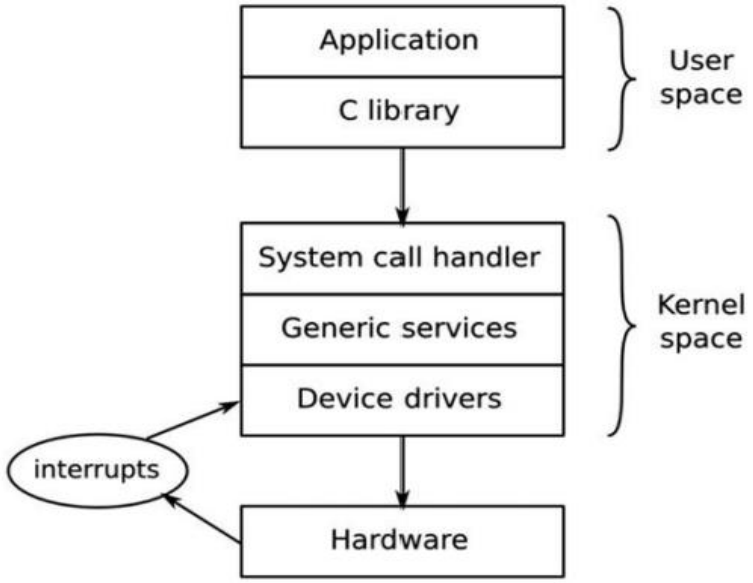
리눅스 커널

커널

- 역할

  자원 관리, 하드웨어와의 인터페이스, 사용자 공간 프로그램에게 유용한 수준의 추상화 API 제공



커널 선택

- 벤더 지원

  SoC나 보드의 벤더로부터 임베디드 리눅스 지원 여부 및 지원 수준 확인 필요

- 라이선스

– 리눅스 소스 코드 : GPL v2 (소스 공개)

– 사용자 공간에서 시스템 호출하는 프로그램은 이 라이선스 적용 안됨.

– 커널의 모듈에 관련된 소스 코드 부분은 논쟁이 되는 부분도 있으나 대체적으로 라이선스 적용이 안되고 있음. 확인 필요

리눅스 커널 개발주기

- 개발 주기 : 초창기 8~12주 단위로 신규 버전 발표, 현재는 속도가 늦어짐.
- 개발 커널 트리 : git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
- 안정 커널 트리 : git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
- 장기 지원 커널 : www.kernel.org
- 커널 버전별 개요 : https://kernelnewbies.org/LinuxVersions

커널의 종류

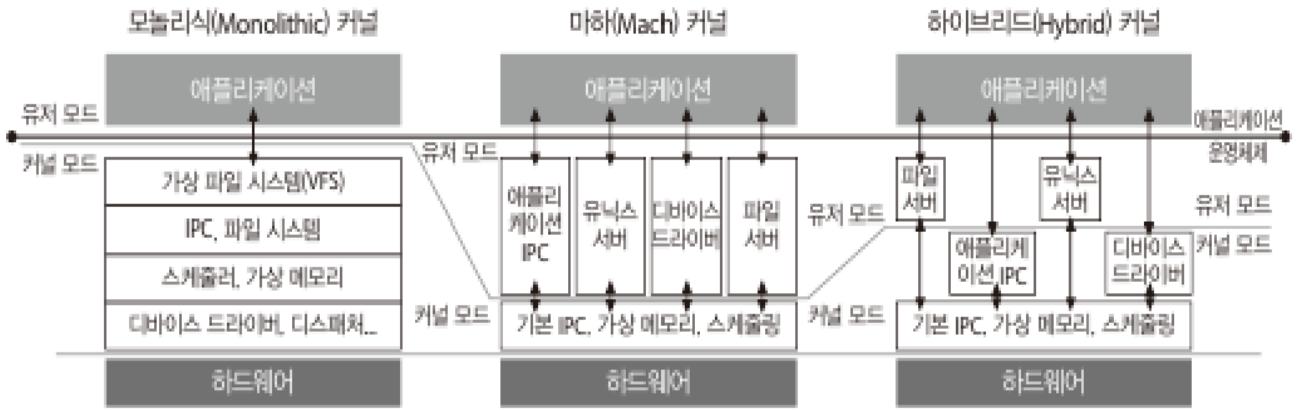


그림 12-2 모놀리식 커널과 마하 커널

- Monolithic 커널

커널이 사용자에게 제공하는 서비스를 모두 하나의 덩어리로 묶어 놓은 구조이다. 각 커널 계층이 커널 프로그램 하나로 통합되고, 현재의 프로세스를 대신하여 커널 모드에서 동작한다.

- Micro 커널

운영 체제의 핵심적인 기능만을 커널에 넣고, 나머지를 서비스 프로세스의 형태로 사용자 프로그램처럼 동작시키는 구조이다.
몇 가지 동기화 원시 함수와 간단한 스케줄러, 프로세스 간 통신 메커니즘을 포함하여 최소의 기능만을 커널에 요구한다.

커널의 종류

- 리눅스 커널

Monolithic 커널 구조를 가지고 있어 속도 면에서의 성능은 Micro로 커널에 비해 상대적으로 좋으나 업그레이드가 어렵다는 단점이 있다. 그러나, 리눅스는 커널 모듈이란 것을 지원함으로써 Micro 커널과 같은 확장성이나 모듈성을 증가시켰다. 모듈은 실행 중에 커널로 링크할 수 있는 객체 파일을 의미한다.
리눅스에서는 디바이스 드라이버를 비롯한 여러 기능들을 모듈로 만들어 관리하고 있다.

모듈 위치 : /lib/modules/〈kernel_version〉/kernel/…/*****.ko

커널 영역과 사용자 영역

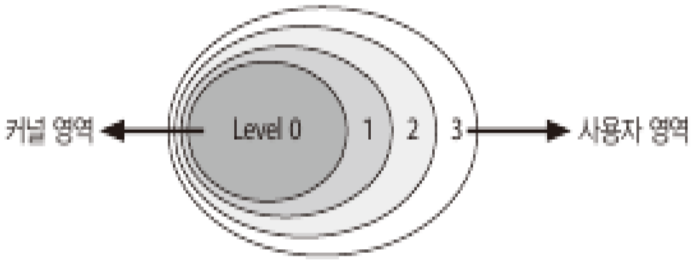


그림 12-3 실행 레벨

- 사용자 영역
  ls, ps, mount와 같은 리눅스의 일반적인 명령어들과 컴파일된 프로그램 등이 수행되는 모드이다.

- 커널 영역
  커널은 디바이스 드라이버와 인터럽트 처리 메커니즘 등 커널 함수를 이용하여 하드웨어와 통신하고 시스템 호출 인터페이스를 이용하여 사용자 수준 응용프로그램과 통신한다. 커널은 특권 레벨로 실행되므로 프로그래밍시 주의를 요한다.

- 시스템 호출(system call) : 커널이 사용자에게 제공하는 서비스 인터페이스
  프로세스 관리자가 제공하는 서비스와 관련된 시스템 호출은 fork(), execve(), getpid(), signal()이다.
  파일 시스템이 제공하는 서비스와 관련된 시스템 호출은 open(), read(), write()
  메모리 관리자가 제공하는 시스템 호출은 brk()이다.
  네트워크 관리 부분과 관련된 시스템 호출에는 socket(), bind(), connect() 등이 있다.

