

운영 체제

CH 0.

- 기본적인 동작

- 1) 파일의 내용을 저장할 공간을 할당 받고 저장. (= Disk Block, 일반적으로 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 만들.
- 5) 바이너리 파일을 실행할 때에는 task라는 객체를 만들.
- 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-Robin¹ 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.

- 3) 하나의 압축 파일로, 기본적인 명령어, 라이브러리 파일을 압축하여 파일 시스템으로 만든다.

- Daemon: 한번만 실행해 놓으면 지속적으로 동작하는 프로그램 또는 명령어

¹ 일정시간 동안 한 task가 CPU를 사용하고 시간이 지나면 다음 task가 사용하는 방식

- 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.

- Microkernel²: 운영체제의 기본적인 기능을 제공하는 kernel만 남기고 소형화한 것.
- Linux는 monolithic structure³를 기반으로 설계되었다. (실제로 kernel 자체는 monolithic이지만 파일 시스템, 디바이스 드라이버, 스케줄링 등 여러 부분에서 모듈을 지원하기에 하이브리드라고 볼 수 있다.)

² Windows NT, Mac OS에서 사용

³ 모듈화와는 다르게 독립성이 없는 소프트웨어 블록들의 집합으로 이루어진 소프트웨어.

Monolithic는 최적화에 좋고, 속도가 빠르며, 경로를 단축시킬 수 있지만 유지보수가 힘들다.

CH2.

- 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

- 2) Abstract resource (matched with physical)

- 물리적 자원을 관리하기 위한 추상화 객체들

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

- 3) 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

- 2) memory manager

- Segment⁴와 page⁵ 관리

- 3) filesystem

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

- 4) network manager

- 네트워크 장치를 socket으로 제공, TCP/IP 같은 통신 프로토콜

- 5) device driver manager.

- 주변 장치를 일관되게 접근하도록 해줌

- gcc는 GNU의 Compiler다.

⁴ 주 기억장치에 load되는 프로그램 분할의 기본 단위

⁵ 프로그램을 한번에 처리할 수 있는 단위. Page 단위로 주 기억 장치에 로드하고 언로드한다.

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

Preprocessor는 전처리 지시자들을 해석하고, compiler는 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

- 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 주소⁶ 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 PID⁷를, 자식 프로세스에는 0을 반환한다. 에러 시에는 적당한 에러 값을 반환한다.

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.

⁷ Process ID

■ 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.

- Linux task model

프로세스는 자원과 자원에서 수행되는 수행 흐름으로 구성된다. 이를 관리하기 위해 task_struct라는 자료 구조를 생성한다. 리눅스 커널은 프로세스와 스레드 모두 task_struct로 관리한다. 프로세스와 스레드는 task_struct에서 해석의 여부에 차이가 있다. 리눅스에서는 프로세스와 스레드를 모두 task라는 객체로 관리한다.

리눅스에서는 fork(), vfork(), clone()을 이용해서 프로세스와 스레드를 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 스레드 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 스레드는 같은 ID를 공유해야 하는데 이것을 위해 tgid(thread group id)라는 개념이 있다. Task가 처음 생성되면 유일한, PID값을 갖게 된다. 여기서 스레드면 tgid값에 부모 스레드의 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.

2) Memory context

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

3) 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.

2) 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)

3) 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_task로부터 시작해, 이중 연결리스트로 연결되어 있으며, task_struct 구조체의 tasks라는 리스트 헤드를 통한다. 또한 RUNNING 상태인 태스크들은 run_list필드를 통해 따로 연결되어 있다.

4) Scheduling information

스케줄링과 관련이 있는 변수. Prio, policy, cpus_allowed, time_slice, rt_priority등이 있다.

5) Signal information

태스크에게 비동기적⁸ 사건을 알리는 매커니즘.

Signal, sighand, blocked, pending

6) Memory information

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

7) File information

Task가 오픈한 파일들에 대한 변수다. Files_struct 구조체의 files 변수로 접근이 가능하며, inode는 fs_struct 구조체의 fs 변수로 접근이 가능하다.

8) Thread structure

문맥 교환을 실행할 때 태스크가 어디까지 실행이 되었는지 저장해두는 공간이다.

⁸ 호출과 응답이 동시에 이뤄지지 않는 것.

9) Time information

태스크의 시작시간, 사용한 자원 시간 등등을 위한 변수로, `start_time`, `real_start_time` 등이 있다.

10) Format

다른 커널의 컴파일과 이진 포맷을 지원하기 위한 필드가 존재한다.

11) Resource limits

태스크가 사용할 수 있는 자원의 한계를 의미한다. `rlim_max`는 최대 허용 자원 수, `rlim_cur`는 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_RUNNING`은 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_task`가 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_UNINTERRUPTIBLE`은 시그널에 반응하지 않는다는 점에서 다르다. 하지만 중요한 `SIGKILL`에도 반응을 하지않으면 곤란하기에 `TASK_KILLABLE`이라는 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 스레드 n개가 커널 영역의 스레드 1개와 매칭된다.

- 커널 수준

CPU에서 커널의 코드를 수행하고 있는 상태다. 사용자 수준과 커널 수준이 1대 1로 매칭이 된다. 스레드가 ready 상태가 되도 동일한 프로세스의 다른 스레드로 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 태스크⁹는 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 자료 구조를 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 DEADLINE을 지닌 태스크부터 대상으로 선정하며, DEADLINE은 Red-Black tree에 의해 정렬된다.

⁹ 시간적으로 여러 제약을 갖는 task

- 이러한 스케줄링은 태스크의 개수가 늘어날수록 스케줄링에 걸리는 시간이 길어졌기에 태스크에 대한 비트맵¹⁰이 생겨났다.

- 일반 스케줄링(CFS¹¹)

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 하게 된다. vruntime값은 주기적으로 갱신되며, 가장 작은 vruntime을 갖는 태스크가 다음 수행을 하게 된다.

스케줄러는 인터럽트 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.

¹⁰ 태스크가 생성되면 비트맵에서 그 태스크의 우선순위에 해당하는 비트를 set하고, 우선 순위에 해당하는 queue에 삽입한다. 스케줄링을 할때에 가장 우선순위가 높은 큐에 있는 태스크들을 선택한다.

¹¹ Completely Fair Scheduler

- 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 하게 되면 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.

CH4

- Virtual memory(가상 메모리)

가상 메모리는 실제 시스템에 존재하는 물리 메모리의 크기와 관계없이 가상적인 주소 공간을 사용자 태스크에게 제공한다. 32bits CPU의 경우 word의 크기가 32bits이기 때문에 표현할 수 있는 주소의 크기가 2^{32} (4GB)이고 64bits CPU의 경우 2^{64} (16EB)다. 따라서 32bits의 CPU인 경우 각 태스크마다 4GB의 메모리 공간을 가지고 있다고 생각할 수 있다.

