TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

2018-04-06 (32 회차)

강사: Innova Lee(이상훈)
gcccompil3r@gmail.com
학생: **정유경**
ucong@naver.com

과제 1. 운영체제 동작비유

[비유 1]
*. 펌웨어레벨의 프로그래머: OS 가 없는 상태에서 구동되는 SW 를 프로그래밍
*. 프로그램의 흐름을 계획하며 모든것을 펌웨어 레벨에서 코딩: OS 없이 타이밍값을 계산하여 스케쥴링 → context switching 하기 위해 메모리를 프로그래머가 관리한 것
*. 스케줄링과 파일시스템, 메모리관리기능을 가지고 있는 zImage: 리눅스 커널의 압축된 이미지
zImage 에 포함된 것은 파일시스템, 부트로더 ,시스템콜 관리 즉, 운영체제가 관리해야 하는 5 대 요소들(파일시스템, 메모리, 네트워크, 디바이스, 프로세스-태스크)
*. Bootloader: 하드웨어 초기화에 필수(Floating 상태와 연산되는 경우 없도록), 가상메모리 레이아웃을 잡는다
*. 시스템다지인에만 충실: HW 를 디자인한다(회로설계나 PCB) ← why: 디바이스 드라이버 덕분(by OS 전문가) 결국 시스템 프로그래머는 HW 나 스케줄링에 신경을 쓰지 않아도 되므로 작업이 단순해진다.
*. 데몬은 누가 일을 시키지 않더라도 해야할 일이 생기면 Running: 시그널이 들어오면 강제로 깨워서 Wait Queue 에서 RUN Queue 로 이동한다,
*. 각각의 태스크가 필요로 할 때마다 메모리를 할당/해제: 페이징

[비유 2]
*. 어떤 운영체제는 FIFO: 절대로 RTOS 는 아니다.
*. 어떤 운영체제는 RM 정책: 우선순위 높은 녀석에게 비율을 많이 할당한다. → RTOS 에 적합
*. 리눅스는 24 개의 태스크를 우선순위대로 정렬한 뒤, 정해진 time slice 만큼씩만 CPU 할당: 범용적인 상황에서는 하이브리드 방식이 쓰인다. (RTOS 는 우선순위가 고정되어 있어 변하지 않는다. 리눅스는 정적, 동적 우선순위 모두 갖는다.)
*. 단하나의 태스크만 실제 Running 상태이고 나머지는 대기중: Single Core CPU 일 경우
멀티코어 CPU 는 프로세스 여러개 실행 가능
(RTOS 는 프로세스 보통 16~32 개)
*. time slice 를 100ms 로 고정: 10 개가 대기중일 경우 RTOS 시스템이 1 초가 밀려서 미사일이 3.4Km 날아오게된다.
*. 리눅스는 태스크의 속성을 고려하여 time slice 결정: 우선순위를 고려하여 time slice 를 결정한다,
*. 우선순위가 높은 태스크의 선점기능을 지원: time slice 가 완료되지 않았지만 실행중인 태스크를 변경한다. (시스템콜은 커널영역이므로 유저 영역보다 무조건 우선순위가 높다, 태스크가 변경되면 완전히 갈아 치워지는 것으로서 스케쥴링을 다시 하게된다. 인터럽트의 경우 실행이 중단된 시점에서 다시 복원된다)
*. 동시에 여러개의 태스크를 하나의 CPU 에서 수행, Context switching 시 현재 태스크가 어디까지 수행되었는지를 꼼꼼하게 기록→ 결국 멀티태스킹 구현한 것

[비유 3]
*. A 태스크가 B 태스크에 전달해줄 사건이 발생했다: 프로세스간 데이터 전달은 IPC or SIGNAL 사용
*. 같은 시그널이 또 배달되지 않도록 블록 시켜놓기로: sigemptyset(mark)하여 시그널블록을 사용자가 직접 셋팅한다 (해당 번호를 막을 수 있다) 시그널 처리 중에 또 다른 시그널이 들어가면 각각의 시그널 핸들러가 실행된다. 그러나 같은 시그널이 입력되면 블록되어 처리되지 않는다.
*. 대량의 정보를 주고받을 수 없다는 단점: 시그널 번호만 주고받을 수 있다.
(대량의 정보는 뮤텍스, 세마포어, 스핀락, 메세지큐, 파이프를 통해서 주고받을 수 있음)
*. 소켓: 유일한 원격 IPC 통신, 동기화, local 이 아닌 network 상에서 운영되는 소켓이 가장 느리다.

