운영 체제

CH 0.

DAY-1

* 기본적인 동작

1. 파일의 내용을 저장할 공간을 할당 받고 저장. (= Disk Block, 일반적으론 4kb 단위)
2. Disk에 inode라는 파일의 정보(만들어진 시간, 사람, 접근 제어 정보 등)을 담고 있는 공간을 할당 받음.
3. Inode와 파일을 연결함
4. 컴파일러를 통해 파일을 수행 가능한 바이너리 파일(0,1로 이루어진 파일)로 만듦.
5. 바이너리 파일을 실행할 때에는 task라는 객체를 만듦.
6. task라는 객체를 만들기 위해 바이너리 파일을 우선 RAM에 load함.
7. Task는 RAM의 일부 공간(page frame)을 할당 받고 관리.
8. 기존에 존재하던 task들과 경쟁하며 CPU를 할당받기 위해 노력함.
9. 운영체제는 이런 task들에게 CPU를 할당해줌. (대표적으로 Round-Robin[[1]](#footnote-1) 방식이 유명)

* 임베디드 부팅을 위해선 3개의 파일이 필요하다.

1. Boot loader: 장치들을 초기화해주고 커널, 파일 시스템을 램에 복사해주는 기능
2. Kernel: 프로그램에 대한 자원 할당을 담당한다. 인터럽트 처리기와 스케줄러 등을 포함한다. 주기억장치에 저장된다.

zImage와 uImage는 리눅스 커널 중 하나다. uImage는 zImage의 특정 명령어를 압축한 것이다.

1. 하나의 압축 파일로, 기본적인 명령어, 라이브러리 파일을 압축하여 파일 시스템으로 만든다.

* Daemon: 한번만 실행해 놓으면 지속적으로 동작하는 프로그램 또는 명령어
* Shell: 이용자와 시스템 사이의 대화를 가능하게 해주는 명령 해석기. 이용자가 입력한 문장을 읽어, 그 문장이 요청하는 시스템 기능을 수행하게 해준다. Kernel과 달리 보조 기억 장치에도 저장이 될 수 있다.
* Idle 상태: 하드웨어 장치에서 다음 수행 명령을 기다리는 동안 별다른 작업을 수행하지 않고 기다리는 상태
* Super Block: 유닉스 시스템에서 파일 시스템의 상태를 설명하는 블록
* metadata: 다른 데이터를 설명해주는 데이터. Contents의 위치와 내용, 작성자 정보, 이용 조건, 내력 등이 담겨있다. metadata는 보통 데이터를 표현하기 위한 목적과 데이터를 더 빠르게 찾기 위한 목적이 있다.

CH1.

* Microkernel[[2]](#footnote-2): 운영체제의 기본적인 기능을 제공하는 kernel만 남기고 소형화한 것.
* Linux는 monolithic structure[[3]](#footnote-3)를 기반으로 설계되었다. (실제로 kernel 자체는 monolithic이지만 파일 시스템, 디바이스 드라이버, 스케줄링 등 여러 부분에서 모듈을 지원하기에 하이브리드라고 볼 수 있다.)

CH2.

Day-2

* OS = resource manager (system call을 통해 task가 resource를 활용할 수 있게 해준다.)

1. Physical resource

CPU, Memory, Disk, terminal, network 등 시스템 구성요소와 주변 장치

1. Abstract resource (matched with physical)

물리적 자원을 관리하기 위한 추상화 객체들

Ex) task, memory segment, page, protocol, packet

1. Abstract resource (Not matched with physical)

Security, ID access control

* Kernel

1. task manager

task의 생성, 실행, 상태 전이, 스케줄링, 시그널 처리, 프로세스 간 통신 지원

1. memory manager

Segment[[4]](#footnote-4)와 page[[5]](#footnote-5) 관리

1. filesystem

파일 생성, 접근 제어, inode 생성, 디렉터리 관리, super block 관리

1. network manager

네트워크 장치를 socket으로 제공, TCP/IP 같은 통신 프로토콜

1. device driver manager.

주변 장치를 일관되게 접근하도록 해줌

* gcc는 GNU의 Complier다.

Preprocessor, complier, assembler, linker 과정을 통해, 실행파일을 생성할 수 있다.

