsyms에 저장해둔다(혹은 symbol table의 entry의 갯수가 될 것이다.) 이 부분은 text와 data의 사이에 해당공간을 점유하고 있으며, 데이터가 8192 bytes로 정렬되어 symbol table이후에 따라 나오고 있다.

```
/* module구조체의 정의는 ~/include/linux/module.h에서 찾을 수 있다.*/
static struct module kernel_module =
{
    size_of_struct:           sizeof(struct module),          /* 이 구조체의 크기 */
    name:                   "",                                /* 모듈의 이름 */
    uc:                     {ATOMIC_INIT(1)},                /* 모듈의 usage count */
    flags:                  MOD_RUNNING,                 /* 모듈의 상태를 나타내는 flag */
    syms:                   __start__ksymtab,             /* 모듈의 커널 symbol table의
시작주소 */
    ex_table_start:          __start__ex_table,            /* Exception table의 시작주소 */
    ex_table_end:            __stop__ex_table,              /* Exception table의 끝주소 */
    kallsyms_start:          __start__kallsyms,            /* 모든 커널 symbol의 시작 주소 */
    kallsyms_end:            __stop__kallsyms,             /* 모든 커널 symbol의 끝주소 */
};
```

코드 1051. kenrel_module구조체의 정의

커널에 올라가는 모듈의 정의는 module구조체가 하고 있다. kernel_module 구조체도 module 구조체의 형을 가지고 정의한다. module 구조체의 정의는 ~/include/linux/module.h에 있으며, 다음과 같이 정리해서 볼 수 있다.

⁴¹⁷ 다른 logging을 위한 job(or task, thread)들이 있다면, 깨워주어야 할 것이다.

⁴¹⁸ mod_bound()와 같은 함수에서 이와 같은 일을 한다.

필드	설명
unsigned long size_of_struct	이 구조체의 크기를 알려준다.
struct module *next	다음 모듈 구조체에 대한 포인터이다.(모듈들은 전부 연결 리스트로 데이터를 유지한다.)
const char *name	모듈의 이름을 알려준다.
unsigned long size	모듈의 크기를 말해준다.
union{ atomic_t usecount	모듈의 사용 개수를 알려준다. 즉, module의 usage count이다.
long pad}uc	모듈의 usage count의 나머지로 padding을 둔다.
unsigned long flags	모듈의 상태를 나타내는 flag으로 자동 제거와 같은 설정을 나타낸다.
unsigned nsyms	모듈에서 export한(즉, 커널의 다른 부분에서 사용할 수 있는) symbol의 수를 나타낸다.
unsigned ndeps	언급되는(referenced) 모듈의 개수를 나타낸다. 이것은 모듈간의 의존관계를 나타내는 것으로 어떤 모듈을 수행하기 위해서 다른 모듈을 필요로 하는 경우에 쓰인다.
struct module_symbol *syms	모듈에서 export하는 심벌의 테이블을 가르킨다.
struct module_ref *deps	언급되는 모듈들의 리스트를 가르키는데 사용하는 포인터이다.
struct module_ref *refs	현재의 모듈을 언급하는 모듈들에 대한 리스트를 가르키는데 사용하는 포인터이다.
int (*init)(void)	모듈의 초기화를 수행하는 함수로 모듈이 적재/loading될 때 호출되어야 하는 함수이다. 이 함수에서 궁극적으로 다른 커널 인터페이스의 처리를 담당하는 함수들이 초기화 될 수 있을 것이며, 또한 각종 자원의 할당이나 디바이스의 등록등을 할 수 있다.
void (*cleanup)(void)	모듈의 해제(unload)할 때 커널에서 호출해주는 함수로 적재시에 했던 일들을 다시 되돌리는 일을 할 것이다. 즉, 등록된 디바이스를 해제하거나, 할당했던 자원들을 회수하는 일이 될 것이다.
const struct exception_table_entry *extable_start	예외 table의 시작 주소를 가진다.
const struct exception_table_entry *ex_table_end	예외 table의 마지막 주소를 가진다.
unsigned long gp	Alpha architecture에서 사용하게되는 필드이다.
const struct modulePersist *persist_start	기본 모듈에 대한 확장이나 아직 특별한 사용이 보이지 않는다.
const struct modulePersist *persist_end	위와 같다.
int (*can_unload)(void)	모듈을 unload하기 전에 호출되는 것으로 현재의 모듈이 unload가 가능한지를 알기 위해서 호출되는 함수이다.
int runsize	현재로서는 사용되지 않는다.
const char *kallsyms_start	모든 커널 심벌들에 대한 시작 주소이다.
const char *kallsyms_end	모든 커널 심벌들에 대한 마지막 주소이다.
const char *archdata_start	모듈을 위해 architecture에 의존적인 데이터를 저장하는 공간의 시작 주소이다.
const char *archdata_end	모듈을 위해 architecture에 의존적인 데이터를 저장하는 공간의 마지막 주소이다.
const char *kernel_data	커널에서 내부적으로 사용하기 위한 데이터들을 위한 공간이다.

표 117. module구조체의 정의

module구조체는 앞에서 커널의 모듈 프로그램을 다룰때 이미 한번 보았다. 여기서는 이것을 표로 다시 한번 보여주는 것에 불과한다. 커널 모듈 프로그램을 참고하기 바란다. 여기서 중요한 점은 앞에서 다루지 않은 커널에서 모듈의 기능을 사용하기 위한 모듈의 초기화에 관련된 것이다.

ARM에의 exception table은 fault를 발생시킬 수 있는 instruction과 이것이 발생했을 때 어디서 수행을 계속할 것인가를 가리키는 포인터로 이루어져 있다. Architecture에 의존적인 데이터 구조는 현재로서는 IA64 architecture에 사용이 보이고 있으며 다른 arhitecture에서는 찾을 수 없다.⁴¹⁹

init_modules()함수를 정리하면, 앞에서 보았듯이 모듈의 가장 상위에 있는 데이터 구조를 초기화 하는 일을 담당하는 것을 알 수 있다. 심벌의 테이블에 대한 포인터와 export되는 심벌의 개수등에 대한 필드를 초기화한다. 나중에 모듈의 컴파일된 object file들을 적재(load)하게 될 때, 여기서 초기화한 모듈 구조체(kernel_module)에 연결될 것이다.

```
/* init_module() 함수 호출 */
if (prof_shift) {
    unsigned int size;
    /* only text is profiled */
    prof_len = (unsigned long) &_etext - (unsigned long) &_stext;
    prof_len >>= prof_shift;

    size = prof_len * sizeof(unsigned int) + PAGE_SIZE-1;
    prof_buffer = (unsigned int *) alloc_bootmem(size);
}
/* kmem_cache_init() 함수 호출 */
```

코드 1052. start_kernel() 함수의 정의(계속)

다시 start_kernel() 함수의 이야기로 돌아가서, 모듈의 초기화 이후에 prof_shift변수에 값이 있고 없음에 따라서, profile을 위한 buffer를 할당하는 부분이 온다. Profile 버퍼를 위해서 필요한 공간은 _etext와 _stext사이의 크기이며, 할당하는 함수는 alloc_bootmem()이다. alloc_bootmem() 함수는 bootmem⁴²⁰에서 해당하는 크기 만큼의 메모리 공간을 할당하는 역할을 담당한다. _etext와 _stext도 역시 앞에서 본 ~/arch/arm/vmlinux_armv.lds.in에서 찾을 수 있을 것이다. 이 부분에 대한 profile을 위한 저장공간을 할당한다고 보면 될 것이다. 이젠 커널에서 데이터 구조체에 대한 메모리를 할당을 위한 초기화를 위해서 kmem_cache_init()를 start_kernel() 함수에서 호출한다.

17.7.3.13.kmem_cache_init() 함수의 분석

kmem_cache_init() 함수는 ~/mm/slab.c에 아래와 같이 정의되어 있다. slab.c에 있는 내용은 이미 이전에 커널 구조에 대해서 이야기 할 때, slab allocator를 설명하면서 이미 보았다. 다시 한번 간략히 요약해 보면, Linux는 기본적으로 메모리를 페이지 단위로 관리하는 buddy algorithm을 사용하며, 커널에서 할당하는 각 object에 대한 것은 slab allocator라는 algorithm을 사용한다. Slab allocator는 기본적으로는 buddy algorithm에 기반한 것으로 페이지를 할당받아서, 각각의 커널 object의 크기에 맞게 특정 크기로 메모리를 이미 나누어서 사용할 수 있도록 하는 algrithm이다.⁴²¹

```
/* Initialisation - setup the `cache' cache. */
void __init kmem_cache_init(void)
{
    size_t left_over;
```

⁴¹⁹ IA64에서 unwind라는 기능을 사용하기 위해서 쓰고 있는 것 같지만, 정확히 무엇을 의미하는지는 아직 알 수 없다.

⁴²⁰ bootmem이란 현재 커널을 위해서 할당한 메모리 공간을 말한다.

⁴²¹ 물론 slab allocator가 buddy algorithm에 기반한다는 말에 조금 문제가 있을 수 있다. 하지만, buddy algorithm을 사용해서 page단위의 메모리를 할당받아서, 이를 작게 나누거나 혹은 합쳐서 사용하고 있다는 점에서는 어느정도 타당성이 있기 때문에 그렇게 썼다.

```

init_MUTEX(&cache_chain_sem);
INIT_LIST_HEAD(&cache_chain);
kmem_cache_estimate(0, cache_cache.objsize, 0,
                    &left_over, &cache_cache.num);
if (!cache_cache.num)
    BUG();
cache_cache.colour = left_over/cache_cache.colour_off;
cache_cache.colour_next = 0;
}

```

코드 1053. kmem_cache_init() 함수의 정의

`kmem_cache_init()` 함수는 `cache_chain`이라는 전역 변수에 전체 커널 메모리 공간에서 가지는 커널 object를 위한 cache를 연결리스트의 형태로 유지한다. 먼저, `cache_chain_sem`의 세마포어를 초기화 시키고(`init_MUTEX()`), `chain_chain` 변수의 리스트 구조를 초기화 한다(`INIT_LIST_HEAD()`). `kmem_cache_estimate()` 함수는 주어진 slab에 대해서 남은 byte의 수와 object의 개수, 그리고, 얼마나 사용하고 있는지를 계산해주는 역할을 한다. 만약 `kmem_cache_estimate()`를 사용해서 구한 object의 수가 0인 경우에는 에러가 되며, `BUG`를 호출하도록 한다. 즉, 해당 slab에 대해서 object가 하나도 존재하지 않는다는 말이므로 오류이다.

Slab에서 사용하는 `colour`는, 여러 slab내에서 object가 떨어져 있는 offset에 대해서 하드웨어 cache의 동일한 위치에 놓일 수 있는 확률이 높게 나타나, cache를 사용하는 효율성을 저하시키는 요인으로 작용할 가능성이 있기 때문에, 임의적인 다른 값을 slab들에 대해서 배정해서 각기 다른 cache line상에 slab을 두기 위해서 사용한다. 위의 코드를 보면, 현재 slab의 `colour` 필드는 남은 크기를 offset으로 나눈 값으로 주고 있음을 볼 수 있다. `colour_next` 필드는 현재 사용하는 `colour` 값을 넣도록 하고 있으며, 이를 이용해서 다음번의 slab에 대한 `colour` 할당에 사용한다. 여기서는 단순히 0으로 초기화만을 했다. 이에 관련된 것을 더 보고자 한다면, 이 문서의 memory 관리 부분을 참조하기 바란다.

```

sti();
calibrate_delay();
#endif CONFIG_BLK_DEV_INITRD
    if (initrd_start && !initrd_below_start_ok &&
        initrd_start < min_low_pfn << PAGE_SHIFT) {
        printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
              "disabling it.\n", initrd_start, min_low_pfn << PAGE_SHIFT);
        initrd_start = 0;
    }
#endif

```

코드 1054. start_kernel() 함수의 정의(계속)

`kmem_cache_init()` 함수를 수행하고 나면, 이제 부터는 인터럽트의 발생 시켜서 처리할 준비가 다 되었으므로, `sti()` 매크로를 호출하도록 한다. 여기서 잠시 ARM에서 정의하고 있는 `sti()`와 `cli()`에 대해서 알아보기로 하자. 각각은 다시 `_sti()`와 `_cli()` 매크로되어 있으며, 코드는 `~/include/asm-arm/proc-armv/system.h`에 아래와 같이 되어 있다.

```

/*
 * Enable IRQs
 */
#define __sti()
    ({ \
        unsigned long temp; \
        __asm__ __volatile__( \
            "mrs    %0, cpsr          @ sti\n" \
            "bic    %0, %0, #128\n" \
            "msr    cpsr_c, %0" \
            : "=r" (temp) \
        ); \
    })

```

```

:
: "memory");
})

```

코드 1055. __sti() 매크로의 정의

__sti() 매크로는 인터럽트를 enable시켜주는 매크로이다. 즉, 현재 cpsr레지스터의 값을 읽어드려서, 이 값에서 I bit을 clear시킨 후, 다시 cpsr에 적어주면 된다.

```

/*
 * Disable IRQs
 */
#define __cli()
({
    unsigned long temp;
    __asm__ __volatile__(
        "mrs    %0, cpsr      @ cli\n"
        "    orr    %0, %0, #128\n"
        "    msr    cpsr_c, %0"
        : "=r" (temp)
        :
        : "memory");
})

```

코드 1056. __cli() 매크로의 정의

__cli() 매크로는 __sti() 매크로와는 반대로 cpsr레지스터의 값을 읽은 후, I bit을 set(=1)시켜서 다시 cpsr레지스터에 쓰는 일을 한다. 여기서 중요한 것은 cpsr_c와 같이 뒤에 “_c”필드가 따라나오는 것인데, 이는 cpsr레지스터의 특정 필드에만 쓰기를 한다는 의미를 가지고 있다. 이것은 호환성 문제를 가지고 있으므로 주의해서 사용해야 할 부분이다. 즉, 그냥 단순히 cpsr레지스터 전부분에 대해서 값을 쓰기보다는 바꾸려고 의도하는 부분에 대해서만 write시켜주는 “_xxx”와 같은 것을 덧붙여서 사용해야 한다는 것이다.⁴²²

나머지 start_kernel() 함수는 CONFIG_BLK_DEV_INITRD가 설정된 경우에 대해서 check 및 initial RAM disk의 시작 주소를 제 설정해주는 역할을 하는 부분이다. 만약 initial RAM disk의 시작번지가 주어져 있고, 하위 번지에 initial RAM disk를 올리는 것을 허락하지 않으며, 시작번지의 주소가 페이지 프레임(frame)의 하한값 보다도 작다면, 이는 잘못된 시작번지를 가지는 initial RAM disk이므로, 에러 메시지를 보여주고, initrd_start값으로 0을 설정하도록 한다. 즉, 적어도 initial RAM disk가 페이지 프레임의 하한선 보다는 큰 값을 가지고 있는지를 확인하는 부분인 것이다.

```

mem_init();
kmem_cache_sizes_init();
#endif CONFIG_3215_CONSOLE
    con3215_activate();
#endif
#ifndef CONFIG_PROC_FS
    proc_root_init();
#endif
    mempages = num_physpages;

```

코드 1057. start_kernel() 함수의 정의(계속)

mem_init() 함수는 메모리 맵에서 사용 가능한 공간을 표시하며, 얼마나 많은 메모리를 사용할 수 있는지를 알려준다. kmem_cache_sizes_init() 함수는 커널의 object를 생성한 크기에 따른 메모리 공간을 초기화

⁴²² 자세한 것은 ARM Architecture Reference Manual을 참고하도록 하자.

하며, `con3215_activate()` 함수는 `CONFIG_3215_CONSOLE`이 설정되었을 경우에만 `activate`되는 `console`이다. 이것은 VM시스템이나 S390과 같은 시스템에서 사용하는 `console`의 일종으로 생각하면 된다. `CONFIG_PROC_FS`는 `proc` 파일 시스템을 사용하는 경우에 설정되는 것이며, `proc_root_init()`함수는 `proc` 파일 시스템의 root를 초기화 하는 일을 한다. 이 `proc` 파일 시스템은 시스템의 정보를 파일 시스템의 형태로 가지고 있도록 만들어서, 일반적인 파일에 대한 접근과 같은 방식으로 시스템의 여러 정보를 제공해 준다. 제일 마지막에 있는 `mempages`는 커널에서 사용하는 메모리 공간에 대한 설정이 끝났으므로, 이제 실제적으로 사용할 수 있는 물리적인 페이지의 수를 가지는 변수로 사용된다.

17.7.3.14. `mem_init()` 함수의 분석

`mem_init()` 함수는 메모리 시스템에 대한 초기화를 담당하는 함수이다. 이 함수의 정의는 `~/arch/arm/mm/init.c`있으며, 아래와 같다. `mem_init()`는 `mem_map`에 있는 사용 가능한 영역(free area)을 표시하고, 얼마 정도의 메모리가 사용 가능한지를 알려준다. 즉, 이 함수는 커널의 이미지 뒤에 있는 영역에 대해서 여러 시스템의 구성요소가 메모리를 요청을 한 후에 일어난다.

```
void __init mem_init(void)
{
    unsigned int codepages, datapages, initpages;
    int i, node;

    codepages = &_etext - &_text;
    datapages = &_end - &_etext;
    initpages = &__init_end - &__init_begin;
    high_memory = (void *)__va(meminfo.end);
    max_mapnr = virt_to_page(high_memory) - mem_map;
    /*
     * We may have non-contiguous memory.
     */
    if (meminfo.nr_banks != 1)
        create_memmap_holes(&meminfo);
    /* this will put all unused low memory onto the freelists */
    for (node = 0; node < numnodes; node++)
        totalram_pages += free_all_bootmem_node(NODE_DATA(node));
#endif CONFIG_SA1111
    /* now that our DMA memory is actually so designated, we can free it */
    free_area(PAGE_OFFSET, (unsigned long)swapper_pg_dir, NULL);
#endif
/*
 * Since our memory may not be contiguous, calculate the
 * real number of pages we have in this system
 */
printk(KERN_INFO "Memory:");
}
```

코드 1058. `mem_init()` 함수의 정의

`codepages`는 커널의 `text(code)`를 가지는 부분을 나타내는 페이지를, `datapages`는 커널의 데이터 영역에 있는 페이지를, `initpages`는 커널 컴파일에서 `_init`로 선언된 부분에 대한 페이지들을 나타낸다. 이들 각각은 앞에서 이미 약간 보았지만, 커널을 컴파일 하는 과정에서 `ld script`의 `input`으로 주어지는 것들이이다. 정의는 `~/arch/arm/vmlinux-armv.lds.in`에 있다. 여기서 말하는 `data`에는 초기화된 데이터를 위한 공간과 초기화 되지 않은 데이터를 위한 `BSS(Block Started by Symbol)`공간을 포함한다. `high_memory`는 `meminfo.end`의 가상 주소(virtual address)를 가지는 변수이며, `max_mapnr`은 최대 `map`의 개수를 가지는 변수이다. 여기서 `high_memory`를 설정하는데 사용하는 `meminfo`변수는 앞에서 이미 메모리에 대한 설정에서 잠시 보았다. `mem_map`은 사용 가능한 메모리의 첫 부분에 위치하는 변수이며, `high_memory`를 `page`단위로 만들어서 이 값을 뺀 값이 사용할 수 있는 최대 페이지의 개수가 된다. `mem_map`은 `mem_map_t` 구조체로 정의된 변수이며, `mem_map_t`은 하나의 페이지 프레임에 대한 기술을 가지는

데이터 타입이다. 따라서, `mem_map`은 시스템에 있는 전체 페이지에 대한 프레임 기술을 가지는 연결 리스트로 생각하면 될 것이다.

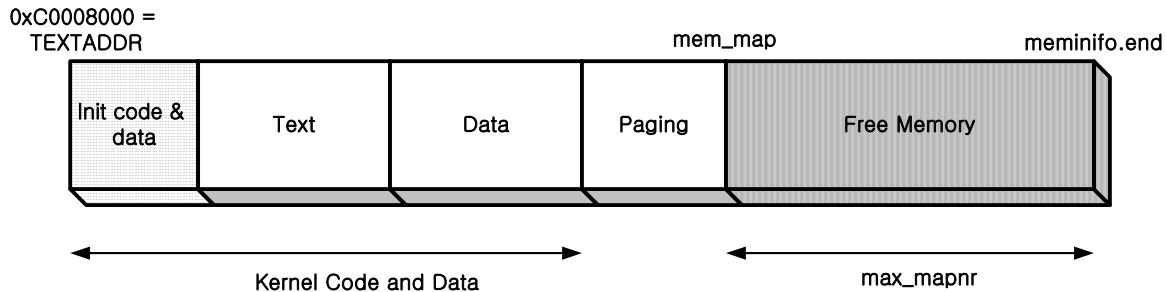


그림 133. 커널의 배치도

만약 `meminfo.nr_banks`의 값이 1이 아니라면, 두개 이상의 메모리 bank를 가지고 있는 경우가 되므로, 메모리 공간이 연속적이지 않을 수 있다. 따라서, `create_memmap_holes()` 함수를 호출해서 사용되지 않는 영역에 대해서 `free`를 시켜준다. 이젠 boot를 위해서 사용했던 메모리 공간을 `free`시켜주기 위해서 각각의 node별로 사용되었던 boot 메모리 공간을 해제하는 `free_all_bootmem_node()` 함수를 호출한다. 이렇게 해제된 영역에 대한 페이지는 `totalram_pages` 변수가 가지도록 한다. 이젠 커널의 메모리 사용 정보를 출력하기 위해서 `printk()` 함수를 호출하고, 이하에서 내용을 출력할 것이다.

```

num_physpages = 0;
for (i = 0; i < meminfo.nr_banks; i++) {
    num_physpages += meminfo.bank[i].size >> PAGE_SHIFT;
    printk("%ldMB", meminfo.bank[i].size >> 20);
}
printk("= %luMB total\n", num_physpages >> (20 - PAGE_SHIFT));
printk(KERN_NOTICE "Memory: %luKB available (%dK code,"
        "%dK data, %dK init)\n",
        (unsigned long) nr_free_pages() << (PAGE_SHIFT-10),
        codepages >> 10, datapages >> 10, initpages >> 10);
if (PAGE_SIZE >= 16384 && num_physpages <= 128) {
    extern int sysctl_overcommit_memory;
    /*
     * On a machine this small we won't get
     * anywhere without overcommit, so turn
     * it on by default.
     */
    sysctl_overcommit_memory = 1;
}

```

코드 1059. `mem_init()` 함수의 정의(계속)

전체 사용할 수 있는 물리적인 메모리의 페이지는 `num_physpages` 변수로 나타낸다. 각각의 `meminfo` 구조체의 각 메모리 bank의 크기를 page 단위로 계산해서 이를 `num_physpages`로 나타낸다. 이렇게 구한 값이 얼마만큼의 메모리 공간을 사용할 수 있는지를 나타내기 위해서 `meminfo` 구조체의 bank 크기를 각각 프린트한다. 이젠 물리적인 페이지가 얼마나 있는지를 알게되었다. 이젠 앞에서 구한 값을 각각 `print`하도록 한다. 만약 page의 크기가 16384bytes(=16Kbytes) 이상이며, 페이지의 갯수가 128이하라면, `sysctl_overcommit_memory` 변수를 1로 설정한다. 여기서 `nr_free_pages()` 함수는 전체 할당 가능한 페이지의 수를 돌려주는 함수이다. 또한 `sysctl_overcommit_memory` 변수는 필요한 공간 이상의 메모리가 존재함을 나타내는 것으로, 나중에 가상 메모리를 할당 받을 때, 충분한 메모리 공간이 있는지를 확인하는 곳에서 사용한다. `num_physpages`는 다시 `start_kernel()` 함수에서 `mempages`를 초기화하는데 사용한다. 이것은 앞에서 이미 보았다.

17.7.3.15.kmem_cache_sizes_init() 함수의 분석

kmem_cache_sizes_init() 함수는 앞에서 이미 커널에서 사용하는 cache 메모리 공간에 대한 초기화를 마쳤기에 여기서 각 크기별로 사용하게 될 cache object를 위한 초기화를 담당한다. 코드는 ~/mm/slab.c에 아래와 같이 정의되어 있다.

```
/* Initialisation - setup remaining internal and general caches.
 * Called after the gfp() functions have been enabled, and before smp_init().
 */
void __init kmem_cache_sizes_init(void)
{
    cache_sizes_t *sizes = cache_sizes;
    char name[20];

    if (num_physpages > (32 << 20) >> PAGE_SHIFT)
        slab_break_gfp_order = BREAK_GFP_ORDER_HI;
    do {
        sprintf(name, "size-%Zd", sizes->cs_size);
        if (!(sizes->cs_cachep =
            kmem_cache_create(name, sizes->cs_size,
                0, SLAB_HWCACHE_ALIGN, NULL, NULL))) {
            BUG();
        }
        /* Inc off-slab bufctl limit until the ceiling is hit. */
        if (!OFF_SLAB(sizes->cs_cachep)) {
            offslab_limit = sizes->cs_size-sizeof(slab_t);
            offslab_limit /= 2;
        }
        sprintf(name, "size-%Zd(DMA)", sizes->cs_size);
        sizes->cs_dmacachep = kmem_cache_create(name, sizes->cs_size, 0,
            SLAB_CACHE_DMA|SLAB_HWCACHE_ALIGN, NULL, NULL);
        if (!sizes->cs_dmacachep)
            BUG();
        sizes++;
    } while (sizes->cs_size);
}
```

코드 1060. kmem_cache_sizes_init() 함수의 정의

kmem_cache_sizes_init() 함수에 대해서는 앞에서 이미 메모리 관리에 대해서 설명하면서 한번 보았다. 이곳에서는 간단히 어떤 일이 일어나는지만, 그 내용은 별도로 하겠다. 실제 메모리에 있는 물리적인 페이지 수(num_physpages)가 0x1000((32<<20)>>PAGE_SHIFT)보다 큰 경우(이것은 32Mbyte보다 큰 메모리를 가지는 machine에 해당하는 이야기이다.)에는 slab_break_gfp_order를 BREAK_GFP_ORDER_HI로 설정해서 상위의 메모리(16Mbytes 이상의 메모리)에 대해서 더 큰 페이지 order⁴²³만을 사용하도록 해준다. do {} while() loop를 돌면서 kmem_cache_create() 함수를 호출해서 각각의 크기에 대해서 캐시를 할당 받는다. offslab_limit은 slab 당 object의 최대 갯수를 나타내며, slab_t를 내부에 유지하지 않는 캐시들에 해당한다. 이것은 kmem_cache_grow() 함수에서 loop를 제어하는데 사용된다. OFF_SLAB() 매크로는 현재 캐시에 CFLGS_OFF_SLAB flag이 설정되어 있는지 확인하는 것이며, 만약 설정되지 않았다면, offslab_limit을 현재 캐시의 크기에서 slab_t를 뺀 값으로 주고, 다시 이 값을 반으로 나눈다. 마지막으로 DMA 가능 영역에 대한 캐시를 할당하기 위해서 다시 kmem_cache_create() 함수를 호출하는데, 넘겨주는 인수로는 SLAB_CACHE_DMA가 설정된다. 만약 할당 받지 못했다면, BUG()를 호출해서 이를 처리하고, 다음번 loop를 위해서 sizes 변수를 증가시킨다.

⁴²³ 여기서 order란 앞에서 이미 나왔지만, 다시한번 상기한다는 생각에서 페이지의 수가 2의 제곱승으로 나타낼 때 그 승수를 말한다.

17.7.3.16.proc_root_init() 함수의 분석

/proc이하의 메모리내에 존재하는 파일 시스템을 만드는 일을 한느 함수가 proc_root_init() 함수이다. 이 함수가 가지는 파일 시스템은 커널에서 제공하는 것으로 각종 시스템의 정보를 담고 있다. 앞에서 이미 slab allocator와 같은 것이 어떤 할당 공간을 가지고 있는지를 보기 위해서 /proc/slabinfo를 본 적이 있을 것이다. 또한 각종 디바이스 및 인터럽트 할당과 프로세스들에 대한 정보도 이 파일 시스템에서 제공받을 수 있다. proc_root_init() 함수의 정의는 ~/fs/proc/root.c에서 찾을 수 있다.

```
void __init proc_root_init(void)
{
    proc_misc_init();
    proc_net = proc_mkdir("net", 0);
#ifdef CONFIG_SYSVIPC
    proc_mkdir("sysv ipc", 0);
#endif
#ifdef CONFIG_SYSCTL
    proc_sys_root = proc_mkdir("sys", 0);
#endif
    proc_root_fs = proc_mkdir("fs", 0);
    proc_root_driver = proc_mkdir("driver", 0);
#ifndef CONFIG_SUN_OPENPROMFS || !defined(CONFIG_SUN_OPENPROMFS_MODULE)
    /* just give it a mountpoint */
    proc_mkdir("openprom", 0);
#endif
    proc_tty_init();
#ifdef CONFIG_PROC_DEVICETREE
    proc_device_tree_init();
#endif
    proc_bus = proc_mkdir("bus", 0);
}
```

코드 1061. proc_root_init() 함수의 정의

proc_root_init()함수는 코드에서 보듯이 많은 부분들에 대한 호출이 일어난다. 먼저 misc에 대한 생성을 위해서 proc_misc_init()를 호출하고, network과 관련된 것 entry를 만들기 위해서 proc_mkdir()을 호출해서 /proc/net 디렉토리를 생성한다. 또한 시스템 V계열의 IPC를 지원한다면, 이를 위해서 proc_mkdir()을 호출해서 /proc/sysv ipc디렉토리를 생성한다. 각종 시스템 정보들에 대한 파일 시스템 entry를 위해서 호출하는하게 되는데, 들어가는 내용으로는 sys(system), fs(file system), driver, openprom등이 있으며, tty에 관련된 entry를 초기화 하기 위해서 proc_tty_init()를, 시스템에 연결된 디바이스 들에 대한 entry를 초기화 하기 위해 proc_device_tree_init()를 호출한다. 마지막으로 시스템의 버스(bus)에 대한 정보를 가지도록 하는 /proc/bus디렉토리를 생성하기 위해서 다시 proc_mkdir()을 호출한다. 이중에서 가장 자주 나오고 있는 proc_mkdir()에 대해서 알아보도록 하자. 정의는 ~/fs/proc/generic.c에 아래와 같이 되어 있다.

```
struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent)
{
    struct proc_dir_entry *ent = NULL;
    const char *fn = name;
    int len;

    if (!parent && xlate_proc_name(name, &parent, &fn) != 0)
        goto out;
    len = strlen(fn);
    ent = kmalloc(sizeof(struct proc_dir_entry) + len + 1, GFP_KERNEL);
    if (!ent)
        goto out;
    memset(ent, 0, sizeof(struct proc_dir_entry));
    memcpy(((char *) ent) + sizeof(*ent), fn, len + 1);
```

```

ent->name = ((char *) ent) + sizeof(*ent);
ent->namelen = len;
ent->proc_fops = &proc_dir_operations;
ent->proc_iops = &proc_dir_inode_operations;
ent->nlink = 2;
ent->mode = S_IFDIR | S_IRUGO | S_IXUGO;
proc_register(parent, ent);

out:
    return ent;
}

```

코드 1062. proc_mkdir() 함수의 정의

proc_mkdir() 함수는 생성하고자 하는 디렉토리의 entry 이름과, 상위의(parent) 디렉토리의 이름을 넘겨받는다. parent가 없고 xlate_proc_name()함수를 호출해서 0이 아닌 값을 돌려받는다면, out으로 곧바로 제어를 옮긴다. 당연히 돌려주는 값은 NULL이 되어야 할 것이다. xlate_proc_name() 함수는 주어진 이름에 대해서 parsing을 하는 역할을 한다. 돌려주는 값이 0이면 올바른 연산을 수행한 것이며, 해당하는 parent의 /proc내 entry와 마지막에 해당하는 entry의 이름이 된다. 예를 들어서 tty/driver/serial을 넘겨주어서 xlate_proc_name() 함수를 호출했다면, proc entry값으로 /proc/tty/driver에 해당하는 proc_dir_entry 구조체가 parent에 넘어오게 되며, 마지막으로 남은 serial이 fn값으로 넘겨질 것이다.

이전 fn의 길이를 구해서 len에 넣고, proc_dir_entry 구조체에 len+1값을 더한 크기 만큼의 공간을 할당 받는다(kmalloc()). 만약 할당받을 수 없다면, out으로 다시 제어를 옮긴다. 이렇게 할당받은 메모리 공간은 초기에 0으로 두도록 한다(memset()). 여기서, proc_dir_entry 구조체에 대해서 잠시 알아보도록 하자.

```

typedef int (read_proc_t)(char *page, char **start, off_t off, int count, int *eof, void *data);
typedef int (write_proc_t)(struct file *file, const char *buffer, unsigned long count, void *data);
typedef int (get_info_t)(char *, char **, off_t, int);

struct proc_dir_entry {
    unsigned short low_ino;                      /* Entry에 할당된 inode번호 */
    unsigned short namelen;                       /* Entry의 이름 크기 */
    const char *name;                            /* Entry의 이름 */
    mode_t mode;                                /* Access permission */
    nlink_t nlink;                             /* link의 개수 */
    uid_t uid;                                 /* Owner의 사용자 ID */
    gid_t gid;                                 /* Owner의 그룹 ID */
    unsigned long size;                          /* Byte단위의 크기 */
    struct inode_operations * proc_iops;        /* Entry에 관련된 inode operation 포인터 */
    struct file_operations * proc_fops;         /* File object에 관련된 operation 포인터 */
    get_info_t *get_info;                        /* Read를 위해서 호출되는 함수에 대한 포인터 */

    struct module *owner;                         /* Proc이 모듈로 올라갈때 가지는 owner */
    struct proc_dir_entry *next, *parent, *subdir; /* Proc 파일 시스템의 연결 리스트 구조 */
    void *data;                                  /* Entry에 고유한 데이터 영역 */
    read_proc_t *read_proc;                      /* Proc entry에 대한 read함수의 포인터 */
    write_proc_t *write_proc;                     /* Proc entry에 대한 write함수의 포인터 */
    atomic_t count;                             /* 사용 count */
    int deleted;                               /* Delete flag */
    kdev_t rdev;                                /* Proc entry에 관련된 디바이스 */
};

/* Proc 파일 시스템 구조체 */

```

코드 1063. proc_dir_entry구조체의 정의

`proc_dir_entry` 구조체는 `~/include/linux/proc_fs.h`에 정의되어 있다. 이 구조체는 `/proc` 이하에 나오는 `entry`들에 대한 디렉토리 구조체를 정의하고 있다. 각각의 필드가 하는 역할은 위의 코드에 있는 `comment`을 참고하도록 하자. 코드에서 보듯이 `proc_dir_entry`는 `proc` 파일 시스템을 구성하는 하나의 `directory entry`에 대한 기술을 담당한다. 각각의 `proc_dir_entry` 구조체는 `tree` 구조를 이루도록 만들어, 마치 파일 시스템과 같이 `directory` 구조체로 접근해서, 읽고/쓰기가 가능한 구조를 지니도록 메모리 상에 존재하는 파일 시스템을 구성하도록 만들어 준다.

이전 남은 것은 해당하는 `proc_dir_entry` 구조체의 필드를 채워주는 일이다. 먼저 이름을 채워주고(`ent->name`), 이름의 길이(`ent->namelen`)를 채워준다. 그리고나서, `proc_dir_entry` 구조체의 디렉토리에 대한 `operation`과 `inode`에 대한 `operation`을 주어서, 이 `proc` 파일 시스템의 `entry`가 일반적인 파일이나 디렉토리에 대한 접근과 같은 방식으로 사용될 수 있도록 만든다. 또한 현재의 `link` 수를 가지도록 만들어주고(`ent->nlink`), 접근 `bit`은 `S_IFDIR | S_IRUGO | S_IXUGO`로 두어서, 이것이 `proc` 파일 시스템의 디렉토리이며, `read`와 `execute`가 사용자와 `group`에 대해서 설정되었음을 나타낸다. 마지막으로 이것을 `proc` 파일 시스템에 등록하기 위해서 `proc_register()` 함수를 호출한다. 즉, 아직 `proc_dir_entry`를 만들어준 것에 불과 하며, `proc` 파일 시스템에 연결 리스트로 들어가 있지 않기 때문이다.

```
/*
 * These are the generic /proc directory operations. They
 * use the in-memory "struct proc_dir_entry" tree to parse
 * the /proc directory.
 */
static struct file_operations proc_dir_operations = {
    read:           generic_read_dir,
    readdir:        proc_readdir,
};

/*
 * proc directories can do almost nothing..
 */
static struct inode_operations proc_dir_inode_operations = {
    lookup:         proc_lookup,
};
```

코드 1064. `proc` 파일 시스템의 `file(directory)` 및 `inode`에 대한 `operation`의 정의

위의 코드는 `proc` 파일 시스템의 디렉토리 `entry`에 대한 `file` 및 `inode`에 대한 연산을 정의한 것이다. 이것을 통해서 `proc` 파일 시스템이 마치 다른 파일 시스템과 마찬가지로 하나의 파일 시스템과 같이 접근될 수 있도록 만들어 준다. 정의하고 있는 파일에 대한 연산은 `read` 및 `readdir`이 있으며, `inode`에 대한 연산으로는 `lookup`이 존재한다. 각각 해당하는 함수에 대한 `proc` 파일 시스템에서 정의하고 있는 함수가 있을 것이다.

```
static int proc_register(struct proc_dir_entry * dir, struct proc_dir_entry * dp)
{
    int      i;

    i = make_inode_number();
    if (i < 0)
        return -EAGAIN;
    dp->low_ino = i;
    dp->next = dir->subdir;
    dp->parent = dir;
    dir->subdir = dp;
    if (S_ISDIR(dp->mode)) {
        if (dp->proc_iops == NULL) {
            dp->proc_fops = &proc_dir_operations;
            dp->proc_iops = &proc_dir_inode_operations;
        }
    }
```

```

        dir->nlink++;
} else if (S_ISLNK(dp->mode)) {
    if (dp->proc_iops == NULL)
        dp->proc_iops = &proc_link_inode_operations;
} else if (S_ISREG(dp->mode)) {
    if (dp->proc_fops == NULL)
        dp->proc_fops = &proc_file_operations;
}
return 0;
}

```

코드 1065. proc_register() 함수의 정의

proc_register() 함수는 proc_dir_entry구조체에 대한 포인터를 두개 넘겨받는다. 첫번째 것은 해당 directory를 말하는 것이며, 두번째는 그 디렉토리의 한 entry가 될 내용이다. 즉, 앞에서 parent(=dir) 디렉토리와 entry를 넘겨받았다. 먼저 inode번호를 만들어서(make_inode_number()), 이를 i에 저장한다. 만약 i가 0보다 작은 값을 가진다면, 에러가 되며 -EAGAIN을 돌려준다. 그렇지 않다면, 만들어진 inode번호로 entry의 inode필드를 채워주고, next는 parent의 subdir(sub directory)을 가르키도록 만든다. Entry의 parent directory는 물론 parent(=dir)이 될 것이며, parent의 subdir필드는 새로이 만드는 entry를 가르키도록 만든다. 이렇게 해서 연결리스트 속에 추가되어 들어가도록 한다.

만약 만들고자 하는 것이 디렉토리라면(S_ISDIR()), 디렉토리에 대한 연산과 inode에 대한 연산을 초기화 시켜준다.⁴²⁴ 만약 만들고자 하는 것이 link라면(S_ISLNK()), inode에 대한 link연산을 주도록 하며, 만약 등록을 위한 것이라면(S_ISREG()), 파일 연산을 초기화 시켜주도록 한다. 복귀 값은 0이다. 이것으로 proc 파일 시스템의 한 entry를 추가하는 작업을 끝냈다. 코드에서 보듯이 각각의 entry가 어떤 역할을 하는지에 따라서, 다른 연산이 정의됨을 볼 수 있었다. 이것은 나중에 커널에서 일반적인 파일 시스템에 대한 접근과 동일한 형태로 호출될 연산들이다.

17.7.3.17.fork_init() 함수의 분석

```

void __init fork_init(unsigned long mempages)
{
/*
 * The default maximum number of threads is set to a safe
 * value: the thread structures can take up at most half
 * of memory.
*/
#if THREAD_SIZE > PAGE_SIZE
    max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 2;
#else
    max_threads = (mempages * PAGE_SIZE) / (2 * THREAD_SIZE);
#endif

    init_task.rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
    init_task.rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
}

```

코드 1066. fork_init() 함수의 정의

fork_init() 함수가 호출에서 넘겨받는 값은 mempages로서 앞에서 구한 num_physpages변수의 값을 가진다. 즉, 사용할 수 있는 물리적인 페이지 수를 가진다. 이 함수는 init_task⁴²⁵의 자원 할당과 관련된 것을 초기화 한다. 즉, task_struct구조체의 rlim 구조체 필드에 있는 rlim_cur필드와 rlim_max필드를 초기화 한다. rlimit 구조체의 타입으로 정의된 rlim 필드는 프로세스가 사용할 수 있는 자원의 한계를 설정하는

⁴²⁴ 앞에서 우린 이미 directory에 대한 생성을 보고 있지만, proc_register() 함수는 directory의 entry를 등록하는데도 사용된다. 따라서, directory인지 아니면, 단순히 entry를 나타내는지를 확인해야 한다.

⁴²⁵ Linux에서 가장 먼저 생성되는 task이다. 이것은 나중에 다시 보게될 init 프로세스와는 다르다.

역할을 한다. rlim_cur필드는 soft limit을 나타내는 값이며, rlim_max는 absolute limit를 나타낸다. 이중에서 RLIMIT_NPROC은 프로세스가 가진 사용자당 생성할 수 있는 프로세스의 최대 개수를 나타내는 필드에 대한 자원 설정을 말하는 것으로 이곳에서는 max_thread/2로 설정하고 있다. 이때, max_thread값은 THREAD_SIZE가 PAGE_SIZE보다 크냐 작냐에 따라서, 그 값이 각각 결정된다. 즉, 최대로 생성할 수 있는 프로세스(여기서는 thread라고 나타내고 있다.)⁴²⁶의 수를 사용가능한 메모리의 양에 따라서 정하고 있음을 확인할 수 있다.

여기서, 프로세스의 자원에 사용의 한계(limit)에 대해서 잠시 더 살펴보도록 하자. 앞에서 보았듯이 RLIMIT_NPROC이외에 아래의 표와 같은 값들이 ~/include/asm-arm/resource.h에 더 정의되어 있다.

ID	Value	Description
RLIMIT_CPU	0	Millisecond단위로 표현된 CPU사용시간(processor time).
RLIMIT_FSIZE	1	최대 파일의 크기로서 byte단위로 나타낸다.
RLIMIT_DATA	2	최대 데이터 segment의 크기로 byte단위로 나타낸다.
RLIMIT_STACK	3	최대 스택 segment의 크기로 byte단위로 나타낸다.
RLIMIT_CORE	4	치명적인 프로그램 에러가 발생했을 때 생성될 core파일의 최대 크기로 byte단위로 나타낸다.
RLIMIT_RSS	5	중앙 메모리에 상주할 수 있는 데이터의 최대 크기로 byte단위로 나타낸다.
RLIMIT_NPROC	6	사용자당 생성할 수 있는 최대 프로세스의 개수를 나타낸다.
RLIMIT_NOFILE	7	최대로 open할 수 있는 파일의 수를 나타낸다.
RLIMIT_MEMLOCK	8	중앙 메모리에 가질 수 있는 최대로 locking된 데이터의 크기를 byte단위로 나타낸다.
RLIMIT_AS	9	최대 주소 공간(address space)공간을 byte단위로 나타낸다.
RLIMIT_LOCKS	10	최대로 가질 수 있는 파일에 대한 lock을 나타낸다.
RLIM_NLIMITS	11	위에서 정의하고 있는 ID의 갯수를 나타낸다(11개).

표 118. 자원의 한계에 대한 상수 값들의 정의

앞에서는 이중에서 RLIMIT_NPROC에 대한 것을 index로 사용해서 soft limit와 absolute limit을 설정했다. 이를 값들은 나중에 프로세스를 생성하는 과정에서 child 프로세스로 상속된다. 물론 이와 같은 과정에서 이들 값들에 대해서 약간씩의 조정이 있을 수 있다. 또한 관련된 시스템 콜(call)로서는 setrlimit()와 getrlimit() 함수등이 존재한다. 각각이 rlimit를 설정하거나, 설정된 값을 읽어오기 위해서 사용된다.

17.7.3.18.proc_caches_init() 함수의 분석

proc_caches_init() 함수는 프로세스가 사용하게 될 커널 object들에 대한 커널의 캐시를 위한 메모리 공간을 초기화 하는 것이다. 앞에서 이미 slab allocator에 대한 초기화를 마쳤기에 이곳에서는 단지 해당하는 object들에 대한 것을 slab에서 생성하도록 하는 함수들만 호출하면 된다. 이 함수의 정의는 ~/kernel/fork.c에 아래와 같이 되어 있다.

```
void __init proc_caches_init(void)
{
    sigact_cachep = kmem_cache_create("signal_act",
                                      sizeof(struct signal_struct), 0,
                                      SLAB_HWCACHE_ALIGN, NULL, NULL);
    if (!sigact_cachep)
        panic("Cannot create signal action SLAB cache");
    files_cachep = kmem_cache_create("files_cache",
                                    sizeof(struct files_struct), 0,
```

⁴²⁶ Linux에서는 실제로 프로세스와 thread에 대해서 task_struct라는 구조체로 같이 나타내고 있다.

```

        SLAB_HWCACHE_ALIGN, NULL, NULL);
if (!files_cachep)
    panic("Cannot create files SLAB cache");
fs_cachep = kmem_cache_create("fs_cache",
                             sizeof(struct fs_struct), 0,
                             SLAB_HWCACHE_ALIGN, NULL, NULL);
if (!fs_cachep)
    panic("Cannot create fs_struct SLAB cache");
vm_area_cachep = kmem_cache_create("vm_area_struct",
                             sizeof(struct vm_area_struct), 0,
                             SLAB_HWCACHE_ALIGN, NULL, NULL);
if (!vm_area_cachep)
    panic("vma_init: Cannot alloc vm_area_struct SLAB cache");
mm_cachep = kmem_cache_create("mm_struct",
                             sizeof(struct mm_struct), 0,
                             SLAB_HWCACHE_ALIGN, NULL, NULL);
if (!mm_cachep)
    panic("vma_init: Cannot alloc mm_struct SLAB cache");
}

```

코드 1067. proc_caches_init() 함수의 정의

하나의 커널 object를 위한 커널의 cache공간(slab)으로 부터 할당 받기위해서는 kmem_cache_create() 함수를 사용한다. 이 함수가 받아 들이는 argument값으로는 cache의 이름과, 할당할 공간의 크기, 페이지 내에서의 사용되는 offset값과, slab을 위한 flag, 할당받는 object에 대한 constructor함수 및 destructor함수에 대한 포인터가 있다. 여기서는 아직 constructor와 destructor를 사용하고 있지 않는다.

할당하는 cache로는 signal의 action을 처리하기 위한 signal_struct에 대한 cache공간과, file에 대한 연산을 처리하게 될 files_struct구조체의 cache, 파일 시스템을 위한 fs_struct구조체의 cache, 프로세스의 가상 메모리 공간을 위한 vm_area_struct에 대한 cache 및 프로세스의 task_struct내에 존재하는 mm필드와 관련된 mm_struct를 위한 cache등을 만들고 있다. 이들 각각은 프로세스를 생성하거나 삭제할 때, 자주 커널에서 할당과 해제가 일어나는 커널 object들이다. 따라서, 이들을 cache의 형태로 사용할 수 있다면, 커널의 성능을 개선하는데 도움을 줄 수 있을 것이다.

17.7.3.19.vfs_caches_init() 함수의 분석

vfs_caches_init() 함수는 VFS(Virtual File System)을 유지하기 위해서 필요한 커널의 object들을 생성하기 위한 cache를 초기화 하는 역할을 한다. 정의는 ~/fs/dcache.c에 아래와 같이 되어 있다.

```

void __init vfs_caches_init(unsigned long mempages)
{
    bh_cachep = kmem_cache_create("buffer_head",
                                sizeof(struct buffer_head), 0,
                                SLAB_HWCACHE_ALIGN, NULL, NULL);
    if (!bh_cachep)
        panic("Cannot create buffer head SLAB cache");
    names_cachep = kmem_cache_create("names_cache",
                                PATH_MAX + 1, 0,
                                SLAB_HWCACHE_ALIGN, NULL, NULL);
    if (!names_cachep)
        panic("Cannot create names SLAB cache");
    filp_cachep = kmem_cache_create("filp",
                                sizeof(struct file), 0,
                                SLAB_HWCACHE_ALIGN, NULL, NULL);
    if (!filp_cachep)
        panic("Cannot create filp SLAB cache");
#ifndef CONFIG_QUOTA
    dquot_cachep = kmem_cache_create("dquot",

```

```

        sizeof(struct dquot), sizeof(unsigned long) * 4,
        SLAB_HWCACHE_ALIGN, NULL, NULL);
    if (!dquot_cachep)
        panic("Cannot create dquot SLAB cache");
#endif
    dcache_init(mempages);
}

```

코드 1068. vfs_caches_init() 함수의 정의

vfs_caches_init() 함수에서 할당하는 object의 cache를 보면, buffer_head에 대한 cache, names_cache, 파일 object를 위한 filp cache, 만약 quota설정이 있는 경우를 대비한 dquot cache와 directory entry를 위한 dcache등이 있다. 마지막에 해당하는 cache는 dcache_init() 함수를 불러서 다시 초기화를 하고 있다. 각각의 cache를 할당하고, 할당받을 수 있는지 없는지를 검사하는 것은 앞에서 본 proc_caches_init()와 동일하다.

dcache_init() 함수는 ~/fs/dcache.c에 다시 아래와 같이 정의된다. 이 함수는 VFS 파일 시스템에서 directory에 대한 entry를 cache하는 목적으로 사용되는 cache이다. 메모리 관리(memory management) 부분에서 directory entry cache와 관련된 것을 읽어보기 바란다.

```

static void __init dcache_init(unsigned long mempages)
{
    struct list_head *d;
    unsigned long order;
    unsigned int nr_hash;
    int i;

    /*
     * A constructor could be added for stable state like the lists,
     * but it is probably not worth it because of the cache nature
     * of the dcache.
     * If fragmentation is too bad then the SLAB_HWCACHE_ALIGN
     * flag could be removed here, to hint to the allocator that
     * it should not try to get multiple page regions.
     */
    dentry_cache = kmem_cache_create("dentry_cache",
                                    sizeof(struct dentry),
                                    0,
                                    SLAB_HWCACHE_ALIGN,
                                    NULL, NULL);

    if (!dentry_cache)
        panic("Cannot create dentry cache");
#ifndef PAGE_SHIFT < 13
    mempages >>= (13 - PAGE_SHIFT);
#endif
    mempages *= sizeof(struct list_head);
    for (order = 0; ((1UL << order) << PAGE_SHIFT) < mempages; order++)
        ;
}

```

코드 1069. dcache_init() 함수의 정의

가장 먼저 directory entry를 위한 cache를 생성한다.(kmem_cache_create()). 만약 생성하지 못한다면, 예러 메시지를 보여주고 시스템은 멈추게 될것이다. PAGE_SHIFT가 13보다 작다면, 13에서 PAGE_SHIFT를 뺀 만큼을 mempages에서 우측으로 shift해줄 것이다. 이것은 13-PAGE_SHIFT만큼의 2의 승수로 나누는 효과가 있으며, 현재 PAGE_SHIFT가 120이므로 1만큼을 mempages를 우측으로 shift한다. 즉, 2로 나눈 값이 mempages에 들어갈 것이다. 여기서 중요한 것은 원래의 mempages의 값은 변화가 없고, 단지 이 함수내에서의 mempages의 값만이 영향을 받는다는 것이다. 이렇게 생성된 mempages의 값에 다시

list_head의 크기 만큼을 곱한다. 이것은 앞에서 계산된 mempages만큼의 공간에 대해서 list_head구조체로 hash table을 생성하기 위해서이다. 이것으로 부터 mempages 만큼을 나타내기 위한 2의 승수를 계산하기 위해서 order라는 변수의 값을 증가 시키면서 ($1UL \ll order$) $\ll PAGE_SHIFT$ 가 mempages보다 큰 order값을 정한다. 이 order 값은 나중에 최종적으로 directory entry cache를 위한 hash table의 크기를 할당받는데 사용된다.

```

do {
    unsigned long tmp;

    nr_hash = (1UL << order) * PAGE_SIZE /
              sizeof(struct list_head);
    d_hash_mask = (nr_hash - 1);
    tmp = nr_hash;
    d_hash_shift = 0;
    while ((tmp >= 1UL) != 0UL)
        d_hash_shift++;
    dentry_hashtable = (struct list_head *)
        __get_free_pages(GFP_ATOMIC, order);
} while (dentry_hashtable == NULL && --order >= 0);
printk("Dentry-cache hash table entries: %d (order: %ld, %ld bytes)\n",
       nr_hash, order, (PAGE_SIZE << order));
if (!dentry_hashtable)
    panic("Failed to allocate dcache hash table\n");
d = dentry_hashtable;
i = nr_hash;
do {
    INIT_LIST_HEAD(d);
    d++;
    i--;
} while (i);
}

```

코드 1070. dcache_init() 함수의 정의(계속)

이전 앞에서 생성된 cache에 대한 hash table을 만들어줄 차례이다. 이것을 위해서 우리는 이미 할당받을 공간의 페이지 크기를 나타내는 2의 승수인 order를 계산했었다. 먼저 전체 hash table의 갯수를 정하는 일이다(nr_hash). 이를 위해서 1UL을 order만큼 좌측으로 shift를 하고, 다시 이 값에 PAGE_SIZE를 곱한다. 그리고나서, 다시 이 값을 list_head의 크기로 나누어 전체 hash table의 entry개수를 구하게 된다. Hash table에 대한 mask값으로 nr_hash에서 1을 뺀 값을 주고, hash table에 대한 shift가 얼마나 일어날 수 있는지를 구한다(d_hash_shift). 이젠 hash table을 메모리에서 할당받아야 할 것이다. 할당된 메모리는 페이지 단위가 되며, 2의 order 제곱승 만큼의 갯수가 할당될 것이다. 만약 할당 받을 수 없다면, order를 하나 줄여서 다시 한번 할당 받을 수 있는지 do{} while() loop를 돌면서 확인해 본다. Loop를 빠져 나올때 까지도 할당할 수 없다면, 에러 메시지를 보여주고 시스템을 멈출 것이다.

자, 이것으로 directory entry에 대한 cache를 할당 했다. 이것을 d로 놓도록 하고, hash table의 list_head와 같이 directory entry들을 연결리스트로 가지기 때문에, 이를 초기화하기 위해서 INIT_LIST_HEAD()를 사용해서 하나 하나의 hash table의 entry를 do{} while() loop를 돌면서 모두 초기하도록 한다. 이것으로 directory entry에 대한 cache의 생성 및 hash table에 대한 생성도 끝나게 된다.

17.7.3.20.buffer_init() 함수의 분석

이전 블록 디바이스의 입출력을 위한 buffer cache를 초기화 할 차례이다. 이를 담당하는 것이 buffer_init() 함수이다. 이 함수에서 초기화가 되는 것은 buffer_head 구조체이다. buffer_head 구조체에 대한 것은 이미 block mode 디바이스를 다룰 때 소개했었다. 이곳에서는 이 구조체의 초기화를 보도록 하자. 정의는 ~/fs/buffer.c에 아래와 같이 되어 있다.

```
void __init buffer_init(unsigned long mempages)
```

```

{
    int order, i;
    unsigned int nr_hash;

    /* The buffer cache hash table is less important these days,
     * trim it a bit.
     */
    mempages >= 14;
    mempages *= sizeof(struct buffer_head *);
    for (order = 0; (1 << order) < mempages; order++)
        ;
    /* try to allocate something until we get it or we're asking
       for something that is really too small */
    do {
        unsigned long tmp;

        nr_hash = (PAGE_SIZE << order) / sizeof(struct buffer_head *);
        bh_hash_mask = (nr_hash - 1);
        tmp = nr_hash;
        bh_hash_shift = 0;
        while((tmp >= 1UL) != 0UL)
            bh_hash_shift++;
        hash_table = (struct buffer_head **)
            __get_free_pages(GFP_ATOMIC, order);
    } while (hash_table == NULL && --order > 0);
    printk("Buffer-cache hash table entries: %d (order: %d, %ld bytes)\n",
           nr_hash, order, (PAGE_SIZE << order));
    if (!hash_table)
        panic("Failed to allocate buffer hash table\n");
    /* Setup hash chains. */
    for(i = 0; i < nr_hash; i++)
        hash_table[i] = NULL;
    /* Setup free lists. */
    for(i = 0; i < NR_SIZES; i++) {
        free_list[i].list = NULL;
        free_list[i].lock = SPIN_LOCK_UNLOCKED;
    }
    /* Setup lru lists. */
    for(i = 0; i < NR_LIST; i++)
        lru_list[i] = NULL;
}

```

코드 1071. buffer_init() 함수의 정의

먼저 mempages를 우측으로 14 bit만큼 shift한다. 다시한번 말하지만, 이것은 2의 14승 만큼을 mempages에서 나누어 주는 일을 한다. 따라서, 16K로 mempages를 나누어준 것이다. 다시 이것을 buffer_head 포인터의 크기 만큼을 곱해서 mempages로 둔다. 역시 바뀌는 것은 mempages의 local copy이다. 이것을 가지고 할당 받을 페이지의 order값을 정한다. 뒤에 __get_free_pages() 함수를 directory cache에서 사용한 것과 같이 사용하기 위해서이다.

이전 do{} while() loop를 돌면서 할당받을 공간이 있는지의 여부를 묻고 할당 받는 부분이다. 먼저 buffer_head 구조체를 위해서 살당해야 하는 hash table의 크기를 정한다(nr_hash). 이 값은 order 만큼을 PAGE_SIZE를 좌측으로 shift한 값을 buffer_head 포인터로 나눈 것이 된다. Hash의 mask값은 nr_hash -1로 둔다(bh_hash_mask). 또한 hash가 shift할 수 있는 값을 구해서 이를 bh_hash_shift로 둔다. 마지막으로 buffer_head를 위한 hash table의 크기 만큼을 메모리에서 페이지 단위로 할당받기 위해서 __get_free_pages() 함수를 호출하도록 한다. 만약 할당 받을 수 없다면, order값을 줄여나가면서 do{}

while() loop를 돌게된다. 만약 buffer_head 구조체를 위한 hash table을 할당 받을 수 없다면, 에러 메시지를 출력하고 시스템은 멈추게 될 것이다(panic()⁴²⁷).

이전 이렇게 할당받은 hash_table들을 전부 NULL로 초기화 한다. 즉, 아직 아무것도 buffer cache를 쓰고 있지 않는다. free_list[] 배열은 사용할 수 있는 buffer head들에 대한 연결 리스트를 나타내며, NR_SIZES(=7) 만큼의 원소(element)를 가진다. 이것을 전부 초기화 하도록 한다. 또한 LRU(Least Recently Used) list를 나타내는 lru_list[] 배열도 이곳에서 NULL로 초기화된다. lru_list[] 배열은 buffer head들에 대한 할당을 처리하는데 있어서 LRU algorithm⁴²⁸을 사용하기 위해서 가지는 커널의 자료구조이다.

17.7.3.21.page_cache_init() 함수의 분석

page_cache_init() 함수의 정의는 ~mm/filemap.c에 있으며, 이 함수의 하는 역할은 페이지(page)에 대한 cache를 초기화 하는 일을 한다.

```
void __init page_cache_init(unsigned long mempages)
{
    unsigned long htable_size, order;

    htable_size = mempages;
    htable_size *= sizeof(struct page *);
    for(order = 0; (PAGE_SIZE << order) < htable_size; order++)
        ;
    do {
        unsigned long tmp = (PAGE_SIZE << order) / sizeof(struct page *);
        page_hash_bits = 0;
        while((tmp >= 1UL) != 0UL)
            page_hash_bits++;
        page_hash_table = (struct page **)
            __get_free_pages(GFP_ATOMIC, order);
        } while(page_hash_table == NULL && -order > 0);
        printk("Page-cache hash table entries: %d (order: %ld, %ld bytes)\n",
            (1 << page_hash_bits), order, (PAGE_SIZE << order));
        if (!page_hash_table)
            panic("Failed to allocate page hash table\n");
        memset((void *)page_hash_table, 0, PAGE_HASH_SIZE * sizeof(struct page *));
    }
```

코드 1072. page_cache_init() 함수의 정의

이곳에서 초기화 되는 것은 페이지의 cache에 대한 연결리스트를 가지는 페이지 hash table이다. 따라서, 페이지의 포인터의 배열이 되며, __get_free_pages() 함수를 사용해서 페이지의 hash table로 사용할 페이지를 할당받는다. __get_free_pages() 함수를 사용하기 위해서 order를 구하는 방법은 이전에 취했던 방법들과 비슷하다. 먼저 페이지들의 개수를 페이지를 가르키는 포인터의 크기와 곱하고, 이를 이용해서 얼마만큼의 페이지를 사용할지를 결정하기 위해서 order만큼씩 PAGE_SIZE를 우측으로 shift해서 이 값이 할당 받을 메모리 공간보다 작은 동안 계속적으로 order를 1씩 증가시킨다.

이전 order값을 구했으므로, page hash를 위해서 사용할 hash bit을 만든다. 그리고나서, 페이지의 hash table을 생성하기 위해서 __get_free_pages() 함수를 앞에서 구한 order값을 넘겨주어서 호출하게 된다. 이것으로 페이지 hash table에 대한 할당을 마쳤다. 만약 충분한 페이지 테이블에 대한 공간을 할당 할 수 없다면, order를 1씩 감소시키면서 더 작은 페이지 hash table을 위한 페이지들을 할당 받을 수

⁴²⁷ 실제로 panic() 함수는 에러 메시지를 표시한 후에 시스템을 restart시켜주는 역할을 하는 함수이다. 이 함수를 호출하게되면, 시스템은 더 이상의 진행은 하지 않고, 메시지를 뿐여준 후, 바로 reset을 누른 것과 같은 일을 할 것이다.

⁴²⁸ 즉, 버퍼 헤드중에서 최근에 사용되지 않은 것들을 그 사용된 시간에 따른 연결 리스트로 만들어서, 그 중에서 가장 오래된 것을 선택해서 새로운 buffer head의 할당 요구를 만족 시켜주는 방법이다. 여기서는 이러한 lru_list[] 배열을 사용하고 있으며, 크기로 NR_LIST(=4)를 두고 있다.

있는지를 살펴본다. 역시 할당받을 공간이 없다면, 에러 메시지를 출력하고 시스템은 멈출 것이다. 마지막으로 할당받은 페이지 hash table의 공간을 0으로 초기화 시켜주는 것을 잊지 말도록 하자. 여기서, PAGE_HASH_SIZE는 페이지 테이블의 크기를 말하는 것으로, 1을 앞에서 구한 page_hash_bits만큼 좌측으로 shift한 값이 된다.

17.7.3.22. kiobuf_setup() 함수의 분석

kiobuf_setup() 함수는 커널에서 사용하는 I/O를 위한 버퍼 캐쉬를 생성하는 일을 한다. 정의는 ~/fs/ibuf.c에 아래와 같이 되어 있다.

```
void __init kiobuf_setup(void)
{
    kiobuf_cachep = kmem_cache_create("kiobuf",
                                      sizeof(struct kiobuf),
                                      0,
                                      SLAB_HWCACHE_ALIGN, NULL, NULL);
    if(!kiobuf_cachep)
        panic("Cannot create kernel ibuf cache\n");
}
```

코드 1073. kiobuf_setup() 함수의 정의

이 함수에서 하는 일은 단순히 kmem_cache_create() 함수를 호출해서 kiobuf object를 위한 cache를 할당하는 것이다. 만약 할당 받을 수 없다면, 에러 메시지를 보이고 시스템은 멈출 것이다. 이곳에서 잠시 kiobuf 구조체를 살펴보도록 하자. 정의는 ~/include/linux/ibuf.h에 있다.

```
/*
 * The kiobuf structure describes a physical set of pages reserved
 * locked for IO. The reference counts on each page will have been
 * incremented, and the flags field will indicate whether or not we have
 * pre-locked all of the pages for IO.
 *
 * kiobufs may be passed in arrays to form a iovec, but we must
 * preserve the property that no page is present more than once over the
 * entire iovec.
 */
#define KIO_MAX_ATOMIC_IO          64                                /* in kb */
#define KIO_MAX_ATOMIC_BYTES(64 * 1024)
#define KIO_STATIC_PAGES          (KIO_MAX_ATOMIC_IO / (PAGE_SIZE >> 10) + 1)
#define KIO_MAX_SECTORS           (KIO_MAX_ATOMIC_IO * 2)
...
struct kiobuf
{
    int             nr_pages; /* 실제로 reference하고 있는 페이지의 수 */
    int             array_len; /* 할당된 리스트의 크기(배열의 크기) */
    int             offset;   /* 유효한 데이터의 시작부분에 대한 offset */
    int             length;   /* 데이터에서 유효한 byte의 수 */
    struct page ** maplist;  /* 메모리 mapping된 device의 페이지 리스트에 대한 포인터 */
    unsigned int    locked : 1; /* 페이지들이 locking되어 있음을 나타내는 flag */
    struct page *   map_array[KIO_STATIC_PAGES]; /* 커널 I/O를 위해서 예약된 페이지 들에
 대한 배열 */
    /* Dynamic state for IO completion: */
    atomic_t        io_count; /* I/O가 진행중인 count */
    int             errno;    /* 처리된 I/O에 대한 상태값 */
    void            (*end_io)(struct kiobuf *); /* I/O의 completion이 끝났을 때 호출되는 함수 */
    wait_queue_head_t wait_queue; /* I/O를 위한 wait queue */
}
```

```
};
```

코드 1074. kiobuf 구조체의 정의

kiobuf구조체도 역시 커널에서 자주 사용하는 object로서 생성과 할당이 자주 일어난다. 따라서, 이를 object cache에서 유지하는 것이 바람직 하다.

17.7.3.23.signals_init() 함수의 분석

signals_init() 함수는 시그널의 처리와 관련된 sigqueue object에 대한 cache를 할당하는 일을 한다. 정의는 ~/kernel/signal.c에 아래와 같이 나와 있다.

```
void __init signals_init(void)
{
    sigqueue_cachep = kmem_cache_create("sigqueue",
                                         sizeof(struct sigqueue),
                                         __alignof__(struct sigqueue),
                                         SIG_SLAB_DEBUG, NULL, NULL);
    if (!sigqueue_cachep)
        panic("signals_init(): cannot create sigqueue SLAB cache");
}
```

코드 1075. signals_init() 함수의 정의

sigqueue에 대한 object를 생성하기 위해서 kmem_cache_create() 함수를 호출하고 있다. 할당 할 수 없다면, 에러 메시지를 출력하고 시스템은 멈추게 될 것이다. sigqueue 구조체는 앞에서 이미 signal에 대한 이야기를 하면서 보았을 것이다. SIG_SLAB_DEBUG은 SLAB_DEBUG_FREE | SLAB_RED_ZONE으로 되어 있는 값이다.

여기서 잠시 kmem_cache_create() 함수에서 사용하고 있는 slab 할당을 위한 flag값을 보기로 하자. 아래의 표와 같다. 정의는 ~/include/linux/slab.h에 있다.⁴²⁹

ID	Value	Description
SLAB_DEBUG_FREE	0x000000100	해제시에 check를 수행하라(이 연산의 cost는 높다.)
SLAB_DEBUG_INITIAL	0x000000200	Constructor를 호출하라(verifier로서 호출한다.)
SLAB_RED_ZONE	0x000000400	Cache의 Red Zone에 있는 object들이다.
SLAB_POISON	0x000000800	Poison object들이다.
SLAB_NO_REAP	0x000001000	Cache로부터 reap를 하지 마라.
SLAB_HWCACHE_ALIGN	0x000002000	Object를 hardware cache에 정렬하라.
SLAB_CACHE_DMA	0x000004000	할당시에 GFP_DMA 메모리를 사용하라(DMA 메모리 공간).

코드 1076. kmem_cache_create() 함수의 호출에 사용하는 flag의 정의

이 표에서 위에서 3가지는 SLAB_DEBUG_SUPPORT가 설정된 경우에만 유효한 값을 가지는 flag이다. SLAB_RED_ZONE과 debugging의 목적으로 사용하고 있는 것만 확인할 수 있으며, SLAB_NO_REAP은 메모리 상황이 좋지 못하더라도, 이 object를 위해서 할당된 cache를 거두어 들이지 말 것을 나타낸다.

17.7.3.24.bdev_init() 함수의 분석

bdev_init() 함수는 block device에서 사용할 block_device구조체의 cache를 생성하는 일을 한다. 파일 시스템의 inode에 대해서 이야기하면서, block_device구조체를 단순히 inode와 관련된 block mode device라고만 보았는데, 이것을 좀더 자세히 보도록 하자. 정의는 ~/include/linux/fs.h에 있다.

⁴²⁹ 이 부분도 이미 앞에서 slab allocator를 소개하면서 한번 본 것이다. 여기서는 기억을 상기하는 것으로 충분하다.

Field	Description
struct list_head bd_hash	Block mode device의 hash리스트
atomic_t bd_count	Block mode device의 usage count.
struct address_space bd_data	Block mode device의 address space(현재는 comment로 처리되어 있음).
dev_t bd_dev	Block mode device의 device ID(search key로 사용된다.)
atomic_t bd_openers	Block mode device를 open한 count.
const struct block_device_operation *bd_op	Block mode device에 대한 operation 구조체의 포인터.
struct semaphore bd_sem	Block mode device에 대한 open/close시에 사용하는 Mutex ⁴³⁰ 필드

표 119. block_device구조체의 정의

OI block_device구조체는 파일 시스템에서 inode의 한 필드로 연결되어 있다. 또한 bd_op에 해당하는 연산은 block mode device에 고유한 연산들로 아래와 같이 정의된다.

```
struct block_device_operations {
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
    int (*check_media_change) (kdev_t);
    int (*revalidate) (kdev_t);
};
```

코드 1077. block_device_operations구조체의 정의

즉, block_device_operations 구조체는 block 모드 디바이스에 적용 가능한 연산들을 정의한 구조체이며, 이러한 연산으로는 open, release, ioctl, check_media_change, revalidate가 있다. 이중에서 마지막에 있는 두개의 연산은 block mode device가 removable하다는 특성에서 현재도 유효한지를 알기 위해서 호출되는 함수들이다. 블록 모드 디바이스에 대한 코드를 보면 이와 관련된 routine을 찾을 수 있을 것이다. Block mode device에 대한 read/write연산은 일반적으로 커널에서 제공하는 default 연산을 사용하기에 이곳에서는 정의하고 있지 않음도 알 수 있을 것이다.

```
void __init bdev_init(void)
{
    int i;
    struct list_head *head = bdev_hashtable;

    i = HASH_SIZE;
    do {
        INIT_LIST_HEAD(head);
        head++;
        i--;
    } while (i);
    bdev_cachep = kmem_cache_create("bdev_cache",
                                    sizeof(struct block_device),
                                    0, SLAB_HWCACHE_ALIGN, init_once,
                                    NULL);
    if (!bdev_cachep)
        panic("Cannot create bdev_cache SLAB cache");
}
```

코드 1078. bdev_init() 함수의 정의

⁴³⁰ Mutual Exclusion을 위한 필드로 한번에 하나의 연산이 접근하도록 하는데 사용한다.

bdev_init() 함수는 ~/fs/block_device.c에 정의되어 있다. 넘겨받는 파라미터는 없으며, block_device 구조체의 hash table을 가르키는 전역 변수로 bdev_hashtable을 사용하고 있다. HASH_SIZE는 HASH_BITS(=6) 만큼 1을 좌측으로 shift한 값(=64) 사용하고 있다. 각각의 hash table의 entry에 대해서 INIT_LIST_HEAD를 호출해서 list 연결 구조를 초기화 하고, bdev_cachep에 해당 cache를 하당하기 위해서 kmem_cache_create() 함수를 호출한다. 이때, constructor 함수로서 init_once() 함수를 넘겨주는데 유의하도록 하자. 이것은 cache의 object를 생성하게 될 때 호출되는 함수이며, 아래와 같은 정의를 가진다.

```
static void init_once(void * foo, kmem_cache_t * cachep, unsigned long flags)
{
    struct block_device * bdev = (struct block_device *) foo;

    if ((flags & (SLAB_CTOR_VERIFY|SLAB_CTOR_CONSTRUCTOR)) ==
        SLAB_CTOR_CONSTRUCTOR)
    {
        memset(bdev, 0, sizeof(*bdev));
        sema_init(&bdev->bd_sem, 1);
    }
}
```

코드 1079. init_once() 함수의 정의

init_once() 함수는 할당되는 block_device 구조체의 cache 영역을 초기화 하는 목적을 가지고 있다. 만약 설정된 constructor가 있다면, 할당할 cache를 0으로 초기화 하고, 세마포어 값을 1로 설정하는 일을 한다.

17.7.3.25.inode_init() 함수의 분석

inode_init() 함수는 커널에서 사용할 inode object에 대한 hash table의 cache를 생성하고, inode object의 cache를 할당하는 일을 한다. 정의는 ~/fs/inode.c에 아래와 같이 되어 있다.

```
void __init inode_init(unsigned long mempages)
{
    struct list_head *head;
    unsigned long order;
    unsigned int nr_hash;
    int i;

    mempages >= (14 - PAGE_SHIFT);
    mempages *= sizeof(struct list_head);
    for (order = 0; ((1UL << order) << PAGE_SHIFT) < mempages; order++)
    ;
    do {
        unsigned long tmp;

        nr_hash = (1UL << order) * PAGE_SIZE /
                  sizeof(struct list_head);
        i_hash_mask = (nr_hash - 1);
        tmp = nr_hash;
        i_hash_shift = 0;
        while ((tmp >= 1UL) != 0UL)
            i_hash_shift++;
        inode_hashtable = (struct list_head *)
            __get_free_pages(GFP_ATOMIC, order);
    } while (inode_hashtable == NULL && --order >= 0);
    printk("Inode-cache hash table entries: %d (order: %ld, %ld bytes)\n",
          nr_hash, order, (PAGE_SIZE << order));
    if (!inode_hashtable)
        panic("Failed to allocate inode hash table\n");
```

```

head = inode_hashtable;
i = nr_hash;
do {
    INIT_LIST_HEAD(head);
    head++;
    i--;
} while (i);
/* inode slab cache */
inode_cachep = kmem_cache_create("inode_cache", sizeof(struct inode),
                                 0, SLAB_HWCACHE_ALIGN, init_once,
                                 NULL);
if (!inode_cachep)
    panic("cannot create inode slab cache");
}

```

코드 1080. inode_init() 함수의 정의

17.7.3.26.ipc_init() 함수의 분석

ipc_init() 함수는 system V계열의 IPC를 지원하기 위한 커널부분을 초기화 하는 일을 한다. 지원하는 IPC의 종류로는 세마포어(semaphore), 메시지 큐, 공유 메모리가 있다. 정의는 ~/ipc/util.c에 아래와 같다.

```

void __init ipc_init (void)
{
    sem_init();
    msg_init();
    shm_init();
    return;
}

```

코드 1081. ipc_init() 함수의 정의

각각의 IPC object를 초기화 하기 위해서 sem_init(), msg_init(), shm_init() 함수를 호출한다. 각각의 함수들은 ~/ipc/sem.c, msg.c, shm.c에 정의되어 있다.

```

void __init sem_init (void)
{
    used_sems = 0;
    ipc_init_ids(&sem_ids, sc_semmni);

#ifndef CONFIG_PROC_FS
    create_proc_read_entry("sysvipc/sem", 0, 0, sysvipc_sem_read_proc, NULL);
#endif
}

```

코드 1082. sem_init() 함수의 정의

sem_init()은 세마포어 IPC에 대해서 사용할 데이터 자료구조를 초기화 하기위해서 ipc_init_ids() 함수를 호출한 이 함수에 대해서는 이미 커널의 IPC에 대해서 이야기 할 때 이미 보았다. 이 함수가 하는 일은 세마포어를 할당하기 위한 자료구조를 생성하고, 초기화 하는 일을 한다.

만약 CONFIG_PROC_FS(proc 파일 시스템을 설정)이 커널 컴파일에서 설정되었다면, 세마포어의 entry를 proc 파일 시스템에 등록하기 위해서 create_proc_read_entry()를 호출하게 되며, 이 함수는 /proc/sysvipc/sem entry를 read만이 가능하도록(read only) 등록할 것이다.

```

void __init msg_init (void)
{
    ipc_init_ids(&msg_ids, msg_ctlmni);
}

```

```
#ifdef CONFIG_PROC_FS
    create_proc_read_entry("sysvipc/msg", 0, 0, sysvipc_msg_read_proc, NULL);
#endif
}
```

코드 1083. msg_init() 함수의 정의

msg_init() 함수도 역시 메시지 큐를 위한 IPC 자료구조를 만들어서 초기화 하는 일을 한다. 역시 CONFIG_PROC_FS가 설정되었다면, /proc/sysvipc/msg를 read only로 생성할 것이다.

```
void __init shm_init(void)
{
    ipc_init_ids(&shm_ids, 1);
    create_proc_read_entry("sysvipc/shm", 0, 0, sysvipc_shm_read_proc, NULL);
}
```

코드 1084. shm_init() 함수의 정의

shm_init() 함수도 마찬가지로 IPC 자료구조를 생성 및 초기화 하고, /proc/sysvipc/shm을 read only로 생성할 것이다. 한가지 차이가 나는 점은 CONFIG_PROC_FS가 설정되지 않아도 생성한다는 점인데, 이것은 커널 버전 2.4.2에서 가지는 오류이며, 커널 버전 2.4.3이상에서는 정정 되었다. 따라서, CONFIG_PROC_FS가 설정된 경우에만 생성한다.

여기서 잠시 create_proc_read_entry() 함수를 보기로 하자. 이것은 앞에서 proc 파일 시스템에 대해서 논의할 때 디렉토리 entry만 생성하는 것까지 보았기에 이곳에서 다루도록 하겠다. 정의는 ~/include/linux/proc_fs.h에 inline함수로 아래와 같이 되어 있다.

```
extern inline struct proc_dir_entry *create_proc_read_entry(const char *name, mode_t mode,
                                                               struct proc_dir_entry *base, read_proc_t *read_proc, void * data)
{
    struct proc_dir_entry *res=create_proc_entry(name,mode,base);
    if (res) {
        res->read_proc=read_proc;
        res->data=data;
    }
    return res;
}
```

코드 1085. create_proc_read_entry() 함수의 정의

넘겨받는 파라미터로, entry에 대한 이름과 접근 허가모드, proc 파일 시스템의 directory entry 및 read 연산을 수행할 함수의 포인터와 데이터 값이다. 먼저 create_proc_entry()를 호출해서 해당 entry를 디렉토리 entry에 추가하고, 이 entry에 대한 read 연산과 데이터를 초기화 한 후, 결과 값을 돌려준다.

```
struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode, struct proc_dir_entry *parent)
{
    struct proc_dir_entry *ent = NULL;
    const char *fn = name;
    int len;

    if (!parent && xlate_proc_name(name, &parent, &fn) != 0)
        goto out;
    len = strlen(fn);
    ent = kmalloc(sizeof(struct proc_dir_entry) + len + 1, GFP_KERNEL);
    if (!ent)
        goto out;
    memset(ent, 0, sizeof(struct proc_dir_entry));
    memcpy(((char *) ent) + sizeof(*ent), fn, len + 1);
```

```

ent->name = ((char *) ent) + sizeof(*ent);
ent->namelen = len;
if (S_ISDIR(mode)) {
    if ((mode & S_IALLUGO) == 0)
        mode |= S_IRUGO | S_IXUGO;
    ent->proc_fops = &proc_dir_operations;
    ent->proc_iops = &proc_dir_inode_operations;
    ent->nlink = 2;
} else {
    if ((mode & S_IFMT) == 0)
        mode |= S_IFREG;
    if ((mode & S_IALLUGO) == 0)
        mode |= S_IRUGO;
    ent->nlink = 1;
}
ent->mode = mode;
proc_register(parent, ent);
out:
return ent;
}

```

코드 1086. create_proc_entry() 함수의 정의

create_proc_entry()는 proc 파일 시스템에 하나의 entry를 생성하는 일을 한다. 넘겨받는 파라미터로, proc 파일 시스템에 등록할 이름과, 접근 허가 모드, 그리고, parent에 대한 포인터이다. 이것을 가지고, proc 파일 시스템에서 해당하는 entry를 찾기 위해 xlate_proc_name() 함수를 호출한다. 만약 찾을 수 없다면, 곧바로 out으로 제어를 옮기도록 한다.

디렉토리 entry를 찾았으므로, 이젠 해당 entry를 생성할 차례이다. 먼저 entry를 위한 메모리를 할당받고(kmalloc()), 이것을 ent가 가르키도록 만든다. 할당받지 못한다면, 다시 out으로 제어는 옮겨갈 것이다. 할당받은 entry를 초기화 하고(memset()), entry의 이름과 이름의 길이를 나타내는 부분을 가져온다. 만약 이것이 디렉토리를 생성하려는 것이라면, 모드를 read와 execute로 설정하며, file operation과 inode operation을 각각 proc 파일 시스템에서 제공하는 연산 구조체로 초기화 한 후, link의 수를 2로 만든다. 즉, 적어도 이 node를 참조하는 것이 두개는 존재한다는 말이다(parent와 ".."가 있기 때문이다.) 만약 디렉토리를 생성하고자 하는 것이 아니라면, 일반 파일을 나타내기 위해서 S_IFREG와 User/Group/Other에 대해서 read-only 접근허가를 나타내는 S_IRUGO를 설정하고, link count를 1로 둔다. 이것을 마치고나면, entry의 모드를 앞에서 설정한 값으로 두고, proc 파일 시스템에 등록하기 위해서 proc_register() 함수를 호출한다. 복귀 값은 할당받은 proc 파일 시스템에 대한 entry이거나 NULL이다.

proc 파일 시스템의 한 entry에 대한 모드를 정할때 사용하는 파일 타입은 ~/include/linux/stat.h에 아래와 같이 정의되어 있다.

ID	Value	Description
S_IFMT	0170000	특수 파일을 나타내는 bit mask와 같은 역할을 한다.
S_IFSOCK	0140000	Socket Interface를 나타낸다.
S_IFLNK	0120000	Link임을 나타낸다.
S_IFREG	0100000	일반 normal 파일임을 나타낸다.
S_IFBLK	0060000	Block device 파일임을 나타낸다.
S_IFDIR	0040000	디렉토리 파일임을 나타낸다.
S_IFCHR	0020000	Character device 파일임을 나타낸다.
S_IFIFO	0010000	FIFO 파일임을 나타낸다.
S_ISUID	0004000	Setuid bit이 설정된 파일임을 나타낸다.
S_ISGID	0002000	Setgid bit이 설정된 파일임을 나타낸다.
S_ISVTX	0001000	Sticky bit이 설정된 파일임을 나타낸다.

표 120. 파일의 타입을 나타내는 상수(constant) 값들

이 값들 이외에 파일에 대한 접근 허가 모드를 나타내는 값들로 아래와 같은 값들이 역시 stat.h에 정의되어 있다.

ID	Value	Description
S_IRWXU	00700	Owner에 대한 write와 execute 권한
S_IRUSR	00400	Owner에 대한 read 권한
S_WUSR	00200	Owner에 대한 write 권한
S_IXUSR	00100	Owner에 대한 execute 권한
S_IRWXG	00070	Group에 대한 write와 execute 권한
S_IRGRP	00040	Group에 대한 read 권한
S_IWGRP	00020	Group에 대한 write 권한
S_IXGRP	00010	Group에 대한 execute 권한
S_IRWXO	00007	Other에 대한 write와 execute 권한
S_IROTH	00004	Other에 대한 read 권한
S_IWOTH	00002	Other에 대한 write 권한
S_IXOTH	00001	Other에 대한 execute 권한

표 121. 접근 허가를 나타내는 상수(constant) 값들

위의 표에서 정의한 값을 조합해서 사용할 경우에는 아래와 같은 것을 사용한다. 이 값들은 모두 커널로 컴파일 될 때⁴³¹ 사용할 수 있는 값들이다.

```
#define S_IRWXUGO      (S_IRWXU|S_IRWXG|S_IRWXO)
#define S_IALLUGO      (S_ISUID|S_ISGID|S_ISVTX|S_IRWXUGO)
#define S_IRUGO        (S_IRUSR|S_IRGRP|S_IROTH)
#define S_IWUGO        (S_IWUSR|S_IWGRP|S_IWOTH)
#define S_IXUGO        (S_IXUSR|S_IXGRP|S_IXOTH)
```

코드 1087. 접근 허가 모드를 나타내는 상수(constant) 값들

따라서, 위의 코드에서 보는 mode는 앞에서 본 표들과 같이 파일의 타입 및 파일에 대한 permission까지를 다 가지고 표현된다.

17.7.3.27.dquot_init_hash() 함수의 분석

dquot_init_hash() 함수는 CONFIG_QUOTA가 설정된 경우에 컴파일 되는 함수이다. 이 함수는 disk quota를 관리하기 위한 hash table을 초기화 하는 역할을 한다. 정의는 ~/fs/dquot.c에 아래와 같이 되어 있다.

```
void __init dquot_init_hash(void)
{
    printk(KERN_NOTICE "VFS: Diskquotas version %s initialized\n", __DQUOT_VERSION__);

    memset(dquot_hash, 0, sizeof(dquot_hash));
    memset((caddr_t)&dqstats, 0, sizeof(dqstats));
}
```

코드 1088. dquot_init_hash() 함수의 정의

⁴³¹ __KERNEL__이 정의된 경우이다.

dquot_init_hash() 함수는 dquot_hash구조체 변수와 dqstats구조체 변수를 0으로 설정하는 일을 수행한다. dquot_hash는 다시 dquot구조체로 정의되며, dqstats는 dqstats구조체로 각각 정의된다. 각각의 구조체의 정의는 ~/include/linux/quota.h에 있다.

```
#define MAX_QUOTA_MESSAGE 75 /* Disk quota 시스템이 생성할 수 있는 최대 메시지의 길이 */
#define DQ_LOCKED    0x01      /* update를 위해서 lock되어 있다.*/
#define DQ_WANT      0x02      /* update가 요구된다.*/
#define DQ_MOD       0x04      /* read이후에 disk quota가 변경되었다.*/
#define DQ_BLKS      0x10      /* Block limit에 대해서 uid/gid가 경고 받았다.*/
#define DQ_INODES    0x20      /* Inode limit에 대해서 uid/gid가 경고 받았다.*/
#define DQ_FAKE     0x40      /* Disk quota에 대해서 한계(limit)가 없다.*/

struct dquot {
    struct dquot *dq_next;           /* 다음 dquot구조체에 대한 포인터 */
    struct dquot **dq_pprev;         /* 앞서 있는 dquot구조체에 대한 포인터 */
    struct list_head dq_free;        /* dq_free : free list의 연결 구조체 */
    struct dquot *dq_hash_next;      /* dquot_hash구조체의 hash 연결 구조체를 위한 포인터 */
    struct dquot **dq_hash_pprev;    /* dquot_hash구조체의 hash 연결 구조체를 위한 포인터 */
    wait_queue_head_t dq_wait;       /* 이 구조체의 wait queue */
    int dq_count;                   /* 이 구조체의 reference count */
    /* fields after this point are cleared when invalidating */
    struct super_block *dq_sb;      /* 이 구조체가 적용되는 superblock object */
    unsigned int dq_id;             /* 이 구조체가 적용되는 UID와 GID */
    kdev_t dq_dev;                 /* 이 구조체가 적용되는 block device */
    short dq_type;                  /* Quota의 type */
    short dq_flags;                 /* Disk quota의 flag 정의 --> 앞에 있는 것을 참조 */
    unsigned long dq_referenced;    /* 이 구조체가 사용되는 동안 reference된 count */
    struct dqblk dq_dqb;           /* dqblk구조체 : 디스크 quota의 사용 현황을 보여준다. */
};

};
```

코드 1089. dquot구조체의 정의

앞에서 이 구조체를 0으로 초기화해 주었다. 즉, 이 구조체로 선언된 dquot_hash[] 배열은 NR_DQHASH(=43) 만큼의 크기를 가지는 hash table과 같은 역할을 하는 것으로, 초기화 시켜주는 일을 한 것이다. 실제적인 disk quota에 대한 설정과 현재의 사용은 dqblk구조체가 나타내고 있다.

```
/*
 * The following structure defines the format of the disk quota file
 * (as it appears on disk) - the file is an array of these structures
 * indexed by user or group number.
 */
struct dqblk {
    __u32 dqb_bhardslimit;          /* Disk block의 할당에 대한 절대 한계(absolute limit) : hard limit */
    __u32 dqb_bsoftlimit;           /* Disk block의 할당에 대한 선호되는 한계(preferred limit): soft limit */
    __u32 dqb_curblocks;            /* 현재 할당된 block의 count */
    __u32 dqb_ihardlimit;           /* 할당된 inode들에 대한 절대 한계(absolute limit) : hard limit */
    __u32 dqb_isoftlimit;           /* Inode의 할당에 대한 선호되는 한계(preferred limit) : soft limit */
    __u32 dqb_curnodes;             /* 현재 할당된 inode들의 개수(혹은 파일의 개수) */
    time_t dqb_btime;               /* 초과되는 디스크 사용에 대한 시간적인 한계 */
    time_t dqb_itime;               /* 초과되는 inode 사용에 대한 시간적인 한계 */
};
```

코드 1090. dqblk구조체의 정의

여기서 한가지 개념을 더 보도록 하자. 즉, dqblk구조체에 정의된 hard limit와 soft limit에 대한 것이다. 각각은 아래와 같이 정의해 볼 수 있다.

- Hard limit(Absolute limit) : 이 한계에 도달하면, 더 이상의 새로운 파일이나 block의 할당에 대해서 커널은 거부(deny)할 것이다.
- Soft limit(Preferred limit) : 이 한계는 일반적으로 hard limit보다는 작은 값을 가진다. 이 한계에 도달하게 되면, 새로운 블록의 할당에 대해서 warning message가 커널에 의해 표시될 것이다. 또한 이 한계에 도달한 시간은 저장된다. 어느 정도의 시간이 흐른뒤에⁴³² Linux 시스템은 사용자가 absolute limit에 도달한 것처럼 생각해서, 더 이상의 자원 할당에 대해서 거부(deny)할 것이다. 따라서, 사용자는 soft limit이하로 낮추어 위해서 디스크의 공간을 차지하고 있는 일부 파일들을 지워야만 할 것이다.

이러한 disk quota를 두는 것은 Linux 시스템이 multi-user를 지원하기 때문에, 어느 일부 사용자에게만 자원의 사용이 집중되는 것을 관리하기 위한 방법이다. 혹은 사용자의 권한이나 특권 레벨에 따라서, 자원 사용의 등급을 주기 위한 것이다.

```
struct dqstats {
    __u32 lookups;                      /* lookup0이 발생한 회수 */
    __u32 drops;                        /* drop0이 발생한 회수 */
    __u32 reads;                        /* read가 발생한 회수 */
    __u32 writes;                       /* write가 발생한 회수 */
    __u32 cache_hits;                  /* cache hit이 발생한 회수 */
    __u32 allocated_dquots;            /* 할당된 disk quota의 양 */
    __u32 free_dquots;                 /* 해제된 disk quota의 양 */
    __u32 syncs;                        /* sync가 발생한 회수 */
};
```

코드 1091. dqstats 구조체의 정의

dqstats는 디스크에 대해서 발생한 각종 연산에 대한 정보를 담는 구조체이다. 이것은 나중에 disk quota를 관리하는 시스템 프로그램에 의해서 읽혀져서 사용될 것이다⁴³³.

```
check_bugs();
printk("POSIX conformance testing by UNIFIX\n");
/*
 *      We count on the initial thread going ok
 *      Like idlers init is an unlocked kernel thread, which will
 *      make syscalls (and thus be locked).
 */
smp_init();
kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
unlock_kernel();
current->need_resched = 1;
cpu_idle();
```

코드 1092. start+kernel() 함수의 정의(계속)

이제 start_kernel() 함수에 대한 이야기가 많이 남지 않았으므로, 우리가 현재 어디까지 왔는지를 알기위해서 잠시 남은 코드를 보기로 하자. 아래와 같다. SMP machine일 경우에 smp_init()를 호출해서

⁴³² 대략적으로 default값 7일 정도를 두고 있다고 한다.

⁴³³ quota와 같은 명령을 사용하면, 현재 사용자에 대한 disk quota양을 확인할 수 있을 것이다.

초기화 시키는 일과 시스템의 초기 프로세스(init process)⁴³⁴ 인 kernel thread를 생성하는 것, 앞에서 진입시에 했던 kernel에 설정된 lock을 해제하는 것과 rescheduling을 요구하는 것, 그리고, 마지막으로 idle process로 만들어주는 일이 start_kernel() 함수의 분석에서 남은 것이다.

17.7.3.28.check_bugs() 함수의 분석

check_bugs() 함수는 일반적으로 정의되는 함수가 아니라, 매크로로 정의된 함수이다. 이것은 프로세서(processor)마다 고유한 자신만의 에러를 찾는 함수를 define해서 사용할 수 있도록 만들어 놓은 것이다. check_bugs() 함수는 multiprocessor와 single processor에서 각각 정의를 가지는데, 각각에 해당하는 정의는 ~/include/arm/proc-fns.h에 있는 정의를 따른다. 즉, single processor인 경우에는 ~/include/arm/cpu-single.h를 사용할 것이며, 그렇지 않은 경우에는 ~/include/arm/cpu-multi26(or 32).h를 사용하게 될 것이다. 일단 single processor를 사용하는 경우를 보도록 하자.

```
#ifdef __STDC__
#define __cpu_fn(name,x)    cpu_##name##x
#else
#define __cpu_fn(name,x)    cpu_/**/name/**/x
#endif
#define cpu_fn(name,x)      __cpu_fn(name,x)
...
#define cpu_data_abort          cpu_fn(CPU_NAME,_data_abort)
#define cpu_check_bugs         cpu_fn(CPU_NAME,_check_bugs)
#define cpu_proc_init          cpu_fn(CPU_NAME,_proc_init)
#define cpu_proc_fin           cpu_fn(CPU_NAME,_proc_fin)
#define cpu_reset              cpu_fn(CPU_NAME,_reset)
#define cpu_do_idle            cpu_fn(CPU_NAME,_do_idle)

#define cpu_cache_clean_invalidate_all   cpu_fn(CPU_NAME,_cache_clean_invalidate_all)
#define cpu_cache_clean_invalidate_range cpu_fn(CPU_NAME,_cache_clean_invalidate_range)
#define cpu_flush_ram_page           cpu_fn(CPU_NAME,_flush_ram_page)

#define cpu_dcache_invalidate_range  cpu_fn(CPU_NAME,_dcache_invalidate_range)
#define cpu_dcache_clean_range       cpu_fn(CPU_NAME,_dcache_clean_range)
#define cpu_dcache_clean_page       cpu_fn(CPU_NAME,_dcache_clean_page)
#define cpu_dcache_clean_entry      cpu_fn(CPU_NAME,_dcache_clean_entry)

#define cpu_icache_invalidate_range cpu_fn(CPU_NAME,_icache_invalidate_range)
#define cpu_icache_invalidate_page  cpu_fn(CPU_NAME,_icache_invalidate_page)

#define cpu_tlb_invalidate_all     cpu_fn(CPU_NAME,_tlb_invalidate_all)
#define cpu_tlb_invalidate_range  cpu_fn(CPU_NAME,_tlb_invalidate_range)
#define cpu_tlb_invalidate_page   cpu_fn(CPU_NAME,_tlb_invalidate_page)

#define cpu_set_pgd               cpu_fn(CPU_NAME,_set_pgd)
#define cpu_set_pmd               cpu_fn(CPU_NAME,_set_pmd)
#define cpu_set_pte               cpu_fn(CPU_NAME,_set_pte)
```

코드 1093. single processor를 사용하는 경우의 CPU에 의존적인 함수들에 대한 정의

즉, __cpu_fn() 매크로에 의해서 각각의 cpu_xxx_xxx() 함수들은 CPU_NAME과 cpu_CPU_NAME_xxx_xxx string을 더한 함수로 정의된다. 따라서, 우리가 보고자 하는 함수는 CPU_NAME이 sa1100이 될 것이므로, sa1100_check_bugs() 함수가 될 것이다. 이 함수는 다시 ~/arch/arm/mm/proc-sa110.S에 아래와 같은 정의를 가진다.

⁴³⁴ 이 init process는 커널 thread로 생성되는 것이며, 나중에 파일 시스템에 있는 init 프로그램을 다시 생성하는 것 부분을 볼 수 있을 것이다. 여기서 생성하는 커널 init thread와 뒤에서 생성하는 init process를 구분해야 한다.

```

...
/*
 * SA1100 and SA1110 share the same function calls
 */
.type    sa1100_processor_functions, #object
ENTRY(sa1100_processor_functions)
    .word    cpu_sa1100_data_abort
    .word    cpu_sa1100_check_bugs
    .word    cpu_sa1100_proc_init
    .word    cpu_sa1100_proc_fin
    .word    cpu_sa1100_reset
    .word    cpu_sa1100_do_idle
...

```

코드 1094. CPU에 의존적인 함수들의 정의

따라서, `cpu_sa1100_check_bugs()`가 우리가 찾으려는 함수임을 알 수 있으며, 정의는 다음과 같이 `~/arch/arm/mm/proc-sa110.S`에 나와 있다.

```

...
/*
 * cpu_sa110_check_bugs()
 */
ENTRY(cpu_sa110_check_bugs)
ENTRY(cpu_sa1100_check_bugs)
    mrs      ip435, cpsr
    bic      ip, ip, #F_BIT
    msr      cpsr, ip
    mov      pc, lr
...