프로그램은 보조기억장치(HDD, SSD)에 저장되어있다가 실행이 될 때 주 기억장치(RAM)에 load(적재)가 되는데 프로그램의 크기가 RAM의 크기보다 크거나 여러 프로그램일 경우 문제가 발생할 수 있다. 따라서 태스크마다 가상 메모리를 지정해주고 프로그램이 실행될 때만 RAM에 load가 된다. 즉, 여러 개의 프로그램이 돌아갈 경우 당장 실행되는 프로그램만 RAM에 적재가 되고, 큰 프로그램일 경우는 프로그램 하나일지라도 모든 부분이 한번에 실행되지는 않기 때문에 실행되고 있는 부분만 RAM에 load하는 방식으로 가상 메모리를 활용한다.

- Physical memory(물리 메모리)

■ SMP(Symmetric Multiprocessing)

메모리 주소에 접근하는데 모든 프로세서가 같은 cost를 갖는 멀티프로세서를 의미한다. 초기에는 cache가 없는 SMP 구조였지만 각 프로세서가 같은 bus를 사용하기 때문에 data traffic을 줄이고 속도를 향상시키기 위해 각 프로세서마다 cache가 있는 형태로 발전했다. UMA(Unified Memory Access)라고 불리기도 한다.

■ NUMA(Non-Unified Memory Access)

NUMA는 멀티프로세서에서 메모리 접근 시간이 로컬 메모리와 프로세서 간의 관계에 의존적인 것을 말한다. 프로세서는 자신의 로컬 메모리에 다른 메모리 접근이 빠르다. NUMA에는 특정 프로세서의 과부하를 줄이고, 태스크와 데이터 간 관련성을 알 수 있다는 장점이 있다.

- Node

리눅스에서는 접근 속도가 같은 메모리의 집합을 [bank](#)라고 하는데 이 bank를 표현하는 구조를 Node라고 한다. 하나의 Node는 pg_data_t 구조체로 표현된다.

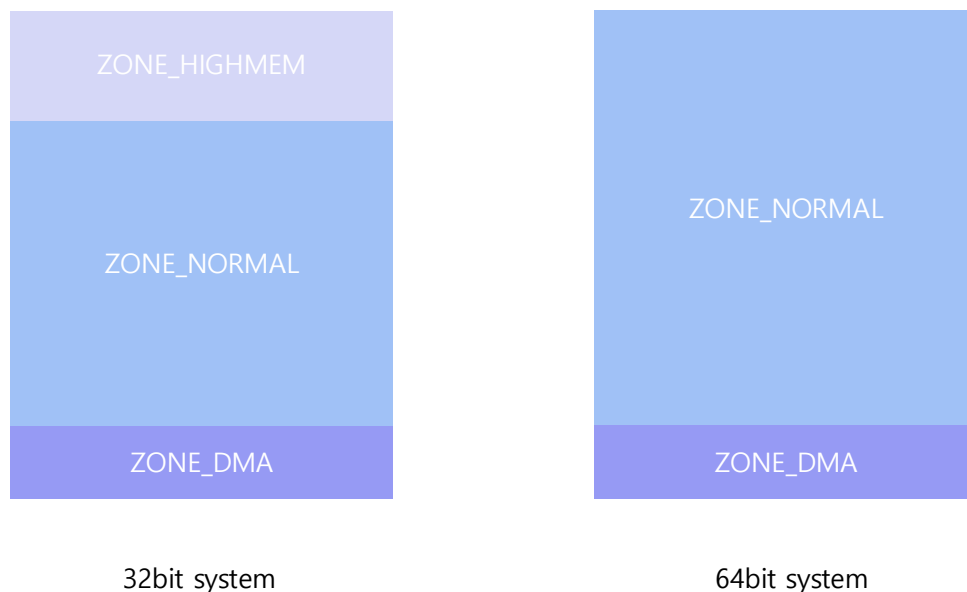
- Zone

Zone은 유사한 속성의 페이지 프레임들의 집합이다.

ISA 버스¹² 기반의 디바이스는 동작을 하기 위해 물리메모리 중 16MB 이하 부분을 할당 해줘야 하는 경우가 있다. ISA 버스 구조에서도 원활히 리눅스가 수행되기 위해 16MB 이하의 부분을 따로 관리할 수 있도록 하는 자료 구조를 zone이라고 한다. 리눅스에서 16MB 이하의 메모리는 ZONE_DMA라 하고, 이상의 메모리를 ZONE_NORMAL이라고 한다.

32bit system에서는 리눅스 커널은 1GB의 가상 공간을 차지한다. 따라서 물리 메모리가 1GB가 넘어간다면 1대 1로 매핑할 수 없다. 따라서 물리 메모리가 1GB 이상이라면 896MB(아키텍처마다 다르다.)는 물리 메모리와 1대 1로 매핑해주고 그 이상은 필요할 때 동적으로 물리 메모리에 할당해준다. 1대 1로 매핑되는 부분을 ZONE_NORMAL이라 하고, 그 이상의 부분을 ZONE_HIGHMEM이라 한다. 1대 1로 물리 메모리와 매핑되는 ZONE_DMA, ZONE_NORMAL은 커널 메모리 할당 시 가장 빠르게 접근할 수 있는 영역이다. 또한 32bit에서는 영역의 크기가 제한되어 있기 때문에 cost가 비싼 영역이고 물리 메모리가 할당될 때, ZONE_HIGHMEM이 모두 소진된 후에 할당이 된다.

64bit system인 경우, 가상 메모리가 모든 물리 메모리 영역에 1대 1로 할당이 될 수 있기 때문에 ZONE_HIGHMEM을 따로 만들지 않는다.



또한 ZONE이 꼭 여러 개가 존재해야 하는 것이 아니라 아키텍처에 따라 1개만 존재할

¹² CPU와 주변 디바이스와의 연결을 위한 16bit 버스 구조이다. 최근에는 32bit, 64bit의 디바이스가 많이 나와, 사라지고 있는 추세이며 PCI 버스로 교체되고 있다.

수도 있다.

또한 ZONE_MOVABLE, ZONE_DEVICE 영역도 존재한다. ZONE_DEVICE는 ZONE_DMA와 다르게 CPU에서 대용량 디바이스 메모리(PMEM¹³, HM)에 접근하여 사용하기 위한 영역이다.

