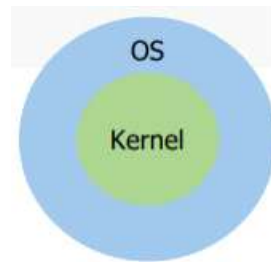


13. Process Management

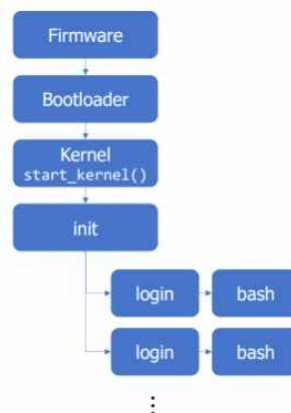
● Linux Kernel

- Kernel : CPU Kernel Mode에서 실행되는 OS의 핵심.
 - => OS 부팅 시 Kernel 실행 : 하나의 실행파일.
- ✓ In Windows -> C:\Windows\System32\ntoskrnl.exe (Kernel 실행파일)
- ✓ In Linux -> /boot/vmlinuz (Kernel 실행파일 : binary 실행파일)
 - => start_kernel() in init/main.c : Linux Kernel을 처음 시작하는 function.
main function을 대체함. (main)



● OS Initialization

- Firmware : ROM에 저장된 기본 하드웨어 초기화 code.
 - => 컴퓨터 부팅 시 최초로 실행되는 파일. (ROM에 구워져 있음)
 - => ex) BIOS and UEFI in PC
- Bootloader : Kernel 파일(/boot/vmlinuz)을 RAM에 Load.
 - => Firmware가 호출함. Kernel 파일을 읽어서 Memory에 copy하여 넣어줌.
- Kernel : 하드웨어를 초기화해주고, 'init process'를 만들어줌,
 - => 이후 Background에서 system calls을 호출하면 대응한다. (In address space)
 - => 더 이상 active하게 실행되지 않고, init process가 실행해줌.
- init (or systemd) : 최초의 process이자 user level의 process.
 - => 처음 실행되는 user level process. (Not kernel level)
- User-level boot procedure : login process, shell process 등을 만들.
 - => init process가 만들어준다.



● Program and Process

- Program : instruction들과 data로 이루어진 disk에 존재하는 생명력 없는 실행파일.
- Process : Program에 대해 isolate한 실행 환경을 제공하는 OS가 관리하는 entity
 - * Firmware 실행은 firmware가 하나의 program이기 때문에 process로 간주하지 않음. Process는 multitasking OS에서만 의미가 있음.
 - => Program이 여러 개가 동시에 실행되기 때문에 보안, 간섭의 위험. Program 사이의 간섭을 없애주고 보안을 지켜줌. (Program 실행 시 필요한 그릇)
 - => OS는 여러 개의 Process를 가지고 있음.

● Shell

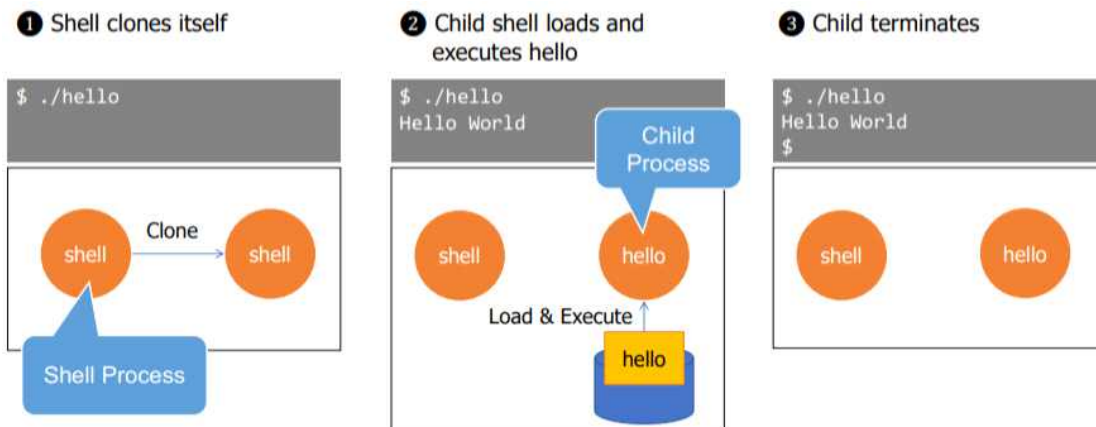
- command를 accept하고 실행하는 하나의 Special application
 - => 명령을 받아서 명령을 실행
- BASH : Linux의 standard shell.
 - jongchank@desktop:~/caros\$
 - > jongchank : username
 - > desktop : hostname
 - > ~/caros : current working directory
 - > \$: normal user, # : super user(admin)(관리자 모드)
 - > 'sudo'를 사용하여 program을 관리자 모드로 실행.
- (1) Built-in command(내장) : shell이 스스로 이해하여 실행하는 command (ex> exit)
- (2) External command(외장) : 외부의 경로에 존재하는 external program을 load하여 실행하는 command(실행을 위한 중간 매개체 역할)
 - ✓ ex> ls => Located in /usr/bin/ls)

● Executing External Programs in Shell

- 실행파일의 정확한 위치를 지정해야 함.
 - => ./hello: Current working directory에 존재하는 hello라는 file을 실행 ('.' - Current working directory)
 - => ../bye: Parent directory에 존재하는 bye라는 file을 실행 ('..' - Parent directory)
- \$PATH : command 실행 시 shell이 검색하게 되는 directory들
 - => ex) ls -> command를 실행하게 되면 \$PATH에 존재하는 경로들을 탐색하며 실행파일을 찾음. 존재하면 실행이 됨.
 - => which ls -> ls의 실행파일이 존재하는 경로를 보여줌.
 - => echo \$PATH로 탐색할 모든 경로를 확인할 수 있음.
 - => export PATH = \$PATH:/your/own/directory로 탐색할 경로를 custom으로 추가할 수 있음. (추가한 경로에 동일한 이름의 실행파일이 존재한다면 기존 기능을 못하게 될 수 있음.)

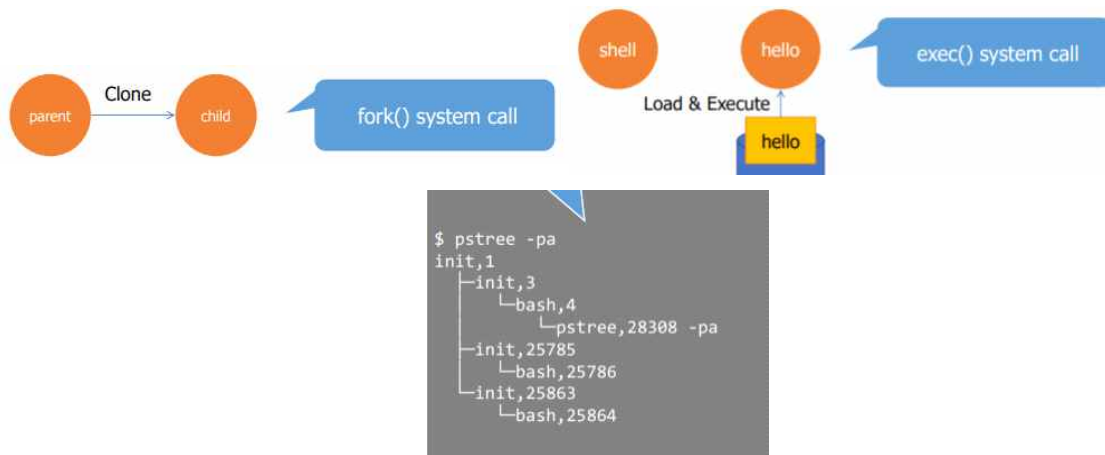
● When you type hello in shell (Linux에서 Program을 실행시키는 유일한 방법)

- (1) Shell이 자기 스스로를 복제함.(Shell이 하나 더 생김.)
 - (2) 복제된 Shell(child process)가 execution file(./hello)을 disk에서 Load하여 자기 자신 위에 overwrite함. 복제된 shell(child process)가 execution file(./hello)로 바뀐.
 - (3) execution file의 main function의 처음부터 실행됨.
 - (4) execution file을 가졌던 복제된 shell(child process)이 없어지게 되며 실행을 종료.
- * Process는 Program과 독립적이다.