Preprocessor는 전처리 지시자들을 해석하고, complier는 high level language를 .s 형태의 어셈블리어 형태로 전환한다. Assembler는 .s 파일을 .o 형태의 object file로 변환한다. Linker는 생성된 object file들을 .out 형태의 execute file로 만든다.

* + Symbol은 주소를 갖는 최소 단위로, 크게 RO, RW, ZI 부분으로 나뉜다. RO부분은 보통 ROM에 담기는 것으로 code(function), const variable등이 포함된다. RW는 global variable 중 초기화가 된 variable을 포함한다. ZI은 0으로 초기화가 되거나 초기화가 되지 않은 global variable이다. 그리고 static이 아닌 local variable들은 ZI(stack이나 heap)에 자리잡는다.
  + Object file의 크게 .text, .data, .bss section으로 구성되어 있다. .text는 RO, .data는 .RW, .bss는 ZI에 해당한다. Linker를 이용하여 ELF file을 만들 때, 각 object file의 section들끼리 묶여, ELF file의 segment가 된다. 다른 object file에서 참조한 변수나 코드들을 .rel.data나 .rel.text 등으로 linker될 때 relocate되어 연결된다. ELF format은 RO segment의 segment header table을 시작으로 .init, .text, .rodata로 구성되어 있고, RW segment의 .data, .bss 부분, 메모리에 load되지 않는 .symtab(symbol table), .debug, .line, .strtab, object file의 section을 설명하는 section header table로 구성되어 있다.
* Makefile

파일간의 종속관계를 나타낸 것으로 컴파일러가 순차적으로 실행될 수 있게 해준다.

Make를 쓰게되면 프로그램의 종속 구조를 파악하기 쉽고, 반복 작업의 최소화가 가능하다.

Target, dependency, command, macro로 구성되며, 코드를 단순화시키고, 수정을 용이하게 하기 위해 macro를 사용한다.

CH3

DAY-3

* Program과 process
  + Process는 동작중인 program이다. 디스크에 저장되어 있는 program(파일 형태)이 kernel로부터 자원을 할당 받아 동적인 객체가 된 것이 process라고 할 수 있다.
* Process의 구조

Process의 생김새는 가상 주소[[6]](#footnote-6) 공간에서의 모양을 의미한다. 프로세스는 크게 text, data, heap, stack 부분으로 나눌 수 있다. User space에서 text(함수, instruction 등)는 제일 하위 주소 공간을 차지하고, 그 다음 data(glob var), heap, stack 순으로 차지한다. Local 변수는 stack에, dynamic var는 heap에 동적으로 할당되는데, stack는 kernel, user space의 경계면에서 아래로 커지고 heap은 위로 커진다. 각 영역을 segment 또는 vm\_area\_struct(가상 메모리 객체)라고 한다.

* Process의 생성

Process는 fork()와 vfork()로 생성된다.

* + fork()

fork()는 자식 프로세스를 생성하고, 메모리 영역을 할당하여, 부모 프로세스의 메모리 공간을 모두 복제한다. 하지만 모두 복제하는 것은 비효율적이기 때문에 최근에는 COW 방식으로 기본적으로는 메모리 공간을 참조만 하고 있다가 변경이 확인되면 복제하는 방식으로 변했다. 부모 프로세스와 자식 프로세스는 변수를 공유하지 않는다. fork()는 자식 프로세스가 생성되면, 부모 프로세스에는 자식 프로세스의 PID[[7]](#footnote-7)를, 자식 프로세스에는 0을 반환한다. 에러 시에는 적당한 에러 값을 반환한다.

* + vfork()

fork() 함수는 주로 fork 뒤에 exec()를 실행함으로써 자식과 부모 프로세스를 동시에 실행하는 경우가 많다. 하지만 이럴 경우 불필요한 복제가 일어나기 때문에 오버헤드가 일어날 경우를 위해 vfork()를 사용한다. vfork()는 부모의 메모리 공간을 공유한다. 따라서 부모와 자식이 변수를 공유한다. 따라서 부모와 자식의 race condition을 막기 위해 자식 프로세스가 생기면 부모 프로세스는 sleep된다. vfork()뒤에 바로 exec() 계열 함수를 호출하지 않으면 자식 프로세스에서 변경한 변수가 부모 프로세스에도 적용된다는 위험성이 있다.