- Page frame

물리 메모리의 최소 할당 단위를 페이지 프레임이라고 한다. (가상 메모리의 최소 할당 단위는 페이지) 즉, 여러 개의 페이지 프레임이 ZONE을 구성하며, 하나 이상의 ZONE이 Node를 구성하고, Bank¹⁴의 개수(UMA or NUMA)에 따라 하나 이상의 Node가 리눅스 전체 물리 메모리를 관리한다.

¹³ 메모리 버스에 상주하는 SSD라고 볼 수 있다. 메모리 버스에서 DRAM과 동일한 성능을 가진다. 즉, DRAM과 동일한 빠르기를 가진 비휘발성 메모리라고 할 수 있는데, DRAM보다 싸고, 크며, 영구적이라는 장점이 있다. 또한 PCI Express의 cache로 사용될 수 있다.

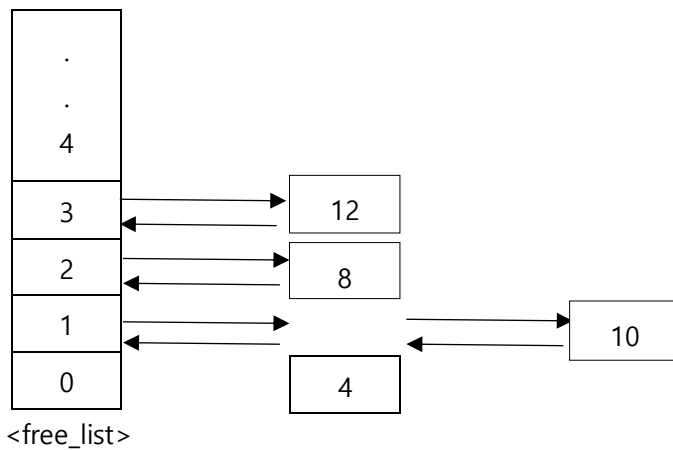
¹⁴ 접근 속도가 같은 메모리 집합

- Buddy allocator

메모리 부하와 외부 단편화를 해결하기 위해 도입되었다.

Buddy 할당자는 ZONE 구조체에서 free_area[] 배열에 의해서 구축된다. Free_area는 10개의 엔트리를 가지고 있는데 free_area가 관리하는 할당의 크기를 나타내며, 즉, 4MB($2^{10} * 4KB$)까지 할당할 수 있다. Free_area 구조체는 free_list와 map을 통해 할당된 페이지 프레임들을 list와 bitmap으로 관리한다.

예를 들어 색으로 칠해진 페이지 프레임들을 사용하고 있고 나머진 free 상태라 볼 때, 색으로 칠해진 비트가 할당되고, 색으로 칠해진 비트가 해제되었다고 하면,



Pages	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
order(0)	0		0		1->0		0		0		1		0		0	
order(1)	0				0 -> 1				1 -> 0				0			
order(2)	0								1							
order(3)	0															

<bitmap>

Order(0)는 free_area[0]와 같은데, order(n)은 연속된 2^n 개의 페이지 프레임이 free 상태로 존재하는지를 나타내 준다. 예를 들어 2개의 연속된 페이지 프레임을 할당받아야 할 때, 비트맵에서 order(1) 부분을 확인해 1로 된 부분이 있으면 bitmap을 0으로 수정하고 할당해 준다. 만약 없다면, 보다 큰 order(n)에서 연속된 2^n 개의 페이지 프레임을 분할해, 할당하고 bitmap을 수정해준다. 이때, 나누어진 두 부분을 buddy라고 부르며 이때문

에 buddy allocator라고 부른다. 해제할때도 마찬가지로, 해당 부분을 해제하고 bitmap을 수정해준다.

- Lazy buddy

Lazy buddy는 할당과 해제로 인한 오버헤드를 줄이기 위한 할당자 구조다. Buddy는 ZONE마다 유지되고 있는 watermark의 값과 사용가능한 페이지 프레임 수를 비교한다. 가용 메모리가 충분한 경우 페이지 병합 작업을 최대한 미루고 메모리가 부족해질 때 병합 작업을 수행한다.

메모리를 반납하는 함수는 `_free_pages()`이고, 할당하는 함수는 `_alloc_pages()`다. `_free_pages()` 함수는 내부적으로, MAX_ORDER(위에서는 10)만큼 loop를 돌며, 해제된 해당 페이지 프레임이 버디(옆 메모리)와 합쳐져 상위 order에서 관리될 수 있는지 확인하고, 가능하다면 order의 `nr_free`를 변경해준다.

- Slab¹⁵ allocator

프레임 페이지보다 작은 메모리를 요청한다면? 그래도 최소 1개의 페이지 프레임을 할당해줘야 하는 내부 단편화가 발생한다. 따라서 slab allocator가 도입되었다.

미리 몇 개의 페이지 프레임을 할당받아 자주 할당되는 크기로 분할해 둔다. 따라서 페이지 프레임보다 작은 메모리가 할당 요청을 받을 시 buddy가 아니라 slab에서 메모리를 할당해주고 해제받는다. 일종의 cache처럼 사용되는 메모리 관리 정책을 slab allocator라고 한다. Cache는 커널 내부에서 자주 할당되는 크기를 가지고 있고, cache가 여러 개의 slab으로 이루어져 있으며, slab은 다시 객체로 구성된다. Slab은 상태에 따라 free, partial, full로 구성되며, slab allocator에서 적당한 크기의 cache에서 객체를 할당해준다.

Cache를 효율적으로 관리하기 위해 `kmem_cache`라는 자료 구조가 정의되어 있다. 이 구조체는 각각의 cache가 가진 객체의 크기를 표현한다. Slab allocator로부터 객체를 할당받는 함수는 `kmem_cache_alloc()`이고, 해제하는 함수는 `kmem_cache_free()`다. Cache가 꽉 찼다면 slab allocator는 buddy allocator로부터 `kmem_cache_grow()`함수를 호출해 더 할당받는다.

하지만 slab은 많은 cache와 그에 따른 메타 데이터 때문에 필요 이상의 메모리를 필요로 하는 경우가 있다. 따라서 이를 보완하기 위해, Slub allocator가 개발되었다. Slub으로 할당된 객체들은 다음 같은 slub내의 다음 free 객체에 대한 포인터를 직접 포함한다. 또한 slub의 메타 데이터는 page 구조체 내부에 존재해, 별도의 메모리를 낭비하지 않는다.

¹⁵ <http://studyfoss.egloos.com/5332580>

또한 동일한 크기의 객체에 대한 cache는 통합하여 관리하기에 데이터 효율을 높였다. 하지만 page 구조체가 지저분해지는 단점이 존재했다.