[비유 4]
*. TCP/IP 스택은 항상 랜카드를 살펴보고 있다가 패킷이 전송되어 오면 커널에 보고하였다. 응용프로그램에 의해 요청을 받은 커널이 다른 컴퓨터와 통신을 위한 연결 요청을 지시하면:: 서버와 동기화 하는 방법 즉, 3 Way Handshaking
1) TCP/IP 스택이 SYN 전송” 너 있어?”
2) 상대방이 ACK 전송하여 수신 ” 나 있어”
3) ACK 전송 “있구나”

4) 소켓을 응용프로그램에 연결하여 통신 활성화!
5) 연결종료시 FIN 전송 “종료할게”
6) 상대방이 FIN+ACK 전송하여 수신 “응 종료하자”(결국 최종적으로 ACK 수신)

[비유 5]
*. 커널은 파일시스템에게 A.txt 라는 파일을 읽어오라고 명령을 내렸다: inode 테이블를 통해 디스크블록이 어딘지 알고 디스크블록을 메모리로 복사할 수 있다.
*. 파일시스템에서 찾기 시작했다: 파일 시스템 최상위에 root(/)가 있으므로, root 를 먼저 찾은 후(task_struct > files_struct > file > path > inode > dentry> super block 보면 루트 디렉토리의 위치가 나온다)
*. 평소 파일을 Ext2 형식에 맞게 저장: file_operations 구조체에 read, write, open, close 의 함수 포인터(파일시스템이 제각기 다르기 때문)가 들어있다.

과제 2. 복습 및 내용 정리(1 장, 2 장, 3 장 일부)

[Ch 1 : 리눅스 소개]
*. 386 보호모드: 인텔계열에만 있다. ARM 계열에는 없다
*. 유닉스는 태스크와 파일이라는 객체로 모든것을 지원한다.
*. 마이크로 커널: 디바이스 드라이버를 탈부착 가능하다.(USB 의 경우 마이크로 커널방식이라 꽂았다 빼도 상관없다) ↔ 모놀리식 커널
*. 리눅스의 경우 위의 두가지 방식을 혼합한 하이브리드 방식을 사용하여 상황에 따라 붙이거나 뗀다.(그래서 네크워크 장비들을 리눅스 사용)
리눅스의 장점
- 사용자 임의대로 재구성이 가능하다(C 언어, 공개소스, 커스터마이징 가능)
- 열악한 환경에서도 HW 자원을 적절히 활용하여 동작한다 (마이크로커널식으로 탈부착하기 때문에 최신버전 커널도 오래된 컴퓨터에서 작동이 가능하다)
*. 커널의 크기가 윈도우 커널보다 작다
*. 뛰어난 안정성(고성능 네트워크 장비에 사용되는 것으로 그 안정석이 검증되었다. 대표적으로 구글서버가 유닉스)
*. 강력한 네트워크 지원(TCP/IP, BSP)

[Ch 2 : 리눅스 구조]

- 커널은 운영체제의 핵심을 이루는 부분으로 시스템 리소스를 관리하고 사용자 프로그램이 시스템 콜에 의해 이를 사용할 수 있도록 한다.

운영체제는 시스템콜을 통해 태스크가 자원을 사용할 수 있게 해주는 자원관리자
1) 물리적 자원: CPU, 메모리, 디스크, 터미널, 네트워크
2) 물리적 자원을 관리하기 위해 추상화 시킨 객체들: → 리눅스 커널의 전반적 동작과정
태스크(CPU 의 추상화), 세그먼트 및 페이지(가상메모리 섹션, 메모리의 추상화), 파일(디스크의 추상화), 통신프로토콜 및 패킷(네트워크)
3) 물리적 자원은 아니면서 추상적 객체로만 존재하는 자원 : 접근제어

리눅스 커널 내부 구성
1) 태스크 관리자: CPU → 태스크
(태스크의 생성, 실행, 상태전이, 스케쥴링, 시그널 처리, IPC)
2) 메모리 관리자: 메모리 → 세그먼트, 페이지