● How to create a new process (새로운 process가 만들어지는 과정)

- fork()라는 system call을 통하여 process를 복제하여 child process를 생성한다.
- exec()라는 system call을 통하여 복제된 process가 스스로에게 program을 Loading하여 execution file을 실행시킨다.
 - => OS kernel이 직접 service한다.
 - => Linux에서 Program을 실행시키는 방법이 Process를 복제하는 방법뿐이기 때문에 Process끼리 Tree구조로 파생되게 된다.(Parent-Child구조)

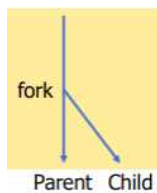


● fork and exec Example

```
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    char *arg[] = {"/bin/ls", "-l", NULL}; => ls command를 실행
    pid_t pid;
    pid = fork(); => shell이 스스로를 복제하여 child process를 만들.
    if (pid == 0) => pid가 0이면 child (복제된 process가 걸리게 됨) (Program 실행)
    {
        /* child */
        execve("/bin/ls", arg, NULL); => 복제된 shell이 스스로의 id를 지우고 자기 자신
                                   에게 program을 올려줌
    }
    else { /* parent */ } => pid가 0이 아니면 parent (원본 process)
    return 0;
}
=> fork()와 exec()가 같은 code에서 실행되기 때문에 process는 결론적으로 2개가 실행되게
됨.
```

● fork System Call

- 호출한 process를 clone하여 new process를 생성(child process)
 - => #include <unistd.h>; pid_t fork(void);
 - => pid_t (return value) : -1 -> Fail (clone fail)
 - 0 -> child process
 - >0 -> parent process (clone한 child process의 수)



=> Process가 복제되며 code가 두 개가 실행됨.

● execve System Call

- filename(실행파일 이름)을 통해 새로운 Program을 실행시킴.
 - => #include <unistd.h>
 - int execve(const char *filename, char *const argv[], char *const envp[]);
 - > *filename : 실행파일 이름
 - > argv[(argument vector) : Program arguments의 문자형 Array.
 - NULL을 만나면 vector read 종료.
 - 보통 argv[0]은 program의 name을 표현.
 - > envp[(environment pointer) : Program의 경로 등이 포함된 환경변수.
 - (ex> user, path, pid...) NULL을 만나면 pointer의 read 종료.

● argc, argv, and envp

- int main(int argc, char *argv[], char *envp[]) { ... }
 - > argc : Argument의 갯수
 - > argv[] : Argument vector(array)
 - > envp[] : NULL 문자열로 끝나는 환경 변수의 array
- \$ export MYENV=itsme => 새로운 환경 변수를 추가해줌. (Shell이 알고 있는 변수)
- \$./argc_envp first second => List of arguments
(argv[0] : ./argc_envp(Program name) // argv[1] : first // argv[2] : second)

unique (sequential) number for each process ↘

● getpid System Call

- PID(Process ID) : 각 process의 고유한 번호.
 - getpid : 호출한 process의 PID를 반환해줌. (자기 자신의 PID)
 - getppid : 호출한 process의 parent process의 PID를 반환해줌. (자신의 부모 PID)
(init process : 1번 PID - OS 초기화 시 할당받음, kernel process : 0번 PID(상징적))
- ```
#include <sys/types.h>; #include <unistd.h>
pid_t getpid(void); pid_t getppid(void);
```
- man getpid : system call에 대한 manual을 보여줌. (getpid에 대한 동작 과정 등)

## ● wait System Call

- child process의 상태가 변경될 때까지 기다림(ex> termination)
- ```
=> #include <sys/types.h>  
#include <sys/wait.h>  
pid_t wait(int *status); pid_t waitpid(pid_t pid, int *status, int options);
```
- child process의 exit status code를 추출함. (child process가 끝날 때 exit status를 parent에 날림. parent는 child의 exit status를 받을 때까지 wait한다.)
 - parent process가 child process가 exit되면 status를 wait()로 확인 후 OS에 요청하면 Process table에서 child process를 제거함.
- * **zombie process(defunct)** : Parent가 child를 만들고 실행 후 child가 먼저 종료되게 되면 parent가 wait()하고 있지 않아 child의 exit status code를 알지 못하고 OS는 해당 process가 끝났다고 알지 못하기 때문에 process가 종료되어도 process table에 남아있게 됨. (wait를 하지 않아 child의 exit status를 확인하지 못할 경우)
- => fork + exec + wait가 모두 실행되어야 zombie process가 발생하지 않음.

● Program Exit Status Code

- main() function의 code가 모두 실행된 후 return되는 value
 - > 0 : Success (정상종료), 1~255 : others (비정상 종료) (return 0)
- process가 종료되며 parent에게 알리기 위해 사용함.(exit status)
- return한 exit code는 \$?에서 받음.

● Process Management Commands

- \$./hello : program으로부터 foreground process를 생성하여 실행함.
-> 실행되는 동안 Terminal이 막혀 해당 Terminal에서 다른 작업을 못함.
- ctrl-c : foreground process를 종료시킴.
- \$ ps -ef : system에 존재하는 모든 process를 보여줌.
- \$ pstree -pa : process의 tree 구조를 모두 보여줌.
- \$ top : 가장 많은 일을 하고 있는 process들을 보여줌.
- \$ kill [-9] : 실행 중인 process를 죽임(다른 곳에서 원격으로도 가능함.)

● Background Process Management Commands

- \$./hello & : program으로부터 background process를 생성하여 실행함.
-> 실행되는 동안 Terminal을 막고 있지 않아 다른 작업 가능.
- ctrl-z : background에서 실행되고 있는 process를 잠시 멈춤.(종료 X)
- ✓ bg : ctrl-z로 잠시 멈추었던 program을 다시 실행함.
- ✓ jobs : background에서 실행되고 있는 process들을 보여줌.
- fg : background로 실행되고 있는 process들을 foreground로 올림.