```

코드 1095. `cpu_sa1100_check_bugs()`의 정의

이곳에서 보면, 일단 `cpu_sa110_check_bugs()` 함수와 같은 코드 사용한다는 것을 알 수 있을 것이다. 즉, SA1110과 SA1100이 같은 함수를 사용하도록 정의되어 있다. 먼저 `cpsr`를 읽어서 `ip register`로 읽어오고, 여기서 `F_BIT`를 clear한 다음 다시 `cpsr`에 적는다. 그리고나서, 곧바로 `lr`에 `pc`를 넣어서 `jump`연산을 한다.

만약 `multiprocessor`로 설정되었다면, 이야기는 조금 달라진다. 먼저 `~/include/asm-arm/cpu-multi32.h`를 보도록 하자.

```

extern const struct processor arm6_processor_functions;
extern const struct processor arm7_processor_functions;
extern const struct processor sa110_processor_functions;

#define cpu_data_abort(pc)           processor._data_abort(pc)
#define cpu_check_bugs()            processor._check_bugs()
#define cpu_proc_init()              processor._proc_init()
#define cpu_proc_fin()               processor._proc_fin()
#define cpu_reset(addr)              processor.reset(addr)
#define cpu_do_idle(mode)            processor._do_idle(mode)

#define cpu_cache_clean_invalidate_all() processor.cache.clean_invalidate_all()
#define cpu_cache_clean_invalidate_range(s,e,f) processor.cache.clean_invalidate_range(s,e,f)
#define cpu_flush_ram_page(vp)       processor.cache._flush_ram_page(vp)

```

⁴³⁵ 여기서 `ip register`은 `r12` 레지스터를 의미한다.

```
#define cpu_dcache_clean_page(vp) processor.dcache.clean_page(vp)
#define cpu_dcache_clean_entry(addr) processor.dcache.clean_entry(addr)
#define cpu_dcache_clean_range(s,e) processor.dcache.clean_range(s,e)
#define cpu_dcache_invalidate_range(s,e) processor.dcache.invalidate_range(s,e)

#define cpu_icache_invalidate_range(s,e) processor.icache.invalidate_range(s,e)
#define cpu_icache_invalidate_page(vp) processor.icache.invalidate_page(vp)

#define cpu_tlb_invalidate_all() processor.tlb.invalidate_all()
#define cpu_tlb_invalidate_range(s,e) processor.tlb.invalidate_range(s,e)
#define cpu_tlb_invalidate_page(vp,f) processor.tlb.invalidate_page(vp,f)

#define cpu_set_pgd(pgd) processor.pgtable.set_pgd(pgd)
#define cpu_set_pmd(pmdp, pmd) processor.pgtable.set_pmd(pmdp, pmd)
#define cpu_set_pte(ptep, pte) processor.pgtable.set_pte(ptep, pte)

#define cpu_switch_mm(pgd,tsk) cpu_set_pgd(__virt_to_phys((unsigned long)(pgd)))
```

코드 1096. Multiprocessor인 경우의 CPU 의존적인 함수들에 대한 정의

즉, `check_bugs()` 함수는 `cpu_check_bugs()`를 매크로로 정의하고 있는 것이되며⁴³⁶, 다시 `cpu_check_bugs()` 함수는 위에서와 같이 `processor._check_bugs()` 함수로 재 정의될 것이다. 여기서 잠시 `processor` 구조체를 보기로 하자. 정의는 `~/include/asm-arm/cpu-multi32(or 26).h`에 있다.⁴³⁷

```
extern struct processor {
    void (*_data_abort)(unsigned long pc);
    void (*_check_bugs)(void);
    void (*_proc_init)(void);
    void (*_proc_fin)(void);
    volatile void (*reset)(unsigned long addr);
    int (*_do_idle)(int mode);
    struct { /* CACHE */
        void (*clean_invalidate_all)(void);
        void (*clean_invalidate_range)(unsigned long address, unsigned long end, int flags);
        void (*flush_ram_page)(void *virt_page);
    } cache;
    struct { /* D-cache */
        void (*invalidate_range)(unsigned long start, unsigned long end);
        void (*clean_range)(unsigned long start, unsigned long end);
        void (*clean_page)(void *virt_page);
        void (*clean_entry)(unsigned long start);
    } dcache;
    struct { /* I-cache */
        void (*invalidate_range)(unsigned long start, unsigned long end);
        void (*invalidate_page)(void *virt_page);
    } icache;
    struct { /* TLB */
        void (*invalidate_all)(void);
        void (*invalidate_range)(unsigned long address, unsigned long end);
        void (*invalidate_page)(unsigned long address, int flags);
    } tlb;
    struct { /* PageTable */
        void (*set_pgd)(unsigned long pgd_phys);
    }
```

⁴³⁶ 이것은 `~/include/asm-arm/bugs.h`에서 찾을 수 있을 것이다.

⁴³⁷ 이곳에서는 32bit processor에 대해서만 보기로 한다.

```

        void (*set_pmd)(pmd_t *pmdp, pmd_t pmd);
        void (*set_pte)(pte_t *ptep, pte_t pte);
    } pgtable;
} processor;

```

코드 1097. processor구조체의 정의

이처럼 processor구조체는 CPU마다 의존적인 특성에 기인한 함수들을 정의하고 있는 구조체에 불과하다. 이곳에서 정의되는 함수로는 data abort bug check, processor의 initialize 및 finalize, reset, idle, cache management, D/I cache management, TLB(Translation Lookaside Buffer) management, page table 설정등이 있다. 따라서, 우리가 찾는 함수는 ARM7 architecture에 기초한 arm7_processor_functions의 필드가 될 것이다. 이 processor의 구조체 정의는 proc-arm6,7.S에 아래와 같이 정의되어 있다.

```

...
/*
 * Purpose : Function pointers used to access above functions - all calls
 *           come through these
 */
.type    arm7_processor_functions, #object
ENTRY(arm7_processor_functions)
    .word    cpu_arm7_data_abort
    .word    cpu_arm7_check_bugs
    .word    cpu_arm7_proc_init
    .word    cpu_arm7_proc_fin
    .word    cpu_arm7_reset
    .word    cpu_arm7_do_idle
...

```

코드 1098. arm7_processor_functions구조체의 정의

따라서, 우리가 찾고 있는 함수는 cpu_arm7_check_bugs() 함수가 된다. 이 함수도 같은 파일에 아래와 같은 정의를 가지고 있다.

```

...
ENTRY(cpu_arm6_check_bugs)
ENTRY(cpu_arm7_check_bugs)
    mrs    ip, cpsr
    bic    ip, ip, #F_BIT
    msr    cpsr, ip
    mov    pc, lr
...

```

코드 1099. cpu_arm7_check_bugs() 함수의 정의

함수에서 보듯이 앞에서 본 single processor의 경우에 해당하는 check_bugs() 함수와 동일함을 알 수 있을 것이다. 단순히 cpsr을 ip register로 읽어서 F_BIT를 clear시킨 후, 다시 이 값을 cpsr에 저장한다. 그리고 나서, 곧바로 return하게 된다. 이곳에서도 ARM6와 ARM7에 대해서 같은 함수를 사용하고 있음을 볼 수 있을 것이다. 이곳에서 자세히 보지 않은 나머지 함수들도 각각 해당하는 일을 필요로 할 경우에는 커널에서 호출될 것이다. 나중에 관련된 함수들의 호출이 있다면, 자세히 보도록 하자. 여기서는 이 정도에서 만족하도록 하자.

17.7.3.29.smp_init() 함수의 분석

smp_init() 함수는 커널의 CONFIG_SMP가 정의되지 않은 경우에는 아무런 일을 하지 않는 do{} while(0) loop로 정의된다. 만약 CONFIG_SMP가 설정되었다면. 아래와 같이 정의될 수 있을 것이다. 코드는 ~/init/main.c에 있다.

```
/* Called by boot processor to activate the rest. */
static void __init smp_init(void)
{
    /* Get other processors into their bootup holding patterns. */
    smp_boot_cpus();
    smp_threads_ready=1;
    smp_commence();
}
```

코드 1100. smp_init() 함수의 정의

smp_init() 함수는 SMP(Symmetric Multi-Processing) machine에서 boot를 담당하는 프로세스에 의해서만 호출된다. 이 함수를 호출함으로써, 다른 processor들을 기동(boot) 시켜주게 되는 것이다. 여기서는 smp_boot_cpus() 함수를 호출해서 각각의 프로세스를 booting시켜주고, smp_threads_ready flag를 1로 설정해서, SMP가 수행될 준비가 끝났음을 나타내게 되며, 최종적으로 SMP machine을 시작시키는 것은 smp_commence() 함수가 된다.⁴³⁸

17.7.3.30.kernel_thread() 함수의 분석

kernel_thread() 함수는 커널에서 사용할 thread를 만드는 역할을 한다. 단순히 하나의 function을 하나의 task_struct 구조와 연결시켜서 생성한다.

```
/* 여기에서 smp_init() 함수를 호출한다.*/
kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
unlock_kernel();
current->need_resched = 1;
cpu_idle();
}
```

코드 1101. start_kernel() 함수의 정의(계속)

kernel_thread() 함수가 넘겨받는 argument값으로는, thread로 실행할 커널의 일부 함수의 이름과, 그 함수의 수행에 필요한 argument를 가지는 포인터, 그 함수가 생성될 때 사용할 flag이 있다. 정의는 ~/arch/arm/kernel/process.c에 아래와 같이 정의된다.

```
pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
{
    pid_t __ret;

    __asm__ __volatile__(
        "orr    r0, %1, %2          @ kernel_thread sys_clone
        mov    r1, #0
        __syscall(clone)"
        "movs   %0, r0              @ if we are the child
        bne   1f
        mov    fp, #0              @ ensure that fp is zero
        mov    r0, %4
        mov    lr, pc
        mov    pc, %3
        b     sys_exit
1:   "
        : "=r" (__ret)
        : "Ir" (flags), "I" (CLONE_VM), "r" (fn), "r" (arg)
        : "r0", "r1", "lr");
```

⁴³⁸ SMP에 대한 것은 Linux의 커널 부분에서 따로 논의하겠다. 이곳에서는 단지 SMP를 완전히 기동하는 시점만을 기억하기 바란다.

```
    return __ret;
}
```

코드 1102. kernel_thread() 함수의 정의

kernel_thread() 함수는 오직 커널의 프로세스만을 생성하기 위해서 사용한다. 먼저, 전달 받은 flags와 CLONE_VM을 OR시켜서 r0에 둔다. r1에는 0을 두고, system call인 clone()을 호출한다. 결과 값을 다시 __ret에 넣고(movs %0, r0), 만약 이 값이 0이 아니라면, lr로 제어를 옮긴다. fp는 frame pointer를 나타내는 register이며 r11이 사용된다. fp register에 0을 넣어서 새로운 프로세스의 수행을 위해서 대비한 다음, r0에 %4를 주어 thread로 생성되는 프로세스의 argument로 넘어가도록 만든다. 마지막으로 프로세스의 수행이 끝난 뒤에 return한 주소를 기억하기 위해서 lr에는 pc를 준다. 그런 후, fn(=%3)을 pc로 주어서 수행을 시작하게 된다. 복귀하면, “b sys_exit”이 수행될 것이다. 이것은 sys_exit() 함수를 호출할 것이며, 프로세스(혹은 이곳에서는 thread)를 제거하는 일을 한다.

__syscall()은 실제로는 system call 번호로 swi(software interrupt)를 발생시켜주는 일을 한다. 정의는 아래와 같다.

```
#define __NR_SYSCALL_BASE      0x9000000
...
#define __NR_clone                (__NR_SYSCALL_BASE+120)
...
#define __sys2(x) #x
#define __sys1(x) __sys2(x)

#ifndef __syscall
#define __syscall(name) "swi\t" __sys1(__NR_##name) "\n\t"
#endif
...
```

코드 1103. __syscall 매크로의 정의

즉, 위의 코드를 보면, __syscall(clone)은 “swi __sys1(__NR_clone)”이 되고, 다시 “swi __NR_SYSCALL_BASE + 120”이 되며, 결국 “swi 0x900078”이 된다. 여기서 0x900078에 해당하는 함수가 sys_clone() 함수가 된다. 정의는 ~/arch/arm/kernel/sys_arm.c에 아래와 같다.

```
asmlinkage int sys_clone(unsigned long clone_flags, unsigned long newsp, struct pt_regs *regs)
{
    if (!newsp)
        newsp = regs->ARM_sp;
    return do_fork(clone_flags, newsp, regs, 0);
}
```

코드 1104. sys_clone() 함수의 정의

sys_clone()함수는 앞에서 r0에 clone flag를 넘겨주었고, 다시 r1에는 0을 주었다. 나머지는 stack frame으로부터 받는 내용이다. 따라서, newsp값은 0이 되므로 newsp는 regs->ARM_sp(stack pointer)를 가지도록 만든다. 마지막으로 실제로 프로세스를 생성하는 do_fork()를 호출해서 프로세스 구조체(task_struct)를 생성하고, 각종 정보로 task_struct 구조체를 만든다.

복귀 값은 do_fork() 함수의 return 값이 될 것이다. 따라서, do_fork() 함수의 복귀 값이 kernel_thread() 함수에서 r0를 차지하게 될 것이다. r0로 값이 바로 프로세스의 PID역할을 한다. 만약 parent 프로세스라면, 이 값은 0이 아닐 것이므로 바로 복귀(return)하게 될 것이며, child process라면 이 값은 0이 되고, child 프로세스가 실행을 시작할 것이다. 수행을 마치면, 앞에서 설명했듯이 “b sys_exit”가 실행되어 프로세스는 종료될 것이다.

이곳에서 수행되는 커널의 thread는 init() 함수이다. 이것에 대해서는 start_kernel() 함수를 끝까지 다보고 난 다음에 다시 논의하도록 하겠다.

17.7.3.31.cpu_idle() 함수의 분석

cpu_idle() 함수는 최종적으로 현재 boot process가 실행하는 함수이다. 이 함수에서 하는 일은 단순히 시스템에 어떤 event가 발생하기를 기다리는 것이다.

```
/* init kernel thread를 여기서 생성했다.*/
unlock_kernel();
current->need_resched = 1;
cpu_idle();
}
```

코드 1105. start_kernel() 함수의 정의(계속)

먼저, cpu_idle()함수를 수행하기에 앞서서, unlock_kernel() 함수를 호출해서 이전에 start_kernel() 함수로 진입하면서 설정했던 커널에 대한 lock를 해제한다. 그리고, 다른 실행할 프로세스가 있으면 실행시켜주기 위해서 현재 process의 need_resched필드를 1로 설정한다. 이것은 나중에 커널에서 사용자 모드로의 전환이 있을 때, 스케줄러를 동원해서 새롭게 실행할 프로세스를 찾아서, 제어를 그 프로세스로 넘겨줄 것이다. 마지막으로 cpu_idle()함수를 호출한다. cpu_idle() 함수의 정의는 ~/arch/arm/kernel/process.c에 아래와 같이 되어 있다.

```
void cpu_idle(void)
{
    /* endless idle loop with no priority at all */
    init_idle();
    current->nice = 20;
    current->counter = -100;

    while (1) {
        void (*idle)(void) = pm_idle;
        if (!idle)
            idle = arch_idle;
        leds_event(led_idle_start);
        while (!current->need_resched)
            idle();
        leds_event(led_idle_end);
        schedule();
#ifndef CONFIG_NO_PGT_CACHE
        check_pgt_cache();
#endif
    }
}
```

코드 1106. cpu_idle() 함수의 정의

init_idle() 함수를 호출해서 idle 프로세스에 대한 환경을 만들어주고, 현재 프로세스의 nice값⁴³⁹을 20으로, 스케줄링 알고리즘에서 우선순위를 계산하는 counter값을 -100으로 설정한다. 이전 무한 loop를 돌면서 시스템에 event가 발생하기를 기다리게 된다(while(1)). pm_idle() 함수는 power management를 위한 idle()

⁴³⁹ nice값은 nice() 시스템 콜로 변경이 가능하다. 이 값은 한 프로세스가 다른 프로세스에 우선 순위를 주겠다는 의미로 사용하기에 nice라는 이름을 가지게 되었다. 시스템 프로세스의 경우에는 우선 순위 높이는데 사용할 수 있지만, 일반적인 사용자 프로세스는 자신의 우선 순위를 낮추기 위해서만 사용할 수 있다.

함수를 정의하는 것으로, 만약 이것이 정의되지 않았다면⁴⁴⁰, `arch_idle()` 함수를 사용하도록 한다. SA1100의 경우에는 이 함수가 아래와 같이 `~/include/asm-arm/arch-sa1100/system.h`에 정의되어 있다. 앞에서 잠시본 `cpu_do_idle()` 함수를 호출하는 것을 알 수 있을 것이다.

Assabet board와 같은 것을 사용한다면, `~/arch/arm/mach-sa1100/leds.c`에 `assabet_leds_event()` 함수가 정의되어 있다. 이 함수가 하는 일은 시스템에 연결된 LED의 색깔을 바꾸는 일을 한다. 즉, idle 상태에서 계속 수행되고 있다면, 이것을 나타내기 위해서 RED 색을 사용할 것이며, idle 모드에서 깨어나게 되면, RED색을 없애줄 것이다.

`need_resched` 플랙은 현재 프로세스가 스케줄러에게 새로운 프로세스를 스케줄링해줄 것을 요구하는 것이다. 즉, 시스템의 event에 따라서, 현재 실행 중인 idle process의 `need_resched` 플랙에 영향을 미치게 되어, 새로운 프로세스를 수행하기 위해서 시스템은 깨어나게 된다. 이것을 가능하게 하는 것이 `schedule()` 함수를 호출하는 것이다. `check_pgt_cache()` 함수는 현재 page table에 대한 cache의 크기가 설정된 water mark 값보다 많은 경우에 page table의 크기를 줄일 목적으로 사용하는 함수이다. 이제는 앞에서 설명하지 않은 함수들에 대해서 간략히 보도록하자.

```
static inline void arch_idle(void)
{
    if (!hlt_counter)
        cpu_do_idle(0);
}
```

코드 1107. `arch_idle()` 함수의 정의

`hlt_counter`는 `halt counter`로 시스템을 정의는 `~/arch/arm/kernel/process.c`에 있다. 이 값으로 시스템을 halt시킬지를 결정하게 된다. 만약 이 값이 0이라면, `cpu_do_idle()` 함수를 수행한다. `cpu_do_idle()` 함수는 다시 `~/arch/arm/mm/proc-sa110.S`에 아래와 같이 `cpu_sa110_do_idle()` 함수로 정의된다.⁴⁴¹

```
...
ENTRY(cpu_sa1100_do_idle)
    mov    r0, r0                      @ 4 nop padding
    mov    r0, r0
    mov    r0, r0
    mov    r0, #0
    ldr    r1, =UNCACHEABLE_ADDR      @ ptr to uncachable address
    mrs    r2, cpsr
    orr    r3, r2, #192               @ disallow interrupts
    msr    cpsr_c, r3
    @ --- aligned to a cache line
    mcr    p15, 0, r0, c15, c2, 2    @ disable clock switching
    ldr    r1, [r1, #0]                @ force switch to MCLK
    mcr    p15, 0, r0, c15, c8, 2    @ wait for interrupt
    mov    r0, r0                      @ safety
    mcr    p15, 0, r0, c15, c1, 2    @ enable clock switching
    msr    cpsr_c, r2                @ allow interrupts
    mov    pc, lr
...
```

코드 1108. `cpu_sa1100_do_idle()` 함수의 정의

Multiprocessor의 경우에는 `~/arch/arm/mm/proc-arm6,7.S`에 아래와 같은 정의를 가지고 있다. ARM6와 ARM7에 대해서 동일한 `do_idle()`함수를 정의하고 있다. `r0`에 계속적으로 `r0`값을 다시 넣다가⁴⁴², 다시 `r0`에

⁴⁴⁰ 현재(커널 버전 2.4.3)로서는 power management관련 idle함수는 정의되어 있지 않다.

⁴⁴¹ Single CPU인 경우이며, 그렇지 않다면 `proc-arm6,7.S`에 `cpu_arm6(or 7)_do_idle()` 함수가 될 것이다.

⁴⁴² 아무런 의미가 없는 연산이다(NOP).

0을 넣고, r1에 cache가 되지 않는 주소를 가르키도록 만든 후, r2에는 cpsr값을 읽어온다. 이것에 192(=0xC0)을 OR시켜서 다시 cpsr의 control bit부분에 채워준다. 이것으로 interrupt가 발생하지 않도록 만든다. UNCACHEABLE_ADDR은 ~/include/asm-arm/arch-sa1100/hardware.h에 0xFA050000으로 정의되어 있다. 보조 프로세스 p15의 register 15(=c15)에 있는 c2(=CRm)과 2(=OPC_2)를 사용해서 r0에 있는 값(=0)으로 써주어었다. 이렇게 함으로써 clock switching을 disable시킨다. 이젠 r1이 가르키는 곳에서 r1값을 다시 읽어오도록 한다. 이것으로 MCLK(Memory Clock)가 가동 되도록 만들고⁴⁴³, interrupt가 발생하기를 기다리기 위해서 p15의 c15레지스터의 c8(=CRm)과 2(=OPC_2)를 사용해서 r0를 넣도록 한다. 다시 r0에는 r0를 넣고(NOP), 이젠 다시 clock switching을 enable시켜준다. Clock switching을 enable시켜주기 위해서는 p15의 c15에 c1(=CRm)과 2(OPC_2)를 써서, r0을 쓰도록 한다. 그리고, 원래 가지고 있었던 cpsr값으로 cpsr를 바꿔주고(r2), 함수를 return한다(mov pc, lr). 어쨌든 종합해서 이야기 하면, 이곳에서도 결국 아무런 일을 하지 않는 것에 불과하다.

```
...
ENTRY(cpu_arm6_do_idle)
ENTRY(cpu_arm7_do_idle)
    mov     r0, #-EINVAL
    mov     pc, lr
...
```

코드 1109. cpu_arm6(or 7)_do_idle() 함수의 정의

cpu_arm6(or 7)_do_idle() 함수는 단순히 r0(return값)에 -EINVAL(=22)를 넣고, 다시 복귀한다(mov pc, lr). 아무 일도 일어나지 않는다.

```
void __init init_idle(void)
{
    struct schedule_data * sched_data;
    sched_data = &aligned_data[smp_processor_id()].schedule_data;

    if (current != &init_task && task_on_rqueue(current)) {
        printk("UGH! (%d:%d) was on the runqueue, removing.\n",
               smp_processor_id(), current->pid);
        del_from_rqueue(current);
    }
    sched_data->curr = current;
    sched_data->last_schedule = get_cycles();
}
```

코드 1110. init_idle() 함수의 정의

init_idle() 함수의 정의는 ~/kernel/sched.c에 있으며, idle 프로세스를 위한 초기화 환경을 만들어주는 역할을 한다. 먼저, 현재의 프로세스를 위한 스케줄링 데이터를 읽어서 이를 sched_data 변수가 가르키도록 만들어 준다. 만약 현재의 프로세스(current)가 init_task와 다르고, 현재의 task⁴⁴⁴가 있는 run queue의 연결 리스트 다음에 task가 있다면, 현재의 프로세스를 run queue에서 제거한다(del_from_rqueue()). 여기서 smp_processor_id() 함수는 현재의 프로세스의 ID값을 돌려주며, single processor라면 0이 항상 나올 것이다. del_from_rqueue() 매크로는 run queue에 있는 실행준비된 프로세스의 수를 감소시키고, 넘겨준 프로세스의 sleep time을 설정한 후, run queue에서 프로세스를 제거하는 일을 한다.⁴⁴⁵ get_cycles() 함수는 inline을 정의된 함수로서 ARM에서는 항상 0을 돌려주는 일을

⁴⁴³ 이것은 인위적으로 DCLK(Actual Core Clock)이 MCLK(Memory Clock)과 같도록 만드는 방법으로, 먼저 clock switching을 disable한 다음, cache miss가 발생하도록 만들어주는 것으로 가능하다. MCLK은 DCLK의 절반정도의 frequency를 가지고 동작한다.

⁴⁴⁴ Linux에서는 task라는 말로 process를 표현하고 있다.

⁴⁴⁵ 즉, 프로세스(혹은 task)는 sleeping 상태가 될 것이다. Run queue에서는 제거가 되었지만, 전체 프로세스를 관리하는 리스트에서는 제거되지 않았다.

한다. 따라서, 이곳에서는 스케줄링 데이터 구조체의 curr필드를 현재 프로세스의 task_struct로 두고, last_schedule 필드를 0으로 설정한다.

schedule_data 구조체의 정의는 ~/kernel/sched.c에 아래와 같이 정의되어 있다. 이 구조체는 CPU별로 하나의 entry를 가지도록하는 cache line에 정렬된 aligned_data를 정의하는데 사용한다.

```
static union {
    struct schedule_data {
        struct task_struct * curr;
        cycles_t last_schedule;
    } schedule_data;
    char __pad [SMP_CACHE_BYTES];
} aligned_data [NR_CPUS] __cacheline_aligned = { { {&init_task,0} } };
```

코드 1111. aligned_data[] 배열의 정의

aligned_data[] 배열은 각각의 CPU에 대한 스케줄링 데이터를 가지는 자료 구조이다. 내부의 자료구조로는 schedule_data 구조체를 가지며, 현재의 task와 마지막 스케줄링이 일어났던 시점을 기록하는 cycles_t(=unsigned long) 타입을 가지는 변수로 정의된다. 즉, 각각의 프로세서마다 어떤 프로세스가 가장 최근에 실행되었는지를 알저주는 역할을 한다. 초기에는 init_task가 모든 CPU의 current task로 들어가게 되며, 스케줄링이 일어난 시점을 0으로 표시하고 있다.

17.7.4. init() 함수의 분석

init() 함수는 실제적으로 시스템의 root 프로세스를 생성하는 일을 한다. 즉, 실제적인 프로세스 PID 1에 해당하는 프로세스를 생성한다.⁴⁴⁶ 정의는 ~/init/main.c에 아래와 같이 되어 있다.

```
static int init(void * unused)
{
    lock_kernel();
    do_basic_setup();
    /*
     * Ok, we have completed the initial bootup, and
     * we're essentially up and running. Get rid of the
     * initmem segments and start the user-mode stuff..
     */
    free_initmem();
    unlock_kernel();
    if (open("/dev/console", O_RDWR, 0) < 0)
        printk("Warning: unable to open an initial console.\n");
    (void) dup(0);
    (void) dup(0);

    /*
     * We try each of these until one succeeds.
     *
     * The Bourne shell can be used instead of init if we are
     * trying to recover a really broken machine.
     */
    if (execute_command)
        execve(execute_command,argv_init,envp_init);
    execve("/sbin/init",argv_init,envp_init);
    execve("/etc/init",argv_init,envp_init);
    execve("/bin/init",argv_init,envp_init);
    execve("/bin/sh",argv_init,envp_init);
```

⁴⁴⁶ 프로세스 PID 0은 커널의 idle process가 된다.

```

    panic("No init found. Try passing init= option to kernel.");
}

```

코드 1112. init() 함수의 분석

init() 함수는 이전에 start_kernel() 함수에서 kernel thread로 생성한 함수이다. 이 함수가 하는 일은 크게 나누어, start_kernel() 함수에서 다 처리하지 않은 시스템의 초기화와 시스템의 초기 프로세스인 PID 0을 생성하는 것이다. 먼저 lock_kernel()을 호출해서 커널에 전역 lock을 설정한다. 그리고나서, do_basic_setup() 함수를 호출해서 시스템의 초기화를 수행하고, free_initmem() 함수를 호출해서 initial 메모리에 해당하는 부분을 커널 메모리에서 해제한다. 이것을 마치고나면, 다시 커널에 설정했던 lock을 해제하고(unlock_kernel()), /dev/console를 Read/Write permission을 설정해서 열고, 다시 표준 입력을 복사한다. 이것으로 표준 입력(stdin)과 표준 출력(stdout), 표준 에러(stderr)가 각각 설정된다.

두번째 부분에서는 이전 시스템의 초기 프로세스를 수행하는 일이 남았다. 먼저 execute_command로 넘어온 부분이 있는지 확인하고, 이것이 있을 경우에는 환경 변수를 넘겨주어서 수행하도록 한다. 이제는 init program을 수행하기 위해서 있을 만한 디렉토리를 찾아서, 해당 프로그램을 수행하도록 한다. 디렉토리들은 /sbin, /etc, /bin이며, init 프로그램이 있을 만한 곳들이다. 없다면, 기본 shell인 /bin/sh를 수행하려고 할 것이다. 이것마저도 없다면, panic()을 호출해서 init 프로세스를 생성할 수 없다는 것을 알려준다.

17.7.4.1. do_basic_setup() 함수의 분석

do_basic_setup() 함수는 앞에서 start_kernel() 함수에서 다 하지 않은 시스템의 나머지 설정을 하는 역할을 한다. 이 함수는 많은 커널 설정에 따라서 수행되는 함수들이 달라지게 되므로, 여기서 보고자 하는 내용은 커널의 설정 사항과 관련이 없이 항상 수행되는 함수들만은 다루도록 하겠다.

```

static void __init do_basic_setup(void)
{
    /* 각종 initialization 함수들이 이곳에서 수행된다.*/
    /* mtrr_init(), sysctl_init(), pci_init(), sbus_init(), ppc_init(), mca_init(), ecard_init(),
       zorro_init(), dio_init(), nubus_init(), isapnp_init(), tc_init() */
    ...
    /* Networking initialization needs a process context */
    sock_init();
#endif CONFIG_BLK_DEV_INITRD
    real_root_dev = ROOT_DEV;
    real_root_mountflags = root_mountflags;
    if (initrd_start && mount_initrd) root_mountflags &= ~MS_RDONLY;
    else mount_initrd = 0;
#endif
    start_context_thread();
    do_initcalls();
    /* .. filesystems .. */
    filesystem_setup();
    /* IRDA : irda_device_init(), PCMCIA : init_pcmcia_ds() 가 수행된다. */
    /* Mount the root filesystem.. */
    mount_root();
    mount_devfs_fs ();
}

```

코드 1113. do_basic_setup() 함수의 정의

이곳까지 진행했다면, 이미 machine에 대한 초기화는 끝났다고 본다. 하지만, 아직도 어떤 device에 대한 것도 초기화 되지 못한 상태이지만, CPU와 메모리 관리 시스템 및 프로세스 관리 시스템은 이미 초기화된 상태이다. 따라서, 각종 디바이스를 초기화 하기 위한 기본 자료구조를 생성하는 역할을 수행하게 된다. 관련된 것으로는 코드에서 보듯이 각종 함수들이 있으며(mtrr_init(), sysctl_init(), pci_init(), sbus_init(), ppc_init(), mca_init(), ecard_init(), zorro_init(), dio_init(), nubus_init(), isapnp_init(), tc_init()), 이것에 IRDA와 PCMCIA가 추가될 것이다. 이것을 제외한 나머지를 보면 다음과 같다.

가장 먼저 나오는 것이 network을 초기화하기 위한 `sock_init()` 함수⁴⁴⁷이다. 이 함수의 수행을 마치고나면, initial RAM disk에 대한 설정을 해준다. 실제 root device(`ROOT_DEV`) 및 mount시에 설정할 모드(mode)를 설정한다. 물론 이것도 `CONFIG_BLK_DEV_INITRD`가 커널 커널 파일에서 설정된 상황이어야 할 것이다. `start_context_thread()` 함수는 keventd 데몬 프로세스를 커널 thread로 생성하는 일을 한다. keventd 쓰레드는 단순히 스케줄러의 queue를 실행시켜주는 데몬 프로세스로 예측 가능한 프로세스의 context를 수행하고자 하는 task에 대해서 제공할 목적으로 고안된 것이다. 2.4.X 버전의 커널에서 존재한다. `do_initcall()` 함수는 나중에 다시 살펴보겠지만, 단지 `_initcall_start`와 `_initcall_end` 사이에 정의된 함수들을 차례로 수행하는 일을 한다. 기본적으로 초기화에만 실행될 필요가 있는 함수들이 이곳에 위치하며, 수행을 마치고나면 메모리가 해제된다. `filesystem_setup()` 함수는 파일 시스템을 초기화 하는 역할을 하는 함수이며, `~/fs/filesystems.c`에 아래와 같이 정의된다⁴⁴⁸.

```
void __init filesystem_setup(void)
{
    init_devfs_fs(); /* Header file may make this empty */

#ifndef CONFIG_NFS_FS
    init_nfs_fs();
#endif
#ifndef CONFIG_DEVPTS_FS
    init_devpts_fs();
#endif
}
```

코드 1114. `filesystem_setup()` 함수의 정의

`filesystem_setup()` 함수는 NFS(Network File System)과 device pts(pseudo-terminal structure) 파일 시스템에 대한 초기화를 수행한다. 각각은 `CONFIG_NFS_FS`와 `CONFIG_DEVPTS_FS`가 정의된 경우에만 수행될 것이다. 따로이 특별한 설명은 이곳에서 보지 않겠다.

`mount_root()` 함수는 root 파일 시스템으로 설정될 파일 시스템을 mount하는 역할을 한다. 또한 `mount_devfs_fs()` 함수는 device 파일 시스템을 mount하는 일을 한다. `mount_root()` 함수는 `ROOT`가 NFS로 설정되었는지, 아니면 initial RAM disk로 설정되었는지에 따라 수행을 달리하게 될 것이며, 만약 network가 기동중이며 NFS로 root를 mount한다면, 다른 host에 개발하려는 기기의 root 파일 시스템을 들 수 있을 것이다. 이는 개발의 초기에 많은 이점을 줄 수 있다. 또한 mount한다는 말은 해당하는 파일 시스템에 대한 super block 구조체를 읽는 함수를 호출하겠다는 말이되며, 만약 이 method가 구현되어 있다면 호출될 것이다. 주의 할 점은 여기서 RAM disk를 loading한다는 점이다. 즉, 우리가 만약 RAM disk로 root file system을 만들어 주었다면, 이곳에서 그것을 mount해서 사용할 것이다. `mount_devfs_fs()` 함수는 `~/fs/devfs/base.c`에 정의된 함수로 device들에 대한 것을 file system과 같은 구조로 RAM에 만들어서 사용하기 위한 것이다. 자세한 것은 `~/fs/devfs`를 참고하기 바란다⁴⁴⁹.

```
#ifdef CONFIG_BLK_DEV_INITRD
    root_mountflags = real_root_mountflags;
    if (mount_initrd && ROOT_DEV != real_root_dev
        && MAJOR(ROOT_DEV) == RAMDISK_MAJOR && MINOR(ROOT_DEV) == 0) {
        int error;
        int i, pid;
        pid = kernel_thread(do_linuxrc, "/linuxrc", SIGCHLD);
        if (pid>0)
            while (pid != wait(&i));
```

⁴⁴⁷ 이 함수는 프로토콜 스택에 대한 초기화 및 Linux의 네트워크에서 가장 중요한 socket buffer 구조체에 대한 초기화를 포함한다.

⁴⁴⁸ 이곳에서 정의한 `filesystem_setup()`은 커널의 버전에 따라서 약간씩 달라질 수 있다.

⁴⁴⁹ `~/Documentation`에 보면, `kernel-docs.txt`에 Device File System(devfs)에 대한 것을 찾기 위해서 <http://www.atnf.csiro.au/~rgooch/linux/docs/devfs.txt>를 참고하라고 나온다.

```

        if (MAJOR(real_root_dev) != RAMDISK_MAJOR
            || MINOR(real_root_dev) != 0) {
            error = change_root(real_root_dev, "/initrd");
            if (error)
                printk(KERN_ERR "Change root to /initrd: "
                    "error %d\n", error);
        }
    }
#endif
}

```

코드 1115. do_basic_setup() 함수의 정의(계속)

이 부분은 CONFIG_BLK_DEV_INITRD가 설정된 상황에서만 수행되는 부분이다. 즉, initial RAM disk를 처리하는 부분이다. mount_initrd가 0이 아닌 값을 가지고, ROOT_DEV가 real_root_dev와 같지 않으며, ROOT_DEV의 major번호가 RAM disk의 major번호와 같고, ROOT_DEV의 minor 번호가 0인 경우에 이하를 수행한다⁴⁵⁰.

즉, do_linuxrc() 함수를 커널 thread로 생성해서, 이 함수에 /linuxrc와 SIGCHLD를 넘겨준다. 만약 pid가 0보다 크다면, 이는 부모 process가 되기에 do_linuxrc의 호출 결과를 기다리도록 한다. do_linuxrc() 쓰레드가 수행이 되고나면, 다음 줄(line)이 수행될 것이다. 여기서는 real_root_dev의 major 번호 값이 RAM disk의 major번호와 다르거나, 혹은 real_root_dev의 minor번호가 0이 아닌 경우에 change_root() 함수를 수행해서, real_root_dev가 /initrd를 가지도록 만든다. 어려가 발생한다면, /initrd로 root를 전환하는데 실패한 것이다. 즉, 이전에 initial RAM disk를 root로 사용했다면, 이곳에서 real_root_dev로 root device를 바꾸는 역할을 하는 함수이다. 정의는 ~/fs/super.c에 있다.

17.7.4.2.sock_init() 함수의 분석

여기서 sock_init() 함수를 좀더 자세히 보려는 이유는 이곳에서 network 하위 시스템에 대한 초기화가 이루어 지기 때문이다. 디바이스 드라이버에 대한 초기화보다는 프로토콜 stack의 초기화가 수행된다고 보면 될 것이다. 또한 네트워크 하위 시스템의 가장 중요한 요소인 socket buffer 구조체의 초기화 및 할당이 이곳에서 수행된다. 정의는 ~/net/socket.c에 아래와 같다.

```

void __init sock_init(void)
{
    int i;

    printk(KERN_INFO "Linux NET4.0 for Linux 2.4\n");
    printk(KERN_INFO "Based upon Swansea University Computer Society NET3.039\n");
    /* Initialize all address (protocol) families. */
    for (i = 0; i < NPROTO; i++)
        net_families[i] = NULL;
    /* Initialize sock SLAB cache. */
    sk_init();
#ifndef SLAB_SKB
    /* Initialize skbuff SLAB cache */
    skb_init();
#endif
    /* Wan router layer. */
#ifndef CONFIG_WAN_ROUTER
    wanrouter_init();
#endif
    /* Initialize the protocols module. */

```

⁴⁵⁰ 즉, RAM disk의 major번호를 ROOT_DEV가 가지는 경우로 우리가 만들어줄 RAM disk를 root로 사용하겠다는 말이된다. 앞에서 이미, mount_root()를 수행함으로써, 어느정도 이 정보를 알게 되었을 것이다.

```

register_filesystem(&sock_fs_type);
sock_mnt = kern_mount(&sock_fs_type);
/* The real protocol initialization is performed when
 * do_initcalls is run.
 */
/* The netlink device handler may be needed early. */
#endif CONFIG_RTNETLINK
    rtinetlink_init();
#endif
#ifndef CONFIG_NETLINK_DEV
    init_netlink();
#endif
#ifndef CONFIG_NETFILTER
    netfilter_init();
#endif
}

```

코드 1116. `sock_init()` 함수의 정의

`net_families[]`는 `protocol families`를 가지는 array이다. 현재 아직 아무런 프로토콜도 초기화 되지 않았으므로, `NULL`을 넣어주도록 한다. `NPROTO`는 32로 충분한 `protocol`들의 `entry`를 가질 정도의 크기를 가르키도록 한다. `sock` 구조체는 `inet layer`와 `socket layer`를 interface하며, `file system`과 같이 `socket`을 사용할 수 있도록 하는 자료구조체이다. 이를 위한 `cache`를 `sk_init()` 함수에서 할당한다. `slab allocator`의 `kmem_create_cache()`를 사용해서 `cache`를 할당받는다. 또한 `skb_init()` 함수는 `inet layer`이하에서 사용하게되는 자료구조체이며, 이것의 연결리스트에 대한 초기화를 수행하는 일을 한다. `Slab allocator`의 `kmem_create_cache()`의 호출을 역시 사용하고 있다. 만약 `WAN router`로 설정되었다면, `wanrouter_init()`가 호출될 것이다. 이곳에서는 하나의 `internet protocol layer`로서 `WAN routeer layer`가 초기화 된다는 사실만 알면 될 것이다.

이젠 `protocol` 모듈들에 대한 설정을 할 차례이다. 이를 위해서 `sock_fs_type`을 파일 시스템을 등록한다(`register_filesystem`). 이렇게 등록된 `file system`은 바로 아래에서 `kern_mount()` 함수를 호출하면서, 같이 등록한 `super block` `read`와 관련된 함수를 수행할 것이다. `super block` `read`를 하는 함수는 `sockfs_read_super()`로서 정의는 `~/net/socket.c`에 아래와 같이 나와 있다.

```

static struct super_block * sockfs_read_super(struct super_block *sb, void *data, int silent)
{
    struct inode *root = new_inode(sb);
    if (!root)
        return NULL;
    root->i_mode = S_IFDIR | S_IRUSR | S_IWUSR;
    root->i_uid = root->i_gid = 0;
    root->i_atime = root->i_mtime = root->i_ctime = CURRENT_TIME;
    sb->s_blocksize = 1024;
    sb->s_blocksize_bits = 10;
    sb->s_magic = SOCKFS_MAGIC;
    sb->s_op = &sockfs_ops;
    sb->s_root = d_alloc(NULL, &(const struct qstr) { "socket:", 7, 0 });
    if (!sb->s_root) {
        iput(root);
        return NULL;
    }
    sb->s_root->d_sb = sb;
    sb->s_root->d_parent = sb->s_root;
    d_instantiate(sb->s_root, root);
    return sb;
}

```

코드 1117. `sockfs_read_super()` 함수의 정의

이 함수는 단순히 하나의 inode entry를 만들고, socket을 위한 super_block구조체를 초기화 하는 역할을 수행한다. 먼저 새로 생성된 inode구조체는 S_IFDIR | S_IRUSR | S_IWUSR이 모드 bit으로 설정되며, UID와 GID는 각각 0을 가진다. 나머지 time에 대한 설정은 CURRENT_TIME⁴⁵¹을 가지도록 만든다. 나머지는 socket file system을 위한 super block 구조체를 초기화 하는 일을 한다. block의 크기는 1024 bytes이며, 10bit을 block size의 크기로 사용한다(즉, 2의 10승이 1024가 된다.). 또한 magic number로 socket임을 나타내는 SOCKFS_MAGIC(=0x534F434B)를 사용하고, 관련된 operation으로는 sockfs_ops구조체를 정의하고 있다. 현재 관련된 super block에 대한 operation으로는 단지 statistics를 제공하는 sockfs_stats()만이 정의되어 있다. d_alloc() 함수는 directory entry를 할당하는 함수로서 “socket:”이라는 이름과 이름의 길이로 7을 hash값으로는 0을 주었다. 할당 받을 수 없다면, inode를 버리고 NULL을 return한다. 이젠 할당 받은 directory entry가 socket의 super block을 가르키도록 만들고, d_instantiate() 함수를 호출해서 directory entry의 inode에 관련된 정보를 채워 넣는다. 복귀 값은 초기화를 마친 socket을 위한 super block 구조체가 될 것이며, sock_init() 함수에서 sock_mnt가 저장한다.

나머지 함수들은 netlink를 위한 초기화 목적을 가진 rtnetlink_init(), init_netlink()가 있으며, NET filter로 커널이 설정된 경우에 이를 처리하기 위한 netfilter_init()함수가 있다. Network filter는 네트워크의 보안과 패킷의 forwarding을 설정하기 위해서 리눅스에서 사용되고 있다. 이의 설정은 2.2버전의 커널과 2.4버전의 커널에서 많은 차이가 있으므로 관련된 것을 보고자 하는 사람은 주의하기 바란다.

17.7.4.3. do_linuxrc() 함수의 분석

do_linuxrc() 함수는 커널 컴파일에서 initial RAM disk가 정의된 경우에 수행되는 함수이다. 즉, RAM disk를 만들어서 사용할 수 있도록 만들어주는 역할을 한다. 이것을 사용하기 위해서는 /linuxrc 프로그램이 필요하다.

```
#ifdef CONFIG_BLK_DEV_INITRD
static int do_linuxrc(void * shell)
{
    static char *argv[] = { "linuxrc", NULL, };

    close(0);close(1);close(2);
    setsid();
    (void) open("/dev/console",O_RDWR,0);
    (void) dup(0);
    (void) dup(0);
    return execve(shell, argv, envp_init);
}
#endif
```

코드 1118. do_linuxrc() 함수의 정의

실행할 프로그램을 위한 argument(argv[])로 linuxrc를 설정하고, 있을지도 모를 stdin, stdout, stderr를 전부 close()를 이용해서 닫는다. 그리고, 프로세스의 session ID를 설정하기 위해서 setsid()함수를 호출한 후, /dev/console을 열어서, 이것을 표준 입력(stdin)과 표준 출력(stdout) 및 표준 에러(stderr)로 설정한다. 나머지는 /linuxrc를 앞에서 만든 argv[]와 envp_init[]를 넘겨주어서 수행하는 일이다(execve()).

여기서 잠시 initial RAM disk가 있을 경우의 시스템의 booting을 추적해 보기로 하자. 간단히 정리하면 아래와 같다.

- 커널의 boot loader는 커널과 initial RAM disk를 load한다.
- 커널은 initrd(initial RAM disk)를 일반적인 RAM disk로 만들고, initrd에 의해서 사용된 메모리를 해제한다.
- initrd는 read/write가 가능한 root로 mount된다.

⁴⁵¹ CURRENT_TIME은 현재 시간을 가지고 있는 변수인 xtime을 참조하고 있다.

- /linuxrc가 실행되며, /linuxrc는 어떤 유효한 실행 가능한 파일이 될 수 있다. 즉, shell script도 될 수 있으며, uid로 0을 가지며, 기본적으로 init가 할 수 있는 모든 것을 할 수 있다.
- /linuxrc가 실제 root file system을 mount한다.
- pivot_root() 시스템 콜을 사용해서 root directory에 root file system을 위치 시킨다.
- 일반적인 boot sequence가 이어진다. 즉, init program을 root file system에서 불러와서 수행하게 된다.
- 마지막으로 initrd file system은 제거된다.

자세한 것을 알기 원한다면, 커널 source의 Documentation 아래에 있는 initrd.txt를 참고하기 바란다.

17.7.4.4. do_initcalls() 함수의 분석

do_initcalls() 함수는 __initcall_start에서 __initcall_end까지 정의된 함수들을 수행하기 위한 함수이다. __initcall_XXX는 ~/arch/arm/vmlinux-armv.lds.in에 정의된 LD의 script에서 정의된 변수들이며, ~/include/linux/init.h에서 정의된 매크로를 이용해서 정의된 함수들이 위치해 있는 곳이다.

```
static void __init do_initcalls(void)
{
    initcall_t *call;

    call = &__initcall_start;
    do {
        (*call)();
        call++;
    } while (call < &__initcall_end);
    /* Make sure there is no pending stuff from the initcall sequence */
    flush_scheduled_tasks();
}
```

코드 1119. do_initcalls() 함수의 정의

do_initcalls() 함수는 단순히 __initcall_start에서 시작해서 정의된 함수들을 각각 호출해주는 역할을 하는 함수이다(*call()). 즉, __initcall_start에서 __initcall_end까지는 전체가 function의 포인터를 가지는 array로 생각하면 될 것이다. 모듈로 정의할 수 있는 함수들이 커널과 static하게 link되어 컴파일 되었다면, 이곳에서 초기화하는 함수가 수행될 것이다. 마지막의 flush_scheduled_tasks() 함수는 스케줄링된 task가 있다면, 이를 수행해주도록 하는 함수이다. 즉, 앞에서 __initcall_start와 __initcall_end 사이에 있는 어떤 함수를 수행해 줄 때, 생성될 수 있는 어떤 task를 수행을 뒤로 미루지 말고, 이곳에서 완전히 정리해 주기 위함이다.

17.7.4.5. free_initmem() 함수의 분석

```
void free_initmem(void)
{
    if (!machine_is_integrator()) {
        free_area((unsigned long)(&__init_begin),
                  (unsigned long)(&__init_end),
                  "init");
    }
}
```

코드 1120. free_initmem() 함수의 분석

machine_is_integrator()는 단순히 machine_arch_type이 MACHINE_TYPE_INTEGRATOR(=21)로 정의된 경우에 해당하는 것으로, 커널의 설정에서 CONFIG_ARCH_INTEGRATOR가 설정된 경우에만 machine_arch_type 값과 같은지를 비교하는 매크로이다. machine_arch_type은 앞에서 이미 보았듯이(setup_arch() 함수에서), __machine_arch_type의 값으로 결정되는 값이다. 현재 이 값은 SA1100에

기반한 WVP(Web Video Phone)으로 설정해 두었기에, 63이 될 것이다. 따라서, machine_is_integrator() 매크로는 0을 return하게 되며, if절은 수행될 것이다. 따라서, free_area() 함수가 수행되며, 정의는 아래와 같다.

```
static inline void free_area(unsigned long addr, unsigned long end, char *s)
{
    unsigned int size = (end - addr) >> 10;

    for (; addr < end; addr += PAGE_SIZE) {
        struct page *page = virt_to_page(addr);
        ClearPageReserved(page);
        set_page_count(page, 1);
        free_page(addr);
        totalram_pages++;
    }
    if (size && s)
        printk("Freeing %s memory: %dK\n", s, size);
}
```

코드 1121. free_area() 함수의 정의

이 함수는 간단히 넘겨받은 시작주소와 끝주소를 가지고, 해당하는 페이지를 찾아서, free_page() 함수를 호출해 해제하는 역할을 수행하는 함수이다. ClearPageReserved()는 페이지 구조체의 flag 필드의 reserve 속성을 clear시켜주며, set_page_count()는 페이지의 reference count를 설정한다. totalram_pages는 사용 가능한 메모리 상의 페이지를 나타내는 전역 변수이며, 해제가 일어나면 하나 증가가 될 것이다.

free_initmem() 함수가 해제하고자 하는 영역은 __init_begin과 __init_end 영역에 들어있는 페이지 들이다. 이 두 변수는 ~/arch/arm/vmlinux-armv.lds.in에 정의된 것으로, initial code와 data를 담고 있는 부분이다. 즉, 시스템의 초기화 시에만 사용하고, 이후에는 사용되지 않는 메모리 영역을 이곳에 둠으로써, 시스템이 완전 가동이 되기 전에 이 영역에 있는 메모리를 해제해서 사용할 수 있는 메모리를 증가시켜주는 역할을 한다. 아래와 같은 것들이 이 영역에 존재한다.

```
.init : {                                     /* Init code and data */
    _stext = .;
    __init_begin = .;
    *(.text.init)
    __proc_info_begin = .;
    *(.proc.info)
    __proc_info_end = .;
    __arch_info_begin = .;
    *(.arch.info)
    __arch_info_end = .;
    *(.data.init)
    .= ALIGN(16);
    __setup_start = .;
    *(.setup.init)
    __setup_end = .;
    __initcall_start = .;
    *(.initcall.init)
    __initcall_end = .;
    .= ALIGN(4096);
    __init_end = .;
}
```

코드 1122. vmlinux-armv.lds.in의 init section에 대한 정의

processor information을 나타내는 구조체와 architecture information 및 setup에 관련된 것들과 initcall등이 이곳에 있음을 확인할 수 있을 것이다. 예를 들어서, 이중에 __initcall_start와 __initcall_end 사이에 있는 영역은 do_basic_setup() 함수에서 do_initcalls() 함수를 호출하면서, 이곳에 있는 함수들을 다 수행했기에 더 이상 존재할 필요가 없는 부분이다.

이제 booting에 대한 하위 레벨에서의 초기화는 다 끝났다. 여기서 부터의 수행은 init program의 수행을 진행해 나가는 것이다. 따라서, 커널의 초기화를 다루는 것은 여기서 마치도록 하겠다. 더 보기를 원하는 사람은 init program이 어떤 수행 경로를 가지는지를 참고해 보아야 할 것이다. 즉, /etc/inittab와 같은 파일에 정의된 RUN level에 따라서, 어떻게 시스템이 설정되는지를 보면 될 것이다. 이것은 우리가 다루고 있는 커널에서 조금 빗나간 부분이 되므로 여기서는 논하지 않기로 하겠다.

17.8. Network Device Driver

네트워크 디바이스 드라이버는 CS8900을 보도록 하자. 이것은 Cyrus Logic에서 개발된 10Mbps를 지원하는 Ethernet Controller이다. CS8900을 보면, 이 칩은 IEEE 802.3 기준을 따르는 ethernet controller로서 Direct ISA-Bus 연결을 사용한다. 또한 PacketPage라는 것을 I/O 및 메모리 공간과 DMA slave로서 동작할 때 사용한다. Full-duplex⁴⁵²를 지원하며, 칩 자체에 Tx와 Rx를 위한 buffer를 가지고 있다. 네트워크 인터페이스로는 10BASE-T와 10BASE2, 10BASE5, 10BASE-F를 지원하기 위한 AUI를 가지고 있으며, Tx시에는 collision⁴⁵³이 있을 때, 자동적인 재전송 및 Padding과 CRC 생성을 해준다. Rx시에는 CPU의 overhead를 줄이기 위해서 StreamTransfer를 제공하며, DMA와 칩에 있는 메모리간의 자동 switch도 또한 제공한다. 덧붙여서 Frame의 pre-processing을 위한 Early Interrupt와 잘못된 packet에 대한 자동적인 rejection 기능도 있다. Jumper가 없는 상황에서의 설정을 위해 EEPROM도 가지고 있으며, disk가 없는 시스템의 booting을 위한 Boot PROM도 가지고 있다. Boundary scan 및 Loopback test, Link 상태와 LAN의 활동상황을 알아보기 위한 LED 드라이버, Standby 및 Suspend Sleep Mode 기능도 가지고 있다. 이와 같은 기능은 나중에 차근차근 이야기를 해 나갈 것이므로 지금 이것을 다 이해할 필요는 없다.

한 가지 눈 여겨서 볼 부분은 ISA interface를 가지고 있다는 점과 EEPROM도 가지고 있다는 점이다. Embedded System을 구현하게 될 때는 System의 Bus는 Address 버스와 Data 버스만을 가지고 있는 경우가 많으므로 ISA bus에 국한되지 않아야 할 것이다. 또한 EEPROM에는 주로 ethernet controller의 하드웨어 주소를 담아두는 경우가 많은데, EEPROM이 없는 embedded system을 구현할 때는 간단히 하드웨어 주소를 인위적으로 할당할 필요도 있을 것이다. 실제로 삼성전자의 Web Video Phone에서 간단히 테스트 할 경우에 EEPROM이 없었다.

17.8.1. Register

더 이상 이야기를 진행하기 전에 여기서 부터는 Hardware에 의존적인 이야기가 많이 나오므로 간단히 CS8900 network card의 register들에 대한 이야기를 간단히 하도록 하겠다⁴⁵⁴.

먼저 PacketPage에 대한 것을 보도록 하겠다. Prefix로 PP_가 붙는 것들이 PacketPage에 대한 레지스터들의 address를 나타낸다. I/O 공간에서의 연산(operation)이 기본적인 CS8900의 reset 후 설정이고, PacketPage 레지스터의 offset은 나중에 보게될 PacketPage Pointer Port에 들어갈 내용이 되며, PacketPage Pointer Port를 통해서 접근된다. PacketPage Pointer Port는 기본주소(base address)에 0x000A(ADD_PORT)를 더해서 결정된다. 레지스터에 있는 데이터를 읽기 위해서는 이 PacketPage Pointer Port에 먼저 레지스터의 offset을 쓰고나서, data port를 통해서 읽어온다. 쓰는 것도 역시 PacketPage Pointer Port에 먼저 레지스터의 offset을 쓰고나서, data port에 쓸 내용을 적어서 처리한다.

ID	Offset	Description
PP_ChipID	0x0000	0h는 vendor의 ID, 2h는 Model/Product Number, 3h는 Chip rev. 번호

⁴⁵² 보내기와 받기를 동시에 할 수 있다는 말이다. 즉, Half-Duplex인 경우에는 보내기와 받기는 가능하지만, 한번에 동시에 할 수 없다는 것을 나타낸다.

⁴⁵³ 이것은 ethernet의 특성으로서 충돌이 생기는 경우가 있는데, 이것을 collision이라고 한다. 만약 이러한 상황이 생긴다면, packet을 보내는 것을 새로 해 주어야 할 것이다.

⁴⁵⁴ 상세한 정보를 원한다면 당연히 CS8900 ethernet controller chip의 spec.을 읽어 보아야 할 것이다. Crystal Semiconductor(Cirrus Logic)에서 직접 찾아보기 바란다.

PP_ISAIOB	0x0020	I/O base address
PP_CS8900_ISAINT	0x0022	ISA interrupt select
PP_CS8920_ISAINT	0x0370	ISA interrupt select
PP_CS8900_ISADMA	0x0024	ISA Receive DMA Channel
PP_CS8920_ISADMA	0x0374	ISA Receive DMA Channel
PP_ISASOF	0x0026	DMA Start Of Frame : ISA DMA offset
PP_DmaFrameCnt	0x0028	DMA Frame Count(12bits) : ISA DMA Frame count
PP_DmaByteCnt	0x002A	RxDMA Byte Count : ISA DMA Byte count
PP_CS8920_ISAMemB	0x002C	Memory Base Address Register(20bits)
PP_CS8920_ISAMemB	0x0348	Memory Base Address Register(20bits)
PP_ISABootBase	0x0030	Boot PROM Base Address
PP_ISABootMask	0x0034	Boot PROM Address Mask

표 122. PacketPage 메모리 맵(1)- 기본 설정부분

두 가지의 동일한 내용이 있는 경우에는 CS8920과 CS8900이 다른 점을 보완하기 위해서이다. 근본적인 차이점은 없다.

ID	Offset	Description
PP_EECMD	0x0040	EEPROM Command Register
PP_EEData	0x0042	EEPROM Data Register
PP_DebugReg	0x0044	Debug Register (CS8900 : Reserved)

표 123. PacketPage 메모리 맵(2)- EEPROM Interface부분

EEPROM에 데이터를 기록하거나 읽기를 위해서는 위에서 정의된 PacketPage 메모리 맵을 본다. Data를 data register에 놓고, 명령을 command register에 놓으면 될 것이다.

ID	Offset	Description
PP_RxCFG	0x0102	Rx Bus configuration
PP_RxCTL	0x0104	Receive Control Register
PP_TxCFG	0x0106	Transmit Config Register
PP_TxCMD	0x0108	Transmit Command Register
PP_BufCFG	0x010A	Bus Configuration Register
PP_LineCTL	0x0112	Line Config Register
PP_SelfCTL	0x0114	Self Command Register
PP_BusCTL	0x0116	ISA Bus Control Register
PP_TestCTL	0x0118	Test Register
PP_AutoNegCTL	0x011C	Auto-negotiation Control Register

표 124. PacketPage 메모리 맵(3)- configuration과 control register부분

Configuration과 control register들의 맵은 Rx와 Tx 및 bus 설정과 line설정, auto-negotiation 관련 설정부분들로 이루어져 있다.

ID	Offset	Description
PP_ISQ	0x0120	Interrupt Status Register
PP_RxEvent	0x0124	Rx Event Register
PP_TxEvent	0x0128	Tx Event Register
PP_BufEvent	0x012C	Bus Event Register
PP_RxMiss	0x0130	Receive Miss Count Register
PP_TxCol	0x0132	Transmit Collision Count Register
PP_LineST	0x0134	Line Status Register
PP_SelfST	0x0136	Self Status Register
PP_BusST	0x0138	Bus Status Register

PP_TDR	0x013C	Time Domain Reflectometry Register
PP_AutoNegST	0x013E	Auto-negotiation Status Register

표 125. PacketPage 메모리 맵(4)- Status와 event register부분

ID	Offset	Description
PP_TxCommand	0x0144	Transmit Command Register
PP_TxLength	0x0146	Transmit Length Register
PP_LAF	0x0150	Address Filter Register(Logical Address Filter: Hash Table)
PP_IA	0x0158	Individual Address(Physical Address Register)
PP_RxStatus	0x0400	Receive Status Register
PP_RxLength	0x0402	Receive Length Register(In Byte)
PP_RxFrame	0x0404	Receive Frame Pointer Register(Receive Frame Location)
PP_TxFrame	0x0A00	Transmit Frame Pointer Register(Transmit Frame Location)

표 126. PacketPage 메모리 맵(5)- Rx와 Tx를 위한 register와 Address관련 register부분

이상에서 우리는 PacketPage에 나타나는 메모리 맵에 대해서 살펴보았다. 각각의 레지스터는 읽기 전용이거나 쓰기 전용, 혹은 읽기와 쓰기가 모두 가능한 경우가 있다. 또한 중간 중간에 예약된 영역을 가질 수 있으니 이 부분들에 대해서는 주의해야 할 것이다. 즉, 예약된 부분에 대한 읽기는 정의되지 않은 데이터를 돌려줄 것이며, 이 부분에 대한 쓰기는 CS8900에 예측할 수 없는 일을 일으킬 가능성이 있다. 또한 모든 레지스터 들에 대한 접근은 word단위로 할 것이다.

이들 공간에 대한 접근을 위해서 위에서 잠시 예를 들었던 Packet Page Pointer Port register를 통해서 접근하는 방법은 아래와 같다.

```
extern int inline readreg(struct net_device *dev, int portno)
{
    outw(portno, dev->base_addr + ADD_PORT);
    return inw(dev->base_addr + DATA_PORT);
}

extern void inline writereg(struct net_device *dev, int portno, int value)
{
    outw(portno, dev->base_addr + ADD_PORT);
    outw(value, dev->base_addr + DATA_PORT);
}
```

코드 1123. PacketPage 레지스터를 읽고 쓰기

readreg()함수는 net_device 구조체와 쓰게될 포트 번호를 파라미터 값으로 받는다. 먼저 기본 주소에 ADD_PORT(=0x000A)를 더한 후(PacketPage Pointer Register를 선택한다.), 이곳에 해당 레지스터의 옵셋을 넣은 후, 기본 주소에 데이터 포트의 옵셋(=DATA_PORT:0x000C)을 더한 주소를 읽어들여서 복귀 값으로 전달한다.

writereg()함수도 마찬가지 일을 하는데, net_device 구조체와 포트 번호 및 쓸 값을 파라미터로 넘겨 받아서, 먼저 기본 주소에 ADD_PORT를 더한 주소에 포트 번호를 쓰고, 다시 기본 주소에 데이터 포트의 주소를 더한 곳에 값을 쓴다. 복귀 값은 없다.

```
extern int inline readword(struct net_device *dev, int portno)
{
    return inw(dev->base_addr + portno);
}

extern void inline writeword(struct net_device *dev, int portno, int value)
{
    outw(value, dev->base_addr + portno);
}
```

코드 1124. Port에 값을 쓰고 읽기

포트에 값을 바로 읽고 쓰기 위해서는 readword() 함수와 writeword() 함수를 사용한다. 각각은 다시 하위의 routine으로 inw()와 outw()를 사용한다. 읽기 위해서는 port번호에 더해서 함께 기본 주소를 넘겨주고, 쓰기 위해서는 쓰기 값과 기본주소 더하기 port번호를 넘겨준다.

PacketPage register들에 대해서 레지스터의 각 bit 설정에 대해서 알아보기로 하자. 관련된 레지스터 별로 각각 나열하도록 하겠다.

ID	Value	Description
SKIP_1	0x0040	마지막에 commit된 received frame을 receive buffer로 부터 지운다.
RX_STREAM_ENBL	0x0080	StreamTransfer mode가 receive frame을 전달할 때 사용된다.
RX_OK_ENBL	0x0100	RxOK interrupt를 예상 없이 frame을 받았을 때 발생시킨다.
RX_DMA_ONLY	0x0200	Receive-DMA mode가 모든 received frame들에 대해서 사용된다.
AUTO_RX_DMA	0x0400	조건들을 만족한다면 자동적으로 Receive-DMA mode로 스위치 한다.
BUFFER_CRC	0x0800	Receive frame buffer에 CRC값이 포함된다. Frame의 길이도 역시 이 값을 계산에 넣은 값이다.
RX_CRC_ERROR_ENBL	0x1000	잘못된 CRC를 가진 frame이 받았을 경우, CRC error interrupt가 발생한다.
RX_RUNT_ENBL	0x2000	64bytes이하의 frame을 받으면 Runt Interrupt를 발생시키도록 한다.
RX_EXTRA_DATA_ENBL	0x4000	1518bytes보다 긴 frame을 받으면 Extradata Interrupt를 발생시키도록 한다.

표 127. PP_RxCfg의 Receive configuration과 interrupt mask bit 정의

PP_RxCfg 레지스터는 받기(receive)를 위한 설정을 담당하는 레지스터이다. 디바이스 드라이버는 나중에 open()함수와 같은 곳에서 이 레지스터를 초기화 시켜주어야 할 것이다.

ID	Value	Description
RX_IA_HASH_ACCEPT	0x0040	Hash filter를 통과한 목적지 주소만을 가진 frame을 받는다.
RX_PROM_ACCEPT	0x0080	주소에 상관없이 모든 frame을 다 받는다(Promiscuous mode).
RX_OK_ACCEPT	0x0100	올바른 CRC와 유효한 길이를 가지는 frame만을 받는다.
RX_MULTICAST_ACCEPT	0x0200	Hast filter를 통과한 multicast address를 가지는 frame을 받는다.
RX_IA_ACCEPT	0x0400	PacketPage base + 0158h에서 015Dh까지의 영역에 있는 Individual Address에 해당하는 패킷을 받는다.
RX_BROADCAST_ACCEPT	0x0800	목적지 주소가 FFFF FFFF FFFFh인 frame을 받겠다(Broadcast).
RX_BAD_CRC_ACCEPT	0x1000	잘못된 CRC를 가지는 목적지 주소 filter를 거친 frame을 받는다.
RX_RUNT_ACCEPT	0x2000	64bytes보다 작은 frame으로 목적지 주소 filter를 통과한 frame을 받는다.
RX_EXTRA_DATA_ACCEPT	0x4000	1518bytes보다 큰 frame으로 목적지 주소 filter를 통과한 frame을 받는다.
RX_ALL_ACCEPT	-	RX_PROM_ACCEPT RX_BAD_CRC_ACCEPT RX_RUNT_ACCEPT RX_EXTRA_DATA_ACCEPT
DEF_RX_ACCEPT	-	RX_IA_ACCEPT RX_BROADCAST_ACCEPT RX_OK_ACCEPT

표 128. PP_RxCTL의 Receive control bit의 정의

PP_RxCTL 레지스터는 Rx를 control하는 레지스터이다. 주로 어떤 패킷들을 받을 것인가를 결정하기 위한 제어를 제공한다.

ID	Value	Description
TX_LOST CRS_ENBL	0x0040	만약 CS8900이 AUI상에서 보내기를 시작하며, preamble의 마지막에 있는 Carrier Sense Signal을 보지 않는다면, 이 bit이 설정되면 interrupt가 생성된다.
TX_SQE_ERROR_ENBL	0x0080	SQE에러가 이 bit이 설정되면 interrupt를 통해서 보고된다.
TX_OK_ENBL	0x0100	전체 패킷이 완전히 전송되면 interrupt를 발생시킨다.
TX_LATE_COL_ENBL	0x0200	Late collision이 발생하면 interrupt를 발생시킨다. Late collision이란 512bit time 이후에 발생하는 collision이다.
TX_JBR_ENBL	0x0400	대략 26ms보다 긴 transmission이 생기면 interrupt를 발생시킨다.
TX_ANY_COL_ENBL	0x0800	패킷의 전송중 하나 이상의 collision이 발생하면, 전송의 마지막에서 interrupt를 발생시킨다.
TX_16_COL_ENBL	0x8000	CS8900이 특정 패킷을 전송하는 동안, 16개의 일반적인 collision이 생긴다면, 전송을 멈춘게 된다. 만약 이 bit이 설정되었다면, 16번째 collision이 발생할 때 interrupt를 발생시킨다.

표 129. PP_TxCFG Tranmit configuration Interrupt Mask bit definition

PP_TxCFG 레지스터는 보내기(transmission)과 관련된 설정을 하는 레지스터이다. 보내기에 대한 확인은 interrupt로 보고 되기 때문에, 어떤 event가 일어났을 때, 인터럽트를 발생시킬 것인가를 결정한다.

ID	Value	Description
TX_START_4_BYTES	0x0000	5 bytes가 CS8900에 있을 때 transmission을 시작하라.
TX_START_64_BYTES	0x0040	64 bytes가 CS8900에 있을 때 transmission을 시작하라.
TX_START_128_BYTES	0x0080	128 bytes가 CS8900에 있을 때 transmission을 시작하라.
TX_START_ALL_BYTES	0x00C0	모든 frame이 CS8900에 있을 때 transmission을 시작하라.
TX_FORCE	0x0100	새로운 Tx command와 함께 설정된 경우, 모든 Tx buffer에서 전송을 대기하고 있는 frame은 지워진다. 만약 이미 전송이 시작된 시점이라면, 64 bit times이내에 bad CRC를 가진 frame으로 끝나게 될 것이다.
TX_ONE_COL	0x0200	이 bit이 설정되며, 단지 하나의 collision이후에 모든 Tx는 끝날 것이다. 설정되지 않으면, 일반적으로 16 번의 collision이 발생한 후에 Tx가 끝날 것이다.
TX_TWO_PART_DEFF_DISABLE	0x0400	NOT defined for CS8900.
TX_NO_CRC	0x1000	CRC가 Tx에 더해지지 않는다.
TX_RUNT	0x2000	CS8920은 이 bit의 설정에 따라서, 60 bytes이하의 frame을 전송할지 안할지를 결정한다. 만약 설정되었다면, 60 bytes이하의 frame에 대해서 padding을 할 것이다.

표 130. PP_TxCMD Transmission command status register bit definition

PP_TxCMD 레지스터는 다음 frame을 어떻게 전달할 것인가를 결정하는 레지스터이다.

ID	Value	Description
GENERATE_SW_INTERRUPT	0x0040	이것이 설정되면, host software에 의해서 요청된 interrupt 가 발생한다. 이 bit은 한번에 하나의 인터럽트만 발생하므로, 다른 인터럽트를 발생시키고자 한다면, 다시 1을 이 bit에 설정해야 한다.
RX_DMA_ENBL	0x0080	Frame을 다받았고 DMA가 끝날 때, interrupt를 발생시킨다.

READY_FOR_TX_ENBL	0x0200	CS8900이 host로 부터 frame을 전송을 위해서 받을 준비가 다 되었을 때, interrupt를 발생시킨다.
RX_MISS_ENBL	0x0400	Receive buffer로 부터 데이터를 뽑아오는데 걸리는 시간으로 인해서 생기는 하나이상의 frame을 놓치게(miss) 될 때 interrupt를 발생시킨다.
RX_128_BYTE_ENBL	0x0800	Frame의 첫 128 bytes를 받은 이후에 interrupt를 발생시킨다. 이것으로 host processor는 목적지의 주소, 소스의 주소, 길이, 연결번호(sequence number) 및 다른 정보들을 전체 frame을 받기전에 검사해 볼 수 있다.
TX_COL_COUNT_OVRFLOW_ENBL	0x1000	TxCOL(Tx Collision)이 발생한 count가 1FFh에서 200h로 증가할 경우에 interrupt를 발생하도록 만든다.
RX_MISS_COUNT_OVRFLOW_ENBL	0x2000	RxMISS(Rx Miss)이 발생한 count가 1FFh에서 200h로 증가할 경우에 interrupt를 발생하도록 만든다.
RX_DEST_MATCH_ENBL	0x8000	RxCTL 레지스터에 정의된 목적지 주소 filter의 criteria를 받은 frame이 통과한다면, interrupt를 발생시킨다.

표 131. PP_BufCFG Buffer configuration Interrupt Mask bit definition

PP_BufCFG 레지스터는 Rx 및 Tx시에 발생하는 buffer의 관리에 대한 설정을 담당한다. 즉, buffer와 관련된 interrupt의 발생을 유발시킬 수 있도록 설정하는 레지스터이다.

ID	Value	Description
SERIAL_RX_ON	0x0040	Receiver를 enable시킨다.
SERIAL_TX_ON	0x0080	Transmitter를 enable시킨다.
AUI_ONLY	0x0100	AUI나 10BASE-T를 고르는데 사용하는 것으로 AUI만 쓴다고 설정.
AUTO_AUI_10Baset	0x0200	마찬가지로 AUI나 10BASE-T를 자동적으로 설정해서 사용한다.
MODIFIED_BACKOFF	0x0800	설정되지 않으면, ISO/IEC의 표준 backoff 알고리즘 ⁴⁵⁵ 을 사용한다. 설정된다면, Modified Backoff 알고리즘을 사용한다.
NO_AUTO_POLARITY	0x1000	10BASE-T receiver의 경우 자동적으로 received signal의 극성(polarity)을 결정한다. 이 bit이 설정되지 않으면, polarity는 정정되지 않는다. 만약 설정되어 있다면, polarity를 정정(correct)을 하려는 노력을 하지 않는다.
TWO_PART_DEFDIS	0x2000	전송을 시작하기전에, CS8900은 지역 procedure를 따른다. 이 bit이 설정된 경우에는 CS8900은 ISO/IEC 8802-3의 4.2.3.2.1에 정의된 two-part deferral을 따르지 않는다.
LOW_RX_SQUELCH	0x4000	이 bit이 설정되지 않으면, 10BASE-T receiver는 squelch thresholds ⁴⁵⁶ 가 ISO/IEC 8802-3 스펙에 정의된 레벨로 설정된다.

표 132. PP_LineCTL Line control bit definition

PP_LineCTL 레지스터는 전송 line의 상태를 제어하기 위해서 사용하는 레지스터이다.

ID	Value	Description
POWER_ON_RESET	0x0040	이 bit이 설정되면, chip의 전 영역에 대한 reset이 이루어진다.
SW_STOP	0x0100	이 bit이 설정되면, CS8900은 software가 초기화 시킨 suspend mode로 들어간다.
SLEEP_ON	0x0200	이 bit이 설정되면, SLEEP pin에 대한 input이 enable된다.

⁴⁵⁵ Tx Collision이 발생한 경우에 delay를 주어서 다시 전송하게 되는데, 이때 사용하는 알고리즘이 backoff 알고리즘이다.

⁴⁵⁶ Squelch thresholds는 어떤 특정 dB(deci-bell)이하의 signal을 무시하는 한계값을 말한다. 즉, 더 긴 cable을 사용한다면, 약한 signal을 감지하기 위해서 이 값은 작아져야 할 것이다.

AUTO_WAKEUP	0x0400	SLEEP_ON bit이 설정되어 있으며, SLEEP pin이 low인 상태라면, 이 bit의 설정에 따라서, Hardware standby mode로 가거나, Hardware suspend mode로 CS8900은 진입하게 된다.
HCB0_ENBL	0x1000	LINELED나 HC0 output pin이 이 control bit으로 선택된다. HC0E가 설정되면, output은 HC0 pin이 되며, HCB0 bit이 이 pin을 제어한다.
HCB1_ENBL	0x2000	BSTATUS나 HC1 output pin이 이 control bit으로 선택된다. HC1E가 설정되면, output은 HC1 pin이 되며, HCB1 bit이 이 pin을 제어한다. 만약 clear된다면, output pin은 BSTATUS가 되며, receiver의 ISA bus activity를 나타낸다.
HCB0	0x4000	HC0 pin을 control하는 bit으로 HC0는 LED나 logic gate를 구동 시킬 것이다.
HCB1	0x8000	HC1 pin을 control하는 bit으로 HC1은 LED나 logic gate를 구동 시킬 것이다.

표 133. PP_SelfCTL Software self control bit definition

PP_SelfCTL 레지스터는 소프트웨어 적으로 self control을 제어한다. 이 control 레지스터를 접근해서 network adapter를 reset하거나 모드를 변환 시킬 수 있다.

ID	Value	Description
RESET_RX_DMA	0x0040	이 bit이 설정되면, PacketPage base + 0x00260이 0으로 reset된다. 다음과 같은 일을 CS8900이 행한다. 먼저 현재 받고 있는 DMA 활동이 있다면 끝낸다. 모든 내부 Rx buffer를 지운다. RxDMA offset pointer를 0으로 재설정한다.
MEMORY_ON	0x0400	이 bit이 설정되면, CS8900은 메모리 모드로 동작한다. 메모리 모드가 disable되면, I/O mode가 항상 enable된다.
DMA_BURST_MODE	0x0800	이 bit이 clear되면, CS8900은 호스트의 메모리로 받은 frame이 완전히 전송될 때까지 DMA를 계속적으로 수행한다. 만약 이 bit이 설정되어 있다면, DMA접근은 28 microsec. 동안으로 한정된다. 이후에 1.3 microsec. 만큼의 시간을 새로운 DMA를 하기전에 필요로 한다.
IO_CHANNEL_READY_ON	0x1000	이 bit이 설정되면, CS8900은 IOCHRDY output pin을 사용하지 않는다. 만약 이 bit이 clear되면, CS8900은 IOCHRDY를 low로 만들어서, I/O read나 메모리 read cycle하는 동안에 여분의 시간을 요구한다. IOCHRDY는 I/O write나, Memory Write, 혹은 DMA read는 영향을 미치지 않는다.
RX_DMA_SIZE_64K	0x2000	이 bit은 receive DMA buffer의 크기를 결정한다. 설정된다면, DMA buffer의 크기는 64Kbytes이다. Clear되면 16Kbytes가 된다.
ENABLE_IRQ	0x8000	이 bit이 설정되면, CS8900은 interrupt event에 대해서 인터럽트를 생성한다. Clear된다면 물론 아무런 interrupt도 발생하지 않을 것이다.

표 134. PP_BusCTL ISA bus control bit definition

PP_BusCTL 레지스터는 bus의 operation과 관련된 설정을 하는 레지스터이다. Rx DMA 및 메모리 모드, I/O를 위한 channel, DMA size와 Interrupt를 관할 한다.

ID	Value	Description
LINK_OFF	0x0080	이 bit이 설정되면, 10BASE-T 인터페이스는 link의 상태에 관계없이 Tx와 RX를 허락한다. 이 bit은 LINK OK bit(Line Status, bit 7)과 같이 사용한다.
ENDEC_LOOPBACK	0x0200	이 bit이 설정되면, CS8900은 internal loopback mode으로 진입한다.

		Output이 다시 Input으로 연결된다는 말이다. 10BASE-T와 AUI 쪽의 Tx와 Rx를 담당하는 것들은 disable된다. Clear되면 일반적인 operation으로 들어간다.
AUI_LOOPBACK	0x0400	이 bit이 설정되면, CS8900은 보내는 동안에 Rx를 허용한다. 이것은 AUI를 위한 lookback test를 위한 것이며, clear되면 CS8900은 일반적인 AUI operation을 하도록 설정된다.
BACKOFF_OFF	0x0800	이 bit이 설정되면, backoff algorithm은 disable된다. CS8900의 transmitter는 새로운 Tx를 시작하기 전에, inter packet gap이 완료되는 것만을 찾는다. Clear된다면, backoff algorithm이 사용된다.
FDX_8900	0x4000	이 bit이 설정되면, 10BASE-T full duplex mode가 enable되며, CRS(Line status, bit 14)는 무시된다. 10BASE-T port에 대한 loopback test를 위해서는 이 bit은 반드시 설정되어야 한다. Clear된다면 CS8900은 규격화된 half-duplex 10BASE-T operation으로 설정된다.
FAST_TEST	0x8000	이 bit이 설정되면, internal counter들과 timer들은 chip testing을 위해서 scaling된다. Clear되면 일반적인 timing이 사용된다.

표 135. PP_TestCTL Test control bit definition

PP_TestCTL 레지스터는 CS8900의 diagnostic test 모드를 제어한다. 나중에 AUI나 10BASE-T와 같은 interface를 선택하게 된다면, 이 레지스터를 설정해서 테스트 하는 것이 필요할 것이다.

ID	Value	Description
RX_IA_HASHED	0x0040	Hash filter에서 받아 들여진 frame의 목적지 주소라면, 이 bit는 RxOK가 set되고, PP_RxCTL의 bit 6 및 RX_HASHED set된 경우에만 set될 것이다.
RX_DRIBBLE	0x0080	이 bit은 받은 frame이 마지막 byte이후에 하나에서 7개의 bits를 더 가지고 있는 경우에 설정된다. 만약 RX_DRIBBLE과 RX_CRC_ERROR 둘다 설정된 경우에는 alignment error가 발생한다.
RX_OK	0x0100	받은 frame이 CRC와 유효한 length를 가졌을 때 설정된다. 만약 RX_OK가 설정되어 있다면, PacketPage base + 0x0402에는 받은 frame의 길이가 들어갈 것이다. 만약 PP_RxCFG의 bit 80이 설정되었다면, 인터럽트가 발생할 것이다.
RX_HASHED	0x0200	만약 설정된 경우라면, Hash filter에 의해서 받은 frame의 목적지 주소 받아 들여졌다는 말이다. 만약 RX_HASHED와 RX_OK가 설정되었다면, PP_RxEvent의 bit 10에서 bit 15까지는 이 frame의 hash table의 index를 가진다.
RX_IA	0x0400	받은 frame이 PacketPage + 0x0158에 있는 Individual Address와 목적지 주소가 일치한다는 것을 말한다. 이 bit는 RX_OK와 PP_RxCTL의 bit 10번이 설정된 경우에만 set 될 수 있다.
RX_BROADCAST	0x0800	만약 받은 frame이 Broadcast address(FFFF FFFF FFFFh)를 목적지 주소로 가진 경우에 설정된다. RX_OK와 PP_RxCTL의 bit 11이 설정된 경우에만 set될 수 있다.
RX_CRC_ERROR	0x1000	받은 frame이 CRC error를 가지고 있다. 만약 PP_RxCFG의 bit 13이 설정된 경우, interrupt 발생한다.
RX_RUNT	0x2000	받은 frame이 64bytes보다 작다. 만약 PP_RxCFG의 bit 13이 설정된 경우에만 interrupt가 발생한다.
RX_EXTRA_DATA	0x4000	받은 frame이 1518bytes보다 큰 경우에 set된다. 모든 1518bytes 이상이 되는 bytes들은 버려진다. 만약 RxCFG의 bit 14가 설정된 경우에는 interrupt가 발생한다.
HASH_INDEX_MASK	0xFC00	이것은 hash filter의 index를 구하기 위해서 사용하는 mask값이다.

표 136. PP_RxEvent Receive event bit definition

PP_RxEvent레지스터는 Rx Interrupt가 발생한 경우에 어떤 원인이 있는지를 확인하기 위해서 사용한다. 즉, 현재 frame의 상태를 보고한다.

ID	Value	Description
TX_LOST_CRS	0x0040	만약 CS8900이 AUI를 통해서 Tx를 하고 있고, Carrier Sense(CRS)를 preamble의 끝에서 볼 수 없는 경우에 Loss-of-Carrier에러가 생기며, 이 bit이 설정된다. 만약 PP_TxCFG의 bit 6가 설정되었다면, 인터럽트가 발생한다.
TX_SQE_ERROR	0x0080	AUI상에서 Tx의 마지막 부분에서 CS8900은 64 bit time동안 collision이 있는지를 감시하게 되는데, 만약 이것이 없다면, SQE 에러가 발생한 것이며 이 bit이 설정된다. 만약 PP_TxCFG의 bit 7이 설정되었다면, 인터럽트가 발생할 것이다.
TX_OK	0x0100	마지막 packet이 완전히 전송된다면 이 bit이 설정된다. 만약 PP_TxCFG의 bit 8이 설정되었다면, 인터럽트가 발생할 것이다.
TX_LATE_COL	0x0200	Preemle의 첫 bit이후의 512 bit time보다 큰 시간에 collision이 발생한 경우에 이 bit이 설정된다. 이것이 발생하면, CS8900은 bad CRC로 생각하고, Tx를 끝낸다. 만약 PP_TxCFG의 bit 10이 설정된 경우라면 인터럽트가 발생할 것이다.
TX_JBR	0x0400	마지막 Tx가 26msec보다 긴 경우에 packet의 output은 jabber logic에 의해서 끝날 것이며, 이 bit이 설정될 것이다. PP_TxCFG의 bit 9가 설정되었다면, 인터럽트가 발생할 것이다.
TX_16_COL	0x8000	16개의 일반적인 collision을 특정 packet을 전송하는 동안에 만난다면, 이 bit이 설정된다. 이것이 발생하면 CS8900은 더 이상 packet을 전달하는 것을 멈춘다. PP_TxCFG의 bit 15가 설정된 경우에는 인터럽트가 발생할 것이다.
TX_SEND_OK_BITS	-	TX_OK TX_LOST_CRS
TX_COL_COUNT_MASK	0x7800	마지막 packet을 전송하는 동안에 발생한 Tx collision의 수를 구하기 위해서 사용하는 mask값이다. 이 레지스터의 bit 11에서 bit 15까지의 4 bit를 collision의 counter로 사용하기 때문이다.

표 137. PP_TxEvent Transmit event bit definition

PP_TxEvent 레지스터는 마지막으로 전송된 패킷의 상태 정보를 제공한다. 이것을 보고 Tx 인터럽트가 발생했을 때, 보낸 packet이 제대로 갔는지를 확인할 수 있을 것이다.

ID	Value	Description
SW_INTERRUPT	0x0040	만약 설정되었다면, software interrupt가 발생했다는 것을 표시한다. 이 bit은 PP_BufCFG의 bit 6와 연계되어서 사용한다.
RX_DMA	0x0080	만약 설정되었다면, 하나나 그 이상의 받은 frame이 slave DMA에 의해서 전달 되었음을 나타낸다. PP_BufCFG의 bit 7이 설정되었다면, 인터럽트가 발생할 것이다.
READY_FOR_TX	0x0100	만약 설정되었다면, CS8900이 host에서 Tx를 위해서 frame을 받을 준비가 되었다는 것을 나타낸다. PP_BufCFG의 bit 8이 설정되었다면, 인터럽트가 발생할 것이다.
TX_UNDERRUN	0x0200	이 bit은 CS8900이 frame의 마지막을 만나기전에 data를 다 사용했음을(underrun) 나타낸다. 만약 PP_BufCFG의 bit 9가 설정된 경우에는 인터럽트가 발생할 것이다.
RX_MISS	0x0400	만약 설정된다면, 하나나 그 이상의 받은 frame이 receiver buffer로부터 data를 빼오는 것이 느린 이유로 인해서 lost되었음을 나타낸다. 만약 PP_BufCFG의 bit 10이 설정되었다면, 인터럽트가 발생할 것이다.

RX_128_BYT	0x0800	들어오는 frame의 첫 128 bytes를 받은 이후에 설정된다. 이 bit은 host가 전체 frame을 다 받기 전에 frame의 data를 pre-processing하는 것을 가능하게 해준다. 만약 PP_BufCFG의 bit 11이 설정되었다면, 인터럽트가 발생할 것이다.
TX_COL_OVRFLW	0x1000	Tx의 collision counter가 overflow되었음을 나타낸다. CS8900의 spec에는 없다.
RX_MISS_OVRFLW	0x2000	Rx의 mission counter가 overflow되었음을 나타낸다. CS8900의 spec에는 없다.
RX_DEST_MATCH	0x8000	이 bit이 설정되면, PP_RxCTL 레지스터에서 정의된 목적지 주소 filter의 criteria를 받은 frame이 통과했다는 것을 보여준다. 이 bit은 들어오는 frame을 미리 알려주는데 유용하며, RX_128_BYT보다 먼저 될 것이다. 만약 PP_BufCFG bit 15가 설정된 경우에는 인터럽트가 발생할 것이다.

표 138. PP_BufEvent Buffer event bit definition

PP_BufEvent는 Tx나 Rx frame의 상태 정보를 주는 역할을 한다.

ID	Value	Description
LINK_OK	0x0080	이 bit이 설정되면, 10BASE-T link는 OK이다. 만약 clear된다면, link는 fail이며, 이유는 CS8900이 resest되었다거나, receiver가 50ms동안 어떤 activity(link pulse나 received packet)도 감지 못하는 것이다.
AUI_ON	0x0100	이 bit이 설정되면, CS8900은 AUI를 사용하고 있다.
TENBASET_ON	0x0200	이 bit이 설정되면, CS8900은 10BASE-T interface를 사용하고 있다.
POLARITY_OK	0x1000	이 bit이 설정되면, 10BASE-T receive signal의 극성(polarity)가 정상이다. 만약 clear되었다면, 극성은 바뀌어 있다. 만약 PP_TestCTL의 bit 12가 설정되어 있다면, 필요한 경우 자동적으로 정정된다. PP_TestCTL bit 12에 독립으로 들어오는 극성의 TRUE 상태를 보여주는 status bit의 역할을 POLARITY_OK가 나타낸다. 따라서, PP_TestCTL bit 12와 POLARITY_OK bit 둘다가 clear되면, receive 극성은 반대로 되어있고, 정정된다.
CRS_OK	0x4000	이 bit은 host에게 incomming frame의 상태를 알려준다. 만약 설정되었다면, frame을 받고 있는 중이다. EOF(End of frame)까지는 CRS는 그대로 남게 되며, EOF에서 CRS는 복구된 데이터의 마지막 low-to-high 전이 이후에, 대략 1.3에서 2.3 bit times가 흐르면 inactive로 간다.

표 139. PP_LineST Ethernet line status bit definition

PP_LineST 레지스터는 Ethernet의 물리적인 interface 상태를 보고하는 레지스터이다.

ID	Value	Description
ACTIVE_33V	0x0040	3.3Volt에서 CS8900이 동작하고 있다면, 이 bit이 설정된다. 만약 CS8900이 5V에서 동작한다면 clear된다.
INIT_DONE	0x0080	이 bit이 설정되었다면, CS8900은 초기화(initialization)가 끝났다. 초기화에는 EEPROM을 읽어들이는 것도 포함한다.
SI_BUSY	0x0100	이 bit이 설정되었다면, EECS output pin이 high가 되어, EEPROM이 현재 읽혀지고 있거나 programm되고 있음을 나타낸다. Host는 반드시 SIBUSY가 clear될 때까지 PacketPage base + 0x0040에 write하지 말아야 할 것이다.
EEPROM_PRESENT	0x0200	EEPROM이 존재함을 나타낸다.

EEPROM_OK	0x0400	이 bit이 설정되면, EEPROM에 대한 read의 checksum이 OK라는 것을 보여준다.
EL_PRESENT	0x0800	Latchable Address bus decode에 대한 external logic이 존재함을 나타낸다.
EE_SIZE_64	0x1000	붙어있는 EEPROM의 크기를 보여주는 것으로 EEPROM_PRESENT bit과 EEPROM_OK bit이 모두 설정된 경우에만 유효하다. Clear되어 있다면, EEPROM은 128 words나 256 words이며, 그렇지 않다면, 64 words이다.

표 140. PP_SelfST Chip software status bit definition

PP_SelfST 레지스터는 EEPROM interface의 상태 및 초기화 과정의 상태를 보고하는 레지스터이다.

ID	Value	Description
TX_BID_ERROR	0x0080	0 bit이 설정되면, CS8900에게 보내지 못하는 frame을 전송하라고 host가 명령했다는 뜻이다. CS8900이 보내지 못하는 frame은, 1514bytes보다 큰 frame이 TxCMD의 bit 12가 clear된 상황인 경우와 1518bytes보다 큰 경우이다.
READY_FOR_TX_NOW	0x0100	CS8900이 host로 부터 Tx를 위해서 frame을 받을 준비가 되었다는 것을 알려준다. 관련된 interrupt가 없기에 host는 CS8900이 Tx가 준비되었는지를 확인하기 위해서 polling을 해야한다.

표 141. PP_BusST ISA bus status bit definition

PP_BusST 레지스터는 Tx를 위한 bus의 상태를 알려주는 역할을 하는 레지스터이다. 예러 및 Tx ready 상태를 알려준다.

ID	Value	Description
RE_NEG_NOW	0x0040	다시 negotiation을 하라고 명령하는 bit이다.
ALLOW_FDX	0x0080	Full Duplex을 허가하는 bit이다.
AUTO_NEG_ENABLE	0x0100	Auto-negotiation을 enable시킨다.
NLP_ENABLE	0x0200	NLP(Normal Link Pulse)를 enable시키는 bit이다.
FORCE_FDX	0x8000	Full Duplex로 만들라고 설정하는 bit이다.
AUTO_NEG_BITS	-	FORCE_FDX NLP_ENABLE AUTO_NEG_ENABLE
AUTO_NEG_MASK	-	FORCE_FDX NLP_ENABLE AUTO_NEG_ENABLE ALLOW_FDX RE_NEG_NOW

표 142. PP_AutoNegCTL Auto negotiation control bit definition

PP_AutoNegCTL 레지스터는 link의 speed에 대해서 negotiation 설정을 담당하는 레지스터이다. 이 부분과 다음에 나오는 PP_AutoNegST를 사용해서 negotiation을 진행한다⁴⁵⁷.

ID	Value	Description
AUTO_NEG_BUSY	0x0080	Auto-negotiation이 busy인 상태를 나타내는 bit이다.
FLP_LINK	0x0100	Fast Link Pulse가 있음을 나타내는 bit이다.
FLP_LINK_GOOD	0x0800	Fast Link Pulse의 상태가 good임을 나타내는 bit이다.
LINKFAULT	0x1000	Link에 문제가 있음을 나타내는 bit이다.
HDX_ACTIVE	0x4000	Half Duplex로 active되어 있음을 나타내는 bit이다.
FDX_ACTIVE	0x8000	Full Duplex로 active되어 있음을 나타내는 bit이다.

표 143. PP_AutoNegST Auto negotiation status bit definition

⁴⁵⁷ 01 부분에 대한 CS8900의 spec.0이 정확히 나와 있지 않다.

PP_AutoNegST 레지스터는 auto negotiation의 상태 정보를 알려주는 레지스터이다. PP_AutoNegCTL을 설정한 후에 이 레지스터를 통해서 결과를 확인할 수 있다.

앞에서 우린 CS8900 ethernet controller에 대한 레지스터들에 대해서 알아보았다. 모든 레지스터들에 대해서 자세히 다 알수는 없지만, 어느정도의 감은 잡을 수 있을 것이다. 즉, 보내기와 받기, 그리고, buffer의 관리를 위해서 어떤 레지스터들을 설정해 줄 것인가와 어떻게 설정할 것인가에 대한 어느정도의 이해가 생겼으리라고 믿는다. 나중에 프로그램의 코드를 보게 될 때, 여기서 나열한 레지스터들이나 각 bit들에 대해서 실제로 어떻게 설정하는지를 유심히 보기 바란다.

17.8.2. Module

현재 CS8900의 디바이스 드라이버는 \${KERNEL_SRC}/drivers/net/cs89x0.c에 들어있다. 가장 먼저 보는 부분은 역시 모듈의 적재와 해제이다. 아래와 같다.

```
#ifdef MODULE
static struct net_device dev_cs89x0 = {
    "",
    0, 0, 0, 0,
    0, 0,
    0, 0, 0, NULL, NULL };

...
static int io=0;
static int irq=0;
static int debug=0;
static char media[8];
static int duplex=-1;

static int use_dma = 0;                                /* These generate unused var warnings if ALLOW_DMA = 0 */
static int dma=0;                                     /* or 64 */
static int dmasize=16;

...
int init_module(void)
{
    struct net_local *lp;
    int ret = 0;

    dev_cs89x0.irq = irq;
    dev_cs89x0.base_addr = io;
    dev_cs89x0.init = cs89x0_probe;
    dev_cs89x0.priv = kmalloc(sizeof(struct net_local), GFP_KERNEL);
    if (dev_cs89x0.priv == 0) {
        printk(KERN_ERR "cs89x0.c: Out of memory.\n");
        return -ENOMEM;
    }
    memset(dev_cs89x0.priv, 0, sizeof(struct net_local));
    lp = (struct net_local *)dev_cs89x0.priv;
#endif ALLOW_DMA
    if (use_dma) {
        lp->use_dma = use_dma;
        lp->dma = dma;
        lp->dmasize = dmasize;
    }
#endif
    spin_lock_init(&lp->lock);
    /* boy, they'd better get these right */
    if (!strcmp(media, "rj45"))
        lp->adapter_cnf = A_CNF_MEDIA_10B_T | A_CNF_10B_T;
```

```

else if (!strcmp(media, "aui"))
    lp->adapter_cnf = A_CNF_MEDIA_AUI | A_CNF_AUI;
else if (!strcmp(media, "bnc"))
    lp->adapter_cnf = A_CNF_MEDIA_10B_2 | A_CNF_10B_2;
else
    lp->adapter_cnf = A_CNF_MEDIA_10B_T | A_CNF_10B_T;
if (duplex == -1)
    lp->auto_neg_cnf = AUTO_NEG_ENABLE;
if (io == 0) {
    printk(KERN_ERR "cs89x0.c: Module autoprobing not allowed.\n");
    printk(KERN_ERR "cs89x0.c: Append io=0xNNN\n");
    ret = -EPERM;
    goto out;
}
#endif
if (use_dma && dmasize != 16 && dmasize != 64) {
    printk(KERN_ERR "cs89x0.c: dma size must be either 16K or 64K, not %dK\n", dmasize);
    ret = -EPERM;
    goto out;
}
#endif
if (register_netdev(&dev_cs89x0) != 0) {
    printk(KERN_ERR "cs89x0.c: No card found at 0x%x\n", io);
    ret = -ENXIO;
    goto out;
}
out:
if (ret)
    kfree(dev_cs89x0.priv);
return ret;
}

```

코드 1125. cs8900 모듈의 적재

전역변수인 dev_cs89x0에 기본적으로 설정된 인터럽트 번호(irq)와 I/O의 기본 주소(base address: io)를 보고하고, 초기화 함수(initialization function)으로는 cs89x0_probe() 함수를 설정한다. 또한 cs89x0 네트워크 디바이스 드라이버 고유로 사용할 메모리를 priv필드에 GFP_KERNEL option을 주어서 kmalloc() 함수를 호출해서 할당받는다. 만약 할당받지 못한다면 -ENOMEM을 돌려주고 복귀한다. 다시 할당받은 메모리 공간을 전부 0으로 초기화 하고(memset()), priv필드가 가르키는 값을 lp가 가지도록 한다. 만약 DMA를 사용한다면, 기본적인 설정 사항들을 저장하기 위해서, priv필드의 use_dma에 use_dma를 넣고, dma에는 dma를, dmasize에는 dmasize를 저장한다.