(물리, 가상메모리 관리, 세그멘테이션, 페이징, 페이지 부재 결함처리)
3) 파일 시스템 : 디스크 → 파일
(파일의 생성, 접근제어, inode 관리, 디렉토리관리, 수퍼블록 관리)
4) 네트워크 관리자: 네트워크장치 → 소켓
(소켓인터페이스 , TCP/IP)
5) 디바이스드라이버관리자: 각종 디바이스 → 일관된접근
(디스크. 터미널, CD, 네트워크카드, 주변장치 구동 드라이버)

*. 리눅스 커널 소스 코드 훑어보기

task_struct > files_struct > file > path > dentry > inode > super_block > start_kernel() 함수

리눅스 커널 소스 구조와 각 디렉터리의 역할

1) kernel → SW 관리 (태스크 관리자 구현, 시그널처리)

cf. arch/kernel → HW(종속적인 태스크)관리, Context switching, 범용레지스터 정보 저장, 부트로더

특히, arch/mm 은 페이징 수행

2) arch : 하드웨어 종속적인 부분 구현, CPU 종류에 따라 하위 디렉토리로 구분 즉, CPU 모델에 따라 바뀌어야 하는 코드들이 들어있다 → Context switching, Bootloader 등등

어째서 Context Switch 및 Boot Code 가 어셈블리어로 만들어질 수 밖에 없는 것인가??

대표적인 아키텍쳐로는 인텔의 i386, ARM, SUN 사의 Sparc, IBM 의 PPC

3) fs : linux 에서 지원하는 파일시스템과 open(), read(), write() 와 같은 시스템 호출이 구현되어 있음.

4) mm : 메모리 관리자가 구현되어 있는 디렉토리, 물리 메모리, 가상 메모리, 동적 메모리 관리 기능이 구현되어 있음. 페이징 기법

5) driver: 리눅스에서 지원하는 디바이스 드라이버가 구현되어 있는 디렉토리

*. 디바이스 드라이버 종류

파일 시스템을 통해 접근되는 블록 디바이스 드라이버

사용자 수준 응용 프로그램이 장치파일을 통해 직접 접근하는 문자 디바이스 드라이버

tcp/ip 를 통해 접근되는 네트워크 디바이스 드라이버

6) net : ppp, wan, wimax, wireless, ham, ca, ethernet, irda(적외선)

7) ipc: 세마포어, 스핀락, 뮤텍스 , 쉐어드메모리등 각종 ipc 메커니즘

8) init : 커널의 초기화 관련 함수들, 하드웨어 관련 초기화

(부트로더 동작, 어셈블리 코드 동작 후→ init 디렉토리에 있는 start_kernel() 함수실행→ rest_init()까지 실행하면 로그인 창이 뜬다)

9) include : 리눅스 커널이 사용하는 헤더파일들이 구현(#include)

10) block: 블록디바이스(SRAM, DRAM, SSD) 드라이버 관련 내용

11) virt : 클라우드 서비스 관련
(virtual box 는 운영체제 위에 운영체제를 프로세스처럼 올릴수 있도록 만드는 기술이다. 이를 nested paging 이라고 한다 → 클라우드 서비스)

12) driver: 디바이스 드라이버

amba-ARM 사에서 만든 버스 프로토콜 fmc-초고속 무선통신 인터페이스 gpu-그래픽카드 power-전원장치관리 tty-터미널 mtd-NAND Flash regulator-정전압생성을 커널에서 관리하는 것을 알수있음	IIO-오실로스코프를 만들수 있다. 오실로스코프의 OS 도리눅스 dma-메모리에 직접 접근하는 기술 phy-유선관련 / wl-무선관련 wireless – b4s, brcm 은 퀄컴 / iwlwifi 는 TI

13) firmware: 빌드업된 펌웨어와 PC 간 통신하기 위해 필요

14) sound: ALSA 드라이버가 대표적

(cf. 비디오드라이버는 대표적으로 V4L2 가 있음 - 영상스트리밍의 핵심)