커널 영역과 사용자 영역

- 커널은 가상주소를 이용하여 자원에 접근
- 실제 하드웨어 디바이스로 접근을 하려면 물리주소를 가상 주소로(커널 영역) 매핑하여 사용

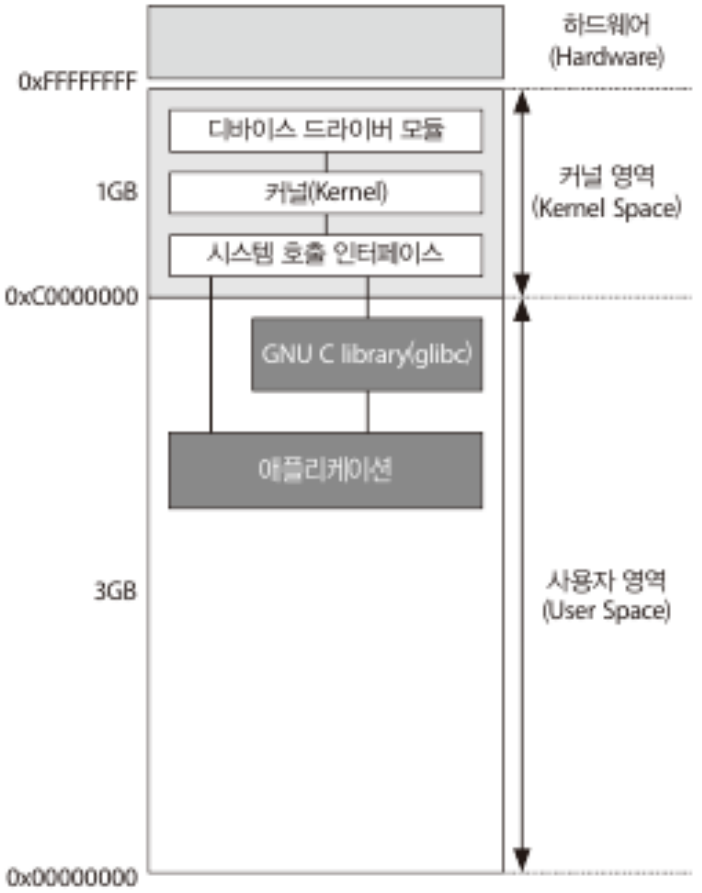


그림 12-4 커널 영역과 사용자 영역

https://www.kernel.org/doc/html/v6.6/arch/arm64/memory.html

리눅스 디바이스 드라이버

- 디바이스 드라이버 : 하드웨어를 제어하기 위한 하드웨어 특성이나 레지스터 설정등 디바이스 제어를 위한 기능을 구현해 놓은 소프트웨어, 커널안에 구현
- 디바이스를 추상화하여 어플리케이션에서 시스템 콜을 이용하여 호출

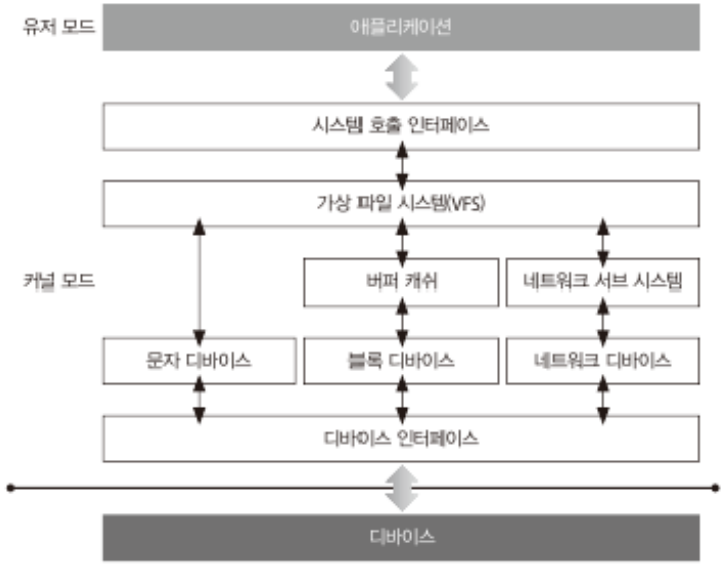


그림 12-5 리눅스 커널과 디바이스 드라이버

커널 소스 다운로드

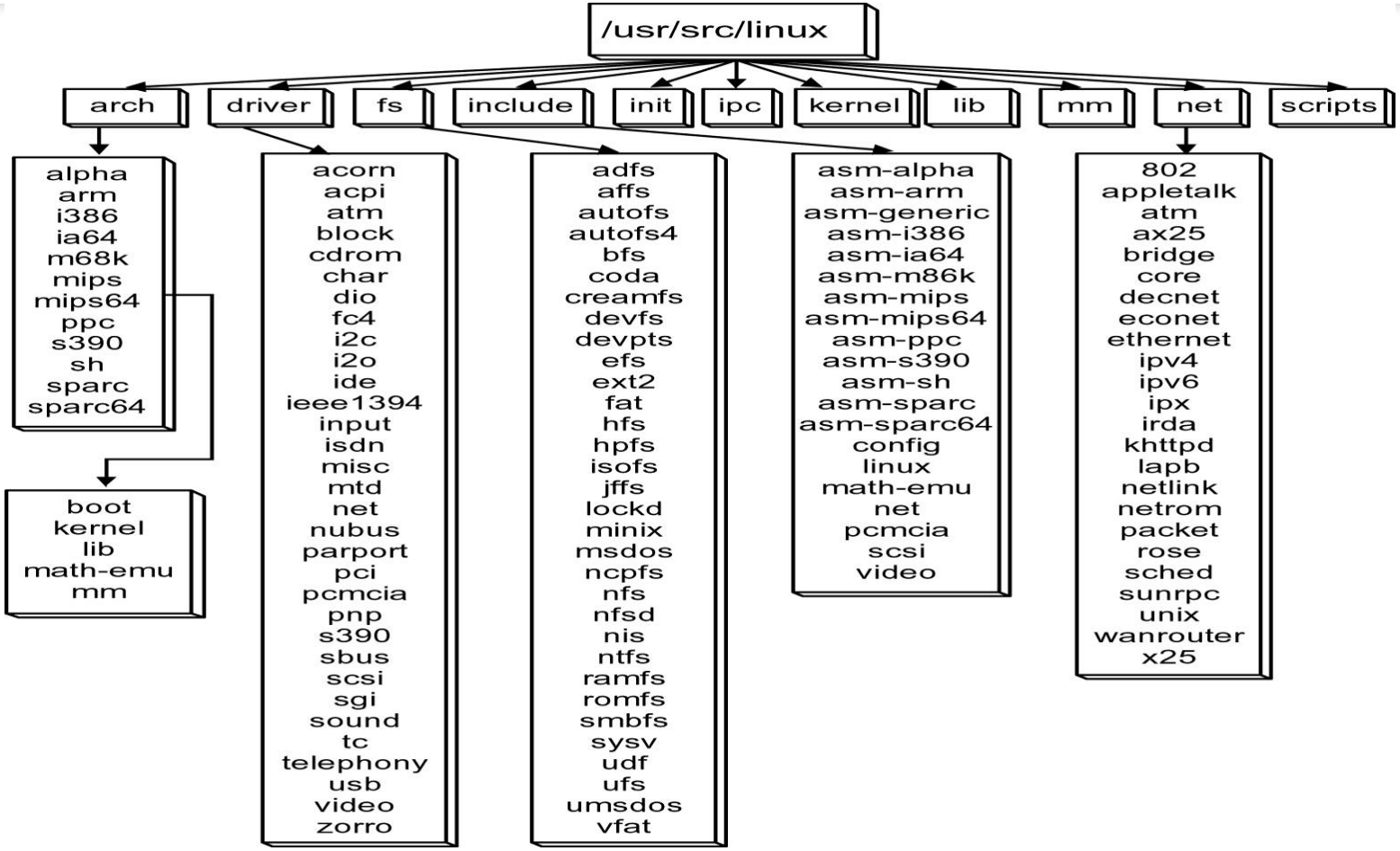
- 라즈베리파이 커널 소스 다운로드

```
$ git clone --depth=1 --branch rpi-6.6.y  https://github.com/raspberrypi/linux
$ sudo mv linux /usr/src
```