이전 cs89x0에서 사용할 spin lock에 대해서 초기화 해준다(spin_lock_init()). 전송 미디어를 선택하기 위해서 다음과 같은 일을 한다. 만약 media가 “rj45”와 같다면 priv필드의 adapter_cnf에 A_CNF_MEDIA_10B_T | A_CNF_10B_T를, “aui”와 같다면 A_CNF_MEDIA_AUI | A_CNF_AUI를, “bnc”와 같다면, A_CNF_MEDIA_10B2 | A_CNF_10B_2를, 해당 사항이 없다면 A_CNF_MEDIA_10B_T | A_CNF_10B_T를 넣어준다.

이전 Negotiation을 위한 준비를 위해서 만약 duplex가 -1로 설정되어 있다면 priv필드의 auto_neg_cnf에 AUTO_NEG_ENABLE(Auto-negotiation Enable)을 설정한다. 기본 I/O주소를 가지는 io가 0이라면, 에러 메시지를 출력하고 복귀값으로 -EPERM을 준 후, out으로 제어를 옮긴다. 다시 DMA를 사용할 경우에는 use_dma가 설정되어 있는지 확인하고, dmasize가 16이 아니며, dmasize가 64가 아닌 경우에는 다시 에러값으로 -ENXIO를 복귀코드로 두고 out으로 제어를 옮긴다.

모듈의 적재에서 마지막으로 해줄 일은 실제로 이 디바이스 드라이버가 network device driver임을 등록하는 것이다. register_netdev() 함수가 처리해줄 것이다. 만약 0이 아닌 값이 넘어온다면 복귀 값으로 -ENXIO를 두고 out으로 제어를 옮긴다.

out에서는 ret값을 확인해서 0이 아닌 값을 가진다면, 이전에 할당했던 private 데이터 구조를 해제한 후(kfree()) 복귀값을 넘겨주고 돌아간다.

```
void
cleanup_module(void)
{
    outw(PP_ChipID, dev_cs89x0.base_addr + ADD_PORT);
    if (dev_cs89x0.priv != NULL) {
        /* Free up the private structure, or leak memory :-) */
        unregister_netdev(&dev_cs89x0);
        kfree(dev_cs89x0.priv);
        dev_cs89x0.priv = NULL; /* gets re-allocated by cs89x0_probe1 */
        /* If we don't do this, we can't re-insmod it later. */
        release_region(dev_cs89x0.base_addr, NETCARD_IO_EXTENT);
    }
}
#endif /* MODULE */
```

코드 1126. cs8900모듈의 해제

복귀를 해제하는 일은 위에서 모듈의 등록에서 했던 일을 반대로 되돌리면 될 것이다. 먼저 outw()로 cs8900을 위한 기본 주소(base_addr)에 ADD_PORT를 더해서 PP_ChipID값을 적어 넣는다. 즉, ADD_PORT에 PP_ChipID값을 넣는 일이다. 따라서, base_addr + ADD_PORT에 PP_ChipID를 넣어서, PacketPage 레지스터들 중에서 PP_ChipID부분의 레지스터를 선택한다.

만약 이미 cs8900을 위한 private데이터 구조체를 할당했다면(Not NULL), 아직 네트워크 디바이스로 등록되어 있다고 보고, unregister_netdev()를 호출해서 등록을 해제한다. 또한 할당받은 private데이터 구조체를 해제하고(kfree) private필드는 NULL로 둔다. 마지막으로 cs8900을 위해서 사용을 요청한 I/O region을 해제해 주어야 하는데, 이것은 probe() 함수내지 open() 함수와 같은 곳에서 request_region()을 해서 할당받은 영역을 해제하는 역할을 한다. 즉, 기본 주소(base_addr)에서 얼마만큼을 cs8900의 레지스터들이 mapping되어 있는지를 예약하고, 다시 이것을 모듈의 해제에서 해제하는 일을 하는 것이다.

17.8.3. Probe

앞에서 initialization 함수로 cs89x0_probe()가 설정되어 있음을 모듈의 적재시에 보았다. 이 함수가 하는 역할은 실제로 시스템에서 CS8900을 사용한 ethernet controller가 있는지를 찾는 것이다. 아래와 같다.

```
int __init cs89x0_probe(struct net_device *dev)
{
    int i;
    int base_addr = dev ? dev->base_addr : 0;

    SET_MODULE_OWNER(dev);
    if (net_debug)
        printk("cs89x0:cs89x0_probe()\n");
    if (base_addr > 0x1ff) /* Check a single specified location. */
        return cs89x0_probe1(dev, base_addr);
    else if (base_addr != 0) /* Don't probe at all. */
        return -ENXIO;
    for (i = 0; netcard_portlist[i]; i++) {
        if (cs89x0_probe1(dev, netcard_portlist[i]) == 0)
            return 0;
    }
    printk(KERN_WARNING "cs89x0: no cs8900 or cs8920 detected. Be sure to disable PnP with SETUP\n");
    return -ENODEV;
}
```

코드 1127. cs89x0_probe()함수의 정의

넘겨받는 값은 net_device구조체를 가르키는 dev이다. 만약 dev가 NULL이라면, 0을 기본주소(base_addr)로 주고, 그렇지 않다면, dev의 base_addr로 설정한다. dev의 모듈 소유자는 SET_MODULE_OWNER()로 정의한다.

만약 특정 base_addr이 선택되어 있다면, 즉, base_addr값이 0x1FF보다 크다면, cs89x0_probe1()함수를 이용해서 찾도록 하고, 만약 base_addr이 0이 아닌 값을 가진다면, 에러가 되므로 -ENXIO를 돌려준다. 따라서, 0x1FF에서 0x01까지의 값이 base_addr의 값을 가질 경우에는 나머지가 진행될 것이다. 이것은 for loop를 돌면서 선언된 netcard_portlist[] 배열을 차례로 cs89x0_probe1()함수에 넘겨주면서 카드가 있는지를 검사하는 것이다. 찾았다면 0을 복귀 값으로 돌려줄 것이다. 카드를 찾지 못한다면 -ENODEV가 복귀 값으로 넘어갈 것이다.

따라서, 실제적인 chip을 검색하는 것은 cs89x0_probe1()함수가 맡고 있다. 아래와 같이 정의 된다. 이것을 보기 전에 충분한 chip에 대한 이해가 반드시 요구된다.

```
static int __init cs89x0_probe1(struct net_device *dev, int ioaddr)
{
    struct net_local *lp;
    static unsigned version_printed = 0;
    int i;
    unsigned rev_type = 0;
    int eeprom_buff[CHKSUM_LEN];
    int retval;

    /* Initialize the device structure. */
    if (dev->priv == NULL) {
        dev->priv = kmalloc(sizeof(struct net_local), GFP_KERNEL);
        if (dev->priv == 0) {
            retval = -ENOMEM;
            goto out;
        }
        lp = (struct net_local *)dev->priv;
        memset(lp, 0, sizeof(*lp));
        spin_lock_init(&lp->lock);
    #if !defined(MODULE) && (ALLOW_DMA != 0)
        if (g_cs89x0_dma) {
            lp->use_dma = 1;
            lp->dma = g_cs89x0_dma;
            lp->dmasize = 16; /* Could make this an option... */
        }
    #endif
    }
    lp = (struct net_local *)dev->priv;

    /* Grab the region so we can find another board if autoIRQ fails. */
    if (!request_region(ioaddr, NETCARD_IO_EXTENT, dev->name)) {
        retval = -EBUSY;
        goto out1;
    }
}
```

코드 1128. cs89x0_probe1()함수의 정의

`_init`로 선언된 함수는 초기화에서만 사용하고 더이상 사용되지 않는다는 것을 나타낸다. 이렇게 선언된 code부분은 커널이 초기화를 마치고나면 더이상 메모리를 차지할 이유가 없으므로 해제할 수 있다. cs89x0_probe1()함수가 넘겨받는 값은 net_device 구조체아 기본 I/O address를 가지는 ioaddr이다. 먼저 net_device구조체의 priv필드가 NULL이라면, network adapter의 private데이터 저장공간으로 사용할 부분을

커널 메모리에서 할당 받는다(kmalloc()). 만약 할당 받을 수 없다면 에러 값으로 -ENOMEM을 넣고, 제어를 out으로 옮긴다.

할당 받은 공간을 lp로 가르키고, 이 공간을 전부 0으로 설정한다(memset()). lp가 가르키는 곳에 있는 spin_lock에 대해서 초기화를 해주고(spin_lock_init()), 만약 MODULE이 정의되지 않았으며, DMA를 허락하지 않는다면, g_cs89x0_dma에 값이 있다면, lp의 use_dma에 1을 넣고, dma필드에는 g_cs89x0_dma를, dmasize에는 16을 넣는다. 이것은 DMA(Direct Memory Access)를 위한 기본 설정이다. 관련된 것으로는 아래와 같은 것이 정의되어 있다.

```
#if !defined(MODULE) && (ALLOW_DMA != 0)
static int g_cs89x0_dma;

static int __init dma_fn(char *str)
{
    g_cs89x0_dma = simple_strtol(str,NULL,0);
    return 1;
}

__setup("cs89x0_dma=", dma_fn);
#endif /* !defined(MODULE) && (ALLOW_DMA != 0) */
```

코드 1129. g_cs89x0_dma에 관련된 설정

즉, MODULE이 정의되지 않았고, DMA를 허가하지 않는다면, g_cs89x0_dma를 static 변수로 정의한다. dma_fn()함수는 단순히 g_cs89x0_dma에 넘겨받은 stirng의 값을 unsigned long값으로 변화해서 넣어주게 된다. __setup()으로 “cs89x0_dma=”과 dma_fn을 넘겨주어서, 커널이 부팅시에 cs89x0_dma에 어떤 string을 저장할 수 있도록 만든다. 나중에 이것은 다시 g_cs89x0_dma에 unsigned long으로 변화되어 저장된다. 즉, 이 값이 나중에 lp->dma에 들어갈 값이 될 것이다.

그럼 이제는 더 이상 진행하기 전에 net_device 구조체의 priv 필드에 들어가는 CS8900에 고유한 데이터 구조체를 살펴보도록 하자. 아래와 같다.

```
/* we allow the user to override various values normally set in the EEPROM */
#define FORCE_RJ45      0x0001    /* pick one of these three */
#define FORCE_AUI        0x0002
#define FORCE_BNC        0x0004

#define FORCE_AUTO       0x0010    /* pick one of these three */
#define FORCE_HALF       0x0020
#define FORCE_FULL       0x0030

/* Information that need to be kept for each board. */
struct net_local {
    struct net_device_stats stats;
    int chip_type;           /* one of: CS8900, CS8920, CS8920M */
    char chip_revision; /* revision letter of the chip ('A'...) */
    int send_cmd;            /* the proper send command: TX_NOW, TX_AFTER_381, or
TX_AFTER_ALL */
    int auto_neg_cnf;        /* auto-negotiation word from EEPROM */
    int adapter_cnf;         /* adapter configuration from EEPROM */
    int isa_config;          /* ISA configuration from EEPROM */
    int irq_map;             /* IRQ map from EEPROM */
    int rx_mode;              /* what mode are we in? 0, RX_MULTCAST_ACCEPT, or
RX_ALL_ACCEPT */
    int curr_rx_cfg;          /* a copy of PP_RxCFG */
    int linect;               /* either 0 or LOW_RX_SQUELCH, depending on configuration. */
    int send_underrun; /* keep track of how many underruns in a row we get */
    int force;                /* force various values; see FORCE* above. */
```

```

spinlock_t lock;
#endif ALLOW_DMA
    int use_dma;           /* Flag: we're using dma */
    int dma;               /* DMA channel */
    int dmasize;           /* 16 or 64 */
    unsigned char *dma_buff; /* points to the beginning of the buffer */
    unsigned char *end_dma_buff; /* points to the end of the buffer */
    unsigned char *rx_dma_ptr; /* points to the next packet */
#endif
};

```

코드 1130. CS8900의 private 데이터 구조체의 정의

net_local 구조체로 정의된 net_device구조체의 priv필드는 각각의 board마다 유지할 필요가 있는 정보들을 보관할 목적으로 device driver 프로그램이 인위적으로 만든 것이다. 먼저 net_device_stat를 보관할 stats와 chip의 타입을 보관할 chip_type, chip의 revision번호를 가질 chip_revision, 취해야 할 적절한 send명령을 보관하는 send_cmd, EEPROM으로부터 읽어온 auto-negotiation과 adapter configuration 정보를 저장할 auto_neg_cfg와 adapter_cfg, EEPROM으로 부터 읽어온 ISA bus 설정과 IRQ map을 저장하는 isa_config와 irq_map, 현재 Rx의 모드를 가지는 rx_mode, 현재의 PP_RxCFG의 값을 저장하는 curr_rx_cfg, 현재의 line control 정보를 가지는 linectl, send에서 발생한 underrun의 수를 기록하는 send_underrun, media 인터페이스 및 negotiation의 force값을 가지는 force, 마지막으로 spin_lock_t을 가지는 lock이 있다. 만약 DMA를 허용한다면 다시 use_dma를 써서 DMA를 사용하고 있음을 나타내고, dma로는 DMA channel 번호를, dmasize로는 한번의 DMA에서 이동되는 데이터의 크기를 나타낸다. 또한 DMA를 위해서 사요할 buffer는 dma_buff로 시작점을 가르키며, end_dma_buff로는 버퍼의 마지막을 가르킨다. 또한 rx_dma_ptr로는 Rx시에 다음에 어느곳에서 frame을 저장할지를 나타낸다.

```

/* if they give us an odd I/O address, then do ONE write to
   the address port, to get it back to address zero, where we
   expect to find the EISA signature word. An IO with a base of 0x3
   will skip the test for the ADD_PORT. */
if (ioaddr & 1) {
    if ((ioaddr & 2) != 2)
        if ((inw((ioaddr & ~3)+ ADD_PORT) & ADD_MASK) != ADD_SIG) {
            retval = -ENODEV;
            goto out2;
        }
    ioaddr &= ~3;
    outw(PP_ChipID, ioaddr + ADD_PORT);
}
if (inw(ioaddr + DATA_PORT) != CHIP_EISA_ID_SIG) {
    retval = ENODEV;
    goto out2;
}
/* Fill in the 'dev' fields. */
dev->base_addr = ioaddr;
/* get the chip type */
rev_type = readreg(dev, PRODUCT_ID_ADD);
lp->chip_type = rev_type &~ REVISION_BITS;
lp->chip_revision = ((rev_type & REVISION_BITS) >> 8) + 'A';
/* Check the chip type and revision in order to set the correct send command
CS8920 revision C and CS8900 revision F can use the faster send. */
lp->send_cmd = TX_AFTER_381;
if (lp->chip_type == CS8900 && lp->chip_revision >= 'F')
    lp->send_cmd = TX_NOW;
if (lp->chip_type != CS8900 && lp->chip_revision >= 'C')
    lp->send_cmd = TX_NOW;

```

```

if (net_debug && version_printed++ == 0)
    printk(version);
printk(KERN_INFO "%s: cs89%c0%sc rev %c found at %#3lx ",
       dev->name,
       lp->chip_type==CS8900?'0':'2',
       lp->chip_type==CS8920M?"M":"",
       lp->chip_revision,
       dev->base_addr);
reset_chip(dev);

```

코드 1131. cs89x0_probe1()함수의 정의(계속)

만약, 넘겨받은 ioaddr이 허수라면, ioaddr에 2를 AND시켜서 2가 나오는지 확인한다. 만약 2가 나온다면, 이는 적어도 3이란 값이 ioaddr의 마지막에 있음을 알 수 있다. 따라서, ioaddr의 보정을 위해서 ioaddr에 3을 NOT한 값을 AND시켜서 이것에 ADD_PORT를 더한 후, ADD_MASK를 시킨다. 이렇게 나온 값이 ADD_SIG와 같지 않다면, ioaddr은 잘못된 주소를 가르키고 있는 것이 되므로, 복귀값에 -ENODEV를 넣고 out2로 제어를 옮긴다. 단순히 허수인 값만을 가진다면 ioaddr에 3을 NOT시켜서 다시 이것을 ioaddr에 AND시킨다. 즉, 적어도 4byte단위로 나누어 떨어지는 주소를 가지도록 만든다. 기본 I/O 주소가 올바른 값을 가지도록 ioaddr + ADD_PORT에 PP_ChipID를 쓴다. 이렇게 해서 ioaddr이 정확히 입출력을 위한 port의 기본 주소를 정확하게 가르키게 만든다.

이렇게 구한 ioaddr주소를 디바이스의 기본 주소로 만든다(base_addr). 이전 chip의 type을 결정하기 위해서 PRODUCT_ID_ADD를 인자로 넘겨주어서 readreg()함수를 호출한다. 기본적으로 net_device구조체의 base_addr를 사용하기에 앞에서 이 값을 먼저 결정해 주었다. 함수의 호출 결과 값은 rev_type에 저장된다. 여기서 revision정보를 얻기위해서 REVISION_BITS를 NOT시켜서 rev_type에 AND시킨다. 이 값으로 chip_type이 결정되며, 다시 REVISION_BITS를 rev_type에 AND시켜서 우측으로 8bit를 shift한 후, 'A'와 더해서 chip_revision을 결정한다. 이것은 PacketPage 메모리 맵을 참고하면 간단히 알 수 있는 것이다.

이전 각각의 chip type에 따라서 다른 Tx를 위한 command를 설정할 차례이다. revision번호에 따라서, TX_NOW나 TX_AFTER_3810이 send command로 들어간다. 나먼지는 얻은 정보를 print하는 부분이며 칩을 초기화 시키기 위해서 reset_chip()함수⁴⁵⁸를 호출해서 reset시킨다.

```

if ((readreg(dev, PP_SelfST) & (EEPROM_OK | EEPROM_PRESENT)) ==
    (EEPROM_OK|EEPROM_PRESENT)) {
    /* Load the MAC. */
    for (i=0; i < ETH_ALEN/2; i++) {
        unsigned int Addr;
        Addr = readreg(dev, PP_IA+i*2);
        dev->dev_addr[i*2] = Addr & 0xFF;
        dev->dev_addr[i*2+1] = Addr >> 8;
    }
    lp->adapter_cnf = 0;
    i = readreg(dev, PP_LineCTL);
    /* Preserve the setting of the HCB1 pin. */
    if ((i & (HCB1 | HCB1_ENBL)) == (HCB1 | HCB1_ENBL))
        lp->adapter_cnf |= A_CNF_DC_DC_POLARITY;
    /* Save the sqelch bit */
    if ((i & LOW_RX_SQUELCH) == LOW_RX_SQUELCH)
        lp->adapter_cnf |= A_CNF_EXTND_10B_2 | A_CNF_LOW_RX_SQUELCH;
    /* Check if the card is in 10Base-t only mode */
    if ((i & (AUI_ONLY | AUTO_AUI_10BASET)) == 0)
        lp->adapter_cnf |= A_CNF_10B_T | A_CNF_MEDIA_10B_T;
    /* Check if the card is in AUI only mode */

```

⁴⁵⁸ 이 함수는 소프트웨어적으로 ethernet controller를 reset시키는 함수이다. 정의는 나중에 다시 볼 것이다.

```

        if ((i & (AUI_ONLY | AUTO_AUI_10BASET)) == AUI_ONLY)
            lp->adapter_cnf |= A_CNF_AUI | A_CNF_MEDIA_AUI;
        /* Check if the card is in Auto mode. */
        if ((i & (AUI_ONLY | AUTO_AUI_10BASET)) == AUTO_AUI_10BASET)
            lp->adapter_cnf |= A_CNF_AUI | A_CNF_10B_T |
                A_CNF_MEDIA_AUI | A_CNF_MEDIA_10B_T | A_CNF_MEDIA_AUTO;
        /* IRQ. Other chips already probe, see below. */
        if (lp->chip_type == CS8900)
            lp->isa_config = readreg(dev, PP_CS8900_ISAINT) & INT_NO_MASK;
        printk( "[Cirrus EEPROM] ");
    }
    printk("\n");
}

```

코드 1132. cs89x0_probe1() 함수의 정의(계속)

PP_SelfST 레지스터를 읽어서 EEPROM이 있으며(EEPROM_PRESENT), 올바르게 동작(EEPROM_OK)한다는 것을 확인한다. EEPROM이 존재한다면, MAC address를 EEPROM에서 읽어오도록 한다. Ethernet address는 6 bytes로 이루어졌기에, 3번에 걸쳐서 2 bytes씩 읽어들인다. 읽어온 ethernet주소는 dev->dev_addr[]에 저장한다.

이전 전송 media인 line의 상태를 알아볼 차례이다. 먼저 priv에 할당받은 adapter_cnf필드를 0으로 초기화시킨다. 나머지 부분은 해당하는 설정이 있는지를 확인한 후, 이를 adapter_cnf필드에 저장하는 일을 한다. PP_LineCTL 레지스터를 읽어서, HCB1과 HCB1_ENBL⁴⁵⁹이 설정되어 있는지를 확인한다. 설정되어 있다면, adapter_cnf에 A_CNF_DC_DC_POLARITY⁴⁶⁰를 설정한다. 같은 방법으로 LOW_RX_SQUELCH가 설정된 경우에는 A_CNF_EXTND_10B_2와 A_CNF_LOW_RX_SQUELCH를 설정하고, AUI_ONLY과 AUTO_AUI_10BASET가 설정되지 않은 경우에는 A_CNF_10B_T와 A_CNF_MEDIA_10B_T를, AUI_ONLY과 AUTO_AUI_10BASET에서 AUI_ONLY이 설정된 경우에는 A_CNF_AUI와 A_CNF_MEDIA_AUI를 설정하며, AUI_ONLY과 AUTO_AUI_10BASET에서 AUTO_AUI_10BASET가 설정된 경우에는 A_CNF_AUI와 A_CNF_10B_T, A_CNF_MEDIA_AUI, A_CNF_MEDIA_10B_T, A_CNF_MEDIA_AUTO를 설정한다. 또한 chip type이 CS8900이라면, isa_config필드에 PP_CS8900_ISAINT를 읽어서, Interrupt masking을 한 후, 해당하는 값을 저장한다. 여기서 AUI와 관련되어 해준 일을 요약하면 아래의 표와 같다.

AUI_ONLY	AUTO_AUI_10BASET	Physical Interface
1	N/A	AUI
0	0	10BASE_T
0	1	Auto-Select

표 144. Selecting Media for CS89x0 Ethernet Controller Chip

따라서, AUI_ONLY이 설정되어 있고, AUTO_AUI_10BASET가 설정되어 있느냐에 따라서, 해당하는 물리적인 interface가 달라진다.

```

/* First check to see if an EEPROM is attached. */
if ((readreg(dev, PP_SelfST) & EEPROM_PRESENT) == 0)
    printk(KERN_WARNING "cs89x0: No EEPROM, relying on command line....\n");
else if (get_eeprom_data(dev, START_EEPROM_DATA, CHKSUM_LEN, eeprom_buff) < 0) {
    printk(KERN_WARNING "\ncs89x0: EEPROM read failed, relying on command line.\n");
} else if (get_eeprom_cksum(START_EEPROM_DATA, CHKSUM_LEN, eeprom_buff) < 0) {
    /* Check if the chip was able to read its own configuration starting
       at 0 in the EEPROM*/
}

```

⁴⁵⁹ HCB0는 network link의 상태(HC0)를 감지하는데 사용하며, HCE1일 설정된 상황에서 Self Control Register에 있는 HCB0을 설정해서 구동한다. HCB1은 bus의 상태(HC1)를 감지하는 것으로 Self Control Register의 HCB1을 설정해서 구동한다.

⁴⁶⁰ 이것이 설정되면,

```

        if ((readreg(dev, PP_SelfST) & (EEPROM_OK | EEPROM_PRESENT)) !=  

            (EEPROM_OK|EEPROM_PRESENT))  

            printk(KERN_WARNING "cs89x0: Extended EEPROM checksum bad and no Cirrus  

EEPROM, relying on command line\n");  

    } else {  

        /* This reads an extended EEPROM that is not documented  

           in the CS8900 datasheet. */  

        /* get transmission control word but keep the autonegotiation bits */  

        if (!lp->auto_neg_cnf) lp->auto_neg_cnf = eeprom_buff[AUTO_NEG_CNF_OFFSET/2];  

        /* Store adapter configuration */  

        if (!lp->adapter_cnf) lp->adapter_cnf = eeprom_buff[ADAPTER_CNF_OFFSET/2];  

        /* Store ISA configuration */  

        lp->isa_config = eeprom_buff[ISA_CNF_OFFSET/2];  

        dev->mem_start = eeprom_buff[PACKET_PAGE_OFFSET/2] << 8;  

        /* eeprom_buff has 32-bit ints, so we can't just memcpy it */  

        /* store the initial memory base address */  

        for (i = 0; i < ETH_ALEN/2; i++) {  

            dev->dev_addr[i*2] = eeprom_buff[i];  

            dev->dev_addr[i*2+1] = eeprom_buff[i] >> 8;  

        }  

    }
}

```

코드 1133. cs89x0_probe1()함수의 정의(계속)

다시 PP_SelfST를 읽어서, EEPROM이 있는지 확인한 후, 없다면, warning 메시지를 출력한다. 이 경우에는 command line상에서 주어진 argument를 사용하는데, 현재는 이것이 bug로 존재한다. 읽어서 넣는 부분을 추가해줄 필요가 있다.

있다면, 해당하는 EEPROM 데이터 영역을 읽는다(get_eeprom_data()). 읽을 수 없다면, 다시 warning 메시지를 출력하고, 다시 EEPROM의 CRC영역을 읽는다(get_eeprom_cksum()). checksum에 문제가 있다면, 다시 warning을 출력해서 확장 EEPROM의 checksum에 문제가 있음을 표시한다. 아무런 문제가 없다면, 이하의 else절을 수행한다. 읽어온 EEPROM의 데이터는 eeprom_buff에 저장된다.

만약 auto_neg_cnf가 0라면(Auto negotiation), auto_neg_cnf에 EEPROM의 해당 부분을 copy한다. 또한, adapter_cnf가 0이라면, 다시 network adapter부분의 설정을 읽어온다. ISA관련 설정에 대한 것 및 메모리 사용에 대한 시작번지(mem_start)도 읽어서 저장한다. 마지막으로 physical network 주소인 ethernet address부분을 읽어서 dev->dev_addr에 저장한다.

get_eeprom_data()함수와 get_eeprom_cksum()함수는 EEPROM으로부터 데이터를 읽고, checksum을 비교하는 함수로서 아래와 같이 정의되어 있다.

```

static int __init get_eeprom_data(struct net_device *dev, int off, int len, int *buffer)  

{  

    int i;  

    if (net_debug > 3) printk("EEPROM data from %x for %x:\n", off, len);  

    for (i = 0; i < len; i++) {  

        if (wait_eeprom_ready(dev) < 0) return -1;  

        /* Now send the EEPROM read command and EEPROM location to read */  

        writereg(dev, PP_EECMD, (off + i) | EEPROM_READ_CMD);  

        if (wait_eeprom_ready(dev) < 0) return -1;  

        buffer[i] = readreg(dev, PP_EEData);  

        if (net_debug > 3) printk("%04x ", buffer[i]);  

    }  

    if (net_debug > 3) printk("\n");  

    return 0;  

}  

static int __init get_eeprom_cksum(int off, int len, int *buffer)

```

```
{
    int i, cksum;

    cksum = 0;
    for (i = 0; i < len; i++)
        cksum += buffer[i];
    cksum &= 0xffff;
    if (cksum == 0)
        return 0;
    return -1;
}
```

코드 1134. get_eeprom_data()함수와 get_eeprom_cksum()함수의 정의

get_eeprom_data()함수는 EEPROM에서 정해진 만큼의 데이터를 읽어오는 일을 한다. wait_eeprom_ready() 함수를 호출해서 eeprom이 ready인지를 확인하고, PP_EECMD 레지스터에 해당 offset과 EEPROM_READ_CMD를 OR시켜서 쓴다. 다시 EEPROM이 ready상태인지를 확인한 후, PP_EEData 레지스터를 읽어서 buffer에 저장한다.

get_eeprom_cksum()은 읽어온 EEPROM의 데이터를 다 더해서 0xFFFF와 AND시켜서 이 값이 0을 가진다면 0을 돌려주고, 그렇지 않다면 -1을 돌려준다. 즉, 읽어온 데이터의 합은 0이 되어야 맞다.

아래에 보이는 것은 wait_eeprom_ready()함수로서 EEPROM에 대한 연산을 수행하기 전에 반드시 EEPROM이 준비된 상태에 있는지를 확인하기 위해서 사용한다.

```
static int __init
wait_eeprom_ready(struct net_device *dev)
{
    int timeout = jiffies;
    /* check to see if the EEPROM is ready, a timeout is used -
       just in case EEPROM is ready when SI_BUSY in the
       PP_SelfST is clear */
    while(readreg(dev, PP_SelfST) & SI_BUSY)
        if (jiffies - timeout >= 40)
            return -1;
    return 0;
}
```

코드 1135. wait_eeprom_ready()함수의 정의

wait_eeprom_ready()함수는 EEPROM에 대해서 timeout(=40 ticks)이 되기전에 PP_SelfST 레지스터를 읽어서 SI_BUSY인지를 확인하는 함수이다. 만약 timeout이되면 -1을 돌려주며, 그렇지 않고, SI_BUSY가 PP_SelfST에서 clear되면 0을 돌려줘서 준비가 되었음을 나타낸다.

```
/* allow them to force multiple transceivers. If they force multiple, autosense */
{
    int count = 0;
    if (lp->force & FORCE_RJ45)      {lp->adapter_cnf |= A_CNF_10B_T; count++; }
    if (lp->force & FORCE_AUI)       {lp->adapter_cnf |= A_CNF_AUI; count++; }
    if (lp->force & FORCE_BNC)       {lp->adapter_cnf |= A_CNF_10B_2; count++; }
    if (count > 1)                  {lp->adapter_cnf |= A_CNF_MEDIA_AUTO; }
    else if (lp->force & FORCE_RJ45){lp->adapter_cnf |= A_CNF_MEDIA_10B_T; }
    else if (lp->force & FORCE_AUI) {lp->adapter_cnf |= A_CNF_MEDIA_AUI; }
    else if (lp->force & FORCE_BNC) {lp->adapter_cnf |= A_CNF_MEDIA_10B_2; }

    printk(KERN_INFO "cs89x0 media %s%s%s",
           (lp->adapter_cnf & A_CNF_10B_T)? "RJ-45,":"",
           (lp->adapter_cnf & A_CNF_AUI)? "AUI,":"",
           (lp->adapter_cnf & A_CNF_10B_2)? "BNC,":");
}
```

```
lp->irq_map = 0xffff;
```

코드 1136. cs89x0_probe1()함수의 정의(계속)

이전 media 타입이 어떤 것을 사용하고 있는지를 확인하는 것이다. 만약 여러개의 transceiver를 사용한다면, auto-sensing을 하도록 만든다. 즉, A_CNF MEDIA AUTO를 adapter_cnf에 설정한다. 또한 irq_map에 0xFFFF를 설정한다. 나중에 이 값을 interrupt의 설정에서 사용할 것이다.

```
/* If this is a CS8900 then no pnp soft */
if (lp->chip_type != CS8900 &&
    /* Check if the ISA IRQ has been set */
    (i = readreg(dev, PP_CS8920_ISAINT) & 0xff,
     (i != 0 && i < CS8920_NO_INTS))) {
    if (!dev->irq)
        dev->irq = i;
} else {
    i = lp->isa_config & INT_NO_MASK;
    if (lp->chip_type == CS8900) {
        /* Translate the IRQ using the IRQ mapping table. */
        if (i > sizeof(cs8900_irq_map)/sizeof(cs8900_irq_map[0]))
            printk("\ncs89x0: bug: isa_config is %d\n", i);
        else
            i = cs8900_irq_map[i];
        lp->irq_map = CS8900_IRQ_MAP; /* fixed IRQ map for CS8900 */
    } else {
        int irq_map_buff[IRQ_MAP_LEN/2];
        if (get_eeprom_data(dev, IRQ_MAP_EEPROM_DATA,
                            IRQ_MAP_LEN/2,
                            irq_map_buff) >= 0) {
            if ((irq_map_buff[0] & 0xff) == PNP_IRQ_FRMT)
                lp->irq_map = (irq_map_buff[0]>>8) | (irq_map_buff[1] << 8);
        }
    }
    if (!dev->irq)
        dev->irq = i;
}
```

코드 1137. cs89x0_probe1()함수의 정의(계속)

만약 chip type이 CS8900이 아니라면, CS8920의 인터럽트 설정을 레지스터에서 읽는다. 이 값이 CS8920_NO_INTS(Maximum CS8920 interrupt select 번호=0x0F)보다 작고, 0이 아니라면, net_device 구조체에 설정된 irq가 0일 때는 이 값을 사용한다. 위의 조건 중에 하나라도 맞지 않는다면 else이하를 실행하는데, 먼저 isa_config에서 INT_NO_MASK를 씌워서 인터럽트 번호를 가져온다. 만약 chip type이 CS8900이라면, IRQ를 IRQ mapping table을 통해서 해당하는 irq번호로 변환한다. 변환에 사용된 IRQ mapping은 CS8900_IRQ_MAP으로 정정해서 irq_map에 넣는다. 만약 chip type이 CS8900이 아니라면, irq_map_buff를 할당 받아서, 이곳에 EEPROM의 IRQ_MAP_EEPROM_DATA로부터 읽어온 데이터로 채운다(get_eeprom_data()). 읽어온 데이터가 있고, irq_map_buff[0]과 0xFF를 AND한 값이 PNP_IRQ_FRMT와 같다면, irq_map에는 EEPROM에서 읽어온 데이터의 첫번째 것을 우측으로 8bit shift한 값과 두번째 것을 좌측으로 8bit shift한 값을 OR시켜서 넣는다. 만약 dev->irq가 0을 가진다면, 해당하는 찾은 index를 가지고 dev->irq를 설정한다(i).

```
printk(" IRQ %d", dev->irq);
#endif ALLOW_DMA
if (lp->use_dma) {
    get_dma_channel(dev);
    printk(" DMA %d", dev->dma);
```

```

    }
else
#endif
{
    printk(", programmed I/O");

    /* print the ethernet address. */
    printk(", MAC ");
    for (i = 0; i < ETH_ALEN; i++)
    {
        printk("%s%02x", i ? ":" : "", dev->dev_addr[i]);
    }
}

```

코드 1138. cs89x0_probe1()함수의 정의(계속)

이전 설정된 IRQ에 대한 정보를 print하고, ALLOW_DMA가 설정된 경우에는 net_device의 priv필드의 use_dma값이 있다면, get_dma_channel()을 호출해서 네트워크 디바이스 드라이버를 위한 DMA channel을 할당 받는다. 할당받은 channel을 print하는 것도 잊지 말자. 이것은 단순히 디버깅을 위한 차원이지만, 사용자에게도 많은 정보를 줄 수 있다. 만약 DMA를 사용하도록 되어있지 않다면 programmed I/O를 사용해서 data의 패킷의 전송과 수신을 처리한다. 나머지는 단순히 디바이스의 MAC 주소를 print하는 것뿐이다.

```

dev->open          = net_open;
dev->stop          = net_close;
dev->tx_timeout    = net_timeout;
dev->watchdog_timeo = HZ;
dev->hard_start_xmit = net_send_packet;
dev->get_stats     = net_get_stats;
dev->set_multicast_list = set_multicast_list;
dev->set_mac_address = set_mac_address;
/* Fill in the fields of the device structure with ethernet values. */
ether_setup(dev);
printk("\n");
if (net_debug)
    printk("cs89x0_probe1() successful\n");
return 0;
out2:
release_region(ioaddr, NETCARD_IO_EXTENT);
out1:
kfree(dev->priv);
dev->priv = 0;
out:
return retval;
}

```

코드 1139. cs89x0_probe1()함수의 정의(계속)

이전 커널이 디바이스 드라이버와의 인터페이스로 사용하는 함수들에 대한 정의를 해줄 차례이다. open, stop, tx_timeout, hard_start_xmit, get_stats, set_multicast_list, set_mac_address등이 이에 해당되며, watchdog_timeo은 watchdog timer의 timeout interval로 사용하는 값으로 HZ(=100)를 주었다. 나머지 net_device 구조체의 필드를 ethernet 디바이스 드라이버를 위한 기본 설정으로 마춰주기 위해서, ether_setup()함수에 net_device 구조체를 넘겨주어서 호출한다.

제대로 지금까지의 연산이 진행되었다면, 0을 복귀값으로 돌려줄 것이며, out2의 경우에는 이미 I/O region에 대한 할당이 있었을 경우(request_region()), 이를 원상태로 되돌리기 위해서 release_region()함수를 호출해준다. 만약 net_device구조체의 priv필드를 위해서 할당받은 공간이 있다면, 이것을 해제해주기

위해서 kfree()함수를 호출한다. 오류가 있을 경우에는 return값으로 오류가 생성된 위치에서 설정한 retval를 넘겨준다.

참고로 이곳에서 커널과 network device driver간의 interface를 정의해 주었으므로, 이전 open() 및 close()함수등의 호출이 가능해졌다. 즉, 이전 디바이스를 실제로 열어서 사용할 수 있는 모든 준비 절차가 다 되었다.

17.8.4. Reset

칩은 실제로 언제든지 소프트웨어적으로 reset이 가능하다. 이런 경우로는 칩의 초기화나 심각한 에러 상황을 발견했을 경우에 해당하며, 칩 자체의 bug를 해결하는 좋은 방법이 될 수도 있을 것이다. 하지만, 주의할 점은 너무 자주 이런 상황이 발생한다면 좀 곤란할 것이다. 사용자가 모르는 상황에서 네트워크 chip의 reset이 가능하다는 점만 알고 있도록 하자.

```
void __init reset_chip(struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;
    int ioaddr = dev->base_addr;
    int reset_start_time;

    writereg(dev, PP_SelfCTL, readreg(dev, PP_SelfCTL) | POWER_ON_RESET);
    /* wait 30 ms */
    current->state = TASK_INTERRUPTIBLE;
    schedule_timeout(30*HZ/1000);
    if (lp->chip_type != CS8900) {
        /* Hardware problem requires PNP registers to be reconfigured after a reset */
        outw(PP_CS8920_ISAINT, ioaddr + ADD_PORT);
        outb(dev->irq, ioaddr + DATA_PORT);
        outb(0, ioaddr + DATA_PORT + 1);
        outw(PP_CS8920_ISAMemB, ioaddr + ADD_PORT);
        outb((dev->mem_start >> 16) & 0xff, ioaddr + DATA_PORT);
        outb((dev->mem_start >> 8) & 0xff, ioaddr + DATA_PORT + 1);
    }
    /* Wait until the chip is reset */
    reset_start_time = jiffies;
    while( (readreg(dev, PP_SelfST) & INIT_DONE) == 0 && jiffies - reset_start_time < 2)
        ;
}
```

코드 1140. reset_chip()함수의 정의

먼저 chip을 reset하기 위해서는 readreg()로 PP_SelfCTL을 읽어온다. 이 레지스터는 chip의 자가 제어(self control)를 위해서 사용하는 레지스터로, 이중에서 POWER_ON_RESET bit를 설정한다(writereg()). Reset 이후에 반응이 나타나기를 기다리기 위해서 30ms를 기다린다. 즉, 현재 수행중인 프로세스의 상태를 TASK_INTERRUPTIBLE로 만든 후, schedule_timeout()함수에 30ms값⁴⁶¹에 해당하는 것을 넘겨주어서 호출된다. 나중에 timeout이 되면, TASK_INTERRUPTIBLE로 현재의 task 상태이므로 깨어나게 될 것이며, schedule_timeout()이후에서 실행을 계속할 것이다.

만약 chip_type이 CS8900이 아니라면, if()절을 수행한다. 즉, CS8900이 아닌 경우에는 PNP에 관련된 레지스터들을 reset이후에 재설정해 주어야 한다. 먼저 PP_CS8920_ISAINT를 base I/O address + ADD_PORT에 넣어서, PP_CS8920_ISAINT에 대해서 쓰는 연산을 할 것이라는 것을 알려주고, 다시 base I/O address + DATA_PORT에 IRQ값으로 설정된 byte를 넣어준다. 바로 다음에 위치한 byte에는 0을 준다. 다시 PP_CS8920_ISAMemB를 base I/O address + ADD_PORT에 넣어서, PP_CS8920_ISAMemB(공유된

⁴⁶¹ CS8900의 spec.에서는 유효함과 설정을 위해서 on-chip analog circuitry와 EEPROM을 읽는데, 일반적으로 10ms정도의 시간이 필요하다고 한다. 이것을 마치고 나면, Self Status Register의 INITD bit이 설정되어 reset이 마무리가 되었다는 것을 표시한다. 또한 SIBUSY bit은 EEPROM이 설정을 마쳤다는 것을 보고해준다.

메모리의 Base Memory Address)에 write함을 나타낸 후, net_device구조체의 mem_start(unsigned long : 32bits) 상위 2 bytes중 하위를 차지하는 1byte를 base I/O address + DATA_PORT에 넣고, 다시 그 다음 byte를 base I/O address + DATA_PORT + 1에 넣는다. 이렇게 해줌으로서 interrupt는 몇번을 사용할지와 데이터의 전송을 위해서 사용할 메모리 주소가 어디에 있는지를 알려주게 된다.

reset이 일어난 시점을 기록하는 reset_start_time에는 현재의 시간을 기입한다(jiffies). 그리고, 초기화가 완전히 일어날 때까지 기다리기 위해서 PP_SelfST을 계속 읽어서 INIT_DONE이 설정되거나, 현재 시간에서 이전에 reset이 된 시점의 시간이 2 ticks를 초과할 때까지 기다린다.

17.8.5. Open

Open은 네트워크 디바이스의 초기화(initialization) 이후에 실제로 네트워크 interface를 사용하기 위해서 가장 먼저 호출되는 함수이다. 따라서, 앞에서 초기화 과정에서 해주지 않았던 초기화와 관련된 설정과, 버퍼등의 관리를 위해서 해주어야 할 일을 및 interrupt를 사용가능한 상태로 만들어 주어야 한다.

```
static int net_open(struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;
    int result = 0;
    int i;
    int ret;

    if (dev->irq < 2) {
        /* Allow interrupts to be generated by the chip */
    /* Cirrus' release had this: */
#ifndef 0
        writereg(dev, PP_BusCTL, readreg(dev, PP_BusCTL)|ENABLE_IRQ );
#endif
    /* And 2.3.47 had this: */
        writereg(dev, PP_BusCTL, ENABLE_IRQ | MEMORY_ON);
        for (i = 2; i < CS8920_NO_INTS; i++) {
            if ((1 << dev->irq) & lp->irq_map) {
                if (request_irq(i, net_interrupt, 0, dev->name, dev) == 0) {
                    dev->irq = i;
                    write_irq(dev, lp->chip_type, i);
                    /* writereg(dev, PP_BufCFG, GENERATE_SW_INTERRUPT); */
                    break;
                }
            }
        }
        if (i >= CS8920_NO_INTS) {
            writereg(dev, PP_BusCTL, 0);          /* disable interrupts. */
            printk(KERN_ERR "cs89x0: can't get an interrupt\n");
            ret = -EAGAIN;
            goto bad_out;
        }
    } else {
        if (((1 << dev->irq) & lp->irq_map) == 0) {
            printk(KERN_ERR "%s: IRQ %d is not in our map of allowable IRQs, which is %x\n",
                   dev->name, dev->irq, lp->irq_map);
            ret = -EAGAIN;
            goto bad_out;
        }
    /* FIXME: Cirrus' release had this: */
        writereg(dev, PP_BusCTL, readreg(dev, PP_BusCTL)|ENABLE_IRQ );
    /* And 2.3.47 had this: */
#ifndef 0
        writereg(dev, PP_BusCTL, ENABLE_IRQ | MEMORY_ON);
#endif
    }
}
```

```
#endif
        write_irq(dev, lp->chip_type, dev->irq);
        ret = request_irq(dev->irq, &net_interrupt, 0, dev->name, dev);
        if (ret) {
            if (net_debug)
                printk(KERN_DEBUG "cs89x0: request_irq(%d) failed\n", dev->irq);
            goto bad_out;
        }
    }
```

코드 1141. net_open()함수의 정의

네트워크 디바이스 드라이버에 설정된 irq값이 2보다 작다면, if() 부분을 실행한다. 먼저 PP_BusCTL에 레지스터에 ENABLE_IRQ와 MEMORY_ON을 설정한다. 그리고나서, 해당하는 인터럽트를 할당 받기 위해서 for loop를 돌면서, CS8920_NO_INTS가 될 때까지 다음과 같은 일을 한다. 먼저 irq로 1을 shift해서 net_device의 priv부분에 정의한 irq_map과 AND 시켜서 어떤 값이 나온다면, 이 값을 가지고 인터럽트를 요청한다(request_irq()). 제대로 요청이 처리가 되었다면, 0을 돌려줄 것이며, 이 값을 우리가 할당받은 인터럽트로 기입한다(dev->irq=i). 그리고나서 PP_BufCFG에 이 값을 인터럽트로 사용한다고 알려준다(write_irq()). request_irq()함수로 넘겨주는 값은, 인터럽트 번호와 인터럽트 핸들러의 주소, flag을 나타내는 long값인 상수, 디바이스의 이름, 그리고 net_device구조체의 포인터이다. 나중에 인터럽트 핸들러가 호출되면, 해당하는 인터럽트의 번호와 넘겨준 net_device구조체의 포인터를 핸들러가 넘겨받을 것이다. 만약 CS8920_NO_INTS에서 할당 받을 수 있는 interrupt가 없다면, PP_BusCTL에 0을 써서, interrupt를 disable시키고 복귀값으로 -EAGAIN을 넣은 후 제어를 bad_out으로 옮긴다.

dev->irq값이 2보다 크다면, 1을 dev->irq만큼 shift해서 lp->irq_map과 AND시킨다. 만약 0이 나온다면, 복귀값을 -EAGAIN으로 두고 bad_out으로 제어를 옮긴다. 그렇지 않다면, 이하를 수행하는데, 먼저 PP_BusCTL에 PP_BusCTL에서 읽은 값과 ENABLE_IRQ를 OR시켜서 넣는다. 그리고나서, buffer관리를 위해서 사용하는 interrupt번호가 무엇인지를 알려주고(write_irq()), 해당 인터럽트를 요청한다(request_irq()). 인터럽트 요청이 받아들려지지 않는다면, bad_out으로 제어를 옮긴다.

```
static void write_irq(struct net_device *dev, int chip_type, int irq)
{
    int i;

    if (chip_type == CS8900) {
        /* Search the mapping table for the corresponding IRQ pin. */
        for (i = 0; i != sizeof(cs8900_irq_map)/sizeof(cs8900_irq_map[0]); i++)
            if (cs8900_irq_map[i] == irq)
                break;
        /* Not found */
        if (i == sizeof(cs8900_irq_map)/sizeof(cs8900_irq_map[0]))
            i = 3;
        writereg(dev, PP_CS8900_ISAINT, i);
    } else {
        writereg(dev, PP_CS8920_ISAINT, irq);
    }
}
```

코드 1142. write_irq()함수의 정의

write_irq()함수는 chip_type이 CS8900인 경우와 그렇지 않은 경우를 달리해서 처리한다. 해당 IRQ pin에 대한 mapping table을 검사해서 cs8900_irq_map[]에서 어느것이 알맞는지 찾는다. 찾을 수 없다면, 즉, 인덱스가 sizeof(cs8900_irq_map)/sizeof(cs8900_irq_map[0])과 같다면 끝까지 검색했다는 것을 나타내므로, 3을 PP_CS8900_ISAINT에 쓴다. chip_type이 CS8900이 아니라면, 넘겨받은 irq값을 PP_CS8920_ISAINT에 넣는다.

```
#if ALLOW_DMA
```

```

if (lp->use_dma) {
    if (lp->isa_config & ANY_ISA_DMA) {
        unsigned long flags;
        lp->dma_buff = (unsigned char *)__get_dma_pages(GFP_KERNEL,
                                                        (lp->dmasize * 1024) / PAGE_SIZE);
        if (!lp->dma_buff) {
            printk(KERN_ERR "%s: cannot get %dK memory for DMA\n", dev->name, lp-
>dmasize);
            goto release_irq;
        }
        if (net_debug > 1) {
            printk("%s: dma %lx %lx\n",
                   dev->name,
                   (unsigned long)lp->dma_buff,
                   (unsigned long)virt_to_bus(lp->dma_buff));
        }
        if ((unsigned long)virt_to_bus(lp->dma_buff) >= MAX_DMA_ADDRESS ||
            !dma_page_eq(lp->dma_buff, lp->dma_buff+lp->dmasize*1024-1)) {
            printk(KERN_ERR "%s: not usable as DMA buffer\n", dev->name);
            goto release_irq;
        }
        memset(lp->dma_buff, 0, lp->dmasize * 1024); /* Why? */
        if (request_dma(dev->dma, dev->name)) {
            printk(KERN_ERR "%s: cannot get dma channel %d\n", dev->name, dev-
>dma);
            goto release_irq;
        }
        write_dma(dev, lp->chip_type, dev->dma);
        lp->rx_dma_ptr = lp->dma_buff;
        lp->end_dma_buff = lp->dma_buff + lp->dmasize*1024;
        spin_lock_irqsave(&lp->lock, flags);
        disable_dma(dev->dma);
        clear_dma_ff(dev->dma);
        set_dma_mode(dev->dma, 0x14); /* auto_init as well */
        set_dma_addr(dev->dma, virt_to_bus(lp->dma_buff));
        set_dma_count(dev->dma, lp->dmasize*1024);
        enable_dma(dev->dma);
        spin_unlock_irqrestore(&lp->lock, flags);
    }
}
#endif /* ALLOW_DMA */

```

코드 1143. net_open()함수의 정의(계속)

앞에서 사용할 interrupt번호를 할당받았다. 이젠 인터럽트를 어떻게 발생 시켜줄 것인가를 제공해야 할 것이다. 먼저 DMA를 사용하는 경우를 고려하도록 하자. DMA를 사용한다면, DMA가 끝나면 interrupt가 발생할 것이다.

net_device구조체의 priv에 들어있는 isa_config에 ANY_ISA_DMA가 설정되어 있는지 본다. 있다면, 아래 부분을 수행한다. 먼저 DMA를 위한 buffer를 할당받기 위해서 __get_dma_pages()함수를 호출한다. dmasize에는 Kilo bytes단위의 크기를 가지므로 이 값에 1024를 곱해서 몇 페이지를 차지하는지 계산해서 넘겨준다. 할당받을 메모리 영역은 GFP_KERNEL로 swapping의 영향을 받지 않아야 한다. 할당받지 못한다면 할당된 IRQ를 release해주기 위해서 release_irq로 제어를 옮긴다. 만약 할당받은 DMA를 위한 buffer의 물리적인 주소가 MAX_DMA_ADDRESS⁴⁶²보다 크거나, 할당받은 DMA buffer의 크기가 원하는 크기와 같지 않다면(dma_page_eq()), 다시 release_irq로 제어를 옮긴다.

⁴⁶² ARM SA1100의 경우에는 DMA가 가능한 주소가 0xFFFFFFFF(=4Gbytes)로 설정되어 있다.

할당받은 DMA buffer는 전부 0으로 설정하고, DMA를 위해서 채널을 할당받기 위해 request_dma()함수를 호출한다. 넘겨주는 값은 DMA channel 번호와 디바이스의 이름이다⁴⁶³. DMA channel을 할당받을 수 없다면, release_irq로 제어를 옮긴다.

이전 DMA제어 레지스터에 할당받은 DMA channel번호를 기록한다(write_dma()). Rx를 위한 DMA buffer는 dma_buff로 설정하고, 버퍼의 마지막은 할당받은 buffer 더하기 buffer의 길이로 표시한다.

이전 DMA와 관련된 레지스터들을 초기화하기 위해서 interrupt가 발생하지 못하도록 lock을 설정하고(spin_lock_irqsave()), disable_dma()로 DMA를 먼저 disable 시킨다. 그리고나서, clear_dma_ff() 함수로 DMA channel과 관련된 Flip-Flop을 지우고, set_dma_mode()함수로 모드를 설정한다. DMA에 사용된 buffer를 위한 물리적인 주소는 set_dma_addr()함수로 설정을 하며, 얼마만큼을 DMA buffer로 사용할지를 set_dma_count() 함수로 나타낸다. 이것을 마친후 enable_dma()로 DMA를 사용가능한 상태로 만들어주고, 설정했던 lock을 해제한다(spin_unlock_irqrestore()).

```
/* set the Ethernet address */
for (i=0; i < ETH_ALEN/2; i++)
    writereg(dev, PP_IA+i*2, dev->dev_addr[i*2] | (dev->dev_addr[i*2+1] << 8));
/* while we're testing the interface, leave interrupts disabled */
writereg(dev, PP_BusCTL, MEMORY_ON);
/* Set the LineCTL quintuplet based on adapter configuration read from EEPROM */
if ((lp->adapter_cnf & A_CNF_EXTND_10B_2) && (lp->adapter_cnf & A_CNF_LOW_RX_SQUELCH))
    lp->linectl = LOW_RX_SQUELCH;
else
    lp->linectl = 0;
/* check to make sure that they have the "right" hardware available */
switch(lp->adapter_cnf & A_CNF_MEDIA_TYPE) {
case A_CNF_MEDIA_10B_T: result = lp->adapter_cnf & A_CNF_10B_T; break;
case A_CNF_MEDIA_AUI:   result = lp->adapter_cnf & A_CNF_AUI; break;
case A_CNF_MEDIA_10B_2: result = lp->adapter_cnf & A_CNF_10B_2; break;
default: result = lp->adapter_cnf & (A_CNF_10B_T | A_CNF_AUI | A_CNF_10B_2);
}
if (!result) {
    printk(KERN_ERR "%s: EEPROM is configured for unavailable media\n", dev->name);
release_irq:
#endif ALLOW_DMA
    release_dma_buff(lp);
#endif
    writereg(dev, PP_LineCTL, readreg(dev, PP_LineCTL) & ~(SERIAL_TX_ON
                                                | SERIAL_RX_ON));
    free_irq(dev->irq, dev);
    ret = -EAGAIN;
    goto bad_out;
}
```

코드 1144. net_open()함수의 정의(계속)

이전 IRQ와 DMA의 설정을 마친 상태이다. 하드웨어 주소를 설정하도록 하자. 이것은 PP_IA에서 시작해서 6 bytes를 ethernet address로 만들어주면 될 것이다(writereg()). 이것을 마치면, 인터페이스에 대한 테스트를 하는 동안 bus와 관련된 interrupt를 disable해주기 위해서 PP_BusCTL에 MEMORY_ON을 시켜준다.

EEPROM에서 읽어온 network card를 위한 설정을 적용하도록 하자. EEPROM에서 읽어온 데이터는 net_device구조의 priv부분에 저장된 adapter_cnf에 들어가 있다. 이 값에 A_CNF_EXTND_10B_2과 A_CNF_LOW_RX_SQUELCH가 설정되었다면, linectl에 LOW_RX_SQUELCH를 설정한다. 그렇지 않다면, 0을 linectl에 준다. 어떤 전송 미디어를 사용하는지를 결정하기 위해서 adapter_cnf에

⁴⁶³ 이것은 나중에 /proc에서 DMA와 관련된 부분에서 디바이스 이름에 해당하는 DMA channel 번호의 할당을 찾을 수 있을 것이다.

A_CNF_MEDIA_TYPE를 AND시켜주고, 여기서 나오는 값으로 A_CNF_10B_T, A_CNF_AUI, A_CNF_10B_2를 adapter_cnf필드와 AND시키고, 이것을 다시 result값으로 둔다. 기본(default)으로 result값은 adapter_cnf와 A_CNF_10B_T, A_CNF_AUI, A_CNF_10B_2에 해당하는 bit으로 설정한다. 만약 해당하는 값이 없다면(result==0), 인터페이스 타입을 찾지 못한 것이되므로, 할당받은 DMA를 위한 buffer를 해제하고(release_dma_buff()), PP_LineCTL레지스의 SERIAL_TX_ON과 SERIAL_RX_ON bit을 지운다. 또한 할당받은 interrupt를 해제하기 위해서 free_irq()함수를 호출하고, 복귀 값으로 -EAGAIN을 설정한 후, bad_out으로 제어를 옮긴다.

```
/* set the hardware to the configured choice */
switch(lp->adapter_cnf & A_CNF_MEDIA_TYPE) {
    case A_CNF_MEDIA_10B_T:
        result = detect_tp(dev);
        if (result==DETECTED_NONE) {
            printk(KERN_WARNING "%s: 10Base-T (RJ-45) has no cable\n", dev->name);
            if (lp->auto_neg_cnf & IMM_BIT) /* check "ignore missing media" bit */
                result = DETECTED_RJ45H; /* Yes! I don't care if I see a link pulse */
        }
        break;
    case A_CNF_MEDIA_AUI:
        result = detect_aui(dev);
        if (result==DETECTED_NONE) {
            printk(KERN_WARNING "%s: 10Base-5 (AUI) has no cable\n", dev->name);
            if (lp->auto_neg_cnf & IMM_BIT) /* check "ignore missing media" bit */
                result = DETECTED_AUI; /* Yes! I don't care if I see a carrier */
        }
        break;
    case A_CNF_MEDIA_10B_2:
        result = detect_bnc(dev);
        if (result==DETECTED_NONE) {
            printk(KERN_WARNING "%s: 10Base-2 (BNC) has no cable\n", dev->name);
            if (lp->auto_neg_cnf & IMM_BIT) /* check "ignore missing media" bit */
                result = DETECTED_BNC; /* Yes! I don't care if I can xmit a packet */
        }
        break;
    case A_CNF_MEDIA_AUTO:
        writereg(dev, PP_LineCTL, lp->linectl | AUTO_AUI_10BASET);
        if (lp->adapter_cnf & A_CNF_10B_T)
            if ((result = detect_tp(dev)) != DETECTED_NONE)
                break;
        if (lp->adapter_cnf & A_CNF_AUI)
            if ((result = detect_aui(dev)) != DETECTED_NONE)
                break;
        if (lp->adapter_cnf & A_CNF_10B_2)
            if ((result = detect_bnc(dev)) != DETECTED_NONE)
                break;
        printk(KERN_ERR "%s: no media detected\n", dev->name);
        goto release_irq;
}
```

코드 1145. net_open()함수의 정의(계속)

이전 적절한 media를 선택하는 일이다. 이것은 앞에서 net_device구조체의 priv에 들어있는 adapter_cnf에 설정된 media의 type으로 해당하는 것이 있는지를 알아보는 함수를 호출하는 구조로 되어 있다. 각각에 해당하는 함수는 A_CNF_MEDIA_10B_T인 경우에는 detect_tp(), A_CNF_MEDIA_AUI인 경우에는 detect_aui(), A_CNF_MEDIA_10B_2인 경우에는 detect_bnc(), 그리고 auto sensing을 하고자 한다면, PP_LineCTL에 net_device구조체의 priv에 들어있는 linectl값에 AUTO_AUI_10BASET를 넣어서 어떤 값이

나오는지, 위에서 본 `detect_xxx()`함수를 사용해서 확인해 본다. 해당하는 것이 없다면 `release_irq`로 제어가 옮겨질 것이다.

```
#define DETECTED_NONE 0
#define DETECTED_RJ45H 1
#define DETECTED_RJ45F 2
#define DETECTED_AUI 3
#define DETECTED_BNC 4

static int detect_tp(struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;
    int timenow = jiffies;
    int fdx;

    if (net_debug > 1) printk("%s: Attempting TP\n", dev->name);
    writereg(dev, PP_LineCTL, lp->linectl &~ AUI_ONLY);
    control_dc_dc(dev, 0);
    /* Delay for the hardware to work out if the TP cable is present - 150ms */
    for (timenow = jiffies; jiffies - timenow < 15; )
        ;
    if ((readreg(dev, PP_LineST) & LINK_OK) == 0)
        return DETECTED_NONE;
    if (lp->chip_type == CS8900) {
        switch (lp->force & 0xf0) {
#ifndef 0
            case FORCE_AUTO:
                printk("%s: cs8900 doesn't autonegotiate\n", dev->name);
                return DETECTED_NONE;
#endif
/* CS8900 doesn't support AUTO, change to HALF*/
            case FORCE_AUTO:
                lp->force &= ~FORCE_AUTO;
                lp->force |= FORCE_HALF;
                break;
            case FORCE_HALF:
                break;
            case FORCE_FULL:
                writereg(dev, PP_TestCTL, readreg(dev, PP_TestCTL) | FDX_8900);
                break;
        }
        fdx = readreg(dev, PP_TestCTL) & FDX_8900;
    }
}
```

코드 1146. `detect_tp()` 함수의 정의

`detect_tp()` 함수는 twisted pair를 검출하는 함수이다. 10BASE-T에 대한 검출이다. 먼저 line control 레지스터에서 AUI_ONLY bit을 지운다. `control_dc_dc()` 함수를 호출해서 HCB1 bit에 대한 설정을 해준후 어느정도의 delay를 주어서, TP cable이 동작할 정도의 시간적인 여유(대략 150ms정도)를 준다. 다시 line의 상태(PP_LineST)를 읽어서 올바른지(LINK_OK) 확인한다. 오류가 있다면, DETECTED_NONE을 돌려준다.

만약 chip type이 CS8900인 경우에는 `lp->force`에 0xF0를 mask로해서 나온 값에 따라서, FORCE_AUTO가 있다면, CS8900은 AUTO negotiation을 지원하지 않기에 Half duplex값으로 바꾼다. FULL duplex로 설정된 경우에는(FORCE_FULL) PP_TestCTL의 레지스터에 FDX_8900(Full Duplex)를 설정한다. 이것을 마치고 나면, PP_TestCTL을 읽어서 Full Duplex로 제대로 write가 되었는지 확인해본다(fdx).

```
} else {
    switch (lp->force & 0xf0) {
```

```

        case FORCE_AUTO:
            lp->auto_neg_cnf = AUTO_NEG_ENABLE;
            break;
        case FORCE_HALF:
            lp->auto_neg_cnf = 0;
            break;
        case FORCE_FULL:
            lp->auto_neg_cnf = RE_NEG_NOW | ALLOW_FDX;
            break;
    }
    writereg(dev, PP_AutoNegCTL, lp->auto_neg_cnf & AUTO_NEG_MASK);
    if ((lp->auto_neg_cnf & AUTO_NEG_BITS) == AUTO_NEG_ENABLE) {
        printk(KERN_INFO "%s: negotiating duplex...\n", dev->name);
        while (readreg(dev, PP_AutoNegST) & AUTO_NEG_BUSY) {
            if (jiffies - timenow > 4000) {
                printk(KERN_ERR "**** Full / half duplex auto-negotiation timed
out ****\n");
                break;
            }
        }
        fdx = readreg(dev, PP_AutoNegST) & FDX_ACTIVE;
    }
    if (fdx)
        return DETECTED_RJ45F;
    else
        return DETECTED_RJ45H;
}

```

코드 1147. detect_tp()함수의 정의(계속)

이 부분은 chip type이 CS8900이 아닌 경우에 해당한다. lp->force값을 0xF0로 masking한 값에 따라, FORCE_AUTO가 있다면, AUTO_NEG_ENABLE을 lp->auto_neg_cnf에 넣고, FORCE_HALF인 경우에는 0을, FORCE_FULL인 경우에는 RE_NEG_NOW와 ALLOW_FDX를 넣는다. 이 값을 AUTO_NEG_MASK와 AND 시켜서 auto negotiation을 관리하는 PP_AutoNegCTL 레지스터에 쓴다. Auto-negotiation 상태를 확인하기 위해서, PP_AutoNegST 레지스터가 AUTO_NEG_BUSY인지를 일정 시간동안(40초) 확인한다. Busy상태가 아니라면 제대로 연산이 수행된 것이다. 이전 Auto-negotiation의 결과를 읽을 차례이다. PP_AutoNegST 레지스터를 읽어서 Full Duplex이 active인지 확인한다(fdx).

마지막으로 앞에서 읽은 Full Duplex의 내용에 따라서, DETECTED_RJ45F와 DETECTED_RJ45H값을 돌려준다. 각각은 Full Duplex와 Half Duplex를 찾았다는 것을 의미한다.

```

static int detect_aui(struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;

    if (net_debug > 1) printk("%s: Attempting AUI\n", dev->name);
    control_dc_dc(dev, 0);
    writereg(dev, PP_LineCTL, (lp->linectl &~ AUTO_AUI_10Baset) | AUI_ONLY);
    if (send_test_pkt(dev))
        return DETECTED_AUI;
    else
        return DETECTED_NONE;
}

```

코드 1148. detect_aui()함수의 정의

`detect_aui()`함수는 AUI의 사용을 검사하며, `control_dc_dc()`함수를 호출해서 HCB1 bit에 대한 설정을 해주고, PP_LineCTL 레지스터에 현재의 line control register에 대한 설정에서 AUTO_AUI_10BASET를 빼고, 다시 AUI_ONLY bit을 설정해서 쓴다. Line control 레지스터에 대한 쓰기가 제대로 되었다면, 패킷을 직접 보내서 제대로 되는지 확인한다. 올바르다면 `DETECTED_AUI`를 돌려주고, 그렇지 않다면 `DETECTED_NONE`을 돌려준다.

```
static int detect_bnc(struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;

    if (net_debug > 1) printk("%s: Attempting BNC\n", dev->name);
    control_dc_dc(dev, 1);
    writereg(dev, PP_LineCTL, (lp->linectl &~ AUTO_AUI_10BASET) | AUI_ONLY);
    if (send_test_pkt(dev))
        return DETECTED_BNC;
    else
        return DETECTED_NONE;
}
```

코드 1149. `detect_bnc()`함수의 정의

`detect_bnc()`함수는 BNC type의 인터페이스를 사용하는지를 검사한다. 먼저 `control_dc_dc()`를 호출해서 HCB1 bit에 대한 설정을 해주고, PP_LineCTL에 현재 설정된 line control에서 AUTO_AUI_10BASET를 clear하고, AUI_ONLY를 설정한다. 마지막으로 packet send 테스트를 해서, 제대로 전송이 된다면 `DETECT_BNC`, 그렇지 않다면 `DETECTED_NONE`을 돌려준다.

```
static void control_dc_dc(struct net_device *dev, int on_not_off)
{
    struct net_local *lp = (struct net_local *)dev->priv;
    unsigned int selfcontrol;
    int timenow = jiffies;

    selfcontrol = HCB1_ENBL; /* Enable the HCB1 bit as an output */
    if (((lp->adapter_cnf & A_CNF_DC_DC_POLARITY) != 0) ^ on_not_off)
        selfcontrol |= HCB1;
    else
        selfcontrol &= ~HCB1;
    writereg(dev, PP_SelfCTL, selfcontrol);
    /* Wait for the DC/DC converter to power up - 500ms */
    while (jiffies - timenow < HZ)
        ;
}
```

코드 1150. `control_dc_dc()`함수의 정의

`control_dc_dc()`함수는 PP_SelfCTL 레지스터에 HCB1 pin에 관련된 설정을 하는데 사용한다. 먼저 현재의 설정을 알기 위해서 `net_device` 구조체의 `priv`부분에 있는 `adapter_cnf` 필드와 `A_CNF_DC_DC_POLARITY`를 AND 시켜서, 해당 bit이 설정되어 있는지 확인한다. 이 값을 `on_not_off`와 exclusive OR 시켜서 `selfcontrol` 변수에 HCB1을 설정할 것인지 말것인지를 결정한다. 결정된 값으로 PP_SelfCTL 레지스터에 써준다. 또한 결과가 `adapter`에 반영될 때까지 잠시동안 대기한다.

간단히 요약하면, `A_CNF_DC_DC_POLARITY`가 이미 설정되어 있다면, `on_not_off`에 따라서 1이라면 HCB1 bit은 clear 될 것이며, 0이라면 HCB1 bit이 set될 것이다. 만약 `A_CNF_DC_DC_POLARITY`가 설정되지 않았다면, `on_not_off`에 따라서 1이라면 HCB1 bit은 set 될 것이며, 0이라면 HCB1 bit은 clear될 것이다.

```
/* send a test packet - return true if carrier bits are ok */
```

```

static int send_test_pkt(struct net_device *dev)
{
    char test_packet[] = { 0,0,0,0,0,0,0,0,0,0,0,
                           0, 46, /* A 46 in network order */
                           0, 0, /* DSAP=0 & SSAP=0 fields */
                           0xf3, 0 /* Control (Test Req + P bit set) };

    long timenow = jiffies;
    writereg(dev, PP_LineCTL, readreg(dev, PP_LineCTL) | SERIAL_TX_ON);
    memcpy(test_packet, dev->dev_addr, ETH_ALEN);
    memcpy(test_packet+ETH_ALEN, dev->dev_addr, ETH_ALEN);
    writeword(dev, TX_CMD_PORT, TX_AFTER_ALL);
    writeword(dev, TX_LEN_PORT, ETH_ZLEN);
    /* Test to see if the chip has allocated memory for the packet */
    while (jiffies - timenow < 5)
        if (readreg(dev, PP_BusST) & READY_FOR_TX_NOW)
            break;
    if (jiffies - timenow >= 5)
        return 0; /* this shouldn't happen */
    /* Write the contents of the packet */
    outsw(dev->base_addr + TX_FRAME_PORT, test_packet, (ETH_ZLEN+1) >>1);
    if (net_debug > 1) printk("Sending test packet ");
    /* wait a couple of jiffies for packet to be received */
    for (timenow = jiffies; jiffies - timenow < 3; )
        ;
    if ((readreg(dev, PP_TxEvent) & TX_SEND_OK_BITS) == TX_OK) {
        if (net_debug > 1) printk("succeeded\n");
        return 1;
    }
    if (net_debug > 1) printk("failed\n");
    return 0;
}

```

코드 1151. send_test_pkt()함수의 정의

send_test_pkt()함수는 해당하는 interface의 type이 제대로 작동하는지를 확인하기 위해서 test packet을 보내는 일을 한다. test_packet[]이 해당하는 테스트 패킷이다.

먼저 현재 시간을 기록하고, PP_LineCTL 레지스터를 읽어서 SERIAL_TX_ON을 설정해서 다시 써주고(이것은 Tx를 ON시킨다.), test_packet에 받는 쪽의 hardware주소(destination address)를 적는다. 다시 이곳에 받는 쪽의 hardware 주소를 적는다. 물론 이 같은 경우, 둘다 현재 ethernet card의 hardware주소로 설정했다. 즉, 패킷을 받은 상대방(ex. Hub)이, 다시 ethernet card 자신에게 패킷을 보내도록 만드는 것이다. TX_CMD_PORT(Tx Command Port)에 TX_AFTER_ALL을 써서, 모든 패킷의 내용을 다 copy한 후에 전송하도록 만들고, TX_LEN_PORT(Tx Packet Length Port)에는 ETH_ZLEN을 주어서 가장 작은 ethernet 패킷의 크기로 만들어준다. 이전 PP_BusST 레지스터를 읽어서 READY_FOR_TX_NOW(Tx를 보낼 준비가 되었는지를 확인한다.)가 나올 때까지 기다린다. 시간값을 검사해서, 초기에 저장했던 시간과 jiffies가 50ms 이상 된다면, 계속 진행하지 않고 0을 돌려준다.

이전 패킷을 보내도록 한다. 이것은 디바이스의 기본주소(base address)에 TX_FRAME_PORT(Tx Frame Port: Tx 할 Frame을 가르키는 레지스터)값으로 test_packet[]을, 길이로는 ETH_ZLEN + 1을 2로 나누어서 준다. 보낸 것을 제대로 보내는지 확인하기 위해서 30ms동안 for loop를 돌면서 확인한다. PP_TxEvent에서 TX_SEND_OK_BITS를 AND시켜서, 이 값이 TX_OK인지를 확인하면 될 것이다. 나온다면, Tx가 제대로 된 것이고, 그렇지 않다면, 그 interface를 통해서 패킷을 전송하지 못한다는 말이 되므로 error이다.

다시 이전의 하던 이야기로 되돌아 가기로 하자. net_open()함수를 계속 설명하도록 하겠다. 앞에서 설명한 routine들을 사용해서 나온 결과 값으로는 DETECTED_NONE, DETECTED_RJ45H, DETECTED_RJ45F, DETECTED_AUI, DETECTED_BNC가 있다. 각각은 아무것도 찾지 못한 경우와 RJ45를 사용하는 Full Duplex, Half Duplex 및 AUI와 BNC에 대해서 찾았다는 것을 나타낸다.

```

switch(result) {
    case DETECTED_NONE:
        printk(KERN_ERR "%s: no network cable attached to configured media\n", dev->name);
        goto release_irq;
    case DETECTED_RJ45H:
        printk(KERN_INFO "%s: using half-duplex 10Base-T (RJ-45)\n", dev->name);
        break;
    case DETECTED_RJ45F:
        printk(KERN_INFO "%s: using full-duplex 10Base-T (RJ-45)\n", dev->name);
        break;
    case DETECTED_AUI:
        printk(KERN_INFO "%s: using 10Base-5 (AUI)\n", dev->name);
        break;
    case DETECTED_BNC:
        printk(KERN_INFO "%s: using 10Base-2 (BNC)\n", dev->name);
        break;
}

```

코드 1152. net_open()함수의 정의(계속)

위에서 하는 일은 앞에서 확인한 정보를 단순히 보여주는 일이다. 만약 아무것도 검출하지 못했다면, 제어는 release_irq로 옮겨질 것이다. 그렇지 않다면, 해당하는 media type에 대한 정보를 보여준다.

```

/* Turn on both receive and transmit operations */
writereg(dev, PP_LineCTL, readreg(dev, PP_LineCTL) | SERIAL_RX_ON | SERIAL_TX_ON);
/* Receive only error free packets addressed to this card */
lp->rx_mode = 0;
writereg(dev, PP_RxCTL, DEF_RX_ACCEPT);
lp->curr_rx_cfg = RX_OK_ENBL | RX_CRC_ERROR_ENBL;
if (lp->isa_config & STREAM_TRANSFER)
    lp->curr_rx_cfg |= RX_STREAM_ENBL;
#endif
set_dma_cfg(dev);
#endif
writereg(dev, PP_RxCFG, lp->curr_rx_cfg);
writereg(dev, PP_TxCFG, TX_LOST_CRS_ENBL | TX_SQE_ERROR_ENBL | TX_OK_ENBL |
    TX_LATE_COL_ENBL | TX_JBR_ENBL | TX_ANY_COL_ENBL | TX_16_COL_ENBL);
writereg(dev, PP_BufCFG, READY_FOR_TX_ENBL | RX_MISS_COUNT_OVRFLW_ENBL |
#endif
dma_bufcfg(dev) |
#endif
    TX_COL_COUNT_OVRFLW_ENBL | TX_UNDERRUN_ENBL);
/* now that we've got our act together, enable everything */
writereg(dev, PP_BusCTL, ENABLE_IRQ
    | (dev->mem_start?MEMORY_ON : 0) /* turn memory on */
#endif
    | dma_busctl(dev)
#endif
);
netif_start_queue(dev);
if (net_debug > 1)
    printk("cs89x0: net_open() succeeded\n");
return 0;
bad_out:
    return ret;
}

```

코드 1153. net_open()함수의 정의(계속)

이전 모든 설정이 끝났으므로, Tx와 Rx를 enable시켜준다. 이것은 PP_LineCTL을 읽어서, SERIAL_RX_ON과 SERIAL_TX_ON을 설정해서 다시 PP_LineCTL에 적는 것으로 가능하다. 현재의 rx_mode를 0으로 초기화 시키고, PP_RxCTL에는 기본적인 Rx만 가능하도록 DEF_RX_ACCEPT를 적는다. 그리고, 현재의 Rx에 대한 설정값으로는 RX_OK_ENBL과 RX_CRC_ERROR_ENBL을 설정하고, isa_config에 STREAM_TRANSFER이 설정된 경우에는 RX_STREAM_ENBL을 추가해주도록 한다. 만약 DMA를 사용한다면 set_dma_cfg()를 호출해서 DMA에 관련된 설정을 해준다.

앞에서 설정한 Rx에 관련된 설정을 이전 직접 레지스터에 적용할 차례이다. 먼저 PP_RxCFG에는 curr_rx_cfg를 적고, PP_TxCFG에는 어떤 조건에 대해서 Tx쪽 인터럽트가 발생할지를 기입한다. 해당하는 것들은 TX_LOST_CRS_ENBL, TX_SQE_ERROR_ENBL, TX_OK_ENBL, TX_LATE_COL_ENBL, TX_JBR_ENBL, TX_ANY_COL_ENBL, TX_16_COL_ENBL)이 있다. 이것은 Tx가 올바른 경우와 error가 있는 경우에 대해서 모두 interrupt를 발생시켜줄 것이다. Interrupt 처리부분에서 관련된 에러나 혹은 OK 상황을 알아서 처리해 주어야 할 것이다. 나머지는 Rx와 Tx시에 사용할 buffer에 관련된 인터럽트를 발생시킬 것인가를 PP_BufCFG 레지스터에 적는 것이다. 관련된 것으로는 READY_FOR_TX_ENBL, RX_MISS_COUNT_OVRFLOW_ENBL, TX_COL_COUNT_OVRFLOW_ENBL, TX_UNDERRUN_ENBL 및, DMA를 사용할 때 해당하는 dma_bufcfg()함수의 return값이 있다. dma_bufcfg()함수는 나중에 DMA를 볼 때 보기로 하겠다.

자, 이제는 해당하는 모든 레지스터의 설정을 마쳤다. 이전 실제로 인터럽트를 발생하도록 만들어 주어야 할 것이다. 이것은 PP_BusCTL에 ENABLE_IRQ와 메인 메모리를 사용할 것인가를 결정하는 net_device구조체의 mem_start의 설정 여부에 따른 MEMORY_ON이나 혹은 0을 OR시켜서 적는다. 또한 추가적으로 DMA를 사용할 경우에는 dma_busctl()의 복귀값도 OR시켜준다.

함수를 복귀하기 전에, 마지막으로 해주어야 할 일은 커널이 이 네트워크 인터페이스를 사용할 수 있다고 알려주기 위해서 netif_start_queue()함수를 호출한다. 이제 부터는 이 네트워크 인터페이스에 대해서 패킷이 queuing될 것이며, 전송 요구가 생겨나게 될 것이다. 복귀값은 0이다. bad_out부분은 단순히 복귀 코드로 에러 값을 전달하기만 한다.

17.8.6. Close

네트워크 인터페이스에 대한 사용을 마치기 위해서 커널은 close()함수를 호출할 것이다. CS8900에서는 net_close()함수를 정의하고 있다.

```
/* The inverse routine to net_open(). */
static int net_close(struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;

    netif_stop_queue(dev);
    writereg(dev, PP_RxCFG, 0);
    writereg(dev, PP_TxCFG, 0);
    writereg(dev, PP_BufCFG, 0);
    writereg(dev, PP_BusCTL, 0);
    free_irq(dev->irq, dev);

#if ALLOW_DMA
    if (lp->use_dma && lp->dma) {
        free_dma(dev->dma);
        release_dma_buff(lp);
    }
#endif
    /* Update the statistics here. */
    return 0;
}
```

코드 1154. net_close()함수의 정의

먼저, 더이상 이 network 인터페이스에 대한 패킷의 큐를 사용하지 않는다는 것을 알려주기 위해서 netif_stop_queue()를 호출한다. 그리고나서, PP_RxCFG와 PP_TxCFG, PP_BufCFG, PP_BusCTL에 각각 0을

쓴다. 할당받은 interrupt를 해제하기 위해서 free_irq()함수를 호출하고, 만약 DMA의 사용을 허가했다면, 할당된 DMA channel도 해제하고(free_dma()), DMA를 위한 buffer도 해제한다(release_dma_buf()). 복귀 코드 0이다. 따라서, 더 이상의 interrupt는 발생하지 않을 것이다.

```
#if ALLOW_DMA
static void release_dma_buff(struct net_local *lp)
{
    if (lp->dma_buff) {
        free_pages((unsigned long)(lp->dma_buff), (lp->dmasize * 1024) / PAGE_SIZE);
        lp->dma_buff = 0;
    }
}
#endif
```

코드 1155. release_dma_buff()함수의 정의

release_dma_buff()함수는 DMA를 허가할 때만 사용되는 것으로 DMA를 위해서 할당된 net_device구조체의 priv 필드에 있는 dma_buff를 페이지 단위로 해제할 것이다(free_pages()). 해제를 마자면, dma_buff에는 0을 둔다.

17.8.7. Send

커널에서 hard_start_xmit()함수를 호출하게 될 때 호출되는 함수이다. 즉, 커널이 패킷을 보내고 싶을 때 호출된다. 중요한 것은 넘겨받는 파라미터로 socket buffer를 받는다는 것이다. 또한 해당하는 net_device구조체를 전달받는다.

```
static int net_send_packet(struct sk_buff *skb, struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;

    if (net_debug > 3) {
        printk("%s: sent %d byte packet of type %x\n",
               dev->name, skb->len,
               (skb->data[ETH_ALEN+ETH_ALEN] << 8) | skb->data[ETH_ALEN+ETH_ALEN+1]);
    }
    spin_lock_irq(&lp->lock);
    netif_stop_queue(dev);
    /* initiate a transmit sequence */
    writeword(dev, TX_CMD_PORT, lp->send_cmd);
    writeword(dev, TX_LEN_PORT, skb->len);
    /* Test to see if the chip has allocated memory for the packet */
    if ((readreg(dev, PP_BusST) & READY_FOR_TX_NOW) == 0) {
        /*
         * Gasp! It hasn't. But that shouldn't happen since
         * we're waiting for TxOk, so return 1 and requeue this packet.
         */
        spin_unlock_irq(&lp->lock);
        if (net_debug) printk("cs89x0: Tx buffer not free!\n");
        return 1;
    }
    /* Write the contents of the packet */
    outsw(dev->base_addr + TX_FRAME_PORT, skb->data, (skb->len+1) >>1);
    spin_unlock_irq(&lp->lock);
    dev->trans_start = jiffies;
    dev_kfree_skb (skb);
    return 0;
}
```

코드 1156. net_send_packet()함수의 정의

net_device구조체로부터 priv필드를 찾는다(lp). 모든 패킷이 chip에 들어가기전에 chip에게 전송요구를 해주기에, 인터럽트가 발생하지 않도록 미리 spin_lock_irq()를 호출한다. 일단 보내는 패킷을 이미 가지고 있으므로 netif_stop_queue()를 호출해서 더 이상의 패킷 전송요구를 커널이 하지 않도록 만든다.

이전 실제적으로 Tx를 요청하는 단계이다. 먼저 TX_CMD_PORT에 send_cmd를 쓴다. 또한 보내고자 하는 packet의 길이(skb->len)를 TX_LEN_PORT에 적는다. Chip이 Tx 패킷을 위해서 메모리를 할당했는지를 알기 위해서 PP_BusST(Bus status)를 읽어서, 이 값이 READY_FOR_TX_NOW와 같은지 확인한다. 만약, 같지 않다면 Tx를 위한 Chip내의 메모리가 없다는 말이되므로 Tx를 처리하지 않는다. 단지 앞에서 설정한 spin lock만 해제하고(spin_unlock_irq()), 1을 복귀 코드를 넘겨준다. 나중에 커널이 다시 요구를 처리해 줄 것이다. 즉, netif_wake_queue()가 호출되어, 여유가 생기는 시점이 될 것이다.

이전 Tx를 할 준비가 chip에서 되었다는 말이되므로, packet의 시작 주소와 길이를 TX_FRAME_PORT에 적는다(outsw()). 이것을 마치고 나면, 전송이 chip에서의 전송요구는 되었으므로 spin lock을 해제하고(spin_unlock_irq()), 전송 시간(trans_start)를 기록한 후, 소켓 버퍼를 해제한다(dev_kfree_skb()). 복귀 코드는 0이다. 여기서 중요한 점은 한번에 하나의 packet에 대한 전송 요구를 처리하도록 만들어졌다는 점이다. 즉, net_send_packet()함수의 마지막 부분에서 netif_wake_queue()를 하지 않는다. 이것은 나중에 interrupt의 처리 부분에서 Tx와 관련된 처리를 하게 될 때, 호출해서 다른 패킷을 처리할 준비가 되었음을 알려준다.

17.8.8. Interrupt

네트워크 디바이스의 인터럽트는 Tx및 Rx에서 발생한다. 또한, 각종 에러상황에 대한 보고로도 사용된다. 따라서, 우리는 인터럽트의 상태를 읽어서, 해당하는 일들을 처리해 주어야 할 것이다. 즉, Tx가 일어났다면, Tx buffer들에 대한 관리와 Rx가 일어났다면, 소켓버퍼로 받은 데이터를 넣어서 상위로 올려주는 일들이 될 것이다. 또한 에러상황의 보를 위해서, net_device구조체의 일부인 priv필드에 들어가는 net_stats구조체 정보도 갱신해 주어야한다. 각각에 대해서 알아보도록 하자.

```
static void net_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct net_device *dev = dev_id;
    struct net_local *lp;
    int ioaddr, status;

    ioaddr = dev->base_addr;
    lp = (struct net_local *)dev->priv;
    while ((status = readword(dev, ISQ_PORT))) {
        if (net_debug > 4) printk("%s: event=%04x\n", dev->name, status);
        switch(status & ISQ_EVENT_MASK) {
            case ISQ_RECEIVER_EVENT:
                /* Got a packet(s). */
                net_rx(dev);
                break;
            case ISQ_TRANSMITTER_EVENT:
                lp->stats.tx_packets++;
                netif_wake_queue(dev);      /* Inform upper layers. */
                if ((status & (TX_OK |
                               TX_LOST_CRS |
                               TX_SQE_ERROR |
                               TX_LATE_COL |
                               TX_16_COL)) != TX_OK) {
                    if ((status & TX_OK) == 0) lp->stats.tx_errors++;
                    if (status & TX_LOST_CRS) lp->stats.tx_carrier_errors++;
                    if (status & TX_SQE_ERROR) lp->stats.tx_heartbeat_errors++;
                    if (status & TX_LATE_COL) lp->stats.tx_window_errors++;
                    if (status & TX_16_COL) lp->stats.tx_aborted_errors++;
                }
        }
    }
}
```

```

        break;
    case ISQ_BUFFER_EVENT:
        if (status & READY_FOR_TX) {
            netif_wake_queue(dev);      /* Inform upper layers. */
        }
        if (status & TX_UNDERRUN) {
            if (net_debug > 0) printk("%s: transmit underrun\n", dev->name);
            lp->send_underrun++;
            if (lp->send_underrun == 3) lp->send_cmd = TX_AFTER_381;
            else if (lp->send_underrun == 6) lp->send_cmd = TX_AFTER_ALL;
            netif_wake_queue(dev);      /* Inform upper layers. */
        }
}

```

코드 1157. net_interrupt()함수의 정의

먼저 ISQ_PORT에서 인터럽트의 상태를 읽어온다. 이 값을 ISQ_EVENT_MASK로 masking시키고, 이 값을 가지고 어떤 event가 생겼는지를 확인하게 된다. 만약 ISQ_RECEIVER_EVENT인 경우에는 net_rx()함수를 호출해서 처리한다. net_rx()함수는 나중에 receive에 대해서 볼 때 분석할 것이다. ISQ_TRANSMITTER_EVENT는 Tx와 관련된 것이다. tx_packets의 수를 증가시키고, Tx를 하고 있는 상위 layer에 패킷을 더 처리할 수 있다는 것을 알려준다(netif_wake_queue()). TX의 상태에 따라서 에러가 있을 경우에는 해당하는 net_stats 구조체의 필드를 업데이트 시켜준다. TX_OK는 제대로 Tx를 했나는 것을 말해주며, TX_LOST_CRS는 Carrier가 없다는 에러이며, TX_SQE_ERROR는 SQE⁴⁶⁴ 에러임을, TX_LATE_COL은 late collision 에러⁴⁶⁵를, TX_16_COL은 16개의 collision이 발생해서 Tx를 멈춘 에러를 말한다.

버퍼와 관련된 인터럽트는 ISQ_BUFFER_EVENT의 상태 값을 가진다. READY_FOR_TX를 상태값이 가진다면, Tx할 준비가 되었다는 것을 나타내므로 netif_wake_queue()를 호출한다. 만약 상태값이 TX_UNDERRUN을 가지고, priv에 있는 send_underrun값이 3이라면, send_cmd에 TX_AFTER_381을 주고, 그렇지 않고, send_underrun이 6을 가진다면, TX_AFTER_ALL을 send_cmd로 준다. 각각은 381 bytes가 copy된 후에 packet을 Tx하라는 것과 모든 packet이 copy가 된 후에 Tx하라는 말이다. 이렇게 하는 이유는 Tx쪽에서 일어날 수 있는 Underrun⁴⁶⁶을 줄여주기 위해서이다. 이전 상위의 layer가 timeout이 될 때까지 기다리지 않고, 다시 packet의 전송 요구를 하도록 해주기 위해서 netif_wake_queue()를 호출한다.

```

#ifndef ALLOW_DMA
    if (lp->use_dma && (status & RX_DMA)) {
        int count = readreg(dev, PP_DmaFrameCnt);
        while(count) {
            if (net_debug > 5)
                printk("%s: receiving %d DMA frames\n", dev->name,
count);
            if (net_debug > 2 && count > 1)
                printk("%s: receiving %d DMA frames\n", dev->name,
count);
            dma_rx(dev);
            if (--count == 0)
                count = readreg(dev, PP_DmaFrameCnt);
            if (net_debug > 2 && count > 0)
                printk("%s: continuing with %d DMA frames\n", dev-
>name, count);
        }
    }
#endif
break;

```

⁴⁶⁴ AUI를 통해서 전송을 하게될 때, 64bit times내에서 collision을 보게되는 경우를 말한다.

⁴⁶⁵ 512 bit times이후에 일어난 에러를 말한다.

⁴⁶⁶ Tx를 하려고 하지만, 충분한 packet이 없어서 발생하는 에러가 Tx underrun이다.

```

        case ISQ_RX_MISS_EVENT:
            lp->stats.rx_missed_errors += (status >>6);
            break;
        case ISQ_TX_COL_EVENT:
            lp->stats.collisions += (status >>6);
            break;
    }
}
}

```

코드 1158. net_interrupt()함수의 정의(계속)

만약 DMA를 사용하도록 설정되었다면(priv필드 즉, net_local구조체의 use_dma와 interrupt상태 값이 RX_DMA인 경우), PP_DmaFrameCnt를 읽어서 얼마나 많은 데이터를 가지는지 알아본다(count). 그리고나서, DMA를 사용하는 받기(receive)를 하기위해서 dma_rx()함수를 호출한다. DMA와 관련된 것은 다음에서 자세히 논의하기로 하고, 여기서는 단순히 DMA를 사용해서 패킷을 처리한다고만 생각하도록 하자.

상태 정보 값이 ISQ_RX_MISS_EVENT나 ISQ_TX_COL_EVENT라면, net_device의 priv에 있는 net_stats의 필드를 갱신한다. 해당하는 에러 정보 필드는 rx_missed_errors와 collisions 필들이다.

17.8.9. DMA

DMA(Direct Memory Access)란 디바이스와 메인 메모리 간에 CPU의 개입이 없이 데이터를 보내고 받기를 하는 방법이다. 따라서, CPU의 개입이 없으므로, CPU는 해당 연산이 일어나는 동안 다른 일을 할 수 있다. 다만, bus master로 동작하지 않는 디바이스의 경우에는 시스템에 희소한 자원인 DMA channel의 할당을 수반하기에 다른 디바이스 드라이버들과 경합 상황(race condition)이 발생할 가능성이 있다.

```

#ifndef ALLOW_DMA
#define dma_page_eq(ptr1, ptr2) ((long)(ptr1)>>17 == (long)(ptr2)>>17)

static void get_dma_channel(struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;

    if (lp->dma) {
        dev->dma = lp->dma;
        lp->isa_config |= ISA_RxDMA;
    } else {
        if (((lp->isa_config & ANY_ISA_DMA) == 0)
            return;
        dev->dma = lp->isa_config & DMA_NO_MASK;
        if (lp->chip_type == CS8900)
            dev->dma += 5;
        if (dev->dma < 5 || dev->dma > 7) {
            lp->isa_config &= ~ANY_ISA_DMA;
            return;
        }
    }
    return;
}

```

코드 1159. get_dma_channel()함수의 정의

지금부터 보게되는 모든 함수는 ALLOW_DMA 설정된 경우로 국한된다. dma_page_eq()는 두개의 주소는 가르키는 변수를 받아서, 우측으로 17bit를 shift해서 같은 값을 확인하는 매크로이다. 즉, 하위 128Kbytes는 무시하고 상위의 가상 주소만을 비교한다.

`get_dma_channel()` 함수는 시스템에 한정된 자원으로 존재하는 DMA를 위한 channel을 할당받기 위해서 내부적인 설정을 해줄때 사용하는 함수이다. 먼저 `net_device` 구조체로부터 `priv` 영역에 대한 포인터를 얻는다(`lp`). `lp`의 `dma` 필드는 현재 설정된 DMA channel을 나타내므로, 만약 이것이 이미 설정되었다면, `net_device` 구조체의 `dma` 필드를 이 값으로 설정하고, `lp`의 ISA 관련 설정을 가지고 있는 `isa_config` 필드에 `ISA_RxDMA`를 OR시킨다. 만약 이미 설정된 DMA channel이 없다면, `lp->isa_config` 필드와 `ANY_ISA_DMA`를 AND시켜서 0 값을 가지면, 해당 DMA channel에 대한 일치하는 값이 없다는 뜻이 되므로 그냥 복귀한다. 그렇지 않다면, `net_device`의 `dma`에는 `lp->isa_config`와 `DMA_NO_MASK`(DMA channel number mask)를 AND 시킨 값을 지정한다. 만약 chip type이 CS8900이라면, 이렇게 결정된 값에 +5를 더하게 되며, 나온 값이 5보다 작거나 7보다 크다면, `lp->isa_config` 필드를 `ANY_ISA_DMA`를 NOT한 값을 AND시켜서 저장한다. 즉, 특정 DMA channel 번호를 사용하겠다는 뜻이다.

```
static void write_dma(struct net_device *dev, int chip_type, int dma)
{
    struct net_local *lp = (struct net_local *)dev->priv;
    if ((lp->isa_config & ANY_ISA_DMA) == 0)
        return;
    if (chip_type == CS8900) {
        writereg(dev, PP_CS8900_ISADMA, dma-5);
    } else {
        writereg(dev, PP_CS8920_ISADMA, dma);
    }
}
```

코드 1160. `write_dma()` 함수의 정의

`write_dma()` 함수는 디바이스의 DMA 설정을 담당한다. 먼저 `net_device` 구조체를 가르키는 `dev`에서 `priv` 부분에 대한 포인터를 가져와서 `lp`를 설정한다. `lp->isa_config` 필드가 `ANY_ISA_DMA` 값이 설정되지 않았다면, 바로 복귀한다. 그렇지 않다면, chip type이 CS8900인 경우에는 `PP_CS8900_ISADMA`에 DMA channel 번호를 가지는 `dma` 값에서 -5를 한 값을 넣고, 다른 chip type인 경우에는 `PP_CS8920_ISADMA`에 `dma` 값을 쓴다. 즉, 해당 DMA channel 설정을 나타내는 레지스터에 DMA channel을 기록한다.

```
static void set_dma_cfg(struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;

    if (lp->use_dma) {
        if ((lp->isa_config & ANY_ISA_DMA) == 0) {
            if (net_debug > 3)
                printk("set_dma_cfg(): no DMA\n");
            return;
        }
        if (lp->isa_config & ISA_RxDMA) {
            lp->curr_rx_cfg |= RX_DMA_ONLY;
            if (net_debug > 3)
                printk("set_dma_cfg(): RX_DMA_ONLY\n");
        } else {
            lp->curr_rx_cfg |= AUTO_RX_DMA; /* not that we support it... */
            if (net_debug > 3)
                printk("set_dma_cfg(): AUTO_RX_DMA\n");
        }
    }
}
```

코드 1161. `set_dma_cfg()` 함수의 정의

이 함수는 DMA에 대한 설정사항을 바꾸는 함수이다. 먼저 net_device구조체로부터 priv부분에 대한 포인터를 구해서 lp를 설정한다. 만약 lp의 use_dma가 설정된 경우라면, 아래의 연산을 수행하고, 그렇지 않은 경우에는 DMA를 사용하지 않으므로 해줄 일이 없다.

lp->isa_config에 ANY_ISA_DMA로 설정되지 않았다면, 앞에서 제대로 DMA에 대한 설정을 해주지 못한 것이 되므로 에러 메시지를 쓰고 바로 복귀한다. lp->isa_config에 ISA_RxDMA가 설정된 경우에는 lp->curr_rx_cfg(현재의 Rx 설정) 사항에 RX_DMA_ONLY bit을 더해준다(OR). ISA_RxDMA가 설정되지 않았다면, lp->curr_rx_cfg에 AUTO_RX_DMA를 더해준다(OR). 즉, ISA DMA설정에 자동적으로 할당받은 Rx DMA channel번호를 쓸 것인가 아니면, 이미 설정된 것을 반영할 것인가를 결정한다.

```
static int dma_bufcfg(struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;
    if (lp->use_dma)
        return (lp->isa_config & ANY_ISA_DMA)? RX_DMA_ENBL : 0;
    else
        return 0;
}
```

코드 1162. dma_bufcfg()함수의 정의

dma_bufcfg()는 DMA의 입출력에 대한 buffer의 설정을 담당한다. 물론 하드웨어적인 설정이 아니라, 현재의 설정사항을 기록하기 위한 것이다. 먼저, net_device구조체의 priv부분에 대한 포인터를 얻어와서 lp를 설정한다. 만약 lp->use_dma가 설정되어 있다면, DMA를 사용하겠다는 말이되며, ANY_ISA_DMA와가 lp->isa_config에 설정되어 있는지를 보고, RX_DMA_ENBL이나 0을 돌려준다. DMA를 사용하지 않는다면, 0을 돌려준다. 이곳에서 return된 값으로 나중에 buffer에대한 DMA 설정관련 레지스터에 쓸 내용이 결정된다.

```
static int dma_busctl(struct net_device *dev)
{
    int retval = 0;
    struct net_local *lp = (struct net_local *)dev->priv;

    if (lp->use_dma) {
        if (lp->isa_config & ANY_ISA_DMA)
            retval |= RESET_RX_DMA; /* Reset the DMA pointer */
        if (lp->isa_config & DMA_BURST)
            retval |= DMA_BURST_MODE; /* Does ISA config specify DMA burst ? */
        if (lp->dmasize == 64)
            retval |= RX_DMA_SIZE_64K; /* did they ask for 64K? */
        retval |= MEMORY_ON; /* we need memory enabled to use DMA. */
    }
    return retval;
}
```

코드 1163. dma_busctl()함수의 정의

dma_busctl()함수는 DMA bus에 대한 설정 정보를 돌려준다. 먼저, net_device구조체의 priv부분에 대한 포인터를 얻어와서 lp를 설정한다. 만약 DMA를 사용한다면(lp->use_dma) 다음과 같은 일을 한다. lp->isa_config에 ANY_ISA_DMA가 설정되 있다면, 복귀값에 RESET_RX_DMA를 추가한다. 만약 lp->isa_config에 DMA_BURST가 설정되어 있다면, DMA_BURST_MODE를 복귀값에 추가한다. 만약 lp->dmasize가 64라면(64Kbytes), 복귀값에 RX_DMA_SIZE_64K를 추가한다. 마지막으로 DMA를 사용하기 위해서 메모리를 ON시키기 위해서 복귀값에 MEMORY_ON을 설정하고 복귀값을 돌려준다. 나중에 이와 같이 설정된 복귀값으로 DMA bus에 대한 설정을 맡는 레지스터에 쓰기를 할 것이다.

```
static void dma_rx(struct net_device *dev)
{
```

```

struct net_local *lp = (struct net_local *)dev->priv;
struct sk_buff *skb;
int status, length;
unsigned char *bp = lp->rx_dma_ptr;

status = bp[0] + (bp[1]<<8);
length = bp[2] + (bp[3]<<8);
bp += 4;
if (net_debug > 5) {
    printk("%s: receiving DMA packet at %lx, status %x, length %x\n",
           dev->name, (unsigned long)bp, status, length);
}
if ((status & RX_OK) == 0) {
    count_rx_errors(status, lp);
    goto skip_this_frame;
}

```

코드 1164. dma_rx()함수의 정의

dma_rx()함수는 DMA를 사용한 Rx를 처리하는 함수이다. 넘겨받는 값으로는 net_device구조체이다. 사용하는 변수들로는 lp가 net_device구조체의 priv부분을 가리키며, skb는 소켓버퍼 구조체를 할당받을 포인터로, status와 length는 DMA상태와 DMA를 한 데이터의 크기를, bp는 DMA buffer에 대한 포인터를 가진다.