15) crypto: 암호화&복호화 관련

(cert, security, crypto)

arch 디렉토리 내부분석

arch 디렉토리에서 우리가 주로 관심을 가져야할 것들

arm/omap : TI Cortex-A , c6x : TI DSP, arm/zynq : Xilinx Cortex-A

blackfin : 아날로그 디바이스 사의 DSP(군용, 오디오, 비디오) openrisc : 오픈소스의 risc 프로세서 x86 : 인텔 cpu **c6x** : TI 사의 DSP ia64 : 인텔의 멸망한 cpu → 병렬처리, 모든 명령어를 싱글로 처리함 성능은 떨어짐 → but 덕분에 그래필카드 발전	microblaze : FPGA 에서 구동되는 가상의 (SW)cpu sparc : 슈퍼 컴퓨터용 cpu **arm, arm64** power pc : 자동차에 탑재 avr32 nios2 : 알테라 FPGA → 인텔에서 인수하여 x86 FPGA 에서 밀고 있는 아키텍쳐

arch/ arm 디렉토리 내부 분석

인텔은 단순하지만 arm 은 하위호환을 하지 않기 때문에 디렉토리가 많고 복잡

exynos: 삼성, 안드로이드 폰에 들어간다 omap: TI zynq: 자일링스 s3c24xx , s3c64xx: 삼성 keystone: TI DSP , DSP 만으로 커널이 올라감(키스톤 2 는 레이더에 사용되며 아직 오픈소스는 아니다) lpc: NXP, 차량용	bcm: 라즈베리파이 davinchi: TI→ 블랙박스, 스마트 tv, CCTV 등 비디오 시장을 장악 stm32: 차량용 processor 로 coretex R 이 사용될 것, 그러면서 커널에 등록됨 coretex R 전용 리눅스가 나올것 tegra: NVIDIA 에서 사용하는 리눅스 플랫폼

[Ch 3 : 태스크 관리]

*. 태스크, 스레드, 프로세스 유사하지만 다른 개념

*. task_struct 생성: fork(), pthread_create()

*. 프로세스 변화: exec()

프로세스, 스레드는 모두 태스크 → 둘다 task_struct 를 생성함

프로세스와 스레드의 구분

vi -t task_struct

pid_t pid
pid_t tgid

→ 프로세스 ID 와 스레드 그룹 ID 가 일치하면 프로세스이다.
즉 스레드 리더가 프로세스이다.
→ tgid 만 같고 pid 가 서로다르면 (같은 스레드 그룹에 속해있는)스레드이다.

ex. 삼성(tgid) , 사원번호(pid), 이재용(쓰레드 리더)
ex. 임베디드강의가 tgid 라면 쓰레드 리더는 프로세스인 선생님 학생들은 하나의 쓰레드 그룹에 속한 쓰레드이다.
각각의 pid 는 모두 다르다
*. fork 시에는 프로세스가 생성되고 pid = tgid 가 같은 세트들이 생성된다
pthread_create 시에는 pid 는 모두 다르고 tgid 값은 전부 같다. (이 중 pid=tgid 인것 하나 있다)

file [실행파일명] or read elf -h [실행파일명]
*. ELF 64bit executable
실행가능한 64 비트 ELF 포맷 → 리눅스 실행파일포맷
(윈도우 실행파일 포맷은 PE, 리눅스 디버깅파일 포맷은 DWARF)
*. 기계어가 어디에서 로드되었는지, 머신타입(x86-64, ARM-64)이 무엇인지 알 수 있다.
머신타입을 알면 구동하고자 하는 실행파일이 어떤 머신에서 동작할 지를 알 수 있다.

사용자 입장에서 프로세스 구조
가상 메모리의 실체는 task_struct > mm_struct >vm_area_struct

*. 사용자 프로세스가 수행되기 위해서는 여러가지 자원을 커널로부터 할당받아야 한다

- task_struct > mm_struct > vm_area_struct : 세그먼트
- pgd : 페이지테이블 관리(가상 → 물리메모리주소로 변환하는 페이징 수행)
- 가상메모리는 메모리공간을 잡지 않는다.
Unsigned long 형으로 start_code, end_code: 텍스트영역의 시작과 끝
start_data, end_data: 데이터 영역의 시작과 끝
start_bit, brk: 힙영역의 시작과 끝
start_stacl: 스택영역의 시작
결국 가상메모리는 8byte 로 처리한다!(long type2 개로 section 하나 결정)
이들은 페이징에 사용된다.

*. 실행파일은 메모리에 올라가야 프로세스가 된다