- 커널 빌드에 필요한 패키지 설치

```
$ sudo apt install bc bison flex libssl-dev make libc6-dev libncurses5-dev
```

커널 소스 디렉토리


/usr/src/linux
arch
driver
fs
include
init
ipc
kernel
lib
mm
net
scripts
alpha
arm
i386
ia64
m68k
mips
mips64
ppc
s390
sh
sparc
sparc64
boot
kernel
lib
math-emu
mm
acorn
acpi
atm
block
cdrom
char
dio
fc4
i2c
i2o
ide
ieee1394
input
isdn
misc
mtd
net
nubus
parport
pci
pcmcia
pnp
s390
sbus
scsi
sgi
sound
tc
telephony
usb
video
zorro
adfs
affs
autofs
autofs4
bfs
coda
creamfs
devfs
devpts
efs
ext2
fat
hfs
hpfs
isofs
jffs
lockd
minix
msdos
ncpfs
nfs
nfsd
nis
ntfs
ramfs
romfs
smbfs
sysv
udf
ufs
umsdos
vfat
asm-alpha
asm-arm
asm-generic
asm-i386
asm-ia64
asm-m86k
asm-mips
asm-mips64
asm-ppc
asm-s390
asm-sh
asm-sparc
asm-sparc64
config
linux
math-emu
net
pcmcia
scsi
video
802
appletalk
atm
ax25
bridge
core
decnet
econet
ethernet
ipv4
ipv6
ipx
irda
khttpd
lapb
netlink
netrom
packet
rose
sched
sunrpc
unix
wanrouter
x25

커널 소스 디렉토리

directory	설명
kernel	프로세스의 생성과 소멸, 프로그램의 실행, 스케줄링, 시그널 처리 등의 기능이 이 디렉터리에 구현되어 있다. 한편 문맥 교환(context switch)과 같은 하드웨어 종속적인 프로세스 관리 부분은 arch/arm/kernel 디렉터리에 구현되어 있다.
arch	하드웨어 종속적인 부분들이 구현된 디렉터리로 CPU의 종류에 따라 하위 디렉터리로 다시 구분된다. ARM CPU를 구현한 arch/arm 디렉터리는 다음과 같은 몇 개의 하위 디렉터리로 구분된다. arch/arm/boot – 시스템 초기화 때 사용되는 부트 스트랩 코드가 구현되어 있으며, 리눅스 커널인 vmlinuz 파일이 생성되는 위치이다. arch/arm/kernel – 프로세스 관리자 중에서 문맥 교환이나 스레드 관리 같은 하드웨어 종속적인 부분이 구현된다. arch/arm/mm – 메모리 관리자 중 페이지 부재 결함 처리 같은 하드웨어 종속적인 부분이 구현된다. arch/arm/lib – 커널이 사용하는 라이브러리 함수가 구현된다. arch/arm/ math-emu – FPU(Floating Point Unit)에 대한 에뮬레이터가 구현된다.

커널 소스 디렉토리

directory	설명
fs	리눅스에서 지원하는 다양한 파일 시스템들과 open(), read(), write() 등의 시스템 호출이 구현된 디렉터리로 ext2, nfs, ufs, msdos 등 다양한 파일 시스템들을 사용자가 일관된 인터페이스로 접근할 수 있도록 하기 위해서 리눅스에는 시스템 호출과 각 파일 시스템 간에 추상화된 VFS(Virtual File System)를 구현해 놓았다
mm	메모리 관리자가 구현된 디렉터리이다. 가상 메모리, 프로세스마다 할당되는 메모리 객체(mm_struct, vm_arem_struct, pgd) 관리, 커널 메모리 할당자(kernel memory allocation) 등의 기능이 구현되어 있다.
driver	리눅스에서 지원하는 디바이스 드라이버가 구현된 디렉터리이다. 버퍼 캐시를 통해 접근하며 임의 접근(random access)이 가능한 블록 디바이스 드라이버는 driver/block이라는 이름의 하위 디렉터리에 구현되어 있다. 반면에 순차적으로 접근되는 문자 디바이스 드라이버는 driver/char 하위 디렉터리에 구현되어 있으며, 네트워크 카드를 위한 드라이버는 driver/net에 있다. 그 외에 cdrom, sound, video 카드를 위한 드라이버는 각각 driver/cdrom, driver/sound, driver/video라는 이름의 하위 디렉터리에 구현되어 있다

커널 소스 디렉토리

directory	설명
net	통신 프로토콜이 구현된 디렉터리이다. 대표적인 통신 프로토콜인 TCP/IP뿐만 아니라 UNIX 도메인 통신 프로토콜, WAN 통신 프로토콜인 X.25 등이 구현되어 있다.
ipc	커널이 지원하는 프로세스 간 통신(Inter Process Communication) 기능이 구현된 디렉터리이다. 대표적인 프로세스 간 통신에는 파이프, 시그널, SVR4 IPC, 소켓 등이 있으며, 이 디렉터리에는 SVR4 IPC인 메시지 패싱, 공유 메모리, 세마포어가 구현되어 있다.
init	커널 초기화 부분, 즉 커널의 메인 시작 함수가 구현된 디렉터리이다. 커널의 시작은 하드웨어 종속적인 초기화를 수행한 후(arch/arm/ kernel/head.S) 커널의 메인 시작 함수를 호출하는데 init/main.c에 구현된 start_kernel() 함수가 바로 그 것이다. 이 함수는 디바이스 드라이버 초기화, 커널 내부 자료 구조 할당 및 초기화, 프로세스 0(idle 프로세스)과 프로세스 1(init, 초기화 프로세스) 생성 등을 수행한다. 이후 init 프로세스가 초기화를 담당하며 각종 데몬의 생성, 파일 시스템 마운트 및 초기화, 터미널 초기화, 네트워크 초기화, 로그인 프로세스 생성 등의 작업을 수행한다.