먼저 DMA의 상태값을 읽는다. bp의 첫번째 바이트와 두번째 바이트를 결합해서 하나의 상태 값으로 만든다. 길이는 세번째와 네번째 값을 통해서 얻는다. 하위의 바이트가 상위의 값을 나타내기에 각각 두번째 바이트를 8bit 왼쪽으로 shift했다. 현재의 buffer pointer(bp)는 4를 증가시켜서 실제 DMA를 사용해서 받은 데이터의 첫부분을 가지도록 만든다. 상태값을 비교해서 RX_OK가 아니라면, Rx에서 발생한 error의 수를 세는 count_rx_errors()함수를 호출하고, skip_this_frame으로 제어를 옮긴다..

```

/* Malloc up new buffer.*/
skb = dev_alloc_skb(length + 2);
if (skb == NULL) {
    if (net_debug) /* I don't think we want to do this to a stressed system */
        printk("%s: Memory squeeze, dropping packet.\n", dev->name);
    lp->stats.rx_dropped++;
    /* AKPM: advance bp to the next frame */
skip_this_frame:
    bp += (length + 3) & ~3;
    if (bp >= lp->end_dma_buff) bp -= lp->dmasize*1024;
    lp->rx_dma_ptr = bp;
    return;
}
skb_reserve(skb, 2);          /* longword align L3 header */
skb->dev = dev;
if (bp + length > lp->end_dma_buff) {
    int semi_cnt = lp->end_dma_buff - bp;
    memcpy(skb_put(skb,semi_cnt), bp, semi_cnt);
    memcpy(skb_put(skb,length - semi_cnt), lp->dma_buff,
           length - semi_cnt);
} else {
    memcpy(skb_put(skb,length), bp, length);
}
bp += (length + 3) & ~3;
if (bp >= lp->end_dma_buff) bp -= lp->dmasize*1024;
lp->rx_dma_ptr = bp;
if (net_debug > 3) {
    printk("%s: received %d byte DMA packet of type %x\n",

```

```

        dev->name, length,
        (skb->data[ETH_ALEN+ETH_ALEN] << 8) | skb->data[ETH_ALEN+ETH_ALEN+1]);
    }
    skb->protocol=eth_type_trans(skb,dev);
    netif_rx(skb);
    dev->last_rx = jiffies;
    lp->stats.rx_packets++;
    lp->stats.rx_bytes += length;
}
#endif /* ALLOW_DMA */

```

코드 1165. dma_rx()함수의 정의(계속)

이전 DMA를 한 데이터를 socket buffer로 만들어서 상위에 알려줄 차례이다. 먼저 socket buffer를 하나 할당받아서 이를 skb가 가르키도록 만든다(dev_alloc_skb()). 만약 할당받을 수 없다면, lp->stats.rx_dropped를 하나 증가시켜서 Rx한 패킷이 drop된 것을 나타낸다. skip_this_frame은 버퍼의 포인터를 옮겨서 현재 처리중인 frame을 제거하는 역할을 한다. 즉, bp를 4bytes로 정렬한 length만큼 증가시키고, 만약 이렇게 증가시킨 값이 DMA buffer의 마지막(lp->end_dma_buff)보다 크거나 같다면, bp에서 lp->dmasize x 1024 만큼 빼주어서, bp를 reset시킨다. 그리고나서 다음번의 DMA buffer를 가르키는 포인터를 위해서 lp->rx_dma_ptr을 bp로 재설정 해주고 복귀한다.

나머지는 socket buffer를 만들어주고 상위의 프로토콜 레이어에 전달하는 일다. 먼저, 4bytes(32bit) 정렬 때문에, 할당 받은 socket buffer의 앞부분에 2bytes를 예약한다(skb_reserve()). 그리고, 소켓버퍼와 관련된 device를 명시해 준다(skb->dev). 만약 DMA를 한 buffer가 DMA buffer의 마지막 주소보다 더 크다면, 두 부분으로 나누어서 socket buffer의 데이터를 채워주게 되는데(memcpy()), 이것은 DMA가 ring buffer의 형태로 일어나기 때문이다. 즉, packet이 DMA buffer의 뒷부분에서 나누어져서, 다시 DMA buffer의 앞부분까지를 차지하게 되는 것을 말한다. 이를 위해서 bp와 DMA buffer의 마지막을 나타내는 end_dma_buff를 빼서, packet의 얼마만큼이 DMA buffer의 뒷부분에 들어와 있는지를 계산하고, 이 값만큼을 먼저 socket buffer에 복사한다. 다시 DMA buffer의 앞부분에서 나머지 만큼을 socket buffer로 복사해서 완전한 socket buffer를 만든다. 이를 간단히 그림으로 나타내면 [그림]와 같다.

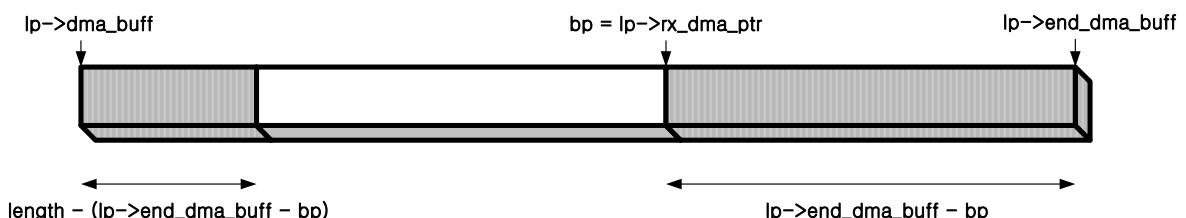


그림 134. Rx에서의 DMA buffer 사용 예(Rx가 ring buffer의 두 부분으로 나뉘어진 경우)

위와 같은 것에 해당하지 않는다면, 단순히 DMA buffer를 socket buffer로 복사해주기만 하면 될 것이다. 여기서 잠시 조금더 살펴볼 부분이 있다. 만약 DMA를 위한 Rx buffer를 이미 socket buffer의 형태로 만들어 준다면, 위에서 했던 DMA buffer를 socket buffer로 복사하는 오버헤드(overhead)를 줄일 수 있다는 점이다. 따라서, 만약 우리가 새로운 디바이스를 고려하고, 디바이스 드라이버를 작성한다면 이점을 간과해서는 안될 것이다⁴⁶⁷.

다시 이야기의 흐름으로 돌아와서, buffer 포인터는 4 byte 정렬을 해주고:(bp += (length + 3) & ~3), buffer 포인터가 lp->end_dma_buff를 벗어난다면, bp에 lp->dmasize x 1024를 빼준다. 이 위치로 다시 Rx를 위한 포인터의 위치를 설정한다. socket buffer의 데이터를 다 채워주었기에 eth_type_trans()를 사용해서 socket buffer의 protocol을 결정하고, netif_rx()를 사용해서 Rx한 패킷이 있음을 상위의 레이어에 알려준다. 마지막 Rx한 시간은 갱신되고(dev->last_rx), Rx packet의 수(lp->stats.rx_packets)와 Rx한 데이터의 크기(lp->stats.rx_bytes)는 각각 증가된다.

⁴⁶⁷ 물론 이것은 전적으로 개인적인 의견이다. 일단 구현해 놓고, 테스트를 해보는 것도 좋을 것이다.

17.8.10. Receive

네트워크 디바이스 드라이버의 받기(receive)와 관련된 연산은 항상 interrupt에 관련된 부분이다. 즉, 보내기와 관련된 것은 커널과의 interface가 정해져 있지만, 받는(receive)것은 일반적인 디바이스 드라이버에 대한 read 연산과 같은 관련성이 없다. 단지 인터럽트가 발생했을 때, 해당 frame(혹은 packet)을 읽어서, 상위의 프로토콜 layer에 알려주기만 할 뿐이다. 따라서, 인터럽트의 발생시에 이 인터럽트가 Rx 인터럽트인지 확인한 후, 이를 처리해준다.

```
static void net_rx(struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;
    struct sk_buff *skb;
    int status, length;

    int ioaddr = dev->base_addr;
    status = inw(ioaddr + RX_FRAME_PORT);
    length = inw(ioaddr + RX_FRAME_PORT);
    if ((status & RX_OK) == 0) {
        count_rx_errors(status, lp);
        return;
    }
    /* Malloc up new buffer. */
    skb = dev_alloc_skb(length + 2);
    if (skb == NULL) {
#if 0           /* Again, this seems a cruel thing to do */
        printk(KERN_WARNING "%s: Memory squeeze, dropping packet.\n", dev->name);
#endif
        lp->stats.rx_dropped++;
        return;
    }
    skb_reserve(skb, 2);          /* longword align L3 header */
    skb->dev = dev;
    insw(ioaddr + RX_FRAME_PORT, skb_put(skb, length), length >> 1);
    if (length & 1)
        skb->data[length-1] = inw(ioaddr + RX_FRAME_PORT);
    if (net_debug > 3) {
        printk("%s: received %d byte packet of type %x\n",
               dev->name, length,
               (skb->data[ETH_ALEN+ETH_ALEN] << 8) | skb->data[ETH_ALEN+ETH_ALEN+1]);
    }
    skb->protocol=eth_type_trans(skb,dev);
    netif_rx(skb);
    dev->last_rx = jiffies;
    lp->stats.rx_packets++;
    lp->stats.rx_bytes += length;
}
```

코드 1166. net_rx() 함수의 정의

먼저 디바이스의 기본 주소를 얻는다. 즉, 입출력과 상태를 보고하는 레지스터를 접근하기 위함이다. 이 기본 주소에 RX_FRAME_PORT를 더한 값으로 status와 길이를 보고 받는다. 만약 RX_OK가 아니라면, 이는 에러가 발생한 경우가 되므로 count_rx_errors() 함수를 호출해서 처리한다. 올바른 패킷을 받았다면(RX_OK), 먼저 패킷을 저장해서 상위의 프로토콜 layer에 올려주기 위해서 필요한 소켓 버퍼(socket buffer)를 하나 할당받는다. 즉, 소켓 버퍼만이 상위의 프로토콜 layer에게는 관심의 대상이 될 뿐이다. 할당 받을 수 없다면, 받은 패킷은 drop이되며, net_device 구조체의 priv부분에 있는 stats.rx_dropped 필드를 하나 증가시켜준다.

할당 받은 소켓 버퍼는 길이에 2를 더한 값으로 이는 소켓버퍼의 앞부분에 2 bytes의 공간을 두도록 만들기 위해서이다. 나중에, 데이터 부분만을 상위의 프로토콜은 다루게 되는데, 이때 32 bit(=4 bytes) 정열을 해주기 위함이다. 즉, 앞부분에 있는 ethernet header가 12 bytes를 차지하며, length 필드로 2 bytes가 따라오기 때문이다. 전체를 더하면, 16bytes의 공간이 앞부분을 차지한다(빈 공간 2bytes를 포함해서). 소켓버퍼의 dev필드를 현재의 디바이스를 가르키도록 만들고, 실제적인 패킷을 소켓 버퍼로 읽어들인다(`insw()`). 이때, `skb_put()`을 사용해서 데이터를 얼마만큼을 읽었는지 소켓 버퍼에 알려준다. 이전 소켓 버퍼가 어떤 프로토콜의 데이터를 가지고 있는지를 나타내어, 어느 프로토콜이 패킷이 도착했는지를 알도록 해주어야 한다. 이를 위해서 사용하는 것이, 소켓 버퍼의 protocol필드이며, 사용하는 함수는 `eth_type_trans()`이다. 프로토콜 layer에 패킷이 도착했음을 알려주어 처리를 요구하는 것은 `netif_rx()`이다. 디바이스의 마지막 Rx의 시간은 jiffies로 갱신되며, `net_stats`필드의 `rx_packet`의 수와, `rx_bytes`의 수가 각각 알맞게 증가된다.

`count_rx_errors()`함수는 Rx 패킷의 에러 정보를 갱신하는 것으로 `net_stats`속에 있는 해당하는 에러 필드를 갱신할 것이다. 아래와 같다.

```
static void
count_rx_errors(int status, struct net_local *lp)
{
    lp->stats.rx_errors++;
    if (status & RX_RUNT) lp->stats.rx_length_errors++;
    if (status & RX_EXTRA_DATA) lp->stats.rx_length_errors++;
    if (status & RX_CRC_ERROR)
        if (!(status & (RX_EXTRA_DATA|RX_RUNT)))
            /* per str 172 */
            lp->stats.rx_crc_errors++;
    if (status & RX_DRIBBLE) lp->stats.rx_frame_errors++;
    return;
}
```

코드 1167. `count_rx_errors()` 함수의 정의

넘겨받는 것은 상태 값과 `net_device`구조체의 `priv`필드를 가지는 `net_local`의 포인터이다. 이곳에서 Rx 에러에 관련된 필드를 업데이트한다. 먼저 `rx_errors`필드를 증가시킨다. 해당하는 에러 코드는 RX_RUNT(Rx under-run error), RX_EXTRA_DATA(Rx overflow error), RX_CRC_ERROR(Rx CRC error), RX_DRIBBLE(RX Dribble bits error)⁴⁶⁸가 있다. 복귀값은 없다.

17.8.11. Statistics

`ifconfig`과 같은 프로그램으로 `network device`의 통계정보를 얻을 수 있다. 이를 위해서 CS8900 디바이스 드라이버는 `net_get_stats()`함수를 제공한다.

```
/* Get the current statistics. This may be called with the card open or closed. */
static struct net_device_stats *net_get_stats(struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;
    unsigned long flags;

    spin_lock_irqsave(&lp->lock, flags);
    /* Update the statistics from the device registers. */
    lp->stats.rx_missed_errors += (readreg(dev, PP_RxMiss) >> 6);
    lp->stats.collisions += (readreg(dev, PP_TxCol) >> 6);
    spin_unlock_irqrestore(&lp->lock, flags);
```

⁴⁶⁸ 일반적인 상황에서 MAC은 받은 마지막 패킷의 byte이후에 있는 추가적인 7bits를 감지해서 dribble bit이라고 생각한다. 이 bit은 무시되며, 이 bit이 있으며, Rx의 상태 bit중에서 dribble bit부분이 설정된다. 만약 dribble bit과 CRC error bit이 동시에 설정되면, alignment error가 발생한다.

```

    return &lp->stats;
}

```

코드 1168. net_get_stats()함수의 정의

net_device의 priv필드에 이미 우리는 net_device_stats 구조체를 위한 공간을 할당 받아두었다. 단순히 이 정보를 return하면 될 것이다. 이를 위해서 먼저 interrupt가 발생하지 않도록 lock을 설정한 후(spin_lock_irqsave()), PP_RxMiss 레지스터와 PP_TxCol을 읽어서 Rx miss error와 Tx collision이 일어난 횟수를 얻어오고(가장 최신의 정보를 가지고 있을 것이기에), 설정한 lock을 풀어준다. 복귀 값은 priv필드의 stats의 주소가 된다.

17.8.12. Multicasting

Multicasting이란 여러개의 host에 동시에 패킷을 보내는 것을 말한다. 일반적은 ethernet controller의 경우, 이러한 multicasting을 위해서 사용할 하드웨어 주소를 저장할 메모리 공간을 가지고 있으며, 해당하는 패킷이 들어올 경우에 이를 받아서 상위로 올려준다. 따라서, multicasting을 위한 주소의 설정이 필요한데, CS8900에서는 set_multicast_list함수가 처리해 준다.

```

static void set_multicast_list(struct net_device *dev)
{
    struct net_local *lp = (struct net_local *)dev->priv;
    unsigned long flags;

    spin_lock_irqsave(&lp->lock, flags);
    if(dev->flags&IFF_PROMISC)
    {
        lp->rx_mode = RX_ALL_ACCEPT;
    }
    else if((dev->flags&IFF_ALLMULTI)||dev->mc_list)
    {
        /* The multicast-accept list is initialized to accept-all, and we
         * rely on higher-level filtering for now. */
        lp->rx_mode = RX_MULTICAST_ACCEPT;
    }
    else
        lp->rx_mode = 0;
    writereg(dev, PP_RxCTL, DEF_RX_ACCEPT | lp->rx_mode);
    /* in promiscuous mode, we accept errored packets, so we have to enable interrupts on them also */
    writereg(dev, PP_RxCFG, lp->curr_rx_cfg |
        (lp->rx_mode==RX_ALL_ACCEPT? (RX_CRC_ERROR_ENBL | RX_RUNT_ENBL |
            RX_EXTRA_DATA_ENBL) : 0));
    spin_unlock_irqrestore(&lp->lock, flags);
}

```

코드 1169. set_multicast_list()함수의 정의

먼저 Rx mode를 재 설정하기 때문에, interrupt가 발생하지 못하도록 lock을 설정한다(spin_lock_irqsave()). 만약 net_device구조체의 flags필드가 IFF_PROMISC로 설정되었다면, promiscuous mode로서 동작한다는 말이되며, 이것은 모든 네트워크 상의 패킷을 전부 받을 수 있도록 만든다. 따라서, priv구조체의 rx_mode필드에 RX_ALL_ACCEPT를 설정한다. 만약 net_device구조체의 flags필드가 IFF_ALLMULTI로 설정되어 있거나, mc_list(Multicast List)가 비어있지 않다면, priv구조체의 rx_mode에는 RX_MULTICAST_ACCEPT를 둔다. 해당 사항이 업다면, rx_mode에는 0이 들어갈 것이다. 참고로 IFF_ALLMULTI는 모든 multicast된 패킷을 다 받는다는 말이다.

이전 이것을 hardware의 register에 적어서 반영할 차례이다. PP_RxCFG에 현재의 Rx 설정과 다음에 설명할 bit을 OR시켜서 적는다. 먼저 rx_mode가 RX_ALL_ACCEPT로 설정된 경우에는 에러 패킷도 들어올 가능성이 있으므로 이에 대해서 interrupt가 발생할 수 있도록 만들어야

한다(RX_CRC_ERROR_ENBL | RX_RUNT_ENBL | RX_EXTRA_DATA_ENBL). 그럴지 않다면, 0을 OR값으로 준다. 마지막으로 함수를 복귀하기 전에 설정했던 lock을 푼다(spin_unlock_irqrestore()). 이렇게 하면, 새로운 Rx 모드 설정이 완료된다.

17.8.13. MAC address

MAC(Media Access Control) Address는 일반적인 ethernet card에서는 EEPROM에 생산시에 정해져서 들어가게 된다. 때로는 EEPROM이 없는 경우도 있으며, 이럴 때는 인위적으로 net_device 구조체의 MAC address에 관련된 부분을 설정해 주어야 할 것이다.

```
static int set_mac_address(struct net_device *dev, void *addr)
{
    int i;

    if (netif_running(dev))
        return -EBUSY;
    if (net_debug) {
        printk("%s: Setting MAC address to ", dev->name);
        for (i = 0; i < 6; i++)
            printk(" %2.2x", dev->dev_addr[i] = ((unsigned char *)addr)[i]);
        printk("\n");
    }
    /* set the Ethernet address */
    for (i=0; i < ETH_ALEN/2; i++)
        writereg(dev, PP_IA+i*2, dev->dev_addr[i*2] | (dev->dev_addr[i*2+1] << 8));
    return 0;
}
```

코드 1170. set_mac_address() 함수의 정의

netif_running은 net_device 구조체의 state 필드를 참조해서 현재 device가 사용중인지를 확인한다. 만약 사용중이라면, -EBUSY를 돌려준다. Debugging이 설정되었다면, device의 물리적인 하드웨어 주소를 출력해 줄 것이다. Ethernet 주소를 설정하는 것은 한번에 1word씩 PP_IA 레지스터에 기록하는 방법을 사용한다. 이렇게 해서 ethernet 주소 6 bytes를 기록한다.

17.8.14. Timeout

Tx에서 일정 시간이 지나서 timeout이 되면, 불려지는 함수로서 net_timeout()이 CS8900에 정의되어 있다. 단순히 다시 network adapter에 패킷을 전송하도록 요구하게 만든다.

```
#define tx_done(dev) 1
...
static void net_timeout(struct net_device *dev)
{
    /* If we get here, some higher level has decided we are broken.
       There should really be a "kick me" function call instead. */
    if (net_debug > 0) printk("%s: transmit timed out, %s?\n", dev->name,
                           tx_done(dev) ? "IRQ conflict ?" : "network cable problem");
    /* Try to restart the adaptor. */
    netif_wake_queue(dev);
}
```

코드 1171. net_timeout() 함수의 정의

단지, netif_wake_queue()를 net_device 구조체를 넘겨주면서 호출한다. tx_done()은 단순히 1로 정의되어 있다. netif_wake_queue() 함수는 Tx를 위한 패킷들이 들어 있는 queue를 새로이 scheduling하도록 요청하는 일을 한다. 따라서, 전송이 완료가 되지 않은 패킷은 재전송 될 것이다.

이제 우리는 network 디바이스 드라이버에 대해서까지 살펴보았다. 중요한 것은 네트워크 디바이스는 일반적인 디바이스 드라이버와는 다른 커널 인터페이스 구조⁴⁶⁹를 가지며, socket buffer라는 것을 사용한다는 점이다. 또한, 최대한 interrupt의 처리를 빨리 하기 위해서 Rx에 대해서 네트워크와 관련된 일부 작업만을 하고, 나머지는 커널이나 상위의 프로토콜 모듈에게 전적으로 의존한다는 점이다. 참고로 socket buffer를 다룰 때는 byte 정렬에 신경을 써주어야 하며, EEPROM과 같은 것이 없을 때는, 인위적으로 이 부분에 대한 초기화와 관련된 일도 해주어야 할 것이다. 네트워크 디바이스 드라이버는 상위의 프로토콜에 의존적이지 않으며, 오로지 자신이 받고 처리하며, 알려주는 데이터 구조는 socket buffer라는 것임을 명심하도록 하자.

⁴⁶⁹ 문자 디바이스 드라이버나 블록 디바이스 드라이버와는 많은 차이가 난다.

18. IR (Infra-Red) Device Driver

이번장에서는 Infra-Red(IR) 디바이스 드라이버의 작성법을 알아보기로 하자. 이를 위해서 선택할 수 있는 제품으로는 NSC(National Semi-Conductor)에서 나온 Geode SP1SC10을 사용하는 platform을 참고하겠다. 관련된 site로는 www.linux4tv.org를 보기 바란다. 이곳에서는 자세한 platform에 대한 설명보다는 IR쪽에 연결되는 device driver를 어떻게 작성하고 있는지 만을 볼 것이다.

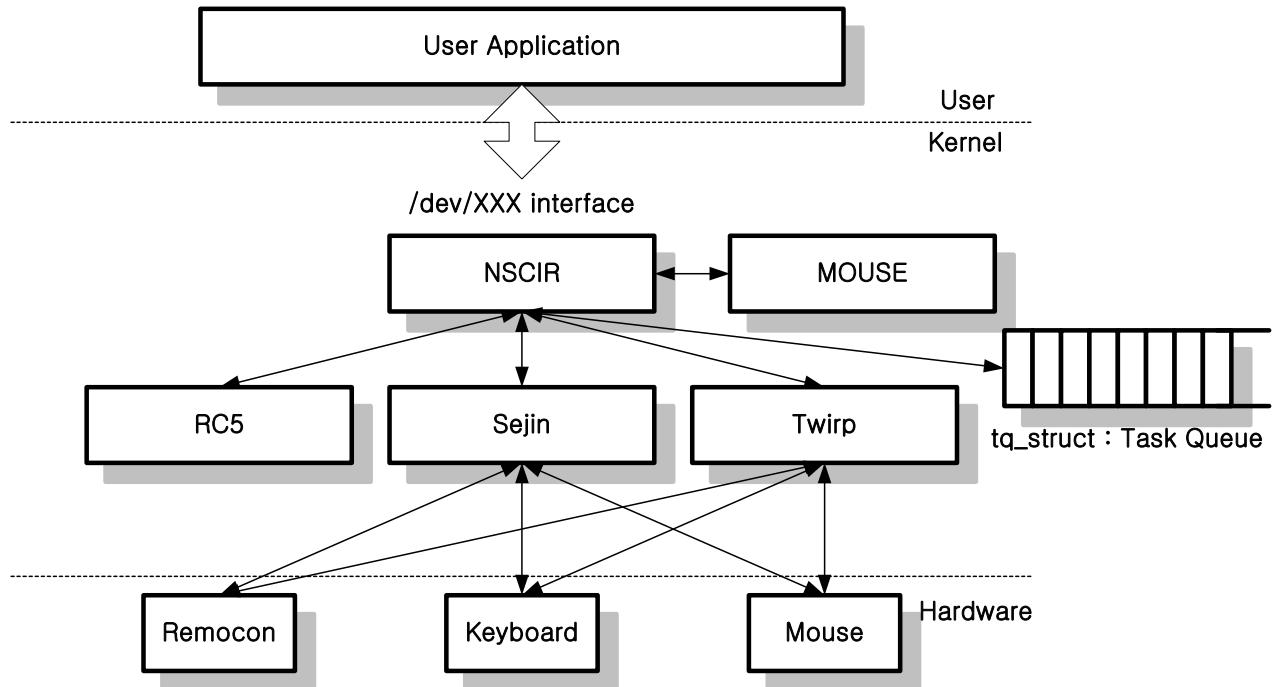


그림 135. NSC Set-top Box의 Infra-Red 디바이스 드라이버의 구성

NSC의 IR 디바이스 드라이버는 NSCIR을 중심으로 각각 IR protocol을 decoding하기 위해서 RC5와 Sejin, Twirp가 하나의 모듈을 이루고 있다. 이들 각각은 사용하는 IR 디바이스의 형태에 따라 정의된다. 또한 Linux의 bus mouse interface를 위한 모듈로는 mouse.c가 사용된다. 먼저 NSCIR에 대해서 살펴보고, MOUSE를 본 후, RC5와 Sejin, Twirp에 대해서 각각 살펴보도록 하겠다.

18.1. NSCIR 모듈의 분석

NSCIR이 모듈로서 동작하게 되면, init_module()과 cleanup_module() 함수가 각각 모듈의 loading과 unloading시에 호출된다. 여기서부터 분석을 시작하도록 하겠다.

```

int init_module (void)
{
    int i;

    PROTOCOL_ID=0;
    // initialization of the protocol registration structure begins
    NscIrReg.count=0;
    NscIrReg.protoMap=0;
    for(i=0;i<MAX_PROTO;i++)
    {
        NscIrReg.proto_init[i]= 0;
        NscIrReg.proto_decode[i]=0;
    }
}

```

```

// initialization of the protocol registration structure ends
outb(07, 0x015c) ; outb(02,0x015d);
return (nscir_init ());

}

void cleanup_module (void)
{
    if (unregister_chrdev (NSCIR_MAJOR, "nscir"))
    {
        printk ("nscir: error unregistering driver\n");
    }
    else
        remove_handler ();
}

```

코드 1172. init_module()과 cleanup_module() 함수의 정의

init_module()에서는 NSCIR에 등록된 IR protocol들에 대한 자료구조를 초기화 하는 역할을 한다. 실제로 IR protocol들은 전부 등록(register)된다고 생각해서 이를 위한 구조체로 NscIrReg라는 것을 사용해서 관리한다.

```

...
#define TWIRP_PROTOCOL 1
#define SEJIN_PROTOCOL 2
#define RC5_PROTOCOL 3
#define BYTE unsigned char
#define MAX_PROTO 8

...
typedef struct
{
    int      count;
    BYTE    protoMap;
    int (*proto_twirp_write)();
    int (*proto_init[MAX_PROTO])();
    int (*proto_decode[MAX_PROTO])();

}NscIrTest;
static NscIrTest NscIrReg;

```

코드 1173. NscIrTest 구조체의 정의

NscIrReg는 NscIrTest라는 구조체로 정의된다. 즉, 등록된 IR protocol이 몇개인지를 나타내는 count와 현재 mapping된 protocol이 뭔지를 나타내는 protoMap, 그리고 twirp에서만 사용하는 proto_twirp_write라는 함수의 포인터와 각각의 IR protocol에 맞게 초기화 및 IR 신호를 decoding하는 함수를 가지는 proto_init[]와 proto_decode[] 배열로 이루어져 있다.

모듈의 loading에서 PROTOCOL_ID를 인자로 넘겨줄 수 있으나, 이곳에서는 PROTOCOL_ID를 초기값인 0으로 두었다. 나머지는 NscIrReg구조체를 초기화 하기 위한 과정이며, outb()는 IR을 관리하는 register에 값을 write하는 과정이다. 각각은 I/O 영역 0x015C와 0x015D번지에 07와 02값을 쓴다. 이것은 Geode의 Super I/O module의 IR 부분을 사용하겠다고 알려주는 것이다⁴⁷⁰. 실제적인 초기화 작업은 nscir_init() 함수에서 수행될 것이며, 이 함수의 복귀 값이 init_module() 함수의 복귀 값이 될 것이다.

cleanup_module() 함수에서는 문자 디바이스로 등록한 것을 해제하기 위해서 unregister_chrdev()를 호출하고, 설치된 handler를 없애기 위해서 remove_handler() 함수를 호출한다.

```
int nscir_init(void)
```

⁴⁷⁰ National Semiconductor의 SC1200 메뉴얼을 참조하기 바란다.

```
{
    // register the character device
    if (register_chrdev (NSCIR_MAJOR, "nscir", &nscir_fops))
    {
        printk ("nscir: Fail to register IR device\n");
        return -EIO;
    }
    nscir_protocol_init();
    return 0;
}
```

코드 1174. nscir_init() 함수의 정의

nscir_init() 함수는 NSC의 IR 디바이스에 대한 실제적인 초기화를 담당한다. 먼저 register_chrdev()를 호출해서 NSCIR_MAJOR(= 241)에 해당하는 문자 디바이스를 등록한다. 이때 등록되는 파일 연산자 벡터 구조체로는 nscir_fops의 주소를 넘겨준다. 나중에 각각의 연산에 대해서 보기로 하고, 다음으로 넘어가 nscir_protocol_init()를 보도록 하자.

```
void nscir_protocol_init(void)
{
    // initialize National's super i/o
    init_superio ();
    // empty IR input queue

    switch(PROTOCOL_ID)
    {
        case TWIRP_PROTOCOL:
        // raw_ir_q_init();
        NscIrReg.proto_init[0]();
        break;
        case SEJIN_PROTOCOL:
        //raw_ir_q_init();
        // NscCirInitializeDetection();
        NscIrReg.proto_init[1]();
        break;
        case RC5_PROTOCOL:
        //raw_ir_q_init();
        //NscCirInitializeDetectionRC5();
        NscIrReg.proto_init[2]();
        break;
        default: break;
    }
    // install_handler ();
}
```

코드 1175. nscir_protocol_init() 함수의 정의

nscir_protocol_init() 함수는 NSC IR이 지원하는 IR protocol에 대한 처리를 담당할 NscIrReg 구조체 부분을 초기화 하는데 사용한다. 먼저 init_superio()를 호출해서 Geode super I/O 모듈에 대한 초기화를 수행한다. 만약 이때 이미 등록된 protocol[0] 있거나, PROTOCOL_ID가 있을 경우에는 해당 초기화 함수(proto_init[#protocol number]())를 호출한다. 하지만, 아직 모듈의 초기화에서는 아무런 protocol driver들이 등록되지 않았으며, PROTOCOL_ID 또한 0으로 주어져 있기 때문에, 그냥 빠져나가게 될 것이다.

```
static void init_superio (void)
{
    set_bank (0);      // force the UART into bank 0
```

```

set_extended_mode ();
stop_reception ();
clear_rx_fifo ();

// Set the prescaler to 13, and the divisor is set to 1.
// This yields 115200bps with a 0.16% error.
// This is around a 3x oversampling of the IR signal.
// Note: The sampling rate seems to depend solely on the divisor, and
// not on the prescaler. Thus, a divisor of 1 is the best we are
// likely to do.
// set_baudrate(PRESCALER_1_625, 1); // baud 7.2k
switch(PROTOCOL_ID)
{
    case TWIRP_PROTOCOL:
        set_baudrate(PRESCALER_13, 1); // baud 115.2k
        break;
    case SEJIN_PROTOCOL:
        set_baudrate(PRESCALER_13, 3); // baud 115.2k
        break;
    case RC5_PROTOCOL:
        set_baudrate(PRESCALER_13, 32);
        break;
    default :
        break;
}
set_fifo_size (); // reset FIFOs to 16 deep

// The internal demodulator isn't used. The following call sets
// the freq to 35.7-39.5kHz, if the demodulator is later used.
// set_demodulator(0xb, 0x1);
// set_demodulator(0xb, 0x2);

setup_ceir();
stop_reception ();
}

```

코드 1176. init_superio() 함수의 정의

init_superio() 함수에는 super I/O의 IR을 위한 설정을 다루는 부분이 많이 있다. set_bank()은 super I/O 모듈에서 앞에서 설정한 0x015C와 0x015D 레지스터의 값에 따라, 사용할 디바이스를 위한 레지스터의 bank를 설정하는 역할을 한다. 이때 이 bank를 접근하는 것으로 해당 디바이스에 제어를 가할 수 있다. 여기서 말하는 bank란 실제로는 여러 레지스터의 배열의 집합이라고 생각하면 된다. 즉, bank하나가 선택되면, 그 bank에 여러개의 레지스터들이 존재하고 있으며, 이 각각의 레지스터들은 디바이스를 제어할 수 있는 프로그래밍 인터페이스의 역할을 수행한다.

set_extended_mode() 함수는 bank 2에 있는 extended control 레지스터에 접근해서 extended 모드를 설정하고, loopback 모드로 동작하지 못하게 한다. 아직 디바이스가 완전히 초기화 되지는 못했으므로, 데이터를 받지 못하게 만들고(stop_reception()함수), Rx를 데이터 버퍼를 지운다(clear_rx_fifo()). 이하는 각 IR 프로토콜 별로 baudrate를 설정하는 부분이다(set_baudrate()). set_fifo_size()는 Rx FIFO의 크기를 설정하는 함수이며, setup_ceir() 함수는 bank 7에 있는 IR과 관련된 레지스터를 설정한다. 이것을 설정하고나서, 앞에서와 마찬가지로 디바이스가 open이 되지 않았으므로, 계속 Rx를 하지 못하도록 stop_reception() 함수를 호출한다.

```

static void set_bank (unsigned char bank)
{
    unsigned char val;
    static const char LINK_CONTROL = 0x00;
}

```

```

static const char BANK_SELECT = 0x80;

// the following values for the LCR are from p18 of the 87108 datasheet
static const char CHARACTER_8_BITS = 0x03;
static const char ONE_STOP_BIT = 0x00;
static const char NO_PARITY = 0x00;
static const char NO_BREAK = 0x00;

// writes into the first two banks may upset LCR values, so reload
static const char DEFAULT_PARAMS =
    CHARACTER_8_BITS | ONE_STOP_BIT | NO_PARITY | NO_BREAK;

// The following values are from Table 4 in the 87108 datasheet, p19
static const char code[8] =
{
    LINK_CONTROL | DEFAULT_PARAMS,           // Bank 0
    BANK_SELECT | DEFAULT_PARAMS,            // Bank 1
    0xe0,                                     // Bank 2
    0xe4,                                     // Bank 3
    0xe8,                                     // Bank 4
    0xec,                                     // Bank 5
    0xF0,                                     // Bank 6
    0xF4,                                     // Bank 7
};

if (bank > 7)
{
    //printk ("nscir: set_bank: bank %d out of range\n", bank);
    return;
}
val = code[bank];
outb (val, BSR_REG);
cur_bank = bank;
}

```

코드 1177. set_bank() 함수의 정의

set_bank() 함수는 파라미터 값으로 선택할 bank의 번호를 받는다. 이렇게 선택된 bank는 나중에 다른 함수들에 의해서 각각의 bank에 속하는 레지스터를 설정하기 위해서 사용될 것이다. Bank select(선택)은 각 bank의 LCR/BSR(Link Control / Bank Select)에 해당하는 0x03에 위치하는 레지스터(BSR_REG = 0x03)가 담당한다. 또한 현재 선택된 bank를 가르키기 위해서 cur_bank라는 변수를 사용한다. LINK_CONTROL과 BANK_SELECT는 각각 bank 0와 bank 1을 설정하기 위해서 사용하며, 나머지 각각의 bank들도 해당 bit을 enable한 값으로 code[] 배열에 select를 위한 값을 가진다. 만약 bank 변수가 7이상의 값을 가진다면, 이것은 잘못된 값이다. 즉, 현재 super I/O의 IR 모듈 설정에 있어서는 bank가 총 8개만을 가지기 때문이다⁴⁷¹. DEFAULT_PARAMS는 bank select에서 사용하지 않는 하위 2 bit을 특정 bank에 대해서 link control을 하기 위해서 넣은 부분이며, bank 0와 bank 1의 경우 WLS bit로 사용된다.

```

static void set_extended_mode (void)
{
    unsigned char value;

    if (cur_bank != 2)
        set_bank (2);

    value = inb (EXCR1_REG);

```

⁴⁷¹ NSC의 SC1200 메뉴얼을 참조하기 바란다. Super I/O에서는 IR Communication Port(IRCP)와 Serial Port 3(SP3)를 PMR(Pin Multiplexing Register)레지스터의 설정에 따라서 switching해서 사용하고 있다.

```

value |= 0x01;           // set extended mode
value &= ~0x10;          // turn off loopback
outb (value, EXCR1_REG);

// set CEIR mode
if (cur_bank != 0)
    set_bank (0);
outb (0xc0, MCR_REG);
}

```

코드 1178. `set_extended_mode()` 함수의 정의

`set_extended_mode()` 함수는 현재 bank가 2가 아닌 경우에 bank를 2로 설정하기 위해서 `set_bank()`함수를 호출한다. Bank 2의 EXCR1_REG 레지스터의 값을 읽어서, 현재 설정된 extended mode를 얻고, 이 값에 특정 값을 설정한 후, 원래의 EXCR1_REG 레지스터에 write한다. 즉, read-modify-write라는 방식으로 레지스터에 접근해서 해당 값을 바꾸는 일을 한다. 원래의 bank 값을 돌려주기 위해서 다시 `set_bank()` 함수를 호출하고, MCR_REG에 0xC0를 써서 extended mode에서 설정해야만 접근할 수 있는 MDSL [2:0] 레지스터의 값을 바꾼다.

```

static void clear_rx_fifo (void)
{
    const unsigned char RXSR = 0x02;

    if (cur_bank != 0)
        set_bank (0);

    // this register is write-only, so just set the desired bit
    outb (RXSR, FCR_REG);
}

```

코드 1179. `clear_rx_fifo()` 함수의 정의

`clear_rx_fifo()` 함수는 bank 0에 있는 FCR_REG(FIFO Control Register) 레지스터에 0x02를 write해서 Rx FIFO를 지우는 역할을 수행한다. FCR_REG는 실제로 EIR(Event Identification Register)와 같은 offset을 공유하는데, 이것은 두 레지스터가 CPU read와 CPU write cycle를 각각 달리 사용하기 때문이다. 즉, read시에는 EIR로, write시에는 FCR로 동작한다. RXSR은 Receiver Soft Reset을 나타내는 것으로 RX_FIFO를 지우도록 하는데 사용한다.

```

static void set_baudrate (unsigned char BaudRatePrescale, unsigned short BaudRateDivisor)
{
    unsigned char value;
    // printk(" set baud \n");

    if (cur_bank != 2)
        set_bank (2);

    value = inb (EXCR2_REG);
    // Clear the following bits:
    //      4-5:  clear prescaler select
    //      6:    reserved, set to 0
    //      7:    lock bit, set to 0 when extended mode selected
    value &= 0x0F;
    value |= (BaudRatePrescale << 4);
    outb (value, EXCR2_REG);

    // Load the Baud Rate Divisor
    value = BaudRateDivisor >> 8;           // high byte of divisor
}

```

```

outb (value, BGDH_REG);
outb (BaudRateDivisor & 0xff, BGDL_REG);
// outb (128, BGDL_REG); // this sets to 1800 bps , for test sejin
// outb (3, BGDL_REG); // this sets to 1800 bps , for test sejin
}

```

코드 1180. set_baudrate() 함수의 정의

Baudrate란 데이터를 전송하는 속도를 의미한다. Super I/O 모듈의 IR부분에서 이러한 baudrate를 조절하기 위해서는 레지스터의 bank를 2로 설정해야 하기에 cur_bank가 2가 아닌 경우에는 set_bank() 함수를 호출해서 2로 바꾼다. 먼저 바꾸기 전에 이전의 값을 읽어들이고(inb(EXCR2_REG)), 이 값에서 baudrate pre-scale값과 baudrate divisor값을 설정하도록 한다. BGDH_REG는 baudrate division high byte를 나타내며, BGDL_REG는 baudrate divisor low byte를 나타내는 위치이다. 각각에 BaudRateDivisor로 설정된 값을 상위와 하위로 나누어서 넣는다.

현재로서는 Twrip와 Sejin, RC5 protocol을 위해서는 PRESCALER_13(= 0x00)을 사용했다. 또한 각각에 대한 baudrate divisor값으로는 1, 3, 32를 사용하고 있다. 이렇게해서, 일단 각각의 IR protocol을 사용하는데 있어서 필요한 전송속도를 맞추는 것은 끝난다.

```

static void set_fifo_size (void)
{
    unsigned char value;

    if (cur_bank != 2)
        set_bank (2);
    switch(PROTOCOL_ID)
    {
        case TWIRP_PROTOCOL:
            value = inb (EXCR2_REG);
            value &= 0x0F;           // reset both FIFOs to 16 deep
            outb (value, EXCR2_REG);
            break;
        case SEJIN_PROTOCOL:
            value = inb (EXCR2_REG);
            value |= 0x04;          // reset both FIFOs to 16 deep
            outb (value, EXCR2_REG);
            break;
        case RC5_PROTOCOL:
            value = inb (EXCR2_REG);
            value &= 0x0F;           // reset both FIFOs to 16 deep
            outb (value, EXCR2_REG);
            break;
        default:
            break;
    }
}

```

코드 1181. set_fifo_size() 함수의 정의

set_fifo_size() 함수는 각각의 프로토콜에 맞는 데이터의 임시적인 저장을 가지는 하드웨어적인 FIFO의 크기를 설정하기 위해서 사용한다. 먼저 이러한 FIFO의 크기를 설정하기 위해서는 bank를 2로 바꾼다(set_bank()). 이전 현재 설정된 프로토콜이 원지를 파악한 후(PROTOCOL_ID), Twrip, Sejin, RC5각각에 대해서 EXCR2_REG(Extended Control Register 2)의 bit에 0x0F를 AND해서 LOCK과 PRESL0-PRESL1 두개의 bit를 지워주거나, 2번 bit(-RF_SIZE0 : Rx FIFO Size 레지스터)를 설정해 주는 일을 한다. 실제로 EXCR2 레지스터의 0번과 1번 bit는 Tx FIFO의 크기를 정해주기 위해서 사용한다. 이 두개의 Rx와 Tx FIFO 크기를 나타내주는 값이 00으로 설정된 경우에는 16 레벨의 FIFO 크기를 사용한다는

것이며, 01로 설정된 경우에는 32 레벨의 FIFO 크기를 사용하도록 한다. PRESL(Prescale)을 위한 두개의 bit는 00 으로 설정된 경우에는 13.0, 01인 경우에는 1.625, 11인 경우에는 1.0으로 24MHz로 동작하는 input clock 주파수를 나누어서 baud generator에 clock을 공급하기 위해서 사용된다. 10인 경우에는 RSVD(Reserved)값이다.

```
static void setup_ceir (void)
{
    unsigned long pmr;
    unsigned long mcr;
    unsigned char value;

    // printk(" set up CIR \n");

    // Sets the IR port signals for SP3.
    pmr = inl (0x9030);           // read PMR
    pmr &= 0xfffffbf;            // clear bit 6, SP3SEL
    outl (pmr, 0x9030);

    // Set MCR bit 4 = 1 (irtxen = IRTX enabled)
    mcr = inl (0x9034);          // read the Misc Configuration Reg
    mcr |= 0x10;                 // set IRTX/SOUT3 is enabled
    outl (mcr, 0x9034);

    if (cur_bank != 7)
        set_bank (7);

    // setup RCCFG register
    // old:  outb (0xd0, IO_BASE + 0x02);
    // for some reason SolutioNets did these as separate writes...

    // clear out RCCFG
    outb (0x0, RCCFG_REG);

    // enable RLE
    value = inb (RCCFG_REG);
    value |= 0x80;
    outb (value, RCCFG_REG);

    // disable internal demodulator
    value = inb (RCCFG_REG);
    value |= 0x10;
    outb (value, RCCFG_REG);

// ===== */
    set_bank (5);
    value = inb (IRCR2_REG);
    value &= ~0x02;                // MSR register and MS interrupt in UART mode
    value &= ~0x01;                // set half duplex mode
    outb (value, IRCR2_REG);

//=====sejin dongle mode ===
    set_bank (7);
    outb(0x48,IRCFG4);
    outb(0x06,IRCFG1);
    set_bank(0);

//=====
}
```

코드 1182. setup_ceir() 함수의 정의

`setup_ceir()` 함수는 Consumer Electronics IR을 설정하기 위한 것이다. PMR(Pin Multiplexing Register = 0x9030)를 먼저 읽어서, 이것을 IR로 사용한다고 알려준다. 즉, Serial Port 3(SP3)를 선택(select)하는 bit을 clear시킨다. 이젠 MCR(Miscellaneous Configuration Register = 0x9034)를 읽고, 이 값에 IRTX/SOUT3를 enable 시키도록 한다. 설정되는 bit은 Infra-red transmitter enable(IRTXEN)이라는 bit으로 IR을 enable 시키도록 만들어준다.

이것을 마치면 bank 7에 있는 RCCFG(Consumer-IR Configuration Register)를 접근하기 위해서 `set_bank()`를 호출한다. RCCFG 레지스터의 7번 bit은 Run Length Control 레지스터로 run length encoding과 decoding을 enable 시키거나 disable 시키는데 사용한다. 이 bit을 설정해서 run length encoding을 가능하도록 만든다. 그리고나서, 이젠 IR을 receive하는 쪽에서 demodulation을 하지 못하도록 RCDM_DS bit(Receiver Demodulation Disable)을 설정한다.

다시 bank 5로 이동해서 IR 제어를 위한 IRCR2(Infra-red Control Register 2)를 본다(`set_bank()`). 이 값을 읽어서, 0x03을 clear 시킨다. 0번 bit은 IR이 full duplex 모드로 동작하도록 만들어주기에 이것을 clear 시킨다는 것은 half duplex로 동작하도록 하는 것이며, 1번 bit은 MSR(Modem Status Register)인데, 이것을 0으로 만들어주면 carrier를 찾고(detect) wake-up event를 알려줄 수 있도록 만들어주기 위한 것이다. 즉, 이 MSR bit이 0으로 되어 있는 경우에는 Modem Status Interrupt가 발생하게되지만, 그렇지 않은 경우에는 일어나지 않는다.

이젠 bank를 7로 다시 바꾸고, IRCFG(Infra-Red Configuration) 레지스터 4에 0x48을 쓴다. 이것은 IRRX_MD(IRRX mode select) bit과 IRSL21_DS(ID/IRSL Pin Direction Select) bit을 설정해 준다. 즉, pin의 direction으로는 output으로 두고, SIR과 MIR/FIR⁴⁷² 각각에 대해서 분리된 input을 사용하겠다는 뜻이다. IRCFG1 레지스터에 0x06을 쓴것은 ID/IRSL[2-1] pin이 선택된 모드에 관계없이 IRSL21_DS가 구동되도록 만들어 준다. 즉, IR의 입력에 따라서 output쪽(시스템)으로 출력을 내보내게 된다는 것을 의미한다.

```
static void stop_reception (void)
{
    const unsigned char RXACT = 0x20;
    unsigned char value;

    if (cur_bank != 0)
        set_bank (0);

    // set the RXACT bit in the ASCR
    value = inb (ASCR_REG);

    value |= RXACT;
    outb (value, ASCR_REG);
}
```

코드 1183. `stop_reception()` 함수의 정의

`stop_reception()` 함수는 데이터를 받는 것을 멈추는 함수이다. 이것을 수행하기 위해서는 먼저 bank를 0으로 바꾸고(`set_bank()`), 현재 Super I/O의 IR 모듈이 이미 extended mode로 동작하고 있기에 bank 0의 0x07은 ASCR 레지스터를 가르키게 되며, ASCR 레지스터의 RXACT를 설정하도록 한다. 역시 데이터를 write하기 전에 어떤 값들이 ASCR_REG에 설정되었는지를 확인하고나서, 여기에 RXACT를 OR시키는 방법을 사용하고 있다.

18.1.1. Protocol의 등록/해지 및 변경

NSC의 IR에서는 여러가지의 IR 장치들이 데이터를 주고 받을 수 있도록 하기위해서 각각이 사용하는 자신들만의 데이터 통신 규칙을 가질 수 있다. 이를 프로토콜이라고 하며, IR 디바이스 드라이버의 핵심 역할을 하는 NSCIR에는 프로토콜을 위한 자료구조를 등록 및 해제하기 위한 함수를 정의하고 있다.

```
int NscIrRegProto(int protocol_id, int (*init_proto)(),int (*decode_proto)(),int (*spl_proto)())
```

⁴⁷² SIR(Single IR)은 데이터 스피드를 115.2Kbps를 지원하며, MIR은 1.152Mbps를 지원한다. FIR의 경우에는 4.0 Mbps를 지원한다.

```
{
    if( NscIrReg.protoMap & (1<<(protocol_id-1)))
    {
        printk (" Protocol is already installed \n");
        return 0;
    }
    if((protocol_id > MAX_PROTO) || (protocol_id<=0))
    {
        printk(" Unknown protocol ");
        return 0;
    }
    if( ( init_proto != 0 ) && ( decode_proto != 0 ) )
    {

        NscIrReg.protoMap |= ( 0x01 << (protocol_id-1) );
        NscIrReg.proto_init[protocol_id-1] = init_proto ;
        NscIrReg.proto_decode[protocol_id-1] = decode_proto ;
        if(protocol_id == 1)
        {
            // because twirp implementation different
            NscIrReg.proto_twirp_write = spl_proto;

        }
        switch_proto( protocol_id );
        NscIrReg.count++;
        return 1;
    }
    printk("Illegal Registartion \n");
    return 0;
}
}
```

코드 1184. NscIrRegProto() 함수의 정의

NscIrRegProto() 함수는 IR 디바이스가 사용하는 프로토콜을 등록하기 위해서 모듈의 외부로 export된다. 따라서, NSCIR 디바이스가 이미 로딩이 되어있다면, 이 함수를 사용해서 IR 디바이스 각각에 대한 프로토콜을 등록할 수 있다. 등록하기 위해서는 초기화와 디코딩(decoding) 및 특정 프로토콜에 대한 write관련 함수의 포인터를 넘겨준다(spl_proto()).⁴⁷³

먼저, 이미 해당 프로토콜이 등록되어 있는지를 NscIrReg.protoMap과 비교해보고, 만약 이미 등록되어 있다면 0을 돌려준다. 이전 프로토콜 ID가 정당한지를 본다(protocol_id). 이전 해당 프로토콜을 적절하게 NscIrReg에 설정해 주는 일을 수행한다. 즉, 프로토콜의 등록 상황과 초기화 함수 및 프로토콜의 decoding을 수행할 함수를 등록한다. 만약 Twirp 프로토콜을 사용한다면, 이를 위해서 spl_proto를 NscIrReg.proto_twirp_write부분에 넣도록 한다. 등록되는 각각의 프로토콜에 대해서 switch_proto() 함수를 호출하고, 등록된 프로토콜의 수(NscIrReg.count)를 증가시켜준다. 복귀 값은 1이다. 만약 init_proto() 함수와 decode_proto() 함수 둘중에서 어떤 것이라도 NULL 값을 가진다면, 오류 메시지를 보이고 0을 돌려줄 것이다.

```
int NscIrUnRegProto(int protocol_id)
{
    int i;

    if( NscIrReg.protoMap & (1<<(protocol_id-1)) )
    {
        if(PROTOCOL_ID == protocol_id )
        {
            if((i=get_next_proto())!=0)
```

⁴⁷³ 현재로서는 Twirp 프로토콜의 경우에만 사용하고 있다. RC5와 Sejin의 경우에는 이 부분을 사용하지 않는다.

```

        switch_proto(i);
    }
    NscIrReg.count--;
    if(NscIrReg.count==0)
    {
        PROTOCOL_ID =0;
        remove_handler();

    }
    NscIrReg.protoMap &= (~(1<<(protocol_id-1))); // mask ,disable
    NscIrReg.proto_init[protocol_id-1] = 0 ;
    NscIrReg.proto_decode[protocol_id-1] = 0 ;
    return 1;
}
else return 0;
}

```

코드 1185. NscIrUnRegProto() 함수의 정의

NscIrUnRegProto() 함수는 앞의 함수의 역을 수행한다. 넘겨받는 값은 해제할 프로토콜의 ID 값이다. 먼저, 해당 프로토콜을 등록 맵(map)에서 해제하고, 현재 설정된 프로토콜과 해제하려고 하는 프로토콜의 ID값이 같다면, 다른 프로토콜이 있는지 찾아본다(get_next_proto()). 만약 있다면, 그 프로토콜로 전이한다(switch_proto()). 이전 count값을 감소시키고, 만약 등록된 프로토콜이 아무것도 없다면, 더 이상 IR 디바이스가 데이터를 전달하더라도 받지 않기 위해서 remove_handler() 함수를 호출한다. 나머지는 앞에서 한 과정을 반대로 수행하는 것이다. 여기까지 오류없이 진행했다면 1을 돌려줄 것이다. 그렇지 않다면 복귀 값은 0이다.

```

void switch_proto( int proto_id)
{
    printk("\n  Switching to %s protocol\n",proto_name[proto_id-1]);

    if(NscIrReg.count) remove_handler();

    PROTOCOL_ID = proto_id;
    nscir_protocol_init();

    install_handler();
}

```

코드 1186. switch_proto() 함수의 정의

switch_proto() 함수는 넘겨받는 프로토콜 ID번호로 프로토콜을 전환하는 일을한다. 이때 이미 하나 이상의 IR 데이터에 대한 프로토콜이 있다면, 이들이 설정하고 있을지도 모를 handler를 먼저 제거하고, 현재 설정된 프로토콜의 ID(PROTOCOL_ID)에 넘겨받은 프로토콜 ID를 넣어준다. 이전 해당 프로토콜에 대한 초기화를 실행하기 위해서 nscir_protocol_init() 함수를 호출하고, 데이터를 받기 위해 인터럽트를 처리할 수 있는 핸들러를 설정한다(install_handler()).

```

int get_next_proto()
{
    int i=0,prev=0;
    unsigned char      gen;

    if(PROTOCOL_ID ==0) return 0;
    gen = NscIrReg.protoMap;
    gen = gen >> PROTOCOL_ID;
    for(i=PROTOCOL_ID;i<MAX_PROTO;i++)
    {

```

```

        if(gen  & 0x01) return i+1;
        gen >>= 1;
    }
    gen = NscIrReg.protoMap;
    gen <=> (MAX_PROTO-PROTOCOL_ID)+1;
    for(i=0; i<PROTOCOL_ID;i++)
    {
        if( gen  & 0x80) prev =i+1;
        gen <=> 1;
    }
    if(prev) return (PROTOCOL_ID -(prev));
    else      return 0;
}

```

코드 1187. get_next_proto() 함수의 정의

get_next_proto() 함수는 다른 등록된 프로토콜이 있는지를 찾아서, 그 프로토콜의 ID를 돌려주는 일을 한다. 먼저 현재의 프로토콜의 ID가 0이라면, 아무런 프로토콜도 등록되지 않았으므로 0을 돌려줄 것이다. 이전 현재 등록된 프로토콜이 어떤 값을 가진다면, 등록 맵(map)에서 가져온 값을 현재 프로토콜의 ID값 만큼 우측으로 SHIFT한다. 즉, 각각의 프로토콜은 등록시에 자신에 해당하는 bit를 설정하도록 하였기에 이렇게 하는 것을 현재 프로토콜의 ID보다 더 높은 ID값을 가지는 프로토콜이나 더 낮은 값을 가지는 프로토콜을 찾을 수 있게 된다. 만약 이 이렇게해서 0이 아닌 맵(map)상에 존재하는 값이 있다면, ID로 변환해서 돌려주게 된다. 이 과정은 상위 bit를 먼저 살펴보고, 다시 하위의 bit를 살펴보도록 되어 있으면, 자신과 가장 가까운 위치에 있는(즉, 절대값 상으로 가장 인접한) 프로토콜 ID값을 돌려준다. 찾을 수 없다면 복귀 값은 0이 될 것이다.

```

static void set_demodulator (unsigned char freq_code, unsigned char bw_code)
{
    unsigned char value = (bw_code << 5) | freq_code;

    if (cur_bank != 7)
        set_bank (7);

    outb (value, IRRXDC_REG);
}

```

코드 1188. set_demodulator() 함수의 정의

set_demodulator() 함수는 정의만 되어있고, 현재로서는 사용하지 않고 있다. 즉, 현재 NSC IR에서는 demodulation을 하지 않고, 바로 데이터를 사용하기 때문이다. 여기서 이 함수를 보는 것은 코드상에는 정의되어 있기 때문이다.

먼저, 전달받는 인수로는 주파수(frequency)를 나타내는 freq_code값과, 대역폭(bandwidth)를 나타내는 bw_code를 받는다. IRRXDC_REG 레지스터는 Infrared Receiver Demodulation Control 레지스터라고 불리며, Sharp IR이나 CEIR(Consumer Electronics IR)로 사용될 경우에 사용될 수 있다. bw_code를 좌측으로 5 bit SHIFT하는 이유는 하위 5 bit를 주파수가 사용하고, 나머지 3 bit만을 대역폭이 사용하기 때문이다.⁴⁷⁴

18.1.2. File Operation Vector

파일 연산 백터는 앞에서 이미 문자 디바이스 드라이버를 등록하면서, 해당 디바이스에 대한 파일 연산시에 적용된다. 따라서, 사용자 프로그램은 NSC IR 디바이스 드라이버에 대한 파일 연산들의 결과로 이 함수들의 복귀 값이 주어질 것이다.

```
static struct file_operations nscir_fops =
```

⁴⁷⁴ PC87108AVHG/JE 메뉴얼에는 이 부분에 대해서 자세한 설정값과 이렇게 설정될 경우에 사용되는 주파수 및 대역폭에 대한 것이 각각의 모드별로 잘 정리되어 있다.

```
{
    poll:          nscir_poll,
    ioctl:         nscir_ioctl,
    open:          nscir_open,
    release:      nscir_release,
};
```

코드 1189. NSC IR의 파일 연산 벡터의 정의

각각의 함수들에 대해서 살펴보도록 하자. 정의되지 않은 나머지 함수들은 전부 NULL이 되며, 나중에 응용프로그램이 해당 함수를 호출하기전에 NULL인지를 확인하는 절차⁴⁷⁵가 있으므로 별문제가 되지 않는다.

```
static int nscir_open (struct inode *inode, struct file *file)
{
    unsigned int minor = MINOR(file -> f_dentry -> d_inode -> i_rdev);

    MOD_INC_USE_COUNT;
    if (open_count == 0)
    {
//        install_handler ();
        printk("National IR Driver: %s\n", VERSION);
    }
    open_count++;
//    printk("\n Mask Value : %x %x \n", inb(0x21),inb(0x20));
    return 0;
}

static int nscir_release (struct inode *inode, struct file *file)
{
    MOD_DEC_USE_COUNT;
    open_count--;
    return 0;
}

static int nscir_ioctl (struct inode *inode, struct file *file,
                      unsigned int cmd, unsigned long arg)
{
    return 0;
}

static unsigned int nscir_poll (struct file *file, poll_table *wait)
{
    return (-ENODEV);
}
```

코드 1190. NSC IR의 파일 연산 함수들의 정의

NSC IR의 파일 연산 함수들은 대부분 거의 아무일도 하지 않는다. 즉, NSC IR이 제어하는 특정 디바이스를 직접적으로 응용 프로그램에서 접근할 필요가 없기 때문이다. 나중에 보면 알 수 있듯이 응용 프로그램은 자신이 받고 싶어하는 데이터가 keyboard나 mouse와 같은 형태로 들어오기에 NSC IR 디바이스를 직접적으로 사용할 가능성은 없다. Open() 함수는 모듈의 사용 계수를 높여주고, open의 회수를 증가시켜주며(open_count) 0을 돌려준다. Release() 함수는 open() 함수의 역을 수행한다. Ioctl() 함수는 단순히 0을 돌려주기만 하며, poll()은 지원하지 않기에 -ENODEV를 복귀 값으로 넘겨준다.

⁴⁷⁵ 물론 이것은 커널에서 이루어지지만, 응용프로그램의 컨텍스트(context)하이다.

실제적인 IR 디바이스에서 생길 수 있는 데이터 처리 요청은 interrupt handler에서 등록된 각각의 프로토콜중 현재 ACTIVE한 것이 처리할 것이다.

18.1.3. Interrupt Handler의 설정과 해제

이전 IR 디바이스에서 발생하는 interrupt의 처리에 대해서 알아보기로 하자. 즉, 인터럽트 핸들러에 대한 것이다. IR로 연결된 디바이스들은 자신이 데이터를 전송하면, host(혹은 IR receiver)쪽에서는 인터럽트를 발생시킬 것이다.

```
static void install_handler (void)
{
    int result;

    result = request_irq (IO_IRQ, interrupt_handler, SA_INTERRUPT,
                         "nscir", NULL);
    if (result)
    {
        printk ("nscir: can't get assigned irq %d\n", IO_IRQ);
        return;
    }
// else printk ("nscir: interrupt %d hooked\n", IO_IRQ);
// initialize task for bottom half of interrupt handler

    twirp_task.routine = raw_ir_in;
    sejin_task.routine = raw_ir_in1;
    rc5_task.routine = raw_ir_in2;

    // initialize queue indexes
    // raw_ir_q_init();

    if (cur_bank != 0)
        set_bank (0);
    // enable Rx interrupts
    outb (0x1, IER_REG);
}
```

코드 1191. install_handler() 함수의 정의

install_handler() 함수는 switch_proto() 함수에서 프로토콜의 초기화가 이루어진 후에 호출된다. 이 함수가 하는 일은 인터럽트 핸들러를 설정하는 것이다. 간단히 request_ir() 함수를 호출하므로써 이루어진다. 설치되는 인터럽트 핸들러는 interrupt_handler() 함수이며, 이 함수는 IO_IRQ(=7)번을 SA_INTERRUPT⁴⁷⁶속성으로 할당받아서 사용한다.

이하는 각각의 프로토콜에서 처리될 task들을 다루게 되는 함수들을 등록하는 것이다. Twirp, Sejin, RC5각각에 대해서 raw_ir_in(), raw_ir_in1(), raw_ir_in2() 함수가 핸들러로 등록된다. 이렇게 하고난 후에 인터럽트를 드디어 enable하게 되는데, 이를 위해서 IER_REG(Interrupt Enable Register)를 접근하도록 bank를 0으로 바꾼다. 바꾼 후에 IER_REG에 0x1⁴⁷⁷을 써서 interrupt를 발생시키도록 만든다. 주의할 점은 먼저 인터럽트 핸들러를 등록한 후에 인터럽트를 enable로 바꾼다는 점이다.

```
static void interrupt_handler (int irq, void *dev_id, struct pt_regs *regs)
```

⁴⁷⁶ 이 속성은 다른 interrupt들이 발생하지 못하게 한 후, 인터럽트 핸들러를 실행하라는 의미로 해석된다. 따라서, 이 인터럽트 핸들러가 수행중인 상황에서는 NON-Maskable 인터럽트를 제외한 다른 인터럽트들은 처리되지 않을 것이다.

⁴⁷⁷ Extended Mode에서 이렇게 설정되는 bit은 RXHDL_IE bit으로 Receiver High-Data-Level Interrupt Enable이다. 이 bit을 설정하면, RX_FIFO가 threshold값보다 크거나 같은 값을 가지거나, 혹은 timeout이 발생했을 때 인터럽트가 발생하도록 만든다.

```

{
    unsigned char lsr;
    unsigned char byte;
    static int ff_cnt =0;
    static unsigned char pre_val =0;

    if (cur_bank != 0) set_bank (0);
    // read all data from the Rx FIFO
    lsr = inb (LSR_REG);
    while (lsr & 0x01)
    {
        byte = inb (RXD_REG);
        switch(PROTOCOL_ID)
        {
            case TWIRP_PROTOCOL :
                if ((byte & 0x7f) == 0x7f)
                    byte |= 0x80;
                // enqueue data, and schedule bh
                // raw_ir_q_write(byte);
                NscIrReg.proto_twirp_write(byte);
                queue_task (&twirp_task, &tq_immediate);
                mark_bh (IMMEDIATE_BH);
                break;
            case SEJIN_PROTOCOL :
                sjnWIndex= (sjnWIndex ) % MAX_SJ_BUF;
                sejinRawBuf[sjnWIndex] = byte;
                sjnDataCount++;
                sjnWIndex++;
                queue_task(&sejin_task,&tq_immediate);
                mark_bh (IMMEDIATE_BH);
                break;
            case RC5_PROTOCOL :
                rc5WIndex= (rc5WIndex ) % MAX_RC5_BUF;
                rc5RawBuf[rc5WIndex] = byte;
                rc5DataCount++;
                rc5WIndex++;
                queue_task(&rc5_task,&tq_immediate);
                mark_bh (IMMEDIATE_BH);
                break;
            default :
                break;
        }
        lsr = inb (LSR_REG);
    }
    return;
}

```

코드 1192. interrupt_handler() 함수의 정의

interrupt_handler() 함수는 인터럽트가 발생했을 때, 현재 설정된 프로토콜(PROTOCOL_ID)에 맞게 발생한 데이터를 읽어서, 해당 프로토콜의 bottom half 핸들러에게 넘겨주는 역할을 한다. 나중에 해당 프로토콜의 bottom half 핸들러가 이렇게 발생된 데이터를 처리할 것이다.

먼저, 현재의 bank가 0이 아닌 경우 발생한 데이터를 읽기 위해서 bank를 0으로 바꾼다(set_bank()). LSR_REG는 Link Status Register로 Rx시에 발생한 사항들에 대한 것을 담고 있다. 0번 bit은 Rx Data Available bit으로 이 값이 1로 되어 있을 경우에 Rx 데이터가 있다는 의미로 해석된다. 따라서, 이 경우에 대해서만 처리하고, 나머지는 오류로 처리하면 될 것이다. LSR의 0번 bit이 설정되었다면, 이전 실제로 데이터를 읽어들일 차례이다. 이것은 bank 0의 index 0x00에 있는 RXD_REG 레지스터이다. 이

레지스터는 Receive Data Port(RXD) 또는 Transmitter Data Port(TXD)로 불리며, 이름에서도 알 수 있듯이 Rx/Tx시에 데이터를 전송하는 port의 역할을 수행한다. FIFO가 enable된 상황에서는 RX_FIFO나 TX_FIFO로부터 데이터를 읽어서 처리하도록 되어 있으며, 전체 8 bit의 데이터를 한번에 Rx/Tx를 할 수 있다.

읽혀진 데이터는 각 프로토콜에 맞게 처리되는데, 이때 크게 3가지의 과정을 거치게 된다. 즉, 프로토콜 모듈들이 관리하는 자신들만의 데이터 buffer에 데이터를 저장하고, 이 데이터를 처리할 task를 활성화 지정한 후에(queue_task()), bottom half를 활성화(mark_bh()) 시켜주는 것이다. Twirp 프로토콜의 경우에는 buffer에 데이터를 저장할 때 조금 특이한 방법으로 데이터의 값을 조정하는데, 읽은 데이터가 0x7F와 같은지를 확인한 후, 만약 같다면 이 값에 0x80을 OR 시켜서 앞에서 프로토콜을 등록할 때 twirp 프로토콜을 위해서 등록된 write() 함수를 호출한다.

Bottom half에서는 각각의 데이터 큐에 대해서 앞에서 install_handler() 함수를 호출할 때 지정해준 bottom half 핸들러 함수들을 호출하게 될 것이다. 이것을 마치면, 다시 새로운 데이터가 처리를 대기하는지를 알기 위해서 LSR_REG를 읽어본다. 이 과정은 데이터가 available하지 않을 때까지 반복하게 될 것이다.

```
static void remove_handler (void)
{
    outb (0x0, IER_REG);           // disable interrupts
    free_irq (IO_IRQ, NULL);      // release IRQ

//    printk ("nscir: unhooked interrupt\n");
}
```

코드 1193. remove_handler() 함수의 정의

remove_handler() 함수는 먼저 인터럽트를 disable 시킨 후, 앞에서 설정했던 인터럽트 핸들러를 해제한다(free_irq()). 하지만, 여기서는 각 프로토콜에 대한 bottom half 핸들러는 해제하고 있지 않다. 이것은 나중에 bottom half를 처리할 경우, 문제가 될 수 있는 여지를 없애주기 위한 것이다. 즉, 인터럽트를 disable시키더라도, bottom half로 처리될 프로토콜의 task가 남아있을 수 있기 때문이다.

18.1.4. Protocol Bottom Half Handler

Bottom half 핸들러는 인터럽트의 수행을 마치고, 사용자 모드로의 진입이 있기전에 호출된다. 따라서, 사용자 모드보다는 우선 순위가 높다고 할 수 있지만, 인터럽트 보다는 우선순위가 낮다⁴⁷⁸. 따라서, 이러한 bottom half를 이용해서 인터럽트 핸들러에서 다 해주지 못한 일을 처리하는 것이 훨씬 더 효율적으로 인터럽트를 사용할 수 있도록 만들어준다.

```
static void raw_ir_in (void * unused)
{
    int ret_val ;
    int i;

//    if( (ret_val=NscIrRawDataTwirp(1))==1 )
    if( (ret_val=NscIrReg.proto_decode[0](1))==1 )
    {
        stop_and_flush ();
        // ret_val=NscIrRawDataTwirp(2);
        ret_val=NscIrReg.proto_decode[0](2);
    }
    if(ret_val == 0xffff)
    {
        if((i=get_next_proto())!=0)
            switch_proto(i);
    }
}
```

⁴⁷⁸ 물론 이것을 우선 순위라는 표현을 써도 될지는 모르겠지만, 처리가 먼저 일어난다는 점은 옳을 것이다.

{}

코드 1194. raw_ir_in() 함수의 정의

raw_ir_in() 함수는 twirp 프로토콜에 대한 decoding을 담당하는 함수이다. 해당 protocol의 decoding 함수를 사용해서 데이터를 처리하려고 한다. 이때 만약 복귀 값이 1이라면, 오류로 처리하도록 한다. 따라서, 받는것을 중단하고, 다시 프로토콜을 초기화 시키기 위해서 stop_and_flush() 함수를 호출한다. 이와 같은 처리를 한 후, 이번에는 decoding함수에 2라는 값을 인자로 주어서 호출하는데, 이것은 twirp 프로토콜의 decoding함수가 크게 2부분으로 나누어져 처리되기 때문이다.

호출의 복귀 값으로 0xFFFF를 받는다면, 다른 프로토콜이 있는지를 알아보게 되며, 만약 발견된다면 해당 프로토콜로 decoding모드를 전환한다(switch_proto())。

```
static void stop_and_flush (void)
{
    stop_reception ();
    // raw_ir_q_init();
    NscIrReg.proto_init[0]();
}
```

코드 1195. stop_and_flush() 함수의 정의

stop_and_flush() 함수는 데이터 받기를 중단하고, 프로토콜을 초기화 시키는 역할을 수행한다. 이 함수는 twirp 프로토콜에 대해서만 수행되므로, 초기화 시키는 프로토콜도 twirp의 초기화 함수를 수행할 것이다.

```
static void raw_ir_in1 (void * unused)
{
    unsigned int i,j;

    while( sjnDataCount > 0 )
    {
        sjnDataCount --;
        //
        if(0xffff==NscIrRawDataSejin(sejinRawBuf[sjnRIndex]))
        if(0xffff==NscIrReg.proto_decode[1](sejinRawBuf[sjnRIndex]))
        {
            //printk(" switching needed \n");
            if((i = get_next_proto()))
                switch_proto(i);
        }
        else
        {
            //
            if(sejinRawBuf[sjnRIndex]!= 0xff)
                printk("%x ",sejinRawBuf[sjnRIndex]);
            sjnRIndex = sjnRIndex + 1 ;
            sjnRIndex= (sjnRIndex ) % MAX_SJ_BUF;
        }
    };
}
```

코드 1196. raw_ir_in1() 함수의 정의

raw_ir_in1() 함수는 Sejin 프로토콜에 대한 decoding을 담당하는 bottom half 핸들러이다. 이 핸들러가 보는 데이터의 buffer는 sejinRawBuf[]이며, 현재 처리해야하는 데이터의 위치는 sjnRIndex라는 변수로 나타낸다. 처리할 데이터의 크기는 sjnDataCount라는 변수로 나타내서, 이 값이 0이상인 동안 계속 수행한다. 물론 sjnDataCount는 한번 처리가 일어날 때마다 그 값이 감소할 것이며, sjnRIndex는 버퍼의 크기 만큼 증가한 후, 링 버퍼(ring buffer)를 만들기 위해서 초기값인 0을 가지게 될 것이다. Decoding은 역시 프로토콜 모듈에서 설치한 함수가 맡을 것이며, 만약 이 decoding함수의 복귀 값이 0xFFFF인

경우에는 다른 설치된 프로토콜 모듈이 있는지를 확인한 후 그 프로토콜로 모드를 전환할 것이다(switch_proto()). 최대 저장할 수 있는 Sejin 프로토콜의 데이터 크기는 MAX_SJ_BUF(= 1024) bytes이다.

```
static void raw_ir_in2 (void * unused)
{
    unsigned int i,j;

    while( rc5DataCount > 0 )
    {
        rc5DataCount --;

        // if(0xffff==NscIrRawDataRC5(rc5RawBuf[rc5RIndex]))
        if(0xffff==NscIrReg.proto_decode[2](rc5RawBuf[rc5RIndex]))
        {
            // printk(" switching is required \n");
            if((i=get_next_proto())!=0)
                switch_proto(i);
        }
        else
        {
            //	printk("<%x> \n",rc5RawBuf[rc5RIndex]);
            rc5RIndex = rc5RIndex + 1 ;
            rc5RIndex= (rc5RIndex ) % MAX_RC5_BUF;
        }
    };
}
```

코드 1197. raw_ir_in2() 함수의 정의

raw_ir_in2() 함수는 RC5 프로토콜 모듈의 데이터 버퍼를 처리하기 위한 bottom half 핸들러 함수이다. Sejin과 동일한 구조를 가지며, 처리하는 데이터 버퍼는 rc5RawBuf[] 배열이, 처리하고자 하는 데이터의 인덱스는 rc5RIndex 변수가 가르킨다. 최대 링 버퍼의 크기는 MAX_RC5_BUF(=1024) bytes이다.

이상으로 대략적으로 나마, National Semi-Conductor의 SC1200에 사용된 PC87364 Super I/O 모듈의 IR(Infrared) 디바이스 드라이버의 핵심 부분에 대한 설명을 마치도록 하겠다. 이하에서는 이제 이 디바이스 드라이버 모듈을 이용하는 프로토콜 드라이버들에 대해서 설명하도록 할 것이다.

18.2. Mouse 모듈의 분석

이곳에는 IR 인터페이스가 어떻게 bus mouse interface로 사용되는지를 살펴보도록 하겠다. 먼저 볼 것은 IR bus mouse 모듈에 대한 설명이며, 이후에 Linux 커널에서 사용되는 bus mouse interface를 위한 자료구조와 함수들이다.

18.2.1. Infrared Bus Mouse 모듈의 분석

NSC IR에서 mouse 모듈은 사용자 프로그램에 bus mouse 인터페이스를 제공하기 위한 모듈로서 실제적으로는 mouse의 움직임을 bus mouse 인터페이스를 이용해서 알려주는 역할만을 수행한다.

```
int init_module (void)
{
    //      bus_init();
    mouse_no = register_busmouse(&bus_ms);
    printk("\n ARUN: registration done : %d \n",mouse_no);

    return 0;
}
```

코드 1198. init_module() 함수의 정의

Mouse 모듈이 커널에 링크(link)될 때, 이 모듈이 bus mouse 디바이스 드라이버로 동작할 것이라는 것을 알려주기 위해서 register_busmouse() 함수를 호출한다. 넘겨주는 파라미터 값은 bus mouse를 나타내는 자료구조의 포인터이다. 이중에서 나중에 복귀 값으로 돌려받는 것은 자신이 등록한 mouse 디바이스의 minor 번호(MOUSE_MINOR = 12)이다. register_busmouse() 함수는 내부적으로 bus mouse를 나타내기 위한 구조체를 할당받고, 관련 필드를 초기화 한다음, miscellaneous 디바이스로 다시 등록하도록 한다. 일단 여기까지 진행했다면, 복귀 값은 0이 될 것이다.

```
void cleanup_module (void)
{
    printk("\n ANSCR cleanup \n");
    unregister_busmouse(mouse_no);
}
```

코드 1199. cleanup_module() 함수의 정의

cleanup_module()은 모듈이 커널과 unlink될 때 호출된다. 앞에서 등록한 bus mouse를 해제하기 위해서, unregister_busmouse() 함수를 호출한다. 마찬가지로, miscellaneous 디바이스로 등록된 것을 해제해 줄 것이다.

```
extern int register_busmouse(struct busmouse *ops);
extern int unregister_busmouse(int mousedev);
```

앞의 모듈을 적재하고 해제하는 부분에서 사용한 함수를 잠시 보도록 하자. register_busmouse()와 unregister_busmouse() 함수이다. 이 두 함수는 bus mouse로서 사용할 수 있도록 자료구조를 등록할 수 있도록 해주는 register_chrdev()/unregister_chrdev() 함수와 동일하다고 생각할 수 있겠다. 이 두 함수의 정의는 Linux 커널의 ~/drivers/char/busmouse.c 파일에 있다. 이들 함수에서 사용되는 자료구조로 디바이스 드라이버 프로그램마가 등록해야 할 것은 busmouse 구조체이다. 정의는 ~/include/linux/busmouse.h에 아래와 같이 되어 있다.

```
/* busmouse.h 파일에서 */
struct busmouse {
    int minor;                                /* Bus mouse 디바이스의 minor 번호 */
    const char *name;                           /* Bus mouse의 이름 */
    struct module *owner;                      /* Bus mouse 디바이스 드라이버 모듈의 owner */
    int (*open)(struct inode * inode, struct file * file); /* Open() 함수 */
    int (*release)(struct inode * inode, struct file * file); /* Release() 함수 */
    int init_button_state;                     /* 초기 bus mouse button의 상태 */
};

...
static struct busmouse bus_ms = {MOUSE_MINOR,"anscir",THIS_MODULE,NULL,NULL,0};
```

코드 1200. busmouse 구조체의 정의

busmouse 구조체는 bus mouse를 등록하기 위해서는 반드시 정의해야 한다. NSC IR의 bus mouse 구조체는 minor 번호로 MOUSE_MINOR(=12), 이름으로는 anscir, 모듈의 소유자(owner)로는 THIS_MODULE, 나머지 open()과 release() 함수로는 NULL을, 마지막으로 초기 마우스 button의 상태로 0을 주었다. 여기서 THIS_MODULE은 현재 커널에 삽입될 모듈을 나타내는 구조에 대한 포인터이다.

```
static int anscir_open (struct inode *inode, struct file *file)
{
    unsigned int minor = MINOR(file -> f_dentry -> d_inode -> i_rdev);
```

```

MOD_INC_USE_COUNT;
printk("\n ANSCIR : OPEN      \n");
if (open_count == 0)
{
    printk("A National IR Driver: %s\n", VERSION);
}
return 0;
}

```

코드 1201. anscir_open() 함수의 정의

실제적으로는 anscir_open() 함수는 bus mouse 드라이버로 등록하는데 아무런 영향도 주지 못한다. 즉, busmouse 구조체에 open() 함수에 연결되지 못했기에, 이 mouse 드라이버가 등록되어 있더라도, open() 파일 연산시에 호출받을 수는 없다. 하지만, 여기서는 코드에서 나온대로 옮긴것으로 설명만 덧붙이도록 하겠다.

먼저 넘겨받는 인자로부터 bus mouse 디바이스의 minor 번호를 얻기 위해서, file->f_dentry->d_inode->i_rdev 를 접근해서 MINOR() 매크로를 사용한다. 즉, i_rdev 필드에는 실제 디바이스에 대한 major와 minor번호를 가지기 때문에 이것이 가능하다는 것이다. 이 minor번호와 자신이 설정한 minor번호를 비교해 보아야만 하겠으나, 여기선 그렇게 하지 않고 있다. 즉, 당연히 같은 것이라고 생각한다. Open()이 되었으므로, 모듈의 사용 카운트 값을 증가시키고(MOD_INC_USE_COUNT), 만약 open된 회수가 0과 같다면, 처음 open하는 것이므로, 각종 정보를 쓰도록 한다.

```

static int anscir_release (struct inode *inode, struct file *file)
{
    printk(" \n ANSCR release \n");
    MOD_DEC_USE_COUNT;
    return 0;
}

```

코드 1202. anscir_release() 함수의 정의

anscir_release() 함수는 bus mouse 디바이스가 close() 될 때 수행되는 함수이다. 따라서, 앞에서 open() 시에 했던 것을 역으로 수행하면된다. 모듈의 사용 카운트만 현재로서는 감소시켜준다(MOD_DEC_USE_COUNT).

```

void bus_init()
{
    bus_ms.open = anscir_open;
    bus_ms.release = anscir_release;
}

ssize_t anscir_write (struct file *file, const char *buf, size_t count, loff_t *ppos)
{
    return 0;
}

ssize_t anscir_read (struct file *file, char *buf, size_t count, loff_t *ppos)
{
    unsigned int minor = MINOR(file -> f_dentry -> d_inode -> i_rdev);
    return -ENODEV;
}

static int anscir_ioctl (struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
    return 0;
}

```

```
static unsigned int anscir_poll (struct file *file, poll_table *wait)
{
    unsigned int minor = MINOR(file -> f_dentry -> d_inode -> i_rdev);

    return (0);
}
```

코드 1203. Bus mouse 디바이스 드라이버의 file operation 연산 벡터들의 정의

bus_init()는 실제로는 모듈의 초기화 시에 수행되도록 되어있다. 하지만, 현재는 comment로 처리되어 사용되지 않는다. 그외의 anscir_write(), anscir_read(), anscir_ioctl(), anscir_poll() 역시 별다른 일을 수행하지는 않는다.

```
unsigned char ansc_mv_mouse( int x, int y, int but)
{
    busmouse_add_movementbuttons(mouse_no,x,y,but);
}
```

코드 1204. ansc_mv_mouse() 함수의 정의

ansc_mv_mouse()함수가 실제적인 bus mouse의 인터페이스를 담당하는 역할을 수행한다. 즉, 하위의 IR 디바이스에 대한 프로토콜 디바이스 드라이버들이 mouse에 대한 event가 발생했을 때, 이 함수를 호출해서 응용프로그램에게 bus mouse event가 발생했다는 것을 알려주게 되는 것이다. 사용하는 방법은 간단히 X, Y축 좌표와 button에 대한 event이다. 이 함수는 내부적으로는 bus mouse event를 알려주기 위해서 busmouse_add_movementbuttons() 함수를 사용한다. 첫번째 인자로 주어진 것은 등록한 bus mouse의 minor 번호이다.

18.2.2. Linux Kernel Bus Mouse Interface의 분석

앞에서는 National Semiconductor에서 사용하는 IR bus mouse 모듈에 대해서 간략하게 살펴보았다. 하지만, Linux 커널의 bus mouse interface를 모른다면, 앞에서 설명한 것 만으로는 부족하다는 느낌을 받을 수 있을 것이다. 따라서, 이하에서는 Linux 커널의 bus mouse interface의 핵심이 되는 부분을 보도록 하겠다.

```
struct busmouse_data {
    struct miscdevice miscdev;          /* Miscellaneous 디바이스 드라이버 등록 */
    struct busmouse *ops;                /* Bus mouse 연산 벡터 및 드라이버 모듈 포인터 */
    spinlock_t lock;                   /* Sync */
    wait_queue_head_t wait;             /* Event wait queue */
    struct fasync_struct *fasyncptr;    /* File asynchronous notification 포인터 */
    char active;                      /* Bus status */
    char buttons;                     /* Button status */
    char ready;                       /* Bus mouse가 준비되었는가? */
    int dxpos;                        /* X position */
    int dypos;                        /* Y position */
};
```

코드 1205. busmouse_data 구조체의 정의

busmouse_data 구조체는 하나의 busmouse를 나타내기 위해서 사용된다. 앞에서 본 busmouse 구조체는 busmouse_data 구조체의 한 entry를 차지하게 된다. 또한, busmouse_data 구조체로 정의된 bus mouse는 전체 bus mouse에 대한 정보를 가지고 있는 전역 변수인 busmouse_data[] 배열의 한 구성원이 된다.

18.2.2.1. Bus Mouse의 등록과 해제

Bus mouse를 등록하고 해제하는 함수는 register_busmouse()와 unregister_busmouse() 함수이다. 이 함수들은 주로, 앞에서 설명한 busmouse_data 구조체를 다루는 것이 핵심이며, misc_register()와 misc_deregister() 함수를 호출해서 miscellaneous 디바이스 드라이버로 등록/해제를 해 준다.

```

int register_busmouse(struct busmouse *ops)
{
    unsigned int msedev = MINOR_TO_MOUSE(ops->minor);
    struct busmouse_data *mse;
    int ret;

    if (msedev >= NR_MICE) {
        printk(KERN_ERR "busmouse: trying to allocate mouse on minor %d\n",
               ops->minor);
        return -EINVAL;
    }
    mse = kmalloc(sizeof(*mse), GFP_KERNEL);
    if (!mse)
        return -ENOMEM;
    down(&mouse_sem);
    if (busmouse_data[msedev])
    {
        up(&mouse_sem);
        kfree(mse);
        return -EBUSY;
    }
    memset(mse, 0, sizeof(*mse));
    mse->miscdev.minor = ops->minor;
    mse->miscdev.name = ops->name;
    mse->miscdev.fops = &busmouse_fops;
    mse->ops = ops;
    mse->lock = (spinlock_t)SPIN_LOCK_UNLOCKED;
    init_waitqueue_head(&mse->wait);
    busmouse_data[msedev] = mse;
    ret = misc_register(&mse->miscdev);
    if (!ret)
        ret = msedev;
    up(&mouse_sem);
    return ret;
}

```

코드 1206. register_busmouse() 함수의 정의

인자로 busmouse 구조체를 넘겨받아, 설정된 minor 번호를 먼저 구한다. 이것은 busmouse_data[] 배열에 대한 index로 사용될 것이므로 msedev 변수에 저장한다. 만약 이 값이 정해진 mouse 개수의 수를 벗어나게 되면(NR_MICE = 15) 오류로 처리하고 -EINVAL을 돌려준다. 이전 busmouse_data 구조체를 하나 할당 받아 이것을 mse가 가르키도록 만든다. 할당받을 수 없다면, -ENOMEM(Error No Memory)을 돌려준다. 전역 변수인 busmouse_data[] 배열에 접근해야 하므로, critical section보호를 위한 세마포어를 설정한다(down(&mouse_sem)). 이전 msedev를 index로 사용해서 busmouse_data[] 배열의 해당하는 자리가 비어있는지를 확인한다. 만약 비어있지 않다면, 세마포어를 증가시키고(up(&mouse_sem)), 할당한 메모리를 해제한 후, -EBUSY(Error Busy)를 돌려준다.

이상에서 오류가 없었다면, 할당받은 메모리를 0으로 초기화하고(memset()), busmouse_data 구조체의 각 필드들을 채워주도록 한다. 이때, bus mouse에 대한 기본 연산 벡터는 busmouse_fops로 설정됨을 기억하기 바란다. 나중에 파일 연산에 대한 것을 사용자 프로그램이 호출하게 되면, busmouse_fops가 호출되어질 것이다. 또한, 각각의 ops에 들어있는 특정 bus mouse의 파일 연산 벡터들도 busmouse_fops 연산 벡터의 수행내에서 호출될 것이다. Lock을 초기화 하는 것은 SPIN_LOCK_UNLOCKED 변수를 넣어서 해준다. Wait queue는 init_waitqueue_head()를 불러서 초기화를 한다. 이전 초기화가 끝난, 하나의 bus mouse 구조체를 busmouse_data[] 배열에 넣고, 이를 misc_register() 함수를 호출해서 miscellaneous 드라이버로 등록한다. 이때 오류가 없다면, busmouse_data[] 배열의 인덱스로 사용된 minor 번호를

돌려주게 되며, 그렇지 않다면 오류 번호가 복귀 값으로 주어진다. 복귀전에 설정된 세마포어도 풀어주어야 할 것이다.

```
int unregister_busmouse(int mousedev)
{
    int err = -EINVAL;

    if (mousedown < 0)
        return 0;
    if (mousedown >= NR_MICE) {
        printk(KERN_ERR "busmouse: trying to free mouse on"
              " mousedev %d\n", mousedev);
        return -EINVAL;
    }
    down(&mouse_sem);
    if (!busmouse_data[mousedev]) {
        printk(KERN_WARNING "busmouse: trying to free free mouse"
              " on mousedev %d\n", mousedev);
        goto fail;
    }
    if (busmouse_data[mousedev]->active) {
        printk(KERN_ERR "busmouse: trying to free active mouse"
              " on mousedev %d\n", mousedev);
        goto fail;
    }
    err = misc_deregister(&busmouse_data[mousedev]->miscdev);
    kfree(busmouse_data[mousedev]);
    busmouse_data[mousedev] = NULL;
fail:
    up(&mouse_sem);
    return err;
}
```

코드 1207. unregister_busmouse() 함수의 정의

unregister_busmouse() 함수는 register_busmouse() 함수의 역을 수행한다. 넘겨받는 것은 해제할 bus mouse를 나타내는 minor번호이다. 이 번호를 검사해서 오류가 있다면 -EINVAL을 돌려준다. 세마포를 감소시켜주고, busmouse_data[] 배열의 mousedev 인덱스의 엔트리가 비어있는지 확인한다. 만약 비어있다면, 없는 것을 해제하려고 했으므로, fail로 제어를 옮긴다. 또한 만약 해당 인덱스의 bus mouse가 현재 active 상태라고 한다면, 삭제할 수 없도록 만들어주어야 할 것이다. 이 경우에도 fail로 제어를 옮긴다. misc_deregister() 함수는 앞에서 miscellaneous 디바이스 드라이버로 등록했던 것을 해제하기 위해서 호출하며, busmouse 구조체를 위해서 할당된 메모리를 해제한다. busmouse_data[] 배열의 해당 엔트리는 이제 NULL로 두면 된다. fail 이하에서는 세마포어를 증가 시키고, 설정된 err코드를 돌려주기만 하면된다.

18.2.2.2. Bus Mouse의 Event 알리기

이제 부터 보는 것은 bus mouse 디바이스 드라이버를 만들었을 때, 이것을 응용 프로그램에 알려주기 위해서 사용하는 함수들이다. 여기에 속하는 것으로는 busmouse_add_movement(), busmouse_add_buttons(), busmouse_add_movementbuttons() 함수들이다. 소스를 보면 알 수 있지만, 실제로는 busmouse_add_movementbuttons() 하나의 함수가 주된 역할을 수행한다.

```
/** 
 *      busmouse_add_movement - notification of a change of mouse position
 *      @mousedown: mouse number
 *      @dx: delta X movement
 *      @dy: delta Y movement
```

```

/*
 * Updates the mouse position. The mousedev parameter is the value
 * returned from register_busmouse. The movement information is
 * updated, and a waiting user thread is woken.
 */
void busmouse_add_movement(int mousedev, int dx, int dy)
{
    struct busmouse_data *mse = busmouse_data[mousedev];

    busmouse_add_movementbuttons(mousedev, dx, dy, mse->buttons);
}

```

코드 1208. busmouse_add_movement() 함수의 정의

busmouse_add_movement() 함수는 단순히 mouse의 이동만을 알리기 위해서 사용한다. 먼저 busmouse_data[] 구조체로 부터 해당 bus mouse의 busmouse_data 구조체를 얻고, 이곳에서 정의된 buttons와 넘겨받은 인자들을 가지고, busmouse_add_movementbuttons() 함수를 호출한다.

```

/***
 * busmouse_add_buttons - notification of a change of button state
 * @mousedown: mouse number
 * @clear: mask of buttons to clear
 * @eor: mask of buttons to change
 *
 * Updates the button state. The mousedev parameter is the value
 * returned from register_busmouse. The buttons are updated by:
 *     new_state = (old_state & ~clear) ^ eor
 * A waiting user thread is woken up.
 */
void busmouse_add_buttons(int mousedev, int clear, int eor)
{
    struct busmouse_data *mse = busmouse_data[mousedev];

    busmouse_add_movementbuttons(mousedev, 0, 0, (mse->buttons & ~clear) ^ eor);
}

```

코드 1209. busmouse_add_buttons() 함수의 정의

busmouse_add_buttons() 함수는 button의 상태를 지우거나 toggle시켜준다. 이 함수가 넘겨받는 인자는 지워야 하는 button에 대한 것과(clear), toggle해야 할 button을 나타내는(eor) 것이며, 해당 bus mouse의 minor 번호이다. Button의 상태만 바꾸기에 마우스의 위치는 바뀌지 않는다. busmouse_add_movementbuttons() 함수의 buttons 인자에만 새로운 값을 설정해서 넘겨준다.

```

/***
 * busmouse_add_movement - notification of a change of mouse position
 * @mousedown: mouse number
 * @dx: delta X movement
 * @dy: delta Y movement
 * @buttons: new button state
 *
 * Updates the mouse position and button information. The mousedev
 * parameter is the value returned from register_busmouse. The
 * movement information is updated, and the new button state is
 * saved. A waiting user thread is woken.
 */
void busmouse_add_movementbuttons(int mousedev, int dx, int dy, int buttons)
{

```

```

struct busmouse_data *mse = busmouse_data[mousedev];
int changed;

spin_lock(&mse->lock);
changed = (dx != 0 || dy != 0 || mse->buttons != buttons);
if (changed) {
    add_mouse_randomness((buttons << 16) + (dy << 8) + dx);
    mse->buttons = buttons;
    mse->dxpos += dx;
    mse->dypos += dy;
    mse->ready = 1;
    /*
     * keep dx/dy reasonable, but still able to track when X (or
     * whatever) must page or is busy (i.e. long waits between
     * reads)
     */
    if (mse->dxpos < -2048)
        mse->dxpos = -2048;
    if (mse->dxpos > 2048)
        mse->dxpos = 2048;
    if (mse->dypos < -2048)
        mse->dypos = -2048;
    if (mse->dypos > 2048)
        mse->dypos = 2048;
}
spin_unlock(&mse->lock);
if (changed) {
    wake_up(&mse->wait);
    kill_fasync(&mse->fasyncptr, SIGIO, POLL_IN);
}
}

```

코드 1210. busmouse_add_movementbuttons() 함수의 정의

하나의 bus mouse에 대해서 여러번에 걸쳐서 이 함수가 호출될 가능성이 있는데, 각각의 busmouse_data 구조체에 있는 spin lock을 설정하도록 한다(spin_lock()). 이전 mouse에 대해서 변경 사항이 있는지를 검사하기 위해서 넘겨받은 인자들을 검사해 보도록 한다. 만약 mouse가 움직였거나, 버튼의 상황에 변경이 있었다면, changed 변수를 설정한다. 이하는 이 changed 변수가 설정된 경우에만 실행될 것이다. add_mouse_randomness() 함수는 시스템의 random 함수 사용에 영향을 미치도록 만들어주는 것으로, 내부적으로는 timer의 randomness를 증가시키도록 하는 역할을 한다.