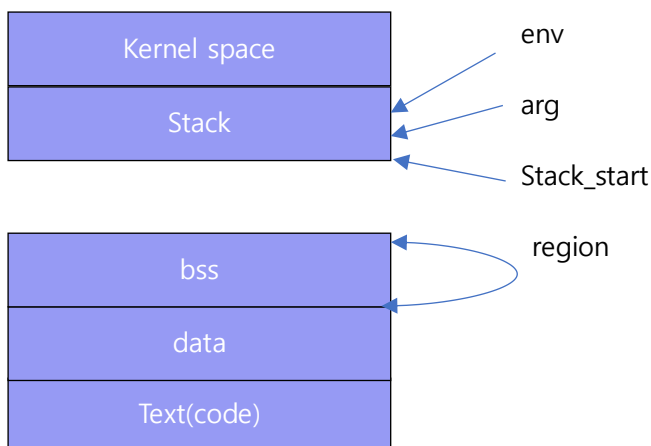
Slob은 slub의 overhead도 줄이기 위한 방법이다. (주로 메모리의 효율적인 사용이 중요한 곳에서 사용) 페이지를 작은 단위로 나눠, 이를 블록으로 묶어서 수행하는 방식이다. Slab이나 Slub과 다르게 특정 객체를 위한 것이 아니라 일정한 크기 내의 모든 할당 요청을 처리한다. First-fit¹⁶ 알고리즘을 사용한다. 구조체의 경우 slub과는 다르게 slob_page 구조체를 따로 만들고 page 구조체와 연결하여, 필요한 필드에만 접근할 수 있도록 하였다.

- Virtual memory

Task_struct 구조체에서 메모리 관련은 mm_field에 있다.

mm_struct는 region¹⁷을 관리하고 RBtree로 연결되어 있는 vm_area_struct, 주소 변환을 위한 pgd, 가상 메모리를 위한 변수들을 갖는다.

가상 메모리에서 변수들은 그림과 같이 0번지부터, start_code, end_code, start_data, end_data, start_brk, brk(heap 영역의 끝)로 구성되어 있으며, stack은 kernel의 아래부분부터, 환경변수와 초기인자 이후부터 아래로 커진다.



¹⁶ 페이지를 하나의 블록으로 구성한 뒤 요청받은 크기만큼 할당해주고, 나머지를 다시 새로운 블록으로 구성한다. 중간에 할당되었던 블록이 해제되면 다음 free블록까지의 정보를 0,1번 unit에 저장하여, free_block_list를 구성한다.

¹⁷ 가상 메모리 공간 중 속성이 같고 연속인 공간

가상 메모리 역시 page 가 모여, vm_area 를 구성하고, vm_area_struct 로 관리한다.

- vm_area(VMA)는 하나의 태스크에 여러 개가 존재할 수 있다. VMA 는 겹치지않으며, 서로 인접할 경우 하나로 생각해 처리할 수 있다.

- 물리 메모리와 가상 메모리의 변환

기본적으로 디스크의 파일 실행은 태스크를 할당하고 ELF 의 포맷 구조에 따라 정해진 부분은 물리 메모리에 할당해주고, 약속된 가상 주소에 나머지를 연결시켜주는 것이다. 리눅스 커널은 페이지들을 페이지 프레임에 적재할 때, page table 을 같이 만들어 주소 변환에 이용한다. 즉, sys_execve() 시스템 호출이 파일의 일부를 읽을 때, free 한 페이지 프레임들을 할당받고, page table 을 만들어, 가상 메모리와 연결한다. 그 후 태스크를 수행할 때 page table 을 따라 주소 변환을 실시한다.

가상 주소를 물리 주소로 변환하는 과정을 paging 이라고 한다. Paging 은 가상 주소 번지를 페이지의 크기로 나누고, 그 몫을 page table 의 entry 로, 나머지를 index 로 하여 물리 주소를 찾는다. 물리 메모리에 적재가 되지 않은 주소는 page fault 가 발생하여 page_fault_handler 가 호출된다. Handler 는 free 한 페이지 프레임을 할당받아 해당 프레임을 읽어 load 시킨다.

- 리눅스에서는 원활한 paging 작업을 위해 paging 을 3 단계로 나눴다.

가상 주소 공간을 이용하여 paging 을 하게 되는데, mm_struct 에서 PGD -> PMD -> PTE -> 실제 물리 메모리 주소의 과정을 거쳐 변환한다.

실제 CPU 의 경우 대부분 주소 변환을 지원하는 MMU 를 가지고 있다.

CH5

- File system

보조 기억 장치(HDD, SSD)를 관리하는 소프트웨어. 이름을 사용자에게 입력 받아, 데이터를 리턴해준다. Meta data 영역과 user 영역이 있는데 meta data 는 data 의 구조, 파일의 속성을 저장하고, 실제 데이터는 user 영역에 속한다.

- HDD

하드는 원판 모양의 트랙들과 트랙을 일정크기로 나눈 섹터로 구분된다. 섹터는 보통 512byte 이며 4KB 가 디스크 블록¹⁸의 크기이기 때문에 디스크 블록 하나는 8 개의 섹터로 이루어진다.

디스크에서 데이터에 접근하는 시간은 헤드를 요청한 곳까지 이동하는 탐색시간, 데이터가 있는 곳까지 원판이 돌아가는 회전 시간, 읽은 데이터를 전송하는 데이터 전송 시간으로 구성된다.

- 할당/해제

■ 연속 할당

불연속할당에 비해 파일을 읽는 속도가 빠르지만 파일의 크기가 변해, 디스크 블록을 더 할당받아야 한다면 불연속이 되는 단점이 있다. 따라서 연속 할당을 위해 다른 곳에 통째로 복사하는 경우는 성능이 저하되는 문제가 있다.

■ 불연속 할당

디스크 블록에 불연속적으로 저장하지만 데이터가 저장된 블록의 번호를 알 수 있는 정보를 기록해야 한다. 따라서, 블록체인 기법, 인덱스 블록 기법, FAT 기법 등을 사용한다.

1) 블록 체인 기법

각 블록에 포인터를 두어, 다음 블록과 연결해 놓는 방식이다. 하지만 블록 하나가 유실되면 모든 데이터를 잃는다는 단점과 끝부분의 데이터만 읽을 때도 모든 데이터를 거쳐야 한다는 단점이 있다.

¹⁸ Cluster라고도 불린다. 파일을 읽고 쓰는 최소 단위.