리눅스 커널의 개요

- 프로세스 관리 – 프로세스와 스레드

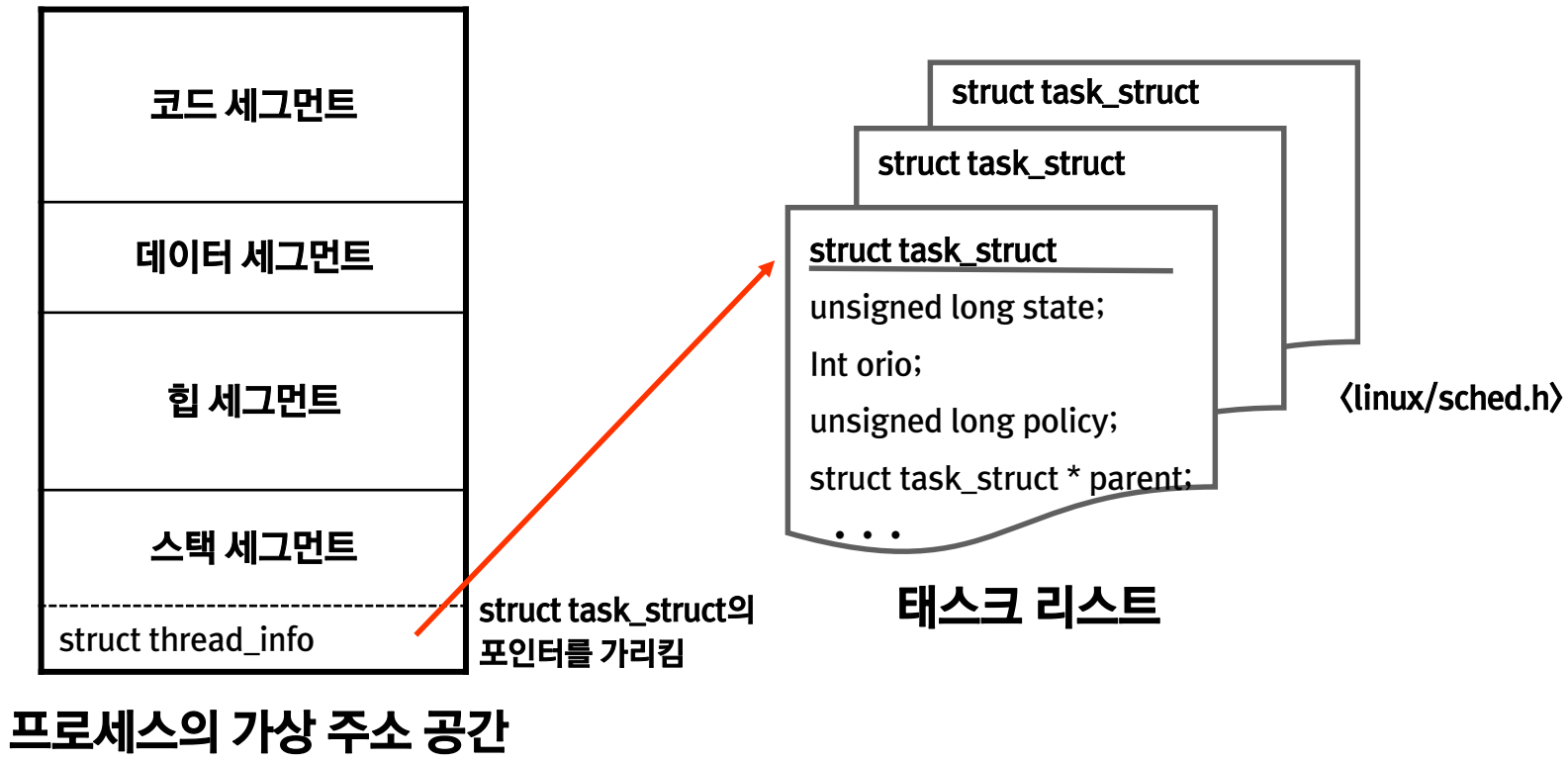
  프로세스 – ‘실행 중인 프로그램’
  독자적인 가상 메모리를 가지고 있는 태스크
  자원(CPU, 메모리, I/O)을 독점하여 사용하는 것으로 생각하며 수행

  스레드 – 프로세스 내에 활동을 가진 객체
  스케줄링의 최소 단위
  하나의 가상 메모리를 여러 태스크들이 공유

- 리눅스에서는 프로세스와 스레드가 같으나 다만 자원의 공유 방식만 차이가 있다.