* + clone()

쓰레드를 생성하는 함수. fork()는 별도의 메모리 공간을 할당한 뒤 부모 프로세스의 영역 값을 복사하지만 clone()는 메모리 자체를 공유한다. clone()으로 생긴 프로세스의 값 변경이 부모 프로세스에도 적용이 되므로 쓰레드라 봐도 무방하다.

* + exec()

exec() 함수는 사용하던 메모리 공간을 해제하고 새로운 실행파일을 로드하고 실행한다. 즉 exec()를 호출한 프로세스는 실행이 중지되고 새로운 프로세스로 교체된다.

* 한 프로세스에 여러 쓰레드가 동작할 수 있다. 이런 모델을 다중 쓰레드 시스템이라 하는데 쓰레드의 생성은 프로세스의 생성보다 resource가 적게 든다. 하지만 자식 쓰레드에서의 결함이 부모 쓰레드로 전파가 된다는 단점이 있다. 따라서, 쓰레드 모델은 지원 공유에 적합하고, 프로세스 모델을 결함 고립에 적합하다.

DAY - 4

* Linux task model

프로세스는 자원과 자원에서 수행되는 수행 흐름으로 구성된다. 이를 관리하기 위해 task\_struct라는 자로 구조를 생성한다. 리눅스 커널은 프로세스와 쓰레드 모두 task\_struct로 관리한다. 프로세스와 쓰레드는 task\_struct에서 해석의 여부에 차이가 있다. 리눅스에서는 프로세스와 쓰레드를 모두 task라는 객체로 관리한다.

리눅스에서는 fork(), vfork(), clone()을 이용해서 프로세스와 쓰레드를 생성한다. fork()와 clone()은 커널의 sys\_clone()을, vfork()는 sys\_vfork() 시스템 호출을 사용한다. sys\_clone()과 sys\_vfork() 모두 커널 내부 함수인 do\_fork()를 호출한다. 리눅스에선 프로세스와 쓰레드 모두 task를 생성하기 때문인데 do\_fork()의 인자로 부모 태스크와의 관계를 지정해, fork()와 clone()을 구별한다.

* + do\_fork()

do\_fork()는 우선적으로 task\_struct를 구성한다. Task의 이름, 부모 task 등을 기록한다. 그 후 task에 자원을 할당한 뒤 수행 가능한 상태로 만들어준다.

Task는 task만의 유일한 id인 PID가 있다. 하지만 한 프로세스 내의 쓰레드는 같은 ID를 공유해야 하는데 이것을 위해 tgid(thread group id)라는 개념이 있다. Task가 처음 생성되면 유일한, PID값을 같게 된다. 여기서 쓰레드면 tgid값에 부모 쓰레드의 PID값을 넣어주고 프로세스면 새로 할당된 PID값을 tgid에 넣어주게 된다.

결과적으로 fork()와 vfork()는 부모 태스크와 pid, tgid가 다르고 clone()은 부모 태스크와 pid는 다르지만 같은 tgid를 가진다.

* task context
  + Context: 태스크와 관련된 모든 정보를 말한다. 태스크 우선순위, CPU 사용량, 태스크 간의 관계, 시그널, 사용 자원, file descriptor 등을 의미한다. Task context는 크게 세가지로 나눌 수 있다.

1. System context

태스크의 정보를 유지하기 위해 커널이 할당한 자료구조.

Task\_struct, file descriptor, file table, segment table, page table 등이 있다.

1. Memory context

Text, data, stack, heap, swap 공간이 포함된다.

1. Hardware Context

문맥 교환을 할 때 수행 현재 실행 위치에 대한 정보를 저장한다.

태스크가 대기 상태나 준비 상태로 전환될 때 다음에 실행될 부분을 저장해둔다.

* task\_struct

1. task identification

태스크를 인식하기 위한 변수들이 있다. PID, TGID, 해쉬.

또한 audit\_struct를 통해 태스크에 대한 사용 접근 권한을 제어하는 UID, EUID, SUID, FSUID등과 사용자 그룹에 대한 접근 관리인 GID, EGID, SGID, FSGID등이 있다.