● ps and pstree

```
Process ID  Parent Process ID
$ ps -ef
UID        PID     PPID  C  STIME TTY          TIME CMD
root         1         0  0   Apr17 ?        00:00:00 /init
root         3         1  0   Apr17 tty1      00:00:00 /init
jongcha+    4         3  0   Apr17 tty1      00:00:01 -bash
root    25785         1  0   Apr28 tty2      00:00:00 /init
jongcha+  25786    25785  0   Apr28 tty2      00:00:00 -bash
root    25863         1  0   Apr28 tty3      00:00:00 /init
jongcha+  25864    25863  0   Apr28 tty3      00:00:00 -bash
jongcha+  28307         4  0  23:57 tty1      00:00:00 ps -ef
$ pstree -pa
init,1
├─init,3
│   └─bash,4
│       └─pstree,28308 -pa
├─init,25785
│   └─bash,25786
├─init,25863
│   └─bash,25864
```

- init with PID 1 is the only handcrafted process created during the OS initialization
- When a parent process terminates, its child processes are adopted by init (called "orphan processes")

- => parent process가 먼저 죽으면 child process는 init의 child로 붙게 됨.
-> orphan processes

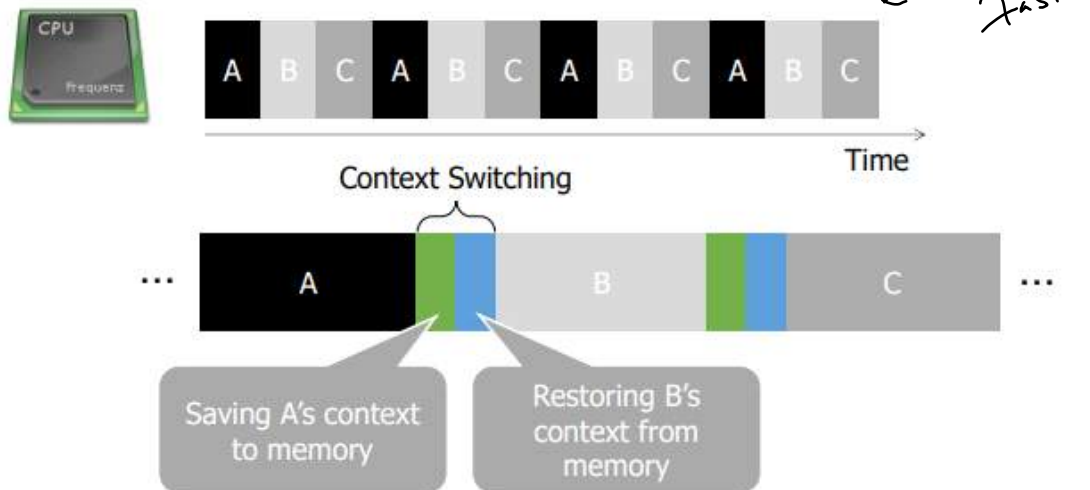
14. Process Scheduling

● Important Concepts

- Multiprogramming : single CPU에서 context switching을 통해 여러 program이 실행됨. (여러 Program을 번갈아가면서 올리며 실행시킴.)
- Multitasking : 여러 작업들이 동시에 시간을 공유하며 실행됨. (짧은 시간 동안 다른 Program을 왔다갔다하며 실행하며 마치 동시에 실행되는 것처럼 사용자에게 illusion을 줌.)
- Multiprocessing or Multiprocessor (vs Uniprocessor)
 - > 하나의 computer에서 여러개의 CPU를 활용한다.
 - > 최근에 Multicore system이 더 많이 사용됨.
- Multithreading : process에서 여러 개의 thread가 실행됨.

● Context Switching

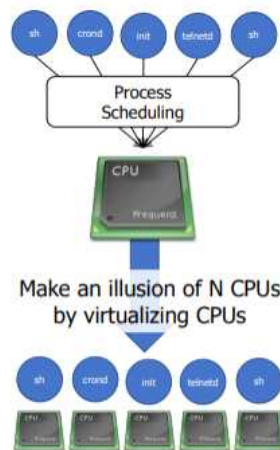
- Context : 특정 시점에서의 CPU의 모든 내부 상태.
 - 일부 CPU는 context의 saving과 restoring에 대한 instruction을 제공함.
 - context switching은 SW module로 구현될 수 있음.
- => Program을 동시에 실행시키는 것처럼 switching하며 실행.



- => context switching되는 순간 잠시 짧은 시간 동안 멈추는 데 멈췄다고 인지하지 못함.
- => 그림에서 표시된 구간에서 A는 memory에 save되고 memory에 save되어 있던 B를 restore하여 CPU에 overwrite하여줌. 해당 구간에서 잠시 멈춰있음을 인지하지 못하기 때문에 마치 B가 계속 실행되었던 것처럼 illusion을 느낌.
- => HW적으로 제공해준다.

● Process Scheduling

- Process는 여러 개지만 CPU가 한 개일 경우 process의 scheduling이 필요하다.
- Scheduling의 type
 - > Real-time scheduling : scheduling의 시작과 끝이 명확하게 존재한다.
 - > Non-real-time scheduling : 각각의 process가 각각의 CPU를 가지고 있는 듯 한 illusion을 느끼게 만들어줌. (범용적인 OS)
- Linux scheduling algorithm
 - > Time-sharing and round-robin (순차적으로 반복하여 실행 : 시간을 공유하는 illusion)
 - > CFS (Completely Fair Scheduler) : 모든 활성화된 process는 CPU의 bandwidth를 공평하게 나눠 가진다. (효율적으로 분배)



● Workload Types

- CPU-bound workload (무한 loop를 돌며 CPU를 계속 사용함.)
 - > 적은 I/O, 많은 연산량, wait하는 동안 CPU 계속 사용, Throughput이 중요함.
- I/O-bound workload (CPU를 계속 사용하지 않아 놓고 있음.)
 - > 많은 blocking I/O, 적은 연산량, wait하는 동안 CPU 사용 X, 응답(Latency)이 중요함.

```
while (1) {
    a = b + c;
    c = a / 2;
    ...
    z = x + 3;
}
```

- CPU-bound

```
while (1) {
    waitForKeyboard(&key);
    printOut(key);
}
```

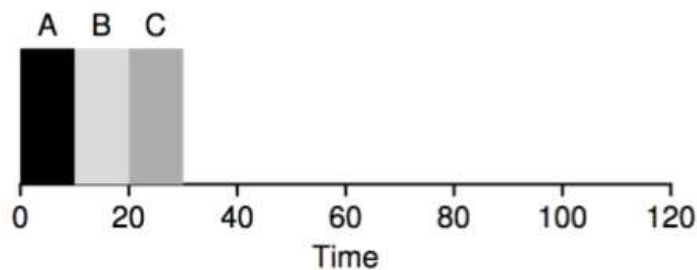
- I/O-bound

● Scheduling Algorithm or Policy (CPU를 어떻게 나눠줄 것인가)

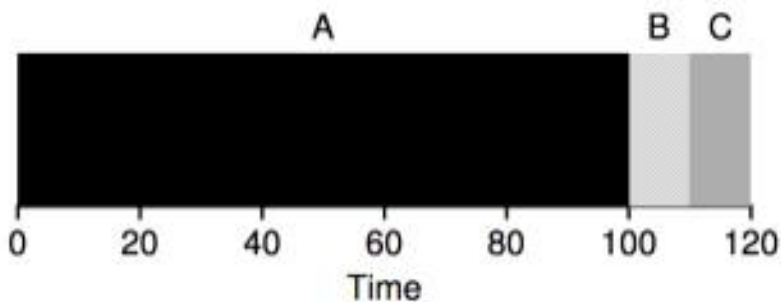
- Fairness : 각각의 process마다 공평하게 나눠줘야 한다.
- No starvation : 모든 process는 CPU에 못 올라가서 죽는 일이 없도록 CPU를 분배해야 한다.
- High throughput : 시간 당 최대한 많은 instruction을 처리해야 한다.
- High utilization or efficiency : 100%의 효율성을 가져야 한다. (모바일은 발열 문제로 예외)
- Small latency or response time : input-output에 대한 응답 delay가 없어야 함.
- High Interactivity : user의 행동에 따른 delay가 없어야 함.
- Deadline guarantee : 작업의 마지막이 약속되어야 함.
- Predictability : Scheduler가 예측 가능하게 동작해야 함.
- Flexibility : workload가 바뀌어도 반응이 잘 되어야 함.