2) 인덱스 블록 기법

블록들에 대한 정보를 기록한 인덱스 블록을 따로 사용하는 방법이다. 원하는 데이터에 대해 바로 접근할 수 있다는 장점이 있지만, 인덱스 블록을 유실했을 때엔 모든 데이터를 유실할 수 있고, 파일이 커져 인덱스 블록의 크기가 한 블록보다 커질 때의 문제도 있다.

3) FAT 기법

같은 파일에 속해있는 블록들의 위치를 FAT 이라는 자료 구조 안에 저장하는 방식이다. 파일 하나하나마다 인덱스 블록이 있는 인덱스 기법과 달리 파일 시스템이 하나의 FAT 자료 구조를 가지고 있어, 모든 파일을 하나의 FAT 자료 구조로 관리하는 것이 특징이다.

- FAT 파일 시스템

FAT 파일 시스템에서는 meta data 를 FAT table, Directory entry, Super block 으로 구성된다.

FAT table 은 위에서 설명한 것과 같이 FAT 구조의 블록마다 다음 데이터의 블록 정보를 담고 있다. 데이터의 끝은 FF 로 쓰여진다.

Directory entry 는 각 파일마다 하나씩 갖는다. (msdos 파일 시스템에서) 리눅스에서는 파일과 디렉토리를 따로 구분하지 않기 때문에 Directory entry 가 모여 Directory 를 만든다. 즉, Directory 는 자신을 포함한 파일들의 이름을 저장하고 있는 특수한 파일이다. Directory entry 에는 파일이름과 확장자를 담는 name, 파일의 속성과 시간 정보(만들어진 시간, 수정된 시간 등등), 파일의 크기, FAT table 을 사용하기 위해 데이터의 첫 블록이 있는 table 번호인 start 등이 있다. 따라서 사용자가 파일의 name 을 변수로 파일 읽기를 하면 같은 name 을 가진 Directory entry 를 찾아, start 변수를 구해 FAT table 에서 데이터에 대한 정보를 받아오는 식으로 데이터 읽기가 진행된다. 이 과정에서 탐색하고자 하는 name 을 가진 Directory entry 를 쉽게 찾기 위해 CWD(Current Work Directory)나 절대 경로를 통해 읽는다.

예를 들어, /home/member/Nakkwan/nak.txt 라는 파일을 읽어올 경우, 파일 시스템은 /의 디렉토리 엔트리를 읽고, block 에서 home 이라는 블록의 번호를 찾아 이동한 후, home 에서 다시 nakkwan 의 번호를 찾은 다음 nak.txt 로 이동하는 방식이다. 앞서 말한 Directory entry 에 nak.txt 가 Directory 가 아니라 파일이라는 것을 알려주기 때문에 파일이라는 것을 알 수 있다. /의 Directory entry 번호는 파일 시스템이 최상위

디렉터리가 위치한 곳을 디스크 맨 앞부분에 저장하고 있기에 언제든지 알 수 있고 /하위의 모든 파일을 찾을 수 있다. 이를 Super block 이라고 부른다.

Super block 에는 sector 의 크기, cluster 의 크기, FAT table 의 위치, root_directory 의 위치 등이 담겨있다.

전체적으로 FAT 의 구조는 다음과 같다. (FAT32 기준)

Reserved Area	FAT Area	Data Area
---------------	----------	-----------

Reserved Area 는 3 부분으로 나뉘는데

Boot Sector	FS Information Sector	More Reserved Sector
-------------	-----------------------	----------------------

- 우선적으로 Boot Sector 는 FAT 파일 시스템에 대한 전체적인 구조와 BootStrapcode 가 들어있다.

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
Jump Boot Code			OEM Name								BPS		SP	RS	
FAT	RDE		Total 16		MT	FAT Size 16		SPT		HEAD		Hidden Sector			
Total 32			FAT Size 32				EXT Flags		FSV		Root Directory Cluster				
FSI		BRBS		Reserved											
DN	Rv	BS	Volume ID				Volume Label								
		File System Type													

파티션의 첫번째 섹터에 기록되는 Boot Sector 의 구조다. 중요한 것들만 설명하자면

- 1) Jump Boot Code: Boot Code 가 저장되어 있는 부분으로 점프하기 위한 곳
- 2) BPS(Bytes Per Sector): 섹터의 크기(ex) 512bytes)
- 3) SP: Sector Per Cluster
- 4) FAT: FAT 개수 (FAT table 이 유실되면 데이터 모두가 유실되기 때문에 여러 개를 저장한다.)
- 5) FAT Size 16, 32: FAT 의 크기를 Sector 의 개수로 표현

6) Root Directory Cluster: /가 있는 데이터 블록의 번호

7) FSI(File System Information): File System Information 구조체의 위치

Boot Sector 는 0 번째 Cluster 와 백업을 위한 6 번째 Cluster 에 저장된다.

- File System Information Sector 는 파일 시스템의 사용현황을 저장하는 Sector 로, 1 번째 Cluster 와 백업을 위한 7 번째 Cluster 에 저장된다.

- More Reserved Area 는 Boot Sector 의 코드 부분이 부족할 경우를 위한 추가적인 Sector 다. 2 번 Cluster 에 저장되고 8 번 Cluster 에 백업된다.

- FAT Area

FAT Area는 FAT 1과 FAT 2 두 부분으로 나뉜다. FAT 2 영역은 FAT 1이 손실되었을 때를 대비한 백업 영역이다.

FAT Area는 위에 설명한 것과 같이 데이터 영역에 저장된 파일들의 할당에 대한 정보를 포함한다.

- Data Area

Data Area 는 FAT 영역이 끝난 후 2 번째 Cluster 부터 시작된다. (Root Directory 의 번호가 대부분 2 번인 것도 같은 이유인 것 같다) 0 번째와 1 번째 Cluster 는 미디어의 타입, 파티션의 상태에 관한 Cluster 다.

- Inode 구조 (Inode descriptor)

Inode 구조는 ext 파일 시스템 계일이 쓰고 있다. Inode 는 FAT 과 비슷하게 파일에 대한 구조를 담고 있다. 파일이 가진 블록의 수, 속성, 접근 제어, 생성자, 소유자, 시간(만든, 수정 시간) 등을 담고 있다. 파일의 속성, 접근 제어를 나타내는 i_mode 필드에 대해 좀 더 알아보면, 16 비트로 구성되어 있다. 상위 4 개의 비트는 파일의 유형을 나타낸다.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
파일 유형				실행 플래그			사용자			그룹			다른 사용자		
				u	G	s	r	w	x	r	w	x	r	w	x

첫 4 개의 bit 는 파일의 유형을 나타낸다. 파일의 유형으로는 정규파일, 디렉터리, 문자 장치 파일, 블록 장치 파일, 링크 파일, 파이프, 소켓 등이 있다. 그 다음 3 개의 bit 는 실행 플래그로 쓰이고, 0~8 비트는 각각 사용자, 그룹, 다른 사용자에게 대한 읽기, 쓰기, 실행에 대한 접근 제어를 위해 쓰인다.