1. State

태스크의 생성부터 소멸까지 태스크의 상태에 관한 변수다.

Ex) TASK\_RUNNING(0), TASK\_INTERRUPTIBLE(1), TASK\_UNINTERRUPTIBLE(2), TASK\_STOPPED(4), TASK\_TRACED(8), EXIT\_DEAD(16), EXIT\_ZOMBIE(32)

1. Task relation

Task의 가족 관계를 나타내는 구조체, 변수들이 포함된다.

현재 태스크를 생성한 부모 태스크의 task\_struct를 가르키는 real\_parent와 현재 현재 부모 태스크의 task\_struct를 가르키는 parent 필드가 존재한다. 또한 자식과 형제 관계인 children, sibling는 그 관계를 리스트로 저장한다.

모든 리눅스 커널의 태스크들은 init\_tastk로부터 시작해, 이중 연결리스트로 연결되어 있으며, task\_struct 구조체의 tasks라는 리스트 헤드를 통한다. 또한 RUNNING 상태인 태스크들은 run\_list필드를 통해 따로 연결되어 있다.

1. Scheduling information

스케쥴링과 관련이 있는 변수. Prio, policy, cpus\_allowed, time\_slice, rt\_priority등이 있다.

1. Signal information

태스크에게 비동기적[[8]](#footnote-8) 사건을 알리는 매커니즘.

Signal, sighand, blocked, pending

1. Memory information

Memory context에 관한 크기, 접근, 제어에 관한 정보 등을 관리하는 변수들이 있다.

1. File information

Task가 오픈한 파일들에 대한 변수다. Files\_struct 구조체의 files 변수로 접근이 가능하며, inode는 fs\_struct 구조체의 fs 변수로 접근이 가능하다.

1. Thread structure

문맥 교환을 실행할 때 태스크가 어디까지 실행이 되었는지 저장해두는 공간이다.

1. Time information

태스크의 시작시간, 사용한 자원 시간 등등을 위한 변수로, start\_time, real\_start\_time 등이 있다.

1. Format

다른 커널의 컴파일과 이진 포맷을 지원하기 위한 필드가 존재한다.

1. Resource limits

태스크가 사용할 수 있는 자원의 한계를 의미한다. rlim\_max는 최대 허용 자원 수, rlim\_cur는 현재 설정된 허용 자원 수를 의미한다.

* State transition

태스크가 생성되면 TASK\_RUNNING 상태가 된다. 이 후에 스케줄러에서는 여러 태스크 중 실행시킬 태스크를 선택하여 수행시키기 때문에 TASK\_RUNNING은 실질적으로 ready 상태와 running 상태 두가지로 분류할 수 있다. 실행중인 태스크가 상태전이를 할 수 있는 여러가지 경우가 있는데,

1. 태스크가 자신의 일을 다 끝내고 exit()을 호출하면 TASK\_DEAD(EXIT\_ZOMBIE) 상태로 전이된다. 이 상태에선 태스크에 할당되어 있던 대부분의 자원을 반납하고, 태스크가 종료된 이유, 자원 정보들을 유지한다. 이 후에 부모 태스크가 wait()등의 함수를 호출하게 되면 TASK\_DEAD(EXIT\_DEAD) 상태로 전이되어 태스크가 완전히 종료되게 된다. Wait()를 호출하기 전에 부모 태스크가 종료되면 init\_task가 부모가 된다.
2. TASK\_RUNNING(running) 상태인 태스크가 할당된 CPU를 모두 사용하거나 더 높은 우선순위를 가지는 태스크가 나타나면 TASK\_RUNNING(ready) 상태가 된다.
3. SIGSTOP, SIGTSTP, SIGTTIN 등의 시그널을 받은 태스크는 TASK\_STOPPED 상태로 전이된다. 이 후 SIGCONT 시그널을 받으면 다시 TASK\_RUNNING(ready) 상태로 전이된다.
4. TASK\_RUNNING(running) 상태의 태스크가 다른 요청을 기다려야하는 경우 TASK\_INTERRUPTIBLE 등으로 전이된다. TASK\_UNINTERRUPTIBLE은 시그널에 반응하지 않는다는 점에서 다르다. 하지만 중요한 SIGKILL에도 반응을 하지않으면 곤란하기에 TASK\_KILLABLE이라는 상태가 추가되었다.