● FIFO (First In, First Out)

- workload가 도착하는 대로 실행 시킨다.

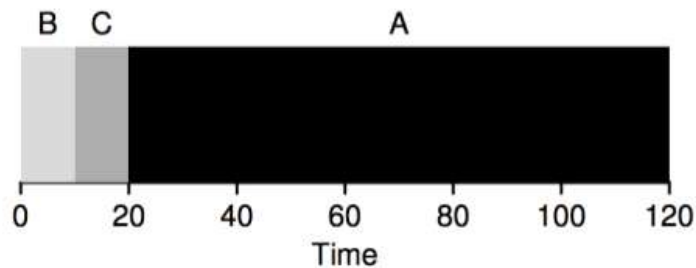


- B와 C는 매우 짧게 CPU를 사용 후 빠지면 되는데 A가 오래 걸려서 B와 C를 빠르게 처리하지 못해 B와 C의 처리에 delay가 생긴.

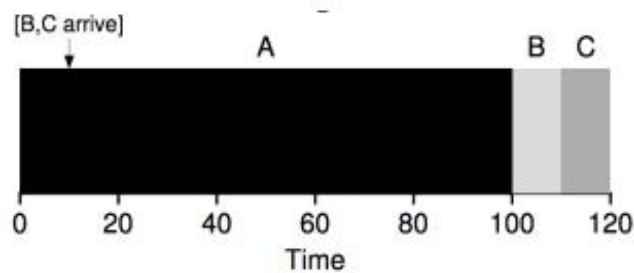


● SJF(Shortest Job First)

- workload의 길이가 짧은 것들을 먼저 처리함.

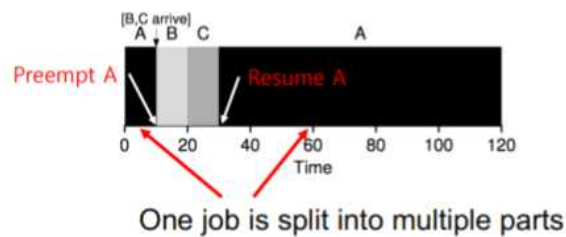


- A가 먼저 도착하여 처리되는 도중에 B와 C가 도착하면 FIFO와 같은 문제가 발생함.



● STCF(Shortest Time-to-Completion First)

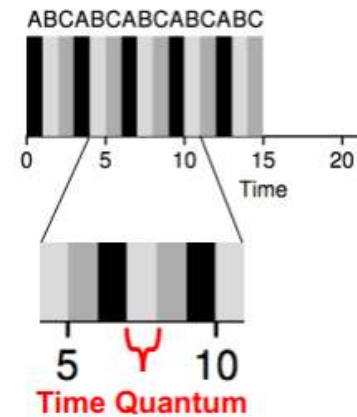
- 각 process의 남은 시간이 얼마인가를 기준으로 time을 비교함.
-> 남아있는 시간이 짧은 것을 먼저 올림.
- OS는 실행중인 process를 preempt(강제로 선점)해야 함.
- latency에 적합함.
- Program의 실행 시간과 남은 시간을 모두 알고 있어야 함으로 현실적으로 구현이 매우 힘들.



- > 위의 그림에서 B와 C가 도착하였을 때 A의 남은 시간이 더 길기 때문에 preempt하여 Memory에 save하고 B, C를 처리하고 A를 다시 Restore하여 재개함.

● RR (Round Robin)

- 짧은 순간 동안 순차적으로 process들을 실행함.
- Time Quantum : process 하나가 중단되기 전까지의 최대 실행 시간.
: 보통 수십 밀리초 이하이다.
- > Too small : context switching만 진행하다가 정작 처리해야 할 일을 못함.
- > Too large : 동시에 실행되는 듯한 illusion을 제공하지 못해 delay가 존재하는 것처럼 보임.



● I/O Types

- Non-blocking I/O
 - > Do not wait(읽을게 없어도 대기 상태에서 계속 loop를 돌음).
 - > CPU-bound(CPU를 태움)
 - > 자발적으로 context switching이 일어나지 않음. (CPU를 강제로 선점) (preempt)
- Blocking I/O
 - > I/O event가 일어날 때까지 기다림.(loop를 돌지 않음)
 - > I/O-bound(CPU를 태우지 않음)
 - > 자발적으로 context switching이 일어남. (자발적으로 CPU를 놓음) (yield)

```
while (1) {
    ret = readSensor(&data)
    if (ret == NODATA)
        continue;
    ... /* process data */
}
```

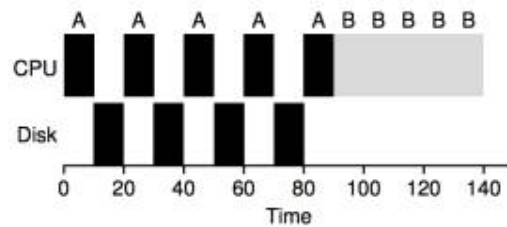
- Non-blocking I/O

```
while (1) {
    waitForSensor(&data);
    ... /* process data */
}
```

- Blocking I/O

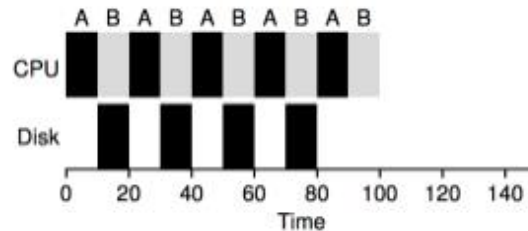
● Blocking I/O and Scheduling

- blocking이 되는 동안 CPU가 필요하지 않음.



-> 중간에 CPU를 자발적으로 놓아 disk에 Save, CPU에 Load

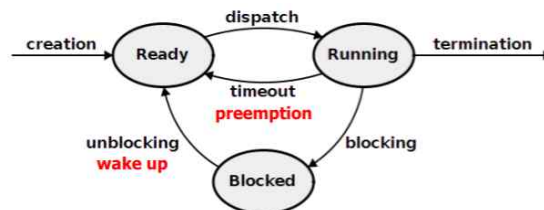
- 다른 process를 실행하기 위해 자발적으로 CPU를 놓음. (yield)



-> 자발적으로 CPU를 놓은 시간 동안 다른 process가 실행. (Not preempt)

● Process State Transition Diagram

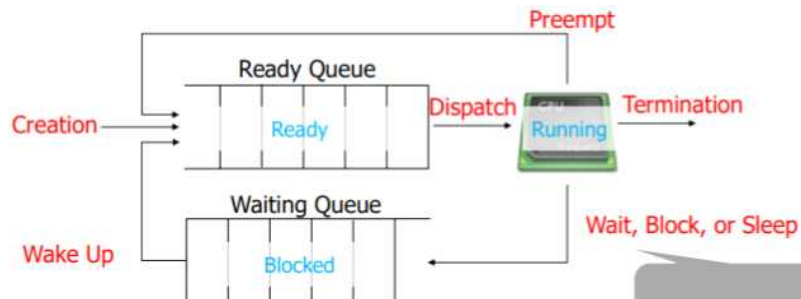
- Process의 한 주기



1. **creation**(fork()) -> **Ready**(대기 중의 여러 개의 process들 중에 1개의 process만 CPU에 올라감) -> **dispatch**(CPU에 1개의 어떤 process를 올리는 결정) -> **Running**(실제로 1개의 process가 실행됨) -> **termination**(Program이 끝나면 termination)(CPU-bound)
2. **creation**(fork()) -> **Ready**(대기 중의 여러 개의 process들 중에 1개의 process만 CPU에 올라감) -> **dispatch**(CPU에 1개의 어떤 process를 올리는 결정) -> **Running**(실제로 1개의 process가 실행됨) -> **timeout**(preemption)(일정 실행시간을 넘기면 CPU를 강제로 뺏음) -> **Ready** -> ...
3. **creation**(fork()) -> **Ready**(대기 중의 여러 개의 process들 중에 1개의 process만 CPU에 올라감) -> **dispatch**(CPU에 1개의 어떤 process를 올리는 결정) -> **Running**(실제로 1개의 process가 실행됨) -> **blocking**(yield)(CPU를 자발적으로 내려놓음) -> **Blocked**(I/O bound)(특정 명령이 들어오기 전까지 계속 Blocked 상태) -> **unblocking**(wake up)(특정 명령이 들어오면 blocked가 풀림) -> **Ready** -> ...