Bus mouse의 button 상태를 변경하고, X축과 Y축의 위치를 갱신한다. 그리고나서, 이 bus mouse에서 데이터를 읽어갈 수 있다는 것을 알려주기 위해서 ready를 1로 설정한다. 나머지는 최대로 설정될 수 있는 bus mouse의 X와 Y좌표를 한계값으로 맞춰주는 일이다. 이것을 마치면, lock을 해제(spin_unlock())하도록 한다. changed가 설정된 경우에는 bus mouse의 대기 큐에 있을지도 모르는 프로세스들을 깨워주기 위해서 wake_up() 함수를 호출한다. 또한 asynchronous I/O를 기대하고 있는 것임을 수 있기에 kill_fasync() 함수를 호출하도록 한다. 시그널로는 SIGIO를 준다.

18.2.2.3. Bus Mouse의 파일 연산 벡터

응용 프로그램들은 어차피 다른 디바이스에 대한 접근과 마찬가지로, 마치 디바이스를 파일과 같이 접근한다. 따라서, bus mouse에 대한 접근도 파일과 같은 형식을 취하게 되므로, 파일 연산 벡터는 응용 프로그램 인터페이스(API)라고 볼 수 있다. 아래와 같은 정의를 가진다.

```

struct file_operations busmouse_fops=
{
    owner:          THIS_MODULE,
    read:           busmouse_read,
}

```

```

write:          busmouse_write,
poll:          busmouse_poll,
open:          busmouse_open,
release:       busmouse_release,
fasync:        busmouse_fasync,
};

```

코드 1211. busmouse_fops 파일 연산 벡터의 정의

Bus mouse 디바이스 드라이버는 파일 연산 중에서 read(), write(), poll(), open(), release(), fasync()만을 제공한다. 그러나, 실제적으로는 write는 일어날 일이 없을 것이다. 각각의 함수들에 대해서 차례로 보도록 하자.

```

static int busmouse_open(struct inode *inode, struct file *file)
{
    struct busmouse_data *mse;
    unsigned int mousedev;
    int ret;

    mousedev = DEV_TO_MOUSE(inode->i_rdev);
    if (mousedown >= NR_MICE)
        return -EINVAL;
    down(&mouse_sem);
    mse = busmouse_data[mousedev];
    ret = -ENODEV;
    if (!mse || !mse->ops) /* shouldn't happen, but... */
        goto end;
    if (mse->ops->owner && !try_inc_mod_count(mse->ops->owner))
        goto end;
    ret = 0;
    if (mse->ops->open) {
        ret = mse->ops->open(inode, file);
        if (ret && mse->ops->owner)
            __MOD_DEC_USE_COUNT(mse->ops->owner);
    }
    if (ret)
        goto end;
    file->private_data = mse;
    if (mse->active++)
        goto end;
    spin_lock_irq(&mse->lock);
    mse->ready = 0;
    mse->dxpos = 0;
    mse->dypos = 0;
    mse->buttons = mse->ops->init_button_state;
    spin_unlock_irq(&mse->lock);
end:
    up(&mouse_sem);
    return ret;
}

```

코드 1212. busmouse_open() 함수의 정의

busmouse_open() 함수는 bus mouse를 사용하려고 하면, 반드시 호출해야 하는 함수이다. 이 함수에서 중요한 것은 등록된 디바이스 노드로 부터 bus mouse에 대한 minor번호를 추출하는 과정과, 파일 구조체의 private 필드에 해당 bus mouse 디바이스 드라이버의 busmouse_data 구조체 포인터를 넣는 것이다. 나중에 다른 파일 함수를 사용하게 될 때 이 부분에 지정된 자료구조를 이용해서 처리할

것이기 때문이다. 먼저 inode의 i_rdev 필드로부터 디바이스의 minor번호를 추출해서 이 값이 옳은가를 확인한다. 그리고나서, 전역 변수인 busmouse_data[] 배열에 접근하기 위해서 세마포어를 감소시키고, 해당 bus mouse의 busmouse_data 구조체를 가져와서 mse가 가르키도록 만든다. 만약 minor 번호에 해당하는 busmouse_data를 찾지 못하거나, 혹은 연산 벡터가 정의되지 않은 경우에는 바로 end로 제어를 옮긴다. 연산 벡터 중에서 open()에 해당하는 것이 정의되어 있다면, 이를 호출한다. 호출의 결과에 오류가 있고, bus mouse로 등록된 모듈의 owner의 필드가 존재할 때는 open()에서 발생했을지도 모를 모듈 사용카운터의 증가를 이곳에서 낮춰준다.(__MOD_DEC_USE_COUNT()). 당현히 제어는 end로 바로 갈 것이다.

여기까지 오류가 없었다면, 이전 file 구조체의 private_data 필드에 해당 busmouse_data 구조체를 연결시켜주고, active 필드를 증가 시켜준다. 만약 처음 이 함수가 호출되었다면, active 필드는 0으로 설정되어 있을 것이므로, 바로 end로 제어를 옮겨서 세마포어를 증가시켜주고 복귀 값을 넘길 것이다. 그렇지 않다면, 벌써 active한 상태라는 이야기가 되며, 즉, 하나 이상의 프로그램에서 이 bus mouse를 사용하고 있다는 이야기가 된다. 따라서, bus mouse에 대한 동기화의 목적으로 spin lock을 해당 bus mouse에 설정하고, ready와 X/Y 좌표를 전부 0으로 둔다. button의 상태는 해당 bus mouse 디바이스 드라이버에서 가정하고 있는 초기상태 값(init_button_state)로 둔 후, lock을 해제한다. 나머지는 세마포어를 풀고 복귀 코드를 넘겨주는 과정이다.

```
static ssize_t busmouse_read(struct file *file, char *buffer, size_t count, loff_t *ppos)
{
    struct busmouse_data *mse = (struct busmouse_data *)file->private_data;
    DECLARE_WAITQUEUE(wait, current);
    int dxpos, dypos, buttons;

    if (count < 3)
        return -EINVAL;
    spin_lock_irq(&mse->lock);
    if (!mse->ready) {
#define BROKEN_MOUSE
        spin_unlock_irq(&mse->lock);
        return -EAGAIN;
#undef BROKEN_MOUSE
    }
    #else
        if (file->f_flags & O_NONBLOCK) {
            spin_unlock_irq(&mse->lock);
            return -EAGAIN;
        }
        add_wait_queue(&mse->wait, &wait);
    #endif
repeat:
    set_current_state(TASK_INTERRUPTIBLE);
    if (!mse->ready && !signal_pending(current)) {
        spin_unlock_irq(&mse->lock);
        schedule();
        spin_lock_irq(&mse->lock);
        goto repeat;
    }
    current->state = TASK_RUNNING;
    remove_wait_queue(&mse->wait, &wait);
    if (signal_pending(current)) {
        spin_unlock_irq(&mse->lock);
        return -ERESTARTSYS;
    }
#endif
}
```

코드 1213. busmouse_read() 함수의 정의

`busmouse_read()` 함수는 응용 프로그램에서 제일 자주 호출되는 함수일 것이다. 먼저 앞에서 `open()`시에 설정한 file 구조체로부터 `private_data`에서 `busmouse_data`의 포인터를 얻는다. 현재 프로세스의 `wait queue`에 들어가기 위한 초기화를 해주고(`DECLARE_WAITQUEUE()`), 읽어야 할 데이터의 크기가 3이하인 경우에는 `-EINVAL`을 돌려준다. 즉, X와 Y좌표, button 상태를 합쳐서 적어도 3이상의 데이터를 읽어야지만 제대로 bus mouse를 접근하는 것이 될 것이다. 이전 해당 bus mouse에 대해 접근해야 하므로, `lock`을 설정한다(`spin_lock_irq()`). 데이터가 `ready`가 되지 않았다면, `if(){}`절을 수행한다. 만약 `BROKEN_MOUSE`가 정의된 경우에는 앞에서 설정한 `lock`을 해제하고, `-EAGAIN`을 에러코드로 돌려준다. 이것은 bus mouse 모듈의 작성자를 위한 것이기에 우리에게는 관심의 대상이 아니므로 무시하도록 한다. 이전 file구조체에 `O_NONBLOCK` 모드가 설정되었는지를 확인한다. 이 경우에는 바로 응용 프로그램에 복귀를 해야하므로(non-blocking I/O) 설정한 `lock`을 해제하고 `-EAGAIN`을 오류코드로 준다. 그렇지 않다면, 응용 프로그램에서 bus mouse로 부터 읽을 데이터가 있을 때까지 대기하겠다는 의미가 되므로, `wait queue`에 현재 프로세스를 넣어주도록 한다(`add_wait_queue()`). 요청한 프로세스의 상태는 `TASK_INTERRUPTIBLE`이 될 것이며(`set_current_state()`), 다시 한번 `ready`나 `pending`된 시그널이 있는지를 확인한다. 없다면 `lock`을 해제하고 `schedule()` 함수를 호출해서 다른 프로세스들이 스케줄링되도록 해준다. 나중에 이 프로세스가 스케줄링되면 `schedule()` 함수 이하에서 수행을 제게할 것이다. 제일 먼저 하는 일은 bus mouse에 대한 `lock`을 얻는 것이다. 그리고나서 다시 `ready`나 `pending`된 시그널이 있는지를 반복적으로 확인하기 위해서 `repeat`로 제어를 옮긴다. 뭔가 데이터가 읽어들일 준비가 되었거나 시그널을 받았다면, 프로세스의 상태를 다시 `TASK_RUNNING`로 바꾸고, `wait queue`에서 제거(`remove_wait_queue()`)한 후, 시그널부터 확인해 본다. `Pending`된 시그널이 있을 시에는 설정한 `lock`을 해제하고, 곧바로 복귀한다. 이때는 데이터를 읽는 도중에 중단된 것이므로, `-ERESTARTSYS` 오류를 돌려준다. 즉, 시스템에서 시스템 콜의 수행이 거부당했다는 것을 알려주는 것이다.

```

dxpos = mse->dxpos;
dypos = mse->dypos;
buttons = mse->buttons;
if (dxpos < -127)
    dxpos = - 127;
if (dxpos > 127)
    dxpos = 127;
if (dypos < -127)
    dypos = - 127;
if (dypos > 127)
    dypos = 127;
mse->dxpos -= dxpos;
mse->dypos -= dypos;
/* This is something that many drivers have apparently
 * forgotten... If the X and Y positions still contain
 * information, we still have some info ready for the
 * user program...
 */
mse->ready = mse->dxpos || mse->dypos;
spin_unlock_irq(&mse->lock);
/* Write out data to the user. Format is:
 *   byte 0 - identifier (0x80) and (inverted) mouse buttons
 *   byte 1 - X delta position +/- 127
 *   byte 2 - Y delta position +/- 127
 */
if (put_user((char)buttons | 128, buffer) ||
    put_user((char)dxpos, buffer + 1) ||
    put_user((char)dypos, buffer + 2))
    return -EFAULT;
if (count > 3 && clear_user(buffer + 3, count - 3))
    return -EFAULT;
file->f_dentry->d_inode->i_atime = CURRENT_TIME;
return count;

```

```
{}
```

코드 1214. busmouse_read() 함수의 분석(계속)

이전 읽을 데이터를 모으도록 한다. X, Y, button 데이터를 bus mouse로부터 읽어서, 이렇게 읽은 데이터가 특정 범위(± 127)에 있도록 만들고, 이를 다시 원래의 bus mouse의 X, Y좌표로 둔다. Button에 대해서는 달리 해주는 일이 없다. 이렇게 해주는 것은, bus mouse가 너무 급격하게 움직이는 것을 막아주는 일을 한다. 그리고나서, X/Y 축에 대한 위치 변경이 일어났다는 것을 다시 bus mouse에 알려주기 위해서 ready를 설정하도록 한다. 더이상 전역 변수에 접근할 일이 없으므로 여기서 lock을 해제하고(spin_unlock()), 사용자 영역으로 데이터를 복사한다. 이때 button은 identifier의 역할을 하는 0x80과 OR시켜서 주게되며, 나머지 X/Y 좌표는 그대로 복사한다(put_user()). 전체 복사가 일어나는 데이터의 크기는 3 bytes로, 복사중에 오류가 있었다면 -EFAULT(Error Fault)를 돌려준다. 만약 3 bytes보다 큰 데이터를 읽기를 요구한다면, 그 이후의 복사 영역은 전부 0으로 지우도록 한다(clear_user()). 이때도 마찬가지로 오류가 있었다면, -EFAULT를 돌려준다. 파일에 대한 접근시간도 조정하도록 한다. 이를 위해서 CURRENT_TIME(xtime.tv_sec)이 사용된다. 복귀 코드는 읽기를 요청한 데이터의 크기인 count이다.

```
static ssize_t busmouse_write(struct file *file, const char *buffer, size_t count, loff_t *ppos)
{
    return -EINVAL;
}
```

코드 1215. busmouse_write() 함수의 정의

busmouse_write()함수는 아무일도 하지않고, 호출되면 -EINVAL을 돌려준다. 즉, bus mouse에 대한 write는 아무런 의미가 없다는 것이다.

```
static int busmouse_release(struct inode *inode, struct file *file)
{
    struct busmouse_data *mse = (struct busmouse_data *)file->private_data;
    int ret = 0;

    lock_kernel();
    busmouse_fasync(-1, file, 0);
    if (--mse->active == 0) {
        if (mse->ops->release)
            ret = mse->ops->release(inode, file);
        if (mse->ops->owner)
            __MOD_DEC_USE_COUNT(mse->ops->owner);
        mse->ready = 0;
    }
    unlock_kernel();
    return ret;
}
```

코드 1216. busmouse_release() 함수의 정의

busmouse_release() 함수는 bus mouse에 대한 close() 파일 연산과 같다. 먼저 해당 bus mouse의 busmouse_data 구조체를 file 구조체의 private_data로부터 얻는다. 커널에 lock을 설정(lock_kernel)하는 것은 다른 프로세스들이 커널 모드로 진입하지 못하도록 만든다. Asynchronous I/O를 호출한 프로세스들은 다시 시그널을 받아서, 이젠 해당 bus mouse를 사용할 수 없다는 것을 알 수 있도록 busmouse_fasync() 함수를 호출한다. active값을 감소시켜서, 이 값이 0과 같다면 더이상 이 bus mouse 디바이스를 open한 프로세스들이 없다는 말이 되므로, 등록된 bus mouse 드라이버의 release() 함수를 호출하도록 한다. 만약 모듈의 owner 필드가 정의되어 있다면, 모듈 사용 카운터도 감소(__MOD_DEC_USE_COUNT())시킨다. 이젠 더이상 처리할 데이터가 없다는 것을 알려주기 위해서 ready는 0으로 두게된다. 만약 active한 것이 있다면, open한 프로세스가 있다는 이야기이므로, 앞에서와

같은 일을 일어나지 않아야 한다. 단순히 active 값을 감소시키는 것으로 충분하다. 복귀 전에는 앞에서 설정한 커널 lock을 해제해 주어야 할 것이다(unlock_kernel()).

```
static int busmouse_fasync(int fd, struct file *filp, int on)
{
    struct busmouse_data *mse = (struct busmouse_data *)filp->private_data;
    int retval;

    retval = fasync_helper(fd, filp, on, &mse->fasyncptr);
    if (retval < 0)
        return retval;
    return 0;
}
```

코드 1217. busmouse_fasync() 함수의 정의

busmouse_fasync() 함수는 asynchronous I/O의 일환으로 호출된다. 여기서는 단순히 fasync_helper() 함수를 호출해서 대부분의 일을 처리하도록 한다. 복귀값 역시 호출의 결과에 오류가 있다면, 그대로 넘겨준다. 그렇지 않다면 0을 돌려준다. fasync_helper() 함수는 on 변수에 넘겨주는 값에 따라서, asynchronous I/O를 위해서 필요한 커널 자료구조를 제거하거나, 혹은 설정하는 역할을 하는 함수이다. 여기서는 사용자 프로그램이 호출했을 경우에는 asynchronous I/O를 설정내지는 해제하는 역할을 해줄 것이며, busmouse_release()와 같은 곳에서 호출된다면 설정했던 것을 해제하는 역할만 한다.

```
static unsigned int busmouse_poll(struct file *file, poll_table *wait)
{
    struct busmouse_data *mse = (struct busmouse_data *)file->private_data;

    poll_wait(file, &mse->wait, wait);
    if (mse->ready)
        return POLLIN | POLLRDNORM;
    return 0;
}
```

코드 1218. busmouse_poll() 함수의 정의

busmouse_poll()은 bus mouse에 대한 select()나 poll() 함수의 호출 결과로 수행되는 함수이다. 따라서, 여기서는 읽을 수 있는 데이터가 준비되었는가만을 판단해서 POLLIN | POLLRDNORM 값을 돌려주고 있다. 데이터가 준비되었다는 것은 ready필드를 참조하면 될 것이다. 역시 file 구조체의 private_data를 통해서 해당 busmouse_data 구조체를 접근하는 방법을 사용한다. poll_wait() 함수는 select()/poll()을 위한 poll_wait()는 poll table의 wait queue에 엔트리를 추가하는 일을 한다⁴⁷⁹.

⁴⁷⁹ 이 부분은 Select()와 Poll()을 분석하는 곳에서 찾을 수 있을 것이다. 여기서는 간략히 사용한다는 것만을 알고 넘어가도록 한다.

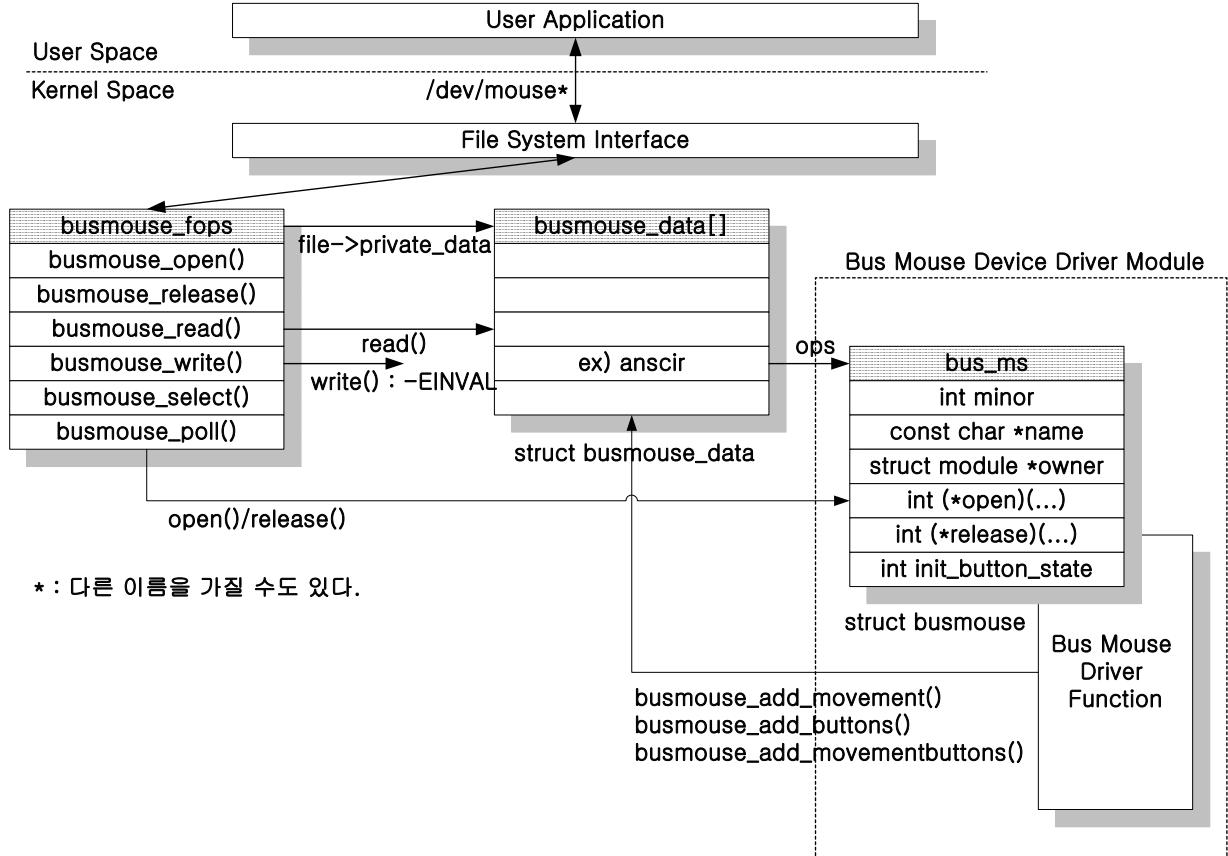


그림 136. Linux의 Bus Mouse Driver의 구성

[그림 136]는 앞에서 본 Linux의 bus mouse 디바이스 드라이버를 정리한 것이다. 그림에서 보듯이 이미 Linux에서는 bus mouse를 위한 구조를 커널내에 담고 있으며, 우리가 작성해야 하는 부분은 그림의 우측에 있는 사각형 내에 있는 부분들이다. 만약 open()이나 release()시에 해줄 일이 없다면, 극히 간단할 것이며, 실제로 앞에서 본 NSC IR의 bus mouse 모듈과 같이 `busmouse_add_XXX()` 함수만으로도 사용자 프로그램과 인터페이스 할 수 있다. 이것으로 Linux 커널에서의 bus mouse 드라이버에 대한 구조를 보는 것을 마치도록 하겠다.

19. Linux I2C Bus System

이번 장에서는 PC 및 일반 가전 제품에서 많이 사용되는 I2C(Inter IC) Bus에 대해서 Linux에서는 어떻게 구현하고 있는지를 살펴보도록 하겠다. I2C의 사용 예를 보면, 일반 PC의 경우 CPU의 온도를 측정하는 곳에서 찾을 수 있으며, Settop-box와 같은 곳에서는 TV tuner 및 NTSC/PAL decoder와 기타 장치들에 대해서 제어를 하는 곳에서 볼 수 있다.

I2C는 저렴하며, 간단한 시리얼(serial) 데이터 통신을 제공하는 방법으로 Philips사에서 개발한 것이다. 현재까지 나와 있는 I2C Bus에 대한 스펙(specification)은 2.0 버전으로 여기서는 이것을 기준으로 설명하도록 하겠다. 먼저, I2C를 구현하는데 있어서 Linux에서 제공하는 메커니즘에 대해서 설명하고, 이후에는 Geode SC1200에서 구현한 I2C의 코드를 살펴보는 것으로 하겠다.

19.1. I2C Bus for Linux의 분석

Linux에서는 I2C를 구현하기 위해서 필수적으로 i2c-core.c와 i2c-dev.c 파일을 두고 있다. 이 두개의 파일은 I2C bus를 관리하고, 커널과 I2C bus상의 디바이스를 연결하는 고리와 같은 역할을 하는 것으로 핵심적인 자료구조 및 인터페이스를 제공한다.

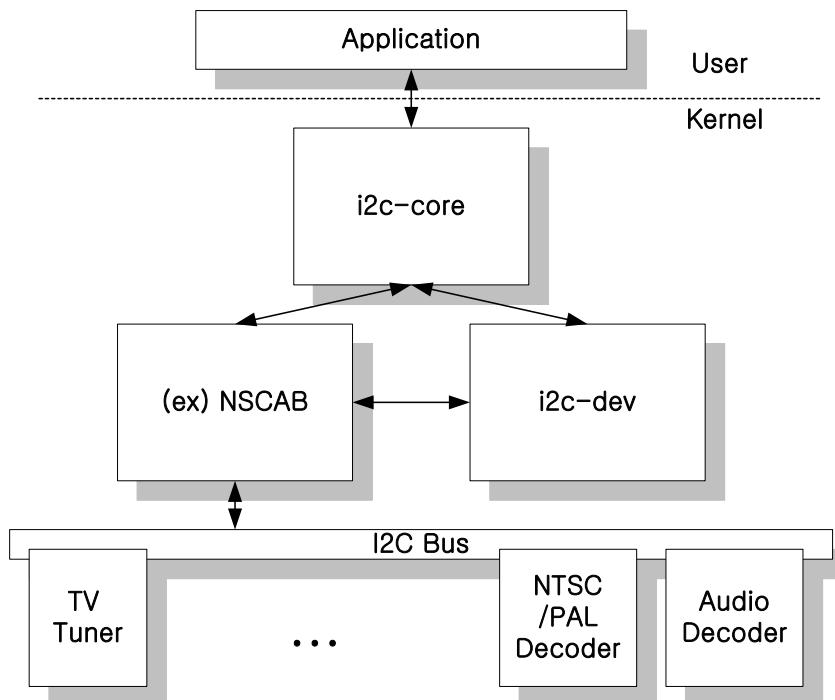


그림 137. I2C Bus의 구성

[그림 137]에서 보듯이 응용 프로그램의 요청을 받은 i2c-core는 I2C Bus에 맞물린 디바이스를 관리하기 위한 i2c-dev와 실제 시스템 보드상에 존재하는 I2C Bus controller에 접근해서 I2C Bus를 통해 각각의 디바이스들을 제어하게 된다. 이때, 실제적인 데이터는 다른 시스템의 bus(혹은 외부로 cable을 통해서)를 통해서 전달될 수 있으며, 단지 제어를 위한 정보만을 I2C Bus를 통해서 전달한다. 즉, I2C Bus 자체가 그렇게 대용량의 데이터를 주고 받기에는 부족하며, 단지 I2C Bus상에 존재하는 디바이스들에 대한 제어 신호만을 보내는 것에 유용하기 때문이다. 예를 들어 실질적인 I2C Bus의 관리는 National Semiconductor의 Geode SC1200에서와 같이 NSCAB(NSC Access Bus)가 담당하고, 나머지 해당 디바이스에 대한 드라이버들은 I2C core 상위에 존재해 디바이스를 제어할 수 있게 된다. Bus와 관련된 디바이스 드라이버들은 대부분이 위와 같은 방식을 취하고 있으며, 실제 adapter의 디바이스 드라이버 상위에 전체

Bus를 관리하는 버스 드라이버가 올라오고, 그 위에 client 디바이스 드라이버들이 위치하며, 이러한 client 디바이스 드라이버들이 각각 응용 프로그램과 관련된 인터페이스를 제공한다.⁴⁸⁰

I2C Bus를 이해하기에 앞서서 이하에서 나오게 될 용어들에 대해서 조금 자세히 알아보고 가기로 하자. 다음과 같은 것들이 있다.

- Bus : 각종 디바이스들이 놓이게 되는 시스템 상의 데이터 이동 경로를 이야기 하는 것으로, 이것을 구동 시키기 위해서는 알맞은 chip이 필요하며, 이에 해당하는 드라이버가 필요하다. 또한 이 드라이버에는 버스에 접근하기 위한 방법들이 포함된다.
- Adapter : 앞에서 버스를 구동하기 위한 chip set이 있다고 했는데, 이러한 chip set을 우린 adapter라고 보기로 하겠다. 따라서, 이 chip set은 bus의 구동과 데이터의 전달을 담당한다. 또한 이 adapter는 자신을 위한 디바이스 드라이버를 가지며, 이 드라이버가 bus에 접근해서 데이터를 얻기 위한 특별한 방식을 가지고 있다. 이것을 알고리즘(algorithm)이라고 한다.
- Algorithm : 특정 adapter에서 bus 상에 놓인 디바이스와 데이터를 주고 받기 위해서 사용하는 메커니즘이다. 즉, 시작과 주소, 데이터 등의 포맷 및 기타 제반 사항을 포함해 전달되어지는 데이터를 인식하는 방법이라고 보면 될 것이다.
- Client : Bus 상에서 데이터를 주고 받는 디바이스를 말한다. 즉, 데이터의 source와 sink가 될 수 있는 디바이스이다. 물론 이를 위한 디바이스 드라이버가 필요하며, bus를 이용하기에 adapter 디바이스 드라이버가 제공하는 방법으로 데이터를 주고 받을 것이다. 하지만, 데이터에 대한 해석은 디바이스마다 달라질 수 있다.
- Driver : 실제 제어할 client마다, 하나씩 존재하는 것으로 driver 모듈이라고 보는 것이 옳바를 것이다. 즉, client를 제어하는 디바이스 드라이버이다.

위와 같은 개념으로 보았을 때, ~/include/linux/i2c.h 파일에는 다음과 같은 자료 구조들이 정의되어 있다. 즉, 실제 물리적인 client 디바이스를 다루게 될 i2c_driver 구조체, I2C Bus상에 놓인 client 디바이스를 나타내는 i2c_client 구조체, I2C Bus를 제어하기 위한 i2c_adapter 구조체 및 이 것이 사용하는 Bus 제어 방법인 i2c_algorithm 구조체 등이 있다.

```
/*
 * A driver is capable of handling one or more physical devices present on
 * I2C adapters. This information is used to inform the driver of adapter
 * events.
 */
struct i2c_driver {
    char name[32];                      /* I2C driver의 이름 */
    int id;                             /* I2C driver의 ID */
    unsigned int flags;                 /* 현재 I2C driver의 상태를 나타내는 flag */
    int (*attach_adapter)(struct i2c_adapter *); /* 새로운 디바이스가 있을 때 호출 */
    int (*detach_client)(struct i2c_client *); /* Client를 제거할 때 호출 */
    int (*command)(struct i2c_client *client,unsigned int cmd, void *arg); /* 디바이스에 명령을 내림:
ioctl()*/
    void (*inc_use)(struct i2c_client *client); /* Client의 사용 카운트를 증가 시킴 */
    void (*dec_use)(struct i2c_client *client); /* Client의 사용 카운트를 감소 시킴 */
};
```

코드 1219. i2c_driver 구조체의 정의

i2c_driver 구조체는 I2C adapter상에 존재할 수 있는 하나나 그 이상의 물리적인 디바이스들을 다루기 위해서 존재한다.

```
/*
```

⁴⁸⁰ 그림에서는 이 부분을 표시하지 않았지만, I2C core 상위에 이를 이용한 디바이스 드라이버가 있다고 생각하면 될 것이다.

```

* i2c_client identifies a single device (i.e. chip) that is connected to an
* i2c bus. The behaviour is defined by the routines of the driver. This
* function is mainly used for lookup & other admin. functions.
*/
struct i2c_client {
    char name[32];           /* I2C client의 이름 */
    int id;                  /* I2C client의 ID */
    unsigned int flags;       /* 현재 client의 상태 */
    unsigned int addr;        /* 이 client의 I2C bus 상에서의 주소(address) : 하위 7 bit만 사용*/
    struct i2c_adapter *adapter; /* 이 client가 놓인 bus를 관리하는 adapter에 대한 포인터 */
    struct i2c_driver *driver; /* 이 client에 접근하기 위한 driver에 대한 포인터 */
    void *data;               /* Client의 private 데이터 */
    int usage_count;          /* 현재 client에 대한 사용 카운트 */
};

```

코드 1220. i2c_client 구조체의 정의

i2c_client는 client 디바이스가 I2C bus상에서 감지될 때마다 하나씩 생성되는 구조체이다. 즉, 각각의 client 디바이스에 대한 정보를 유지하기 위해서 사용한다.

```

...
/* i2c_driver 구조체의 flags 값 정의 */
#define I2C_DF_NOTIFY      0x01           /* Bus상에 attach나 detach가 발생했을 때 알려달라 */
*/
#define I2C_DF_DUMMY      0x02           /* Dummy driver이다. 다른 client에 연결하지 말라.*/
/* i2c_client 구조체의 flags 값 정의 */
#define I2C_CLIENT_ALLOW_USE    0x01 /* Client가 접근(access)을 허가한다.*/
#define I2C_CLIENT_ALLOW_MULTIPLE_USE 0x02 /* Multiple(다중) 접근을 lock을 통해서
허가한다.*/
...

```

코드 1221. i2c Bus 자료구조의 flags 값 정의

앞에서 본 i2c_driver구조체와 i2c_client 구조체의 flag 필드는 위와 같은 값으로 정의된다. 즉, driver가 알기를 원하는 event가 무엇인지를 표현하거나, i2c client가 어떤 접근을 허가하는지를 표현하기 위한 것이다.

```

/*
 * The following structs are for those who like to implement new bus drivers:
 * i2c_algorithm is the interface to a class of hardware solutions which can
 * be addressed using the same bus algorithms - i.e. bit-banging or the PCF8584
 * to name two of the most common.
*/
struct i2c_algorithm {
    char name[32];           /* Algorithm의 이름 */
    unsigned int id;          /* Algorithm의 ID */

    /* 아래의 두개의 함수 포인터는 I2C Bus상에서 데이터를 주고 받기 위한 방법이다.
    만약 adapter가 I2C 레벨의 bus access를 하지 못한다면, master_xfer 함수 포인터를
    NULL로 두어라. 또한 SMBus(System Management Bus) access를 지원하다면,
    smbus_xfer 함수 포인터를 설정하라. 만약 이것이 NULL로 설정된다면, SMBus protocol을
    일반적인 I2C message를 이용해서 simulation할 것이다.*/
    int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg msgs[],
                      int num);
    int (*smbus_xfer)(struct i2c_adapter *adap, u16 addr,

```

```

        unsigned short flags, char read_write,
        u8 command, int size, union i2c_smbus_data * data);

/* 이것은 특정 adapter의 경우에 option으로 사용되거나, 혹은 나중을 대비하기 위한 것이다.*/
int (*slave_send)(struct i2c_adapter *,char*,int); /* Bus의 slave로 데이터를 전달한다.*/
int (*slave_recv)(struct i2c_adapter *,char*,int); /* Bus의 slave로 데이터를 받는다.*/

/* ioctl() */
int (*algo_control)(struct i2c_adapter *, unsigned int, unsigned long); /* Adapter에 대한 제어를 한다. */

u32 (*functionality)(struct i2c_adapter *); /* Adapter의 지원 사항을 알아보기 위해서 사용한다. */
};

}

```

코드 1222. i2c_algorithm 구조체의 정의

i2c_algorithm은 실제 adapter를 통해서 특정 client 디바이스에 데이터를 전달하거나 받기를 원할 때 사용한다. 또한 Bus에 대한 제어와 기능 등을 알기를 원할 때도 사용될 수 있을 것이다.

```

/*
 * i2c_adapter is the structure used to identify a physical i2c bus along
 * with the access algorithms necessary to access it.
 */
struct i2c_adapter {
    char name[32];           /* I2C bus adapter의 이름 */
    unsigned int id;          /* I2C bus adapter의 ID = algorithm ID OR hardware ID : i2c-id.h 참고*/
    struct i2c_algorithm *algo; /* I2C bus를 접근하기 위한 algorithm에 대한 포인터 */
    void *algo_data;          /* Algorithm의 private data에 대한 포인터 */

    /* Module의 사용 카운트를 증가/감소 시킴 :NULL로 설정할 경우 반드시 모듈
     카운트를 따로 증가/감소 시켜주어야 함.*/
    void (*inc_use)(struct i2c_adapter *);
    void (*dec_use)(struct i2c_adapter *);

    int (*client_register)(struct i2c_client *);      /* Client의 등록 */
    int (*client_unregister)(struct i2c_client *);     /* Client의 해제 */

    void *data;           /* Adapter를 위한 private 데이터 포인터 */
    struct semaphore lock; /* Adapter의 자료구조 수정을 위한 세마포어 */
    unsigned int flags;   /* Adapter의 현재 상태를 나타냄 */
    struct i2c_client *clients[I2C_CLIENT_MAX]; /* 등록된 client들에 대한 포인터 배열 */
    int client_count;    /* 등록된 client의 개수 */
    int timeout;          /* Adapter의 timeout 간격 */
    int retries;          /* Adapter의 retry 회수 */

/* 이하는 proc file system상에 I2C adapter의 상태를 나타내기 위해서 사용하는 자료 구조이다.*/
#endif CONFIG_PROC_FS
    /* No need to set this when you initialize the adapter */
    int inode;
#endif LINUX_VERSION_CODE < KERNEL_VERSION(2,1,29)
    struct proc_dir_entry *proc_entry;
#endif
#endif /* def CONFIG_PROC_FS */
};

}

```

코드 1223. i2c_adapter 구조체의 정의

i2c_adapter는 I2C bus의 디바이스 드라이버에서 핵심이라고 볼 수 있는 자료 구조이다. 즉, I2C Bus 상에 있는 디바이스를 접근하기 위해서 구동하는 I2C bus chip set을 나타내기 때문이다. 이 chip set이 bus를 접근하기 위한 방법은 i2c_algorithm의 포인터인 algo를 사용하게 되며, 새로운 client 디바이스가 검출될 때마다, i2c_client라는 구조체의 포인터 배열인 clients에 들어가게 될 것이다.

이전 앞에서 설명한 각각의 자료구조 간의 관계를 보도록 하자. [그림 138]은 이러한 관계를 보여주고 있으며, 각각이 어떤 필드를 통해서 접근할 수 있는지를 화살표로 표시했다.

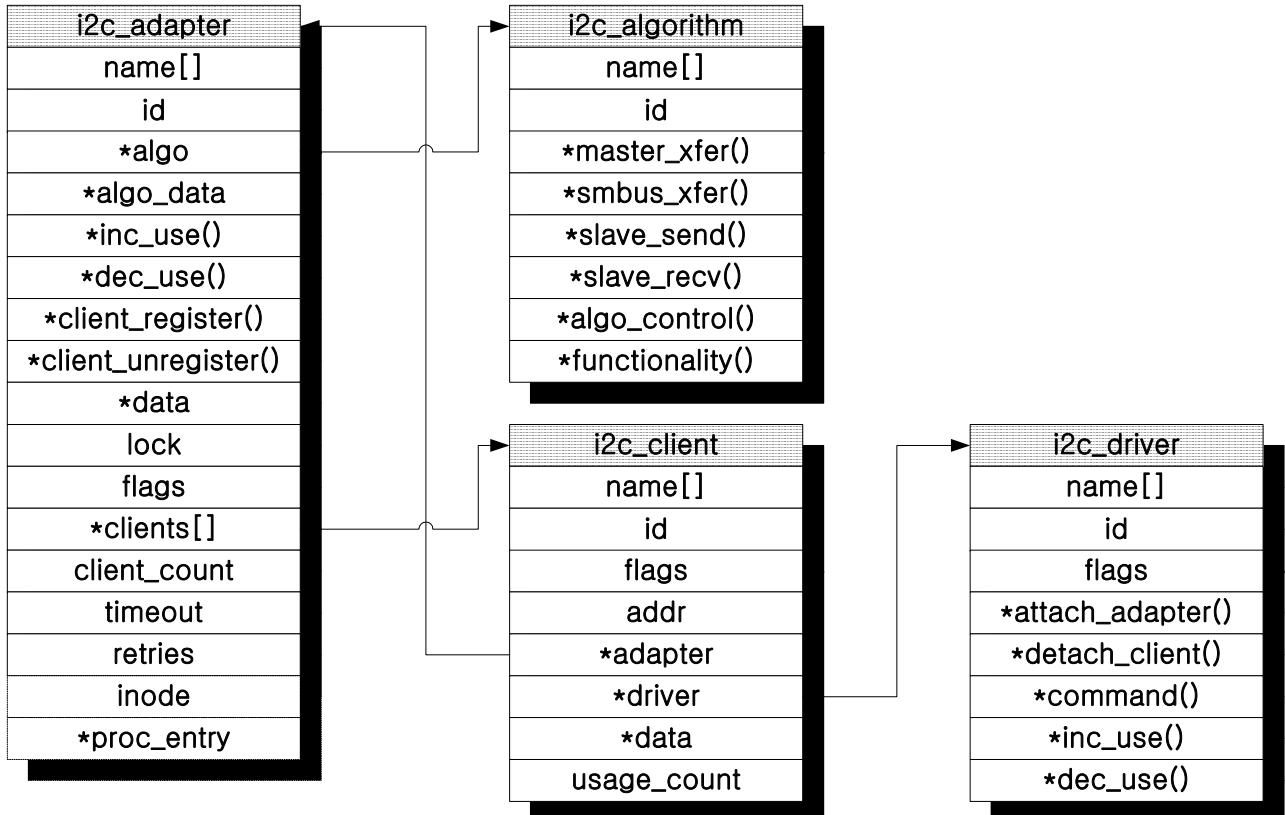


그림 138. I2C Bus의 자료구조 간의 관계

그림에서 보듯이 I2C bus를 관리하는 chip은 하나의 adapter로 등록되며, 이 chip에서 I2C bus의 client들과 연결하기 위한 방법으로 i2c_algorithm 구조체를 사용한다. 따라서, 각각의 adapter들은 자신들만의 i2c_algorithm 구조체를 가지고 있어야 할 것이다. 또한, 특정 I2C bus상의 디바이스를 표현하기 위해서는 i2c_client 구조체를 선언해야 하며, 이는 i2c_client 디바이스 드라이버를 만들어서 i2c_adapter 구조체의 clients[] 필드에 연결되어야 한다. 또한 이러한 i2c_client들은 자신을 관리하기 위한 i2c_driver라는 구조체를 필요로 하며, 이 구조체가 I2C bus client에 대한 디바이스 드라이버의 역할을 수행한다.

자, 그럼 이제 차근차근 하나씩 보기로 하자. 가장 먼저 볼 것은 I2C core가 될 것이다. 파일은 ~/drivers/i2c/i2c-core.c이다.

19.1.1. i2c-core.c의 분석

I2C의 초기화는 i2c_init() 함수에서 이루어진다. 모듈로 로딩될 때도 역시 이 함수를 호출해서 초기화를 수행한다.

```

static int __init i2c_init(void)
{
    printk("i2c-core.o: i2c core module\n");
    memset(adapters, 0, sizeof(adapters));
}
  
```

```

memset(drivers,0,sizeof(drivers));
adap_count=0;
driver_count=0;

init_MUTEX(&adap_lock);
init_MUTEX(&driver_lock);

i2cproc_init();

return 0;
}

```

코드 1224. i2c_init() 함수의 정의

i2c_init() 함수는 I2C bus를 사용하기 위한 초기 동작을 수행한다. 먼저 i2c_adapter 구조체로 선언된 adapters와 i2c_driver 구조체로 선언된 drivers를 전부 0으로 초기화 한다(memset()). 또한, 각각에 대한 카운트를 0으로 초기화 하고, 각각의 자료구조를 접근하기 위한 lock을 초기화하기 위해서 init_MUTEX()를 호출한다. 이것을 마치면, i2cproc_init()를 호출해서 I2C bus를 위한 proc 파일 시스템의 등록을 초기화 한다. 복귀 값은 0이다.

```

...
struct semaphore adap_lock;
struct semaphore driver_lock;

/** adapter list */
static struct i2c_adapter *adapters[I2C_ADAP_MAX];
static int adap_count;

/** drivers list */
static struct i2c_driver *drivers[I2C_DRIVER_MAX];
static int driver_count;
...

```

코드 1225. I2C 전역 변수에 대한 정의

i2c_adapter와 i2c_driver 구조체들을 유지하기 위해서 I2C bus에서는 adapters[]와 drivers[]라는 포인터의 배열을 정의하고 있다. 각각은 또한 자신들의 배열에서 유효한 엔트리(entry)가 얼마나 되는지를 유지하기 위해서 adap_count와 driver_count를 가진다. 각각은 자료구조의 접근에 대한 동기화를 위해서 세마포어로 adap_lock과 driver_lock을 또한 가진다. 여기서 배열의 크기가 되는 값은 각각 I2C_ADAP_MAX(= 16)와 I2C_DRIVER_MAX(= 16)이다. 따라서, 최대 16개의 adapter와 16개의 client 디바이스를 지원한다. init_MUTEX()는 단순히 세마포어를 초기값으로 설정하는 역할을 수행한다.

```

int i2cproc_init(void)
{
    struct proc_dir_entry *proc_bus_i2c;
    i2cproc_initialized = 0;

    if (!proc_bus) {
        printk("i2c-core.o: /proc/bus/ does not exist");
        i2cproc_cleanup();
        return -ENOENT;
    }
    proc_bus_i2c = create_proc_entry("i2c",0,proc_bus);
    if (!proc_bus_i2c) {
        printk("i2c-core.o: Could not create /proc/bus/i2c");
        i2cproc_cleanup();
    }
}

```

```

        return -ENOENT;
    }
    proc_bus_i2c->read_proc = &read_bus_i2c;
#endif /* (LINUX_VERSION_CODE >= KERNEL_VERSION(2,3,27)) */
    proc_bus_i2c->owner = THIS_MODULE;
#else
    proc_bus_i2c->fill_inode = &monitor_bus_i2c;
#endif /* (LINUX_VERSION_CODE >= KERNEL_VERSION(2,3,27)) */
    i2cproc_initialized += 2;
    return 0;
}

```

코드 1226. i2cproc_init() 함수와 i2cproc_cleanup() 함수의 정의

i2cproc_init()와 i2cproc_cleanup() 함수는 모두 커널이 CONFIG_PROC_FS라는 컴파일 option이 주어진 경우에만 사용된다. 즉, I2C bus에 대한 proc 파일 시스템의 엔트리를 설정하고, 해제하는 목적으로 사용될 수 있다. 먼저 proc 파일 시스템이 bus에 대한 인터페이스를 제공하는지를 보기위해서 proc_bus를 확인한다. 없다면, 오류 메시지를 보이고, i2cproc_cleanup() 함수를 호출한다. 복귀 값은 -ENOENT(Error No Entry)가 될 것이다. 이젠 I2C bus의 entry를 /proc/bus이하에 생성하기 위해서 create_proc_entry() 함수를 호출한다. 생성되는 proc 파일 시스템의 entry는 /proc/bus/i2c가 될 것이다. 생성에 오류가 있었다면, 앞에서와 같이 오류 메시지를 보이고, i2cproc_cleanup() 함수를 호출한 후, -ENOENT를 돌려준다. 이젠 proc 파일 시스템에 대한 read() 함수를 연결하고, 커널 버전에 맞는 일을 수행한다. 커널 버전 2.3.27보다 크다면, 생성된 proc 파일 시스템의 entry를 위한 owner 필드에 현재 모듈을 연결하고, 그렇지 않다면, fill_inode 함수에 monitor_bus_i2c를 연결한다. 이젠 proc 파일 시스템에 대한 I2C bus의 초기화를 마쳤으므로 i2cproc_initialized에 2를 더한다⁴⁸¹.

```

int i2cproc_cleanup(void)
{
    if (i2cproc_initialized >= 1) {
        remove_proc_entry("i2c", proc_bus);
        i2cproc_initialized -= 2;
    }
    return 0;
}

```

코드 1227. i2cproc_cleanup() 함수의 정의

i2cproc_cleanup() 함수 역시 CONFIG_PROC_FS가 설정된 경우에만 유효하다. i2cproc_initialized가 1이상의 값을 가지는 경우에만 proc 파일 시스템에서 “i2c” 엔트리를 지우도록 한다(remove_proc_entry()). 앞에서 i2cproc_initialized를 2증가 시켰으므로, 여기선 2를 감소시키도록 한다. 복귀 값은 0이다.

```

#if (LINUX_VERSION_CODE <= KERNEL_VERSION(2,3,27))
/* Monitor access to /proc/bus/i2c*; make unloading i2c-proc impossible
   if some process still uses it or some file in it */
void monitor_bus_i2c(struct inode *inode, int fill)
{
    if (fill)
        MOD_INC_USE_COUNT;
    else
        MOD_DEC_USE_COUNT;
}
#endif /* (LINUX_VERSION_CODE <= KERNEL_VERSION(2,3,37)) */

```

⁴⁸¹ 2를 더하는 이유는 확실하지 않다. 1을 더해도 I2C bus가 초기화 되었다는 것을 표현하는 데는 지장이 없으리라 생각한다.

```

...
/* This function generates the output for /proc/bus/i2c */
int read_bus_i2c(char *buf, char **start, off_t offset, int len, int *eof,
                  void *private)
{
    int i;
    int nr = 0;
    /* Note that it is safe to write a `little` beyond len. Yes, really. */
    for (i = 0; (i < I2C_ADAP_MAX) && (nr < len); i++)
        if (adapters[i]) {
            nr += sprintf(buf+nr, "i2c-%d\t", i);
            if (adapters[i]->algo->smbus_xfer) {
                if (adapters[i]->algo->master_xfer)
                    nr += sprintf(buf+nr,"smbus/i2c");
                else
                    nr += sprintf(buf+nr,"smbus      ");
            } else if (adapters[i]->algo->master_xfer)
                nr += sprintf(buf+nr,"i2c      ");
            else
                nr += sprintf(buf+nr,"dummy      ");
            nr += sprintf(buf+nr,"%t%-32s%t%-32s\n",
                          adapters[i]->name,
                          adapters[i]->algo->name);
        }
    return nr;
}

```

코드 1228. monitor_bus_i2c()와 read_bus_i2c() 함수의 정의

monitor_bus_i2c() 함수는 만약 /proc/bus/i2c에 접근이 있는 경우에 i2c-core 모듈의 unloading을 하지 못하게 하기 위함이다. 따라서, 넘겨받은 fill 값에 따라, MOD_INC_USE_COUNT나 MOD_DEC_USE_COUNT를 사용해서 모듈의 사용 카운트를 증가 시켜준다. 사용중인 모듈에 대해서는 rmmod와 같은 것으로 unloading을 하지 못한다.

read_bus_i2c() 함수는 proc 파일 시스템에 등록된 I2C bus에 대한 관련 사항을 읽기 위해서 호출된다. 즉, 사용자 프로그램에서 /proc/bus/i2c를 open()한 후, read() 함수와 같은 것을 호출할 때 실행될 것이다. 여기서는 사용자 데이터 영역(buf)으로 I2C bus상의 adapter의 정보를 복사한다. 즉, adapters[] 배열을 차례로 접근해서 smbus_xfer 필드가 정의된 경우에는 “smbus/i2c”를, 그렇지 않다면, “smbus”를 buf로 복사하고, master_xfer 필드가 정의된 경우에는 “i2c ”, 그렇지 않다면, “dummy ”를 복사한다. 또한 adapter의 이름과 algorithm의 이름 필드도 함께, 그 다음 buf의 위치에 복사한다. 복귀 값은 이렇게 복사한 데이터의 크기를 나타내는 nr이다.

```

int i2c_add_adapter(struct i2c_adapter *adap)
{
    int i,j,res;

    ADAP_LOCK();
    for (i = 0; i < I2C_ADAP_MAX; i++)
        if (NULL == adapters[i])
            break;
    if (I2C_ADAP_MAX == i) {
        printk(KERN_WARNING
              " i2c-core.o: register_adapter(%s) - enlarge I2C_ADAP_MAX.\n",
              adap->name);
        res = -ENOMEM;
        goto ERROR0;
    }
    adapters[i] = adap;

```

```

adap_count++;
ADAP_UNLOCK();
/* init data types */
init_MUTEX(&adap->lock);

```

코드 1229. i2c_add_adapter() 함수의 정의

i2c_add_adapter() 함수는 새로운 adapter를 등록을 위해, algorithm layer에서 호출될 것이다. 먼저 adapter에 대한 lock을 얻고(ADAP_LOCK()), adapters[] 배열에서 아직 사용되지 않는 빈자리가 있는지를 찾는다. 만약 빈자리가 없다면, 오류 메시지를 표시하고(printk()), -ENOMEM(Error No Memory)를 복귀 값으로 저장하고, ERROR0로 제어를 옮긴다. 그렇지 않다면, 넘겨받는 i2c_adapter 구조체의 포인터를 adapters[] 배열의 빈자리에 넣고, adapter의 카운트를 증가시킨다(adap_count++). 이전 앞에서 설정한 lock을 해제한다. 이전 넘겨받은 i2c_adapter 구조체의 lock을 초기화 시키기 위해 init_MUTEX()를 호출한다. 이하는 proc 파일 시스템에 대한 등록과정과 이미 등록되어 있을지 모를 driver들에 대한 event 알리기 등이다.

```

#endif CONFIG_PROC_FS
    if (i2cproc_initialized) {
        char name[8];
        struct proc_dir_entry *proc_entry;

        sprintf(name, "i2c-%d", i);
        proc_entry = create_proc_entry(name, 0, proc_bus);
        if (!proc_entry) {
            printk("i2c-core.o: Could not create /proc/bus/%s\n",
                  name);
            res = -ENOENT;
            goto ERROR1;
        }
#ifndef LINUX_VERSION_CODE
        proc_entry->proc_fops = &i2cproc_operations;
#else
        proc_entry->ops = &i2cproc_inode_operations;
#endif
#ifndef LINUX_VERSION_CODE
        proc_entry->owner = THIS_MODULE;
#else
        proc_entry->fill_inode = &monitor_bus_i2c;
#endif /* (LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,58)) */
        adap->inode = proc_entry->low_ino;
    }
#endif /* def CONFIG_PROC_FS */
    /* inform drivers of new adapters */
    DRV_LOCK();
    for (j=0;j<I2C_DRIVER_MAX;j++)
        if (drivers[j]!=NULL &&
            (drivers[j]->flags&(I2C_DF_NOTIFY|I2C_DF_DUMMY)))
            /* We ignore the return code; if it fails, too bad */
            drivers[j]->attach_adapter(adap);

    DRV_UNLOCK();
    DEB	printk("i2c-core.o: adapter %s registered as adapter %d.\n",
              adap->name,i));
    return 0;
ERROR1:
    ADAP_LOCK();
    adapters[i] = NULL;
    adap_count--;

```

ERROR0:

```
    ADAP_UNLOCK();
    return res;
}
```

코드 1230. i2c_add_adapter() 함수의 정의(계속)

CONFIG_PROC_FS가 정의되었다면, i2cproc_initialized가 0이 아닌 값을 가진다면, proc 파일 시스템의 /proc/bus아하에 i2c-XXX라는 엔트리를 생성한다(create_proc_entry()), 제대로 생성되지 않았다면, 오류 메시지를 내보내고, 결과값으로(res) -ENOENT(Error No Entry)를 설정한 후 ERROR1으로 제어를 옮긴다. 제대로 생성되었다면, 커널 버전에 따라 proc 파일 시스템에 대한 연산자 벡터를 설정한다. 설정하는 연산자 벡터는 커널 버전 2.3.48이상인 경우에는 i2cproc_operations가 들어가게 되며, 그렇지 않다면, i2cproc_inode_operations가 설정된다. 또한 커널 버전 2.3.27보다 큰 경우에는 관련 모듈의 owner(소유자)를 설정하도록 하며, 그렇지 않을 경우에는 앞에서와 마찬가지로 monitor_bus_i2c() 함수로 설정한다. 마지막으로 adapter와 관련된 proc 파일 시스템의 inode 번호를 설정하면서, proc 파일 시스템에 대한 초기화를 마친다.

이전 등록된 드라이버(driver)를 볼 차례이다. 먼저 드라이버로 진입하기 앞서 관련된 lock을 설정한다(DRV_LOCK()). 만약 등록된 드라이버가 있고, 이 드라이버가 DUMMY(I2C_DF_DUMMY)이거나 혹은 adapter의 추가 시에 event를 받기를 원한다면(I2C_DF_NOTIFY), drivers[] 배열의 해당 i2c_driver구조체의 attach_adapter() 함수를 호출하도록 한다. 넘겨주는 인자는 i2c_add_adapter() 함수가 받은 넘겨받은 인자이다. 따라서, 등록된 드라이버에서는 이렇게 넘겨받은 값을 이용해서 원하는 일을 할 수 있다. 전체 drivers[] 배열에 대해서 이것을 마치고 나면, lock을 해제한다(DRV_UNLOCK()). 복귀 값은 0일 것이다. 오류가 있었을 경우에는 그때까지 했던 일을 되돌리는 절차가 진행되고, 결과 값을 복귀 값으로 넘겨준다.

```
...
/* To implement the dynamic /proc/bus/i2c-? files, we need our own
   implementation of the read hook */
static struct file_operations i2cproc_operations = {
    read:           i2cproc_bus_read,
};

#if (LINUX_VERSION_CODE < KERNEL_VERSION(2,3,48))
static struct inode_operations i2cproc_inode_operations = {
    &i2cproc_operations
};
...
```

코드 1231. proc 파일 시스템 인터페이스 함수의 정의

inode_operations과 file_operations구조체는 둘 다 proc 파일 시스템에 대한 사용자 인터페이스를 정의하기 위한 것이며, 다만 커널 버전에 따라 사용하는 경우가 달라진다. 핵심은 i2cproc_bus_read() 함수가 처리하는 것은 마찬가지이다.

```
/* This function generates the output for /proc/bus/i2c-? */
ssize_t i2cproc_bus_read(struct file * file, char * buf, size_t count, loff_t * ppos)
{
    struct inode * inode = file->f_dentry->d_inode;
    char *kbuf;
    struct i2c_client *client;
    int i,j,k,order_nr,len=0,len_total;
    int order[I2C_CLIENT_MAX];

    if (count > 4000)
        return -EINVAL;
    len_total = file->f_pos + count;
```

```
/* Too bad if this gets longer (unlikely) */
if (len_total > 4000)
    len_total = 4000;
```

코드 1232. i2cproc_bus_read() 함수의 정의

i2cproc_bus_read() 함수는 proc 파일 시스템에 대해서 실제로 사용자 프로그램에서 접근할 수 있도록 만들어주는 역할을 한다. 따라서, read() 연산에서 수행된다. read_bus_i2c() 함수와 다른 점은 이 함수는 넘겨받은 inode 번호와 같은 I2C adapter에 해당하는 모든 정보를 보여준다는 점이다. 이 정보에 들어갈 수 있는 것으로는 adapter가 관리하는 모든 client 디바이스들이다.

먼저 넘겨받은 파일 구조체로부터 해당 inode의 정보를 얻어서 inode에 저장한다. 만약 버퍼의 크기(count)가 4000이상이라면 유효하지 않은 값이라고 생각해 -EINVAL(Error Invalid)를 복귀 값으로 돌려준다. 그렇지 않을 경우에는 len_total을 현재 파일 연산에 대한 위치(f_pos)에 count를 더한 값으로 설정한다. 만약 이 값이 4000보다 크다면, 4000으로 새로 설정한다. 즉, 4000 bytes를 한계 데이터길이 값으로 둔다는 의미이다.

```
for (i = 0; i < I2C_ADAP_MAX; i++)
    if (adapters[i]->inode == inode->i_ino)
    {
        /* We need a bit of slack in the kernel buffer; this makes the sprintf safe. */
        if (! (kbuf = kmalloc(count + 80,GFP_KERNEL)))
            return -ENOMEM;
        /* Order will hold the indexes of the clients
           sorted by address */
        order_nr=0;
        for (j = 0; j < I2C_CLIENT_MAX; j++) {
            if ((client = adapters[i]->clients[j]) &&
                (client->driver->id != I2C_DRIVERID_I2CDEV)) {
                for(k = order_nr;
                    (k > 0) && adapters[i]->clients[order[k-1]]->addr > client-
>addr;
                    k--)
                    order[k] = order[k-1];
                order[k] = j;
                order_nr++;
            }
        }
        for (j = 0; (j < order_nr) && (len < len_total); j++) {
            client = adapters[i]->clients[order[j]];
            len += sprintf(kbuf+len,"%02x\t%-32s\t%-32s\n",
                           client->addr,
                           client->name,
                           client->driver->name);
        }
        len = len - file->f_pos;
        if (len > count)
            len = count;
        if (len < 0)
            len = 0;
        if (copy_to_user (buf,kbuf+file->f_pos, len)) {
            kfree(kbuf);
            return -EFAULT;
        }
        file->f_pos += len;
        kfree(kbuf);
        return len;
    }
```

```
    return -ENOENT;
}
```

코드 1233. i2cproc_bus_read() 함수의 정의(계속)

이전 등록된 adapter에서 inode에 해당하는 것을 찾아서, 이 adapter의 정보를 복사하는 일이 남았다. 앞에서 proc 파일 시스템을 위한 설정에서 해당 inode번호를 i2c_adapter구조체의 inode에 두는 것을 보았을 것이다. 이 값이 넘겨받은 inode번호(i_ino)와 비교해서 같은 것일 때만 수행한다.

먼저 count+80이라는 크기의 메모리를 커널에서 할당 받는다. 이것은 나중에 데이터를 복사하기 위한 임시 buffer로 사용된다. 할당 받을 수 없다면 -ENOMEN(Error No Memory)를 돌려준다. 이전 adapter상에서 존재하는 모든 client들에 대한 자료를 찾도록 한다. 이때 I2C_CLIENT_MAX(=32)는 하나의 adapter가 관리할 수 있는 client 디바이스를 나타내는 자료구조의 최대 개수를 의미한다. order_nr은 client들에 대한 정보를 무작위 적으로 보여주기 보다는 정렬(sorting)된 형태로 보여주기 위한 것이다. 이때 정렬하기 위한 기준이 되는 것이 client들의 주소이다. 이 주소를 가지고 정렬을 다했다면, 이전 할당 받은 커널의 버퍼에 client의 주소와 이름, 드라이버의 이름을 넣는다. 전체 데이터의 길이는 len이 유지한다. 이렇게 생성된 길이에서 file 구조체의 f_pos필드를 빼서, 얼마나 많은 데이터를 사용자 영역에 복사할지를 결정한다. 만약 이러한 길이가 0보다 작거나 count값보다 큰 경우에는 0이나, count값으로 각각의 경우에 대해서 다시 설정하도록 한다. 이전 사용자 영역으로 데이터를 복사하기 위해서 copy_to_user() 함수를 호출한다. 만약 copy_to_user()에서 오류가 발생했다면, 할당 받은 버퍼를 해제하고(kfree()), -EFAULT(Error Fault)를 돌려주도록 한다. 오류가 없었다면, file 구조체의 f_pos에 len을 더해서 다음 번의 파일 연산에 대한 위치를 변경해주고, 커널 버퍼를 해제한 후, 읽은 데이터의 길이를 가지는 len을 돌려준다. 만약 inode번호에 해당하는 adapter의 inode번호가 없을 경우에는 -ENOENT(Error No Entry)를 복귀 값으로 돌려준다.

```
int i2c_del_adapter(struct i2c_adapter *adap)
{
    int i,j,res;

    ADAP_LOCK();
    for (i = 0; i < I2C_ADAP_MAX; i++)
        if (adap == adapters[i])
            break;
    if (I2C_ADAP_MAX == i) {
        printk("i2c-core.o: unregister_adapter adap [%s] not found.\n",
               adap->name);
        res = -ENODEV;
        goto ERROR0;
    }
    /* DUMMY drivers do not register their clients, so we have to
     * use a trick here: we call driver->attach_adapter to
     * *detach* it! Of course, each dummy driver should know about
     * this or hell will break loose...
     */
    DRV_LOCK();
    for (j = 0; j < I2C_DRIVER_MAX; j++)
        if (drivers[j] && (drivers[j]->flags & I2C_DF_DUMMY))
            if ((res = drivers[j]->attach_adapter(adap))) {
                printk("i2c-core.o: can't detach adapter %s "
                      "while detaching driver %s: driver not "
                      "detached!",adap->name,drivers[j]->name);
                goto ERROR1;
            }
    DRV_UNLOCK();
```

코드 1234. i2c_del_adapter() 함수의 정의

i2c_del_adapter() 함수는 I2C bus의 adapter를 삭제하기 위해서 호출된다. 이 함수 역시 algorithm 드라이버로부터 해당 adapter를 제거하기 위해서 호출된다. 넘겨받는 인자는 i2c_adapter 구조체에 대한 포인터이다. i2c_adapter에 대한 것을 다루게 되므로, lock을 설정하기 위해서 ADAP_LOCK()을 호출한다. 이전 해당 adapter가 adapters[] 배열에 있는지를 찾는다. 만약 찾을 수 없다면, -ENODEV(Error No Device)를 결과 값으로 저장하고 ERROR0로 제어를 옮긴다. 이전 드라이버에 대한 연산을 수행하기 위해서 DRV_LOCK()을 호출해서 driver에 대한 lock을 설정한다. Dummy로 등록된 드라이버의 경우에는 attach_adapter()에 연결된 함수를 수행해서 detach를 실행한다. 물론 이에 대해서 등록되는 dummy 드라이버들은 미리 대비를 하고 있어야 할 것이다. 만약 여기서 문제가 발생했다면, 에러 메시지를 보이고, ERROR1으로 제어를 옮긴다. 드라이버에 대한 lock은 될 수 있는 한 빨리 제거한다(DRV_UNLOCK()).

```

/* detach any active clients. This must be done first, because
 * it can fail; in which case we give up. */
for (j=0;j<I2C_CLIENT_MAX;j++) {
    struct i2c_client *client = adap->clients[j];
    if (client!=NULL)
        /* detaching devices is unconditional of the set notify
         * flag, as _all_clients that reside on the adapter
         * must be deleted, as this would cause invalid states.
         */
        if ((res=client->driver->detach_client(client))) {
            printk("i2c-core.o: adapter %s not "
                  "unregistered, because client at "
                  "address %02x can't be detached. ",
                  adap->name, client->addr);
            goto ERROR0;
        }
}
#endif CONFIG_PROC_FS
if (i2cproc_initialized) {
    char name[8];
    sprintf(name,"i2c-%d", i);
    remove_proc_entry(name,proc_bus);
}
#endif /* def CONFIG_PROC_FS */
adapters[i] = NULL;
adap_count--;
ADAP_UNLOCK();
DEB	printk("i2c-core.o: adapter unregistered: %s\n",adap->name));
return 0;
ERROR0:
    ADAP_UNLOCK();
    return res;
ERROR1:
    DRV_UNLOCK();
    return res;
}

```

코드 1235. i2c_del_adapter() 함수의 정의(계속)

자, 이젠 adapter에 연결된 client들을 제거하도록 하자. adapter가 가르키는 client들을 하나씩 가져와서 client로 둔다. 이러한 client가 NULL이 아닌 경우, i2c_client 구조체가 가지고 있는 driver 필드에 등록된 detach_client()를 호출해서 해당 client를 제거하도록 한다. 오류가 있었다면, 해당 client에 대한 정보를 보여주고, ERROR0로 제어를 옮긴다. 모든 adapter가 관리하는 client에 대한 detach가 완료되었다면, 이전 proc 파일 시스템의 entry를 제거할 차례이다. 만약 i2cproc_initialized가 설정되었다면, i2c-XXX 엔트리를 제거하기 위해서 remove_proc_entry() 함수를 호출한다. adapters[] 배열에는 해당 adapter에 대한 포인터를

NULL로 설정하고, adapter의 사용 카운트를 감소시킨 후(adap_count--), adapter에 대해서 설정한 lock을 해제하도록 한다(ADAP_UNLOCK()). 여기까지 진행되어 오류가 없었다면 복귀 값은 0이다. 나머지 ERROR0와 ERROR1은 각각 설정했던 lock을 해제하고, 오류의 결과를 복귀 값으로 넘겨주는 부분이다.

```
int i2c_add_driver(struct i2c_driver *driver)
{
    int i;

    DRV_LOCK();
    for (i = 0; i < I2C_DRIVER_MAX; i++)
        if (NULL == drivers[i])
            break;
    if (I2C_DRIVER_MAX == i) {
        printk(KERN_WARNING
                " i2c-core.o: register_driver(%s) "
                "- enlarge I2C_DRIVER_MAX.\n",
                driver->name);
        DRV_UNLOCK();
        return -ENOMEM;
    }
    drivers[i] = driver;
    driver_count++;
    DRV_UNLOCK(); /* driver was successfully added */
    DEB(printh("i2c-core.o: driver %s registered.\n", driver->name));
    ADAP_LOCK();
    /* now look for instances of driver on our adapters
     */
    if (driver->flags & (I2C_DF_NOTIFY | I2C_DF_DUMMY)) {
        for (i=0;i<I2C_ADAPTER_MAX;i++)
            if (adapters[i]!=NULL)
                /* Ignore errors */
                driver->attach_adapter(adapters[i]);
    }
    ADAP_UNLOCK();
    return 0;
}
```

코드 1236. i2c_add_driver() 함수의 정의

i2c_add_driver() 함수는 i2c_driver 구조체를 등록하기 위해서 호출된다. 이 함수는 등록할 i2c_driver 구조체에 대한 포인터를 넘겨받고 있다. 먼저 i2c_driver 구조체의 배열인 drivers[]에 접근하기에 앞서 lock을 설정한다(DRV_LOCK()). 이전 drivers[] 배열에서 NULL인 빈곳을 찾도록 한다. 만약 빈자리가 없을 경우에는 오류 메시지를 보여주고, lock을 해제한 후(DRV_UNLOCK()), -ENOMEM(Error No memory)를 돌려주도록 한다.

빈 drivers[] 배열의 엔트리를 찾았다면, 이곳에 넘겨받은 i2c_driver 구조체의 포인터를 넣고, 드라이버 카운트 값을 증가시킨(driver_count++) 후, lock을 해제한다(DRV_UNLOCK()). Adapter에 대한 연산을 하기 위해 앞서 adapter에 대한 lock을 설정하고(ADAP_LOCK()), 만약 등록하고자 하는 드라이버의 flags 필드에 I2C_DF_NOTIFY나 I2C_DF_DUMMY가 설정된 경우에는 모든 adapter에 대해서 driver에서 제공하는 attach_adapter() 함수를 호출하도록 한다. 이때 사용되는 것은 비어있지 않은 adapters[] 배열의 엔트리들이다. 이것을 마치고 나면, 이전 adapter에 설정했던 lock을 해제하고(ADAP_UNLOCK()) 0을 돌려준다. 여기서 driver의 attach_adapter() 함수는 이 drvier가 어떤 adapter에 대해서 등록되어야 할 것인가를 결정하기 위해서 불려지는 것이다.

```
int i2c_del_driver(struct i2c_driver *driver)
{
    int i,j,k,res;
```

```

DRV_LOCK();
for (i = 0; i < I2C_DRIVER_MAX; i++)
    if (driver == drivers[i])
        break;
if (I2C_DRIVER_MAX == i) {
    printk(KERN_WARNING " i2c-core.o: unregister_driver: "
           "[%s] not found\n",
           driver->name);
    DRV_UNLOCK();
    return -ENODEV;
}
/* Have a look at each adapter, if clients of this driver are still
 * attached. If so, detach them to be able to kill the driver
 * afterwards.
 */
DEB2	printk("i2c-core.o: unregister_driver - looking for clients.\n");

```

코드 1237. i2c_del_driver() 함수의 정의

i2c_del_driver() 함수는 등록된 i2c_driver 구조체를 해제하기 위한 함수이다. 앞에서 본 i2c_add_driver() 함수의 쌍이다. 넘겨받는 인수는 해제할 i2c_driver 구조체에 대한 포인터이다. drivers[]에 대한 접근을 위해서 DRV_LOCK()을 호출해서 lock을 설정하고, 등록된 드라이버 중에서 해제할 것과 같은 것인지를 찾는다. 만약 없다면, 오류 메시지를 보여주고, -ENODEV(Error No Device)를 에러 값으로 넘겨주도록 한다. 물론 설정된 lock은 복귀 전에 해제해주어야 할 것이다(DRV_UNLOCK()). 나머지는 해제할 드라이버의 정보를 보여주는 debugging 코드이다.

```

/* removing clients does not depend on the notify flag, else
 * invalid operation might (will!) result, when using stale client
 * pointers.
 */
ADAP_LOCK(); /* should be moved inside the if statement... */
for (k=0;k<I2C_ADAP_MAX;k++) {
    struct i2c_adapter *adap = adapters[k];
    if (adap == NULL) /* skip empty entries. */
        continue;
    DEB2	printk("i2c-core.o: examining adapter %s:\n",
               adap->name));
    if (driver->flags & I2C_DF_DUMMY) {
        /* DUMMY drivers do not register their clients, so we have to
         * use a trick here: we call driver->attach_adapter to
         * *detach* it! Of course, each dummy driver should know about
         * this or hell will break loose...
        */
        if ((res = driver->attach_adapter(adap))) {
            printk("i2c-core.o: while unregistering "
                   "dummy driver %s, adapter %s could "
                   "not be detached properly; driver "
                   "not unloaded!", driver->name,
                   adap->name);
            ADAP_UNLOCK();
            return res;
        }
    } else {
        for (j=0;j<I2C_CLIENT_MAX;j++) {
            struct i2c_client *client = adap->clients[j];

```

```

        if (client != NULL &&
            client->driver == driver) {
            DEB2(printh("i2c-core.o: "
                        "detaching client %s:\n",
                        client->name));
            if ((res = driver->
                detach_client(client))) {
                printk("i2c-core.o: while "
                    "unregistering driver "
                    "'%s', the client at "
                    "address %02x of
                    adapter '%s' could not
                    be detached; driver
                    not unloaded!",
                    driver->name,
                    client->addr,
                    adap->name);
                ADAP_UNLOCK();
                return res;
            }
        }
    }
    ADAP_UNLOCK();
    drivers[i] = NULL;
    driver_count--;
    DRV_UNLOCK();
    DEB(printh("i2c-core.o: driver unregistered: %s\n", driver->name));
    return 0;
}

```

코드 1238. i2c_del_driver() 함수의 정의(계속)

이전 adapters[]에 대한 등록해제를 할 차례이다. 이를 위해서 adapter lock을 설정한다(ADAP_LOCK()). 등록된 각각의 adapter별로 수행하도록 한다. 만약 해제할 드라이버의 flags 필드에 I2C_DF_DUMMY가 있는 경우에는 앞에서 보았던 것과 마찬가지로 i2c_driver 구조체의 attach_adapter() 함수를 호출해서 detach시키도록 한다. 이때 오류가 있었다면, 오류 값을 결과 값(res)로 놓고, LOCK을 해제한 후(ADAP_UNLOCK()) 복귀 하도록 한다.

Dummy 드라이버가 아닌 경우에는 각각의 adapter에 등록된 client들에 대해서 다음과 같은 일을 하도록 한다. 등록된 client가 해제하려고 하는 driver에 대한 연결을 가진 경우, detach_client() 함수를 실행해서 client를 detach하도록 한다. 오류가 있었다면, 결과 값(res)을 설정하고, lock을 해제(ADAP_UNLOCK()) 후 복귀 한다.

위와 같은 과정은 전체 등록된 adapter들에 대해서 순환적으로 계속 수행된다. 전체 adapter에 대한 것을 다 수행했다면, adapter에 대한 lock을 해제한 후(ADAP_UNLOCK()), drivers[] 배열에 있는 해제하려고 하는 드라이버의 엔트리에는 NULL을 넣고, driver의 카운트를 감소시킨(driver_count--) 후, 드라이버의 lock도 해제한다(DRV_UNLOCK()). 여기까지 진행할 수 있었다면, 오류가 없다고 보고 복귀 값으로 0을 돌려준다.

```

int i2c_attach_client(struct i2c_client *client)
{
    struct i2c_adapter *adapter = client->adapter;
    int i;

    if (i2c_check_addr(adapter, client->addr))

```

```

        return -EBUSY;
    for (i = 0; i < I2C_CLIENT_MAX; i++)
        if (NULL == adapter->clients[i])
            break;
    if (I2C_CLIENT_MAX == i) {
        printk(KERN_WARNING
            " i2c-core.o: attach_client(%s) - enlarge I2C_CLIENT_MAX.\n",
            client->name);
        return -ENOMEM;
    }
    adapter->clients[i] = client;
    adapter->client_count++;
    if (adapter->client_register)
        if (adapter->client_register(client))
            printk("i2c-core.o: warning: client_register seems "
                "to have failed for client %02x at adapter %s\n",
                client->addr, adapter->name);
    DEB	printk("i2c-core.o: client [%s] registered to adapter [%s](pos. %d).\n",
        client->name, adapter->name, i));
    if (client->flags & I2C_CLIENT_ALLOW_USE)
        client->usage_count = 0;
    return 0;
}

```

코드 1239. i2c_attach_client() 함수의 정의

i2c_attach_client() 함수는 i2c_client 구조체를 추가하기 위해서 호출된다. 넘겨받는 인수 역시 i2c_client 구조체에 대한 포인터이다. i2c_client 구조체로부터 adapter에 대한 포인터를 얻는다. 이 adapter에 우리가 넘겨준 i2c_client 구조체를 등록할 것이므로, 이미 i2c_adapter 구조체를 얻어서, i2c_client구조체를 생성할 때 넣어주어야 할 것이다. i2c_check_addr() 함수는 client의 주소를 확인하는 함수이다. 만약 오류가 있다면, -EBUSY를 돌려주게 된다.

이전 adapters[] 배열에서 빈 곳을 찾도록 한다. 만약 이러한 곳이 없다면, 오류 메시지를 보여주고 -ENOMEM(Error No Memory)를 돌려준다. 찾았다면, 해당 adapter[] 배열의 clients[]에 넘겨받은 i2c_client에 대한 포인터를 넣고, client의 count값을 증가 시켜준다(adapter->client_count++). 만약 i2c_adapter구조체의 client_register() 함수가 정의되어 있다면, 이 함수를 호출해주도록 한다. 이때 오류가 발생하면, 오류 메시지를 보여준다. 하지만, 등록을 해제하지는 않고, 단순히 warning으로 처리하도록 한다. 추가하려는 i2c_client 구조체의 flags에 I2C_CLIENT_ALLOW_USE가 설정되었다면, i2c_client구조체의 usage_count에는 0을 두어 초기화 시켜준다. 이제 i2c_client구조체는 사용될 준비를 마쳤다. 여기까지 진행했다면, 복귀 값은 0이다.

```

int i2c_check_addr (struct i2c_adapter *adapter, int addr)
{
    int i;
    for (i = 0; i < I2C_CLIENT_MAX ; i++)
        if (adapter->clients[i] && (adapter->clients[i]->addr == addr))
            return -EBUSY;
    return 0;
}

```

코드 1240. i2c_check_addr() 함수의 정의

i2c_check_addr() 함수는 I2C bus의 client가 제대로 된 주소를 가지고 있는지를 확인하는 함수이다. 즉, adapter가 관리하는 client의 주소 중에서 검사하려는 주소에 해당하는 것이 있는지를 확인한다. 만약 같은 주소가 이미 사용중이라면 -EBUSY(Error Busy)를 돌려준다. 그렇지 않다면, 0을 돌려줄 것이다.

```
int i2c_detach_client(struct i2c_client *client)
```

```
{
    struct i2c_adapter *adapter = client->adapter;
    int i,res;

    for (i = 0; i < I2C_CLIENT_MAX; i++)
        if (client == adapter->clients[i])
            break;
    if (I2C_CLIENT_MAX == i) {
        printk(KERN_WARNING " i2c-core.o: unregister_client "
               "[%s] not found\n",
               client->name);
        return -ENODEV;
    }
    if( (client->flags & I2C_CLIENT_ALLOW_USE) &&
        (client->usage_count>0))
        return -EBUSY;
    if (adapter->client_unregister != NULL)
        if ((res = adapter->client_unregister(client))) {
            printk("i2c-core.o: client_unregister [%s] failed, "
                   "client not detached",client->name);
            return res;
        }
    adapter->clients[i] = NULL;
    adapter->client_count--;
    DEB	printk("i2c-core.o: client [%s] unregistered.\n",client->name));
    return 0;
}
```

코드 1241. i2c_detach_client() 함수의 정의

i2c_detach_client() 함수는 i2c_attach_client() 함수의 역을 수행한다. 먼저 detach를 하려는 i2c_client 구조체의 포인터로부터 관련된 i2c_adapter 구조체를 찾는다. 이전 해당 i2c_client 구조체가 이렇게 찾은 i2c_adapter 구조체의 client로 있는지를 확인한다. 만약 없다면, -ENODEV(Error No Device)를 돌려준다. 있다면, i2c_client 구조체의 flags에 I2C_CLIENT_ALLOW_USE가 설정되어 있고, 사용카운트가 0보다 큰 경우에는 client가 사용주이므로 -EBUSY(Error Busy)를 돌려준다. 그렇지 않다면, i2c_adapter 구조체의 client_unregister() 함수가 정의된 경우에 한해서 client_unregister() 함수를 i2c_client 구조체의 포인터를 넘겨주어서 호출한다. 이때 오류가 있었다면, 오류 메시지를 표시하고 결과 값을 돌려준다. 앞의 과정에서 오류가 없었다면, 해당 i2c_adapter 구조체의 clients[] 배열에서 detach하려는 client의 엔트리를 NULL로 만들고, client의 개수를 감소시키고(adapter->client_count--) 나서 0을 돌려준다.

```
void i2c_inc_use_client(struct i2c_client *client)
{
    if (client->driver->inc_use != NULL)
        client->driver->inc_use(client);
    if (client->adapter->inc_use != NULL)
        client->adapter->inc_use(client->adapter);
}

void i2c_dec_use_client(struct i2c_client *client)
{
    if (client->driver->dec_use != NULL)
        client->driver->dec_use(client);
    if (client->adapter->dec_use != NULL)
        client->adapter->dec_use(client->adapter);
}
```

코드 1242. i2c_inc_use_client() 함수와 i2c_dec_use_client() 함수의 정의

i2c_inc_use_client() 함수와 i2c_dec_use_client() 함수는 등록된 i2c_client 구조체의 i2c_driver 구조체 및 i2c_adapter의 사용 카운터를 증가/감소 시키기 위해서 호출한다. 각각은 먼저 driver의 사용 카운터를 증가/감소시키고 나서, adapter의 사용 카운터를 감소시켜준다. 물론 이러한 일을 수행하기에 앞서 관련 함수들이 정의되어 있는가를 먼저 살펴본다. 각각은 증가/감소시키기 위한 i2c_client 구조체의 포인터를 넘겨받는다.

```
int i2c_use_client(struct i2c_client *client)
{
    if(client->flags & I2C_CLIENT_ALLOW_USE) {
        if (client->flags & I2C_CLIENT_ALLOW_MULTIPLE_USE)
            client->usage_count++;
        else {
            if(client->usage_count > 0)
                return -EBUSY;
            else
                client->usage_count++;
        }
    }
    i2c_inc_use_client(client);
    return 0;
}

int i2c_release_client(struct i2c_client *client)
{
    if(client->flags & I2C_CLIENT_ALLOW_USE) {
        if(client->usage_count>0)
            client->usage_count--;
        else
        {
            printk(KERN_WARNING " i2c-core.o: dec_use_client used one too many times\n");
            return -EPERM;
        }
    }
    i2c_dec_use_client(client);
    return 0;
}
```

코드 1243. i2c_use_client() 함수와 i2c_release_client() 함수의 정의

i2c_use_client() 함수는 client의 사용 카운트 및 driver와 adapter의 사용 카운트까지도 증가시키기 위해서 사용한다. i2c_release_client() 함수는 i2c_use_client() 함수의 역으로 client 및 driver와 adapter의 사용 카운트를 감소시킨다. 결과적으로 client가 사용되기 시작하면, i2c_use_client() 함수를 호출하고, 더 이상 사용되지 않는다면, i2c_release_client() 함수를 호출하면 될 것이다. 각 함수는 i2c_client 구조체의 포인터를 넘겨받아, 이 client가 I2C_CLIENT_ALLOW_USE를 가질 때, client의 사용 카운트에 대한 증가/감소를 행하고, i2c_inc_use_client() / i2c_dec_use_client() 함수를 호출해서 driver 및 adapter에 대한 사용 카운트를 증가/감소 시킨다. 이때, 만약 client가 다중 접근(Multiple Use)을 허가하지 않는다면, 사용 카운트가 0이상일 때는 사용 카운트를 증가시키기 대신에 -EBUSY(Error Busy)를 돌려줄 것이다. 또한, 사용 카운트가 0보다 작거나 같은 값을 가지는 client에 대한 사용 카운트 감소도 역시 오류로 처리해서 -EPERM(Error Permission)을 돌려줄 것이다. 정상적으로 수행되었다면, 둘 다 0을 돌려준다.

```
struct i2c_client *i2c_get_client(int driver_id, int adapter_id, struct i2c_client *prev)
{
    int i,j;

    i = j = 0;
    /* set starting point */
```

```

if(prev)
{
    if(!(prev->adapter))
        return (struct i2c_client *) -EINVAL;
    for(j=0; j < I2C_ADAP_MAX; j++)
        if(prev->adapter == adapters[j])
            break;
    /* invalid starting point? */
    if (I2C_ADAP_MAX == j) {
        printk(KERN_WARNING " i2c-core.o: get_client adapter for client:[%s] not found\n",
               prev->name);
        return (struct i2c_client *) -ENODEV;
    }
    for(i=0; i < I2C_CLIENT_MAX; i++)
        if(prev == adapters[j]->clients[i])
            break;
    /* invalid starting point? */
    if (I2C_CLIENT_MAX == i) {
        printk(KERN_WARNING " i2c-core.o: get_client client:[%s] not found\n",
               prev->name);
        return (struct i2c_client *) -ENODEV;
    }
    i++; /* start from one after prev */
}

```

코드 1244. i2c_get_client() 함수의 정의

i2c_get_client() 함수는 드라이버의 ID와 adapter의 ID, i2c_client구조체에 대한 포인터인 prev를 넘겨받아서, 해당 client를 찾아, 포인터를 돌려주는 일을 한다. 따라서, 이 함수는 특정 드라이버와 adapter에 존재하는 client를 찾는데 사용할 수 있을 것이다. 이때 시작 위치는 prev가 가르키는 client를 기준으로 한다.

먼저 prev에 어떤 값이 있을 경우, prev가 adapter를 가지는지를 본다. 가지고 있지 않다면, -EINVAL(Error Invalid)을 돌려준다. 이전 최대 adapters[] 배열에서 같은 i2c_adapter 구조체가 있는지를 찾아 본다. 만약 찾을 수 없다면, 에러 메시지를 보여주고 -ENODEV(Error No Device)를 돌려준다. 그렇지 않다면, 이전 앞에서 찾은 i2c_adapter구조체에 있는 i2c_client구조체를 하나씩 찾아나간다. 여기서 다음번의 검색을 위해서 j 변수가 설정된다. prev와 같은 것을 찾을 수 없다면, 역시 오류 메시지를 보이고, 복귀 값으로 -ENODEV(Error No Device)를 돌려준다. 여기서 client에 대한 다음 index는 i 변수가 가지며, prev이후의 client에 대한 검색을 위해서 사용한다. 즉, prev 다음에 있는 client를 찾기 위한 것이다.

```

for(; j < I2C_ADAP_MAX; j++)
{
    if(!adapters[j])
        continue;
    if(adapter_id && (adapters[j]->id != adapter_id))
        continue;
    for(; i < I2C_CLIENT_MAX; i++)
    {
        if(!adapters[j]->clients[i])
            continue;
        if(driver_id && (adapters[j]->clients[i]->driver->id != driver_id))
            continue;
        if(adapters[j]->clients[i]->flags & I2C_CLIENT_ALLOW_USE)
            return adapters[j]->clients[i];
    }
    i = 0;
}
return 0;

```

{}

코드 1245. i2c_get_client() 함수의 정의(계속)

이전 adapter의 ID와 driver의 ID를 사용할 차례이다. adapters[] 배열에서 빈 곳은 건너뛰고, 비지 않은 곳에 대해서 adapter의 ID가 0이 아닌 값이 주어졌을 때 이를 비교한다. 같지 않다면, 역시 다음 adapters[] 배열의 엔트리로 건너뛴다. 만약 같은 것을 찾았다면, 이전 client가 가지고 있는 driver의 ID를 비교한다. 이때 넘겨받은 driver의 ID가 0이 아닌 경우에 비교를 하게 되며, 같지 않다면, 다음 번 client로 이동하게 된다. 하지만 만약 넘겨받은 adapter의 ID와 driver의 ID가 둘 다 0이라면, 여기서는 I2C_CLINET_ALLOW_USE를 client의 flags에 가지고 있는 client드라이버가 첫번째로 선택되게 될 것이다.

앞에서는 i2c-core.c를 이루는 자료구조를 위주로 해서 설명했다. 이젠 실제적인 디바이스의 검출과 데이터의 전송 및 I2C bus에 대한 제어에 대해서 보기로 하겠다. 먼저 볼 것은 I2C client 드라이버에서 I2C bus상에 존재하는 디바이스를 검출하기 위해서 호출하는 i2c_probe() 함수이다. 아래와 같다.

```
int i2c_probe(struct i2c_adapter *adapter, struct i2c_client_address_data *address_data,
              i2c_client_found_addr_proc *found_proc)
{
    int addr,i,found,err;
    int adap_id = i2c_adapter_id(adapter);

    /* Forget it if we can't probe using SMBUS_QUICK */
    if (!i2c_check_functionality(adapter,I2C_FUNC_SMBUS_QUICK))
        return -1;
    for (addr = 0x00; addr <= 0x7f; addr++) {
        /* Skip if already in use */
        if (i2c_check_addr(adapter,addr))
            continue;
        /* If it is in one of the force entries, we don't do any detection at all */
        found = 0;
        for (i = 0; !found && (address_data->force[i] != I2C_CLIENT_END); i += 3) {
            if (((adap_id == address_data->force[i]) ||
                 (address_data->force[i] == ANY_I2C_BUS)) &&
                (addr == address_data->force[i+1])) {
                DEB2(printh("i2c-core.o: found force parameter for adapter %d, addr %04x\n",
                            adap_id,addr));
                if ((err = found_proc(adapter,addr,0,0)))
                    return err;
                found = 1;
            }
        }
        if (found)
            continue;
    }
}
```

코드 1246. i2c_probe() 함수의 정의

i2c_probe() 함수는 이름에서 알 수 있듯이 I2C bus상에 존재하는 디바이스를 검출하기 위해서 사용하는 함수이다. 검출하기 위해서는 먼저 i2c_adapter구조체와 i2c_client_address_data 구조체 및 실제 검출을 하게 되는 i2c_client_found_addr_proc등이 필요하다. 먼저 넘겨받은 i2c_adapter 구조체의 포인터로부터 adapter의 ID를 얻는다(i2c_adapter_id()). 하지만, 여기서 말하는 adapter의 ID는 앞에서 설명한 i2c_adapter 구조체 필드에 있는 ID와는 다르다. 즉, adapters[] 배열의 할당된 index값을 돌려준다. i2c_check_functionality() 함수는 I2C bus의 algorithm 드라이버가 제공하는 functionality 값으로 i2c_algorithm 구조체의 functionality 필드에 설정된 함수를 호출해서 얻는다. 만약 정의된 것이 없다면, 0xFFFFFFFF이란 값이 주어질 것이다. 예외인 경우에는 0을 돌려 받는다. 이 함수는 단순히 넘겨주는 functionality bit flag을 제공하는가를 묻는 것이다. 나중에 다시 볼 것이다.

이전 주소 0x00부터 0x7F까지를 1씩 증가시키면서 해당 address를 사용하는 디바이스가 있는지를 확인하는 for(){ } loop이다. I2C bus는 7 bit address를 사용하며⁴⁸², 실제로는 상위 7 bit을 사용하지만 i2c-core.c에서는 오른쪽으로 한 bit을 SHIFT한 값을 client의 address로 사용한다. 따라서, 가질 수 있는 address의 최대값은 0x7F가 될 것이다. 이전 해당 adapter에 등록된 client중에서 주소(addr)를 가진 것이 있는가를 확인하기 위해서 i2c_check_addr() 함수를 호출한다. 없다면(0이 아닌 값을 돌려준다면) 당연히 다음 번 loop로 진행할 것이다.

있다면, 아래로 계속 진행한다. 여기서 우린 더 이상 진행하기에 앞서 여기서 Linux에서 사용하는 I2C bus의 address에 관련된 자료구조를 하나 보기로 하자.

```
struct i2c_client_address_data {
    unsigned short *normal_i2c;           /* 일반적으로 사용할 address를 나타낸다.*/
    unsigned short *normal_i2c_range;      /* 일반적으로 사용할 address의 범위를 나타낸다.*/
    unsigned short *probe;                /* 검사(probe)할 address를 나타낸다.*/
    unsigned short *probe_range;          /* 검사(probe)할 범위를 나타낸다.*/
    unsigned short *ignore;               /* 무시(ignore)할 address를 나타낸다.*/
    unsigned short *ignore_range;         /* 무시(ignore)할 address의 범위를 나타낸다.*/
    unsigned short *force;                /* 반드시 보아야 하는 address를 나타낸다.*/
};
```

코드 1247. i2c_client_address_data 구조체의 정의

i2c_client_address_data 구조체는 client가 사용하게 될 주소를 표현하는 방법을 기술하는 자료구조 이다. 이와 같은 자료구조를 사용하게 될 때 다음과 같은 방법을 사용하도록 한다. 먼저, 이미 정의된 상수 값을 사용하기 위해서는 반드시 i2c.h 파일을 include하여야 한다. 또한 내부적으로 i2c_client_address_data 구조체로 정의된 전역 변수로는 아래와 같은 것이 사용되기에 각각의 필드에 들어가야 할 값을 client 디바이스 드라이버에서 정의해 주어야 할 것이다.

```
/* This is the one you want to use in your own modules */
#define I2C_CLIENT_INSMOD \
I2C_CLIENT_MODULE_PARM(probe, \
    "List of adapter,address pairs to scan additionally"); \
I2C_CLIENT_MODULE_PARM(probe_range, \
    "List of adapter,start-addr,end-addr triples to scan " \
    "additionally"); \
I2C_CLIENT_MODULE_PARM(ignore, \
    "List of adapter,address pairs not to scan"); \
I2C_CLIENT_MODULE_PARM(ignore_range, \
    "List of adapter,start-addr,end-addr triples not to " \
    "scan"); \
I2C_CLIENT_MODULE_PARM(force, \
    "List of adapter,address pairs to boldly assume " \
    "to be present"); \
static struct i2c_client_address_data addr_data = \
    {normal_i2c, normal_i2c_range, \
     probe, probe_range, \
     ignore, ignore_range, \
     force}
```

코드 1248. I2C_CLIENT_INSMOD 매크로의 정의

즉, 우리가 정의해 줄 수 있는 변수로는 normal_i2c, normal_i2c_range, probe, probe_range, ignore, ignore_range, force가 있을 수 있다. 여기서 하나의 변수를 정의할 때, 제일 마지막에는

⁴⁸² 따라서, i2c_probe() 함수는 I2C Bus Specification에서 정의하고 있는 10 bit addressing은 사용하지 않는다는 것을 알 수 있다.

I2C_CLIENT_END(=0xFFFFE)를 사용해서 마지막 field에 도달했음을 알려준다. 물론 이때 각각의 변수에 들어가는 값들은 주소에 해당하는 것으로 원래 주소 값을 우측으로 1 bit SHIFT한 값을 넣어야 할 것이다. 이렇게 정의를 하고 난 후에,

```
int i2c_adapter_id(struct i2c_adapter *adap)
{
    int i;
    for (i = 0; i < I2C_ADAP_MAX; i++)
        if (adap == adapters[i])
            return i;
    return -1;
}
```

코드 1249. i2c_adapter_id() 함수의 정의

i2c_adapter_id() 함수는 앞에서 이야기 했듯이 해당 i2c_adapter 구조체가 들어있는 adapters[] 배열의 인덱스 값을 돌려준다. 해당 i2c_adapter 구조체를 찾을 수 없다면, -1을 돌려줄 것이다.

```
u32 i2c_get_functionality (struct i2c_adapter *adap)
{
    if (adap->algo->functionality)
        return adap->algo->functionality(adap);
    else
        return 0xffffffff;
}

int i2c_check_functionality (struct i2c_adapter *adap, u32 func)
{
    u32 adap_func = i2c_get_functionality (adap);
    return (func & adap_func) == func;
}
```

코드 1250. i2c_check_functionality() 함수의 정의

i2c_check_functionality() 함수는 i2c_get_functionality() 함수를 호출해서 adapter에서 제공하는 functionality를 구한다. 이렇게 구한 값과 지원하는지의 여부를 물기 위해서 func를 bit-wise AND한 값을 원래의 값과 비교한 결과를 돌려준다. 지원한다면 1을, 그렇지 않다면 0을 돌려줄 것이다. i2c_get_functionality() 함수는 i2c_adapter 구조체의 i2c_algorithm 구조체를 나타내는 algo 필드를 접근해서 functionality 필드의 함수가 정의된 경우에 이 함수를 호출해서 복귀 값을 넘겨준다. 그렇지 않다면, 0xFFFFFFFF를 돌려준다.

#define I2C_FUNC_I2C	0x00000001
#define I2C_FUNC_10BIT_ADDR	0x00000002
#define I2C_FUNC_PROTOCOL_MANGLING	0x00000004 /* I2C_M_{REV_DIR_ADDR,NOSTART} */
#define I2C_FUNC_SMBUS_QUICK	0x00010000
#define I2C_FUNC_SMBUS_READ_BYTE	0x00020000
#define I2C_FUNC_SMBUS_WRITE_BYTE	0x00040000
#define I2C_FUNC_SMBUS_READ_BYTE_DATA	0x00080000
#define I2C_FUNC_SMBUS_WRITE_BYTE_DATA	0x00100000
#define I2C_FUNC_SMBUS_READ_WORD_DATA	0x00200000
#define I2C_FUNC_SMBUS_WRITE_WORD_DATA	0x00400000
#define I2C_FUNC_SMBUS_PROC_CALL	0x00800000
#define I2C_FUNC_SMBUS_READ_BLOCK_DATA	0x01000000
#define I2C_FUNC_SMBUS_WRITE_BLOCK_DATA	0x02000000
#define I2C_FUNC_SMBUS_READ_I2C_BLOCK	0x04000000 /* New I2C-like block */
#define I2C_FUNC_SMBUS_WRITE_I2C_BLOCK	0x08000000 /* transfer */
#define I2C_FUNC_SMBUS_BYTE I2C_FUNC_SMBUS_READ_BYTE \	

```

I2C_FUNC_SMBUS_WRITE_BYT
#define I2C_FUNC_SMBUS_BYTE_DATA I2C_FUNC_SMBUS_READ_BYT_DATA \\ 
    I2C_FUNC_SMBUS_WRITE_BYT_DATA
#define I2C_FUNC_SMBUS_WORD_DATA I2C_FUNC_SMBUS_READ_WORD_DATA \\ 
    I2C_FUNC_SMBUS_WRITE_WORD_DATA
#define I2C_FUNC_SMBUS_BLOCK_DATA I2C_FUNC_SMBUS_READ_BLOCK_DATA \\ 
    I2C_FUNC_SMBUS_WRITE_BLOCK_DATA
#define I2C_FUNC_SMBUS_I2C_BLOCK I2C_FUNC_SMBUS_READ_I2C_BLOCK \\ 
    I2C_FUNC_SMBUS_WRITE_I2C_BLOCK

#define I2C_FUNC_SMBUS_EMUL I2C_FUNC_SMBUS_QUICK \\ 
    I2C_FUNC_SMBUS_BYT \\ 
    I2C_FUNC_SMBUS_BYT_DATA \\ 
    I2C_FUNC_SMBUS_WORD_DATA \\ 
    I2C_FUNC_SMBUS_PROC_CALL \\ 
    I2C_FUNC_SMBUS_WRITE_BLOCK_DATA

```

코드 1251. I2C Adapter의 Functionality 정의

이중에서, I2C_FUNC_I2C는 I2C bus의 functionality를 지원한다는 뜻이며, I2C_FUNC_10BIT_ADDR은 10 bit addressing을 지원한다는 것이며, I2C_FUNC_PROTOCOL_MANGLING은 protocol mangling을 지원한다는 것이다. 나머지 I2C_FUNC_SMBUS_XXX들은 전부 SMBus(System Management Bus)에 대한 이야기이다. 이것은 차후에 SMBus를 보게 될 때 논의하도록 하겠다.