* 사용자 수준

사용자가 만든 응용프로그램이나 라이브러리를 실행하고 있는 상태다.

커널을 호출하지 않고 독립적으로 스케줄링이 이루어지기 때문에 이식성이 높다. 따라서 context switching이 일어나지 않기 때문에 overhead가 감소한다. 동일한 프로세스의 스레드 1개가 ready 상태가 되면 다른 스레드도 실행을 하지 못하게 되는 단점이 있다. 따라서 blocked를 사용하지 못하고 non\_blocked를 사용해야하는 단점이 있다.

사용자 영역의 쓰레드 n개가 커널 영역의 쓰레드 1개와 매칭된다.

* 커널 수준

CPU에서 커널의 코드를 수행하고 있는 상태다. 사용자 수준과 커널 수준이 1 : 1로 매칭이 된다. 쓰레드가 ready 상태가 되도 동일한 프로세스의 다른 쓰레드로 대체가 된다. 하지만 context switching이 많이 일어나기에 overhead가 더 많이 일어난다.

* 사용자 수준에서 커널 수준으로 전이할 수 있는 방법은 시스템 호출 사용과 인터럽트가 있다.

커널 수준에서 실행되는 코드는 리눅스 그 자체다. 따라서 태스크가 생성되면 태스크별로 일정 크기의 stack을 할당한다. 따라서 시스템 호출이 일어나면 stack을 이용하여 작업을 처리해준다.

또한 시스템 호출을 했다면 작업 후 원래 사용자 수준 실행 상태로 돌아가야 하는데, 그 전 작업 상황까지의 상태를 저장해놓는 공간을 pt\_regs라고 한다.

* Runqueue & scheduling
  + Scheduling

여러 개의 태스크 중에서 다음에 실행시킬 태스크를 선택하여 자원을 할당하는 과정을 일컫는다. 실시간 태스크[[9]](#footnote-9)는 0 ~ 99 단계를 사용하며, 일반태스크는 100 ~ 139 단계를 사용한다. 실시간 태스크는 일반 태스크보다 항상 먼저 실행된다.

* 일반적으로 OS에서는 실행가능한 상태의 태스크를 자료 구조를 통해 관리하는데 이 자료구조를 Runqueue라고 한다. 부팅 이후, CPU 별로 하나씩 유지된다. Task가 처음 생성되면 init\_task를 헤드로 하는 이중 연결리스트에 연결되는데 이것을 tasklist라고 한다. TASK\_RUNNING 상태가 되면 runqueue중 하나로 소속된다. 새로 생성된 태스크는 보통 부모 태스크의 runqueue로 삽입된다. 하지만 한쪽의 runqueue가 부하가 일어난다면 다른 runqueue로 이동시키기도 한다. 이런 이동은 비교적 같거나 가까운 도메인의 runqueue로 일어난다.
* FIFO, RR, DEADLINE

실시간 태스크는 우선 순위 결정을 위해 rt\_priority 필드를 사용한다.

* + RR

동일 우선 순위를 가지는 태스크가 여러 개일 경우 타임 슬라이스를 기반으로 스케줄링이 된다.

* + FIFO

동일한 우선순위를 가지는 태스크가 없을 경우, 우선 순위가 높은 태스크부터 수행된다.

* + DEADLINE

현재시간 + runtime < deadline을 이용해, 새로운 DEADLINE 태스크의 완료 여부를 확정적으로 결정할 수 있다. 가장 가까운 DAEDLINE을 지닌 태스크부터 대상으로 선정하며, DEADLINE은 Red-Black tree에 의해 정렬된다.

* 이러한 스케줄링은 태스크의 개수가 늘어날수록 스케줄링에 걸리는 시간이 길어졌기에 태스크에 대한 비트맵[[10]](#footnote-10)이 생겨났다.
* 일반 스케줄링(CFS[[11]](#footnote-11))

CFS는 각 태스크의 CPU 사용시간이 같게 한다. N개의 태스크가 존재하면 정해진 시간을 N으로 나눠 할당해준다. 시간을 너무 촘촘히 나누면 context switching으로 인한 overhead가 일어날 가능성이 크기 때문에 시간 설정을 잘 해주어야 한다. 만약 태스크에 우선 순위가 있다면 그만큼 가중치를 두어 할당 시간을 늘려준다.