● Ready and Wait Queue

- OS가 scheduler를 구현하는 방법(CPU에 무엇을 올릴 것인가)



- Creation : fork()되어 process가 복제되어 생성됨. 이후 Ready Queue에 넣어줌.
- Ready Queue : 실행 대기상태인 process들이 모두 존재함. 내부 ordering에서 많은 알고리즘이 사용됨. (ordering된 순서대로 줄을 서 있음.)
- Dispatch : Ready Queue에서 최종 선정된 process를 CPU에 올려줌.
- Preempt(비자발적) : CPU를 강제로 뺏어버리고 아직 처리할 것이 남아 있기 때문에 Ready Queue에 다시 입력됨.
- Termination : Program이 모두 종료되면 해당 process를 없앴.
- Waiting Queue(Blocked) : 자발적으로 CPU를 내려놓은 process들이 특정 명령이 발생할 때까지 기다림. (ordering된 순서대로 줄을 서 있음.)
- Wait, Block, or Sleep(자발적) : CPU를 자발적으로 내려놓고 Waiting Queue(Blocked)에 입력됨.
- Wake Up : 특정 명령이 실행되면 해당 process가 Ready Queue에 입력되어 실행 대기 상태에 접어들.

● Non-Preemptive to Preemptive

- Non-Preemptive Scheduling
 - > window 3.1까지 해당.
 - > Process 하나가 무한 loop에 빠지게 되면 CPU를 preempt하지 못하여 전체 system이 멈춰버린다.
- Preemptive Scheduling
 - > window 95부터 해당.
 - > Process 하나가 무한 loop에 빠져도 일정 시간이 지나면 CPU를 preempt하여 다른 process로 changing되기 때문에 다른 process들은 영향을 받지 않고 계속 돌게 된다.

● PCB (Process Control Block)

- Process들의 정보를 담고 있는 Table (각 process에 대한 세부항목)
 - > Process scheduling 상태
 - > Process ID (PID)
 - > Saved CPU context (저장되어 있는 CPU의 내부 상태) ...
- Linux : task_struct (process의 정보를 관리하는 자료구조)

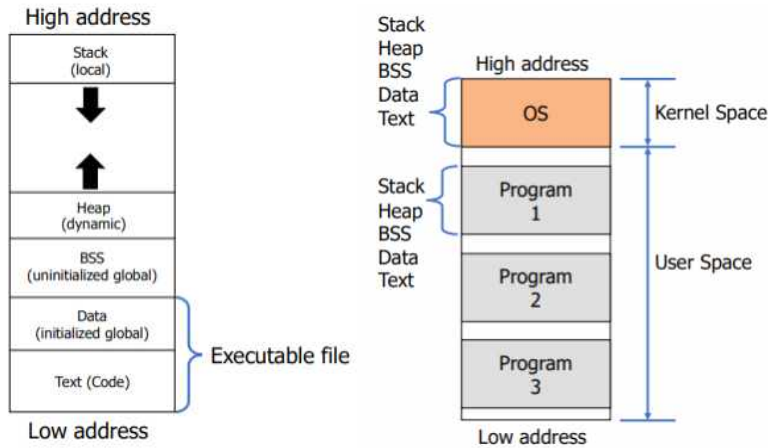
● Nice Value

- Linux는 기본적으로 공평한 scheduling을 사용함.
- 하지만, superuser(관리자)는 각 process마다 nice value를 세팅해줄 수 있다.
- Process마다 약간의 우선순위를 주어 Process의 처리 순서에 관여함.
- -20 (가장 높은 우선순위) ~ 0 (default) ~ +19 (가장 낮은 우선순위)
- `sudo nice -n -20 ./hello` : hello라는 실행파일 실행 시 우선순위를 가장 높게 줌.

15. Virtual Memory

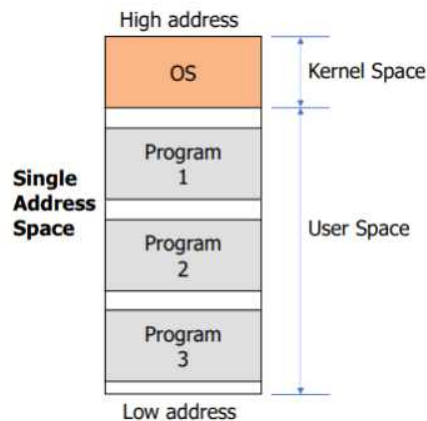
● Singleprogramming vs. Multiprogramming

- 여러 개의 program이 single address space에 load됨.



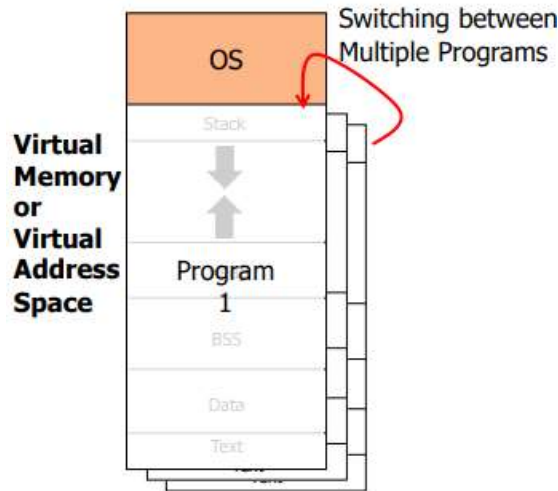
- Single program
- Multi program
- * Multi program
 - > single address space에 여러개의 program이 올라감.(공간을 공유함)
 - > 각 program마다 Stack, Heap, BSS, Data, Text 영역을 가지고 있음.
 - > Kernel Space : OS가 올라간 영역(User program이 접근하면 안됨.)
 - > User Space : User program(사용자의 program)이 올라가는 영역
(system이 훼손될 수 있기 때문에 kernel space에 접근 불가.)

● Single Address Space



- Program address는 compile time에 address가 고정적이어야 compile이 가능하다. 하지만 single address space에서 program이 추가되면 address가 변경되어 모든 다른 program이 모두 다시 compile 되어야 한다.
- 또한 User program 간의 보호가 되지 않는다. program끼리 공간 침범이 가능하게 되어 문제가 생긴다.