다시 앞의 i2c_probe() 함수로 돌아가서 found를 0으로 초기화하는 부분부터 보자. 일단 아무것도 찾지 못했으므로 found를 0으로 두었다. 검색은 address_data가 가르키는 client의 address 구조체(i2c_client_address_data)의 첫번째 엔트리부터 시작하도록 한다. 이때, 제일 먼저 보는 것은 force에 정의된 주소를 찾는 것이다. 이 값이 I2C_CLIENT_END가 아니고, found가 0인 동안 지속하도록 한다. 이때 force가 가르키는 곳에 들어가는 client의 주소의 형식을 알아야 할 것이다. 코드에서 유추할 수 있는 것은 먼저 force가 3개의 엔트리를 하나의 그룹으로 인식한다는 점이다. 따라서, 첫번째 엔트리에는 adapter의 ID를 가르키는 값이 들어가고, 두 번째 엔트리에는 실제 그 client의 address가 들어간다는 것이다. 만약 adapter의 ID가 force의 첫번째 엔트리와 같거나, adapter의 ID가 ANY_I2C_BUS(=0xFFFF)와 같다면, 두 번째 엔트리에 있는 address 값을 이용해서 해당하는 client를 찾았는지를 확인하게 된다. 확인하는 절차는 found_proc() 함수를 사용하는데, 이것은 client 드라이버에서 제공하는 method가 된다. 따라서, client 드라이버는 자신이 관리할 client 디바이스가 존재하는지에 대한 여부를 판단하기 위해서 found_proc() 함수를 제공해야 한다. 이 함수의 호출 결과 오류가 있었다면, 이 값을 즉시 돌려주고, 그렇지 않다면, found를 1로 설정한 후에 loop를 계속 진행한다. 이때 증가되는 index는 3이다. 즉, 마지막 세 번째에 있는 force의 엔트리는 사용하지 않는다는 것을 알 수 있다. 만약 found가 0이 아닌 값을 가진다면, 이전 다음 번 client의 주소로 넘어가도록 한다(addr을 증가시켜서 계속 확인하도록 한다.).

```

/* If this address is in one of the ignores, we can forget about it right now */
for (i = 0; !found && (address_data->ignore[i] != I2C_CLIENT_END); i += 2) {
    if (((adap_id == address_data->ignore[i]) ||
        ((address_data->ignore[i] == ANY_I2C_BUS))) &&
        (addr == address_data->ignore[i+1])) {
        DEB2(printh("i2c-core.o: found ignore parameter for adapter %d, "
                    "addr %04x\n", adap_id ,addr));
        found = 1;
    }
}
for (i = 0; !found && (address_data->ignore_range[i] != I2C_CLIENT_END); i += 3) {
    if (((adap_id == address_data->ignore_range[i]) ||
        ((address_data->ignore_range[i]==ANY_I2C_BUS))) &&
        (addr >= address_data->ignore_range[i+1]) &&
        (addr <= address_data->ignore_range[i+2])) {
        DEB2(printh("i2c-core.o: found ignore_range parameter for adapter %d, "

```

```

        "addr %04x\n", adap_id,addr));
    found = 1;
}
if (found)
    continue;

```

코드 1252. i2c_probe() 함수의 정의(계속)

이번에는 ignore 리스트에 있는가를 보는 것이다. 여기서는 ignore list를 2개씩 그룹을 지어서 살펴본다. 이때, found가 for(){ } loop의 조건으로 들어가기에 앞에서 이미 찾았다면, 여기서 해줄 일이 없다. 역시 adapter의 ID가 ignore 리스트의 그룹에서 첫번째 엔트리가 되며, 그 다음이 address이다. 만약 ignore 그룹의 첫번째 엔트리가 adapter의 ID와 같거나, ANY_I2C_BUS와 같다면, ignore 그룹의 두 번째 엔트리와 client의 주소를 비교해 본다. 같다면, 찾기는 찾았으나 ignore(무시) list에 들어있음을 확인했으므로, 그냥 found만 1로 설정하고, 다음 번 loop로 진행한다. 이번에는 ignore range에 속하는지를 확인하는 것이다. 역시 같은 방법이며, 단지 range를 나타내는 값이 사용되기에 ignore range는 3개의 엔트리를 하나의 group으로 보고있다. 속하는 address를 사용했다면, found는 1로 설정된다. 만약 found가 1로 설정된 경우라면, 다음 번 address로 찾기를 계속한다.

```

/* Now, we will do a detection, but only if it is in the normal or
probe entries */
for (i = 0; !found && (address_data->normal_i2c[i] != I2C_CLIENT_END); i += 1) {
    if (addr == address_data->normal_i2c[i]) {
        found = 1;
        DEB2(printh("i2c-core.o: found normal i2c entry for adapter %d, "
                    "addr %02x", adap_id,addr));
    }
}
for (i = 0; !found && (address_data->normal_i2c_range[i] != I2C_CLIENT_END); i += 2) {
    if ((addr >= address_data->normal_i2c_range[i]) &&
        (addr <= address_data->normal_i2c_range[i+1])) {
        found = 1;
        DEB2(printh("i2c-core.o: found normal i2c_range entry for adapter %d, "
                    "addr %04x\n", adap_id,addr));
    }
}

```

코드 1253. i2c_probe() 함수의 정의(계속)

이번에는 normal에 속하는 client address를 찾는 부분이다. normal은 앞에서 본 force와 ignore와는 달리, client의 address만을 비교한다. 찾았을 경우에는 found를 역시 1로 설정한다. range를 검색하는 것에서도 adapter의 ID는 비교되지 않는다.

```

for (i = 0; !found && (address_data->probe[i] != I2C_CLIENT_END); i += 2) {
    if (((adap_id == address_data->probe[i]) ||
         ((address_data->probe[i] == ANY_I2C_BUS))) &&
        (addr == address_data->probe[i+1])) {
        found = 1;
        DEB2(printh("i2c-core.o: found probe parameter for adapter %d, "
                    "addr %04x\n", adap_id,addr));
    }
}
for (i = 0; !found && (address_data->probe_range[i] != I2C_CLIENT_END); i += 3) {
    if (((adap_id == address_data->probe_range[i]) ||
         (address_data->probe_range[i] == ANY_I2C_BUS)) &&
        (addr >= address_data->probe_range[i+1]) &&
        (addr <= address_data->probe_range[i+2])) {

```

```

        found = 1;
        DEB2(prtik("i2c-core.o: found probe_range parameter for adapter %d, "
                    "addr %04x\n", adap_id,addr));
    }
}
if (!found)
    continue;
/* OK, so we really should examine this address. First check
   whether there is some client here at all! */
if (i2c_smbus_xfer(adapter,addr,0,0,0,I2C_SMBUS_QUICK,NULL) >= 0)
    if ((err = found_proc(adapter,addr,0,-1)))
        return err;
}
return 0;
}

```

코드 1254. i2c_probe() 함수의 정의(계속)

이전 probe를 조사할 차례이다. 이것은 앞에서 본 force와 ignore의 경우와 동일하다. 다만, 사용되는 client의 address를 나타내는 자료구조만 차이가 나는 것 뿐이다. 여기를 다 마치고 나면, found 값을 확인한다. 만약 찾지 못했다면(found == 0), 다음 client의 address로 진행할 것이지만, 그렇지 않다면, i2c_smbus_xfer() 함수를 호출해서 결과값이 0이상인 경우에만 found_proc() 함수를 불러 client 디바이스 드라이버에 자신이 관리하는 디바이스를 찾도록 만들어준다. found_proc()에서 오류가 있었다면, 오류 값을 즉시 돌려준다.

```

s32 i2c_smbus_xfer(struct i2c_adapter * adapter, u16 addr, unsigned short flags, char read_write, u8 command, int size,
                     union i2c_smbus_data * data)
{
    s32 res;
    flags = flags & I2C_M_TEN;

    if (adapter->algo->smbus_xfer) {
        I2C_LOCK(adapter);
        res = adapter->algo->smbus_xfer(adapter,addr,flags,read_write,
                                         command,size,data);
        I2C_UNLOCK(adapter);
    } else
        res = i2c_smbus_xfer_emulated(adapter,addr,flags,read_write,
                                       command,size,data);
    return res;
}

```

코드 1255. i2c_smbus_xfer() 함수의 정의

i2c_smbus_xfer() 함수는 System Management bus에 데이터를 전송하기 위해서 호출되는 함수이다. 다른 I2C 함수에서 많이 사용되고 있는 함수로서, i2c_adapter 구조체의 algo 필드(i2c_algorithm 구조체)에 정의된 함수를 이용해서 데이터를 전달한다. 결과적으로 하위 함수들에서는 I2C bus에 데이터를 보내기 위해서 메시지(message)구조체를 사용하게 되는데 다음과 같이 정의되어 있다.

```

/*
 * I2C Message - used for pure i2c transaction, also from /dev interface
 */
struct i2c_msg {
    __u16 addr;           /* 데이터의 receiver 주소 */
    unsigned short flags; /* 메시지의 flag */
#define I2C_M_TEN      0x10 /* 10 bit로 이루어진 address를 사용한다. */

```

```
#define I2C_M_RD          0x01      /* 메시지가 읽기(read)를 요청한다.*/
#define I2C_M_NOSTART 0x4000      /* 메시지가 시작(start) bit을 사용하지 않는다.*/
#define I2C_M_REV_DIR_ADDR 0x2000    /* 메시지의 반대 방향(reverse direction) 조수를 가진다.
*/
short len;                      /* 메시지의 길이 */
char *buf;                      /* 실제 메시지 데이터를 가르키는 포인터 */
};
```

코드 1256. i2c_msg 구조체의 정의

따라서, 위에서 데이터의 전달을 위해서 넘겨주는 파라미터 값들은 전부 이러한 i2c_msg 구조체를 형성하고, 전달하는 방법을 결정하기 위해서 사용된다고 보면 될 것이다. 만약 adapter의 algo 필드에 smbus_xfer 필드가 특정 함수를 가지고 있다면, 이 함수를 이용해서 데이터를 전달한다. 이때 adapter에 대한 lock을 설정해서 adapter의 자료구조를 동기화(synchronize) 시켜주어야 할 것이다. 만약, 위와 같은 smbus_xfer 필드의 정의가 없다면, 이 adapter는 SMBus를 지원하는 것이 아니므로, SMBus에서의 데이터 전송을 emulation하기 위해서 i2c_smbus_xfer_emulated() 함수를 호출한다. 호출의 결과 값이 복귀 코드로 사용될 것이다.

i2c-core에 대한 이야기를 계속하려면, SMBus에 대한 이해가 필요하다. 즉, i2c-core에서는 데이터의 전송을 위해서 SMBus에서 사용하는 protocol을 사용하기 때문이다. SMBus는 I2C specification에 기본을 두고 있으며, battery의 잔여 시간 검사나 CPU의 온도를 측정하는데 사용되는 bus이다. SMBus에서 사용하는 protocol은 network layer라는 곳에서 정의하고 있다. 이것은 bus상에 있는 다른 device와 어떻게 데이터를 주고 받는지에 대한 sequence를 정한 것으로 bus상에 어떤 데이터가 어떻게 놀이는가에 따라서 달라지게 된다. 다음과 같은 것으로 나눠볼 수 있다. 이때 데이터를 버스를 구동하는 디바이스를 master라고 하고, 데이터의 최종 목적지가 되는 디바이스를 slave라고 표현한다.

- Quick Command : 단지 디바이스에 command를 보내고자 할 때 사용한다. 따라서, Start(S) bit와 slave를 나타내는 7 bits의 주소, Read/Write 1 bit(Rd/Wr) 및 slave로부터의 acknowledge(A)를 나타내는 1 bit과 Stop를 나타내는 1 bit(P)로 구성된다.
- Send Byte : 이것은 1 byte의 데이터를 보내기 위해서 사용한다. Quick Command의 A bit이 하에 데이터 byte가 오고, 다시 A bit이 오는 구조이다. 추가적으로 PEC(Packet Error Code)를 동반할 경우에 PEC가 오고, A bit, P bit 순으로 끝난다.
- Receive Byte : 이것은 slave로부터 1 byte의 데이터를 받기 위해서 사용한다. Send byte에서 데이터 byte가 slave쪽에서 보내진다.
- Write Byte/Word : 이것은 첫번째 데이터 byte에 command를 보내고, 그 후의 데이터 byte들에 대해서는 실제 데이터를 실어서 보내는 것이다. Word를 보낼 경우에는 low byte가 먼저, 그 후에 high byte가 온다.
- Read Byte/Word : 이것은 Write Byte/Word의 slave쪽 전송을 위한 것으로 Write Byte/Word보다는 조금 더 복잡하다. 순서는 command code, A, Repeated Start bit(S), slave address, Rd, A, Data byte순으로 온다. 이때는 read의 완료를 위해서 마지막 data byte나 PEC 이후의 A bit 값으로 1을 사용한다. 이것은 NACK(Not Acknowledge)를 나타낸다.
- Process Call : 이것은 command와 관련된 데이터를 slave에 보내고, 결과를 받기를 원할 때 사용한다. 즉, 해당 command를 전달한 데이터에 대해서 실행하고, 이의 결과 값을 slave가 돌려주기를 기다리는 것이다. 순서는 S bit, Slave address, Wr, A, command code, A, byte low, byte high, A, ..., Sr(Repeated Start), slave address, Rd, A, Byte low from slave, A(master), Byte high(from slave), A(master), P의 순서이다. 이때도 마지막 A bit 앞에 PEC가 올 수 있다.
- Block Write/Read : 이것은 대량의 데이터를 write/read하기 위해서 사용한다. 앞에서 Byte/Word에서 첫번째 데이터 byte대신에, 보낼 데이터의 byte의 count값을 사용하는 것이다. 그 이후에 데이터를 나타내는 각 byte들과 ACK가 연속적으로 오는 구조이다. PEC가 붙을 경우도 있다. 전송의 끝을 나타낼 경우에는 read에 대해서 P bit앞에 오는 A bit이 1로 되어 NACK로 주도록 한다.

- Block write-block read process call : 이것은 앞에서 보았던 것들의 조합이다. Write와 Read를 합쳐서 32 bytes를 넘을 수 없으며, 먼저 write가 오고, 이후에 process call의 결과로, read가 오게 된다. 또한 Write와 Read 각각에 대해서 command byte와 전달될 데이터의 크기를 나타내는 count byte가 먼저 오게 된다. 데이터는 그 후에 byte 단위로 뒤따른다. 또한 PEC가 P bit앞에 올 수 있다.
- SMBus host notify protocol : 이것은 알려지지 않은 디바이스로부터 알려지지 않은 형식의 메시지를 받는 것을 막아주는 방법이다. Write Word를 조금 변경한 형식을 취한다. 즉, Start bit, SMB host address, Wr, A, 디바이스 주소, A, low byte, A, high byte, A, P의 순서로 동작한다. 이때 주어지는 A bit들은 전부 master에서 주어지는 것이며, 그 외의 bit이나 bytes들은 전부 slave에서 주어지는 것이다.

위에서 정한 각각의 경우에 대해서 Linux에서는 아래와 같은 값을 정해서, 각각의 경우에 대해서 사용할 전달 방법을 선택할 수 있도록 하고 있다.

```
...
#define I2C_FUNC_SMBUS_QUICK          0x00010000
#define I2C_FUNC_SMBUS_READ_BYT       0x00020000
#define I2C_FUNC_SMBUS_WRITE_BYT      0x00040000
#define I2C_FUNC_SMBUS_READ_BYT_DATA  0x00080000
#define I2C_FUNC_SMBUS_WRITE_BYT_DATA 0x00100000
#define I2C_FUNC_SMBUS_READ_WORD_DATA 0x00200000
#define I2C_FUNC_SMBUS_WRITE_WORD_DATA 0x00400000
#define I2C_FUNC_SMBUS_PROC_CALL      0x00800000
#define I2C_FUNC_SMBUS_READ_BLOCK_DATA 0x01000000
#define I2C_FUNC_SMBUS_WRITE_BLOCK_DATA 0x02000000
#define I2C_FUNC_SMBUS_READ_I2C_BLOCK 0x04000000 /* New I2C-like block */
#define I2C_FUNC_SMBUS_WRITE_I2C_BLOCK 0x08000000 /* transfer */

#define I2C_FUNC_SMBUS_BYT I2C_FUNC_SMBUS_READ_BYT | \
                         I2C_FUNC_SMBUS_WRITE_BYT
#define I2C_FUNC_SMBUS_BYT_DATA I2C_FUNC_SMBUS_READ_BYT_DATA | \
                            I2C_FUNC_SMBUS_WRITE_BYT_DATA
#define I2C_FUNC_SMBUS_WORD_DATA I2C_FUNC_SMBUS_READ_WORD_DATA | \
                             I2C_FUNC_SMBUS_WRITE_WORD_DATA
#define I2C_FUNC_SMBUS_BLOCK_DATA I2C_FUNC_SMBUS_READ_BLOCK_DATA | \
                              I2C_FUNC_SMBUS_WRITE_BLOCK_DATA
#define I2C_FUNC_SMBUS_I2C_BLOCK I2C_FUNC_SMBUS_READ_I2C_BLOCK | \
                             I2C_FUNC_SMBUS_WRITE_I2C_BLOCK

#define I2C_FUNC_SMBUS_EMUL I2C_FUNC_SMBUS_QUICK | \
                           I2C_FUNC_SMBUS_BYT | \
                           I2C_FUNC_SMBUS_BYT_DATA | \
                           I2C_FUNC_SMBUS_WORD_DATA | \
                           I2C_FUNC_SMBUS_PROC_CALL | \
                           I2C_FUNC_SMBUS_WRITE_BLOCK_DATA
...
```

코드 1257. SMBus Protocol 번호 정의

위와 같이 정의된 값들은 데이터의 전송에서 case문을 통해서 분기되며, 앞에서 보았듯이 어떤 bit(혹은 byte)를 어떻게 전달해야 하는지와 데이터를 언제 받을 것인가를 정하게 된다.

또한 보내는 message가 byte, word, 혹은 block으로 정의되는데, 이때 block의 경우에는 첫번째 byte는 count를 나타내고, 나머지 32 byte의 데이터를 가질 수 있는 구조체로 되어 있다. 아래를 보도록 하자.

```
/*
 * Data for SMBus Messages
```

```
/*
union i2c_smbus_data {
    __u8 byte;          /* 8 bit 데이터 */
    __u16 word;         /* 16 bit 데이터 */
    __u8 block[33];     /* 33 bytes 블록 데이터 : Length + Real Data */
};
```

코드 1258. i2c_smbus_data 구조체의 정의

i2c_smbus_data 구조체는 union으로 정의되어 있는데, 보내고자 하는 데이터가 byte, word, block(33 bytes)길이로 보내진다는 것을 알 수 있다. 이것은 나중에 SMBus상에 데이터를 보내기 위한 다른 자료구조를 정의하는데 자주 사용될 것이다. 자, 이젠 앞에서 하던 이야기로 다시 돌아가서, SMBus에서의 데이터 전송을 보도록 하자.

```
/* Simulate a SMBus command using the i2c protocol. No checking of parameters is done! */
static s32 i2c_smbus_xfer_emulated(struct i2c_adapter * adapter, u16 addr, unsigned short flags,
                                    char read_write, u8 command, int size, union i2c_smbus_data * data)
{
    /* So we need to generate a series of msgs. In the case of writing, we
       need to use only one message; when reading, we need two. We initialize
       most things with sane defaults, to keep the code below somewhat
       simpler. */
    unsigned char msgbuf0[34];
    unsigned char msgbuf1[34];
    int num = read_write == I2C_SMBUS_READ?2:1;
    struct i2c_msg msg[2] = { { addr, flags, 1, msgbuf0 }, { addr, flags | I2C_M_RD, 0, msgbuf1 } };
    int i;

    msgbuf0[0] = command;
    switch(size) {
        case I2C_SMBUS_QUICK:
            msg[0].len = 0;
            /* Special case: The read/write field is used as data */
            msg[0].flags = flags | (read_write==I2C_SMBUS_READ)?I2C_M_RD:0;
            num = 1;
            break;
        case I2C_SMBUS_BYTE:
            if (read_write == I2C_SMBUS_READ) {
                /* Special case: only a read! */
                msg[0].flags = I2C_M_RD | flags;
                num = 1;
            }
            break;
        case I2C_SMBUS_BYTE_DATA:
            if (read_write == I2C_SMBUS_READ)
                msg[1].len = 1;
            else {
                msg[0].len = 2;
                msgbuf0[1] = data->byte;
            }
            break;
        case I2C_SMBUS_WORD_DATA:
            if (read_write == I2C_SMBUS_READ)
                msg[1].len = 2;
            else {
                msg[0].len=3;
                msgbuf0[1] = data->word & 0xff;
            }
    }
```

```

        msgbuf0[2] = (data->word >> 8) & 0xff;
    }
    break;
case I2C_SMBUS_PROC_CALL:
    num = 2; /* Special case */
    msg[0].len = 3;
    msg[1].len = 2;
    msgbuf0[1] = data->word & 0xff;
    msgbuf0[2] = (data->word >> 8) & 0xff;
    break;
case I2C_SMBUS_BLOCK_DATA:
    if (read_write == I2C_SMBUS_READ) {
        printk("i2c-core.o: Block read not supported under "
               "I2C emulation!\n");
        return -1;
    } else {
        msg[0].len = data->block[0] + 2;
        if (msg[0].len > 34) {
            printk("i2c-core.o: smbus_access called with "
                   "invalid block write size (%d)\n",
                   msg[0].len);
            return -1;
        }
        for (i = 1; i <= msg[0].len; i++)
            msgbuf0[i] = data->block[i-1];
    }
    break;
default:
    printk("i2c-core.o: smbus_access called with invalid size (%d)\n",
           size);
    return -1;
}

```

코드 1259. i2c_smbus_xfer_emulated() 함수의 분석

i2c_smbus_xfer_emulated() 함수는 SMBus에서의 데이터 전송을 emulation하는 함수이다. 함수의 코드를 보면, size 파라미터를 통해서 넘어온 값에 따라, switch() 문에서 분기되어 실행된다. 각각의 경우에 대해서 보내고자 하는 메시지 데이터를 생성하는 역할을 이 부분에서 수행한다. 이후에는 이렇게 생성된 메시지를 보내고, 그 결과 값을 받는 과정이 될 것이다. 여기서는 Quick command, Byte write/read, Word write/read, Process call, Block Write/Read를 다루고 있으며, Block write--block read process call은 보이지 않는다. 하지만, 이것도 i2c_smbus_xfer_emulated()를 여러 차례로 나누어서 호출해 만들어 줄 수 있을 것이다. 또한 SMBus host notify protocol에 대한 것은 없다는 것을 볼 수 있을 것이다.

```

if (i2c_transfer(adapter, msg, num) < 0)
    return -1;
if (read_write == I2C_SMBUS_READ)
    switch(size) {
        case I2C_SMBUS_BYTE:
            data->byte = msgbuf0[0];
            break;
        case I2C_SMBUS_BYTE_DATA:
            data->byte = msgbuf1[0];
            break;
        case I2C_SMBUS_WORD_DATA:
        case I2C_SMBUS_PROC_CALL:
            data->word = msgbuf1[0] | (msgbuf1[1] << 8);
            break;
    }

```

```

        }
        return 0;
    }
}

```

코드 1260. i2c_smbus_xfer_emulated() 함수의 정의(계속)

실제적인 데이터의 전달을 위해서는 i2c_transfer() 함수를 사용한다. 이 함수의 복귀 코드가 0보다 작다면, 오류가 있었다는 것을 나타내며 -1을 돌려준다. 그렇지 않다면, read_write로 넘어온 코드 값에 따라서, I2C_SMBUS_READ인 경우에 대해서 읽은 데이터를 i2c_msg 구조체인 data에 넣어주도록 한다. 여기까지 문제가 없었다면 0을 돌려주어 데이터의 전송이 올바로 수행되었음을 알려준다.

```

int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg msgs[],int num)
{
    int ret;

    if (adap->algo->master_xfer) {
        DEB2	printk("i2c-core.o: master_xfer: %s with %d msgs.\n",
                   adap->name,num));
        I2C_LOCK(adap);
        ret = adap->algo->master_xfer(adap,msgs,num);
        I2C_UNLOCK(adap);
        return ret;
    } else {
        printk("i2c-core.o: I2C adapter %04x: I2C level transfers not supported\n",
               adap->id);
        return -ENOSYS;
    }
}

```

코드 1261. i2c_transfer() 함수의 정의

i2c_transfer() 함수는 실제적인 데이터의 전달을 맡는다. 넘겨받는 인수를 보면, 먼저 사용하게 될 i2c_adapter구조체의 포인터와 전달하거나 받을 데이터를 저장하기 위한 i2c_msg 구조체의 배열(msgs[]), 및 데이터의 카운트를 나타내는 num이 있다. 이 함수는 adapter에서 가지고 있는 i2c_algorithm 구조체에 master_xfer() 함수 필드가 정의되어 있는지를 확인하고, 있다면, 이 함수는 호출해서 데이터를 전달한다. 호출의 전후에는 adapter에 대한 lock을 설정하고 해제하는 일을 한다. 복귀 값은 algorithm의 master_xfer() 함수의 복귀 값을 주도록 한다. 만약 master_xfer() 함수가 정의되지 않았다면, 이 함수에서는 에러 코드로 -ENOSYS(Error No System)을 돌려준다. 따라서, 이 함수와 앞에서 본, i2c_smbus_xfer() 함수의 i2c_algorithm 구조체의 smbus_xfer() 함수 필드 중 적어도 하나는 정의되어 있어야지만 데이터를 전송할 수 있다.

```

int i2c_master_send(struct i2c_client *client,const char *buf ,int count)
{
    int ret;
    struct i2c_adapter *adap=client->adapter;
    struct i2c_msg msg;

    if (client->adapter->algo->master_xfer) {
        msg.addr     = client->addr;
        msg.flags = client->flags & I2C_M_TEN;
        msg.len = count;
        (const char *)msg.buf = buf;

        DEB2	printk("i2c-core.o: master_send: writing %d bytes on %s.\n",
                   count,client->adapter->name));
        I2C_LOCK(adap);
        ret = adap->algo->master_xfer(adap,&msg,1);
    }
}

```

```

I2C_UNLOCK(adap);
/* if everything went ok (i.e. 1 msg transmitted), return #bytes
 * transmitted, else error code.
 */
return (ret == 1)? count : ret;
} else {
    printk("i2c-core.o: I2C adapter %04x: I2C level transfers not supported\n",
           client->adapter->id);
    return -ENOSYS;
}
}

```

코드 1262. i2c_master_send() 함수의 정의

i2c_master_send() 함수는 master로 동작하는 host adapter와 같은 디바이스가 I2C Bus상의 slave 디바이스에 데이터를 전달하기 위해서 호출한다. i2c_client 구조체에 대한 포인터(client)와 보내고자 하는 데이터 가르키는 buf, 그리고, 보내고자 하는 데이터의 크기를 가지는 count를 넘겨받는다. client로부터 해당 i2c_adapter 구조체를 찾아내 adap에 두고, i2c_algorithm 구조체의 master_xfer() 함수 포인터 필드가 정의되어 있다면, 이것을 이용해서 데이터를 전달한다. 이때 메시지는 i2c_msg 구조체에 들어가게 된다. i2c_msg 구조체의 addr(디바이스의 address를 나타낸다.)에는 i2c_client 구조체가 가지고 있는 주소(address)를 넣어주게 되고, flags 필드에는 I2C_M_TEN과 client의 flags를 AND시킨 값을 넣도록 한다. 즉, 10 bit address를 사용하는 client가 있을 때는 이를 메시지에 반영하도록 하는 것이다. 보낼 데이터의 길이는 len 필드가, 마지막으로 실제 전송될 데이터는 buf를 차지한다. i2c_algorithm에 정의된 master_xfer() 함수를 호출하기 전에 i2c_adapter 구조체에 대한 lock을 설정하는 것을 빼먹어선 안될 것이다. 즉, 데이터가 전송중인 중간에는 i2c_adapter 구조체에 변경이 있어서는 안될 것이다. 제대로 전송이 완료되었다면, master_xfer() 함수의 복귀 값은 1이 될 것이며, 이때는 보낸 데이터의 크기를 돌려준다. 그렇지 않다면, master_xfer() 함수의 복귀 값을 그대로 돌려준다. 만약 master_xfer()이 정의되어 있지 않다면, 당연히 -ENOSYS(Error No System)을 돌려줄 것이다.

```

int i2c_master_recv(struct i2c_client *client, char *buf ,int count)
{
    struct i2c_adapter *adap=client->adapter;
    struct i2c_msg msg;
    int ret;

    if (client->adapter->algo->master_xfer) {
        msg.addr    = client->addr;
        msg.flags = client->flags & I2C_M_TEN;
        msg.flags |= I2C_M_RD;
        msg.len = count;
        msg.buf = buf;

        DEB2	printk("i2c-core.o: master_recv: reading %d bytes on %s.\n",
                   count,client->adapter->name));
        I2C_LOCK(adap);
        ret = adap->algo->master_xfer(adap,&msg,1);
        I2C_UNLOCK(adap);
        DEB2	printk("i2c-core.o: master_recv: return:%d (count:%d, addr:0x%02x)\n",
                   ret, count, client->addr));
        /* if everything went ok (i.e. 1 msg transmitted), return #bytes
         * transmitted, else error code.
         */
        return (ret == 1)? count : ret;
    } else {
        printk("i2c-core.o: I2C adapter %04x: I2C level transfers not supported\n",
               client->adapter->id);
    }
}

```

```

        return -ENOSYS;
    }
}

```

코드 1263. i2c_master_recv() 함수의 정의

i2c_master_recv() 함수는 master가 데이터를 받고자 할 때 호출한다. 같은 i2c_algorithm 구조체의 master_xfer() 함수를 사용하지만, 메시지의 flags 필드에 I2C_M_RD를 사용해서 read로 master_xfer() 함수를 호출한다는 점이 다르다. 나머지는 i2c_master_send() 함수와 동일하다.

```

int i2c_control(struct i2c_client *client,
                unsigned int cmd, unsigned long arg)
{
    int ret = 0;
    struct i2c_adapter *adap = client->adapter;

    DEB2(printh("i2c-core.o: i2c ioctl, cmd: 0x%x, arg: %#lx\n", cmd, arg));
    switch (cmd) {
        case I2C_RETRIES:
            adap->retries = arg;
            break;
        case I2C_TIMEOUT:
            adap->timeout = arg;
            break;
        default:
            if (adap->algo->algo_control!=NULL)
                ret = adap->algo->algo_control(adap,cmd,arg);
    }
    return ret;
}

```

코드 1264. i2c_control() 함수의 정의

i2c_control() 함수는 일종의 일반적인 디바이스 드라이버의 IOCTL 메소드와 동일한 역할을 수행한다. 즉, i2c_control() 함수는 특정 adapter에 설정된 것을 변경하고자 할 때 사용할 수 있다. 이곳에서는 기본적으로 adapter level에서 처리되는 두개의 command가 있음을 알 수 있다. 즉, I2C_RETRIES와 I2C_TIMEOUT이 바로 그것이다. I2C_RETRIES는 adapter에서 데이터를 전송할 때 retry를 실행할 회수를 변경하기 위해서 사용하는 것이며, I2C_TIMEOUT은 timeout interval을 다시 설정하기 위해서 사용한다. 그 외의 경우에 대해서는 i2c_algorithm 구조체에 정의된 algo_control() 함수를 호출해서 처리하고 있다. 복귀 값은 I2C_RETRIES와 I2C_TIMEOUT에 대해서는 0이며, 그 외의 경우에 대해서는 algo_control() 함수가 정의된 경우에 대해서 호출 결과를 돌려준다.

앞에서는 i2c-core.c의 I2C 관련 함수들에 대해서 보았다. 여기서부터는 I2C Bus의 SMBus와 관련된 데이터의 처리를 어떻게 하는지를 볼 것이다. 앞에서 본 SMBus network layer에서 사용하는 protocol의 구현을 다루게 될 것이다. 각각의 함수들에 대해서 하나씩 보도록 하자.

```

/* The SMBus parts */
extern s32 i2c_smbus_write_quick(struct i2c_client * client, u8 value)
{
    return i2c_smbus_xfer(client->adapter,client->addr,client->flags,
                          value,0,I2C_SMBUS_QUICK,NULL);
}

```

코드 1265. i2c_smbus_write_quick() 함수의 정의

i2c_smbus_write_quick() 함수는 Quick Command를 처리하기 위한 것이다. i2c_smbus_xfer() 함수⁴⁸³를 호출해서 처리하며, 넘겨주는 command로는 I2C_SMBUS_QUICK을 준다. i2c_smbus_xfer() 함수를 호출하기 위해서는 client가 사용하는 adapter의 포인터와 client의 주소, 현재 client의 상태를 나타내는 flags, read/write를 나타내는 값, command(여기서는 command로 사용할 값을 넣었다.), size(command의 크기는 알 필요가 없으므로 0을 사용한다.), 전송될 데이터의 크기를 나타내는 size, 그리고, 마지막으로 실제 데이터를 가지는 buffer에 대한 포인터인 data로 이루어져 있다.

```
/* smbus_access read or write markers */
#define I2C_SMBUS_READ 1
#define I2C_SMBUS_WRITE 0

/* SMBus transaction types (size parameter in the above functions)
   Note: these no longer correspond to the (arbitrary) PII4 internal codes! */
#define I2C_SMBUS_QUICK 0
#define I2C_SMBUS_BYTE 1
#define I2C_SMBUS_BYTE_DATA 2
#define I2C_SMBUS_WORD_DATA 3
#define I2C_SMBUS_PROC_CALL 4
#define I2C_SMBUS_BLOCK_DATA 5
#define I2C_SMBUS_I2C_BLOCK_DATA 6
```

코드 1266. i2c_smbus_xfer() 함수에 사용되는 파라미터 값 정의

i2c_smbus_xfer에서는 read나 write를 위해서 I2C_SMBUS_READ/I2C_SMBUS_WRITE라는 것을 통해서 나타내며, 전달될 데이터를 크기를 나타내기 위해, I2C_SMBUS_XXXX_DATA와 같은 것들을 사용한다. 위에서 보여주는 i2c_smbus_xfer() 함수에 사용되는 파라미터 값은 앞으로도 계속 사용될 것이기에 기억하도록 하자.

```
extern s32 i2c_smbus_read_byte(struct i2c_client * client)
{
    union i2c_smbus_data data;
    if (i2c_smbus_xfer(client->adapter,client->addr,client->flags,
                        I2C_SMBUS_READ,0,I2C_SMBUS_BYTE, &data))
        return -1;
    else
        return 0x0FF & data.byte;
}
```

코드 1267. i2c_smbus_read_byte() 함수의 정의

i2c_smbus_read_byte()는 I2C_SMBUS_READ를 통해서 read를 한다는 것을 나타내고, 전달될 command는 없으므로 0을, 데이터의 크기는 byte 이므로, I2C_SMBUS_BYTE를 사용했다. 실제 전달될 데이터는 data가 가르키고 있다.

```
extern s32 i2c_smbus_write_byte(struct i2c_client * client, u8 value)
{
    return i2c_smbus_xfer(client->adapter,client->addr,client->flags,
                          I2C_SMBUS_WRITE,value, I2C_SMBUS_BYTE,NULL);
}
```

코드 1268. i2c_smbus_write_byte() 함수의 정의

⁴⁸³ i2c_smbus_xfer() 함수는 앞에서 이미 보았듯이, i2c_algorithm 구조체의 smbus_xfer() 함수 필드가 정의된 경우에는 그것을 호출하고, 그렇지 못할 때는 i2c_smbus_xfer_emulated() 함수를 호출한다.

i2c_smbus_write_byte() 함수는 I2C_SMBUS_WRITE를 사용하여, write할 데이터를 value가 가지고 있고, I2C_SMBUS_BYTEx로 byte를 write한다는 것을 의미한다. 여기서 주의할 것은 앞에서는 데이터를 보관할 목적으로 i2c_smbus_data의 포인터를 사용했지만, 여기서는 command대신에 전달할 데이터 byte를 사용하고 있다는 것이다.

```
extern s32 i2c_smbus_read_byte_data(struct i2c_client * client, u8 command)
{
    union i2c_smbus_data data;
    if (i2c_smbus_xfer(client->adapter,client->addr,client->flags,
                        I2C_SMBUS_READ,command,I2C_SMBUS_BYTE_DATA,&data))
        return -1;
    else
        return 0x0FF & data.byte;
}
```

코드 1269. i2c_smbus_read_byte_data() 함수의 정의

i2c_smbus_read_byte_data() 함수는 command를 보내서 데이터를 얻는 경우가 된다. I2C_SMBUS_READ를 read/write에 사용하고, command는 그대로 전달하며, 읽고자 하는 데이터의 크기가 byte이므로, I2C_SMBUS_BYTEx를, 읽은 데이터는 data에 저장한다. 복귀 값은 오류가 있을 경우에는 -1을 그렇지 않을 경우에는 byte값 만을 돌려주도록 한다.

```
extern s32 i2c_smbus_write_byte_data(struct i2c_client * client, u8 command, u8 value)
{
    union i2c_smbus_data data;
    data.byte = value;

    return i2c_smbus_xfer(client->adapter,client->addr,client->flags,
                          I2C_SMBUS_WRITE,command,
                          I2C_SMBUS_BYTE_DATA,&data);
}
```

코드 1270. i2c_smbus_write_byte_data() 함수의 정의

i2c_smbus_write_byte_data() 함수는 I2C_SMBUS_WRITE로 write한다는 것을 나타내며, write할 command는 넘겨받은 것을 그대로 사용하고, write할 데이터의 크기를 byte로 명시하기 위해 I2C_SMBUS_BYTEx_DATE를 쓰고 있다. 또한 write할 데이터 자체는 i2c_smbus_data 구조체를 가지는 data의 byte에 넣어서 보내고 있다.

```
extern s32 i2c_smbus_read_word_data(struct i2c_client * client, u8 command)
{
    union i2c_smbus_data data;
    if (i2c_smbus_xfer(client->adapter,client->addr,client->flags,
                        I2C_SMBUS_READ,command,I2C_SMBUS_WORD_DATA, &data))
        return -1;
    else
        return 0xFFFF & data.word;
}
```

코드 1271. i2c_smbus_read_word_data() 함수의 정의

i2c_smbus_read-word_data는 word(=16 bits) 데이터를 읽기 위해서 사용한다. 따라서, I2C_SMBUS_READ가 들어가며, command는 그대로 사용하고, 데이터의 크기를 I2C_SMBUS_WORD_DATA로 둔다. 데이터를 읽어서 저장하는 곳은 i2c_smbus_data구조체인 data의 포인터를 사용하고 있다. 넘어온 데이터의 word부분만을 돌려주도록 0xFFFF와 AND시킨 값을 복귀 값으로 사용한다. 오류가 있었다면 -1을 돌려줄 것이다.

```
extern s32 i2c_smbus_write_word_data(struct i2c_client * client, u8 command, u16 value)
{
    union i2c_smbus_data data;
    data.word = value;

    return i2c_smbus_xfer(client->adapter,client->addr,client->flags,
                          I2C_SMBUS_WRITE,command,
                          I2C_SMBUS_WORD_DATA,&data);
}
```

코드 1272. i2c_smbus_write-word_data() 함수의 정의

i2c_smbus_write_word_data() 함수는 보내고자 하는 데이터의 크기가 16 bit(= word)인 경우에 사용한다. I2C_SMBUS_WRITE를 사용해서 write한다는 것을 알고, command는 그대로 사용하도록 한다. 데이터의 크기는 word이므로, I2C_SMBUS_WORD_DATA를, 보낼 데이터는 i2c_smbus_data 구조체인 data에 넣어서(value) 포인터를 넘겨준다.

```
extern s32 i2c_smbus_process_call(struct i2c_client * client, u8 command, u16 value)
{
    union i2c_smbus_data data;
    data.word = value;
    if (i2c_smbus_xfer(client->adapter,client->addr,client->flags,
                        I2C_SMBUS_WRITE,command,
                        I2C_SMBUS_PROC_CALL, &data))
        return -1;
    else
        return 0xFFFF & data.word;
}
```

코드 1273. i2c_smbus_process_call() 함수의 정의

i2c_smbus_process_call() 함수는 특정 디바이스에 command와 데이터를 보낸 후, 그 결과를 얻고자 할 때 사용한다. 따라서, I2C_SMBUS_WRITE를 사용하고 있으며, command는 그대로 보낸다. 데이터의 크기 필드를 나타내는 곳에는 I2C_SMBUS_PROC_CALL을 써서, 이것이 process call을 위한 데이터임을 나타내도록 하며, 실제 보낼 데이터는 i2c_smbus_data 구조체인 data에 넣도록(value) 한다. 나중에 처리된 후에 읽은 데이터는 다시 data에 있을 것이므로, 이중에서 word만을 취해서 넘겨준다. 오류가 있었다면, 당연히 -1을 돌려줄 것이다.

```
extern s32 i2c_smbus_read_block_data(struct i2c_client * client, u8 command, u8 *values)
{
    union i2c_smbus_data data;
    int i;

    if (i2c_smbus_xfer(client->adapter,client->addr,client->flags,
                        I2C_SMBUS_READ,command,
                        I2C_SMBUS_BLOCK_DATA,&data))
        return -1;
    else {
        for (i = 1; i <= data.block[0]; i++)
            values[i-1] = data.block[i];
        return data.block[0];
    }
}
```

코드 1274. i2c_smbus_read_block_data() 함수의 정의

i2c_smbus_read_block_data() 함수는 블록 데이터를 읽기 위한 것이다. block데이터의 최대 크기는 32 byte를 넘을 수 없다. 먼저 읽기를 나타내기 위해서 I2C_SMBUS_READ를 사용하고, command를 그대로 주고 있으며, I2C_SMBUS_BLOCK_DATA로 읽을 데이터가 블록 데이터임을 나타낸다. 또한 읽은 데이터를 i2c_smbus_data 구조체인 data에 저장했다가, 이를 i2c_smbus_data구조체가 블록 데이터를 가르키게 될 때, 데이터의 길이로 사용하는 첫번째 byte를 이용해서 value필드에 복사해 둔다. 오류가 있었을 때는 -1을, 그렇지 않다면 읽은 데이터의 길이를 돌려주도록 한다.

```
extern s32 i2c_smbus_write_block_data(struct i2c_client * client, u8 command, u8 length, u8 *values)
{
    union i2c_smbus_data data;
    int i;

    if (length > 32)
        length = 32;
    for (i = 1; i <= length; i++)
        data.block[i] = values[i-1];
    data.block[0] = length;
    return i2c_smbus_xfer(client->adapter,client->addr,client->flags,
                          I2C_SMBUS_WRITE,command,
                          I2C_SMBUS_BLOCK_DATA,&data);
}
```

코드 1275. i2c_smbus_write_block_data() 함수의 정의

i2c_smbus_write_block_data() 함수는 블록 데이터를 client 디바이스에 쓰기를 하기 위해서 사용한다. 역시 쓰려고 하는 블록 데이터의 크기는 32 bytes이상 될 수 없다. 이 데이터는 values가 가르키게 되며, values로부터 i2c_smbus_data 구조체를 가지는 data로 복사한다. 이때 길이는 data의 첫번째 byte에 들어가도록 한다.

```
extern s32 i2c_smbus_write_i2c_block_data(struct i2c_client * client, u8 command, u8 length, u8 *values)
{
    union i2c_smbus_data data;
    int i;
    if (length > 32)
        length = 32;
    for (i = 1; i <= length; i++)
        data.block[i] = values[i-1];
    data.block[0] = length;
    return i2c_smbus_xfer(client->adapter,client->addr,client->flags,
                          I2C_SMBUS_WRITE,command,
                          I2C_SMBUS_I2C_BLOCK_DATA,&data);
}
```

코드 1276. i2c_smbus_write_i2c_block_data() 함수의 정의

i2c_smbus_write_i2c_block_data() 함수는 i2c_smbus_write_block_data()와 동일하나, write할 데이터의 길이를 나타내는 I2C_SMBUS_I2C_BLOCK_DATA를 사용한다는 점에서 차이가 난다. 이 함수는 앞에서 본 block write 함수와는 달리 smbus상에서 I2C bus의 block 데이터를 write하기 위해서 사용한다.

[To be continued here!!!]

20. Real-Time Linux(RTLinux)

이번장에서는 Linux상에서 Real-Time을 구현할 수 있는 방법의 한 가지로, RTLinux에 대한 설명을 하고자 한다. 먼저 embedded system과 real-time system이 어떤 면에서 다르며, 어떤것을 embedded system 혹은 real-time system이라고 하는지를 알아 보겠다.

20.1. Embedded System VS. Realtime System

Embedded system이란 무엇인가? Embedded system에 대한 정의는 여러 곳에서 찾을 수 있겠지만, 여기서는 다음과 같은 말로 표현하도록 하겠다.

“An embedded system is a microprocessor-based system that is built to control a function or range of functions and is not designed to be programmed by the end user in the same way that a PC is.”⁴⁸⁴

즉, 이 말은 PC와 같은 사용자(end user)의 programming이 되는 것을 목적으로 하지 않는 시스템을 통칭해서 쓰는 말이다. Embedded system은 이미 특정한 목적으로 만들어져서 사용자에게 제공되기 때문에, 사용자가 특별히 프로그램해서 사용할 일이 없는 시스템을 말한다.

Real-time system이란 다시 아래와 같은 말로 정의될 수 있을 것이다. 여기서, 정의한 내용은 “The Oxford Dictionary of Computing”에 있는 것을 인용한 것이다.

“Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to the same movement. The lag from input to output time must be sufficiently small for acceptable timeliness.”⁴⁸⁵

즉, 시스템에서 input을 받아서 output을 내는데 있어서, 연산의 결과도 중요하지만, 결과가 나오는 시간도 중요하다는 말이다. 따라서, 어떤 system이 실시간 시스템이라고 말할 때는 특정한 주어진 시간 안에 원하는 결과를 도출해줄 수 있다는 것을 말한다. 만약 그 시간이 지난 후에 어떤 결과(즉, 정확한 결과가 나오더라도 인정할 수 없다는 말이다.

대체로 embedded system이라고 말하면, real-time 시스템이라고 생각하면 될 것이지만, 현재 개발되고 있는 시스템들은 엄밀하게 말하면, 완전한 real-time 시스템이라고는 할 수 없는 것들이 많으며, embedded 시스템을 platform으로 가져가고 있음을 알 수 있을 것이다. 즉, 시간적인 제약은 줄어들고 있으며, 기능(function)이 다양한 embedded system이 중요성을 더해가고 있다. 따라서, 다음과 같이 embedded system의 의미를 확장하도록 하겠다. “Embedded system은 사용자의 직접적인 programming을 요구하지 않으며, 시간에 따라 연산의 결과값의 유효성이 감소하는 시스템이다. 이와 같은 시스템의 대표적인 것이 multimedia를 재생해주는 시스템 같은 것들이 될 수 있을 것이다. Multimedia의 경우 화상이나 음성이 사람에게 전달되는 시간에 대해서 민감하므로 제대로 재생이 되지 않을 경우에는 화질의 열화나 음성의 동기화가 잘 맞지 않게 된다. 따라서, 이러한 시스템은 시간의 흐름에 따라 결과의 유효성이 감소한다고 볼 수 있다.

20.2. What is RTLinux?

일반적으로 Linux는 hard real-time을 요하는 시스템에서는 적합하지 않다. 이는 Linux 커널이 hard real-time 응용 프로그램에 대한 지원을 고려해서 설계된 시스템이 아니라, 일반적인 desktop 응용 프로그램을 수행할 목적으로 time-sharing에 기반하기 때문이다. 즉, 시스템을 사용하는데 있어서 균등한 배분만을

⁴⁸⁴ 이 말은 Steve Heath의 “Embedded Systems Design, Newnes, 1997.”의 1 page에서 인용한 것이다.

⁴⁸⁵ 이 말이 인용된 책은 앞에서 밝힌 것과 같지만, 이것을 인용하고 있는 책은 Rick Grehan, Robert Moote, Ingo Cyliax가 쓴 “Real-Time System Programming”, Addison-Wesley, 1998.”의 9 page이다.

고려했지, hard real-time 시스템과 같은 deadline이 앞선 우선순위가 높은 task(or process)⁴⁸⁶로의 kernel내에서 preemption을 지원하지 않기 때문이다. Linux에서 지원하는 scheduling policy에는 아래와 같은 것이다.

- SCED_FIFO : FIFO 방식으로 task를 scheduling하는 방법이다. Real-time task를 위한 scheduling policy이며, 우선 순위가 높은 실시간 프로세스는 자신이 blocking되거나 CPU를 양보(yield)하기 전에는 계속 수행되는 것이 보장된다.
- SCED_RR : Round-Robin 방식으로 task를 scheduling하는 방법이다. Real-time task를 위한 scheduling policy이며, 우선 순위가 높은 실시간 프로세스가 자신이 가진 quantum동안만 수행되며, 이 quantum을 다 소비하고 나면, task의 run queue의 마지막으로 이동하게 된다. 즉, 공정하게 이 scheduling policy를 가진 프로세스간에 CPU를 공유하도록 하는 것이다.
- SCED_OTHER : 앞에서 설명한 real-time task가 없는 경우에 수행되며, 우선 순위는 CPU를 사용한 것에 비례해서 낮아지게 된다. 또한, 프로세스는 자신이 가진 quantum을 다 소비하게 되면, 다른 task를 scheduling할 수 있도록 만들어준다. 일반적으로 Linux상에서 동작하는 대부분의 응용프로그램들은 이것을 이용해서 scheduling이 될 것이다.

위에서 real-time process에 대한 scheduling policy를 Linux가 가지고 있다고 이야기했는데, 이는 Linux가 hard real-time을 만족시키는 것이 아니라, soft real-time을 지원한다는 말이다. 즉, 커널 내에서는 preemption이 되지 않기에 우선 순위가 더 높은 real-time task가 있다손 치더라도, 그 프로세스가 곧바로 CPU를 차지하지는 못한다. 단지, 실시간 scheduling class에 속한 프로세스가 다른 프로세스들 보다 우선적으로 스케줄링이 되고, 우선 순위의 변동이 없단는 것만을 제공할 뿐이다. 따라서, 실시간 event가 들어오더라도 곧바로 그 실시간 process를 실행 시켜 줄 수 있는 것은 아니다. 결과적으로 delay가 발생한다고 생각하면 된다.

이러한 Linux가 가지는 특성을 Real-Time Linux(RTLinux)⁴⁸⁷에서는 우회하는 방법을 사용하고 있다. 즉, 커널이 hard real-time process를 직접적으로 지원하는 것은 Linux 커널 전체를 새로이 디자인하는 만큼의 노력이 들기 때문에, Linux 커널을 하나의 real-time task로 보고, 다른 real-time task가 없을 경우에만 Linux를 수행하도록 만드는 방법을 사용하고 있는 것이다. 이것은 실질적으로 하나의 컴퓨터 시스템에 두개의 커널을 동작시키는 방법으로 hard real-time task에 우선 순위를 둔다는 것이다. 이렇게 함으로서 Linux 커널을 수정해주어야 하는 노력을 최소화 하고, 동시에 hard real-time을 구현할 수 있는 방법을 제시하는 것이 RTLinux가 생각한 방법이다. RTLinux가 취하고 있는 디자인 철학은 다음과 같이 요약해 볼 수 있다.

“Real-time programs should be split into small and simple parts with hard real-time constraints, and the larger parts that do more sophisticated processing”

즉, 응급한 hard real-time 특성을 가지는 프로그램을 작게 유지하고, 나머지 데이터에 대한 복잡한 처리는 특별한 인터페이스를 통해서 Linux task로 전달되어 처리되도록 만들겠다는 것이다. 여기서 사용되는 것이 바로 **real-time FIFO**이다. 이 real-time FIFO를 통해서 real-time task와 일반 Linux task가 상호 작용할 있는 인터페이스를 가지게 된다.

RTLinux에서 real-time쪽을 차지하는 소프트웨어 컴포넌트들은 Linux 커널이 제공하는 module로서 동적으로 loading되어 커널과 link된다.⁴⁸⁸ 이 모듈 내에서 RTLinux를 위한 각종 데이터구조체를

⁴⁸⁶ 이하에서는 task를 process와 같은 의미로 사용하도록 하겠다. 실제로 Linux에서는 모든 thread가 일반적인 process와 동일하게 구성된다. 다만, thread가 사용하는 address space가 프로세스와 동일하다는 점만이 프로세스와 구분되는 점이다. 하지만, 이는 thread를 관리하는 mechanism에서는 별반 문제시 되지 않는다.

⁴⁸⁷ 이하에서는 Real-Time Linux를 RTLinux로 쓰도록 하겠다.

⁴⁸⁸ 실제로 커널과의 link를 위해서 커널의 RTLinux patch를 적용할 때 이러한 부분을 null pointer로 만들어둔다. 나중에 직접적으로 코드를 볼때 검토하도록 하겠다.

정의하거나 header파일을 include하게 되며, 그 중 중요한 것으로는 `rt_fifo.h`와 `rt_sched.h` 및 `RT_TASK` 구조체가 있다.⁴⁸⁹

실제적으로 real-time FIFO는 real-time task가 Linux 프로세스와 통신할 수 있는 유일한 인터페이스 역할을 하며, 또한 real-time task가 데이터를 읽거나 쓸 때 생길 수 있는 blocking 문제를 해결할 목적으로 사용하고 있다. 전체적인 RTLinux의 구조는 아래의 [그림 139]과 같다.

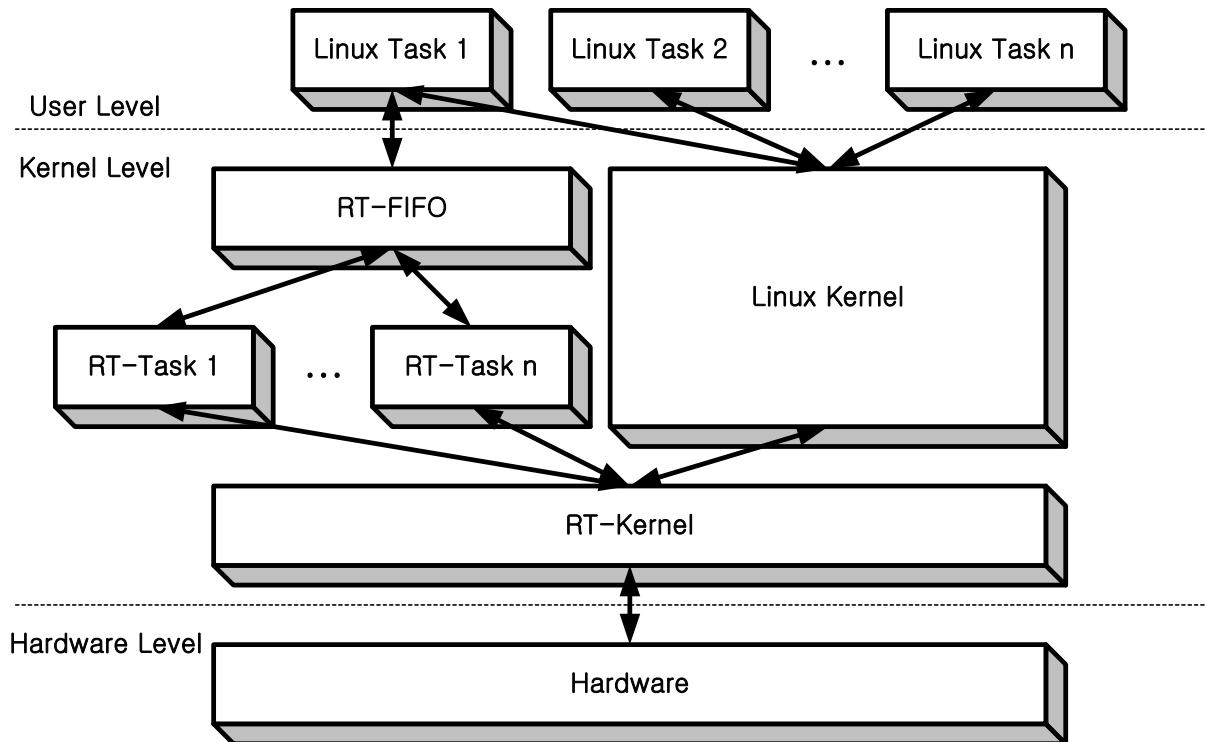


그림 139. RTLinux의 Architecture

[그림 139]에서 보듯이 hardware에서 생성된 event는 RTLinux의 커널(RT-Kernel)로 전달되며, 만약 이것이 Real-time Task(RT-Task)가 원하는 것이라면, RT-Task를 scheduling해서 데이터를 Real-time FIFO(RT-FIFO)에 저장하게 되며, 다시 Linux 커널을 scheduling해서 사용자 레벨의 Linux task가 RT-FIFO에서 읽어갈 수 있도록 하고 있는 것을 알 수 있다. RT-Kernel에 있는 scheduler 자체도 역시 하나의 모듈로서 loading된다. RTLinux 2.0에서는 이 scheduler로 주기적으로 RT-Task를 실행하는 역할을 하며, 또 다른 한 방법으로는 hardware의 interrupt가 발생했을 경우에만 RT-task를 실행시켜주는 scheduler가 있을 수 있다.

여기서 잠시 Linux 커널에서 제공하는 module loading과 unloading을 살펴보도록 하자. 일반적으로 module로서 loading이 되는 것은 디바이스 드라이버들이다. 그 외에도 파일 시스템과 같은 것이 module로서 loading될 수 있으며, loading이 되는 순간 커널과 link가 일어나게 된다. 따라서, 커널에서 제공하는 API⁴⁹⁰를 사용할 수 있게 된다. 또한 커널도 이 모듈에서 제공하는 API 함수들을 접근해서 호출할 수 있게 된다. 모듈을 loading하기 위해서는 `module_init()`/`module_exit()`⁴⁹¹ 매크로를 이용해서 정의된 entry point를 가진다. 모듈은 내부적인 자료구조를 가질 수 있으며, process로서 생성되는 것이

⁴⁸⁹ 물론 이것은 RTLinux 2.0 버전에서의 이야기이며, 현재는 모든 header 파일의 정의가 `rtl_XXX.h`로 되어 있다. 현재 이 문서가 참고하고 있는 document는 RTLinux 2.0 version이며, 이것만으로도 충분한 RTLinux에 대한 이해가 가능하다고 보기 때문이다. 실제적인 구현을 볼 때는 RTLinux Pro 3.0 버전과 Linux 커널 2.4.7에 대한 modification을 보도록 할 것이다.

⁴⁹⁰ 이 API는 Linux 커널에서 제공하는 system call과는 다른 것이다. 즉, 커널에서 export한 함수들을 말한다.

⁴⁹¹ 이 macro들은 Linux 커널 2.4버전에서 나온 것이다.

아니라, 일종의 함수로서 커널 내에 존재하게 된다. 따라서, `module_init()`와 같은 곳에서 RT-Task의 초기화를 위한 일이 수행될 것이며, `module_exit()`와 같은 곳에서 그 역할을 수행할 것이다.

기본적으로 RTLinux가 수행되기 위해서 필요로 하는 모듈로는 다음과 같은 것들이 있다. 각각이 하는 기능 역할과 같이 보도록 하자.

- `rtl.o` : RTLinux의 핵심 core를 이루는 모듈로서 interrupt handling을 담당한다.
- `rtl_time.o` : RTLinux의 timer에 관련된 모듈로서 real time clock을 관리한다.
- `rtl_posixio.o` : RTLinux의 POSIX I/O standard interface 역할을 수행하는 모듈이다. 옵션으로 동작한다.
- `rtl_fifo.o` : RTLinux와 Linux process와의 interface를 담당하는 모듈로 FIFO를 구현한다. Linux process에서는 /dev/rtf 디바이스로 입출력을 요청할 수 있다.
- `rtl_sched.o` : RTLinux의 scheduling을 담당하는 모듈이다. RTLinux task와 Linux task로 구성된 task간의 scheduling을 담당한다.
- `rtl_debug.o` : RTLinux의 디버깅 모듈이다.
- `psdd.o` : RTLinux의 user space pthread API interface를 관리하는 모듈이다.⁴⁹²

기본적으로 이와 같은 모듈들이 patch된 Linux 커널과 같이 RTLinux를 수행하기 위해서 올라가야 한다. 그 외의 RT-task들은 일반적으로 커널에 module을 삽입하는 것과 같이 수행된다.

20.2.1. Open RTLinux와 RTLinux Pro

Open RTLinux는 공개된 버전이며, RTLinux Pro는 FSMLab에서 사야지만 사용할 수 있는 것이다. 둘간의 기능적인 차이는 없지만, Open RTLinux보다는 RTLinux Pro가 지원하는 기능면에서나 안정성 측면에서 더 나은 기능을 제공한다. 하지만, 대부분의 기능은 동일하게 동작하기에 별다른 차이는 없을 것이다. 이곳에서 설명하는 코드들은 대부분이 Open RTLinux에 속하는 것들이다. Source code의 download는 다음과 같은 site에서 처리할 수 있다.

<ftp://www.rtlinux.com/pub/rtlinux/v3/rtlinux-3.0.tar.gz>
<ftp://www.rtlinux.com/pub/rtlinux/v3/rtldoc-3.0.tar.gz>

관련된 kernel은 <ftp://ftp.kernel.org>에서 받기 바란다. 저자가 RTLinux 커널을 test할 당시의 버전은 Linux 커널 2.4.2 버전이었으며, 현재 안정된 버전으로는 2.4.17이 나온 것으로 확인했다.

앞에서 이미 조금 이야기를 했지만, RTLinux에서 제공하는 module이 Open RTLinux와 RTLinux Pro간에 차이가 있다. 이를 요약하면 다음과 같다.

1. Open RTLinux에서 제공하는 module : `rtl.o`, `rtl_time.o`, `rtl_sched.o`, `rtl_posix.o`, `rtl_fifo.o`, `psc.o`, `rtl_debug.o`등은 Open RTLinux에서 기본적으로 제공하는 모듈들이다.
2. RTLinux Pro에서 제공하는 module : `psdd.o`, `mmic.o`, `Inet_X.o`등은 RTLinux Pro에서만 제공하는 module들이다.
3. 이외에 contribution으로 제공하는 module : `mbuff.o` (Thomas Montylewsky), `rt_com.o` (Jochen Kuepper)등의 모듈들이다.

앞에서 본 모듈들 이외에 `mmic.o`는 Manufacturing Management Information and Control의 약자를 취한 것으로 RTLinux 프로그램내의 각종 parameter 변화를 일으키기 위한 것이며, 각종 scripting 언어내에 embedded 될 수 있다. 또한 관련된 모듈로는 `mmic_intercessor`은 MMIC module로부터 상태 정보를 얻기위한 interface를 하는 데몬(daemon)이며, RTLinux에서 발생하는 모든 경고 메시지를 프린트 해준다. 추가적인 기능으로는 RTLinux 프로그램들에 의해서 요청되는 shell command를 실행하는 역할도 수행해 준다.

⁴⁹² 여기서, 보여준 module들은 RTLinuxpro의 scripts 디렉토리 이하의 insrtl script를 통해서 올라오는 것들이다.

20.2.2. MiniRTL

MiniRTL은 RTLinux을 embedded system에 적합한 형태로 구현한 것이다. 먼저 MiniRTL은 FSMLab의 Austria에서 Nicholas McGuire에 의해서 만들어진 것으로 1.44 Mbytes의 floppy disk에서 booting이 가능하다. 대부분의 표준적인 RTLinux의 기능에 대해서 호환되며, Linux에 대해서도 마찬가지다. 또한 개발하기 위해서 특별한 tool이 필요한 것도 아니다. 현재는 Linux 커널 2.2.X에서 test중이며, RTLinux 3.0, 3.1에서 test하고 있다. 사용할 수 있는 가장 최근버전의 GNU C Library는 2.0.7이나 2.1.3이며, 대부분의 hardware를 다 지원한다.

지원하고 있는 네트워크 protocol로는 TCP/UDP, IP, ICMP등이며, telnet, tftp, ssh, scp, syslog, SMTP등의 application을 사용할 수 있으며, CGI를 지원하는 mini-http와 같은 web server도 가진다. DNS, DHCP, NTP와 같은 네트워크 service도 사용할 수 있다.

MiniRTL에서 사용할 수 있는 옵션으로는 네트워크 service로 routed, named, NFS등이 있으며, 사용자 프로그램들이나 기타 library들이 있을 수 있다. 또한 MiniRTL 역시 core가 되는 구성 요소들이 모듈로서 존재하기에 사용하고자 하는 모듈만 필요에 따라 로딩/loading)하는 것이 가능하다.

대략 RTLinux가 어떤 것이며, 어떤 module로서 구성되었는지를 보았다. 이하에서는 RTLinux의 핵심으로 들어가 RTLinux가 구동되는 원리와 어떤 방식의 구현을 가지는지를 좀더 자세히 보도록 하겠다. 먼저 RTLinux를 설치하는 방법부터 보도록 하자.

20.3. RTLinux의 설치

RTLinux를 실행하기 위해서는 반드시 Linux 커널을 patch해야 한다. 즉, 원래의 커널에는 RTLinux에 대한 고려가 전혀 없기에 이 과정이 필요하다. Patch하는 부분도 작으면 작을수록 좋을 것이다. 이를 위해서 RTLinux의 실제 코드를 잠시 보도록 하자. 주로 patch의 대상이 되는 디렉토리는 /usr/src/linux/arch/xxx/kernel이며, entry.S 파일이다. 이 파일의 주목적은 interrupt나 system call등에 대한 처리를 다루는 것이다. 따라서, RTLinux가 Linux 커널보다 먼저 제어를 넘겨받기에 좋은 곳이다.

```
* (c) Victor Yodaiken 1999
* RTLINUX_IRET emulates the turn on interrupts effect of
* iret since, under RTLinux we may have pended interrupts
* that will not be automatically enabled on an iret.
*/
#ifndef CONFIG_RTLINUX
#define RTLINUX_IRET \
    movl rtl_emulate_iret,%eax; \
    testl %eax, %eax; \
    je 1f; \
    call *%eax; \
1:
#else
#define RTLINUX_IRET
#endif

#ifndef CONFIG_RTLINUX
#define RTLINUX_CLI \
    movl irq_control+8,%eax; \
    call *%eax; \
1:
#else
#define RTLINUX_CLI cli
#endif
```

코드 1277. RTLinux의 IRET(Interrupt Return)과 CLI(Clear Interrupt) 코드

여기서 보여주고자 하는 것은 RTLinux가 설정에 따라서, 어떤 행동을 취하기 위해서 어떤 patch를 가하고 있는가이다. 코드를 보면, CONFIG_RTLINUX의 유무에 따라서, 두가지의 macro로

RTLINUX_IRET와 RTLINUX_CLI를 정의하고 있다. 아래의 코드는 이러한 macro가 어떻게 사용되는지를 보여준다.

```

ENTRY(system_call)
    pushl %eax                      # save orig_eax
    SAVE_ALL
#endif CONFIG_RTILINE
    movl rtl_syscall_intercept,%ebx
    testl %ebx, %ebx
    je 991f
    call *%ebx
    movl ORIG_EAX(%esp), %eax
991:
#endif
    GET_CURRENT(%ebx)
    cmpl $(NR_syscalls),%eax
    jae badsys
    testb $0x02,tsk_ptrace(%ebx)# PT_TRACESYS
    jne tracesys
    call *SYMBOL_NAME(sys_call_table)(,%eax,4)
    movl %eax,EAX(%esp)             # save the return value
ENTRY(ret_from_sys_call)
    RTLINE_CLI                     # need_resched and signals atomic test
    cmpl $0,need_resched(%ebx)
    jne reschedule
    cmpl $0,sigpending(%ebx)
    jne signal_return
restore_all:
    RTLINE_IRET
    RESTORE_ALL

```

코드 1278. System Call의 Entry point에서의 RTLinux의 macro 사용

먼저, CONFIG_RTILINE가 정의되어 있다면, 모든 register값을 저장한 후에(SAVE_ALL), rtl_syscall_intercept의 여부에 따라서, 다음번의 제어 위치가 달라지게 된다. 실제로 rtl_syscall_intercept는 default값으로 아무런 값을 가지고 있지 않으며, 나중에 RTLinux의 core module이 loading될 때 그 주소 값이 결정된다. 따라서, RTLinux의 loading여부에 따라서, 원래의 Linux 커널을 수행할 것인지 아니면, RTLinux 커널을 수행할 것인가를 결정해주는 비교 부분이 들어가 있는 것이다. 만약 정의된 것이 있다면, 그 함수를 곧바로 호출할 것이다. rtl_unix.c를 참고로 보면 아래와 같은 정의가 있음을 확인할 수 있을 것이다.⁴⁹³

```

void *rtl_emulate_iret = 0;
void *rtl_exception_intercept = 0;
void *rtl_syscall_intercept = 0;

```

코드 1279. RTLinux의 함수 포인터의 정의

또한 ret_from_sys_call()이라는 함수 부분을 보면, 이곳에 앞에서 정의한 RTLINE_CLI와 RTLINE_IRET를 사용하고 있음을 볼 수 있다. 먼저 RTLINE_CLI는 cli(Clear Interrupt)를 emulation하는 것으로 rtl_unix.c에 아래와 같은 정의를 가진다.

```

struct irq_control_s irq_control = {
    rtl_hard_save_flags_f,
}

```

⁴⁹³ 현재 보고 있는 커널은 Linux 2.4.2 커널 버전이며, RTLinux patch를 한 상태이다. 따라서, 일반적인 커널 코드상에서는 찾을 수 없다.

```

rtl_hard_restore_flags_f,
rtl_hard_cli_f,
rtl_hard_sti_f,
rtl_hard_local_irq_save_f,
rtl_hard_local_irq_restore_f
};

```

코드 1280. irq_control의 자료구조

따라서, RTLINUX_CLI 매크로는 단지 이곳에 존재하는 함수(rtl_hard_cli_f())를 수행하는 시켜주는 역할을 해준다. 이것은 cli를 일으켜서 실시간으로 발생하는 interrupt를 완전히 disable시키기 보다는 일종의 flag에 현재 disable시켰다는 것을 나타내 주기만하고, interrupt의 발생을 용인한 후, 발생한 interrupt가 어떤 것들인가만을 기억하기 위한 것이다. 나중에 이를 다시 flag이 enable되어을때 하나씩 처리해 줄 것이기 때문이다. 이렇게 함으로써 실시간 interrupt를 처리하는 것을 가능하게 만들어 준다. RTLINUX_IRET 역시 마찬가지의 iret(Interrupt Return)를 emulation하는 일을 처리하게 된다.

결론적으로 말하면, RTLinux patch를 하면, Linux 커널을 일종의 우선 순위가 가장 낮은 real-time task로 만들어 준다는 이야기다. 다만, 이러한 것이 가능하기 위해서는 RTLinux의 module들이 커널과 link되어서 동작해야만 가능하다.

Linux 커널을 patch하는 것은 간단히 다음과 같은 명령을 사용해서 처리할 수 있다. 먼저 해당 Linux 커널이 있는 곳으로 이동해서 patch 명령을 수행하는 것이다.

```
% cd /usr/src/linux
% patch -p1 < rtlinux_kernel_patch-2.4.2
```

여기서, rtlinux_kernel_patch_2.4.2는 patch하려는 버전이나 rtlinux professional을 사용할 경우에는 다른 파일 이름으로 바뀔 수 있다. 이 같은 과정을 수행하고 나서 Linux 커널을 컴파일하기 위해, ‘make xconfig’과 같은 명령을 사용하면, RTLinux에 관련된 compile 옵션을 볼 수 있을 것이다. 단순히 RTLinux 모듈을 사용할 것인가만을 결정해 주면 된다. 여기서는 RTLinux를 위한 커널의 설정을 다룬 것이며, 다음은 RTLinux를 위한 자체 설정 및 compile을 보도록 하자.

RTLinux의 document를 설치하는 것은 다음과 같은 과정을 따른다. 먼저 설치하려고 하는 RTLinux의 document 파일을 가지고 왔고 현재의 directory에 그 파일이 있다고 생각한다.

```
% tar xvfz rtldoc-3.0.tar.gz
% vi /etc/man.config
...
MANPATH      /usr/rtlinux/man
...
```

즉, 압축을 해제한 directory가 /usr/rtlinux라고 한다면, 이 directory의 이름을 /etc/man.config파일에 입력해서 manual을 찾는 경로(path)에 덧붙여둔다. 이것으로 RTLinux의 document에 대한 설치가 끝나며, 나중에 RTLinux의 API를 찾고 싶다면, 간단히 man XXX라는 형식으로 찾으면 될 것이다.

RTLinux를 compile하는 것도 커널을 compile하는 것과 동일한 과정을 따른다. 먼저 RTLinux의 압축을 해제한 후, 그 압축을 해제한 directory가 /usr/src/rtlinux-3.0이라고 할 경우 아래와 같이 하면 될 것이다.

```
% cd /usr/src/rtlinux-3.0
% make mrproper
% make xconfig
% make; make devices
% make install
```

즉, make xconfig을 하면, X window상에 Linux 커널을 설정하는 것과 같은 window가 하나 보일 것이며, 이 window상에서 필요한 module이나 기능을 설정한 후, make와 make devices를 통해서 RTLinux 커널을 compile하고, RTLinux가 필요로 하는 디바이스 파일들을 /dev 디렉토리 이하에 만들어 주게되며, make

install이라는 명령으로 최종적인 설치를 마무리 하게된다. 여기서 주의할 것은 RTLinux에서는 APM(Advanced Power Management)를 사용해서는 안된다는 것이다. 따라서, Linux 커널을 설정할 때도 이 부분을 사용한다고 해서는 안될 것이다.⁴⁹⁴

이것을 다 마쳤다면, 다음과 같은 파일이 있는지를 확인하도록 하자. 즉, /usr/rtlinux/bin/rtlinux와 /usr/rtlinux/bin/rtl-config 두 파일이 있는지를 보면 될 것이다. 설치가 끝나면 정확히 설치가 되었는지를 확인하기 위해서 다음을 수행한다.

```
% cd /usr/rtlinux
% ./regression.sh
```

위와 같은 명령어를 수행하면, RTLinux가 적절하게 설치가 되었는지를 자동으로 테스트해서 결과를 알려줄 것이다. 만약 설치가 잘 못되었다면, 처음부터 다시 설치해야 할 것이다. 이것으로 RTLinux에 대한 설치는 마무리 되었다. 이젠 Linux를 rebooting 시키는 일이 남았다. 이를 위해서는 먼저 /etc/lilo.conf의 설정을 바꿔서 lilo를 이용해서 booting할 때 loading되어야 하는 커널이 어떤 것인가를 알려주어야 할 것이다. 다음과 같이 하도록 한다.

```
% cd /etc
% vi lilo.conf
...
default = RTLinux
...
image=/boot/linux-2.4.2
    label=RTLinux
    read-only
    root=/dev/hda1
...
...
```

여기서는 RTLinux를 기본 boot 커널 image로 설정하고, 우리가 새로 RTLinux patch를 해서 compile한 커널 image를 linux-2.4.2라고 했을 때의 이야기를 하는 것이다. root 파일 시스템으로 사용할 장치는 /dev/hda1이다. 이렇게 하고난 후에, 다시 lilo의 image를 update하기 위해서 lilo를 입력하도록 한다.

% lilo

자, 이것으로 RTLinux를 사용하기 위한 준비는 다 끝났다. 이젠 computer를 새로 booting하기만 하면 될 것이다. 본격적인 RTLinux에 대한 이야기를 하기 전에 먼저 간단히 RTLinux의 sample program을 하나 보기로 하자. 이것을 통해서 우리는 RTLinux의 application의 기본 구조를 파악할 수 있을 것이다.

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>
RTLINUX_MODULE(hello);

pthread_t thread;

void * start_routine(void *arg)
{
    struct sched_param p;
    hrtime_t next = clock_gettime(CLOCK_REALTIME) + 1000000000;
    p . sched_priority = 1;
    pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
```

⁴⁹⁴ RTLinux에서 말하길, APM은 clock frequency와 memory timing, bus frequency등의 비용을 지불해야 하기에 예측 가능한 시스템을 목표로 하는 RTOS(Real-Time OS)에는 적합하지 않다는 것이다. 따라서, BIOS 및 Linux 커널 설정(configuration)에서도 disable(불가) 시키도록 권장하고 있다.

```

while (1) {
    clock_nanosleep (CLOCK_REALTIME, TIMER_ABSTIME,
                    hrt2ts(next), NULL);
    next += 500000000;
    rtl_printf("I'm here; my arg is %x\n", (unsigned) arg);
}
return 0;
}

int init_module(void) {
    return pthread_create (&thread, NULL, start_routine, 0);
}

void cleanup_module(void) {
    pthread_cancel (thread);
    pthread_join (thread, NULL);
/*    pthread_delete_np (thread); */
}

```

코드 1281. RTLinux Sample 프로그램 Hello.c

먼저, RTLinux용의 application을 만들기 위해서는 path를 잘 설정해서 사용해야 할 것이다. 즉, RTlinux application에서 사용하게 되는 header 파일이나 기타 환경을 잘 이해해야 한다. 기본적으로 RTLinux에서는 application을 compile하기 위해서 Makefile을 제공하고 있으므로 이를 이용하면 될 것이다. 간단히 해당 directory로 가서 make를 치면, application의 build가 끝날 것이다.

RTLINUX_MODULE()이라는 macro는 실제로는 아무런 역할을 하지 않지만, 일반적으로 Linux 커널 module을 정의하는데 사용한다고 생각하도록 하자. init_module()함수와 cleanup_module() 함수는 각각 module의 초기화와 끝내기를 담당하며, 여기서는 pthread_XXX()와 같은 함수를 사용하고 있다. pthread_XXX() 함수는 RTLinux가 마치 POSIX standard의 pthread API와 유사한 형태로 응용 프로그램 프로그래머에게 API interface를 제공하기 위한 것이다. 완전히 pthread API를 전부 구현한 것이 아니라, 일부만 유사하도록 구현한 것으로 혼동하지 말도록 하자. 일단 pthread_create()는 새로운 thread를 생성하는 것이며, pthread_cancel()은 생성된 thread를 취소(cancel)하며, pthread_join()은 thread가 실행을 마치도록 기다리는 것이며, pthread_delete_np()는 non-posix 확장으로 pthread를 지워준다. 따라서, pthread_create()를 module의 초기화 시에 사용해서 새로운 thread를 생성한다. 생성된 thread는 start_routine() 함수를 수행하는 thread가 될 것이다. 하지만, 역시 여기서 말하는 thread는 일반적인 Linux의 thread와는 다르므로 주의하기 바란다. 즉, Real-time task를 수행하는 thread가 되기 때문이다.

start_routine() 함수는 먼저 현재의 시간을 구한다음(clock_gettime()), 여기에 1초(=1000000000nsec)를 더해서 다음 번에 수행될 시간을 선택한다(next). Thread의 우선 순위는 1로 두고(p.sched_priority), 이 우선 순위를 반영하기 위해서 pthread_setschedparam() 함수를 사용한다. 우선 순위에 따른 scheduling 방식은 FIFO(SCHED_FIFO)를 사용하도록 한다. 이젠 마지막으로 무한 loop를 돌면서 일정한 간격으로 rtl_printf()를 호출해서 console에 어떤 값을 계속적으로 표시한다. rtl_printf() 함수는 Linux 커널의 printk() 함수와 동일하다고 보면 된다. 다음번 수행시간은 여기서 5초(=5000000000nsec)후가 될 것이다. 수행 간격을 조정하기 위해서 clock_nanosleep()라는 함수를 사용해서 이를 수행한다.

이제 이 프로그램을 수행하기 위해서는 다음과 같은 명령을 사용하도록 한다. 여기서의 주안점은 위와 같은 프로그램이 수행되기 위해서 기본적으로 RTLinux 커널 모듈이 Linux 커널과 link가 되어야 하기 때문에 먼저 loading 시켜 준다는 것이다.

% rtlinux start hello.o

rtlinux는 앞에서 우리가 RTLinux를 설치하는 동안에 생긴 실행할 수 있는 script 파일이며, start는 RTLinux의 application을 수행하기 위한 module들을 loading하라는 것이다. 마지막으로 우리가 실행시켜줄 hello.o 모듈을 loading하라는 뜻이다. 따라서, Linux를 이용해서 module로서 구현된 RTLinux의 application은 전부 Linux의 커널 영역에서 수행된다는 점이다. 따라서, kernel thread와 유사하다고 생각하면 된다. 한가지 유념할 것은 Linux의 커널 thread는 Linux 커널에서 제공해 주는 다양한 함수들을

사용할 수 있지만, RTLinux에서 생성하는 thread는 이러한 function들을 전부다 사용할 수는 없다. 즉, Linux 커널 내에서 sleep하게 되는 경우가 발생하면, Linux 커널을 깨워줄 방법이 없기에 이럴 경우에는 시스템에 치명적인 오류를 발생시킬 수 있다.

자, 이제 우리는 대략적으로 나마 RTLinux를 설치하고, 간단한 RTLinux application을 수행할 수 있는 능력을 가지게 되었다. 이하에서는 RTLinux에 대한 구현 원리에 대해서 보게 될 것이기에 더 이상의 설치와 수행에 대한 이야기는 그만하도록 하겠다.

20.4. RTLinux의 Thread 구현

RTLinux에서는 최소 스케줄링의 단위(UNIT)⁴⁹⁵가 되는 것이 바로 pthread⁴⁹⁶라는 것이다. 자료구조는 rtl_thread_struct 구조체로 정의되며 코드는 RTLinux/schedulers/rtl_sched.h에 있다. Thread와 관련된 부분으로 RTLinux의 스케줄링을 담당하는 모듈은 RTLinux/schedulers/rtl_sched.c⁴⁹⁷이다. 스케줄링에 대해서는 나중에 다시 볼 것이며, 먼저 thread의 자료구조부터 보면 RTLinux가 생각하는 thread를 구성하는 것은 다음과 같다.

```
struct rtl_thread_struct {
    int *stack;           /* hardcoded */
    int fpu_initialized;
    RTL_FPU_CONTEXT fpu_regs;
#ifndef CONFIG_RTL_MMU_SUPPORT
    rtl_gpos_task_t mmu_gpos_task;
*/
    rtl_mmu_state_t mmu_state;
    rtl_pthread_t mmu_next;
*/
    int uses_mmu;
    int userid;
#endif
    int uses_fp;
    int *kmalloc_stack_bottom;
    struct rtl_sched_param schedpar;
    struct rtl_thread_struct *next;
    int cpu;
    hrtimer_t resume_time;
    hrtimer_t period;
    hrtimer_t timeval;
    struct module *creator;
    void (*abort)(void *);
    void *abortdata;
    int threadflags;
    rtl_sigset_t pending;
    rtl_sigset_t blocked;
    int errno_val;
    struct rtl_cleanup_struct *cleanup;
    int magic;
    struct rtl_posix_thread_struct posix_data;
}
/* Thread가 사용할 stack에 대한 포인터 */
/* Floating Pointer Unit의 초기화 여부 */
/* FPU의 context 저장 */
/* MMU를 사용한 연산을 하려고 할 경우 */
/* MMU를 사용하는 Linux 프로세스에대한 정보 */
/* MMU의 상태 */
/* MMU를 사용하는 다음 thread에 대한 포인터 */
/* 현재 MMU를 사용한다고 표시 */
/* 사용자의 ID */
/* Floating Point를 사용함 */
/* Kernel malloc 함수의 stack bottom */
/* Thread의 scheduling parameter */
/* 다음 thread에대한 포인터 */
/* 사용하는 CPU의 번호 */
/* 다시 수행될 시간 */
/* 실행되는 주기 */
/* 시간 값을 저장 */
/* 모듈의 생성자에 대한 포인터 : parent */
/* Thread의 실행 중지 함수에 대한 포인터 */
/* Abort 함수에 넘겨질 argument 값의 포인터 */
/* Thread의 상태를 표시하는 flag */
/* 처리되지 않은 signal의 집합 */
/* Blocking된 signal들의 집합 */
/* Thread의 error 값 : return시에 설정되는 errno */
/* Thread module의 cleanup을 위한 자료 구조 */
/* Magic Number */
/* POSIX thread 호환 자료 구조 */
```

⁴⁹⁵ 이것은 일반적인 운영체제에서 주로 Thread를 정의하는 용어로서 사용하는 경우가 많다.

⁴⁹⁶ 이것은 명확히 POSIX standard thread와는 다르다. 호환이 되도록 만들어졌지만, POSIX standard의 모든 API를 지원하는 것은 아니다. 이것은 프로그래머에게 익숙한 환경을 제공하려는 의도라고 생각된다.

⁴⁹⁷ 이하에서 하는 모든 설명은 RTLinux를 RTLinux가 설치된 directory라고 생각하고 이야기 하도록 하겠다.

```
void *tsd [RTL_PTHREAD_KEYS_MAX];           /* POSIX thread의 key값을 저장하는 자료구조 */
};
```

코드 1282. rtl_thread_struct 구조체의 정의

rtl_thread_struct 구조체는 Linux의 task_struct 구조체와 같은 역할을 한다고 생각하면 될 것이다. 즉, RTLinux가 thread를 생성할 때, 하나의 thread당 생성해 주는 자료 구조체로서 이것을 기준으로 모든 thread의 정보를 기술해 줄 수 있다. 앞에서 CONFIG_RTL_MMU_SUPPORT는 Linux 커널내에서는 floating point unit연산을 수행하지 않지만, RTLinux에서 application thread가 일반 Linux 프로세스와 같이 floating point 연산을 하기 위한 context를 유지하기 위해서 필요하다. 만약 현재 진행중인 thread가 context switching을 해야 한다면, 이 thread가 floating point연산을 수행할 경우에 한해서 floating point context도 같이 저장해 줄 수 있어야 한다. 그래야만, 다음번에 thread가 실행 기회를 얻게 되었을 때, 여기서 floating point연산의 context를 복구 할 수 있다.

20.4.1. PThread의 생성

RTLinux에서 사용하는 pthread의 생성은 __pthread_create() 함수가 맡고 있다. 정의는 RTLinux/schedulers/rtl_sched.c에 있다.

```
int __pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg, struct module *creator)
{
    int *stack_addr;
    long interrupt_state;
    struct rtl_thread_struct *task;
    pthread_attr_t default_attr;
    int stack_size;
    if (!attr) {
        pthread_attr_init(&default_attr);
        attr = &default_attr;
    }

    stack_size = attr->stack_size;
    stack_addr = (int *) attr->stack_addr;

    if (!stack_addr) {
        if (pthread_self() != pthread_linux())
            return EAGAIN;
    }
    stack_addr = (int *) rtl_gpos_malloc(stack_size);
    task = (struct rtl_thread_struct *) stack_addr;
    if (!task)
        return EAGAIN;
    }
    task->kmalloc_stack_bottom = stack_addr;
} else {
    task = (struct rtl_thread_struct *) stack_addr;
    task->kmalloc_stack_bottom = 0;
}

*thread = task;
```

코드 1283. __pthread_create() 함수의 정의

__pthread_create() 함수는 먼저 pthread_t 구조체와 pthread_attr_t이라는 pthread의 attribute를 나타내는 구조체 및 thread로 실행할 함수에 대한 포인터와 그 함수에 대한 argument, 그리고 마지막으로 생성한 모듈을 나타내는 포인터를 넘겨받는다. 만약 attribute가 NULL이라면, 기본적으로 가져야 하는

attribute(default_attr)로 초기화 시킨다. Pthread의 stack 크기 및 stack의 주소는 여기서 결정된다. 만약 stack의 주소가 NULL 값을 가진다면, 생성하려는 thread가 Linux thread가 아닐 경우에는 EAGAIN을 돌려준다. 그렇지 않다면, rtl_gpos_malloc()을 호출해서 Linux 커널로부터 thread의 stack을 위한 공간을 할당 받는다. 이곳에 rtl_thread_struct 구조체를 앞 부분에 생성하도록 한다(task). 만약 생성할 수 없다면 EAGAIN을 돌려준다. 이전 stack의 bottom을 나타내는 필드를 stack이 할당된 주소로 저장한다(stack_addr). 만약 앞에서 stack의 주소를 지정한 경우에는 그 주소를 사용하도록 한다. 이때는 stack의 bottom 주소를 0으로 둔다. Thread는 rtl_thread_struct를 가지게 될 것이다.

```

task -> threadflags = 0;
if (attr->detachstate == PTHREAD_CREATE_JOINABLE) {
    set_bit(RTL_THREAD_JOINABLE, &task->threadflags);
}

task->magic = RTL_THREAD_MAGIC;
task->creator = creator;
task->pending = attr->initial_state;
task->blocked = 0;
task->abort = 0;
task->cpu = attr->cpu;
task->cleanup = 0;
task->resume_time = HRTIME_INFINITY;
task->period = 0;
task->schedpar = attr->schedpar;
task->stack = stack_addr + stack_size / sizeof(int);
rtl_init_stack(task,start_routine,arg,rtl_startup);
rtl_no_interrupts(interrupt_state);

task->fpu_initialized = 0;
task->uses_fp = attr->use_fp;

```

코드 1284. __pthread_create() 함수의 정의(계속)

이전 threadflag를 0으로 초기화 시킨다. 만약 thread의 attribute가 PTHREAD_CREATE_JOINABLE이라면, threadflag 필드에 RTL_THREAD_JOINABLE이라고 표시한다.⁴⁹⁸ magic과 create, pending 필드는 각각 RTL_THREAD_MAGIC과 creator(생성 모듈), attribute의 initial_state 값으로 두고, blocked 필드는 초기에 생성 중이므로 0으로 초기화 한다. abort 필드도 역시 0으로 두고, cpu 값은 attribute의 cpu 값으로 cleanup은 0으로 둔다. resume_time은 아직 변경이 되지 않았으므로, HRTIME_INFINITY로 두어서 무한대 시간에 다시 시작하도록 둔다. 즉, 나중에 resume_time이 변경되면 적절한 값으로 설정될 것이다. period도 0으로 두어서 실행간격을 초기화한다. 스케줄링 파라미터는 attribute의 schedpar 값으로 설정하고, stack 필드는 스택으로 할당 받은 영역의 주소에 스택의 크기를 integer(정수) 크기 값으로 나눈 값을 더한 것으로 설정한다. 즉, 대략 2로 나눈 크기를 thread의 stack 영역으로 사용한다고 보면 될 것이다. 이전 앞에서 설정한 stack 영역에 대한 초기화를 수행하고(rtl_init_stack()), 인터럽트의 발생을 막도록 한다. FPU(Floating Point Unit)은 초기화를 아직 않았기에 0으로 두게 되며, 나머지 FPU에 대한 사용여부는 attribute의 use_fp 값으로 초기화 시킨다.

```

#endif CONFIG_RTL_MMU_SUPPORT
task->userid = 0;
task->uses_mmu = 0;
if (attr->use_mmu) {
    if (pthread_self() == pthread_linux()) {
        task->mmu_gpos_task = current;
    } else if (pthread_self()->uses_mmu) {
        task->mmu_gpos_task = pthread_self()->mmu_gpos_task;
    }
}

```

⁴⁹⁸ 이것은 POSIX thread library의 join과 유사하다. 참고해서 보도록 하자.

```

    } else {
        rtl_printf("RTL: an mmu-less task trying to create an mmu task\n");
        rtl_restore_interrupts(interrupt_state);
        return EINVAL;
    }
    rtl_mmu_init(task); /* clone Linux's current mmu state */
    task->uses_mmu = 1;
/*     rtl_printf("task size = %#x\n", sizeof(*task)); */
}
#endif
{
    schedule_t *s = sched_data(task->cpu);
#endif CONFIG_SMP
if (task->cpu != rtl_getcpuid()) {
    rtl_rawlock (&s->rtl_tasks_lock);
    task->next = s -> rtl_new_tasks;
    s->rtl_new_tasks = task;
    rtl_rawunlock (&s->rtl_tasks_lock);
    rtl_reschedule (task->cpu);
} else
#endif
{
    rtl_rawlock (&s->rtl_tasks_lock);
    add_to_task_list(task);
    rtl_rawunlock (&s->rtl_tasks_lock);
    rtl_schedule();
}
}
rtl_restore_interrupts(interrupt_state);
return 0;
}

```

코드 1285. __pthread_create() 함수의 정의(계속)

만약 RTLinux가 MMU를 지원한다고 하면(CONFIG_RTL_MMU_SUPPORT), #ifdef이하를 실행한다. userid 필드는 0으로 두고, uses_mmu는 0으로 초기화 한다. 만약 attribute의 use_mmu값이 있다면, 현재 thread가 Linux 인지를 확인하고, 같다면 mmu_gpos_task에 current값을 넣어서 현재 진행중인 Linux task(혹은 process)를 둔다. 그렇지 않다면, pthread의 uses_mmu값을 보고 어떤 값이 있다면, mmu_gpos_task에는 현재 실행중인 pthread의 mmu_gpos_task으로 초기화 한다. 앞에서와 같은 것에 해당 사항이 없다면, 현재 pthread task가 MMU를 사용하지 않는데, MMU를 사용하는 task를 생성하려고 하기 때문에, EINVAL를 돌려주게된다. 복귀전에 앞에서 저장한 interrupt의 상태를 복구해야한다(rtl_restore_interrupts()). 이젠 MMU를 사용하도록 task를 초기화 시키기 위해서 rtl_mmu_init() 함수를 호출하고, uses_mmu필드에는 MMU를 사용한다는 것을 알려주기 위해서 1로 설정한다.

Task가 스케줄링 될 CPU의 스케줄링 데이터를 s가 가르키도록 한 후, SMP(Symmetric Multi-Processor)을 지원한다면, if() ~ else까지를 실행한다. 먼저 CPU의 ID를 가져와서 task의 CPU ID와 비교한 후 같다면, lock을 설정해서 스케줄링이 일어나는 queue에 새롭게 생성한 task를 삽입하고, rtl_reschedule() 함수를 호출해서 새로운 task를 스케줄링 하도록 요구한다. add_to_task_list() 함수는 task를 스케줄링 list에 삽입하는 역할을 한다. 복귀하기 전에 앞에서 설정한 interrupt에 대한 lock을 해제하기 위해서 rtl_restore_interrupts()를 호출하게 되며, 복귀 값은 0이다.

20.4.2. PThread의 종료

Pthread를 종료하기 위해서 사용하는 API는 pthread_exit()이다. 이 함수를 호출한 후 thread의 종료가 값을 얻기 위해서는 retval에 전역적으로 사용할 수 있는 데이터에 대한 포인터를 넘겨 주어야 할 것이다.

void pthread_exit(void *retval)

```
{
    rtl_irqstate_t flags;
    int found = 0;
    pthread_t self = pthread_self();
    schedule_t *sched = sched_data(self->cpu);
    int error = 0;
    rtl_thread_t * tmp;

    error = ((self ->cpu != rtl_getcpuid())? NOT_ISONRIGHTCPU : 0);
    if(!error) {
        is_task_on_taskq(sched->rtl_tasks, self, tmp, found);
        if(!found) error = NOT_ISTASK;
    }
    if(error){
        rtl_printf("RTL SICK: pthread exit task is not on its sched q %d\n", error);
        return;
    }

    rtl_no_interrupts(flags); /* TODO this is way too long with interrupts disabled */

    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    do_cleanups(self);
    rtl_posix_cleanup(retval);
    set_bit(RTL_THREAD_FINISHED, &self->threadflags);
    rtl_posix_on_delete(self);
}
```

코드 1286. `pthread_exit()` 함수의 정의

현재 진행중인 thread를 self로 가르키게 한다(`pthread_self()`). 이 thread가 사용하는 스케줄링 데이터는 sched로 두도록 한다. 현재 task가 scheduling되고 있는 CPU가 정확한지를 확인하고, 만약 정확하지 않다면(!error), 그 task가 스케줄링 queue에 들어가 있는지를 다시 확인하다. 없다면, error값에는 NOT_ISTASK가 들어가게 될 것이다. 어쨌든, 앞에서 어떤 error가 발생했다면 그냥 복귀하게 된다. 즉, 제대로 task가 알맞은 CPU상에서 제대로 스케줄링 되고 있지 않다는 것이다. 이전 task를 스케줄링 queue에서 제거하는 일이다. 먼저 interrupt를 막도록 한다(`rtl_no_interrupts()`). Thread의 상태를 PTHREAD_CANCELE_DISABLE로 두고, `do_cleanups()` 함수를 호출해서 cleanup으로 지정된 일이 있을 경우에 이를 처리하도록 한다. `rtl_posix_cleanup()`은 POSIX style의 cleanup(지우기)위한 것으로 이것도 역시 있다면, 처리하도록 해야 할 것이다. `threadflags` 필드에 RTL_THREAD_FINISHED로 설정해서 실행이 끝났음을 표시하고(`set_bit()`). `rtl_posix_on_delete()`를 호출해서 delete시에 실행할 일들을 처리해 주도록 한다.