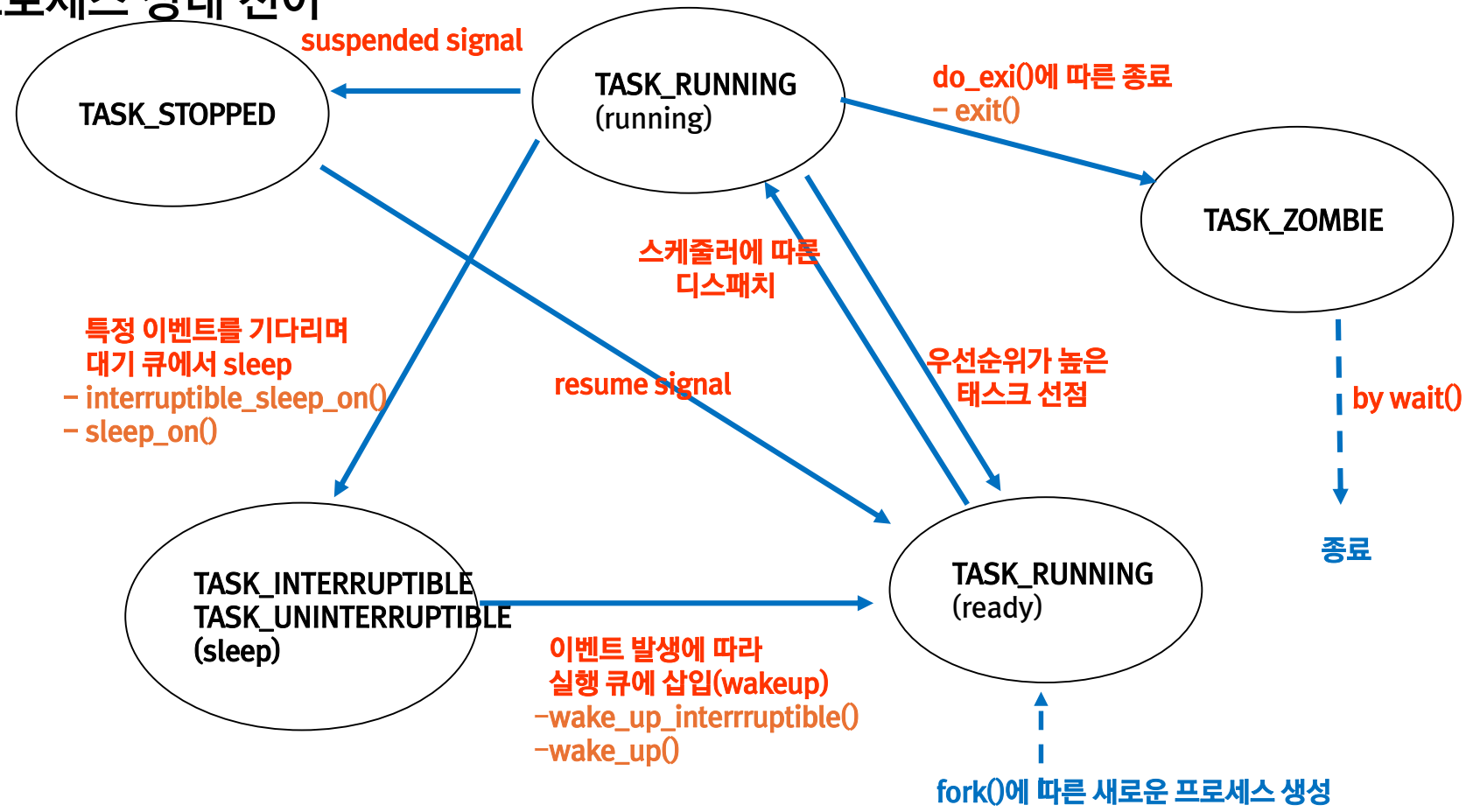
리눅스 커널의 개요

- 프로세스 관리 – 프로세스 기술자와 태스크 구조체



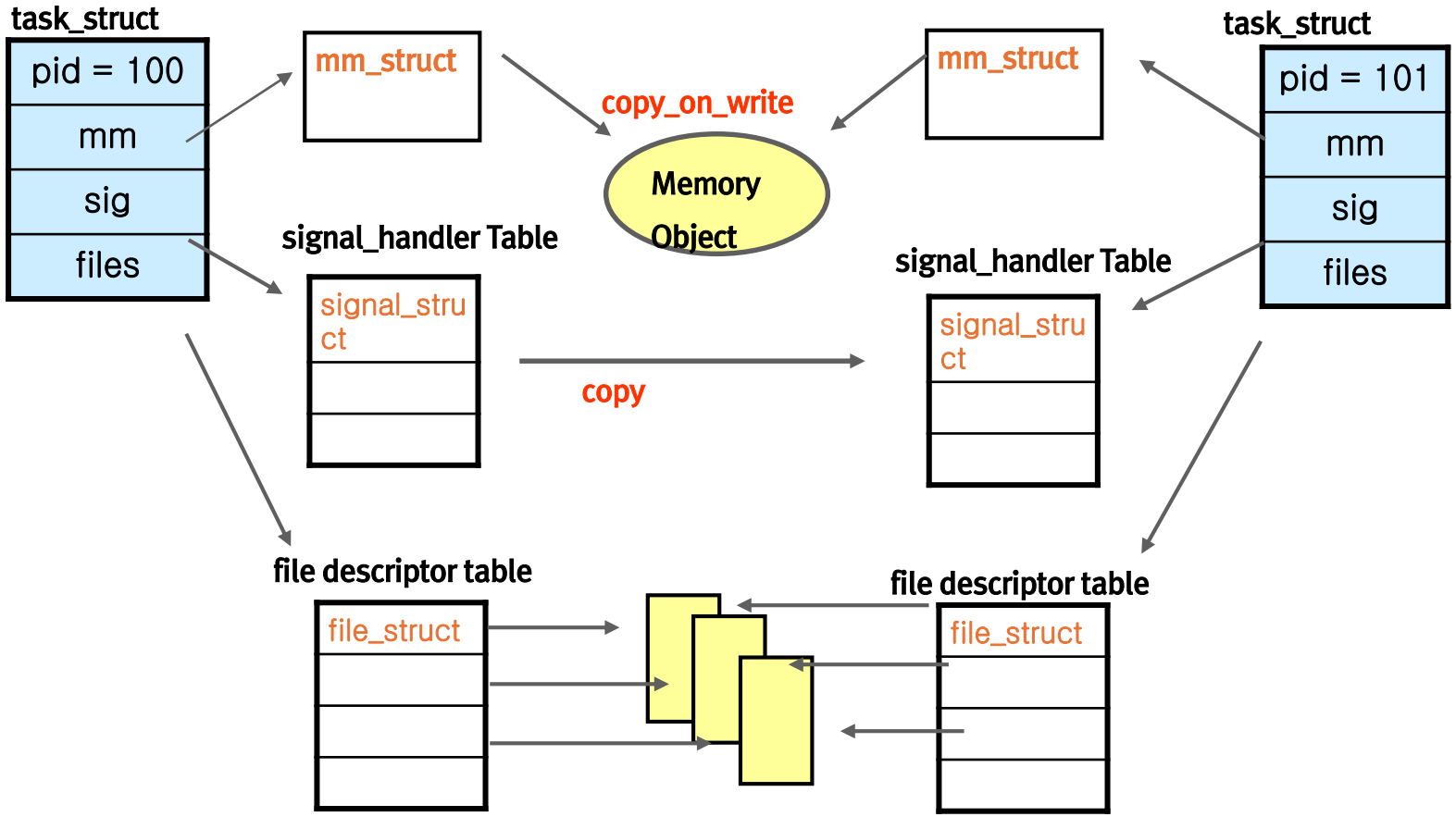
리눅스 커널의 개요

- 프로세스 관리 – 프로세스 상태 천이



리눅스 커널의 개요

- 프로세스 관리 – fork() 시스템 호출에 따른 자식 프로세스 생성



리눅스 커널의 개요

- **프로세스 관리 – 스케줄링**
- 멀티태스킹 운영 체제의 기본 요소
- 수행 가능(runnable) 프로세스 중에서 다음 실행할 프로세스 선택
- 리눅스 스케줄링 기법

  I/O 중심의 프로세스 – 높은 우선순위.
  프로세서 중심의 프로세스 – 낮은 우선순위.

- 스케줄링 정책에 따라 선택

  SCHED_RR – 정적 우선순위의 실시간 프로세스, 가장 우선순위가 높은 프로세스 선택, 선점이 불가능하다.
  SCHED_FIFO – 동적 우선순위의 실시간 프로세스이다.
  SCHED_OTHER – 일반적인 time sharing 프로세스이다. 실행 시간이 길수록 우선순위가 낮아진다.