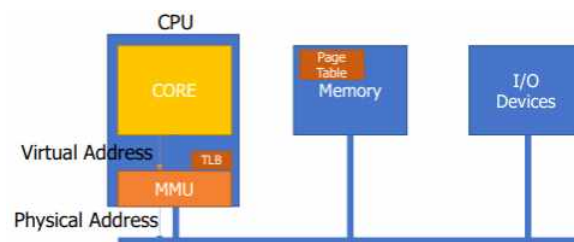
● Virtual Address Space (최근에 사용하는 방식)



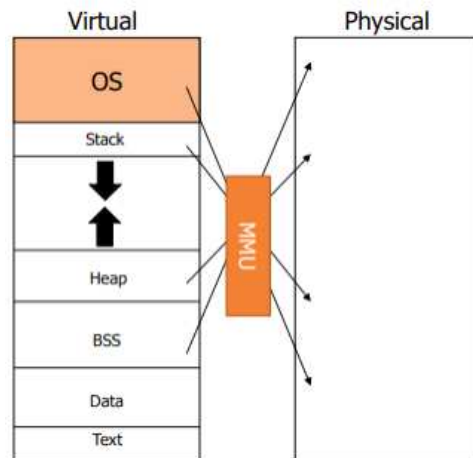
- Program간의 memory 침범이 일어나지 않음.(해킹이나 오류로 인한 프로그램 파괴가 일어나지 않음.)
- 아예 다른 공간을 Program끼리 사용하기 때문에 OS만 다른 Program의 address에만 접근할 수 있다. (Card Switching하듯이 Memory를 switching하여 실행 program을 교체하는 개념이다. 각각의 Address space가 쌓여 있고 맨 위에는 현재 실행하는 Program이 올라가 있음.) => Address 전체를 혼자 사용하고 있는 듯한 illusion
- OS(kernel space)는 고정적으로 올라가 있음. (OS는 program에 접근 가능, Program은 OS에 접근 불가능, Program끼리 접근 불가능.)

● MMU (Memory Management Unit)

- System bus에 접근하기 직전에 변환 layer를 제공한다. (CPU 내부에서)
- MMU가 돌게 되면 physical address space가 virtual space에 의해 가려짐.
- Virtual address와 physical address 사이에 변환을 도와주는 역할.



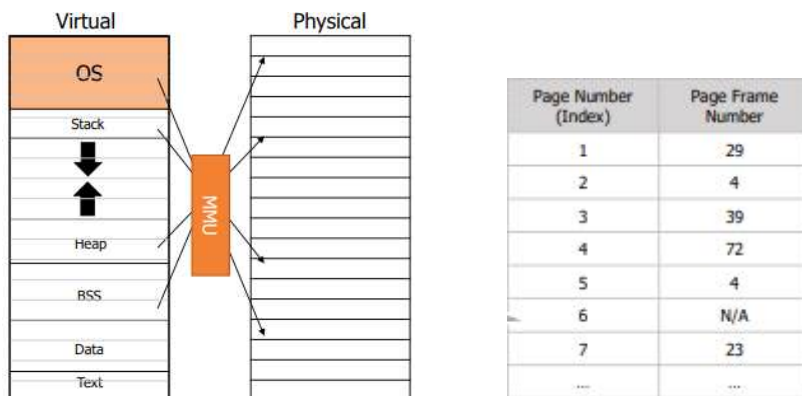
- (1) System booting시에 MMU가 꺼진 상태로 시작함.
 - (2) CPU에서 돌아가는 program이 physical address를 봄.
 - (3) MMU를 키게 되면 모든게 사라지고 SW는 나만의 커다란 virtual 공간이 보임.
 - (4) 해당 virtual 공간에 접근을 하게 되면 MMU를 통과하며 physical address에 접근하여 data를 가져오게 됨. (SW는 virtual 공간이라는 것을 알지 못함.)
- MMU가 Core에서 virtual address(ISA)를 받음.(Memory 어디를 읽어주세요, 써주세요.) -> MMU가 Memory에 존재하는 Page Table을 확인하여 특정 Virtual page에 해당하는 Physical page frame을 읽어 Physical address를 받아옴. (TLB에 존재하면 TLB에서) -> 받아온 Physical address를 system bus에 넣어줌.



- MMU가 Virtual address와 Physical address 사이에서 address를 자동으로 변경해줌.
- 이 과정을 SW는 알 수 없고 virtual에서 예쁘게 정렬된 듯하게 보여줌.

● Page Table

- MMU는 virtual과 physical address 사이에 변환을 위한 mapping table이 필요함.
 - => 4GB의 virtual address와 4GB의 physical address 사이에 byte-level의 mapping table의 크기는 얼마인가? (32bit system이라고 가정)
 - > 각 process에 대해 4byte의 table의 entry가 필요하다고 가정하면 16GB가 필요함. (4GB*4) - 독점 위치를 지정하려면 process당 4byte가 필요함.
 - > 그러므로 table을 더 크게 잘라서 mapping해야 함.
- Page-level mapping (4KB usually) => Table도 memory를 사용함.
 - > Page (virtual page), Page frame (physical page)
 - > Size = 4MB (4*4GB/4KB) for each process (각 entry마다 4bytes가 필요하고, 4GB의 공간을 4KB의 크기씩 자르기 때문에 $4 \times (2^{32}/2^{12})$)

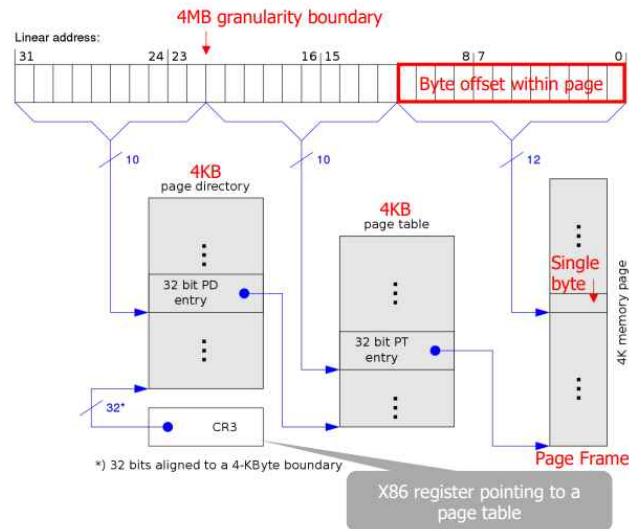


- > 보통 Heap과 Stack 사이에는 빈 공간이 광활하게 존재한다. (unused page)
 - (ex> Page Number 6 -> Page Frame Number N/A)
- > 같은 page frame에 mapping되기도 한다. (table마다 자른 공간이 크기 때문에)
 - (ex> Page Number 2, 5 -> Page Frame Number 4)
- > process의 수가 증가하면 mapping에 소비하는 memory 공간 증가.
 - (100 process * 4MB = 400MB)

● Multi-level Page Table

- Page Table은 너무 많은 RAM을 차지할 수 있음. (각 Process마다 4MB)

* Two level Page Table



- Byte offset within page : 4096 bytes의 offset (Page(4KB) 안쪽의 address이기 때문에 신경 쓰지 않아도 됨.)

-> 실제 physical page frame의 address와 내부 내용들을 담고 있음.

- 4MB의 경계를 끊어서 address space 구분. (20bits를 반으로 쪼갬)

- MSB가 포함되어 경계 된 10bits (Page directory)

-> 1024개의 토막이 나게 되어 address space를 1차적으로 구분함.

(ex> MSB가 0이라면 address space의 앞 공간의 절반, 1이라면 address space의 뒤 공간의 절반 => 10bits가 모두 0이라면 첫번째 토막임)

-> 해당 공간에 1024개의 entry가 생기게 되고 공간은 4bytes의 크기의 주소값을 가지는 entry들이 존재함. entry는 page table을 가리키는 address space가 존재하게 됨. 따라서 page directory의 크기는 $2^{10} * 2^2 = 2^{12} = 4KB(1 \text{ Page})$ 가 됨.