```

rtl_rawlock(&sched->rtl_tasks_lock);
if_fpu_owner_giveup(sched, self);
if_mmu_owner_giveup(sched, self);
deq_taskq(sched->rtl_tasks, self, tmp, found);
if(!found){
    rtl_printf("RTL: pthread_exit on thread=%x (GPOS= %x) can't remove\n", self, sched->rtl_linux_task);
}
/* if found, then we are not on the runq anymore! */
rtl_rawunlock(&sched->rtl_tasks_lock);
if(self ->kmalloc_stack_bottom){
    struct timespec tm;
    while(zombieq_enq(&sched->zq, self->kmalloc_stack_bottom)){
        clock_gettime(CLOCK_REALTIME, &tm);
        timespec_add_ns(&tm, 10000000); /* wait 10 milliseconds */
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &tm, NULL);
    }
}
```

```

    }
    /* now we are absolutely committed. Our memory may be on the
     * way to the trash head. Gotta go
     */
    self->magic = 0;
    rtl_schedule();
    /* will never reach this line */
    rtl_printf("RTL: Monster error: pthread_exit() returned!\n");
    rtl_schedule(); /* this would be very bad */
}

```

코드 1287. pthread_exit() 함수의 정의(계속)

이전 FPU의 사용에 대한 처리를 하도록 한다. 먼저 lock을 걸어서 현재 CPU의 스케줄링 데이터가 깨지지 않도록 보호하고(rtl_rawlock()), FPU와 MMU에 대한 처리를 해준다(if_fpu_owner_giveup(), if_mmu_owner_giveup()). 이전 스케줄링 queue에서 제거하기 위해서 deque_taskq() 함수를 호출하고, 만약 제거하려는 task를 찾지 못한다면, rtl_printf()를 호출해서 화면에 원가를 쓸 것이다. 앞에서 설정한 lock을 해제하고 한다(rtl_rawunlock()).

만약 stack 영역을 따로 할당받아서 사용하고 있었다면, 잠시 task의 실행 정보에 대한 것을 상위 생성 task에 알려주기 위해서 zombie queue에 넣어둔다(zombieq_enq()).⁴⁹⁹ 잠시 실행을 대기하기 위해서 clock_nanosleep() 함수를 호출해서 10 milliseconds를 쉰다. 제대로 처리가 되었다면, magic에는 0을 두고, 새로운 task를 찾아서 실행하기 위해서 rtl_schedule()을 호출한다. 이하의 부분은 실행되지는 않을 것이다.

20.4.3. PThread의 취소

RTLinux에서 pthread를 cancel하는 함수는 pthread_cancel()로 정의되며, 실제로는 rtl(pthread_cancel()) 함수가 이를 대신한다. 아래와 같이 정의된다.

```

/* rtl_sched.h에서 */
#define RTL_PRIO(th) ((th)->schedpar.sched_priority)
...
/* rtl_sched.c에서 */
int rtl(pthread_t thread)
{
    if (RTL_PRIO(thread) < RTL_PRIO(pthread_linux())) {
        RTL_PRIO(thread) = RTL_PRIO(pthread_linux()) + 3;
    }
    return pthread_kill(thread, RTL_SIGNAL_CANCEL);
}

```

코드 1288. rtl(pthread_cancel()) 함수의 정의

rtl(pthread_cancel()) 함수는 단순히 특정 thread에 RTL_SIGNAL_CANCEL이라는 signal을 전달하는 일만 한다. 이때 만약 target이 되는 thread가 Linux보다 낮은 우선 순위를 가지게 될 경우에는 Linux thread보다 우선 순위를 높이기 위해서 3을 우선 순위에 더하게 된다. Signal을 보내는 것은 pthread_kill()이 하게 된다.

20.4.4. PThread의 Signal

```

#define CHECK_VALID(thread) do { if (((thread)->magic != RTL_THREAD_MAGIC) return ESRCH; } while (0)

int pthread_kill(pthread_t thread, int signal)
{

```

⁴⁹⁹ 일반 Linux에서도 task의 상태가 완전히 종료되어 제거되기 전에 ZOMBIE상태로 바뀌는 것과 동일하다고 보면 될 것이다.

```

if ((unsigned) signal <= RTL_MAX_SIGNAL) {
    CHECK_VALID(thread);
    if (signal == 0) {
        return 0;
    }
    set_bit(signal, &thread->pending);
    return 0;
} else if(thread != pthread_linux()) {
    return EINVAL;
} else { /* the signal number means something else */
    /* TODO one range for local one for global */
    if((signal < RTL_LINUX_MIN_SIGNAL)
        || (signal > RTL_LINUX_MAX_SIGNAL))
        return EINVAL;
    else if(signal < RTL_LINUX_MIN_LOCAL_SIGNAL){ /* it's global */
        rtl_global_pend_irq (signal - RTL_LINUX_MIN_SIGNAL);
    }
#endif
#endif
#endif
}

```

코드 1289. pthread_kill() 함수의 정의

RTLinux에서는 thread에 시그널(signal)을 전달하기 위해 pthread_kill() 함수를 사용한다. 이미 앞에서 pthread_cancel()에서 보듯이 특정 thread를 대상으로 시그널을 전달한다. CHECK_VALID() 매크로는 target이 되는 thread가 RTLinux의 thread인지를 확인하기 위해서 사용한다. 넘겨받는 정보는 target이 되는 thread와 시그널이다.

먼저 보내고자 하는 시그널이 RTL_MAX_SIGNAL(=31)이하 인지를 확인한다. 만약 그렇지 않다면, 받는 thread가 Linux thread인지를 확인하고 그렇지 않다면, EINVAL을 넘겨준다. RTL_MAX_SIGNAL이하의 값을 가진다면, 유효한 thread인지를 확인하고(CHECK_VALID()), 시그널 값이 0이라면 바로 0을 return한다. 그렇지 않다면, 해당 thread의 pending된 시그널에 이 시그널을 더한다(set_bit()). 복귀 값은 0이다.

만약 RTL_MAX_SIGNAL보다 큰 값을 가지면서 target이 되는 thread가 Linux threed가 아닌 경우에는 시그널 값이 다른 원가를 의미하는지를 확인한다. 먼저 signal이 RTL_LINUX_MIN_SIGNAL(= 256)보다 작거나, RTL_LINUX_MAX_SIGNAL(= 1024) 보다 큰 값을 가진다면 EINVAL을 돌려주고, 256보다는 크기만 1024보다는 작고, 또한 RTL_LINUX_MIN_SIGNAL(= 512) 보다도 작다면 rtl_global_pend_irq()를 호출해서 현재 pending된 global interrupt값으로 처리한다. 이때 처리되는 시그널은 원래의 시그널에서 RTL_LINUX_MIN_SIGNAL값을 뺀 값이다. 만약 SMP를 지원하도록 RTLinux가 설정되었다면, rtl_local_pend_vec()를 호출해서 local interrupt값으로 사용하도록 한다. 이때는 특정 CPU상 국한된 것으로서 rtl_getcpuid()를 호출해서 어떤 CPU에 pending된 것인가를 나타낸다. 복귀 값은 0이 될 것이다. 여기서 특이한 점은 특정 interrupt와 같은 것을 signal을 전달하는 것과 흡사한 방법으로 보낸다는 것이다. 나중에 이렇게 pending된 인터럽트나 signal이 있다면, 해당 thread나 해당 CPU의 interrupt handler와 같은 곳에서 처리가 이루어 질 것이다.

20.5. RTLinux의 Timer 구현

X86계열에서 timer를 구현하는 방식은 크게 두 가지로 나뉘어진다. 즉, TSC(Time Stamp Counter)가 있는 경우의 CPU와 그렇지 않은 경우의 CPU로 분류된다. TSC는 일반적으로 Pentium과 P6계열의 CPU에는 다 있으며, 64 bit 크기의 counter로 processor의 hardware적인 reset에 의해서 0으로 설정된다. 카운터는 모든

pcoesssor의 clock cycle에 대해서 하나씩 증가되기에 대략 적으로 10년 정도 버틸 수 있는 bit 개수(64)를 가지고 있다. 만약 이러한 것이 없다면 8254 chip을 사용해서 timer를 구현한다. 아래와 같다.

```
hrtimer_t global_8254_gettime (void)
{
    register unsigned int c2;
    int flags;
    long t;

    rtl_spin_lock_irqsave (&lock8254, flags);
    LATCH_CNT2();
    READ_CNT2(c2);
    offset_time += ((c2 < last_c2) ? (last_c2 - c2) / 2 : (last_c2 - c2 + LATC2) / 2);
    last_c2 = c2;
    if (offset_time >= CLOCK_TICK_RATE) {
        offset_time -= CLOCK_TICK_RATE;
        base_time += HRTICKS_PER_SEC;
    }

#ifndef HRTICKS_PER_SEC != CLOCK_TICK_RATE
    __asm__ ("shl $10, %%eax\n\t"
            "mul %%ebx\n\t"
            :"=d" (t) : "b" (scaler_8254_to_hrtimer), "a" (offset_time));
#else
    t = offset_time;
#endif
    last_8254_time = base_time + t;
    rtl_spin_unlock_irqrestore (&lock8254, flags);
    return last_8254_time;
}
```

코드 1290. 8254 Chip을 이용한 time구하기

먼저 interrupt를 발생시키지 않도록 rtl_spin_lock_irqsave()를 호출해서 lock을 설정하고, LATCH_CNT2()와 READ_CNT2() 매크로를 호출해서 현재의 chip에 있는 time tick값을 읽는다. 이것을 구해서 offset_time의 값을 구한다. 만약 offset_time이 CLOCK_TICK_RATE보다 크다면, CLOCK_TICK_RATE값을 offset_time에서 빼주고, 다시 base_time값에 HRTICKS_PER_SEC를 더해 초당 발생한 hard real-time tick을 증가 시켜준다. 만약 HRTICKS_PER_SEC와 CLOCK_TICK_RATE값이 일치하지 않는다면, 8254 chip에서 읽은 값을 HRT(Hard Real-Time)로 바꾸기 위한 일을 하게 되며, 그렇지 않다면 t에 offset_time을 넣어준다. 마지막으로 읽은 8254의 time을 갱신(last_8254_time)하고, 앞에서 설정한 lock을 풀어준다(rtl_spin_unlock_irqrestore()). 복귀 값은 last_8254_time이다.

```
hrtimer_t pent_gettime(void)
{
    hrtimer_t t;

    /* time = counter * scaler_pentium_to_hrtimer / 2^32 * 2^5; */
    /* Why 2^5? Because the slowest Pentiums run at 60 MHz */

    __asm__ ("rdtsc\n\t"
            "mov %%edx, %%ecx\n\t"
            "mul %%ebx\n\t"           /* multiply the low 32 bits of the counter by the scaler_pentium */
            "mov %%ecx, %%eax\n\t"
            "mov %%edx, %%ecx\n\t"   /* save the high 32 bits of the product */
            "mul %%ebx\n\t"          /* now the high 32 bits of the counter */
            "add %%ecx, %%eax\n\t"
```

```

        "adc $0, %%edx\n\t"
#endif HRTICKS_PER_SEC == NSECS_PER_SEC
        "shld $5, %%eax, %%edx\n\t"
        "shl $5, %%eax\n\t"
#endif
        :"=A" (t) : "b" (scaler_pentium_to_hrtime) : "cx");
    return t;
}

```

코드 1291. TSC를 이용한 time구하기

TSC를 이용해서 time stamp를 구하는 것은 직접 TSC에 접근하기 위한 특수한 명령어를 사용한다(RDTSC : Read TSC). 이렇게 읽은 값은 다시 HRT값으로 변환하기 위해서 절절한 값으로 곱하게 되며(scaler_pentium_to_hrtime/2^32), 결과 값은 다시 HRTICKS_PER_SEC와 NSECS_PER_SEC값이 동일할 경우와 그렇지 않을 경우에 대해서 다시 계산된다(2^32). 결과 값은 t에 저장될 것이며, 복귀 값으로 사용한다.

여기서는 time를 구하는 것만을 보았다. 즉, 얼마정도의 resolution을 가지는 time를 구할 수 있는가가 hard real-time timer를 구현하는 필수 요소이기 때문에, 직접적으로 hardware에 접근해서 읽은 값으로 time을 구하고 있다. 실제로 특정한 시스템에 time을 주기 위해서는 다음과 같은 rtl_clock이라는 구조체가 필요하다.

```

struct rtl_clock {
    int (*init)(struct rtl_clock *);                                /* time을 초기화 한다.*/
    void (*uninit)(struct rtl_clock *);                             /* 초기화를 해제한다.*/
    hrtime_t (*gethrttime)(struct rtl_clock *);                     /* Hard real-time값을 구한다.*/
    int (*sethrttime)(struct rtl_clock *, hrtime_t t);            /* Hard real-time 값을 설정한다.*/
    int (*settimer)(struct rtl_clock *, hrtime_t interval);       /* Interval timer를 설정한다.*/
    int (*settimermode)(struct rtl_clock *, int mode);             /* Timer 모드를 설정한다.*/
    clock_irq_handler_t handler;                                    /* Interrupt를 처리한다 */
    int mode;                                                       /* 현재 timer의 모드 */
    hrtime_t resolution;                                         /* Time의 정확도(resolution) */
    hrtime_t value;                                                 /* Time */
    hrtime_t delta;                                                /* 시간 보정값 */
    pthread_spinlock_t lock;                                       /* Spinlock 설정 */
    struct rtl_clock_arch arch;                                     /* Architecture에 의존적인 값 */
};

```

코드 1292. rtl_clock 구조체의 정의

실제로 clock_nanosleep()과 같은 함수에서는 위와 같은 구현된 rtl_clock구조체의 gethrttime() 필드를 이용해서 해당 시스템의 현재 시간을 구하고 있다. 예를 들어서 8254 chip을 이용한다면, gethrttime() 필드에 들어갈 수 있는 것은 _gethrttime() 함수이며, 다시 이 함수는 gethrttime() 함수를 호출한다. 다시 이 함수는 앞에서 본 rtl_do_get_time()이라는 함수 포인터의 값으로 설정될 수 있는 global_8254_gettime() 함수나 pent_gettime() 함수를 호출하도록 되어있다.

그러면, 여기서 RTLinux상에서 자주 사용하게 되는 API중의 하나인 clock_nanosleep()에 대한 구현을 잠시 보도록 하자. 해당되는 함수로는 rtl_clock_nanosleep()이다. 정의는 RTLinux/schedulers/rtl_sched.c에 있다.

```

int rtl_clock_nanosleep(clockid_t clock_id, int flags,
                        const struct timespec *rqtp, struct timespec *rmtp)
{
    hrtime_t timeout;

```

```

rtl_irqstate_t irqstate;
hrtme_t save_resume_time;
pthread_t self = pthread_self();
int ret;

if (rqtp == (const struct timespec *) &self->timeval) {
    timeout = self->timeval;
} else {
    timeout = timespec_to_ns(rqtp);
}
rtl_no_interrupts (irqstate);
if (!(flags & TIMER_ABSTIME)) {
    timeout += clock_gettime(CLOCK_RTL_SCHED);
} else {
    timeout = __rtl_fix_timeout_for_clock(clock_id, timeout);
}
save_resume_time = self->resume_time;
RTL_MARK_SUSPENDED(self);
__rtl_setup_timeout (self, timeout);
ret = rtl_schedule();
pthread_testcancel();
if (RTL_TIMED_OUT(&ret)) {
    self->resume_time = save_resume_time;
    rtl_restore_interrupts (irqstate);
    return 0;
}
/* interrupted by a signal */
if (!(flags & TIMER_ABSTIME) && rmtp) {
    *rmtp = timespec_from_ns(self->resume_time - clock_gettime(CLOCK_RTL_SCHED));
}
self->resume_time = save_resume_time;
rtl_restore_interrupts (irqstate);
return EINTR;
}

```

코드 1293. `rtl_clock_nanosleep()` 함수의 정의

`rtl_clock_nanosleep()` 함수는 현재 thread를 특정한 시간까지 실행을 연기(suspend)하는 역할을 한다. 만약 flags에 `TIMER_ABSTIME`이 설정되어 있다면 `rqtp`에 있는 시간 값은 절대적인 시간이 되며, 그렇지 않다면 현재 시간으로부터 상대적인 시간까지를 말한다. `timespec` 구조체는 초 단위와 nanosecond 단위의 값을 가지는 필드로 구성된 구조체로서 특정 시간을 나타내기 위해서 사용하는 POSIX standard에 있는 자료구조이다. 만약 `rqtp`(Request Timer Period) 값이 현재 thread의 `timeval`과 같다면 `timeout` 값으로 `timeval`을 그대로 사용하고, 그렇지 않다면 nanosecond 단위로 `rqtp`를 고친 값으로 `timeout` 변수를 설정한다. 인터럽트의 발생을 금지 시킨 후에(`rtl_no_interrupts()`), flags에 `TIMER_ABSTIME`이 설정되었는지를 확인해서 다시 `timeout`을 바꾼다. 즉, 현재 시간부터를 이야기하는지 아니면 절대 시간을 이야기하는지를 보는 것이다. 그리고 나서 재 실행 시간(`save_resume_time`)을 thread의 `resume_time`으로 맞춘 후, 현재의 thread를 suspend 상태로 표시한다. `__rtl_setup_timeout()` 매크로 thread의 `resume_time`을 재설정하고, thread에 timer가 장전되었다는 것을 표시하며(`RTL_THREAD_TIMERARMED`), CPU별 스케줄링 데이터의 flag에 `RTL_THREAD_TIMERARMED` bit를 지운다. 이렇게 함으로써 스케줄링이 일어나게 될 때 timer를 설정한 task가 어떤 것인가를 확인하고, 시간 값을 비교해서 어떤 task가 `timeout`이 되었는지를 알 수 있게 된다. 마지막으로 다른 task가 실행될 수 있도록 `rtl_schedule()`⁵⁰⁰ 함수를 불러서 스케줄링이 일어나도록 한다. 이후의 코드는 현재 thread가 다시 스케줄링이 되어서 실행될 때가 될 것이다. `pthread_testcancel()` 함수는 thread가 cancel 시그널을 받았는지를 확인하는 함수이다. `RTL_TIMED_OUT()` 매크로는 `timeout`이 발생했는지를 확인하게 되며, 만약 `timeout`이 되었다면 앞에서 저장한

⁵⁰⁰ RTLinux의 스케줄링을 담당하는 함수이다. 다음 장에서 자세히 볼 것이다.

`save_resume_time`값으로 `thread`의 원래 `resume_time`값을 회복시켜주고, 앞에서 인터럽트 불가로 만들어준 부분을 가능상태로 만들어 주기 위해서 `rtl_restore_interrupts()` 함수를 불러준다. 복귀 값은 0이며, `timeout`이 제대로 된 경우가 된다.

그렇지 않고, `thread`가 깨어난 것이라면 이것은 특정 시그널에 의해서 `thread`가 인터럽트(방해)가 된 것이므로, `flags`에 설정된 값에 따라서 남은 시간(rmtp: Remain Time Period) 값을 다시 설정해 주고, `resume_time`에는 원래의 값(`save_resume_time`)을 준 후, `rtl_restore_interrupts()`를 호출해서 인터럽트 가능으로 만들고 나서 `EINTR`을 복귀 값으로 넘겨준다. 즉, 다른 `thread`나 운영체제로부터 인터럽트(방해)되었음을 알려주게 된다.

20.6. RTLinux의 Scheduler 구현

Linux 커널의 스케줄링을 담당하는 함수가 `schedule()` 함수이듯, RTLinux에서 `thread`의 스케줄링을 담당하는 함수는 `rtl_schedule()`이다. 정의는 RTLinux/schedulers/rtl_sched.c에 있다.

```
int rtl_schedule (void)
{
    schedule_t *sched;
    struct rtl_thread_struct *t;
    struct rtl_thread_struct *new_task;
    struct rtl_thread_struct *preemptor = 0;
    unsigned long interrupt_state;
    int ret;
    int cpu_id = rtl_getcpuid();
    hrtime_t now;

    rtl_no_interrupts(interrupt_state);
    rtl_trace2 (RTL_TRACE_SCHED_IN, (long) pthread_self());
    /* new_task = &sched->rtl_linux_task; */

    new_task = 0;
    sched = sched_data(cpu_id);

    now = sched->clock->gethrttime(sched->clock);

    if (sched->clock->mode == RTL_CLOCK_MODE_ONESHOT) {
        sched->clock->value = now;
    }
}
```

코드 1294. `rtl_schedule()` 함수의 정의

실제적인 모든 RTLinux의 스케줄링을 담당하는 함수는 `rtl_schedule()`이다. 이 함수의 중요한 역할은 먼저 RTLinux의 `thread`가 `ready`상태에 있는것이 있다면, 우선 순위가 높은 것을 먼저 실행시키고, 만약 `ready` 상태인 RTLinux `thread`가 없다면, Linux에게 실행의 기회를 넘겨 주는 것이다.

`rtl_no_interrupt()` 함수(매크로로 정의된다.)는 인터럽트의 발생을 멈추도록 하며, `rtl_trace2()` 함수는 단순히 RTLinux의 실행을 추적(tracing)하기 위한 것이다. 먼저 `new_task`를 0으로 두어 새로이 실행할 task가 어떤 것인가를 고른다. `sched_data()` 함수(역시 매크로이다.) CPU 별로 가지고 있는 스케줄링 `data`를 가져와서 현재 스케줄링 하는 시점을 구한다(`now`). CPU의 `clock`이 `RTL_CLOCK_MODE_ONESHOT`이라고 한다면, 이렇게 구한 값을 다시 `clock`의 값(`value`)로 두도록 한다.

```
for (t = sched->rtl_tasks; t = t->next) {
    /* expire timers */
    if (test_bit(RTL_THREAD_TIMERARMED, &t->threadflags)) {
        if (now >= t->resume_time) {
            clear_bit(RTL_THREAD_TIMERARMED, &t->threadflags);
            rtl_sigaddset (&t->pending, RTL_SIGNAL_TIMER);
            if (t->period != 0) { /* periodic */

```

```

        t->resume_time += t->period;
        /* timer overrun */
#endif CONFIG_RTL_OLD_TIMER_BEHAVIOUR
{
    while (now >= t->resume_time) {
        t->resume_time += t->period;
        rtl_printf("overrun");
    }
}
#endif
} else {
    t->resume_time = HRTIME_INFINITY;
}
}
}

/* and find highest priority runnable task */
if ((t->pending & ~t->blocked) && (!new_task ||
    (t->schedpar.sched_priority > new_task->schedpar.sched_priority))) {
    new_task = t;
}
}
}

```

코드 1295. rtl_schedule() 함수의 정의(계속)

이전 각 task가 가진 timer 값을 갱신(update)해서 timer가 expire되었는지를 확인하는 단계이다. RTLinux에서는 모든 thread(혹은 task)가 주기적으로 동작한다는 가정이 있기에 timer를 사용하는 것이 꼭 필요하다. RTLinux의 특정 task가 만약 주기적으로 동작하지 않고, 계속 실행되면 전체 시스템의 제어를 넘겨주지 않으므로 주의해야 한다. 각 CPU 별로 실행되는 task들은 sched->rtl_tasks에 연결리스트로 구성되어 있으므로, 여기서부터 찾아 나가면 될 것이다. Task의 timer가 있는지를 확인하고(test_bit()), 만약 task의 재개 시간(resume_time)이 현재 시간(now)보다 작은 경우에는 timer가 이미 지난 시간을 가르키므로 실행될 수 있도록 만들어주기 위해서 RTLinux의 signal인 RTL_SIGNAL_TIMER를 task의 pending signal의 집합에 설정한다(rtl_sigaddset()). 만약 task가 가진 timer가 주기적이라면(t->period != 0), 다음번 재개 시간을 설정하고(t->resume_time += t->period), 만약 이렇게 설정된 값이 현재 시간보다 작은 경우에는 주기를 계속 더해서 크도록 만든다. 즉, 이것은 하나의 task가 너무 짧은 주기로 설정되어 자주 실행되는 것을 막기 위한 것이다. 만약 주기 적인 task가 아니라면, 재개 시간에는 HRTIME_INFINITY를 두어서 특정한 event에만 task가 깨어나도록 만들어 준다.

이전 가장 높은 우선 순위를 가진 task를 찾아내는 것이다. 즉, task에 blocking되지 않은 pending signal이 있고, 새로운 task가 만약 0이라면(!new_task), 새로운 task에 현재 스케줄링을 하기 위해서 보고 있는 task를 넣어주고, 그렇기 않다면 우선 순위를 비교해서 (schedpar.sched_priority) 우선순위가 더 높은 task를 new_task로 만든다.

```

if (sched->clock->mode == RTL_CLOCK_MODE_ONESHOT && !test_bit (RTL_SCHED_TIMER_OK,
&sched->sched_flags)) {
    if ( (preemptor = find_preemptor(sched,new_task))) {
        (sched->clock)->settimer(sched->clock, preemptor->resume_time - now);
    } else {
        (sched->clock)->settimer(sched->clock, (HRTICKS_PER_SEC / HZ) / 2);
    }
    set_bit (RTL_SCHED_TIMER_OK, &sched->sched_flags);
}
ret = rtl_fast_switch(new_task);
rtl_restore_interrupts(interrupt_state);
return ret;
}

```

코드 1296. rtl_schedule() 함수의 정의(계속)

스케줄링 데이터의 clock 모드가 RTL_CLOCK_MODE_ONESHOT이고, RTL_SCHED_TIMER_OK가 스케줄링 데이터의 sched_flags에 설정이 되지 않았다면, if 절(clause)를 수행한다. find_preemptor()는 위에서 선택한 task(new_task)보다 더 우선 순위가 높으면서, resume_time이 더 짧은 task가 있는지를 확인하는 task로, 만약 이러한 task가 있다면, 이 task의 resume_time을 현재시간(now)에서 뺀 값으로 timer를 설정한다(settimer). 그러한 task가 없다면, HRTICKS_PER_SEC(=1000000000)⁵⁰¹를 HZ(=100)로 나눈 값을 다시 2로 나누어서 다음번 timer가 발생할 간격을 설정한다(settimer). 이전 timer의 설정이 되었다는 것을 나타내기 위해서 set_bit()를 호출해서 RTL_SCHED_TIMER_OK bit를 스케줄링 데이터의 flag(sched_flags)에 설정한다.

마지막으로 하는 일은 이전 rtl_fast_switch()를 호출해서 context switching을 하는 것이다. 이것을 호출한 후에는 현재의 task의 context가 아니다 다른 task의 context에서 진행하게 된다. 이후에 다시 이 task가 실행될 기회를 얻게 되면, 다시 rtl_fast_switch() 함수의 이하 부분을 수행할 것이다. 즉, 앞에서 한 interrupt를 멈추게 하는 것을 다시 가능하도록 만들어 주는 것이다(rtl_restore_interrupts()).

```
int inline rtl_fast_switch (pthread_t new_task)
{
    rtl_sigset_t mask;
    schedule_t *sched = sched_data(new_task->cpu); // cpu: we assume the caller has some brains

    if (new_task != sched->rtl_current) { /* switch out old, switch in new */
        if (new_task == &sched->rtl_linux_task) {
            rtl_make_rt_system_idle();
        } else {
            rtl_make_rt_system_active();
        }

        rtl_trace2 (RTL_TRACE_SCHED_CTX_SWITCH, (long) new_task);
        rtl_switch_to(&sched->rtl_current, new_task);
        rtl_schedule_tail(sched->rtl_current);
    }

    mask = pthread_self()->pending;

    if (pthread_self()->pending & ~(1 << RTL_SIGNAL_READY)) do_signal(pthread_self());

    rtl_trace2 (RTL_TRACE_SCHED_OUT, (long) pthread_self());
    return mask;
}
```

코드 1297. rtl_fast_switch() 함수의 정의

rtl_fast_switch() 함수는 inline으로 정의된 것이다. 따라서, 실제로 compile할 경우에는 함수의 호출이 일어나는 것이 아니라, 그 부분에 직접적으로 code가 삽입된다. 이것 역시 실행 속도를 높이기 위한 방법이다. 먼저 선택된 task가(new_task)가 현재 실행되고 있는 task가 아닌 경우에만 context switching한다. 만약 선택된 task가 Linux라면(즉, RTLinux thread가 아닌 Linux 본래의 것이라면), RTLinux를 idle상태로 만들고(rtl_make_rt_system_idle()), 그렇지 않다면 RTLinux를 active상태로 만든다(rtl_make_system_active). 실질적인 context swiching은 rtl_switch_to() 함수(혹은 macro)가 맡는다. 만약 나중에 다시 실행될 기회를 얻는다면, 현재 실행중인 task는 스케줄링을 위한 queue의 제일 마지막으로 보내지게 된다(rtl_schedule_tail()). mask 변수는 현재 스케줄링된 task의 pending된 signal을 보게 되며, 만약 pending된 signal에 RTL_SIGNAL_READY가 아닌 것이 있다면, 처리할 signal이 있다는 것을 알리기 위해서 do_signal()함수를 호출한다. 결과 값은 앞에서 설정한 mask가 된다.

⁵⁰¹ RTLinux는 nanosecond단위의 timer를 처리 할 수 있도록 설계되었다. 일반적인 Linux커널의 대부분의 activity는 jiffies라는 값에 의존하며, 이 값은 대략 10ms로 되어 있다. 따라서, RTLinux가 좀 더 real-time에 적합한 timer service를 제공해 줄 수 있을 것이다.

```
#define rtl_switch_to(current_task_ptr, new_task) \
    __asm__ __volatile__( \
    "pushl %%eax\n\t" \
    "pushl %%ebp\n\t" \
    "pushl %%edi\n\t" \
    "pushl %%esi\n\t" \
    "pushl %%edx\n\t" \
    "pushl %%ecx\n\t" \
    "pushl %%ebx\n\t" \
    "movl (%%ebx), %%edx\n\t/* get current */\n" \
    "pushl $1f\n\t" \
    "movl %%esp, (%%edx)\n\t" \
    "movl (%%ecx), %%esp\n\t" \
    "movl %%ecx, (%%ebx)\n\t/* store current */\n" \
    "ret\n\t" \
1:   "popl %%ebx\n\t" \
    "popl %%ecx\n\t" \
    "popl %%edx\n\t" \
    "popl %%esi\n\t" \
    "popl %%edi\n\t" \
    "popl %%ebp\n\t" \
    "popl %%eax\n\t" \
    : /* no output */ \
    : "c" (new_task), "b" (current_task_ptr) \
    );
```

코드 1298. `rtl_switch_to()` 함수(혹은 매크로)⁵⁰²의 정의

`rtl_switch_to()`는 매크로로 정의된 assembly언어를 사용하고 있다. 이와같은 assembly언어를 사용하는 것은 직접적으로 register레벨에서 바꿔치기를 해야 하기 때문이며, 빠른 실행을 보장하기 위한 방법이기도 하다. 정의는 RTLinux/include/i386/rtl_switch.h와 같은 곳에 각 architecture에 맞게 있을 것이다.

먼저 현재 사용하고 있는 register들을 저장해야 할 것이다. 즉, eax, ebp, edi, esi, edx, ecx, ebx를 push한다. 현재 ECX register에는 new_task가 EBX register에서는 current_task_ptr이 들어있다. 따라서, EBX register에 들어있는 current_task_ptr을 EDX register로 옮기고, 나중에 실행을 할 부분을 저장한다(push \$1f). 다시, ESP register를 현재 task를 가르키는 포인터인 EDX register로 넣어서 stack 포인터를 저장한다.⁵⁰³ 이젠 ECX register가 가리키는 stack으로 ESP register를 바꿔주고, 다시 ECX register를 EBX register가 가지도록 만든다. 즉, current_task_ptr은 이제 new_task를 가르키게 된다. 마지막으로 ret(return)을 실행하면, 앞에서 저장했던 1f부분으로 제어가 옮겨가게 될 것이다. 단, 여기서 중요한 것은 새로운 task(new_task)의 context내에서 진행해 간다는 점이다. 따라서, 앞에서 저장했던 각 register들을 반대되는 순서대로 pop한다.

종합하자면, RTLinux의 scheduling algorithm이라고 해야할 만한 것은 우선 순위에 기초를 둔 방법이라는 점과, 이러한 우선순위가 새로운 scheduling이 일어난다고 해서 바뀌지는 않는다는 점이다⁵⁰⁴. 따라서, 기존의 Linux에서 사용한 시간에 따른 우선 순위의 감소와 같은 것은 일어나지 않고, 항상 고정된 우선 순위를 사용해서 task를 실행해 준다는 것이다. 또한, 앞에서도 이미 이야기 했듯이 실행에 있어서는 task가 될 수 있으면 짧은 시간동안에만 CPU를 점할 수 있어야 하며, 이를 위해서 nanosleep()과 같은 API를 사용한다는 것이다. Scheduling시에는 이렇게 해서 잠든 task중에서 expired된 task만을 scheduling 대상으로 둔다. 다음으로 볼 것은 RTLinux thread와 Linux process간의 통신 통로 역할을 하게되는 RTLinux FIFO이다.

⁵⁰² 함수 혹은 매크로라고 하고 있는 이유는 실제로 그 부분이 반드시 macro로 정의되지는 않을 수도 있기 때문이다. 반드시 실제로 구현된 부분을 찾아서 확인하기 바란다. 여기서는 I386 architecture에 대한 것만 다룬다.

⁵⁰³ 이것은 앞에서 `rtl_thread_struct` 구조체를 볼 때, hard coding된 부분이기에 상관없다.

⁵⁰⁴ 바꾸기 위해서는 우선 순위를 다른 값으로 명시하는 RTLinux의 API를 사용해야 한다.

20.7. RTLinux의 FIFO(First In First Out)

RTLinux의 FIFO는 Linux process와 통신할 수 있는 창구 역할을 하는 매커니즘이다. Linux에서 말하는 FIFO와 거의 동일한 개념이지만, 일종의 device node의 형태로 구현해서 Linux process에서는 단지 이 device file을 open해서 읽고/쓰기를 수행할 수만 있다. RTLinux의 thread는 이러한 FIFO에 자신이 획득한 정보를 기입하면, 나중에 Linux process에서 이를 읽어간다. 구현은 RTLinux/fifos/rtl_fifo.c와 RTLinux/include/rtl_fifo.h를 참고하도록 하자.

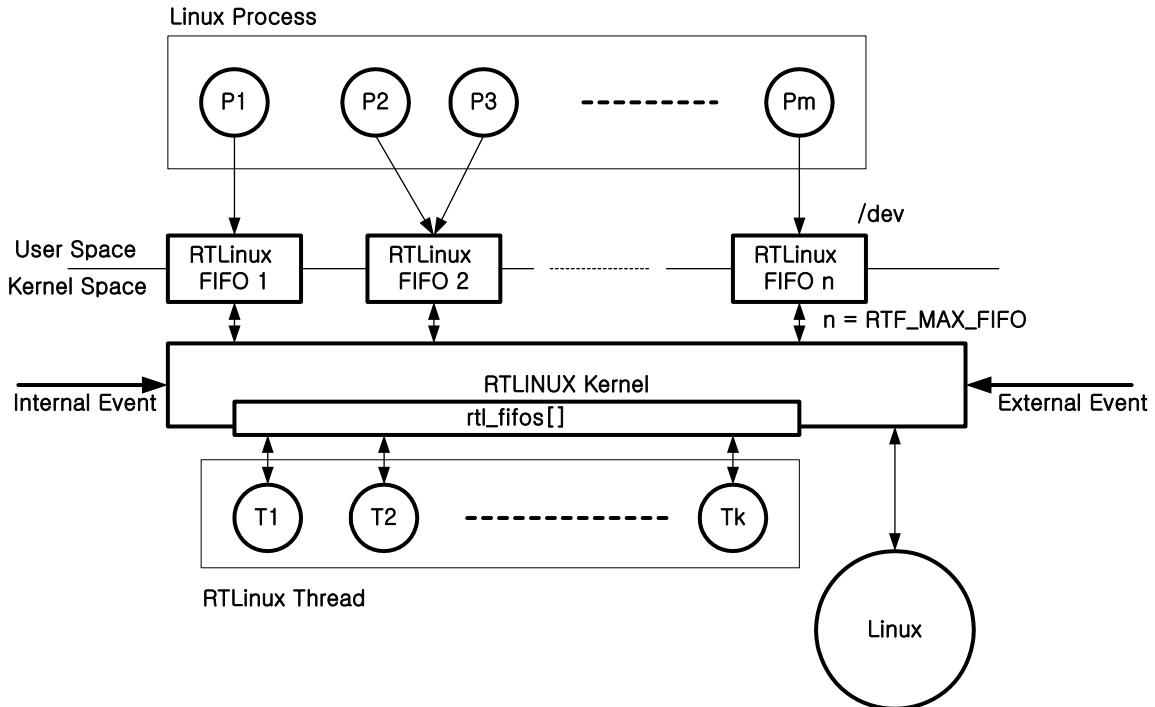


그림 140. RTLinux FIFO의 구조(Architecture)

[그림 140]은 RTLinux의 FIFO의 구성을 보여준다. 그림에서 보듯이 모든 external/internal한 event는 먼저 RTLinux의 관심이 되며, Linux porocess들은 사용자 영역에서 그리고, RTLinux thread들은 커널 영역에서 수행되기에 이들간의 연결 고리 역할을 하는 것이 바로 RTLinux의 FIFO라는 것을 알 수 있다. Linux 커널 자체는 RTLinux의 우선 순위가 가장 낮은 thread로 생각되기에 하위에 그렸다. 실제로는 thread라기보다, 커널 모드에서 실행중인 Linux process라고 보는 것이 더 정확할 것이다. 커널 자체는 process가 아니라, 일종의 library역할을 하기 때문이다. 자, 그럼 이젠 실제로 RTLinux의 FIFO를 어떻게 구현했는지를 보도록 하자.

```
struct rt_fifo_struct {
    int allocated;                                /* 할당되었는지를 표시한다. */
    int bidirectional;                            /* FIFO가 양방향(bidirectional)인가? */
    int user_open;                                /* 사용자가 open을 했는가? */
    char *base;                                   /* FIFO에서 사용하는 메모리의 base */
    int bufsize;                                  /* Buffer의 크기 */
    int start;                                    /* FIFO의 start offset */
    int len;                                      /* 길이( Length ) */
    rtl_spinlock_t fifo_spinlock;                /* FIFO에 대한 spin lock */
    int (*user_handler)(unsigned int fifo);        /* 사용자 handler 함수의 pointer */
    int (*rt_handler)(unsigned int fifo);          /* RTLinux handler 함수의 pointer */
    int (*user_ioctl)(unsigned int fifo, unsigned int cmd, unsigned long arg); /* 사용자 ioctl 함수의 pointer */
    struct module *creator;                      /* FIFO의 생성자(creator) 모듈 정보 */
};
```

```
};
```

코드 1299. rt_fifo_struct 구조체의 정의

RTLinux FIFO는 `rt_fifo_struct` 구조체로 관리된다. 이 구조체를 가지고 실제로 생성되는 것은 `rtl_fifo[]`이며, 각각은 실제로 /dev 디렉토리 이하에 node를 가진다.

20.7.1. FIFO 모듈의 초기화와 해제

RTLinux의 FIFO는 일종의 디바이스로서 사용자에게 보여진다는 것을 이미 앞에서 이야기 했다. 따라서, FIFO module을 loading할 때 이러한 디바이스로서의 역할을 수행하기 위한 등록 과정이 있어야 할 것이며, 다시 해제시에는 반대되는 일을 해줄 것이다. 아래와 같다.

```
int init_module(void)
{
    int ret;
    ret = rtf_init();
    if (ret < 0) {
        return ret;
    }
#ifndef CONFIG_RTL_POSIX_IO
    if (rtl_register_chrdev(RTF_MAJOR, "rtf", &rtl_fifo_fops)) {
        printk("RT-FIFO: unable to get RTLinux major %d\n", RTF_MAJOR);
        rtf_uninit();
        return -EIO;
    }
#endif
    return 0;
}
void cleanup_module(void)
{
#ifndef CONFIG_RTL_POSIX_IO
    rtl_unregister_chrdev(RTF_MAJOR, "rtf");
#endif
    rtf_uninit();
}
```

코드 1300. init_module() 함수와 cleanup_module() 함수의 정의

`init_module()` 함수에서는 `rtf_init()` 함수를 호출해서 FIFO에 대한 초기화를 수행하게 되며, 그 `return` 값을 돌려준다. 만약 RTLinux가 POSIX standard에서 정한 I/O interface를 제공한다면, `rtl_register_chrdev()` 함수를 호출해서 RTLinux의 thread에서도 사용하는 문자 디바이스로 등록한다. `RTF_MAJOR(=150505)`를 major번호로, `rtf`를 그 이름으로 그리고, 해당 FIFO에 대한 연산을 수행하기 위한 연산 vector로는 `rtl_fifo_fops` 구조체를 준다. 만약 위와 같은 과정에서 error가 발생하면, `rtf_uninit()` 함수를 수행해서 `undo`를 한다. 예러값은 `-EIO`이다. 모든 과정에서 문제가 없다면 0을 돌려줄 것이다.

`cleanup_module()`은 앞에서 본 것을 반대로 수행하면 된다. POSIX standard I/O를 지원한다면, `rtl_unregister_chrdev()`를 호출해서 RTLinux 문자 디바이스에 대한 entry를 제거하고, 다시 `rtf_uninit()`를 수행한다.

```
#define MAX_RTL_FILES 128
/* RTLinux에서 사용하는 파일 object들의 배열 */
static struct rtl_file rtl_files[MAX_RTL_FILES] /* TODO __attribute__ ((aligned (64))) */ = {
    { NULL, 0 },
};
```

⁵⁰⁵ Linux인 경우이다. NetBSD와 같은 곳에서는 다른 번호를 사용한다.

```

struct device_struct {
    const char * name;                                /* 장치의 이름 */
    struct rtl_file_operations * fops;                /* 관련된 file 연산 vector의 포인터 */
};

/* 모든 RTLinux 디바이스들을 관리하기 위한 RTLinux device의 배열 */
static struct device_struct rtldevs[RTL_MAX_DEV] = {
    { NULL, NULL },
};

int rtl_register_rtldev(unsigned int major, const char * name, struct rtl_file_operations *fops)
{
    if (major >= RTL_MAX_DEV)
        return -EINVAL;
    if (rtldevs[major].fops && rtldevs[major].fops != fops) {
        return -EBUSY;
    }
    rtldevs[major].name = name;
    rtldevs[major].fops = fops;
    return 0;
}

int rtl_unregister_rtldev(unsigned int major, const char * name)
{
    if (major >= RTL_MAX_DEV)
        return -EINVAL;
    if (!rtldevs[major].fops)
        return -EINVAL;
    if (strcmp(rtldevs[major].name, name))
        return -EINVAL;
    rtldevs[major].name = NULL;
    rtldevs[major].fops = NULL;
    return 0;
}

```

코드 1301. `rtl_register_rtldev()` 함수와 `rtl_unregister_rtldev()` 함수의 정의

`rtl_register_chrdev()` 함수와 `rtl_unregister_chrdev()` 함수는 실제로 `rtl_register_rtldev()` 함수와 `rtl_unregister_rtldev()` 함수로 정의되어 있다. 이 함수들의 역할은 간단히 RTLinux에서 사용하는 디바이스 구조체의 배열에 넣고 빼는 역할을 수행한다. `rtl_register_rtldev()` 함수는 빈 자리가 없으면 -EBUSY를, 그렇지 않으면, MAJOR번호에 맞는 곳에 장치의 이름과 연산 vector를 넣는 일을 하며, `rtl_unregister_rtldev()` 함수는 MAJOR번호를 index로 사용해서 원래 등록된 자리의 장치 이름과 연산 vector에 NULL로 두는 일을 한다.

```

int rtf_init (void)
{
    int irq = -1;
    int i;

    if (register_chrdev (RTF_MAJOR, "rtf", &rtf_fops)) {
        printk ("RT-FIFO: unable to get major %d\n", RTF_MAJOR);
        return -EIO;
    }
    for (i = 0; i < RTF_MAX_FIFO; i++) {
        rtl_fifo_to_wakeup[i] = 0;
    }
    irq = rtl_get_soft_irq (fifo_irq_handler, "RTLinux FIFO");

```

```

if (irq > 0) {
    rtl_fifo_irq = irq;
} else {
    unregister_chrdev(RTF_MAJOR, "rtf");
    printk("Can't get an irq for rt fifos");
    return -EIO; /* should have a different return */
}
return 0;
}

void rtf_uninit(void)
{
    if (rtl_fifo_irq) {
        rtl_free_soft_irq(rtl_fifo_irq);
    }
    unregister_chrdev(RTF_MAJOR, "rtf");
}

```

코드 1302. rtf_init() 함수와 rtf_uninit() 함수의 정의

rtf_init()는 RTF_MAJOR(=150)으로 “rtf”라는 문자 디바이스 드라이버로 등록하는 역할을 한다. 관련된 연산 vector는 rtf_fops 구조체이다. 실패할 경우에는 -EIO를 돌려준다. 그리고나서, RTLinux FIFO에서 wakeup되기를 원하는 list를 관리하는 rtl_fifo_to_wakeup[] 배열을 초기화한다. 이와같이 한 후, RTLinux FIFO가 처리하는 software interrupt를 위한 handler로 fifo_irq_handler를 등록시키고(rtl_get_soft_irq()), 등록한 RTLinux 인터럽트 번호(irq) rtl_fifo_irq에 넣어준다. 예러가 있을 때는 문자 디바이스 드라이버의 등록을 해제하고, -EIO를 돌려준다. 복귀값은 0이다.

rtf_uninit()는 역시 앞에서 했던 일을 반대로 수행한다. 즉, soft interrupt handler를 해제하고(rtl_free_soft_irq()), 문자 디바이스 드라이버로 등록된 “rtf”를 해제(unregister)한다.

```

static struct file_operations rtf_fops =
{
    llseek: rtf_llseek,
    read: rtf_read,
    write: rtf_write,
    poll: rtf_poll,
    ioctl: rtf_ioctl,
    open: rtf_open,
    release: rtf_release,
};

```

코드 1303. rtf_fops 구조체의 정의

rtf_fops구조체는 file 연산을 위한 함수들에 대한 정보를 가진다. 나중에 사용자 프로세스가 파일에 대한 연산을 수행할 때, 이 함수들을 부를 것이다. 정의는 RTLinux/main/linux_sys/rtl_fifoP.h에 있다. 관련해서 CONFIG_RTL POSIX_IO가 선택되었을 때 RTLinux의 문자 디바이스 드라이버로 등록할 때 사용되는 rtl_fifo_fops 구조체는 다음과 같이 정의된다.

```

static struct rtl_file_operations rtl_fifo_fops = {
    NULL,
    rtl_rtf_read,
    rtl_rtf_write,
    rtl_rtf_ioctl,
    NULL,
    rtl_rtf_open,
    rtl_rtf_release
};

```

코드 1304. rtl_fifo_fops 구조체의 정의

rtl_fifo_fops 구조체는 rtl_file_operations라는 타입의 구조체이며, 거의 Linux의 file operation 구조체와 동일하다고 생각할 수 있다. 여기에 있는 함수들은 RTLinux thread(혹은 task)가 호출하는 함수들로 구성된다. rtf_fops 구조체는 실제로 PIPE와 같은 것을 구현하는 것과 별다른 차이가 없으므로 분석하지 않고, rtl_fifo_fops 구조체를 분석하도록 하겠다.⁵⁰⁶

```
struct rtl_file {
    struct rtl_file_operations *f_op;           /* 파일 연산 vector에 대한 포인터 */
    int f_minor;                                /* Minor 번호 */
    int f_flags;                                 /* RTLinux file 구조체의 flag */
    off_t          f_pos;                        /* 현재 offset : read/write position */
};

struct rtl_file_operations {
    off_t (*llseek) (struct rtl_file *, off_t, int);
    ssize_t (*read) (struct rtl_file *, char *, size_t, off_t *);
    ssize_t (*write) (struct rtl_file *, const char *, size_t, off_t *);
    int (*ioctl) (struct rtl_file *, unsigned int, unsigned long);
    int (*mmap) (struct rtl_file *, void *start, size_t length, int prot, int flags, off_t offset, caddr_t *result);
    int (*open) (struct rtl_file *);
    int (*release) (struct rtl_file *);
};
```

코드 1305. rtl_file 및 rtl_file_operations 구조체의 정의

먼저, 각 함수를 분석하기 위해서는 rtl_file 구조체와 rtl_file_operations 구조체를 보아야 할 것이다. 코드는 RTLinux/include/rtl_posixio.h에 있다. rtl_file은 실제적인 Linux의 file object를 생성하는 것이 아니라, RTLinux에서 사용하는 데이터 구조체이다.

rtl_file 구조체는 rtl_file_operations 구조체에 정의된 각 연산 vector의 첫번째 인자로서 넘어가게 된다. 각각의 연산 함수들이 하는 역할은 일반적인 Linux 문자 디바이스 드라이버에서 정의한 함수들과 크게 다르지 않다.⁵⁰⁷ 따라서, 여기서 부가적인 설명은 하지 않는다.

20.7.2. Open/Release 함수

```
static int rtl_rtf_open (struct rtl_file *filp)
{
    if (!(filp->f_flags & O_NONBLOCK)) {
        return -EACCES; /* TODO: implement blocking IO */
    }
    if( (filp->f_flags & O_CREAT) && !RTF_ALLOCATED(filp->f_minor))
    {
        /* better be calling from Linux and not RT mode unless
         there are preallocted fifos still */
        __rtf_create(filp->f_minor, RTF_DEFAULT_SIZE, &__this_module);
    }
    if(!RTF_ALLOCATED(filp->f_minor)){
        return -ENODEV; // protocol driver not attached
    }
    return 0;
}
```

⁵⁰⁶ 자세히 알고자 하는 사람은 RTLinux/main/linux_sys/rtl_fifoP.h를 참조하라.

⁵⁰⁷ RTLinux는 많은 부분에서 Linux 커널 구현에 사용된 trick을 동일하게 사용하고 있다. 그렇다고, 독창성이 없는 것은 아니므로 이해하지 말도록 하자.

```
static int rtl_rtf_release (struct rtl_file *filp)
{
    int minor = filp->f_minor;
    char *old = RTF_BASE(minor);
    if (RTF_ALLOCATED(minor) && old && (filp->f_flags & O_CREAT)) {
        rtf_destroy(minor);
    }
    return 0;
}
```

코드 1306. rtl_rtf_open() 함수와 rtl_rtf_release() 함수의 정의

rtl_rtf_open() 함수는 rtl_file 구조체를 넘겨받는다. 먼저 rtl_file 구조체의 file flag에 O_NONBLOCK 설정되지 않았다면, -EACCESS를 돌려준다. 다시 O_CREATE를 flag에 설정된 경우, minor 번호를 보고 이미 할당된 것인가를 확인한다(RTF_ALLOCATED()). 할당된 것이 아니라면, _rtf_create()를 호출해서 기본 크기로(RTF_DEFAULT_SIZE=8192), 이 모듈을 owner로 해서 새로 생성하도록 한다. 만약 이미 할당된 것이라고 할 때는 -ENODEV를 돌려준다. 복귀값은 0이 될 것이다.

rtl_rtf_release() 함수는 앞에서 open한 rtl_file을 없애주는 일을 한다. Minor번호에 해당하는 것이 할당되었으며, FIFO가 기본주소를 가지고 있고, rtl_file 구조체의 flag에 O_CREAT가 설정되어 생성된 것이라면, rtf_destroy()를 호출해서 minor번호에 해당하는 것을 없애주는 것이다. 복귀 값은 0이다.

```
int __rtf_create(unsigned int minor, int size, struct module *creator)
{
    int ret;

    if (minor >= RTF_MAX_FIFO) {
        return -ENODEV;
    }
    if (RTF_ALLOCATED(minor)) {
        return -EBUSY;
    }
    rtl_spin_lock_init(&RTF_SPIN(minor));
    RTF_BI(minor) = 0;
    if ((ret = rtf_resize(minor, size)) < 0) {
        return -ENOMEM;
    }
    RTF_ADDR(minor)->creator = creator;
    RTF_USER_OPEN(minor) = 0;
    RTF_HANDLER(minor) = &default_handler;
    RTF_RT_HANDLER(minor) = &default_handler;
    RTF_USER_IOCTL(minor) = 0;

    RTF_INIT_GPOS(minor);
    RTF_ALLOCATED(minor) = 1;
    return 0;
}
```

코드 1307. __rtf_create() 함수의 정의

__rtf_create() 함수는 minor번호와 크기, 그리고 생성자의 모듈을 나타내는 포인터를 넘겨받는다. 넘겨받은 파라미터 값에 대한 오류에 따라 -ENODEV나 혹은 -EBUSY를 돌려주고, 연산에 들어가기 전에 spin lock을 초기화한다(rtl_spin_lock_init()). Bidirection field에는 0을 넣어서 초기화 시키고, rtf_resize() 함수를 호출해서 크기 만큼의 메모리를 할당 받는다. 여기서 에러가 있다면 -ENOMEM을 돌려줄 것이다. 생성자를 나타내는 필드에는 모듈의 포인터를 넣어주고, open은 아직 된 것이 아니므로 0을 두고, USER handler와 RTLinux handler에는 default_handler() 함수의 주소를 넣어준다. 사용자의 함수는

0으로 초기화 하고, RTF_INIT_GPOS() 매크로를 이용해서 일반 Linux process를 나타내기 위한 구조체를 초기화 시킨다. 마지막으로 return하기 전에 할당되었다는 것을 나타내는 것도 잊지 말도록 하자. 복귀 값은 0이다. default_handler() 함수는 단순히 0을 돌려주는 일만 한다.

```
#include <linux/vmalloc.h>
#ifndef LINUX_VERSION_CODE >= 0x020300
#define RTF_GPOS struct { \
    struct task_struct *opener; \
    wait_queue_head_t wait; }
#define RTF_INIT_GPOS(minor) \
    RTF_OPENER(minor) = 0; \
    init_waitqueue_head(&RTF_WAIT(minor));
#else
#define RTF_GPOS struct { \
    struct task_struct *opener; \
    struct wait_queue *wait; }
#define RTF_INIT_GPOS(minor) \
    RTF_OPENER(minor) = 0; \
    init_waitqueue_head(&RTF_WAIT(minor));
#endif

static RTF_GPOS rtl_gpos[RTF_MAX_FIFO];
#define RTF_WAIT(minor) (rtl_gpos[minor].wait)
#define RTF_OPENER(minor) (rtl_gpos[minor].opener)
#define fifo_alloc(size) vmalloc(size)
#define fifo_free(ptr) vfree(ptr)
```

코드 1308. RTF_XXX 매크로들의 정의

RTF_GPOS 구조체는 일종의 FIFO같은 file과 관련된 연산에 필요한 대기 queue를 만들기 위해서 사용하고 있다. 즉, 일반적인 Linux상에서 프로세스들이 입출력이 완료되는 것을 대기하기 위한 queue이다. RTF_INIT_GPOS()는 이 queue를 초기화 하는 것이며, 이것을 초기화 하는 방법이 Linux 커널 버전에 따라 달라진다. 이러한 queue들은 FIFO의 갯수 만큼 가지기 위해서 rtl_gpos[] 배열을 사용한다. fifo_alloc()이나 fifo_free()는 FIFO에 사용할 메모리를 할당하는 함수들로 Linux에 있는 커널 API를 차용해서 쓴다. 여기서 보여주지 않은 매크로나 정의들은 나중에 다시 보게 될 것이므로 다음으로 넘어가도록 하겠다.

```
int rtf_destroy(unsigned int minor)
{
    if (minor >= RTF_MAX_FIFO) {
        return -ENODEV;
    }
    if (RTF_USER_OPEN(minor)) {
        return -EINVAL;
    }
    if (!RTF_ALLOCATED(minor)) {
        return -EINVAL;
    }
    RTF_ADDR(minor)->creator = 0;
    RTF_HANDLER(minor) = &default_handler;
    RTF_RT_HANDLER(minor) = &default_handler;
    if (!free_prealloc(RTF_BASE(minor))){
        fifo_free(RTF_BASE(minor));
    }
    RTF_ALLOCATED(minor) = 0;
    return 0;
```

{

코드 1309. rtf_destroy() 함수의 정의

`rtf_destroy()` 함수는 `_rtf_create()` 함수의 역을 수행한다. 먼저 파라미터 값이 올바른 가를 확인하고, 그렇지 않다면 에러값으로 `-ENODEV`나 `-EINVAL`을 돌려줄 것이다. 할당된 것이 아닌 것을 지우려고(destroy)하면 다시 `-EINVAL`을 돌려줄 것이다. 생성자는 `NULL`(혹은 `0`)으로 두고, 사용자 handler와 RTLinux handler를 모두 `default_handle()` 함수로 만든다. 만약 미리 할당된(pre-allocated) 영역에서 메모리를 해제한다면, `free_preallo()`를 호출하고, 그렇지 않다면 `fifo_free()`를 호출해서 메모리를 제거한다. 마지막으로 할당되지 않았다는 것을 표시한다(`RTF_ALLOCATED()`). 복귀값은 `0`이다.

20.7.3. 기타 FIFO와 관련된 함수들

여기서 보는 함수들은 직접적으로 real-time task나 인터럽트 핸들러에서 호출되어 간단한 FIFO의 정보를 알려주거나 FIFO에 연산을 하는 것들이다.

```
int rtf_isempty(unsigned int minor)
{
    return RTF_LEN(minor) == 0;
}

int rtf_isused(unsigned int minor)
{
    return RTF_USER_OPEN(minor) != 0;
}

int rtf_flush(unsigned int minor)
{
    rtl_irqstate_t interrupt_state;
    rtl_spin_lock_irqsave(&RTF_SPIN(minor), interrupt_state);
    RTF_LEN(minor) = 0;
    rtl_spin_unlock_irqrestore(&RTF_SPIN(minor), interrupt_state);
    return 0;
}
```

코드 1310. rtf_isempty()/rtf_isused()/rtf_flush() 함수의 정의

`rtf_isempty()` 함수는 단순히 FIFO가 비었는지를 확인하는 역할을 하며, `rtf_isused()` 함수는 사용자가 `open`을 했는지를 알려준다. `rtf_flush()` 함수는 FIFO를 비워주는 일을 하는데, 실제적으로는 코드에서 보듯이 FIFO의 데이터 길이가 `0`이라고 하는 것으로 끝난다. 하지만, 이렇게 하기 위해서는 `spin lock`을 설정하고, 해제하는 것이 필요하다. 즉, 다른 thread나 프로세스의 간섭을 배제하기 위한 것이다.

20.7.4. Read/Write 함수

RTLinux FIFO의 `read/write`시에 사용하는 함수는 `rtl_rtf_write()`와 `rtl_rtf_read()`이다. 이 두 함수는 `rtl_file` 구조체와 `buffer`, `buffer 크기` 및 연산이 일어나는 위치를 나타내는 `offset`을 파라미터 값으로 넘겨 받는다. 정의는 아래와 같다.

```
static ssize_t rtl_rtf_write(struct rtl_file *filp, const char *buf, size_t count, off_t* ppos)
{
    return rtf_put(RTL_MINOR_FROM_FILEPTR(filp), (char *) buf, count);
}

static ssize_t rtl_rtf_read(struct rtl_file *filp, char *buf, size_t count, off_t* ppos)
{
    return rtf_get(RTL_MINOR_FROM_FILEPTR(filp), buf, count);
```

```
}
```

코드 1311. rtl_rtf_write() 함수와 rtl_rtf_read() 함수의 정의

이 두 함수는 넘겨받은 rtl_file 구조체로부터 장치의 minor번호를 구해서 다시 rtf_put()과 rtf_get() 함수를 각각 호출한다.

```
int rtf_put(unsigned int minor, void *buf, int count)
{
    rtl_irqstate_t interrupt_state;
    int chars = 0, free = 0, written = 0;
    char *pipebuf;

    if (minor >= RTF_MAX_FIFO) {
        return -ENODEV;
    }
    if (!RTF_ALLOCATED(minor))
        return -EINVAL;
    rtl_spin_lock_irqsave(&RTF_SPIN(minor), interrupt_state);
    if (RTF_FREE(minor) < count) {
        rtl_spin_unlock_irqrestore(&RTF_SPIN(minor), interrupt_state);
        return -ENOSPC;
    }
    while (count > 0 && (free = RTF_FREE(minor))) {
        chars = RTF_MAX_WCHUNK(minor);
        if (chars > count)
            chars = count;
        if (chars > free)
            chars = free;
        pipebuf = RTF_BASE(minor) + RTF_END(minor);
        written += chars;
        RTF_LEN(minor) += chars;
        count -= chars;
        memcpy(pipebuf, buf, chars);
        buf += chars;
    }
    rtl_spin_unlock_irqrestore(&RTF_SPIN(minor), interrupt_state);
    (*RTF_RT_HANDLER(minor))(minor);
    if (RTF_USER_OPEN(minor)) {
        fifo_wake_sleepers(minor - (RTF_BI(minor) < 0));
    }
    return written;
}
```

코드 1312. rtf_put() 함수의 정의

다시 rtf_put() 함수는 먼저 넘겨받은 minor번호가 RTF_MAX_FIFO보다 크거나 같은 값을 가지는지를 확인해서 에러가 있다면, -ENODEV를 넘겨주고, minor번호에 해당하는 디바이스가 현재 할당(allocate)되었는지를 본다. 할당된 것이 아니라면 -EINVAL을 돌려준다. 이전 해당 연산을 시작해야 하기에 공유되는 자료구조를 보호하기 위해서 rtl_spin_lock_irqsave()를 호출해서 인터럽트의 영향을 받지 않도록 만든다. 만약 요구 받은 크기(count)보다 작은 공간이 있다면 -ENOSPC를 돌려준다. 즉, 더 이상의 FIFO에 공간이 없다는 것을 뜻한다. 이전 while() loop를 돌면서 FIFO에 넘겨받은 buffer의 데이터를 copy(memcpy)한다. 여기서 사용하는 매크로들의 정의는 다음과 같다.

#define RTF_ADDR(minor)	(&rtl_fifos[minor])	/* Point to rtl_fifos[] element]
-------------------------	---------------------	-----------------------------------

```

#define RTF_BI(minor)           (RTF_ADDR(minor)->bidirectional) /* Bidirectional ? */
#define RTF_ALLOCATED(minor)    (RTF_ADDR(minor)->allocated)   /* Allocated ? */
#define RTF_USER_OPEN(minor)    (RTF_ADDR(minor)->user_open)   /* User open ? */
#define RTF_BASE(minor)         (RTF_ADDR(minor)->base)        /* Base address */
#define RTF_SPIN(minor)         (RTF_ADDR(minor)->fifo_spinlock) /* FIFO spinlock */
#define RTF_BUF(minor)          (RTF_ADDR(minor)->bufsize)     /* Buffer size */
#define RTF_START(minor)        (RTF_ADDR(minor)->start)       /* Start */
#define RTF_HANDLER(minor)      (RTF_ADDR(minor)->user_handler) /* User handler function */
#define RTF_RT_HANDLER(minor)   (RTF_ADDR(minor)->rt_handler)  /* RTLinux handler function */
#define RTF_USER_IOCTL(minor)   (RTF_ADDR(minor)->user_ioctl)  /* User IOCTL function */
#define RTF_LEN(minor)          (RTF_ADDR(minor)->len)        /* FIFO length */
#define RTF_FREE(minor)         (RTF_BUF(minor) - RTF_LEN(minor)) /* Free space length */

#define RTF_WRAP(minor,pos)     ((pos) < RTF_BUF(minor)? (pos) : (pos) - RTF_BUF(minor)) /* Wrap : Ring */
*/
#define RTF_END(minor)          RTF_WRAP(minor, RTF_START(minor)+RTF_LEN(minor)) /* End of
FIFO */
#define RTF_EMPTY(minor)        (RTF_LEN(minor)==0)           /* FIFO empty ? */
#define RTF_FULL(minor)         (RTF_FREE(minor)==0)           /* FIFO full ? */
#define RTF_MAX_RCHUNK(minor)  (RTF_BUF(minor) - RTF_START(minor)) /* Read chunk : 읽기크기 */
#define RTF_MAX_WCHUNK(minor)(RTF_BUF(minor) - RTF_END(minor)) /* Write chunk : 쓰기크기 */

```

코드 1313. RTLinux FIFO의 매크로 정의

FIFO에 대한 쓰기를 마치면, 앞에서 획득한 lock를 해제(rtl_spin_unlock_irqrestore())한다. 만약 RTLinux가 설치한 FIFO와 관련된 handler가 있다면, 여기서 이것을 실행해준다. 또한 사용자가 RTLinux FIFO를 open했다면⁵⁰⁸, 사용자 FIFO에서 잠들어 있는 프로세스들을 깨우기 위해서 fifo_wake_sleepers() 함수를 호출한다. 함수의 복귀 값은 write된 크기이다.

```

int rtf_get(unsigned int minor, void *buf, int count)
{
    rtl_irqstate_t interrupt_state;
    int chars = 0, size = 0, read = 0;
    char *pipebuf;

    if (minor >= RTF_MAX_FIFO) {
        return -ENODEV;
    }
    if (!RTF_ALLOCATED(minor))
        return -EINVAL;
    rtl_spin_lock_irqsave(&RTF_SPIN(minor), interrupt_state);
    while (count > 0 && (size = RTF_LEN(minor))) {
        chars = RTF_MAX_RCHUNK(minor);
        if (chars > count)
            chars = count;
        if (chars > size)
            chars = size;
        read += chars;
        pipebuf = RTF_BASE(minor) + RTF_START(minor);
        RTF_START(minor) += chars;
        RTF_START(minor) = RTF_WRAP(minor, RTF_START(minor));
        RTF_LEN(minor) -= chars;
        count -= chars;
        memcpy(buf, pipebuf, chars);
        buf += chars;
    }
}

```

⁵⁰⁸ 현재로서는 GDB를 이용한 debugging에서만 사용하는 것 같다.

```

    }
    rtl_spin_unlock_irqrestore(&RTF_SPIN(minor), interrupt_state);
    (*RTF_RT_HANDLER(minor))(minor);
    if (RTF_USER_OPEN(minor)) {
        fifo_wake_sleepers(minor);
    }
    return read;
}

```

코드 1314. rtf_get() 함수의 정의

rtf_get() 함수도 rtf_put() 함수와 거의 유사한 구조를 가진다. 물론 rtf_get() 함수는 데이터를 읽어간다는 점이 다르다. 읽어간 데이터는 FIFO에 남지 않는다는 점에 유념하기 바란다. 먼저 minor번호가 RTF_MAX_FIFO보가 크거나 같은지를 확인해서 오류가 있다면 -ENODEV를 돌려준다. 또한 할당된 것이 아닌 경우에는 -EINVAL을 돌려준다. 연산의 시작과 끝부분에는 인터럽트의 영향을 받지 않도록 spinlock을 설정하고, 실제적인 처리는 while() loop를 돌면서 된다. 읽을 수 있는 최대 크기를 찾고, 이를 이용해서 사용자의 buffer에 memcpy() 함수를 사용해서 메모리 copy를 시켜준다. 만약 RTLinux handler가 있다면, 이를 호출해서 처리하고, 사용자가 open하고 있는 상황이라면, 잠들어 있을지도 모르는 process를 깨우기 위해서 fifo_wake_sleepers() 함수를 호출한다. 함수의 복귀값은 읽은 크기가 될 것이다.

20.7.5. IOCTL 함수

IOCTL 함수의 경우 일반적인 Linux에서는 특정한 디바이스에 특정한 명령을 수행하기 위해서 호출되는 메쏘드(method)이다. 여기서도 마찬가지다.⁵⁰⁹

```

static int rtl_rtf_ioctl (struct rtl_file *filp, unsigned int req, unsigned long arg)
{
    int minor = RTL_MINOR_FROM_FILEPTR(filp);
    if (!RTF_ALLOCATED(minor)) {
        return -EINVAL;
    }
    if (req == RTF_SETSIZE) {
        if (rtf_resize(minor, arg) < 0) {
            return -EINVAL;
        }
    } else {
        return -EINVAL;
    }
    return 0;
}

```

코드 1315. rtl_rtf_ioctl() 함수의 정의

rtl_rtf_ioctl() 함수는 넘겨받은 rtl_file 구조체의 포인터로 부터 minor번호를 찾은 후, 해당 minor번호가 할당되었는지를 확인한다. 만약 할당되지 않았다면 -EINVAL을 돌려준다. 처리하는 명령(requset)는 단지 RTF_SETSIZE밖에 없다. 나머지에 대해서는 전부 -EINVAL을 돌려준다. 복귀값은 0이다. 실제적인 크기 변화(RTF_SETSIZE)를 처리하는 함수는 rtf_resize()이다.

```

/* These are for use in the init and exit code of real-time modules
DO NOT call these from a RT task */

```

⁵⁰⁹ 일반적으로 모든 UNIX와 같은 계열의 운영체제에 속하는 디바이스 드라이버에서는 이와 같은 방법으로 디바이스를 제어(control)하는 것을 허가하고 있다. 따라서, 대부분의 디바이스 드라이버는 거의 비슷한 context를 취하고 있는데, windows와 같은 경우에도 디바이스에 대한 처리를 file이라는 개념으로 하고 있기에 다르다고 보지는 않는다. 하지만, 내부적으로 IRP(I/O Request Packet)과 같은 구조등으로 인해서 달라지게 된다. 하지만, 사용자의 입장에서는 거의 동일하게 모든 장치를 접근할 수 있다.

```

int rtf_resize(unsigned int minor, int size)
{
    void *mem=0;
    void *old;
    rtl_irqstate_t interrupt_state;

    if (size <= 0) {
        return -EINVAL;
    }
    if (minor >= RTF_MAX_FIFO) {
        return -ENODEV;
    }
    if(size == PREALLOC_SIZE){
        mem=get_prealloc();
    }
    if(!mem){
        if (!rtl_rt_system_is_idle()) {
            return -EINVAL;
        }
        mem = fifo_alloc(size);
    }
    if (!mem) {
        return -ENOMEM;
    }
    memset(mem, 0, size);
    old = RTF_BASE(minor);
    rtl_spin_lock_irqsave(&RTF_SPIN(minor), interrupt_state);
    RTF_BASE(minor) = mem;
    RTF_BUF(minor) = size;
    RTF_START(minor) = 0;
    RTF_LEN(minor) = 0;
    rtl_spin_unlock_irqrestore(&RTF_SPIN(minor), interrupt_state);
    if (RTF_ALLOCATED(minor) && old && !free_prealloc(old)){
        fifo_free(old);
    }
    return 0;
}

```

코드 1316. rtf_resize() 함수의 정의

rtf_resize() 함수는 먼저 넘겨받은 파라미터 값이 올바른 값을 가지는지를 확인하고, 그렇지 않다면 -EINVAL이나 -ENODEV를 넘겨준다. 만약 넘겨받은 크기 값이 PREALLOC_SIZE(=8192)⁵¹⁰과 동일하다면, mem에 get_prealloc()함수를 호출해서 미리 FIFO를 위해서 할당된 버퍼에서 가져온 메모리의 주소로 초기화 시킨다. 만약 이와같지 않다면, rtl_rt_system_is_idle() 함수를 호출해서 RTLinux 커널이 idle 상태인지를 확인한 후, fifo_alloc()을 호출해서 초기화 한다. RTLinux 커널이 idle인 상황이 아니라면, -EINVAL을 돌려줄 것이다. 여기서 이 함수의 서두 부분에 있는 comment를 보면 real-time task에서는 이 함수를 호출하지 못하도록 하고 있는데, 이는 fifo_alloc() 함수가 vmalloc() 함수와 같은 Linux 커널 API를 사용하기 때문이다. 따라서, real timer kernel module의 초기화 함수나 등록해제 함수와 같은 곳에서만 이 함수를 호출할 수 있다.⁵¹¹

만약 메모리를 할당할 수 없다면 -ENOMEM을 돌려준다. 할당된 메모리는 0으로 초기화 하고(memset()), 원래의 FIFO의 기본 주소를 old로 가져온다. 이전 FIFO의 크기에 변화를 주어야 하기에 spin lock을

⁵¹⁰ 특별히 다른 설정(CONFIG_PREALLOC)이 없다고 한다면, 8192를 기본적인 크기로 가지게 될 것이다.

⁵¹¹ vmalloc() 함수는 Linux 커널의 API로서 연속되지 않은 메모리 공간을 할당할 경우에 사용한다.

호출하는 process가 요구된 메모리 보다 작은 시스템의 메모리 공간이 있을 경우에는 blocking될 가능성이 있다. fifo_free() 역시 vfree()를 사용한다.

설정하고(`rtl_spin_lock_irqsave()`), 해당 메모리를 새롭게 할당된 메모리로 바꾼다. 관련된 변수들도 초기화해 주어야 한다. 이것을 마치면 `spin lock`을 풀고(`rtl_spin_unlock_irqrestore()`), 원래 할당된 메모리를 해제한다. 해제할 때 미리 할당된 메모리 공간이라면, `free_preallo()`을 호출하고, 그렇지 않다면 `fifo_free()`를 호출한다. 복귀 값은 0이 될 것이다.

이것은 RTLinux의 핵심적인 부분에 대한 설명을 어느정도 마쳤다. 더 자세한 것을 원하는 사람이 있다면, 직접 코드를 분석해보기 바란다. 여기서는 이 정도로 만족했다고 생각하기에 이젠 RTLinux에 대한 성능을 분석하도록 하겠다.

20.8. RTLinux의 Performance분석

앞에서 우리는 RTLinux의 기본적인 구조와 어떤 메커니즘으로 Linux를 이용해서 real-time성을 구현하는지를 보았다. 이번 장에서는 RTLinux가 가진 real-time에 대한 성능을 분석하도록 하겠다. Real-time 운영체제의 성능을 분석하는데 필요한 요소 중 다음을 중점으로 해서 RTLinux의 성능을 평가하도록 하겠다.

- Interrupt Latency : CPU가 하드웨어로부터 interrupt 요청을 받은 후, ISR의 첫번째 연산이 시작되기 전까지의 시간을 말한다.
- Context Switching Time(Task Switching Time) : 실행가능하며 동일한 우선순위를 가지고 있는 두개의 독립적인 task간에 실행을 switching하는데 걸리는 평균적인 시간을 말한다.
- Preemption Time : 우선순위가 낮은 task가 우선순위가 높은 task에게 제어(control)를 전달하는데 걸리는 시간을 말한다. 이것은 위에서 설명한 context switching time과는 다른 개념으로 먼저 운영체제가 높은 우선 순위를 가지는 task를 활성화시키는 event를 인지할 수 있어야 하며, 현재 실행중인 task와 요청된 task의 상대적인 우선순위를 판별해야 하며, 마지막으로 context switching을 해야 하기 때문이다.

위와 같은 항목에 대해서 RTLinux는 얼마 만큼의 시간을 요하는지를 아래에서 살펴보도록 하겠다. 또한, 가능하다면, 현재 Linux에서는 얼마 만큼의 시간을 또한 요구하는지 참고적으로 보도록 하겠다.

주의할 점은 이것은 저자의 생각과 저자의 방법을 택해서 한 것이므로, 생각하는 것이나 측정하는 방법이 다르면, 달라질 수 있다는 것이다. 따라서, 절대적인 판단의 근거가 될 수 없으며, 이에 대한 책임은 전적으로 측정자의 몫이라는 것이다. 절대 여기에서 제시하는 자료를 다른 곳에 적용하지 않기를 바란다.

20.8.1. Interrupt Latency

이곳에서 측정하고자 하는 인터럽트 latency는 인터럽트 signal이 들어온 이후에 인터럽트를 처리하는 시점까지 소요되는 시점이다. Hardware적으로 인터럽트가 생성되는 시점은 측정하기 어렵기 때문에, 커널에서 인식하고 난 이후에 인터럽트 핸들러를 호출하는데 까지 걸리는 시간을 보도록 할 것이다.

먼저, RTLinux는 초기화 시에 Linux로 전달되는 인터럽트를 intercept한다. 만약 자신이 원하는 인터럽트인 경우에는 자신이 가지고 있는 인터럽트 핸들러를 호출하고, 그렇지 않은 경우에만 Linux의 인터럽트 핸들러로 forwarding하는 메커니즘을 가지고 있다.

RTLinux는 모든 인터럽트의 처리가 실제적으로 `rtl_intercept()` 함수에서 일어나게 된다. 이것은 RTLinux가 초기화 되면서 원래 Linux에서 interrupt 처리를 담당하는 `do_IRQ()` 함수를 `rtl_intercept()`가 가로채서 처리하기 때문이다. `rtl_intercept()`에서 RTLinux에서 처리할 interrupt인지, 아니면, Linux로 forwarding해야 할 것인가에 따라서, 인터럽트의 처리를 달리한다. 이곳에서 보고자 하는 것은 실제적인 인터럽트 핸들러의 호출 이전까지만을 보도록 하겠다. 즉, `handle()`과 같은 함수를 호출하기 전까지만 본다. Intel 계열의 CPU는 모든 사용 가능한 인터럽트들에 대해서 인터럽트 descriptor를 가지도록 설정하고 있으며, 이 descriptor내에 interrupt가 발생했을 때 control이 전달될 handler의 주소를 가지고 있다. Linux의 경우 이것을 일반적인 interrupt의 경우에 `do_IRQ()` 함수를 가르키도록 하고 있다. 결과는 다음과 같다.

구분	RTLinux	Linux
Interrupt Latency	1110 clock cycles	1130 clock cycles

	(2018 nsec = 2 usec)	(2054 nsec = 2 usec)
--	----------------------	----------------------

표 145. Interrupt Latency의 측정

물론, 이곳에서 제공하는 시간 값이 즉각적인 인터럽트 latency를 정확히 표현한다고 보기는 어렵다. 하지만, 상대적으로 얼마나 긴 시간이 interrupt handler를 호출하는데 걸리는가를 알 수 있는 한 척도로서 보기 바란다.

[표 145]에서 보듯이 RTLinux와 Linux의 interrupt latency time은 별 다른 차이를 보이지 않는다. 미세한 차이를 제외하곤 거의 동일하다고 볼 수 있다. 이것은 RTLinux와 Linux가 동일한 메커니즘을 사용해서 interrupt를 처리하기 때문에 생기는 것으로 생각할 수 있다. 여기서 한가지 간과한 것은 RTLinux의 경우 Linux용의 interrupt가 들어오면, 위의 값에 덧붙여서 Linux interrupt handler인 do_IRQ()를 호출하는 시간이 더 필요하다는 점이다. 따라서, RTLinux 상에서 일반 Linux를 사용하는 overhead가 빠졌다는 점에 주의하기 바란다. 그 외는 거의 동일한 결과값을 보여주고 있다.

20.8.2. Context Switching Time

Context switching time을 측정하기 위해서 x86의 CPU architecture에 있는 Time Stamp Counter(TSC)를 사용하기로 하겠다. 현재 RTLinux에서 제공하는 rtlinuxpro tool은 x86이외에 여러 가지의 architecture 버전이 있지만, x86에서 측정하는 것이 가장 쉬워서 그러한 환경을 사용했다. 먼저 RTLinux에서는 RT-task간의 context switch를 관장하는 routine으로 rtl_fast_switch()가 있다. 일반적인 Linux에서는 swich_to()라는 함수를 이용한다.

TSC는 CPU에 input으로 들어가는 clock tick을 counting하는 register로서 “rdtsc”와 같은 특별한 명령어를 사용해서 얻어올 수 있는 값이다. 이때 출력으로 받을 수 있는 값은 64bit의 값으로 커널에서는 floating point 연산을 지원하지 않으므로, 하위 32bit만을 사용해서 걸리는 시간을 측정할 수 있을 것이다. 테스트 하려는 machine의 spec. 아래와 같다.

- CPU : Pentium III 550MHz
- RAM : 128 MBytes

예를 들어, t clock cycle의 context switching에 소요된 시간이라면, 다음과 같은 식으로 시간으로 변환 가능하다.

$$\left(\frac{t}{550} \times 10^9 \right) \div 10^6 = \frac{t}{550} \times 10^3 (\text{nsec})$$

따라서, 550이라는 CPU cycle을 사용했다면, 1000 nanosecond(or 1 microsecond)가 context switching시에 걸린 시간이 될 것이다.

일반적으로 RTLinux의 경우에는 context switching이라고 해서, 사용하고 있는 address space가 바뀌거나, Intel architecture에 특수한 TSS descriptor와 같은 것에 변화가 있는 것은 아니다. 즉, RT-task가 kernel level에서 동작하는 thread로서 생성되기 때문에, 이러한 일이 필요 없다. 하지만, 일반적인 Linux의 경우에는 이를 위해서 준비 단계들이 필요하기 때문에, 더 많은 시간을 소비할 것이다. 실험은 1000번의 시행을 통해서 얻은 값을 평균한 결과이다. 아래와 같다.

구분	RTLinux	Linux
Context Switching Time	125 nsec	대략 250~350 정도의 clock cycles (≈445 ~ 636 nsec)

표 146. Context Switching Time 측정⁵¹²

⁵¹² RTLinux의 경우에는 gethrtime()이라는 함수에서 nano second 단위의 시간 측정을 지원하고 있지만, Linux의 경우에는 “rdtsc”와 같은 architecture dependent한 assembly 명령을 통해서 가능하다. 이렇게 구한 값을 CPU speed에 맞게 새로이 계산해 주어야 한다.

결과에서 보듯이 RTLinux가 거의 2배 이상 더 빠르게 context switching을 수행하고 있음을 알 수 있다. 하지만, 이것은 어디까지나 context switching 시의 overhead가 Linux보다 적다는 것에 기인하고 있다는 것이다. 즉, Linux는 하나의 process가 가지는 context가 heavy한 반면에, RTLinux는 커널 내에서 공통된 주소 공간을 사용한다는 이점이 있기 때문에 light하다는 것이다. Linux의 경우 context switching time에 있어서 편차가 많이 나는 이유는 code를 optimization하기 위한 노력을 많이 가했기 때문에, 여러 가지의 조건들이 추가되었기 때문이다. 예를 들어서, 자신이 사용하는 address space를 가져오기 위한 연산에 있어서, TLB를 flush한다거나, 혹은 이전에 쓰던 address space를 공유하는 경우에는 이를 하지 않게 되는 등의 판단을 요하는 연산이 추가적으로 더 있기 때문이다.

어쨌든 결과적으로 놓고 보았을 때, RTLinux를 사용해서 어떤 특정 event에 반응하게 하는데 필요한 overhead가 적다는 것에는 동의할 수 있을 것이다.

20.8.3. Preemption Time

Linux의 경우에는 task preemption이 일어나는데 까지 걸리는 시간을 측정하는 것이 사실상 어렵다. 즉, 커널 내에서 process가 진행중인 경우에는 task preemption이 일어나지 않으며, kernel 모드에서 user 모드로의 전환에서 현재 진행중인 task의 “need_resched”이라는 필드를 보고 schedule() 함수를 호출해서 새로운 task로의 context switch가 일어나게 된다. 또한 우선 순위가 높은 task가 READY상태가 된다고 해서, 이것이 무시되는 것은 아니다.

Linux에서 SCHED_OTHER에 해당하는 task가 최대로 실행될 수 있는 시간은 대략 200msec 정도이다. 이 정도의 시간이 지나고 나면, 새로운 task로의 context switching이 자동으로 일어나게 된다. 따라서, CPU oriented된 작업이 system call을 사용하지 않고, 수행될 수 있는 시간은 최대 200msec로 가정할 수 있을 것이다. 또한, 이 경우 거의 동시에 우선 순위가 더 높은 task가 READY가 된다고 하더라도, preemption이 일어나지 않으므로, 최악의 경우 대략 200msec 이하의 시간을 필요로 한다고 볼 수 있을 것이다.

RTLinux의 경우에는 이와는 조금 다르다. 즉, task가 커널 내부에 존재하며, 모든 interrupt와 system call은 RTLinux에 의해서 intercept된다. 따라서, 이 경우 RTLinux는 새로운 task를 scheduling할 수 있는 기회를 얻게 되며, 이때 우선 순위가 더 높은 RT-task에 CPU를 차지할 수 있는 기회를 줄 수 있다. 또한 RTLinux의 각 RT-task들은 주기성을 가진다고 본다. 즉, RT-task가 자신만을 완전히 수행하는 경우에는 다른 RT-task는 수행될 기회를 가지지 못한다. 다른 말로 바꿔서 말하면, RT-task가 완전한 preemption을 제공하지는 못한다는 것이다. 이것을 해결하기 위한 방법의 일환으로 nanosleep()과 같은 RTLinux API를 가지고 있으며, RT-task가 waiting 상태로 있을 경우에만 다른 RT-task⁵¹³를 scheduling할 수 있다. 이것은 RTLinux가 Linux의 커널 address space에서 수행되며, 일반적인 task와는 그 구조가 다르기 때문에 생기는 문제라고 할 수 있다.

예를 들어 어떤 특정한 RT-task가 무한 loop를 도는 경우를 보도록 하자. 이때, 이 무한 loop에서 이 RT-task에게서 control을 뺏을 수 있는 기회는 timer interrupt와 같은 외부의 interrupt를 이용하는 방법이 있을 것이다. 하지만, 현재 RTLinux에서는 이런 식의 scheduling mechanism을 가지고 있지 못한 것으로 보인다.⁵¹⁴ 물론 이것은 RTLinux에서 가정하는 일반적인 RT-task의 가정을 위반한 것이라지만, 커널이 모든 제어를 가질 수 있어야 한다는 점을 증명하기 위한 한 방법이었다.

따라서, 이러한 RTLinux에서 상에서 preemption time을 측정한다는 것은 무의미 하므로, 측정 시간과 같은 데이터를 보여주지는 못한다. 하지만, 이곳에서 이야기 할 수 있는 것은 RTLinux는 현재 진행중인 RT-task의 maximum control path의 시간 이하에서 우선 순위를 고려한 scheduling을 할 수 있다는 것이다. 참고적으로 말하자면, RTLinux가 가지고 있는 scheduling policy에는 FIFO(First-In-First-Out)와 RR(Round-Robin), OTHER(일반 RT-task scheduling policy)가 있다는 것이다. 하지만, code내에서는 아직 이러한 scheduling policy들은 반영되지 않는 것 같으며, 단지 priority에 기초한 scheduling만을 고려하고 있다.

⁵¹³ 여기에는 Linux kernel이 포함된다. 즉, 다른 RT-task가 수행되지 않는 경우에 Linux가 수행될 기회를 얻을 수 있다.

⁵¹⁴ 실험을 했을 때, 커널이 멈춰버렸다. 즉, 더 이상의 interrupt를 받지 못하고 있으며, RT-task의 제어를 회수할 수 없었다.

RT-task가 가질 수 있는 maximum control path는 하나의 function 정도라고 생각하면 될 것이다. 물론 더 복잡해 질 수 있지만, 주기적으로 수행되는 하나의 function이 될 것이다. 이곳에서 많은 시간을 소비하게 될 수록 전체적인 시스템의 performance 및 RT-event에 대한 반응 시간이 낮아질 것이다.

20.9. RTLinux의 응용

여기서는 RTLinux를 어떤 곳에서 사용할 수 있는가를 보도록 하겠다. 물론 여러 가지 분야에서 적용 가능 하지만, 대표적인 경우를 보도록 할 것이다. [그림 141]은 RTLinux를 적용한 전형적인 예를 보여준다. 그림에서는 real-time task로서 카메라의 image를 capture⁵¹⁵하는 것이 있으며, 이를 받아서 DB(Data Base)에 보관하거나 원격지의 제어 센터에 보내고 화면에 표시 시켜주는 Linux 프로세스가 하나 존재한다. 이러한 두개의 task(혹은 process)의 연결 창구 역할을 하는 것이 바로 RT-FIFO이며, 화면에 display하기 위해서는 X-Window system을 사용하고, 원격지로 보내기 위해서 internet을 이용한다.

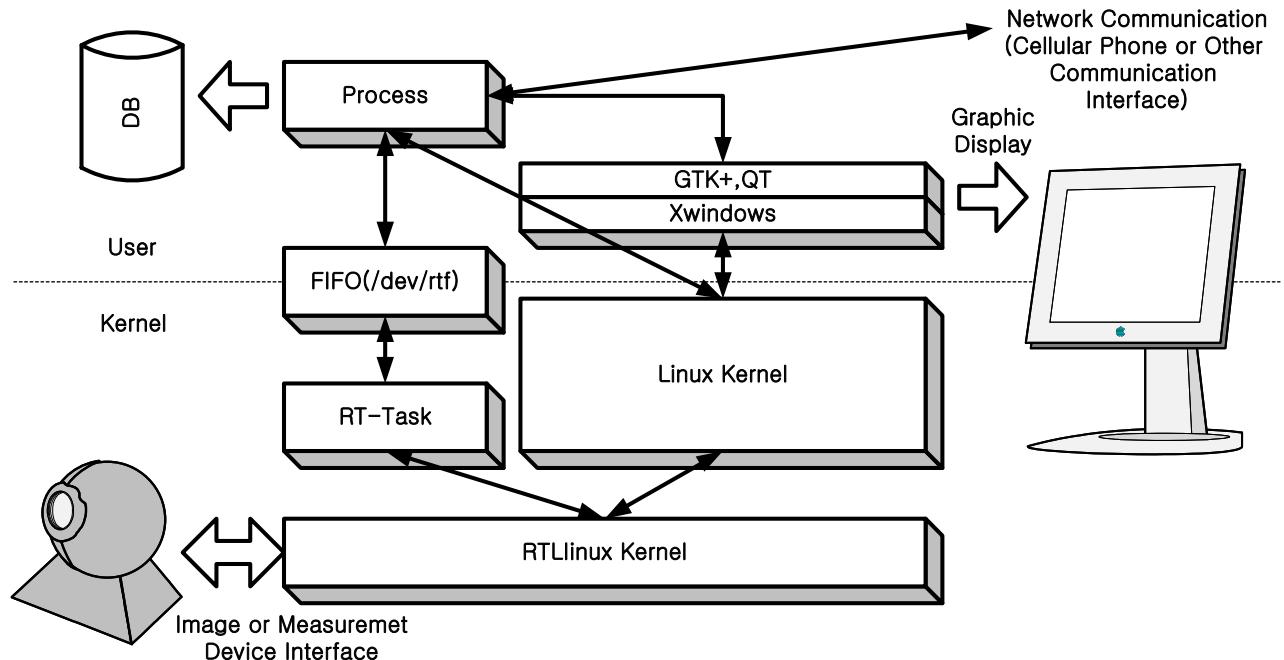


그림 141. RTLinux의 적용 사례

그림에서 알 수 있듯이 image를 capturing하는 것은 실시간 성을 요구하기에 RT-task를 적용하는 것이 좋으며, 최대한 빠른 동작을 반복적으로 수행하기 위해서, 측정된 데이터의 처리 및 가공은 Linux 프로세스로 넘겨지게 된다. FIFO는 이를 위해서 반드시 필요한 인터페이스가 될 것이다. 즉, 프로세스는 FIFO를 일반 파일과 같이 open해서 계속적으로 FIFO의 데이터를 읽어나가게 되고, RT-task는 FIFO에 데이터를 계속 쌓아 나가기만 하면 될 것이다.

20.10. 결론(Conclusion)

이전의 장들에서는 RTLinux가 가지는 특징들을 살펴보았다. RTLinux에 대해서 완전히 다 보지는 못했지만, 중요한 기능들의 구현에 대해서 어느 정도는 이해가 되었으리라고 생각한다. 이번 장에서는 위에서 보았던 것을 간략히 정리하고, 어떤 application field에 RTLinux를 적용하는 것이 좋은가와 RTLinux의 장점 및 단점에 대해서 보도록 하겠다. 다시 한번 말하지만, 여기서 저자가 말하는 바는 주관적인 것으로 보는 관점에 따라서 달라질 수 있다는 것을 유념하기 바란다.

⁵¹⁵ Image의 capture와 더불어서 각종 수치 측정이 이를 대신할 수 있을 것이다. 가령 예를 들어서 댐의 수위를 측정한다던가, 날씨를 측정하는 것도 한 응용분야가 될 수 있을 것이다.

RTLinux가 가지는 장점은 앞에서 이미 보았듯이 real time event에 대해서 더 빠르게 반응해 줄 수 있다는 점이다. 이것은 기본적으로 RTLinux가 가지는 특성에 기인하고 있으며, 이를 Linux와 결합시켜서 Linux가 가진 기능을 확장하고 보완했다는 것이다. 또한 일관된 interface를 사용해서 programmer에게 효과적인 개발 방법을 제시하고 있는데, 그 중의 하나가 바로 pthread API interface를 제공하고 있다는 것이다. 물론 전체 pthread API interface가 제공되지는 않지만, 부분적으로 나마, 그러한 interface를 가지고 있으므로 해서 다른 시스템으로의 porting이 용이하다는 것이다. 이것 역시 RTLinux의 function이지, POSIX thread와 같은 library가 아니다. 요약하면 아래와 같이 볼 수 있다.

- 빠른 응답성을 보장한다(RT-task의 경우).
- Pthread API interface(부분적)를 통한 RT-task의 구현.
- Linux/FreeBSD application을 그대로 사용해서 real-time성을 만족시켜주고 있다(Linux와 FreeBSD 커널의 수정을 최소화 함.).
- Multiprocessor를 지원한다.

RTLinux가 가지는 단점은 모든 task가 kernel level에서 동작한다는 것이다. 즉, Linux커널 이외에도 kernel level에서 동작해야 되는 것들이 많아 짐으로 해서, 커널에 상주해야 하는 프로그램 코드의 양이 많아지게 된다. 또한, 사용자 프로세스와의 interface를 RT-FIFO이외에는 별다른 interface를 제공해 주지 않는다는 것도 단점이 될 수 있다. 커널 내에서 동작하는 RT-task는 일반적인 사용자 프로세스와는 하는 일이 한정되며, 될 수 있는 한 빠르게 처리될 필요가 있는 routine들만을 가지도록 하는 제약이 있으며, 만약 여기에 overload가 걸리면 전체적인 performance가 낮아질 수 있다. 기존에 동작하던 프로그램을 RTLinux상에서 수행하기 위해서는 프로그램의 전체적인 re-structuring이 필요하며, 커널에서 동작하는 routine도 Linux 커널 API를 사용하기 보다는 RTLinux에서 제공해 주는 API를 통해야만 안정성을 보장 받을 수 있다. 커널 내에서 동작한다는 점에서 또한 debugging이 어렵다는 점이 문제가 될 수 있으며, 조금만 프로그래머가 실수를 해도 시스템 전체가 down될 가능성이 많다. 오로지 “printf()”⁵¹⁶와 같은 값을 통해서만 어떤 결과를 알 수 있다. 또한 RTLinux의 각 task는 주기성을 가진다는 점에서 일반적으로 생각하는 real-time system에서 수행할 수 있는 task와는 조금 다르다. 즉, 주기성을 가진 task에 대한 지원을 강조하는 반면, 그렇지 못한 task들이 RT-task로 생성되었을 경우에는 시스템의 전체적인 performance에 악영향을 미칠 수 있다. 만약 이러한 task를 강제 종료하려고 할 때도, 제어를 kernel이 받을 수 없기 때문에 강제 종료가 불가능하게 된다. 요약하면 아래와 같다.

- RT-task가 코드양이 많으면 커널에 상주해야 하는 코드가 커지며, RT-task로 인한 시스템의 load가 커진다.
- 기존의 software를 re-structuring해 주어야 한다(RT-task와 normal Linux process로 구분).
- Debugging이 곤란하다.
- RT-task의 오류가 전체 시스템의 오류로 이어진다.
- RT-task와 normal process의 interface가 부족하다(FIFO를 통한 interface만을 제공).
- RTLinux 커널만 따로 사용할 수 없다.
- 완벽한 RTLinux Kernel내에서의 preemption 미비.

결론적으로 RTLinux에 대한 평가를 한다면, 일단 복잡하고 많은 것을 수행할 필요가 있는 project에는 적합한 방법이 아니며⁵¹⁷, 간단한 real-time성을 요하는 project에는 적용할 수 있는 충분한 가능성이 있다는 것이다. 이것은 신규 project를 고려할 때, 설계 초기단계에서부터 requirement를 분석해서, 이를 RT-task와 그럴지 않은 task로 분리하는 과정을 필요로 하며, 지나치게 많은 부분을 RT-task에서 수행하려고 해서는 안 된다는 것을 강조해 주어야 할 것이다.

RTLinux는 Linux를 적용한 한가지 분야를 새로이 개척했다는 점에서는 분명히 그 의미가 깊다. 또한, 기존의 Unix-like한 시스템에서 해결하지 못했던 real-time 특성을 dual-kernel이라는 구조를 사용해서 해결했다는 것에 대해서는 상당히 독창적이라고 할 수 있겠다. 현재는 RTLinux가 Linux이외에

⁵¹⁶ 실제로 printf와 같은 rtl_printf()를 가지고 있으며, rtl_printf()는 또한

⁵¹⁷ 물론 이에 대해서 의견이 있을 수 있겠으나, 커널 내에서 다양한 일을 해야 한다면, 좋지 못한 영향을 줄 수 있기 때문이다.

FreeBSD에서도 사용할 수 있다는 이야기를 들었는데, 점차 그러한 free software가 영역을 확장하고 있다는 점에서도 반길 일이다. 하지만, 아직 대규모의 project에 사용된 예가 적으며, dual-kernel 구조 자체가 특허(patent)에 둑여 있으며, 완전한 RTLinux를 사용하기 위해서는 대가를 지급해야 한다는 점에서 신경을 써야 할 것이다.

21. 앞으로 남은 일

앞으로 해야 할 일은 아주 많이 남았다. 아직 모르는 부분들이 산적해 있으며, 제대로 설명을 하지 못한 부분들도 많다. 특히 하드웨어와 관련된 것은 이해하기 어려운 부분들이 많으며, LILO를 이용한 부팅 부분도 아직 다 정리하지 못했다. 메모리와 관련된 것은 특히 많이 남았다. Buddy 알고리즘과 Slab 및 커널의 초기화에서 일어나는 메모리 설정을 완전히 뜯어서 보지는 못했다.

프로세스와 관련된 것으로는 프로세스의 context를 정의하는 부분과 task switch시에 일어나는 일련의 활동 및 초기 메모리를 그리지 못했으며, 파일 시스템에 관련된 부분에서는 일관성이 없는 설명으로 조금 머리를 복잡하게 만들었을지도 모른다. 네트워크는 특히 설명되지 않고 넘어가는 부분들이 많은데, 주소 프로토콜의 옵션과 헤더 부분이 될 것이다. 이를 다 분석하는 것도 힘들고, 내가 아는 지식의 범위가 아직 짧은 탓이다. 하지만 기본적인 이해는 어느 정도 가능할 것이다.
각종 디바이스 드라이버들은 아직 chip에 의존적인 것들이 많은 관계로 하드웨어와 관련되지 않은 디바이스 드라이버를 위주로 설명했다. 그리고, IEEE 1394와 같은 새로운 디바이스에 대한 것이 미흡하다. USB도 하위 layer 있는 드라이버들은 설명하지 않았으며, PCMCIA같은 것도 역시 하위 부분이 보이지 않는다. 다 내 탓이라고 들려야 할 부분들이다.

디버깅 방법은 극히 기본적인 것만 이야기 할 뿐이며, 버스들에 대한 설명도 단편적인 것에 그치고 있다. 즉, 하드웨어가 초기화 될 때, 커널에서 생성하는 데이터 구조에 대해서는 별로 많은 것을 찾아볼 수 없을 것이다. ELF 파일 포맷이외의 binary 파일 포맷도 없다. ELF를 다루고는 있지만 완전한 설명은 지면 관계상(그리고, 저자의 실력상) 허락하지 않기에 필요한 부분만 주섬주섬 주워왔다.

이렇게 보니, 완전히 잘못된 문서를 작성한 느낌마저 드는데, 하지만 저자가 이와 같은 노력을 기울이면서 머리 속에서는 항상 어떤 그림을 떠올렸으며, 그 그림을 끝없이 맞춰가려고 노력했다. 마치 조각으로 나누어진 큰 그림을 어떤 틀 속에 하나하나 윤곽만으로 맞춰나가듯이 정리했다. 최선이라고는 볼 수 없지만, 적어도 타이핑에 들어간 손의 노력만은 잊지 않기를 부탁 드리고 싶다. 만약 다음 판이 나온다면, 아마 앞에서 이야기한 부분들을 정리할 것이며, 운영체제 자체에 대한 이론적인 이야기를 더 추가할까 한다. 여러 가지의 이론들이 실제로 Linux라는 운영체제에 어떻게 적용되었는지를 알기 위한 것이다.

남은 것이 있기에, 더 할 수 있다는 아쉬움과 기회만 주어지면 그것을 완성하겠다는 마음의 자세도 생긴다. 그럼, 이것으로 간략히 나마 앞으로 남은 일에 대한 정리를 마친다.

22. Appendix

22.1. Linux Network Device Driver 개발 예

여기서 부터는 앞에서 보였던 Network 디바이스에 대해서, 필자가 실제로 개발을 담당했던 삼성전자의 Ethernet 100Mbps/10Mbps card인 ks8920 chip을 이용한 network device driver의 실제적인 source code를 보기로 하겠다. 이 부분의 문서는 이미 web에 있기에 혹시 필요한 분들을 위해서 전체 내용을 다시 실었을 뿐이다. 하지만, 주의해야 할것은 커널 버전 2.0.X 상에서 개발되었기에 현재의 커널에서 컴파일 하고자 한다면, 수정해 줄 부분들이 있다. 가령 예를 들어서 socket buffer에 대한 할당이나 해제, PCI에 관련된 함수들, 그리고 모듈의 초기화등이 있을 것이다.

가장 먼저 디바이스 드라이버를 작성하는 프로그램자는 source code에 대한 분석에 앞서서 그 device에서 사용하고 있는 chip에 대해서 먼저 알아야 한다. 주로 network카드를 구입하게 될 때 manual의 형태로 제공되는데, 이곳에는 device에서 사용하는 register의 set들에 대한 자세한 정보를 담고 있다. 이하에서 설명하는 것은 그에 대한 이해가 있어야만 완전히 이해가 가능하지만 큰 줄기에 대해서 짚어나가는 식으로 설명하도록 한다. 시작하기 전에 DMA를 사용하기 위해서 사용하는 data의 구조와 이것들 간의 관계를 잠시 보도록 하자.