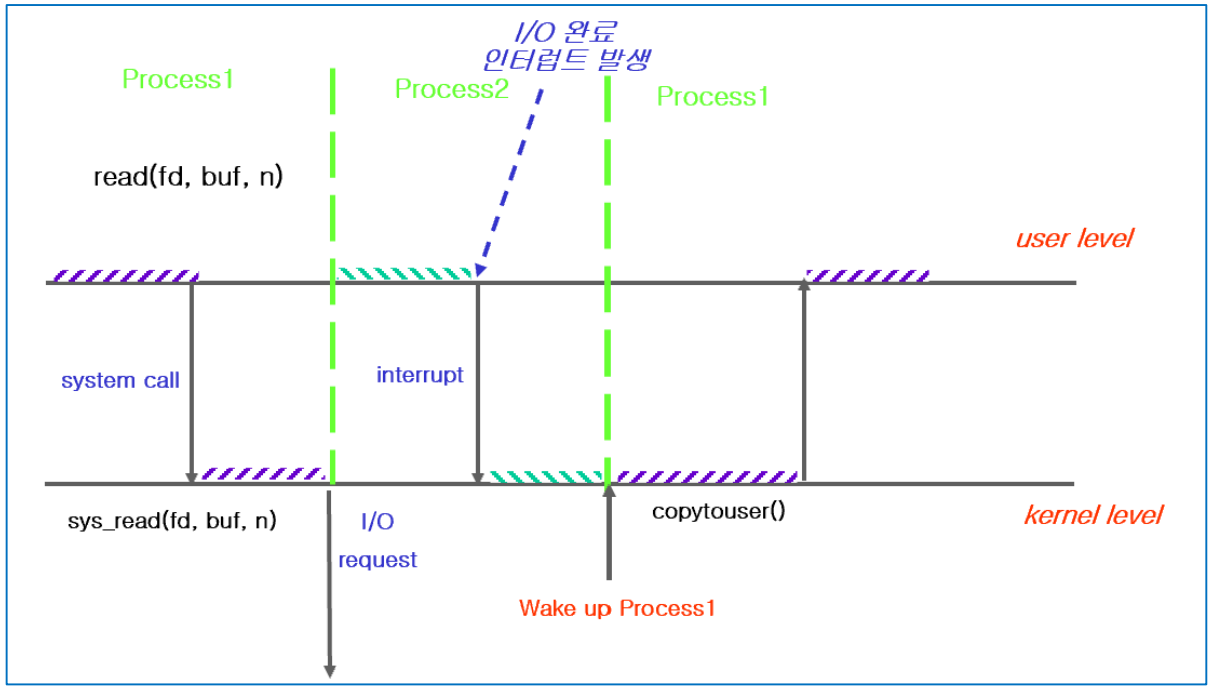
- 문맥 교환(Context Switching)

리눅스 커널의 개요

- 프로세스 스케줄링

schedule() 함수가 호출되는 경우
- 수행 중인 프로세스에 할당된 타임 퀀텀을 다 소비하는 경우 호출된다.
- 수행 중인 프로세스가 블록 상태로 변화될 때 직접 스케줄러를 호출한다.
- 더 높은 우선순위의 프로세스가 wakeup 되는 경우 호출한다.
- 시스템 호출 종료 및 인터럽트 처리 종료시 호출한다.

schedule() 호출 예



current->state = TASK_INTERRUPTIBLE;

schedule();

current 매크로 : 현재 수행되고 있는 프로세스에 대한 포인터

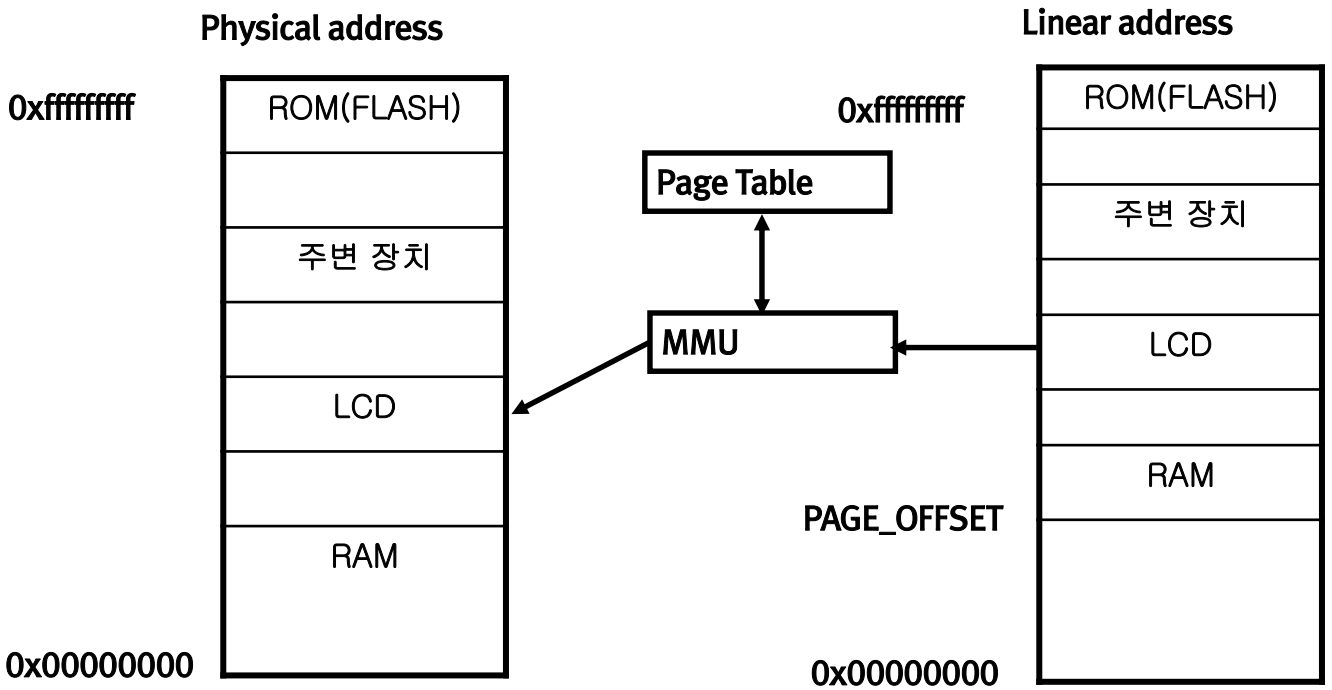
리눅스 커널의 개요

- **메모리 관리**
  - 페이지
    메모리 관리의 기본 단위, MMU 처리 단위
    32비트 아키텍처 – 4KB
    64비트 아키텍처 – 8KB
    페이지 구조체 struct page 선언 – 〈linux/mm.h〉

  - 리눅스에서 주소 타입
    User Virtual Address : 일반적으로 사용자 프로그램에서 사용하는 주소
    Physical Address : CPU와 시스템의 물리적 메모리가 사용하는 주소
    Linear Address : 커널 내부 함수 kmalloc()에 따라 할당되는 메모리의 주소