- 남은 경계 된 10bits (Page table)

-> 1024개의 토막이 나게 되어 address space를 2차적으로 구분함.

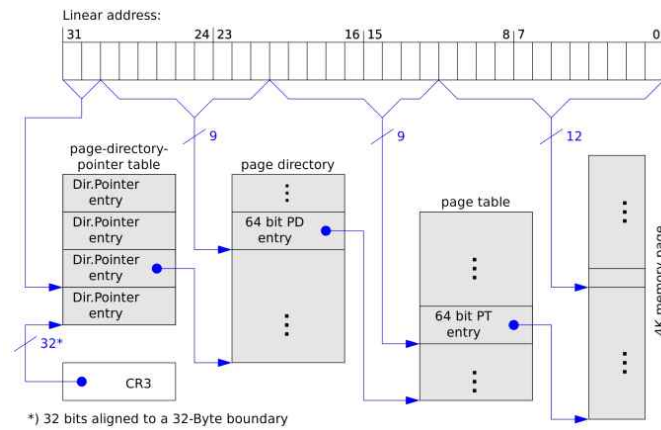
-> Page directory의 하나의 entry에 address space가 존재한다면, (Page directory가 비어 있다면 NULL로 채움) 각 entry마다 가리키고 있는 page table의 $2^{10} * 2^2 = 2^{12} = 4KB(1 \text{ Page})$ 의 공간을 가지고 있게 됨. 해당 page table의 각 entry의 크기는 4bytes이며 physical page frame의 address space를 가지고 있음.

=> 예를 들어 Page directory의 1024개의 토막 중 1000개가 NULL이라면 24개의 page table을 가지고 각각의 page table의 NULL이 없다고 가정하면 $24 * 4 * 4KB = 384KB$ 의 page table을 가지게 됨.

=> 이처럼 address space를 구분하여 RAM의 사용공간을 최소화하고 접근을 빠르게 함.

* CR3 : Page table의 위치를 가리키는 X86 register. MMU가 해당 register의 내용을 참고하여서 physical page frame을 결정해줌.

* Three level Page Table

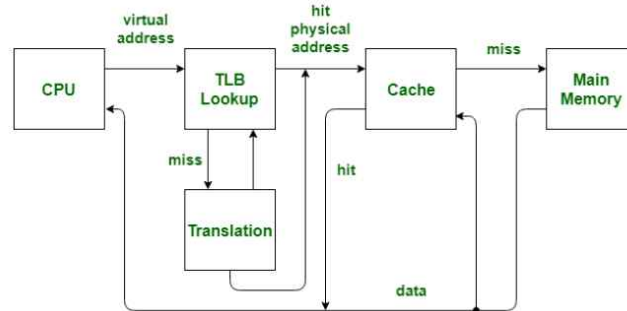


- Two level page table보다 1단계 더 쪼갬.
- Byte offset within page : 4096 bytes의 offset (Page(4KB) 안쪽의 address이기 때문에 신경 쓰지 않아도 됨.)
-> 실제 physical page frame의 address와 내부 내용들을 담고 있음.
- 앞의 20bits의 address space를 page directory가 9bits, page table이 9bits를 사용함. MSB부분의 맨 앞 2bits는 page-directory pointer table로 사용하여 1GB씩 entry를 나누어 4토막으로 구분함.
- Address space를 확인하기 위해 MMU가 접근하면 해당 page-directory pointer table의 빈 곳은 고려하지 않고 접근하지 않음. 만약 비어 있지 않다면 해당 entry에 접근하여 연결되어 있는 page directory -> page table로 접근하여 address space를 사용함.
- 만약 모든 공간이 꽉 차있지 않으면 Memory적으로 굉장히 효율적이지만, 4GB를 모두 mapping해야 한다면 Two level보다 더 계층적이기 때문에 Memory를 더 소모하게 됨.
- 2bits를 page-directory pointer table로 사용하기 때문에 page directory와 page table은 각각 8bytes씩의 entry를 가지게 됨.

=> Address space의 page는 대부분 비어있다는 것을 이용하여 Page table을 memory 효율적으로 design (MMU inputs : 32bits, outputs : 32bits)

● TLB(Translation Lookaside Buffer)

- Page Table을 위한 cache : locality로 자주 접근하는 곳만 접근함. (빈 공간이 대부분)
- 가장 최근에 virtual-physical을 변경해준 entry만 저장해주는 작지만 빠른 cache.
- Memory를 읽기 위해 Memory를 또 한 번 접근해야 하는 비효율성을 개선함.
- TLB는 address의 변경을 가속해주고, CPU Cache는 Memory의 접근을 가속해줌.
- TLB와 cache를 고려한 Memory access



1. TLB, Cache hit

CPU -> virtual address를 전송 -> TLB 확인 -> 해당 virtual이 존재하면 -> hit physical address -> Cache -> 해당 physical address가 존재하면 -> hit -> return physical address to CPU

2. TLB hit, Cache miss

CPU -> virtual address를 전송 -> TLB 확인 -> 해당 virtual이 존재하면 -> hit physical address -> Cache -> 해당 physical address가 존재하지 않음 -> miss -> memory에 접근하여 해당 physical address를 cache에 올림 -> return physical address to CPU

3. TLB miss, Cache hit

CPU -> virtual address를 전송 -> TLB 확인 -> 해당 virtual이 존재하지 않음 -> miss -> Translation(main memory에 직접 접근하여 physical address를 불러옴) -> Cache -> 해당 physical address가 존재하면 -> hit -> return physical address to CPU

4. TLB miss, Cache miss

CPU -> virtual address를 전송 -> TLB 확인 -> 해당 virtual이 존재하지 않음 -> miss -> Translation(main memory에 직접 접근하여 physical address를 불러옴) -> Cache -> 해당 physical address가 존재하지 않음 -> miss -> memory에 접근하여 해당 physical address를 cache에 올림 -> return physical address to CPU

* Translation과정에서 multi-level page table의 level이 몇 개로 나뉘어져 있냐에 따라서 memory에 접근하는 횟수가 정해짐. (ex> Two level이고, Cache miss이면 총 3번 접근함.)

● What if no free memory? (만약 Memory의 빈 공간이 없다면?)

* Physical space가 매우 작아도 virtual space는 클 수 있음. Process가 여러 개일 경우 해당 physical space에 virtual space의 data들이 다 들어가지 못해 process를 실행 시키지 못할 수도 있음.

- Out-Of-Memory Killer

- > kernel에 killer code가 존재함.
- > process를 희생시켜서 kill한다. (process 선정 시 매우 복잡한 알고리즘)
- > 오랫동안 사용하지 않은 process를 최우선으로 kill하고, 해당되는 process가 없다면 Memory를 가장 많이 차지하는 process를 kill함.
- > 어떠한 process가 kill될지 모르기 때문에 process가 kill되면 다시 실행시키기 까다로운 임베디드 시스템에서는 사용하지 않는다.

- Swapping (Program 단위로 전체에 대해)

- > 희생시킬 process를 disk에 save하고 memory에서 clear함. (swap out)
- > clear했던 process를 사용하려면 disk에서 다시 memory에 store함. (swap in)
- > swap out과 swap in이 빈번하게 일어나게 되면 system이 매우 느려지게 되기 때문에 성능이 중요한 임베디드 시스템에서는 사용하지 않는다.