스케줄링에는 vruntime이 사용된다. vruntime 또한 key값으로 하여 RBtree에 정렬된다.

결과적으로 태스크가 생성되면 가장 작은 vruntime값을 가지게 되어 빠른 수행을 하게된다. vruntime값은 주기적으로 갱신되며, 가장 작은 vruntime을 갖는 태스크가 다음 수행을 하게 된다.

스케줄러는 언터럽트 후에 need\_resched 필드가 활성화되었거나, 현재 태스크가 타임 슬라이스를 모두 사용했거나, 태스크가 새로이 생겨나거나, 스케줄에 관련된 시스템 호출을 할 경우에 일어난다.

또한 CFS는 한 프로세스의 CPU 점유를 막기위해 group scheduling을 지원한다.

* Context switch

수행 중이던 태스크의 동작을 멈추고, 다른 태스크로 전환하는 과정을 context switch라고 한다.

Kernel은 context switch가 일어나면 태스크가 그 시점에 어디까지 수행했는지, 현재 CPU의 register 값은 어떤지 저장해둔다(context save). 이런 값들은 thread\_struct 필드에 저장된다.

스케줄링과 context switch, save는 모두 커널 수준에서 동작한다. 따라서 context switch가 일어나면 사용자 수준에서 커널 수준으로 상태를 전이해야 하고, 태스크 A에 할당된 커널 스택에 CPU 레지스터 정보가 저장된다. 그 후 스케줄링을 통해 context switch를 하게되면 thread\_struct에 CPU register 정보를 저장하게 된다. 그 다음 실행될 태스크 B의 CPU register 정보를 복원하고, 커널 수준에서 작업이 모두 완료되었다면, B의 커널 스택에서 CPU정보를 복원한 후 사용자 수준으로 상태 전이를 하게 된다.

1. 일정시간 동안 한 task가 CPU를 사용하고 시간이 지나면 다음 task가 사용하는 방식 [↑](#footnote-ref-1)
2. Windows NT, Mac OS에서 사용 [↑](#footnote-ref-2)
3. 모듈화와는 다르게 독립성이 없는 소프트웨어 블록들의 집합으로 이루어진 소프트웨어.

   Monolithic는 최적화에 좋고, 속도가 빠르며, 경로를 단축시킬 수 있지만 유지보수가 힘들다. [↑](#footnote-ref-3)
4. 주 기억장치에 load되는 프로그램 분할의 기본 단위 [↑](#footnote-ref-4)
5. 프로그램을 한번에 처리할 수 있는 단위. Page 단위로 주 기억 장치에 로드하고 언로드한다. [↑](#footnote-ref-5)
6. 한정된 RAM의 한계를 극복하고자 disk같은 보조 저장 장치를 이용해, 디스크 공간을 메모리처럼 활용할 수 있게 해주는 기법이다. Disk에 존재하는 파일을 paging file이라고 하며, process는 가상 주소를 사용하고, 읽고 쓸 때만 MNU를 사용해 물리주소로 변환한다. Process가 RAM의 공간을 초과해, disk를 사용할 때는 물론, 성능이 저하된다. Thread는 각 process에 할당된 주소 공간에만 접근할 수 있다.

   가상메모리는 kernel space와 user space로 나뉘는데 kernel space는 kernel, driver 등을 실행시키기 위한 예비 메모리고 user space는 process에 할당되는 공간이다. [↑](#footnote-ref-6)
7. Process ID [↑](#footnote-ref-7)
8. 호출과 응답이 동시에 이뤄지지 않는 것. [↑](#footnote-ref-8)
9. 시간적으로 여러 제약을 갖는 task [↑](#footnote-ref-9)
10. 태스크가 생성되면 비트맵에서 그 태스크의 우선순위에 해당하는 비트를 set하고, 우선 순위에 해당하는 queue에 삽입한다. 스케줄링을 할때에 가장 우선순위가 높은 큐에 있는 태스크들을 선택한다. [↑](#footnote-ref-10)
11. Completely Fair Schedluer [↑](#footnote-ref-11)