또한 이러한 필드들 다음 inode 의 하단부에는 i_blocks[15] 필드가 존재한다. 이 15 개의 entry 중 12개는 직접 블록, 3개는 간접 블록이다. 간접 블록이란 디스크 블록을 가르키는 포인터 블록이라는 뜻인데, 3 개의 간접 블록은 각각 단일, 이중, 삼중 간접 블록으로 나뉜다.

만약 원하는 파일의 offset 이 10000 이라면 어떤 entry 에 접근을 해야하는가? 데이터 블록의 크기가 4KB 라면 $10000 / 4KB$, 즉 3 번째 데이터 블록에 접근해야 한다. Offset 이 50000 일 경우에는 첫번째 간접 블록에 접근해야 한다.

왜 간접 블록을 쓰는 것인가? 직접 블록은 $12 * 4KB$, 즉 48KB 의 파일 크기 밖에 지원하지 못한다. 따라서 간접 블록을 통해 더 큰 파일에 대해서도 지원한다.

단일 간접 블록의 경우, 인덱스 블록은 다음 블록을 가르키는 포인터 블록으로 구성된다. 따라서 각 포인터가 4byte 가 필요하다면 하나의 디스크 블록에 1024 개의 포인터가 존재한다. 따라서 단일 간접 블록의 경우, $1024 * 4KB = 4MB$ 의 파일 크기를 지원할 수 있다. 마찬가지로 이중 간접 블록의 경우, 포인터를 가진 블록이 두개이기 때문에 4GB, 삼중 간접 블록의 경우 4TB 의 파일 크기를 지원할 수 있게 된다.

Inode 역시 Directory entry 로 파일 이름과 연결된다. Directory entry 에 파일의 name, inode 의 번호 (inode table 에서의 inode 번호를 의미) 등을 담고 있다.

FAT 구조와 inode 구조의 다른 점은 FAT 은 모든 파일을 하나의 Directory entry 에서 관리하지만 inode 는 파일마다 Directory entry 가 존재한다.

- Ext2 파일 시스템

파일 시스템은 디스크를 관리한다. 디스크가 장착되면 사용자가 partition 을 나눌 수 있다. 파일 시스템은 이 partition 당 하나씩 생성되게 된다.

파티션에 ext2 파일 시스템이 만들어지면 파티션은 복수개의 Block Group 으로 나뉜다. Ext2 는 디스크의 데이터 탐색 시간을 줄이기 위해 인접한 실린더를 block group 으로 묶는다. 따라서 구축된 파일 시스템은 Boot block 과 block group 들로 이루어지는데 block group 은 Super block, block group descriptor, block bitmap, inode bitmap, inode table, data block 등으로 이루어 진다.

1) Super block

각 block group 의 첫번째에 위치한다. 해당 파일 시스템에 관련된 정보를 저장한다. (블록의 크기, 총 블록의 개수, 블록 그룹의 개수, inode 개수, 그룹 내의 inode/block 의 개수)

Super block 이 손상되면 전체 파일 시스템에 대한 정보를 잃기 때문에 Super block 은 모든 Block group 에 저장한다.

2) Group Descriptor Table

Super block 다음에 위치한다. 각 Block group 에 대한 정보를 저장한다. (Block bitmap 의 블록 번호, inode bitmap 의 블록 번호, 첫 번째 inode table block 의 번호, 그룹안의 빈 블록수, inode 수 등등) 마찬가지로 손상되면 안되기 때문에 모든 block group 에 저장된다.

3) Block Bitmap

그룹 내의 각 블록의 사용여부를 나타낸다. 4KB 가 블록의 default 면 32KB 개의 블록의 사용여부를 표시가능. 빈 공간을 관리하기 위해 사용한다. Bitmap 은 한 블록 내에만 존재해야 한다. 따라서 블록의 크기가 4KB 인 파일 시스템은 Block Group 당 $4096 * 8$ 개 이상의 블록을 가질 수 없다.

4) Inode Bitmap

그룹 내의 각 inode 의 사용여부를 나타낸다. 역시 빈 공간을 관리하기 위해 사용한다.

5) Inode Table

Inode 의 정보를 나타내는 [inode descriptor](#) 로 구성됨

6) Data block

실제로 데이터가 저장되어 있는 block

Super Block(1 개)	Group Descriptor(K 개)	Datablock Bitmap(1 개)	Inode Bitmap(1 개)	Inode Table(N 개)	Data Blocks(M 개)
------------------	-----------------------	-----------------------	-------------------	------------------	------------------

만약 파일을 읽어온다면

- 1) Group descriptor 에서 root Dir 의 inode 번호와, Inode table 의 위치를 받아와 Root Dir 의 Inode descriptor 에 접근한다.
- 2) Root Dir 의 data 가 저장되어 있는 데이터 위치를 읽어, 데이터에 접근한다.
- 3) Data 에서 원하는 파일이 있는 Dir 의 Inode, 혹은 원하는 파일이 있다면 그 파일의 Inode 번호를 읽는다.
- 4) Inode table 에서 파일/Dir 의 Inode descriptor 의 위치를 찾아 접근한다.

- Ext3, Ext4

Ext3 와 ext4 는 ext2 의 성능을 향상시킨 것이다.

■ Ext3

- 1) 단순 연결 리스트로 Dir 를 연결한 ext2 와 달리 Hash 기반 Htree 로 구성해 빠른 Dir 탐색을 가능하게 했다.
- 2) Journaling 기술을 도입해, 읽기/쓰기 도중 오류를 허용할 수 있도록 했다. Journal 은 write transaction 이 일어나는 영역을 따로 할당했다. Transaction 이 디스크에 쓰는 것을 완료하면 journal 안의 데이터를 commit 하고, 해당 작업이 commit 되기 전에 손상을 입는다면 새로 재부팅된 시스템은 journal 영역만을 엮어, 기존 파일의 손상을 보호했다. Journaling 에는 세가지 모드가 있다.
 - ◆ Journal: 위험성이 가장 낮지만 성능이 크게 저하될 수 있는 모드다. 메타 데이터와 파일 데이터를 모두 저널에 기록한 후 파일 시스템이 commit 한다.
 - ◆ Ordered: 메타 데이터는 저널에 기록하지만 파일 데이터는 파일 시스템에 직접 쓴다. Ordered 라는 이름처럼 순서가 있는데, 첫째, meta data 를 journal 에 기록한다. 둘째, data 를 파일 시스템에 쓴다. 셋째, journal 에 있던 meta data 가 파일 시스템에 쓰인다. 기록 도중 쓰이고 있던 파일 데이터는 보존 못하지만 이미 쓰인 파일에 대해서는 보호할 수 있다.