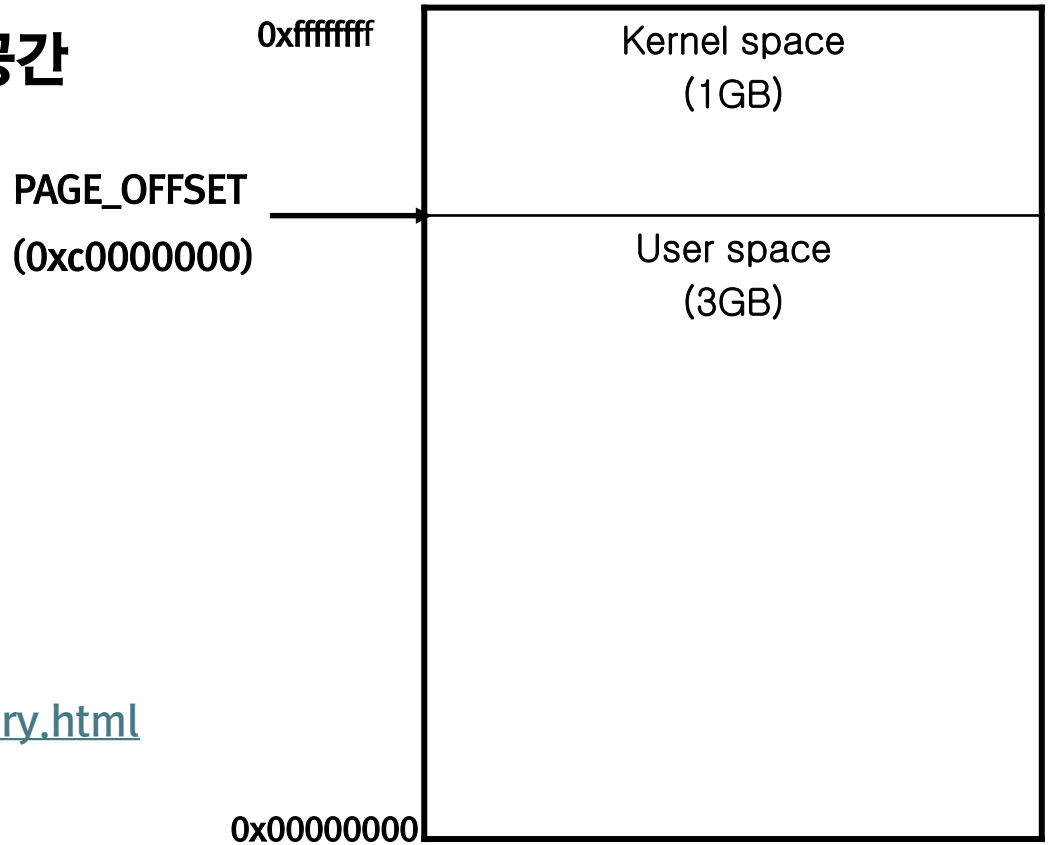
리눅스 커널의 개요

- 메모리 관리 – Physical Address와 Linear Address



리눅스 커널의 개요

- **메모리 관리 – 커널 주소 공간과 사용자 주소 공간**



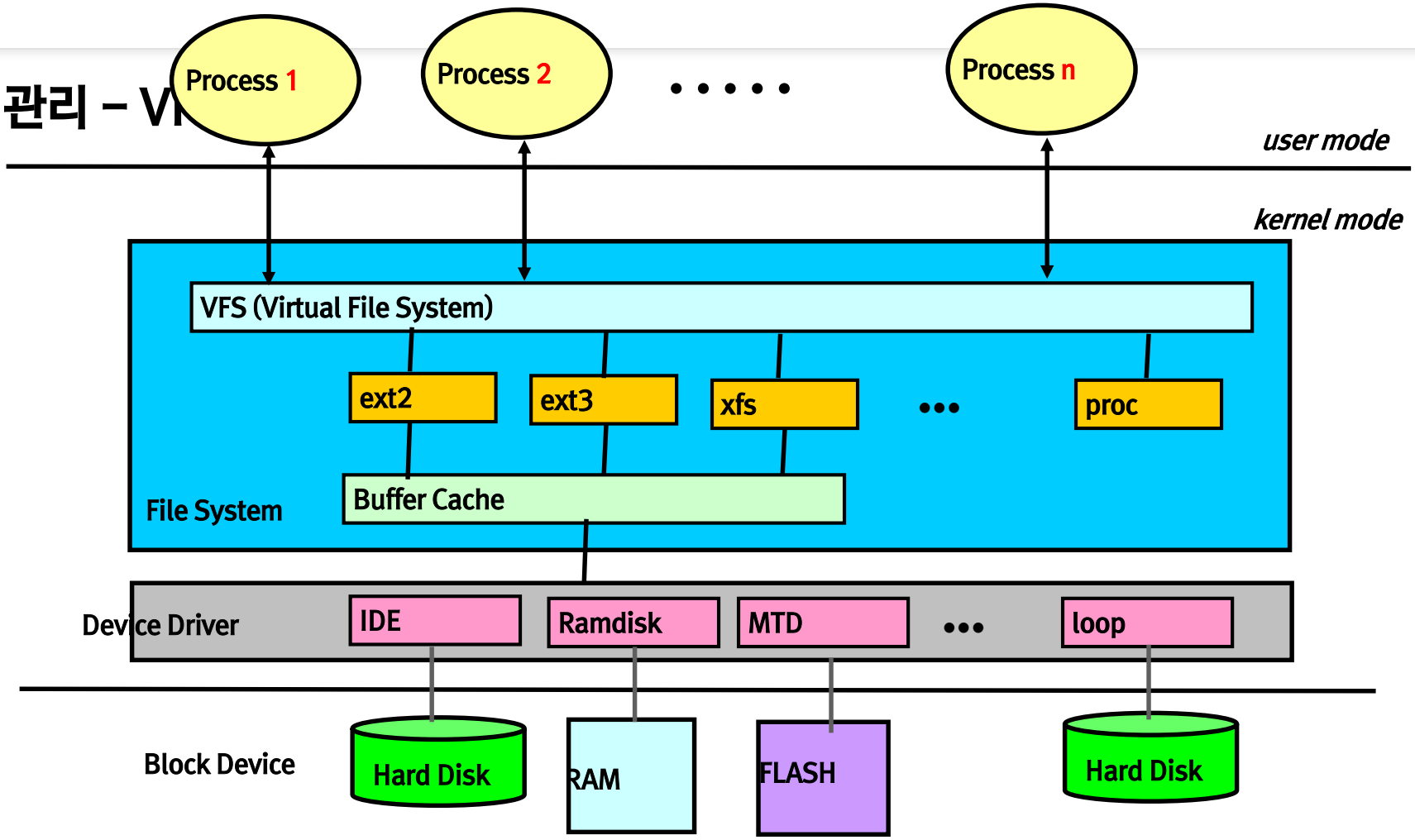
64비트의 경우:
https://www.kernel.org/doc/html/v6.6/arch/arm64/memory.html

리눅스 커널의 개요

- **리눅스 파일 시스템 관리**
  - 리눅스에서는 모든 것을 파일 개념으로 취급
  - 다양한 파일 시스템 지원(Virtual File System)
  - 주요 오브젝트 : 슈퍼 블록 객체
    inode 객체
    파일 객체
    dentry 객체
  - Block I/O를 위한 버퍼 캐시를 이용
  - 파일 시스템 종류 : ext2/ext3/ext4
    cramfs
    xfs
    jffs2

리눅스 커널의 개요

- 리눅스 파일 시스템 관리 – VF



리눅스 커널의 개요

- **리눅스 디바이스 및 네트워크 관리**

– 디바이스 관리

커널에서 물리적 장치 제어

디바이스 드라이버에서 장치의 각 동작을 기술

입 · 출력 요청 스케줄링

주변 장치와 커널 간의 메모리 전송 – CPU, DMA

인터럽트 요청 및 처리

– 네트워크 관리

네트워크 서브 시스템과 디바이스 드라이버로 구성

통신 프로토콜을 구현 – TCP/IP

라우팅 및 Address Resolution Method

네트워크 드라이버를 관리

리눅스 커널의 개요

- 커널과 디바이스의 통신
  - 커널은 사용자와 하드웨어 사이에 존재하며, 하드웨어를 관리하고 이에 대한 서비스를 사용자에게 제공한다. 그리고 디바이스 드라이버와 인터럽트 처리 메커니즘 등을 이용하여 하드웨어와 통신한다. 뿐만 아니라 시스템 호출 인터페이스를 이용하여 사용자 수준 응용과 통신한다.