```

typedef struct {
    u32 data;
    u16 length;
    u8 RxBDID;
    union {
        u8 bdctl;
        struct {
            u8 RxBDSeqN : 7;
            u8 COwnsFD : 1;
        }s;
    }u;
} BD;

/* frame descriptor structure*/
typedef struct {                                     /* Transmit frame descriptor set. */
    u32 next;           /* FDNext */
//    u32 system;
    struct sk_buff* tx_skbuff; // system field
    u16 stat;           /* FDStat */
    u16 reserved;
    u16 length;
    union {
        u16 fdctl;
        struct {
            u16 BDCount : 5;
            u16 Reserved : 5;
            u16 FrmOpt : 5;
            u16 COwnsFD : 1;
        }s;
    }u;
    BD freebufflist[TX_FD_BD_NUMBER]; // 2
} TxFD;

/* The Rx buffer list frame descriptors.*/
typedef struct {                                     /* Receive frame descriptor. */
    u32 next;           /* FDNext */

```

```

u32 system;
u16 stat;           /* FDStat */
u16 reserved;
u16 length;
union {
    u16 fdctl;
    struct {
        u16 BDCount : 5;
        u16 Reserved : 5;
        u16 FrmOpt : 5;
        u16 COwnsFD : 1;
    }s;
}u;
BD freebufflist[RX_RING_SIZE];
} RxBL;

/* The Rx FDA descriptors */
typedef struct {
    u32 next;          /* FDNext */
    u32 system;
    u16 stat;           /* FDStat */
    u16 reserved;
    u16 length;
    union {
        u16 fdctl;
        struct {
            u16 BDCount : 5;
            u16 Reserved : 5;
            u16 FrmOpt : 5;
            u16 COwnsFD : 1;
        }s;
    }u;
    BD freebufflist[RX_FDA_BD_NUMBER]; // 2
} RxFDA;

```

코드 1317. 디바이스 드라이버에서 정의하고 있는 buffer에 대한 설명

위의 코드에서는 DMA 입출력에 사용할 것으로, Tx용과 Rx용으로 사용할 buffer descriptor들에 대한 것과 현재 free한 descriptor들의 list에 대한 structure를 정의 하고 있다. DMA에서는 이렇게 미리 정해진 descriptor들의 구조에 대한 pointer를 DMA controller에 알려주어, 각각의 field에 대한 처리를 맡기는 것이다. 나중에 device driver에서는 ownership에 대한 COwnsFD를 검사하여 data가 device driver에 유효한가를 알게 되며, 그것을 이용해서 buffer descriptor들을 처리하게 된다. 이것은 program source를 살펴보면서 보도록 하자.

이것을 그림으로 보면 크게 [그림86]와 같은 구조를 가지게 된다. 이 구조에서는 Frame Descriptor에 대해서 하나의 Buffer Descriptor를 두고 있으나, 여러 개를 두어도 상관이 없다. 따라서, Buffer Descriptor에 Data의 read나 혹은 write가 있을 경우, 이에 대한 Status가 자동적으로 변동을 일으키게 되고, 이것을 지켜봄으로 해서 DMA가 진행되었는지를 확인 할 수 있게 되는 것이다.

이러한 데이터 구조를 가지는 것으로는 Transmit Queue와 Buffer List Initialization, Free Descriptor Area Initialization이 있다. 이 데이터 구조를 사용하는 방법에 대해서 알아보도록 하자.

22.1.1. Data Structure의 사용법

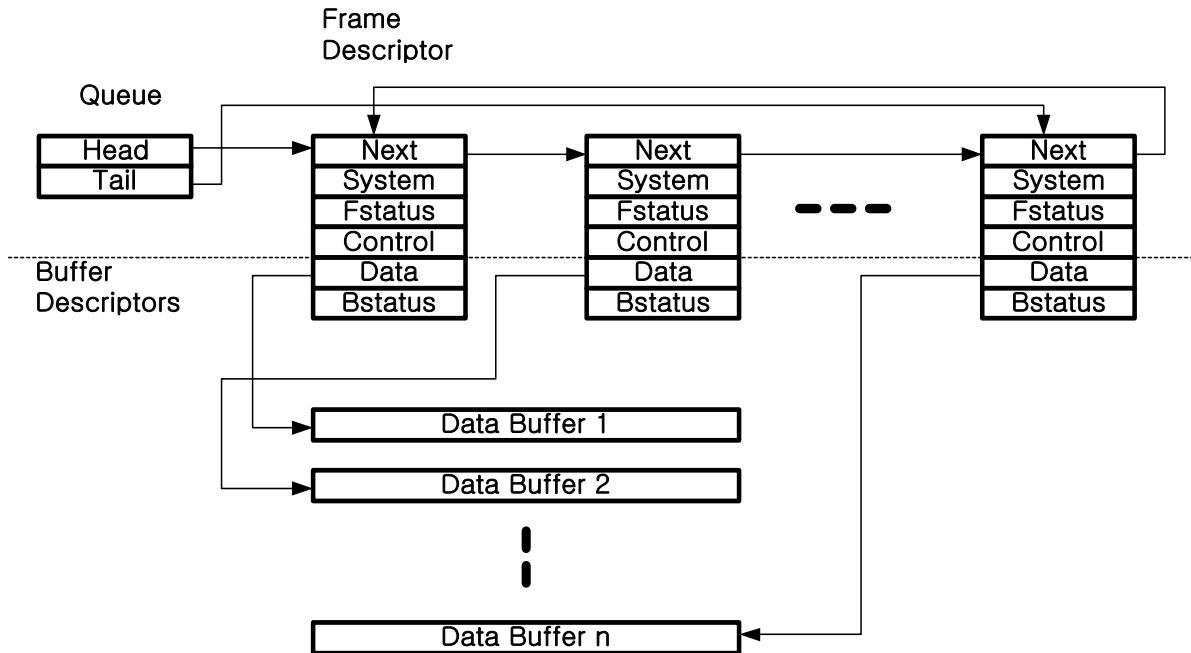


그림 142. Frame 및 Buffer Descriptor에 대한 이해

22.1.2. Transmit Queue Initialization

먼저 Transmit Queue는 controller가 보내고자 하는 frame을 찾는 곳이다. 다음과 같은 두 가지 모드로 동작 할 수 있다.

- Batch Processing Mode – 이 모드에서는 FD(frame descriptor)를 linked list로서 구성하고, FD의 ownership bit를 controller로 set한다. 마지막에 있는 FD는 EOL(End of link) bit를 두어서 마지막임을 나타내주게 된다. 전송을 시작하기 위해서는 첫번째 FD의 주소를 Transmit Frame Pointer Register에 써 준다.
- Continuous Polling Mode – 이 모드에서는 FD의 linked list를 만든 후, ownership bit를 controller로 set한다. 마지막 FD는 dummy FD로 ownership bit를 system own으로 설정하게 된다. 즉, 보내고자 하는 데이터의 마지막을 나타내게 된다. 역시 전송을 시작하기 위해서는 첫번째의 FD의 주소를 Transmit Frame Pointer Register에 써주게 된다. 모든 valid한 packet의 전송이 일어난 후에는 controller는 polling mode⁵¹⁸로 가게 되며, System이 새로운 packet을 보낼 준비가 되면, dummy FD는 새로운 linked list가 되며, 다시 ownership bit가 controller가 된다. System은 controller가 polling하기를 기다리게 되거나, 혹은 DMA Control Register의 TxWakeUp bit를 set해서 polling할 수 있다.

여기서 설명한 방법에 대해서는 실제의 code를 보면서 좀더 자세히 살펴보도록 하겠다.

다음과 같이 Frame Descriptor의 각 field를 사용하도록 한다.

- FDNext : 다음 번의 frame descriptor의 주소. Batch processing을 위해서는 queue의 마지막 frame descriptor는 EOL indicator를 가져야 한다.
- FDSys : System field.
- FDStat : Don't care.
- CownsFD : 1로 되어있으면 Controller가 ownership을 소유한다.(dummy frame descriptor에서는 polling을 하기 위해서 0으로 setting된다.)
- FrmOpt : Packet 단위의 control을 지원한다.

⁵¹⁸ 주기적으로 확인을 하게 되는 것을 polling이라고 한다.

- BDCount : Frame Descriptor에 있는 Buffer Descriptor의 수를 나타낸다.

Buffer Descriptor의 field들은 다음과 같이 사용한다.

- BuffDtPt : Data를 저장하기 위한 storage의 32 bit address.
- BuffLength : Buffer 속에 있는 전송하고자 하는 data의 수.

물론 제대로 이야기 하자면 manual을 자세히 읽어보는 것과 실제 프로그램에서 어떻게 하는지를 보는 것이 중요하다. 반드시 network card를 살 때, manual을 청기도록 하자. 나중에라도 이것을 사용하게 될 기회가 있을지 모른다. 다음으로는 buffer list를 보도록 하겠다.

22.1.3. Buffer List Initialization

BL(Buffer List)는 controller가 받은 data를 저장하고자 할 때 보는 곳이다. 이것은 frame descriptor의 list이며, free descriptor의 list를 가지고 있다. 다음과 같이 구성될 수 있겠다.

1. 하나의 FD가 Free BD들을 가지게 되는 경우이다. FD의 FDNext field는 EOL indicator를 가진다.
2. FD의 linked list로 되어있는 경우로 마지막 FD는 EOL indicator를 가지게 된다.
3. 하나의 FD가 FDNext Field가 자기 자신을 가리키게 된다. 이것은 circular queue를 만드는 한가지 방법이다.
4. FD의 linked list를 만들어서 마지막 FD의 FDNext field가 첫번째의 FD를 가리기도록 하는 방법이다. 물론 circular queue를 만들게 되는 것이다.
- 5.

FD 다음과 같이 초기화 되어진다.

- FDNext : 다음번의 FD의 주소를 가진다. 이것이 마지막 FD라면, EOL indicator를 가지던가, 혹은 circular queue의 첫번째 FD를 가리기도록 만들어준다.
- FDSystem : System field.
- FDStat : Don't care.
- COwnsFD : Packet 단위의 control.
- FrmOpt : Packet 단위의 control.
- FDLength : FD에 있는 free BD의 개수.

BD는 다음과 같이 초기화 된다.

- BuffDtPt : Data를 저장하기 위한 공간의 32bit address.
- COwnsBD : Controller가 소유하도록 1로 set한다.
- RxBDSeqN : Don't care.
- RxBDID : Buffer ID 값이다. 고유한 값으로 256가지가 있을 수 있다.
- BuffLength : Buffer의 크기이다.

BL과 관련된 Register로는 Buffer List Frame Pointer Register가 있다.

22.1.4. Free Descriptor Area Initialization.

Free Descriptor Area(FDA)는 system memory에 예약된 공간으로 controller에 의해서 받은 packet에 대해서 receive queue를 generate하는데 사용된다. FDA는 반드시 연속적이어야 하며, 16 byte block들로 되어있다. 각각의 block들에 대해서, 마지막 Byte의 MSB(Most Significant bit)는 “1”로 set되며, 이것은 Controller ownership를 나타낸다. 이것과 관련된 Register로는 Free Descriptor Area Base Register와 Free Descriptor Area Limit Register가 있다.

이상에서 간단히 buffer들을 위한 data structure에 대해서 알아보았다. 이와 같은 data structure들을 어떻게 사용하는지에 대해서 살펴 보기로 한다.

22.1.5. Device private structure

Device의 private structure는 kernel에 일부분으로 kernel에 등록되어 device driver에서 전역변수를 유지하고자 할 때 사용한다. 이것은 반드시 정의되어 있어야 하며, 이것은 device driver의 routine을 호출하게 될 때, parameter값으로 넘겨지게 되며, Device driver routine은 이것을 이용해서 작업을 처리하게 된다. Kernel에는 device structure와 함께 등록되며, kernel interface routine내에서 device structure를 통해서 접근된다.

```
struct ks8920_private {
    char devname[8];                                /* Used only for kernel debugging. */
    const char *product_name;
    struct device *next_module;
    int dummy;
    /* Tx descriptor ring */
    TxFD tx_ring[TX_RING_SIZE];
    /* Rx descriptor ring & addresses of receive-in-place skbuffs. */
    RxFDA rx_ring[RX_RING_SIZE];
    RxBL rx_bufflist;
    struct sk_buff* rx_skbuff[RX_RING_SIZE];
    struct enet_statistics stats;
    int in_interrupt;
    unsigned int enq_rx, enq_tx;                      /* The next free ring entry */
    unsigned int deque_rx, deque_tx;                   /* The ring entries to be free(ed. */
    unsigned int tx_full:1;                           /* The Tx queue is full. */
    unsigned int link_speed:2;                        /* link speed : 0 : 10M, 1:100M , 2:auto_sense */
    unsigned int full_duplex:1;                       /* Full-duplex operation requested. */
    unsigned int default_port:1;                      /* Last dev->if_port value. */
};
```

코드 1318. 네트워크 디바이스 구조체의 private 필드를 위한 정의

ks8920_private 구조체는 우리가 작성하는 device driver의 device structure를 정의해 놓은 것이다. Device의 이름과 product의 이름, 다음 module에 대한 interface, 그리고, dummy field를 넣어서 32bit alignment를 하도록 넣었으며, 이것은 device에서 memory의 physical address를 access할 때, 32bit 단위로 access하기 때문에 넣은 것이다. 다음 field로는 DMA에 사용되는 receive buffer와 transmission buffer가 온다.

Network device driver의 statistics를 유지하기 위한 공간과, interrupt의 발생을 enable과 disable시켜줄 field, 현재 queue의 위치를 나타내는 pointer들이 나온다. 또한 transmission queue의 상태(full, or empty)를 나타내는 field, 현재 link가 100Mbps/10Mbps를 타내는 field와 full duplex⁵¹⁹를 나타낸 field, default port가 온다. 이러한 field값들은 driver의 초기부분에서 결정되면 이 나중에 쓰기 위해서 보관되어 진다.

22.1.6. Module적재

init_module()함수는 module의 적재에 대해서 보여주고 있다. Kernel은 insmod를 실행시키면 driver의 이 부분을 호출한다. Source를 보면 먼저 device structure에 대한 pointer를 null, driver의 초기화를 한다. Device의 초기화에 대한 결과를 kernel로 돌려준다. Kernel은 이 결과를 가지고 insmod의 결과를 도출하게 된다.

```
int init_module(void)
{
    int found;
    root_ks8920_dev = NULL;
    found = ks8920_init(NULL);
    return found ? 0 : -ENODEV;
}
```

⁵¹⁹ Half duplex, full duplex가 있다. 한쪽에서 보낼 때 다른 쪽에서는 보내지 못하면, half duplex, full duplex는 양방향으로 동시에 보내고 받을 수 있을 경우를 나타낸다.

코드 1319. init_module()함수의 정의**22.1.7. Module 해제**

Kernel은 rmmod를 수행하여, 필요 없는 module을 kernel로부터 분리한다. 이곳에서는 initialization routine에서 만든 device driver를 kernel로부터 분리하고, 메모리를 kernel로 돌려준다. 또한 device에서 사용하던 register들을 위해서 예약된 주소공간을 해제한다.

```
void cleanup_module(void)
{
    struct device *next_dev;
#ifdef KDEBUG
    printk(KERN_INFO "samsung ethernet : cleanup_module()\n");
#endif
    /* No need to check MOD_IN_USE, as sys_delete_module() checks. */
    while (root_ks8920_dev) {
        next_dev = ((struct ks8920_private *)root_ks8920_dev->priv)->next_module;
        unregister_netdev(root_ks8920_dev);
        release_region(root_ks8920_dev->base_addr, KS8920_TOTAL_SIZE);
        kfree(root_ks8920_dev);
        root_ks8920_dev = next_dev;
    }
}
```

코드 1320. cleanup_module()함수

cleanup_module()함수를 보면 unregister_netdev()는 kernel에 등록된 network device를 등록해제하며, release_region()에서는 현재 device driver에서 쓰는 register들의 address공간을 해제한다. 이것은 전체 우리가 사용하고 있는 device driver module들에 대해서 반복하게 된다.

22.1.8. Device의 초기화

Device의 초기화는 device가 PCI device에 속한다는 것을 알고 있으므로 그것을 검사하여 device driver가 control하고 싶어하는 device를 찾아서 그에 해당하는 device structure를 만들어 주는 것이다. 또한 내부에 유지할 변수들을 초기화 하는 작업을 하게 된다.

```
int ks8920_init(struct device *dev)
{
#ifdef CONFIG_PCI
    int adapters_found = 0;

    if( pcibios_present() ) {
        static int pci_index = 0;

        for ( ; pci_index < 8; pci_index++ ) {
            unsigned char pci_bus, pci_device_fn, pci_latency;
            long ioaddr;
            int irq;

            u16 pci_command, new_command;

            if (pcibios_find_device(PCI_VENDOR_ID_SAMSUNG, // 0x10c3
                                   PCI_DEVICE_ID_KS8920, // 0x8920
                                   pci_index, &pci_bus,
                                   &pci_device_fn))
                break;
            /* Get and check the bus-master and latency values. */
            pcibios_read_config_word(pci_bus, pci_device_fn, PCI_COMMAND, &pci_command);
            new_command = (pci_command & !PCI_COMMAND_MEMORY) | PCI_COMMAND_MASTER | PCI_COMMAND_IO;
            if (pci_command != new_command) {
                printk(KERN_INFO " The PCI BIOS has not enabled this"

```

```

        " device! Updating PCI command %4.4x->%4.4x.\n", pci_command,
new_command);
    }
    pcibios_write_config_word(pci_bus, pci_device_fn, PCI_COMMAND, new_command);
}

pcibios_read_config_byte(pci_bus, pci_device_fn, PCI_LATENCY_TIMER, &pci_latency);
if (pci_latency < 32) {
    printk(" PCI latency timer (CFLT) is unreasonably low at %d."
          " Setting to 32 clocks.\n", pci_latency);
    pcibios_write_config_byte(pci_bus, pci_device_fn, PCI_LATENCY_TIMER, 32);
} else
    printk(" PCI latency timer (CFLT) is %#x.\n", pci_latency);

#endif LINUX_VERSION_CODE >= 0x20155 || PCI_SUPPORT_1
{
    struct pci_dev *pdev = pci_find_slot(pci_bus, pci_device_fn);
    ioaddr = pdev->base_address[1]; /* Use [0] to mem-map */
    irq = pdev->irq;
}
#else
{
    u32 pci_ioaddr;
    u8 pci_irq_line;

    printk( KERN_INFO "linux_version_code >= 0x20155 or PCI_SUPPORT_1\n");
    pcibios_read_config_byte(pci_bus, pci_device_fn, PCI_INTERRUPT_LINE,
&pci_irq_line);
    &pci_ioaddr);
    /* Note: BASE_ADDRESS_1 is for memory-mapping the registers. */
    pcibios_read_config_dword(pci_bus, pci_device_fn, PCI_BASE_ADDRESS_0,
ioaddr = pci_ioaddr;
    irq = pci_irq_line;
}
#endif
/* Remove I/O space marker in bit 0. */
ioaddr &= ~3;

#endif KDEBUG
Debug_ioaddr = ioaddr;
printk( KERN_INFO "Found Samsung ks8920 PCI controller at I/O %lx, IRQ %d.\n",
ioaddr, irq);

ks8920_probe( dev, ioaddr, irq, adapters_found );
dev = NULL;
adapters_found++;

#endif KDEBUG
printk("ks8920_init() : %d Adapter(s) found.\n", adapters_found);
#endif
}

```

코드 1321. 디바이스의 초기화

`ks8920_init()`함수는 device의 초기화에 관련된 부분이다. 여기서는 PCI configuration area space를 읽고, 우리가 찾고자 하는 device 있는지를 확인한다. 이것은 Device ID와 Vendor ID를 읽어서 비교해 봄으로써 알 수 있다. PCI command부분을 읽어서 우리가 요구하는 수준에 해당하는지를 확인한 다음 그럴지 않을 경우에는 직접 PCI configuration area space에 직접 write한다. 요구하는 것으로는 I/O mapped I/O를 지원해야 하고, PCI master로서 동작해야 하는 것 들이다.

다음으로 해주어야 하는 일은 device의 latency를 조정한다. 그리고 나서, PCI bus와 function을 이용해서 base address와 interrupt number를 읽어 들인다. 여기서 읽어드린 PCI Base address는 하위의 LSB에서부터 시작해서 4개의 bit가 다음과 같은 의미를 지니기에, 2bit을 지워주었다(clear).

- Bit 0 : 0이 bit은 space bit으로 0은 이 영역이 memory 주소 공간을 mapping한다는 말이며, 1은 I/O 주소 공간을 mapping한다는 말이다.
- Bit 1과 2 : 00인 경우 – 32 bit decoder를 사용하며, 4 GBytes 이하의 어디에도 위치할 수 있다. 01인 경우 – 1MBytes 이하에 위치한다. 10인 경우 – 64bit decoder를 사용하며, 2의 64승 메모리 공간에 위치한다. 이것은 이 레지스터가 64 bit 크기임을 암시하며, 다음에 있는 double word(4 Bytes)까지도 포함하고 있음을 나타낸다. 11인 경우 – 예약되었다.
- Bit 3 : prefetchable을 나타내는 값으로 해당 메모리 공간에 있는 값을 미리 얻어올 수 있는가를 나타낸다.

다음으로 하게 되는 일은 device structure를 초기화하고 ethernet device driver로 kernel에 등록시키는 것이다.

22.1.9. Device structure의 등록

Device structure는 앞으로 kernel이 device driver를 호출하게 될 때, 계속적으로 사용하게 된다. ks8920_probe()함수를 보도록 하자.

```
dev = init_etherdev(dev, sizeof(struct ks8920_private));
```

에서 device structure에 우리가 사용하는 private structure를 등록하고, 이것을 ethernet device를 나타내도록 만들어준다. 물론 ethernet device structure가 되도록 하기 위해서 초기화해 주는 역할도 한다.

```
static void ks8920_probe(struct device *dev, long ioaddr, int irq, int card_idx )
{
    static int print_version = 0;                                /* Already printed version info. */
    struct ks8920_private *sp;
    int i, option;
    unsigned short reg_value;
    unsigned char subvendor_ID, subsystem_ID;

    /* Print Version Information */
    if( print_version++ == 0)
        printk(KERN_INFO "%s\n", version);

    dev = init_etherdev(dev, sizeof(struct ks8920_private));

    if (dev->mem_start > 0)
        option = dev->mem_start;
    else
        option = 0;

    /* Get the Subvendor ID, Subsystem ID and Hardware Address */
    reg_value = read_eeprom( ioaddr,0 );

    subvendor_ID = reg_value;
    subsystem_ID = reg_value >> 8;
#ifndef KDEBUG
    printk( KERN_INFO "subvendor id : %x\n", subvendor_ID );
#endif

    // Get the network address from serial eeprom.
    for (i=0;i<3;i++) {
        reg_value = read_eeprom( ioaddr,i+2 );
    /*
        // Issue a eeprom read command.
        outl((NODE_ID_START_ADDRESS+i)|0xc000, ioaddr+PROM_Ctl);
    */

    // Wait for read operation done.
}
```

```

        do {
            reg_value = inl(ioaddr+PROM_Ctl);
        } while(reg_value & 0x8000);

        // Read a word.
        reg_value = inl(ioaddr+PROM_Data );
        dev->dev_addr[i*2] = reg_value;
        dev->dev_addr[i*2+1] = reg_value >> 8;
    }

    /* Reset the chip */
    outl( 0x0000200c, ioaddr+MAC_Ctl);
    udelay( 2000 ); /* 1msec delay */

#ifndef KDEBUG
    printk(KERN_INFO "%s: Samsung 100/10 Mbps PCI Ethernet Adapter at %#3lx, ", dev->name, ioaddr);
    for (i = 0; i < 5; i++)
        printk("%2.2X:", dev->dev_addr[i]);
    printk("%2.2X, IRQ %d.%n", dev->dev_addr[i], irq);
#endif
/* We do a request_region() only to register /proc/ioports info. */
request_region(ioaddr, KS8920_TOTAL_SIZE, "SAMSUNG KS8920 Ethernet");

    dev->base_addr = ioaddr;
    dev->irq = irq;

    if (dev->priv == NULL)
        dev->priv = kmalloc(sizeof(*sp), GFP_KERNEL);
    sp = dev->priv;
    memset(sp, 0, sizeof(*sp));
    sp->next_module = root_ks8920_dev;
    root_ks8920_dev = dev;

    if(option > 0) {
        sp->link_speed = (option & 0x20) ? 1 : 0;
        sp->link_speed = (option & 0x60) ? 2 : sp->link_speed;
        sp->full_duplex =(option & 0x10) ? 1 : 0;
    }
    else {
        // default : auto sense
        sp->link_speed = 2;
    }

    /* The ks8920-specific entries in the device structure. */
    dev->open = &ks8920_open;
    dev->hard_start_xmit = &ks8920_start_xmit;
    dev->stop = &ks8920_close;
    dev->get_stats = &ks8920_get_stats;
    dev->set_multicast_list = &ks8920_set_multi_list;
    dev->do_ioctl = &ks8920_ioctl;

    return;
}

```

코드 1322. 디바이스의 probing

그리고 나서 PROM으로부터 ethernet address를 읽어 들이게 된다, 여기서 주의해야 할 점이 PROM에 대한 operation을 마칠 때까지 기다려서 결과를 나타내는 register를 읽어야 한다는 점이다. Ethernet주소를 다 읽고 나면, chipset을 reset한 다음, device의 register들이 사용할 주소 공간을 예약한다. 다음으로는 link speed에 대한 default값을 설정해 준 후, kernel과 device driver의 entry point를 설정한다. 이곳에서는 device driver에 대한 open, transmission, statistics, close, multicast setting, ioctl에 대한 kernel interface를 정하는 것이다.

22.1.10. Device open

이곳이 device에 초기화에 대한 적합한 부분이 된다. Interrupt를 사용할 수 있도록 등록하게 되며, module이 사용중인 counter의 수를 증가시켜주며, Tx queue과 Rx queue를 초기화하게 되고, 다음을 호출하여 device의 link speed를 결정, CAM register의 내용을 초기화 한다.

ks8920_initialize()

또한, 이 routine에서 하는 일은 Tx descriptor에 대한 초기화와 Rx 측에서의 socket buffer를 할당하고, 이것을 chain으로 만들어서 DMA에서 사용하도록 만든다. 여기서 중요한 것은 memory의 주소를 주게될 때, physical address로 주어야 한다는 것이다. 이것을 위해서, *virt_to_bus()*를 사용한다.

```
static int ks8920_open(struct device *dev)
{
    struct ks8920_private *sp = dev->priv;

#ifndef KDEBUG
    printk(KERN_DEBUG "%s: ks8920_open() irq %d.%w", dev->name, dev->irq);
#endif
    if( request_irq(dev->irq, &ks8920_interrupt, SA_SHIRQ, "Samsung 100/10 Mbps PCI Ethernet Card", dev))
        return -EAGAIN;

    MOD_INC_USE_COUNT;
    sp->enq_tx = sp->deq_tx = sp->enq_rx = sp->deq_rx =0;

    dev->if_port = 0;
    dev->tbusy = 0;
    dev->interrupt = 0;
    dev->start = 1;
    sp->in_interrupt = 0;
    sp->tx_full = 0;

    return ks8920_initialize(dev);
}
```

코드 1323. ks8920_open() 함수

그리고 나서 하는 일은 register들에 대한 초기화이다. 이 초기화가 끝나면 receive쪽에서의 interrupt는 enable이 되므로 주의해야 할 것이다.

```
...
ks_outl( DMA_Ctl, DMACtl_DmBurst );
ks_outl( TxFrmPtr, virt_to_bus( &sp->tx_ring[0] ) );
ks_outl( TxThrsh, TX_THRSH_SIZE );
ks_outl( TxPollCtr, TX_POLL_CTL_SIZE );
ks_outl( BLFrmPtr, virt_to_bus( &sp->rx_bufflist ) );
ks_outl( RxFragSize, 0x00000100 );
// Interrupt Enable..
ks_outl( FDA_Bas, virt_to_bus( &sp->rx_ring[0] ) );
ks_outl( FDA_Lim, sizeof(RxFDA)*(RX_RING_SIZE-1) );
ks_outl( Int_En, IntEn_DParDEn | IntEn_DParErrEn | IntEn_SSysErrEn | IntEn_RMAsAbtEn | IntEn_RTargAbtEn |
IntEn_STargAbtEn ); // | IntEn_BLEn | IntEn_FDAExE );

if( sp->full_duplex )
    ks_outl( MAC_Ctl, MACCtl_EnMissRoll | MACCtl_FullDup );
else
    ks_outl( MAC_Ctl, MACCtl_EnMissRoll );

ks_outl( CAM_Ctl, CAMCtl_CompEn | CAMCtl_BroadAcc );
ks_outl( Miss_Cnt, 0x00000000 );
ks_outl( Rx_Ctl, RxCtl_RxEn | RxCtl_EnAlign | RxCtl_EnCRCErr | RxCtl_EnOver | RxCtl_EnLongErr | RxCtl_EnGood );
```

```
ks_outl( Tx_Ctl, TxCtl_EnComp | TxCtl_TxEn );
...
```

코드 1324. 디바이스 레지스터들의 초기화

DMA가 어떻게 일어나게 될지를 결정하고, FDA(Frame Descriptor Area)와 FDA가 어디까지 사용하게 되는지, 그리고 interrupt에 대한 enable 및을 하는 것을 볼 수 있다. 또한 CAM에 대해서 Compare 기능을 enable시키고, broadcast를 받아들이도록 setting한다. 자, 이와 같은 과정을 마쳤으므로 이제 packet을 주고 받을 수 있도록 하는 준비과정을 마쳤다.

22.1.11. Packet의 전송

ks8920_start_xmit() 함수를 참조하자. 이곳에서는 보내고자 하는 packet을 Tx FDA에 setting한 다음, Tx를 enable시키는 register를 on시킨다. 이렇게 하면, transmission이 일어나게 된다. 여기서 사용할 수 있는 방법에는 두 가지를 chip에서 제공하는데, 첫번째 방법으로는 주기적으로 polling해서 보내는 방법과, 보내고자 하는 packet이 있을 때마다 즉시 보내는 방법이 있다. 여기에선 생기는 즉시 wakeup시켜서 보내도록 하고 있다.

```
static int ks8920_start_xmit(struct sk_buff *skb, struct device *dev)
{
    struct ks8920_private *sp = (struct ks8920_private *)dev->priv;
    unsigned long flags;
    int entry;
    u32 reg_value;

#ifdef KDEBUG
//      printk(KERN_INFO "%s: ks8920_start_xmit()%n", dev->name );
#endif
    if (test_and_set_bit(0, (void*)&dev->tbusy) != 0) {
#ifdef KDEBUG
        printk(KERN_INFO "tx full(tbusy set)%n" );
#endif
#ifndef KDEBUG
        return 1;
#endif
    }
    if(skb == NULL ) {
        dev_tint( dev );
        return 0;
    }
    if( sp->tx_full ) {
#ifdef KDEBUG
        printk(KERN_INFO "tx full%n" );
#endif
#ifndef KDEBUG
        return 1;
#endif
    }
    save_flags(flags);
    cli();
    /* Calculate the Tx descriptor entry. */
    entry = sp->enq_tx;
    sp->tx_ring[entry].tx_skbuff = skb;
    sp->tx_ring[entry].freebufflist[0].data = virt_to_bus(skb->data);
    sp->tx_ring[entry].freebufflist[0].length = ETH_ZLEN < skb->len ? skb->len : ETH_ZLEN; /* minimum len */
    sp->tx_ring[entry].freebufflist[0].u.s.COwnsFD = BD_OWNER_CONTROLLER;
    sp->tx_ring[entry].u.s.BDCount = 1;
    sp->tx_ring[entry].u.s.COwnsFD = FD_OWNER_CONTROLLER;
    sp->enq_tx = (sp->enq_tx+1)%TX_RING_SIZE;
    restore_flags(flags);
    if( sp->deq_tx == ((sp->enq_tx+2)%TX_RING_SIZE) )
        sp->tx_full = 1;
    clear_bit(0, (void*)&dev->tbusy);
    dev->trans_start = jiffies; /* save the timestamp */

    reg_value = ks_inl( DMA_Ctl );
}
```

```

ks_outl( DMA_Ctl, reg_value | DMACtl_TxWakeUp );
#ifndef KDEBUG
    if( sp->tx_full )
        printk( KERN_INFO "enq_tx:%d, deq_tx:%d, tx_full:%d\n", sp->enq_tx, sp->deq_tx, sp->tx_full );
//print_register_map();
#endif
    return 0;
}

```

코드 1325. 데이터의 전송

보내고자 하는 packet은 kernel로부터 socket buffer의 형태로 넘겨 받으며, 이것은 IP packet이 된다. 이곳에서는 Tx Queue에 대한 조작이 있게 되므로 critical section problem이 일어날 가능성이 있게 된다. 따라서 이와 같은 문제를 해결하기 위해서 device driver가 사용 중인 경우에는 보내지 못하도록 하며, 또한 interrupt가 발생해서 Tx queue에 대한 조작을 못하도록 막아주게 된다. 여기서 사용하는 제공되는 함수⁵²⁰로는 test_and_set_bit(0, (void*)&dev->tbusy), clear_bit(0, (void*)&dev->tbusy)와 interrupt가 일어나지 못하도록 하는, cli()(clear interrupt)를 사용한다. 중요한 점은 우리가 circular queue를 사용해서 Tx queue와 Rx queue를 관리하므로 indexing에 신경을 많이 써야 한다는 점이다. FDA에 대한 설정이 끝나면 owner를 controller로 바꾸어 주게 되며, DMA가 일어난 후에는 다시 System으로 자동으로 ownership이 바뀌게 된다. 우리는 이 ownership을 이용해서 queue의 어디까지 전송이 일어났는지를 확인할 수 있게 된다. 이것은 이후에 interrupt처리 routine에서 보게 된다.

22.1.12. Interrupt의 처리

ks8920_interrupt()함수는 인터럽트를 처리하는 함수이다. 앞에서 이미 request_irq()함수를 통해서 인터럽트 핸들러 함수로 등록된다.

```

static void ks8920_interrupt(int irq, void *dev_instance, struct pt_regs *regs)
{
    struct device *dev = (struct device *)dev_instance;
    struct ks8920_private *sp;
    unsigned short status;
    u32 reg_value;
    int ret=0;

    // Disable Interrupt
    reg_value = ks_inl(DMA_Ctl);
    ks_outl(DMA_Ctl, reg_value | DMACtl_IntMask);
    sp = (struct ks8920_private *)dev->priv;
    if(test_and_set_bit(0, (void*)&sp->in_interrupt)) {
        printk(KERN_ERR "%s: SMP simultaneous entry of an interrupt handler.\n", dev->name);
        return;
    }
    dev->interrupt = 1;
    status = ks_inl( Int_Src );

    if( status&IntSrc_IntMacRx )
        ret = ks8920_rx( dev );
    if( status&IntSrc_IntMacTx ) {
        ret = ks8920_tx_proc( dev );
        mark_bh(NET_BH);
    }
    dev->interrupt = 0;
    clear_bit(0, (void*)&sp->in_interrupt);
    ks_outl( Int_Src, IntSrc_IntMacTx | IntSrc_IntMacRx | IntSrc_IntEarNot | IntSrc_IntExBD | IntSrc_IntTxCtlCmp |
    IntSrc_FDAEx | IntSrc_BLEEx | IntSrc_NRAbt );
    // Enable Interrupt
    ks_outl(DMA_Ctl, reg_value & (~DMACtl_IntMask));
    if(ret)

```

⁵²⁰ 실제로는 macro가 될 수도 있다. 따라서 compiler option에서 사용하는 O에 대해서 조심하도록 하자.

```

    ks8920_sw_reset(dev);
    return;
}

```

코드 1326. 인터럽트의 처리

Interrupt의 처리는 간단하다. 먼저 interrupt가 더 이상 일어나지 않도록 하고, device structure에는 interrupt처리가 진행 중이라고 나타낸다. 그리고 나서 interrupt를 일으킨 source register에 대한 조사를 한 후에 그에 맞는 interrupt처리 routine을 호출하게 된다. 즉, Tx가 끝났던가 아니면, Rx가 interrupt가 발생 했는지에 대한 판단을 하게 되는 것이다. 호출이 끝나게 되면, 다시 interrupt를 일어나도록 만들어 주며, 만약 Tx나 혹은 Rx 측에서 buffer exhaustion이 일어났다면 software적으로 device를 다시 reset한다. 이러한 경우는 너무 많은 packet이 들어오던지 아니면, 미처 chip에서 buffer descriptor에 대한 처리를 하지 못하게 될 경우에 발생하게 되며, 또한 device가 hang된 상태에서도 reset을 해주어야 한다. 이런 조건들에 대한 검사를 수행하는 것 역시 device driver writer의 몫이 된다. 다음으로는 Tx측과 Rx측에서의 interrupt처리에 대해서 보도록 하자. 여기서 주요한 일은 Buffer queue의 관리이다.

22.1.13. Reception interrupt처리

Receive가 일어나게 되는 시점은 receive가 일어나서 interrupt가 발생하는 시점이다. 따라서, 이곳에서의 처리는 가능한 한 빨리 일어나야 할 것이다. 중요부분을 살펴보면, Rx FD(Frame Descriptor)의 ownership를 조사한 하여, Controller로 되어 있는 부분에 대해서만 처리하게 된다.

```

static int ks8920_rx(struct device *dev)
{
    struct ks8920_private *sp = (struct ks8920_private *)dev->priv;
    int entry = sp->deq_rx;
    u32 state;
    int ret=0;

#ifdef KDEBUG
    //printk(KERN_INFO "%s: ks8920_rx()\n", dev->name );
#endif
    while(1) {
        struct sk_buff *skb;

        if( sp->rx_ring[entry].u.s.COwnsFD == FD_OWNER_CONTROLLER ) {
#ifdef KDEBUG
            //printk(KERN_INFO "%s: ks8920_rx() (entry=%d)\n", dev->name, entry );
#endif
#endif
        }
        if( (state=sp->rx_ring[entry].stat) & RxStat_Good ) {
            int pkt_len = sp->rx_ring[entry].freebufflist[0].length;
            struct sk_buff *skb;

            skb = sp->rx_skbuff[entry];
            if (skb == NULL) {
                printk(KERN_ERR "%s: Inconsistent Rx descriptor chain.\n", dev->name);
                break;
            }
            sp->rx_skbuff[entry] = NULL;
            skb_put(skb, pkt_len);
            skb->protocol = eth_type_trans(skb, dev);
            skb->ip_summed = CHECKSUM_UNNECESSARY; /* don't check it */
            sp->stats.rx_packets++;
            netif_rx(skb);
#endif
        }
#ifdef KDEBUG
        //printk( KERN_INFO "netif_rx\n");
#endif
    }
}

```

```

        else {
            if( !state )
                ret = 1;
                //ks8920_sw_reset( dev );
                // The frame was received with errors:
                // Update the statistics based on the Receive status.
                if( state & RxStat_AlignErr )
                    sp->stats.rx_frame_errors++;
                if( state & RxStat_CRCER )
                    sp->stats.rx_crc_errors++;
                if( state & RxStat_Overflow )
                    sp->stats.rx_fifo_errors++;
                if( state & RxStat_LongErr )
                    sp->stats.rx_length_errors++;
        }

        // Initialize Buffer
        skb = dev_alloc_skb(MAX_BUFFER_SIZE);
        sp->rx_skbuff[entry] = skb;
        if(skb == NULL) {
            printk(KERN_INFO "%s: dev_alloc_skb() error\n", dev->name);
            break;
        }
        skb->dev = dev; /* Mark as being used by this device. */
        sp->rx_bufflist.freebufflist[entry].data = virt_to_bus( skb->tail ); // data?
        sp->rx_bufflist.freebufflist[entry].u.s.COwNSFD = BD_OWNER_CONTROLLER;
        sp->rx_bufflist.freebufflist[entry].length = MAX_BUFFER_SIZE;
        sp->rx_bufflist.freebufflist[entry].RxBID= entry;
        sp->rx_ring[entry].u.s.COwNSFD = FD_OWNER_CONTROLLER;
        sp->rx_ring[entry].freebufflist[0].u.s.COwNSFD = BD_OWNER_CONTROLLER;
        sp->rx_ring[entry].freebufflist[1].u.s.COwNSFD = BD_OWNER_CONTROLLER;
        entry = sp->deq_rx = (sp->deq_rx+1) % RX_RING_SIZE;
    }
    return ret;
}

```

코드 1327. 받기 인터럽트의 처리

Bad packet들에 대한 statistics를 조사한 후, 제대로 된 packet의 경우에는 netif_rx()를 호출하여 상위의 protocol에 알려주게 되며, 새로운 socket buffer를 할당하여 다른 packet을 받을 준비를 하게 된다.

22.1.14. Transmission Interrupt 처리

Tx Interrupt는 packet의 전송이 다 끝났을 경우에 device에서 CPU로 처리가 끝났음을 알려주기 위해서 발생한다. 이곳에서의 일은 packet이 정확히 갔는지에 대한 확인과, Tx Queue의 관리가 주를 이룬다.

```

static int ks8920_tx_proc( struct device *dev )
{
    struct ks8920_private *sp = (struct ks8920_private *)dev->priv;
    u32 state, entry;
    int ret=0;

#ifndef KDEBUG
//      printk(KERN_INFO "%s: ks8920_tx_proc()\n", dev->name );
#endif

    while( sp->deq_tx != sp->enq_tx ) {
        entry = sp->deq_tx;
        if( sp->tx_ring[entry].u.s.COwNSFD == FD_OWNER_CONTROLLER )
            break;
        dev_kfree_skb( sp->tx_ring[entry].tx_skbuff, FREE_WRITE ); /* release the skb buffer */
        state=sp->tx_ring[entry].stat;
        //sp->tx_ring[entry].next |= EOL_BIT;
    }
}

```

```

sp->tx_ring[entry].length = 0;
sp->tx_ring[entry].u.fdctl = 0;
//sp->tx_ring[entry].freebufflist[0].u.s.COwnsFD = BD_OWNER_SYSTEM;
//sp->tx_ring[entry].u.s.COwnsFD = FD_OWNER_SYSTEM;

if( state & TxStat_Comp )
    sp->stats.tx_packets++;
if( state & TxStat_TxPar )
    sp->stats.tx_frame_errors++;
if( state & TxStat_ExColl )
    sp->stats.tx_aborted_errors++;
if( state & TxStat_LateColl )
    sp->stats.tx_window_errors++;
if( state & TxStat_Under ) {
    sp->stats.tx_fifo_errors++;
    //ks8920_sw_reset(dev);
    ret = 1;
#endif KDEBUG
#endif
        printk(KERN_INFO "%s: under run errors\n", dev->name );
}
}
if( state & TxStat_LostCrs )
    sp->stats.tx_fifo_errors++;
if( state & TxStat_TxDefer )
    sp->stats.tx_deferred++; */
if( state & TxStat_TxColl )
    sp->stats.collisions += (state & TxStat_TxColl);
sp->deq_tx = (sp->deq_tx+1)%TX_RING_SIZE;
sp->tx_full = 0;
}
return ret;
}

```

코드 1328. 받기 인터럽트의 처리

이곳에서도 역시 Ownership에 대해서 살펴봄으로써 device가 처리를 어디까지 했는지를 확인할 수 있게 되며, 또한 statistics에 대한 정보 수집이 이루어지고 있다.

22.1.15. Software Reset

Software reset은 hardware의 치명적인 오류로부터 생겨나는 예기치 못한 상황에 대한 대비로 좋은 해결책이 된다. 만약 server로 쓰는 기계가 작동을 멈추거나 하게 되는 경우가 생긴다면, 자동으로 스스로 판단해서 올바른 상태로 system을 만들어 주기 위해서 사용하게 되는 것이다. 우리가 만들려고 하는 device driver도 이러한 상황이 생길 가능성이 있기에 이것에 대비하기 위해서 만들어 준다. 완벽한 것은 없기에 미리미리 대비한다는 의미에서도 좋은 방법이 될 것이다.

```

static void ks8920_sw_reset( struct device *dev )
{
    u32 reg_value;
    int i;
    struct ks8920_private *sp = (struct ks8920_private *)dev->priv;
#ifndef KDEBUG
    printk(KERN_INFO "ks8920_sw_reset()\n");
#endif
    // Reset ks8920
    reg_value = ks_inl( MAC_Ctl );
    ks_outl( MAC_Ctl, reg_value|0x04 );
    udelay( 2000 );
    for( i=sp->deq_tx;i<sp->enq_tx;i++ )
        dev_kfree_skb( sp->tx_ring[i].tx_skbuff, FREE_WRITE ); /* release the skb buffer */
    for( i=sp->deq_rx;i<=sp->enq_rx;i++ ){
        sp->rx_bufflist.freebufflist[i].u.s.COwnsFD = BD_OWNER_CONTROLLER;
    }
}

```

```

    sp->rx_bufflist.freebufflist[i].length = MAX_BUFFER_SIZE;
    sp->rx_bufflist.freebufflist[i].RxBID= i;
    sp->rx_ring[i].u.s.COwnsFD = FD_OWNER_CONTROLLER;
    sp->rx_ring[i].freebufflist[0].u.s.COwnsFD = BD_OWNER_CONTROLLER;
    sp->rx_ring[i].freebufflist[1].u.s.COwnsFD = BD_OWNER_CONTROLLER;
}
ks_outl( TxFrmPtr, virt_to_bus( &sp->tx_ring[0] ) );
ks_outl( BLFrmPtr, virt_to_bus( &sp->rx_bufflist ) );
ks_outl( Rx_Ctl, RxCtl_RxEn | RxCtl_EnAlign | RxCtl_EnCRCErr | RxCtl_EnOver | RxCtl_EnLongErr | RxCtl_EnGood );
ks_outl( Tx_Ctl, TxCtl_EnComp | TxCtl_TxEn );
sp->enq_tx = sp->deq_tx = sp->enq_rx = sp->deq_rx =0;
dev->tbusy = 0;
sp->tx_full = 0;
}

```

코드 1329. 소프트웨어 Reset

주로 하게 되는 일은 Tx와 Rx Buffer queue에 대한 관리와 chip의 reset으로 볼 수 있다. 먼저 이미 가지고 있는 Rx Buffer들에 대해서 release해주며, Rx Free Buffer Descriptor들에 대한 ownership의 reset을 행한다.

22.1.16. Multicast Address의 Setting

우리가 다루는 device는 multicast address과 broadcast address들에 대한 연산을 목적으로 21개의 ethernet address가 존재 될 수 있는 CAM(Contents Addressable Memory)를 제공한다. 이것은 multicast나 broadcast된 packet에 대해서 주소를 빠르게 비교하여 받아 들일 packet을 선택하는데, 사용된다.

```

static void ks8920_set_multi_list(struct device *dev)
{
    struct dev_mc_list *mc_ptr;
    EndianTran TranD;
    int i, offset,j;
    u32 EnableBit;

    if(dev->flags & IFF_PROMISC) { /* Set promiscuous. */
        // StationACC | GroupAcc | BroadAcc
        ks_outl( CAM_Ctl, CAMCtl_StationAcc | CAMCtl_GroupAcc | CAMCtl_BroadAcc );
        return;
    }
    if((dev->flags & IFF_ALLMULTI) || dev->mc_count > (MAX_MULTICAST_LIMIT-1) ) {
        // get all multicast_packet
        // GroupAcc
        ks_outl( CAM_Ctl, CAMCtl_GroupAcc );
        return;
    }
    // clear CAM
    TranD.data = 0;
    ks_outl( CAM_Adr, 4 );
    TranD.tdata[2] = dev->dev_addr[5];
    TranD.tdata[3] = dev->dev_addr[4];
    ks_outl( CAM_Data, TranD.data );
    if( dev->mc_count == 0 ) { // get only own packets
        // CompEn
        ks_outl( CAM_Ctl, CAMCtl_CompEn | CAMCtl_BroadAcc );
        return;
    }
    // set mc list
    for( i=1,offset=4,mc_ptr=dev->mc_list; mc_ptr; mc_ptr=mc_ptr->next, offset+=4, i++ ) {
        // mc_ptr->dmi_addr
        if( i%2 ) {
            TranD.tdata[0] = mc_ptr->dmi_addr[1];

```

```

        TranD.tdata[1] = mc_ptr->dmi_addr[0];
        ks_outl( CAM_Adr, offset );
        ks_outl( CAM_Data, TranD.data );
        TranD.tdata[0] = mc_ptr->dmi_addr[5];
        TranD.tdata[1] = mc_ptr->dmi_addr[4];
        TranD.tdata[2] = mc_ptr->dmi_addr[3];
        TranD.tdata[3] = mc_ptr->dmi_addr[2];
        offset += 4;
        ks_outl( CAM_Adr, offset );
        ks_outl( CAM_Data, TranD.data );
        printk( KERN_INFO "CAM address - %02x:%02x:%02x:%02x:%02x:%02xWn", TranD.tdata[1],
TranD.tdata[0], TranD.tdata[3], TranD.tdata[2], TranD.tdata[1], TranD.tdata[0] );
    }
    else {
        TranD.tdata[0] = mc_ptr->dmi_addr[3];
        TranD.tdata[1] = mc_ptr->dmi_addr[2];
        TranD.tdata[2] = mc_ptr->dmi_addr[1];
        TranD.tdata[3] = mc_ptr->dmi_addr[0];
        ks_outl( CAM_Adr, offset );
        ks_outl( CAM_Data, TranD.data );
        TranD.tdata[2] = mc_ptr->dmi_addr[5];
        TranD.tdata[3] = mc_ptr->dmi_addr[4];
    }
}
if( !(i%2) ) {
    TranD.tdata[0] = TranD.tdata[1] = 0;
    ks_outl( CAM_Adr, offset );
    ks_outl( CAM_Data, TranD.data );
}
for( j=EnableBit=0;j<i;j++)
    EnableBit += (1<<j);
ks_outl( CAM_Ena, EnableBit );
// CompAen
ks_outl( CAM_Ctl, CAMCtl_CompEn | CAMCtl_BroadAcc );
}

```

코드 1330. Multicast Address의 설정

한가지 짚고 넘어갈 mode는 promiscuous mode이다. 이것은 multicast나 broadcast mode와는 다른 모든 packet에 대해서 받아 들인다는 mode로 주로 network에서 날아다니는 packet들에 대한 분석 프로그램에서 이용하는 mode다. 이 mode를 set해 두면, network의 모든 packet들을 application program에서 받을 수 있게 된다. 이 같은 mode는 device의 flag field에 setting해 주어서 현재 device가 어떤 mode에서 동작하는지를 기억해 둔다.

또 한가지 주의할 점은 우리가 addressing하는 memory는 4byte의 공간이나, ethernet address는 6 byte로 이루어져 있다는 점이다. 또한, 상위의 byte와 하위의 byte의 저장되는 순서가 뒤바뀌어 있다는 점인데, 이것에 유의해서 CAM을 programming해 주어야 한다.

22.1.17. Device I/O control

이것에 대해서 우리가 만들고자 하는 device driver에서는 아무런 일도 해주지 않는다.

```

static int ks8920_ioctl(struct device *dev, struct ifreq *rq, int cmd)
{
    switch(cmd) {
        default:
            return -EOPNOTSUPP;
    }
}

```

코드 1331. ioctl() 메소드의 정의

간단히 EOPNOTSUPP를 return함으로써 처리를 끝낸다. 물론 만약 debugging의 목적으로 상위의 program에서 정보를 얻기 위해서 device I/O control을 호출했다면 그에 해당하는 정보를 return하는 것으로 만족될 수도 있을 것이다.

22.1.18. Device Driver의 Close

모든 device에 대한 operation이 종료하게 되면, 이전에 할당했던 resource에 대한 release를 행해 주어야 한다. 이것이 가장 적절한 부분이 바로 device를 close하는 부분이 된다. 주로 open에서 해주었던 buffer들에 대한 할당을 release하게 되며, interrupt를 다시 system에 돌려주는 일이다.

```
static int ks8920_close(struct device *dev)
{
    struct ks8920_private *sp = (struct ks8920_private *)dev->priv;
    int i;
    u32 reg_value;

    dev->start = 0;
    dev->tbusy = 1;

#ifndef KDEBUG
    printk(KERN_DEBUG "%s: Shutting down ethercard.%n", dev->name );
#endif

    /* Disable interrupts */
    reg_value = ks_inl( DMA_Ctl );
    ks_outl( DMA_Ctl, reg_value | DMACtl_IntMask );
    free_irq( dev->irq, dev );

    /* Free all the skbuffs in the Rx*/
    for (i = 0; i < RX_RING_SIZE; i++) {
        struct sk_buff *skb = sp->rx_skbuff[i];
        sp->rx_skbuff[i] = 0;
        /* Clear the Rx descriptors. */
        if (skb)
            dev_free_skb(skb);
    }
    MOD_DEC_USE_COUNT;
    return 0;
}
```

코드 1332. 디바이스 드라이버 닫기(close)

물론 device driver module에 사용에 대한 count를 decrement하는 것도 잊지 않도록 하자. 즉, kernel은 device driver module의 usage counter가 0이 될 때 실제로 device driver module을 unload하게 된다. 또한 interrupt를 release하기 전에 먼저 device의 interrupt를 일어나지 않게 해주는 것도 잊어선 안 된다.

이곳에서 논하지 않은 것으로는 Timer에 대한 처리와 driver가 사용되는 도중에 생기는 Media의 속도 변화에 대해서 대처해 주는 부분이다. Timer는 timeout의 처리와 현재상태의 media의 speed를 점검하고 그에 맞춰서 적당히 chip의 register를 다시 설정해주는 역할을 한다. 또한 Queue가 잘못되었을 경우 이것을 회복하는 기능도 해주고 있다.

22.1.19. Network의 Setting

자 이젠 실제적으로 code를 compile 한 후에 이것을 실제로 쓰는 과정만이 남았다. 드라이버를 설치한 후에는 Network 상황에 따라 적절한 setting이 필요한데, 여기에는 IP address의 할당 및 network mask, broadcast address를 주어야 하며, routing table에 등록 시켜야 한다. 이와 같은 일을 하기위해서는 ifconfig과 route라는 명령어를 사용한다. 반드시 root권한을 가진 사용자만 해주어야 한다.

다음과 같은 방법으로 해보자.

```
(ex) ifconfig eth0 165.213.175.46
     ifconfig eth0 broadcast 165.213.175.63
     ifconfig eth0 netmask 255.255.255.192
     route add -net 165.213.175.0 netmask 255.255.255.192 dev eth0
     route add default gw 165.213.175.1
```

예제 5. Network의 setting

위의 예에서 eth0는 device에 대한 interface name이 되며, Ethernet card가 둘 이상이 존재할 경우에는 eth0, eth1, eth2, .. 순으로 운영체제가 Ethernet device에 이름을 할당한다. 따라서, 명령어를 실행하기에 앞서, 자신이 설치한 device의 이름을 확인하는 것이 필요하다. 이 경우, ifconfig 명령을 사용해서 현재 설치된 network device들의 이름을 확인할 수 있을 것이다.

위의 예제는 참고로서 사용하기 바라며, 자신의 network상황에 대해서 잘 이해한 후에 하기 바란다. Driver module을 제거하기 전에 반드시 다음과 같은 명령어로 더 이상 이 network interface를 사용하지 않는다는 것을 알려 주어야 한다. 주의 하기 바란다.

ex) ifconfig eth0 down

네트워크 디바이스 드라이버는 받기와 보내기가 주안점이다. 보내는 도중에도 받기가 일어날 수 있기에 항상 그에 대비해야 할 것이며, 보내는 것과 받는 것은 같이 움직이게 되므로, 디바이스 드라이버의 일부분을 짰을 때, 그것이 제대로 수행되는지를 확인해야 할 것이다. 많은 시행착오를 이미 각오하리라고 생각한다.

22.2. Init.h 파일의 분석

init.h 파일은 아마도 Linux를 분석하는데 있어서, 시작점이 될 수 있는 부분들 중에서 가장 중요한 한 부분이 될 수 있을 것이다. 우리가 이것을 보고자 하는 이유는 초기화에 관련된 대부분의 함수들이 바로 여기서 정의된 것을 사용하고 있으며, Compiler 및 Linker와도 밀접한 관련을 가지기 때문이다. 크게 보면 다음과 같은 두 가지의 것들로 분류되는 것들에 대한 것을 이 파일에서 제공한다.

- 커널의 초기화 시에 자동으로 실행해 주어야 하는 부분.
- 커널의 초기화 시에 수동으로 일일이 호출해 주어야 하는 부분.

이것은 모듈로 작성된 드라이버나 기타 등등의 커널 구성 요소 중에서도 자신을 초기화하기 위해서 수행해야 하는 함수(function)와 커널이 하위의 시스템을 초기화하기 위해서 호출하는 함수들이 그 구성을 이루고 있다.

이 파일은 다시 크게 보면, 모듈을 컴파일 할 때 사용하는 부분과 그렇지 않은 부분으로 나뉘어 진다(`#ifndef MODULE ~ #else ~ #endif`). 따라서, 모듈을 프로그램하는 사람들은 가장 먼저 MODULE이라는 것을 정의해야 할 것이다. 또한, MODULE을 정의하지 않은 부분(`#ifndef MODULE`)은 다시 `_ASSEMBLY_`를 정의한 부분과 그렇지 않은 부분으로 다시 나뉜다.

22.2.1. 초기화(initialization) 코드의 수행

`_ASSEMBLY_`가 정의되어 있지 않다면, 아래와 같은 것들이 사용될 것이다. 즉, 초기화 시에 수행되는 함수들과 커널에 넘겨주는 파라미터(parameter) 값을 정의하는 것들이다.

```
#ifndef __ASSEMBLY__
/*
 * Used for initialization calls..
 */
typedef int (*initcall_t)(void);
```

```

typedef void (*exitcall_t)(void);
extern initcall_t __initcall_start, __initcall_end;
#define __initcall(fn) \
    static initcall_t __initcall_##fn __init_call = fn \
#define __exitcall(fn) \
    static exitcall_t __exitcall_##fn __exit_call = fn \
...

```

코드 1333. 커널 초기화 함수 및 커널 파라미터에 대한 macro 정의

`initcall_t`과 `exitcall_t`은 초기화 함수들이 가질 수 있는 함수의 type에 대한 것을 정의하고 있다. `__initcall_start`와 `__initcall_end`는 실제로는 LD(Linker)의 script로 사용되는 파일에 정의된 것으로 각각은 초기화 함수들의 시작번지와 초기화 함수의 마지막 번지를 나타낸다. 정의는 i386인 경우에는 `linux/arch/i386/vmlinux.lds`라는 파일에 아래와 같이 나와 있다.

```

...
.= ALIGN(4096);           /* Init code and data */521
__init_begin = .;
.text.init : { *(.text.init) }
.data.init : { *(.data.init) }
.= ALIGN(16);
__setup_start = .;
.setup.init : { *(.setup.init) }
__setup_end = .;
__initcall_start = .;
.initcall.init : { *(.initcall.init) }
__initcall_end = .;
.= ALIGN(4096);
__init_end = .;
...

```

코드 1334. `__init_begin`과 `__init_end` 사이에 정의된 값들

`vmlinux.lds`는 커널을 build할 때 컴파일된 커널의 object파일을 실행 파일로 만들고자 할 때, 각각의 section⁵²²들을 어떻게 배치할 것인가를 결정하는 Linker의 input으로 사용된다. 여기서 주목할 점은 바로, `__init_begin`과 `__init_end`라고 정의된 변수이다. 시작부분에 있어서, 4096이라는 alignment 단위로 배치된다는 것을 알 수 있고, 나중에 `__init_end`앞에서 다시 4096이라는 값으로 alignment된다. 이것은 i386에서 사용하는 page size에 맞춰서 정렬한 다음, 나중에 이를 다시 회수하기 위한 것이다. 즉, 초기화를 마치고 나면, 더 이상 이곳에 있는 코드는 사용할 필요가 없기 때문에, `__init_begin`에서 `__init_end`까지의 메모리 공간은 해제된다. 나중에 해제된 이 부분은 다른 목적으로 다시 할당될 것이다.

`__initcall(fn)/__exitcall(fn)`이라는 매크로를 사용해서 정의된 함수들은 `__initcall_init`와 `__initcall_end` 사이에 전부 `__initcall_fn`이라는 함수의 포인터를 가지게 되며, 이 함수의 포인터에 자신의 함수(fn)를 가르키도록 만들어 준다. 즉, `__initcall(fn)`은 `__initcall_fn`이라는 함수 포인터를 `fn`으로 가르키도록 만들어

⁵²¹ 여기서 “.”이 의미하는 바는 생성되는 binary image파일에서의 현재의 위치를 말한다.“.”은 자동으로 증가하게 되며, 각 변수의 시작 위치를 정하는데 주로 사용된다.

⁵²² ELF(Executable and Linking Format)에 정의된 section이다. 현재 Linux 커널은 ELF 파일 format으로 compile되며, 최초의 실행될 코드는(즉, BSP : Board Support Package) 커널 image의 제일 첫 부분에 오도록 만들어진다. 이것은 LD의 input으로 사용되는 linker script에 의해서 압축된 커널의 이미지가 어떻게 놓이게 될지를 결정해 줄때 알 수 있다. 현재는 0xC0000000 + 0x1000000에서부터 커널의 image가 올라오리라는 것을 `vmlinux.lds`를 보고 예상할 수 있다. 따라서, boot loader는 이 위치에 zImage를 옮겨놓을 수 있어야 한다. 이는 커널의 symbol들이 `vmlinux.lds`에 의해서 이 위치를 기준으로 해서 모두 생성되기 때문이다.

준다. 나중에 `__initcall_fn`이라고 정의된 함수들은 전부 `linux/init/main.c`에 있는 `do_initcalls()`라는 함수에서 정의된 순서대로 차례로 실행될 것이다⁵²³.

```
static void __init do_initcalls(void)
{
    initcall_t *call;

    call = &__initcall_start;
    do {
        (*call)();
        call++;
    } while (call < &__initcall_end);
    /* Make sure there is no pending stuff from the initcall sequence */
    flush_scheduled_tasks();
}
```

코드 1335. `do_initcalls()` 함수의 정의

`do_initcalls()` 함수는 `__initcall_start`에서 함수를 하나씩 읽어 들여서 하나씩 수행해 나가는 역할을 수행한다. 이때, `__initcall_end`를 만나면, 더 이상 수행하지 않고, `flush_scheduled_tasks()` 함수를 호출해서 `__initcall_start`에서부터 수행하던 함수들에 의해서 스케줄링된 task를 다 수행하도록 만들어준다.

```
static struct tq_struct dummy_task;

void flush_scheduled_tasks(void)
{
    int count;
    DECLARE_WAITQUEUE(wait, current);

    add_wait_queue(&context_task_done, &wait);
    for (count = 0; count < 2; count++) {
        set_current_state(TASK_UNINTERRUPTIBLE);

        /* Queue a dummy task to make sure we get kicked */
        schedule_task(&dummy_task);

        /* Wait for it to complete */
        schedule();
    }
    remove_wait_queue(&context_task_done, &wait);
}
```

코드 1336. `flush_scheduled_tasks()` 함수의 정의

`flush_scheduled_tasks()` 함수는 먼저 자신이 현재의 task(여기서는 `current`로 표현되는 process라고 보아야 할 것이다.)를 `context_task_done`이라는 wait queue에 넣고(`add_wait_queue()`), `count`가 2보다 작은 값인 동안 for loop를 돌게 된다. Loop안에서 현재 task(혹은 process)의 상태를 `TASK_UNINTERRUPTIBLE`로 만들어서, 반드시 수행될 때까지 대기하도록 만들고, `schedule_task()` 함수를 `dummy_task`에 대한 포인터를 넘겨주고 호출한다.

⁵²³ `__initexit(fn)`도 마찬가지로 `__exitcall_fn`이라는 함수의 포인터를 만들어준다. 하지만, 실제로 이렇게 정의된 함수는 버려지게 되는(discard) section으로 전부 모이게 된다. `.text.exit`, `.data.exit`, `.exitcall.exit`등의 section들에 들어가는 코드들이 이에 해당한다. 여기에 있는 코드들은 커널이 초기화 및 working중인 동안에는 결코 수행될 일이 없을 것이다. 따라서, 커널의 image에도 반영되지 못한다. 이것은 실제로 Linker의 output으로 생성되지 못한다고, LD manual의 3.6.7에서 말하고 있다.

```
int schedule_task(struct tq_struct *task)
{
    int ret;

    need_keventd(__FUNCTION__);
    ret = queue_task(task, &tq_context);
    wake_up(&context_task_wq);
    return ret;
}
```

코드 1337. schedule_task() 함수의 정의

schedule_task() 함수는 먼저 need_keventd() 함수를 호출해서 현재 kernel event daemon이 실행 중인지를 확인하고, queue_task()를 호출해서 tq_context task queue에 넘겨받은 task를 넣어준다. wake_up() 함수는 대기 큐(context_task_wq)가 현재 사용되고 있다.)에서 task들을 깨워주는 일을 수행한다. 복귀 값을 queue_task() 함수(매크로로 실제로 정의되어 있다.)의 결과 값을 넘겨줄 것이다.

flush_scheduled_tasks()로 돌아가서, schedule_task() 함수의 호출 후에는 바로, schedule() 함수⁵²⁴를 호출한다. 이곳에서 수행할 task들이 있다면, 다 수행하게 될 것이다. 여기서 count가 2가 될 때까지 기다리는 이유는 이미 kernel event daemon thread가 run_task_queue() 함수를 이미 진행했을 가능성이 있기 때문이다. 따라서, 확실하게 해두기 위해서 두 번에 걸쳐서 실행하는 것이다. 이제 실행할 task들은 다 수행되었으므로, remove_wait_queue()를 호출해서 context_task_done에 넣어두었던 wait 구조체를 삭제하도록 한다.

이때 한가지 보고 넘어갈 것이 있다. 즉, context_task이다. 이 task는 do_initcall() 함수를 호출하기에 앞서 do_basic_setup() 함수에서 start_context_thread() 함수를 호출해서 생성된다. start_context_thread() 함수는 linux/kernel/context.c에 아래와 같이 되어 있다.

```
int start_context_thread(void)
{
    static struct completion startup __initdata = COMPLETION_INITIALIZER(startup);

    kernel_thread(context_thread, &startup, CLONE_FS | CLONE_FILES);
    wait_for_completion(&startup);
    return 0;
}
EXPORT_SYMBOL(schedule_task);
EXPORT_SYMBOL(flush_scheduled_tasks);
```

코드 1338. start_context_thread() 함수의 정의

Context thread는 이미 init kernel thread가 만들어진 이후에 생성되는 것으로 init thread로부터 사용할 수 있는 자원들에 대한 것을 얻을 수 있기 때문에, 준비작업이 필요없이 바로 kernel_thread()를 호출해서 생성할 수 있다. 동기화를 위해서 completion구조체로 정의된 startup을 사용하고 있으며, 이것이 kernel thread로 생성될 context_thread에 인자로 넘겨진다. EXPORT_SYMBOL()⁵²⁵은 전역(global)로 access가 가능하도록 만들기 위해서 사용하는 방법이다. schedule_task()와 flush_scheduled_tasks()함수가 이렇게 선언된 것들이다.

```
static DECLARE_TASK_QUEUE(tq_context);
static DECLARE_WAIT_QUEUE_HEAD(context_task_wq);
static DECLARE_WAIT_QUEUE_HEAD(context_task_done);
static int keventd_running;
```

⁵²⁴ schedule() 함수는 Linux에서 process나 대기중인 task를 수행하기 위해서 호출하는 함수이다. Scheduling 부분을 참조하도록 하자.

⁵²⁵ EXPORT_SYMBOL()은 실제로 linux/include/linux/module.h에 정의되어 있다.

```

static struct task_struct *keventd_task;
...
static int context_thread(void *startup)
{
    struct task_struct *curtask = current;
    DECLARE_WAITQUEUE(wait, curtask);
    struct k_sigaction sa;

    daemonize();
    strcpy(curtask->comm, "keventd");
    keventd_running = 1;
    keventd_task = curtask;

    spin_lock_irq(&curtask->sigmask_lock);
    siginitsetinv(&curtask->blocked, sigmask(SIGCHLD));
    recalc_sigpending(curtask);
    spin_unlock_irq(&curtask->sigmask_lock);

    complete((struct completion *)startup);

    /* Install a handler so SIGCLD is delivered */
    sa.sa.sa_handler = SIG_IGN;
    sa.sa.sa_flags = 0;
    siginitset(&sa.sa.sa_mask, sigmask(SIGCHLD));
    do_sigaction(SIGCHLD, &sa, (struct k_sigaction *)0);
...

```

코드 1339. context_thread() 함수의 정의

Context thread에서 사용하는 queue로는 tq_context가 task queue로, context_task_wq와 context_task_done을 wait queue로 선언하고 있다. 또한 kernel event daemon이 활성화 중인지를 나타내는 keventd_running flag가 있으며, kernel event daemon이 사용하는 task_struct 구조체에 대한 포인터로 keventd_task를 두고 있다. 먼저 현재 진행중인 task를 curtask로 두고, wait queue를 정의하고 초기화하기 위해서 DECLARE_WAITQUEUE() 매크로를 사용했다. 또한 signal에 대한 처리를 위해서 k_sigaction 구조체로 정의된 sa를 둔다. daemonize() 함수는 daemon process로 생성하기 위해서 필요한 일을 전부 모아서 처리하기 위해서 호출한다. curtask->comm은 현재 실행중인 프로그램의 이름을 명시하는 것으로 이것을 keventd로 두었다. 즉, 나중에 이것은 ps와 같은 명령을 사용해서 shell에서 볼 수 있는 이름이 될 것이다. 또한 keventd_running를 1로 설정해서 이전 RUNNING상태인 context thread(or kernel event daemon)이 있다는 것을 나타낸다. 또한 이 kernel event daemon의 task struct에 대한 포인터를 설정하기 위해서 curtask를 keventd_task에 둔다. siginitsetinv()는 signal set을 나타내는 값에 해당 signal을 설정하는 일을 한다. 여기서는 SIGCHLD(Signal Child)라 signal을 blocked에 설정하기 위해서 사용한다. 즉, blocking된 signal에 SIGCHLD를 설정하도록 한다. recalc_sigpending()은 pending된 signal이 있는가를 확인하는 함수이다. Blocking되지 않은 signal이 도착했는지를 본다. 이 두 가지의 함수가 수행되는 것은 signal mask의 lock을 이용하고 있다. 따라서, 이 두 개의 함수가 수행되는 중에는 다른 곳에서 signal mask를 바꿀 수 없도록 한다. complete() 함수는 completion 구조체의 포인터로 주어진 startup 변수에 처리가 완료되었음을 나타내는 flag를 설정하는 일을 한다.

```

struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};

```

코드 1340. completion 구조체의 정의

completion 구조체에는 done이라는 처리가 완료되었음을 나타내는 flag 필드와 처리가 끝나기를 기다리고 있는 task의 list에 대한 wait queue를 가지는 wait 필드가 있다. complete() 함수⁵²⁶를 호출하는 것으로 done은 증가할 것이며, wait queue에서 대기하고 있는 task들은 깨어나게 될 것이다. 이전 SIGCHLD signal을 받았을 경우에 호출될 handler를 설정하도록 한다. 실제로는 SIG_IGN을 두어서 간단히 ignore(무시)로 두게된다.

```

for (;;) {
    set_task_state(curtask, TASK_INTERRUPTIBLE);
    add_wait_queue(&context_task_wq, &wait);
    if (TQ_ACTIVE(tq_context))
        set_task_state(curtask, TASK_RUNNING);
    schedule();
    remove_wait_queue(&context_task_wq, &wait);
    run_task_queue(&tq_context);
    wake_up(&context_task_done);
    if (signal_pending(curtask)) {
        while (waitpid(-1, (unsigned int *)0, __WALL|WNOHANG) > 0)
            ;
        spin_lock_irq(&curtask->sigmask_lock);
        flush_signals(curtask);
        recalc_sigpending(curtask);
        spin_unlock_irq(&curtask->sigmask_lock);
    }
}
}

```

코드 1341. context_thread() 함수의 정의(계속)

for() loop를 돌면서 이전 keventd가 해주는 일을 보도록 하자. 우린 이미 context kernel thread의 이름이 keventd가 된다는 것을 앞에서 보았다. 현재의 task는 역시 keventd가 될 것이다. 따라서, curtak도 keventd의 task_struct를 가르키고 있다. set_task_state() 함수는 task의 상태를 TASK_INTERRUPTIBLE로 바꾼다. 그리고, add_wait_queue()를 호출해서 context_task_wq wait queue에 넣도록 한다. 만약 TQ_ACTIVE()를 사용해서 검사한 tq_context의 task queue 상태가 활성(active)이라면, set_task_state()를 호출해서 keventd의 상태를 TASK_RUNNING으로 변경한다. 어쨌든 여기까지 진행했다면, schedule를 호출해서 실행 가능한 task들이 실행될 수 있도록 만들어준다. 만약 keventd가 다시 수행될 기회를 얻는다면, remove_wait_queue()에서부터 수행될 것이다. context_task_wq에서 떼어낸 다음, tq_context task queue에 있을 수 있는 task들을 run_task_queue()를 호출해서 하나씩 수행하도록 한다. 이제 context_task_done에 있는 context task(or keventd)이 다 끝나기를 기다리는 task들을 수행하기 위해서 wake_up() 함수를 호출한다.

만약 keventd에 pending된 signal이 있다면, if() 절을 수행한다.

```

if (signal_pending(curtask)) {
    while (waitpid(-1, (unsigned int *)0, __WALL|WNOHANG) > 0)
        ;
    spin_lock_irq(&curtask->sigmask_lock);
    flush_signals(curtask);
    recalc_sigpending(curtask);
    spin_unlock_irq(&curtask->sigmask_lock);
}
}

```

조금 이야기의 논점에서 벗어나긴 했지만, 어쨌든 대략적인 수행 순서를 보았다는 것에 만족하도록 하고, 다음 이야기로 진행하도록 하겠다.

⁵²⁶ 정의는 linux/kernel/sched.c에 있다.

22.2.2. 커널 Parameter에 대한 해석

커널에 넘겨주는 parameter들은 사용자가 커널이 booting할 때 마치 프로그램을 수행할 때와 동일하게 command line을 주는 것과 동일한 역할을 수행한다. 이를 처리하기 위해서 아래와 같은 kernel_param이라는 특별한 구조체를 정의해서 사용한다.

```
...
/*
 * Used for kernel command line parameter setup
 */

struct kernel_param {
    const char *str;
    int (*setup_func)(char *);
};

extern struct kernel_param __setup_start, __setup_end;
#define __setup(str, fn)
    static char __setup_str_##fn[] __initdata = str;
    static struct kernel_param __setup_##fn __attribute__((unused)) __initsetup = { __setup_str_##fn, fn }
#endif /* __ASSEMBLY__ */
...
```

코드 1342. kernel_param구조체 및 관련 매크로의 정의

커널에 넘겨주는 parameter의 구조체는 kernel_param라는 것으로 정의된다. 이 구조체에는 등록할 parameter를 나타내는 문자열(string)과 이 문자열을 해석할 함수(setup_func())가 들어가게 된다. 즉, 커널에 넘겨진 파라미터들을 해석할 함수가 차지하게 된다. 커널에 대한 파라미터들의 시작과 끝은 각각 __setup_start와 __setup_end로 나타내게 되며, 이것 역시 __init_begin과 __init_end사이에 정의된 값이다. 이곳에 있는 커널의 파라미터를 해석하는 것은 linux/init/main.c에 있는 checksetup() 함수이다.

```
static int __init checksetup(char *line)
{
    struct kernel_param *p;

    p = &__setup_start;
    do {
        int n = strlen(p->str);
        if (!strncmp(line, p->str, n)) {
            if (p->setup_func(line+n))
                return 1;
        }
        p++;
    } while (p < &__setup_end);
    return 0;
}
```

코드 1343. checksetup() 함수의 정의

checksetup() 함수는 커널 파라미터 구조체의 문자열(string : str)을 가져와서 이 문자열과 checksetup()이 넘겨받은 line이라는 변수와 비교해서 같은 값을 가진다면, kernel_param 구조체에 정의된 setup_func() 함수를 수행하는 것이다. 이때, setup_func() 함수로 넘겨지는 값은 string은 제외한 값이 들어간다. 따라서, line에는 커널 파라미터에 대한 값이 문자열과 같이 정의되어 있다는 것을 알 수 있다. 호출에 문제가 없었다면, 1을 넘겨주고 그렇지 않다면, __setup_end까지 계속해서 해당하는 문자열을 찾으려고 노력할 것이다. 결과적으로 오류가 있었다면 0을 돌려준다. 또한, checksetup() 함수는 parse_options() 함수에서 호출되며, parse_options() 함수는 start_kernel()에서 architecture dependent한 설정이 끝나고 난 이후에 커널에 주어진 command line을 해석해서 수행하기 위해서 호출된다.

`setup()` 매크로는 `__setup_str_fn[]`이라는 문자열에 str을 넣고, `__setup_fn`이라는 `kernel_param` 구조체에 `__setup_str_fn`을 str 필드에 fn을 `setup_func()` 필드에 넣어준다. 이때 문자열의 다음이나 함수의 다음에 사용되는 `__initdata`와 `__attribute__((unused))` `__initsetup`은 전부 이 데이터나 코드가 어떤 특성을 지니는가를 기술하기 위해서 사용된다. 이것은 나중에 GCC의 `__attribute__` 사용에 대해서 이야기할 때 다시 볼 수 있을 것이다. 여기서는 단지 compile된 코드가 들어갈 section의 위치를 가르쳐준다는 사실만 기억하도록 하자.

22.2.3. Assembly에서 사용되는 section 정의

이전 함수의 선언 뒤에서 GCC의 `__attribute__`⁵²⁷로 주어지는 것을 볼 차례이다. 이것은 ANSI-C에서는 찾아볼 수 없는 GCC만의 독특한 확장기능(extension)을 사용하는 것으로 일반적인 C programmer에게는 익숙치 않은 것들이다.

```
...
/*
 * Mark functions and data as being only used at initialization
 * or exit time.
 */
#define __init           __attribute__ ((__section__ (".text.init")))
#define __exit          __attribute__ ((unused, __section__(".text.exit")))
#define __initdata      __attribute__ ((__section__ (".data.init")))
#define __exitdata      __attribute__ ((unused, __section__ (".data.exit")))
#define __initsetup     __attribute__ ((unused, __section__ (".setup.init")))
#define __init_call     __attribute__ ((unused, __section__ (".initcall.init")))
#define __exit_call     __attribute__ ((unused, __section__ (".exitcall.exit")))

/* For assembly routines */
#define __INIT          .section ".text.init","ax"
#define __FINIT         .previous
#define __INITDATA      .section ".data.init","aw"

#define module_init(x)  __initcall(x);
#define module_exit(x)  __exitcall(x);

#else   /* MODULE */
...

```

코드 1344. Assembly과정에서 사용되는 section들에 대한 매크로들에 대한 정의

이곳에서 정의된 것들은 모두 함수 및 변수 선언의 뒷부분에 놓이게 되는 것으로, 선언되는 함수나 변수가 놓이게 되는 section을 결정해준다. 크게 2가지의 `__attribute__`의 list에 올 수 있는 것을 사용하는데, 각각은 `unused`와 `__section__`이다. `__attribute__`는 “((“와 “))”사이에 attribute의 이 list가 오도록 한다.

- `unused` : 이것은 함수나 변수의 선언 이후에 오는 것으로 함수나 변수가 거의 사용되지 않을 것이라는 것을 의미하는데 쓰인다. `__exit`와 `__exitdata`, `__initsetup`, `__init_call`, `__exit_call`등에 이것을 사용하고 있는 것을 확인할 수 있다.
- `__section__` : 이것은 함수나 변수의 선언 이후에 오는 것으로 section과 동일한 의미를 가진다. 동일한 `__section__`것을 가지는 이유는 변수나 macro의 정의와 혼동되는 것을 방지하기 위한 것으로 컴파일된 binary object가 놓이게 되는 section의 위치를 알려주기 위한 것이다. 나중에 Linker에 의해서 정의된 section에 적절히 배치될 것이다. 놓이는 순서는 Linker에 주어지는 input의 순서로 결정된다. 따라서, 같은 attribute로 선언된 것들도 Linker에 들어가는 input의 순서에 의해서 그 순서가 달라질 수 있다.

⁵²⁷ GCC의 manual인 “Using the GNU C Compiler Collection, Richard M. Stallman”의 3.2 version을 기준으로 한다. 참고하려는 부분은 page 172부터 나와 있다.

여기서는 .text.init, .text.exit, .data.init, .data.exit, .setup.init, .initcall.init, .exitcall.exit 등이 binary object들이 놓이게 되는 section의 값으로 각각 정의되어 있다.

여기서는 또한 __INIT, __FINIT, __INITDATA와 같은 macro 정의가 assembly어로 작성된 program을 위해서 정의되어 있는데, .section으로 자신의 binary image가 위치하게 될 section의 위치와 해당 section의 allocatable(a)/writable(w)/executable(x) 가능여부를 기술하고 있다.⁵²⁸ 또한 .previous로 정의된 __FINIT 매크로는 현재의 section에 대한 정의를 끝내고, 바로 이전의 section에 대한 연속이라는 것을 의미한다.

Module을 program할 때 많이 보게 되는 module_init(x)와 module_exit(x)는 MODULE이 정의되지 않았다면, 여기서와 같이 커널과 static하게 결합하게 될 때 사용된다. 즉, __initcall(x)와 __exitcall(x)로 바뀌게 된다. 따라서, module_init(x)와 module_exit(x)를 사용했을 경우에는 .initcall.init section에 __initcall_x라는 이름의 함수가 들어가게 될 것이다. 따라서, 예를 들어 커널 compile시에 모듈로 설정하지 않는 각종 디바이스 드라이버의 초기화 함수들은 전부 __initcall_x라는 형태의 함수이름을 호출하면서 처리될 것이다. 대부분의 경우에는 module_init()나 module_exit() 매크로의 input으로 들어가는 함수의 정의에 __init나 __exit라는 것을 동반하는 경우가 있는데, 이때는 함수의 이름이 다시 .text.init/.text.exit section에도 나오게 된다. 이것과 __initcall_x라는 함수는 동일한 함수이며, 단지 do_initcalls() 함수에서 호출되도록 만들어 주기 위해서 .initcall.init에 __initcall_x라는 형태로 넣어둔 것 뿐이다. 즉, module로서 loading이 되지 않는 함수는 커널에서 인위적으로 일일이 호출해서 초기화 하지 않고, do_inicalls()에서 자동으로 전부 초기화하도록 만들어 준 것이다.

22.2.4. 모듈(Module)이 정의된 경우의 init.h 파일의 해석

이제 남은 것은 MODULE이라는 것이 compile 옵션으로 들어간 경우가 될 것이다. 이때는 모듈로서 커널과 동적으로 linking을 해야 하기 때문에, 앞에서 정의한 초기화 section은 이미 커널이 working한 후에는 사라지고 없을 것이다. 따라서, 관련된 부분은 전부 NULL(혹은 공란)로 정의된다.

```
#define __init
#define __exit
#define __initdata
#define __exitdata
#define __initcall(fn)
/* For assembly routines */
#define __INIT
#define __FINIT
#define __INITDATA

/* These macros create a dummy inline: gcc 2.9x does not count alias
   as usage, hence the 'unused function' warning when __init functions
   are declared static. We use the dummy __*_module_inline functions
   both to kill the warning and check the type of the init/cleanup
   function. */

typedef int (*__init_module_func_t)(void);
typedef void (*__cleanup_module_func_t)(void);
#define module_init(x) \
    int init_module(void) __attribute__((alias(#x))); \
    static inline __init_module_func_t __init_module_inline(void) \
    { return x; }
#define module_exit(x) \
    void cleanup_module(void) __attribute__((alias(#x))); \
    static inline __cleanup_module_func_t __cleanup_module_inline(void) \
    { return x; }

#define __setup(str,func) /* nothing */
```

⁵²⁸ 이것은 GNU Assembler인 GAS 2.9.1을 참조한 것이다. www.gnu.org에서 확인하기 바란다.

```
#endif /* !MODULE */
```

코드 1345. 모듈(MODULE)로 정의된 경우에 관련된 init.h 파일

`_init`, `_exit`, `_initdata`, `_exitdata`, `_inicall(fn)` 및 `_INIT`, `_FINIT`, `_INITDATA`등은 전부 아무런 값도 가지지 않게 된다. 이젠 중요한 것은 `_init_module_func_t`과 `_cleanup_module_func_t`으로 정의되는 함수의 포인터가 될 것이다. 관련된 매크로로는 `module_init(x)`와 `module_exit(x)`가 있다. 각각은 `init_module()`이라는 함수와 `cleanup_module()`이라는 함수를 x라는 함수의 alias로 생성하고, `_init_module_inline(void)`와 `_cleanup_module_inline(void)`라는 단지 모듈의 초기화와 해제에 관련된 함수의 이름을 둘려주는 inline함수를 정의한다. 여기서 중요한 것은 모든 모듈의 초기화 및 해제에 관련된 함수의 이름이 동일하도록 만들어준다는 점이다. 즉, 모듈을 loading하고, unloading할 때 그 entry point가 되는 함수를 미리 동일하게 정의해 준다는 것이다. 마치 일반적인 프로그램에서 `main()`이라는 entry point를 모두 동일하게 가지는 것과 같다.

여기서 다시 GCC의 attribute의 중의 하나인 alias에 대해서 좀더 알아보기로 하자. 예를 들어 다음과 같은 정의를 했다고 가정하자.

```
#include <stdio.h>
void __foo( int x )
{
    printf("Test print : %d.\n", x );
}

void foo() __attribute__ ((alias("__foo")));

int main(int argc, char** argv )
{
    foo( 10 );
    return 0;
}
```

코드 1346. GCC __attribute__의 alias 예제

이때 `foo()` 함수를 호출하는 것은 `__foo()` 함수를 호출하는 것과 동일하다는 것이다. 즉, `foo()`와 `__foo()`를 서로 같은 함수로 취급하도록 만들어주는 함수의 또 다른 이름(alias)을 정의하는 것이다. 이와 관련된 정의가 많이 보이는 예로는 GLIBC(GNU C Library)가 가장 대표적일 것이다. GNU C Library에서는 사용자가 override할 수 있도록 weak attribute와 같이 사용해서 `__attribute__ ((weak, alias("xxx")))`과 같은 것을 많이 사용하고 있다.

22.2.5. 그 외의 정의들

그 외에 `init.h` 파일에서 찾아 볼 수 있는 것으로는 `CONFIG_HOTPLUG`가 있는 경우에 사용하는 macro들이다.

```
#ifdef CONFIG_HOTPLUG
#define __devinit
#define __devinitdata
#define __devexit
#define __devexitdata
#else
#define __devinit __init
#define __devinitdata __initdata
#define __devexit __exit
#define __devexitdata __exitdata
#endif
```

```

/* Functions marked as __devexit may be discarded at kernel link time, depending
   on config options. Newer versions of binutils detect references from
   retained sections to discarded sections and flag an error. Pointers to
   __devexit functions must use __devexit_p(function_name), the wrapper will
   insert either the function_name or NULL, depending on the config options.

*/
#ifndef MODULE || defined(CONFIG_HOTPLUG)
#define __devexit_p(x) x
#else
#define __devexit_p(x) NULL
#endif

```

코드 1347. HOTPLUG에 관련된 macro의 정의

이때도 CONFIG_HOTPLUG가 있는 경우에는 __devinit, __devinitdata, __devexit, __devexitdata등은 아무런 역할을 수행하지 않으며, CONFIG_HOTPLUG가 정의되지 않은 경우에만 각각이 __init, __initdata, __exit 및 __exitdata등으로 정의된다. 만약 MODULE이나 CONFIG_HOTPLUG가 정의되어 있지만 하다면, __devexit_p(x)는 그냥 x로 그럴지 않다면, NULL이라는 값을 가지도록 해준다. CONFIG_HOTPLUG가 영향을 주는 경우는 PCMCIA 및 Hotplug를 지원하는 몇몇 PCI chipset뿐이며, 여기서의 논의에는 벗어나기에 더 이상 다루지 않을 것이다.

이상에서 우린 init.h 파일이 어떤 역할을 수행하는지를 자세히 보았다. 즉, 우리가 작성한 코드가 커널에서 working하도록 만들어주기 위해서 어떤 일을 하는지를 보았으며, 모듈로 작성한 코드는 어떻게 초기화가 수행되는지도 보았다. 즉, __init와 같은 것을 사용한 경우에는 반드시 누군가가 초기화를 해주어야 하며, 이는 커널이 되던가 아니면 사용자가 커널의 일부를 수정해서 해당 함수를 호출해서 주어야 한다. 다만 module_init()/module_exit() 매크로를 사용해서 정의된 __init attribute를 가지는 함수의 경우에는 do_initcalls()라는 곳에서 일률적으로 전부 호출된다는 것을 보았다. 또한, kernel와 정적으로 link되지 않고, 모듈로 loading되는 함수들은 전부 init_module()과 cleanup_module()이라는 함수를 가지고 해당 초기화 및 해제 함수에 alias해서 사용하고 있다는 사실을 보았으며, 이들이 module의 entry point와 같은 역할을 수행한다는 것도 보았다.

23. 참고 문헌

- [1] USB Device Driver Programming guide, Detler Fligel, 2000.
- [2] USB Specification 1.1, <<http://www.usb.org>>
- [3] USB Class Specification, <<http://www.usb.org>>
- [5] Programming the Microsoft Windows Driver Model, Walter Oney, Microsoft Press, 1999.
- [6] Developing Windows NT Device Drivers : A programmer's handbook, Edward N. Dekker, Joseph M. Newcomer, Addison-Wesley, 1999.
- [7] Linux Device Drivers, Alessandro Rubini, O'Reilly & Associates, Inc. 1998.
- [8] Internetworking with TCP/IP Vol I: Principles, Protocols, and Architecture 3rd Edition, Douglas E. Comer, Prentice Hall, 1997.
- [9] Linux Device Driver 2nd Edition, Alessandro Rubini, O'reilly & Associates, Inc., 1998.
- [10] Developing Windows NT Device Drivers: A programmer's handbook, Edward N. Dekker, Joseph M. Newcomer, Addison Wesley, 1999.
- [11] Samsung Fast Ethernet Controller for Network interface card User's Manual ks8920, Samsung electronics, October 1998.
- [12] TCP/IP Illustrated Volume 1, 2, W. Richard Stevens, Addison Wesley, 1994.
- [13] Linux Kernel Internals, M Beck, H Böhme, M Dziadzka, U Kunitz, R Magnus, D Verworner, 2th edition, Addison Wesley, 1997.
- [14] Understanding the Linux Kernel, Daniel P. Bovet, Marco Cesati, O'reilly, Jan. 2001.
- [15] IA-32 Intel Architecture Software Developer's Manual Volume 3, System Programming Guide, Intel corp. 2001.
- [16] Linux hacker들을 위한 Unix Kernel 완전 분석으로 가는 길, 박장수, 1995.
- [17] Executable and Linkable Format (ELF), TIS Committe version 1.1
- [18] Unix Network Programming, Networking APIs : Sockets and XTI, W. Richard Stevens, Volume 1, 2nd Edition, Prentice Hall, 1998.
- [19] Unix Internal, a practical approach, Steve D Pate, Addison-wesley, 1996.
- [20] SCO MDI driver documentation.
- [21] Writing A Unix Device Driver, second edition, Janet I. Egan, Thomas J. Teixeira, John Wiley & Sons, Inc. 1992.
- [22] Plug And Play System Architecture, Tom Shanley, MindShare, Inc. Addison-wesley, 1996.
- [23] Writing Device Drivers for SCO Unix A Practical Approach, Peter Kettle and Steve Starler, Addison-wesley, 1993.
- [24] Solaris Internals, Core Kernel Architectures, Jim Mauro, Richard McDougall, Sun Microsystems Press, Prentice Hall, 2001
- [25] Intel StrongARM SA-1100 Microprocessor Developer's Manual, August 1999.
- [26] Linux IP Networking, Glenn Herrin, May 31, 2000.
- [27] Building Powerful Platforms with Windows CE, James Y. Wilson, Aspi Havewala, 2001.
- [28] ARM Architecture Reference Manual.
- [29] Linux PCMCIA Programmer's Guide, David Hinds, May 2001.
- [30] Unix Internal, The New Frontiers, Uresh Vahalia, Prentice Hall, 1996.
- [31] The Magic Garden Explained, The Internals of Unix® System V Release 4, Berny Goodheart & James Cox, Prentice Hall, 1994.
- [32] PCMCIA System Architecture : 16-bit PC Cards, Don Anderson, 2nd edition, MindShare, Inc. 1995.
- [33] CardBus System Architecture, Don Anderson/Toni Shanley, MindShare, Inc. 1996.
- [34] PC Card/PCMCIA Software Developer's Handbook, Steven M. Kipisz, Brian E. Moore, Dana L. Beatty, 2nd Edition, Peer-to-Peer 1999.
- [35] FSMLab RTLinux professional edition, 2001.
- [36] FireWire System Architeture : IEEE 1394a, Don Anderson, 2nd Edition, Mindshare, Inc. 1999.
- [37] Advanced Power Management(APM) BIOS Interface Specification, Revision 1.2, Intel, Microsoft, February 1996.

-마지막으로 알리고 싶은 짧은 이야기-

- 나는 오랜 시간 소설이란 것에 미쳐서 지낸 적이 있다. 그렇게 많은 책을 보지는 않았지만, 적어도 내면에 세계는 항상 그것을 갈구하고 그것을 읽도록 만들었다. 때론 쓰고 싶은 생각이 곁가지를 틀어, 드디어 원가를 내뱉지 않고는 못 견딜 만큼의 열정이 생기기를 기대했었다. 하지만, 아직도 부족한 어떤 것이 있기에 쓰지 못한다. 소설과도 같은 이런 문서만을 쓸 뿐...