◆ Writeback: ordered 처럼 meta data 는 journal 에 쓰이지만 파일 시스템에 기록되는 순서는 없다. 파일 시스템에 대한 보호만 가능하여 쓰고 있던 data 는 물론, 이전에 기록된 data 의 보호에도 취약하다.

- 3) Online-resizing 이 가능하다. Ext2 에서는 한번 파일 시스템이 구축되면, 파티션의 용량을 늘이거나 줄일 수 없었다. 하지만 ext3 에서는 LVM¹⁹ (Logical Volume Manager)을 통해, 파일 시스템이 관리하는 용량을 동적으로 늘이고 줄일 수 있다.
- 4) Ext2 구조의 파일을 손상없이 ext3 로 바꿀 수 있다.

■ Ext4

- 1) 파일 시스템의 일관성과 속도 향상을 위해 preallocation 기법을 도입했다. 미리 어느 정도의 블록을 할당해 줌으로써 되도록 연속적인 블록을 할당하도록 했다. 먼저 할당된 블록의 크기가 모자라면 extent 라는 단위로 할당하여, 파일 블록들은 위치와 블록 수, 다음 extent 의 포인터로 구성된다.
- 2) Multiblock allocation 이 가능하게 했다. 이전에는 256 개의 블록을 할당한다 하면, 256 번의 블록 할당을 호출했어야 하지만 ext4에서는 한번의 호출로 많은 블록을 할당할 수 있게 되었다.
- 3) 단편화 방지를 위해 Delayed allocation 기법을 도입했다. File data 가 journal 에 있을때부터 블록을 할당하던 이전 버전은 file data 가 계속해서 커지면 파일이 계속 써질 때, data 가 커진다는 것을 알지 못하는 단점이 있다. 따라서 delayed allocation 은 journal 의 data 를 commit 할 때 할당을 한다.
- 4) 대용량 파일의 meta data 를 줄일 수 있는 extent 기반 데이터 블록 유지다. 데이터 블록의 개수가 많은 파일의 경우 inode 에 모드 블록의 번호를 써줘야한다. 하지만 ext4 에서는 (500, 50)같은 방법으로 500 번부터 연속된 50 개의 블록 같은 형식을 지원해, meta data 의 관리를 용이하게 해주었다.
- 5) 하위 디렉터리가 32000 개로 제한되던 이전 버전과 달리 ext4 는 하위 디렉터리의 개수에 제한이 없어졌다.
- 6) Journaling Checksum 기능을 지원한다. 파일 시스템이 깨지거나 디스크에 문제가 생겼을 경우 data 나 journal 항목들을 롤백하여 손상을 방지해준다.

¹⁹ <https://tech.cloud.nongshim.co.kr/2018/11/23/lvmlogical-volume-manager-1-%EA%B0%9C%EB%85%90/>

7) Online-defragmentation 을 지원한다.

8) Ext2, ext3 구조의 파일을 손상없이 ext4 로 바꿀 수 있다.

- Virtual File System

User space 에서 open(), read(), write() 같은 함수로 파일 시스템에 접근하려하면, 사용자가 원하는 일을 해줄 수 있는 파일 시스템의 고유 함수를 호출해준다. 운영체제는 같지만 파일 시스템 체계가 다른 경우가 있어, 같은 동작에 호출되는 함수가 파일 시스템마다 다르기 때문이다.

b.txt 파일을 읽어온다고 하면, VFS 는 b.txt 파일이 어느 파일 시스템에 속해 있는지 판단하고, 정보를 담은 객체를 생성한 후 파일 시스템에 맞는 open() 함수 호출을 한다. 그러면 파일 시스템은 종류에 따라 FAT, inode 등의 정보를 객체에 넘긴다. 따라서 사용자는 파일 시스템에 상관없이 하나의 호출 함수를 통해 파일에 접근할 수 있게 된다.

VFS 에는 4 개의 객체가 존재한다.

- 1) Super block: 가지고 있는 파일 시스템마다 하나씩 주어져서, 각 파일 시스템의 고유한 특성을 저장해두고 VFS 가 관리한다.
- 2) Inode: VFS 는 inode 객체를 생성하고, 파일 시스템에 파일 대한 특정 정보를 요청하면 메타 데이터를 읽어 객체에 넘겨준다. msdos 라면 디렉터리 엔트리를, ext2 라면 inode 와 디렉터리 엔트리를 넘길것이다.
- 3) File: 태스크가 open 한 파일에 관한 정보를 담는다. 하나의 파일에 여러 태스크가 접근한다면 inode 객체는 하나겠지만 접근한 태스크들에 대한 정보는 달라야한다. 따라서 파일과 태스크가 연관된 정보에 관한 것을 저장한 것이 file 객체다.
- 4) 디엔트리: inode 객체와 file 객체를 더 빠르게 연결하기 위한 cache 역할을 하는 객체다.

APPENDIX

A. Difference of kernel space, user space

RAM 은 User space 와 Kernel space 두 부분으로 나뉘어 있다.

User space 는 일반적인 사용자 프로세스(프로그램)가 돌아가는 영역이다. 프로세스는 커널에 직접적으로 접근할 수 없다.

하지만 Kernel space 에서는 System call 을 호출해 kernel 에 접근할 수 있다. System call 은 kernel 에서 software interrupt 처럼 동작한다.

커널은 메모리의 일부분에서 실행되는데 User space 의 application/process 를 관리하는 역할을 해준다. System call 을 통해 interrupt 가 Kernel 에 보내지면 적당한 interrupt handler 가 역할을 수행한다.