  - 디바이스 드라이버가 제공하는 시스템 호출은 없다.
  - 리눅스 커널에서 디바이스 드라이버는 사용자 수준 응용을 이용해 직접 호출되지 않는다. 그 대신 파일 시스템을 거쳐 디바이스 드라이버의 서비스가 호출된다

커널 빌드

그림 12-6 리눅스 커널의 빌드 순서

표 12-1 커널의 빌드 옵션[10]

옵션	내용	비고
mrproper	커널의 설정을 초기화한다.	
menuconfig	커널을 설정할 수 있는 메뉴를 띄운다.	xconfig, gconfig, nconfig
dep	커널을 컴파일하기 전에 의존관계(dependency)를 검사한다.	config 파일을 저장하면 자동으로 수행된다.
bzImage†	압축된 커널 이미지를 생성한다(빌드된 이미지는 커널 소스 /usr/src/linux/arch/i386/boot에 위치).	zImage
modules	커널의 모듈들을 빌드한다.	
modules_install	빌드된 커널 모듈을 설치한다(/lib/modules에 설치).	
depmod	모듈 간의 의존관계를 검사해서 /lib/modules/modules.dep 파일을 생성한다.	모듈을 생성하면 자동으로 수행된다.
clean	컴파일 과정에서 생성되었던 중간 파일들을 삭제한다.	

† 속도가 너무 느린 플로피 디스크에서 부팅(로딩) 속도를 빠르게 하기 위해 압축된 이미지를 사용하였다.

커널 빌드

- 라즈베리파이 커널 빌드 (https://www.raspberrypi.com/documentation/computers/linux_kernel.html)

- 커널 설정(Kconfig) – 64bit config

```
$ cd linux
$ KERNEL=kernel#
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm271#_defconfig
```

- 커널 빌드

```
make -j4 ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image modules dtbs
```

커널 설치 (on SD)

- SD 카드 연결

```
# lsblk 명령으로 파티션 확인
mkdir -p mnt/fat32 mnt/ext4
sudo mount /dev/sdb1 mnt/fat32
sudo mount /dev/sdb2 mnt/ext4
```

- SD 카드에 새로 빌드 된 모듈 설치

```
sudo env PATH=$PATH make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- \
  INSTALL_MOD_PATH=mnt/ext4  modules_install
```

커널 설치 (on SD)

- 이전 커널 백업 및 새로운 커널과 장치 트리 blob 복사

```
sudo cp mnt/fat32/$KERNEL.img mnt/fat32/$KERNEL-backup.img
sudo cp arch/arm64/boot/Image mnt/fat32/$KERNEL.img
sudo cp arch/arm64/boot/dts/broadcom/*.dtb mnt/fat32/
sudo cp arch/arm64/boot/dts/overlays/*.dtb* mnt/fat32/overlays/
sudo cp arch/arm64/boot/dts/overlays/README mnt/fat32/overlays/
sudo umount mnt/fat32
sudo umount mnt/ext4
```

주요 커널 함수들

표 12-4 주요 커널 함수들[20]

함수	내용	헤더 파일
printk( )	콘솔에 로그를 출력한다.	<linux/kernel.h>
	int printk(const char *fmt, ...)	
kmalloc( )	메모리를 할당한다.	<linux/malloc.h>
	void * kmalloc(size_t size, int priority);	
kfree( )	할당된 메모리를 해제한다.	
	void kfree(void *__ptr);	
get_user( )	사용자 영역에서 커널 영역으로 데이터를 가져온다.	<asm/uaccess.h>
	err = get_user(x, ptr);	
put_user( )	커널 영역에서 사용자 영역으로 데이터를 보내준다.	
	err = put_user(x, ptr);	
copy_from_user( )	사용자 영역에서 커널 영역으로 메모리 블록을 복사한다.	
	bytes = copy_from_user(void *to, const void *from, unsigned long n);	
copy_to_user( )	커널 영역에서 사용자 영역으로 메모리 블록을 복사해준다.	
	bytes = copy_to_user(void *to, const void *from, unsigned long n);	
access_ok( )†	사용자 영역을 접근할 수 있는지 확인한다.	<asm/uaccess.h>
	access_ok(addr, size);	
cli( )	인터럽트를 지운다(비활성화한다).	<asm/system.h>
	extern void cli( );	
sti( )	인터럽트를 설정한다(활성화한다).	
	extern void sti( );	

커널 모듈

- 모놀로딕 커널의 단점
- 새로운 기능을 추가하거나 기존 기능 삭제시 시스템 재기동 필요

- 리눅스 커널 v2 에서 모듈 방식으로 개선, 시스템 재부팅 없이 기능 추가 및 삭제 가능

- 대표적인 모듈 사용 예 : 디바이스 드라이버

커널 모듈

- 모듈의 동적 사용

표 12-5 리눅스 커널 모듈 관련 명령어(mod-utils)

명령어	내용
lsmod	현재 커널에 올라와 있는 커널 모듈의 리스트를 표시한다.
insmod	해당 커널 모듈들을 올린다(load).
rmmod	해당 커널 모듈들을 내린다(unload).
modprobe	해당 커널 모듈들을 올릴 때 필요한 의존관계가 있는 모든 모듈들을 올린다.
depmod	modprobe 명령어를 위해 모듈들의 의존관계 리스트를 출력한다.
modinfo	현재 모듈에 대한 정보를 출력한다.

모듈 프로그래밍

- module_init(함수명) : 모듈 초기화 함수
- module_exit(함수명) : 모듈 해제 함수

```
#include <linux/module.h>

/* Each module must use one module_init(). */
#define module_init(initfn)                                     \
        static inline initcall_t __maybe_unused __inittest(void)                \
        { return initfn; }                                      \
        int init_module(void) __attribute__((alias(#initfn)));

/* This is only required if you want to be unloadable. */
#define module_exit(exitfn)                                     \
        static inline exitcall_t __maybe_unused __exittest(void)                \
        { return exitfn; }                                      \
        void cleanup_module(void) __attribute__((alias(#exitfn)));
```

모듈 프로그래밍

- Makefile

```
obj-m := hello_module.o
KDIR := /usr/src/linux

default:
	make ARCH=arm CROSS_COMPILE=aarch64-linux-gnu- -C$(KDIR) M=$(shell pwd) modules

clean:
	make -C$(KDIR) M=$(shell pwd) clean
```

모듈 프로그래밍

- on 라즈베리파이

```
$ modinfo hello_module.ko
  . . .
$ journalctl –f
```

```
$ sudo insmod hello_module.ko

$ lsmod | grep hello

$ sudo rmmod hello_module
```

디바이스 드라이버

- 교재 12.3 시스템 레지스터와 LED 출력
- 교재 12.4 GPIO 드라이버 프로그래밍