B.Type of Memory, RAM

- ROM: Read Only Memory 로써, 한번 쓰면 수정이 불가능하다. 하지만 읽는 속도가 빠르기 때문에 바뀌지 않고 지워지면 안되는 곳에 사용한다.
 - EPROM: ROM 과는 다르게 프로그래밍이 가능하지만(자외선을 이용해서 지움), 내용을 삭제하려면 칩을 빼줘야 한다.
 - EEPROM: EPROM과는 다르게 보드에서 빼지 않고도 삭제가 block단위로 가능하다. 하지만 용량이 작고 느리다
-
- SRAM(Static RAM): transistor가 6개가 사용되는 flip-flop에 bit를 저장한다. 전원이 유지되면 내용이 유지되며, 집적도가 DRAM에 비해 떨어지지만 속도가 빨라, cache에 사용된다.
 - DRAM(Dynamic memory): transistor 1개와 축전기 1개로 구성되어있으며, 축전기에 전하의 형태로 bit가 저장된다. 집적도가 SRAM에 비해 뛰어나지만, 축전기에 전하가 방전이 되기 때문에 주기적으로 refresh를 해주어야 한다.
-
- Flash Memory: Cell transistor라고 불리는 MOSFET(정확히는 MOS의 floating gate)에 전하를 축적하여 데이터를 보존한다. 축적된 전하는 산화막에 갇혀 비휘발성을 지닌다.
 - NAND: 직렬 연결 방식이고, 집적도가 높아, 대용량으로 만들기 쉽다. 읽을 때 Random Access가 불가능하기 때문에 읽기속도에 비해 쓰기/지우기 속도가 빠르다.

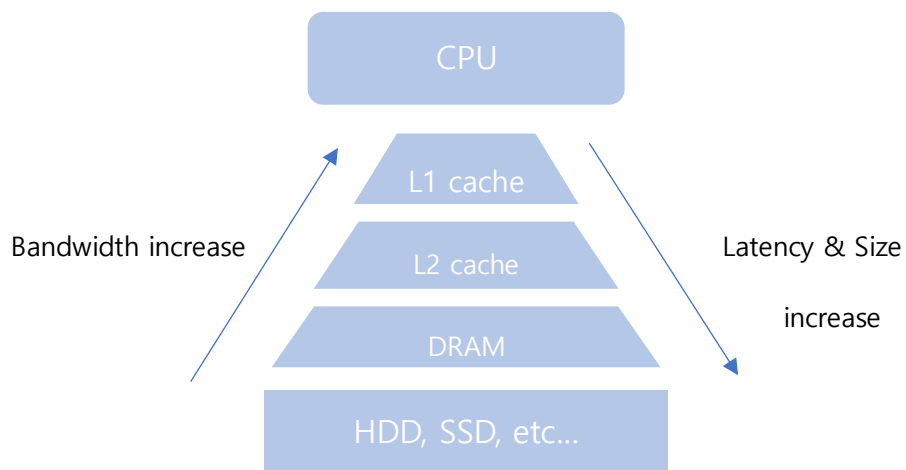
직렬 연결 방식이기 때문에 지우거나 쓸 때 block 단위로 수행을 해야하며, 한 부분이라도 고장이 나면 전체를 바꿔야하는 매커니즘이 NAND와 비슷해, NAND FLASH MEMORY라고 불린다.
 - NOR: 병렬 연결 방식이고, 집적도가 낮다. 읽기속도는 빠르나, 쓰기/지우기를 할 때, 속도가 느리다. 프로그램의 bit화가 가능하기에 NOR FLASH MEMORY라고 불린다.

C. Memory Access in hardware

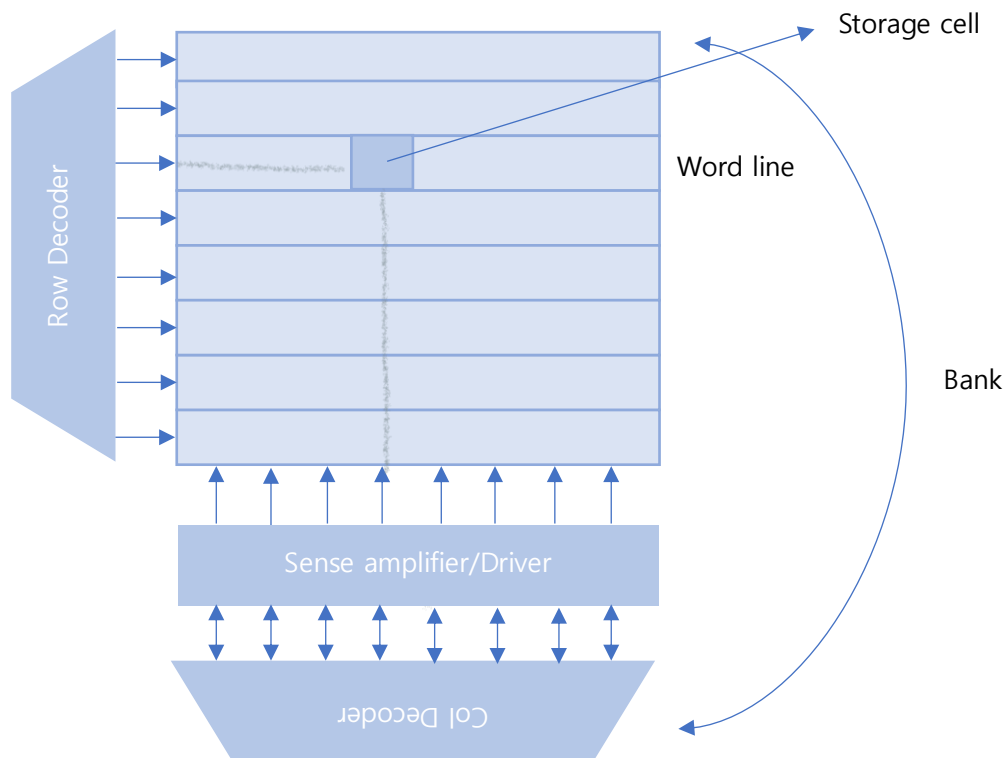
메모리에서 locality(접근성)는 크게 두가지로 나뉜다.

- 1) Temporal(시간적 접근성): 최근에 쓴 데이터를 다시 쓸 확률이 높다.
- 2) Spatial(공간적 접근성): 인접해 있는 데이터에 접근할 확률이 높다.

또한 CPU에서 메모리를 참조할 때의 관계는



RAM architecture에서 CPU의 RAM으로의 데이터 추출 과정은 다음과 같다.



- 1) CPU가 메인보드에 데이터 요청
- 2) Chip set이 데이터의 행 주소를 메모리에 전송
- 3) 메모리의 행 주소 버퍼에 주소가 들어오면 sense amp가 행의 모든 cell을 읽음 (RAS, Row Access Strobe)
- 4) 열 주소를 버퍼에 받고 해당 cell을 읽음 (Column Access Strobe)
- 5) 읽은 데이터를 출력 버퍼에 load
- 6) 메인보드의 chip set이 내용을 읽고 CPU에 전달

따라서 위해 이전의 데이터 같은 행에 있는 데이터를 요청할 시 RAS-CAS latency가 줄어들어, 데이터를 읽는 속도가 더 빠르다. (CAS과정만 하면 되기 때문이다.)