- Paging (4KB(Block의 단위) 단위로)

- > 가장 최근에 사용하지 않은 page가 포함된 page frame들을 새로운 page로 교체함. (LRU 알고리즘으로 교체할 page를 선택함.) (page out, (4KB 단위))
- > 희생할 page들이 disk의 file로 존재하는 경우에만 사용. (대부분 Text section)
=> Memory에서 날려도 disk에 써놓지는 않음.
- > Text section (disk에 있는 program을 memory에 copy해 놓은 영역) : 해당 page는 날려도 나중에 필요시 다시 disk에서 올려서 사용하면 됨. (page in)

- Demand paging (lazy loading)

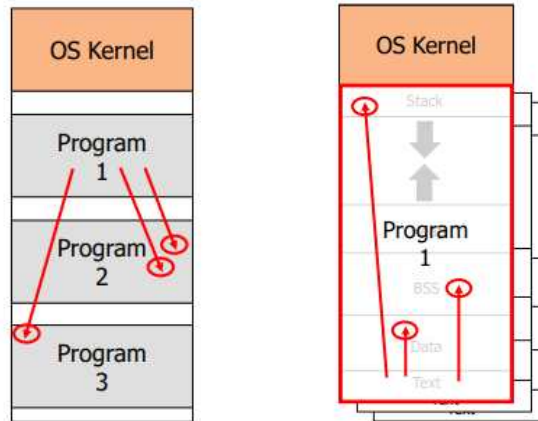
- > 보통 program을 loading할 때 전부 다 loading하지 않고 1Page만 loading 함.(main function의 첫번째 line이 속해있는 page(4KB))
- > 실제로 access하기 전까지는 page를 load하지 않음.
- > Page table에 mapping되어 있지 않은 page에 접근 시, page에 대응되는 page frame이 없어서 page fault가 발생하고 이를 OS에게 알림.
- > Page fault가 발생하면 OS가 disk에서 해당 page를 읽어서 memory에 loading 해주고 page Table에 page frame을 적어주고 program을 다시 resume.
(접근이 불가능하면 Page in, Page out)

disk → memory
→ page table

16. Inter-Process Communication

● Isolation Between Processes

- Process들은 각자 서로 다른 process의 address space에 접근할 수 없음.
- process들끼리 협력하여 큰 system을 구축하는 방법.



-> Single memory system에서는 program끼리 서로 접근이 가능하여 data를 주고 받을 수 있지만 virtual memory system에서는 program끼리 data를 주고 받는 것이 불가능함.

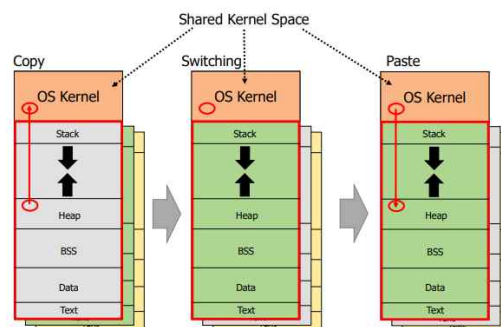
● Inter-Process Communication (IPC) (Process들 간의 소통방법)

- Filesystem, Signal, Pipe, FIFO

✓ Message Queue, Semaphore, Shared Memory => Called System V IPC
✓ Network (Socket) programming (확장된 IPC mechanism)

● Kernel-Assisted IPC (Filesystem, Signal, pipe, FIFO, Message Queue, Semaphore)

- Process가 switching되기 전에 kernel space(OS kernel)에 data를 저장.
- Context switching 이후 switching된 process에서 kernel space에 저장된 data를 copy하여 불러옴.

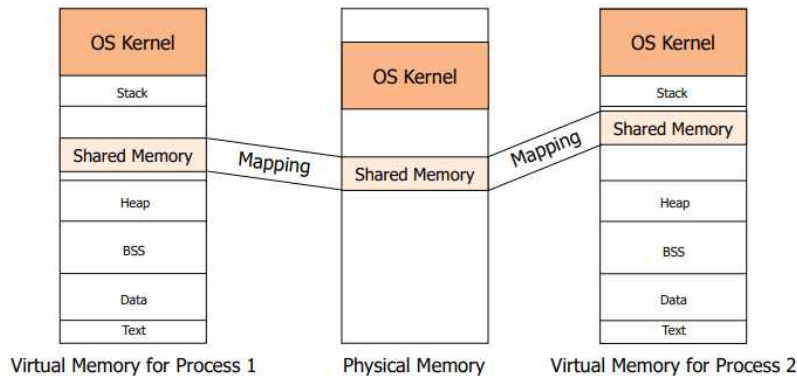


-> kernel(OS) space는 항상 고정으로 맨 위에 존재하고, foreground에 process가 올라가면 맨 앞의 process만 보이게 됨.

-> System call을 통해 Trap으로 kernel에 data를 저장하고 Context switching 이후 변경된 process에서 system call을 통해 Trap으로 해당 data를 불러옴.

● Shared Memory-based IPC

- 서로 다른 process의 address space에 하나의 page frame을 mapping시킴.
- Process의 page table을 조작함.



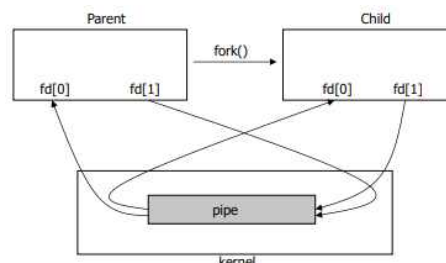
- 서로 다른 process의 address space에 공통된 physical memory가 같이 보임. (kernel에 copy할 필요 없음)
- 만약 Process1의 해당 공간에 접근하여 작업을 하게 되면 mapping된 physical memory에 접근하는 것이므로, Process2의 공간에서도 같은 physical memory가 mapping되어 있어 작업이 공유되게 됨.

● Filesystem

- File을 통신 매체로 사용함.
- possible disk I/O로 인해 느림.
- 하나의 process에서 write하는 도중에 다른 process에서 read할 수 있으므로 file 손상이 일어남. => File locking이 필요함 (fcntl file lock -> sync)

● Pipe

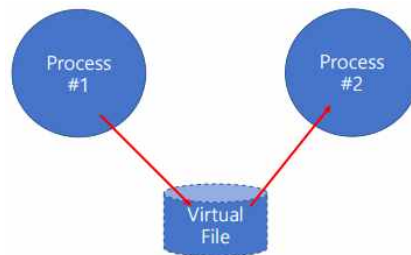
- Parent와 child process 사이에서의 양방향 pipe. (parent-child 관계가 아니면 불가능)
- pipe() system call -> kernel에서 pipe를 생성하고, read와 write를 위한 두 개의 file descriptors를 제공.
- PIPE_BUF(4096 bytes) 보다 적은 writing은 atomic함. (pipe의 크기 -> 4KB(1Page))
- 4KB보다 큰 data면 pipe를 여러번 사용해야 함.



- > fork()가 되면 서로 다른 memory에 존재하기 때문에 내장 변수는 각자 다른 변수로 취급됨.
- > 0과 1에 비밀번호 같은 번호를 부여 (서로 통신을 위한 ID)
- > 0 : Output, 1 : Input // 자기 자신에게는 pipe할 필요가 없음

● FIFO (Named Pipe)

- Pipe는 parent와 child process 관계에서만 사용이 가능함.
- FIFO는 filesystem에서 virtual file을 사용함으로써 더 일반적으로 사용이 가능함.
- mkfifo() function 또는 mkfifo command는 FIFO를 생성함. (virtual file을 생성)
- PIPE_BUF(4096 bytes) 보다 적은 writing은 atomic함. (pipe의 크기 -> 4KB(1Page))



- > virtual file을 생성하여 process끼리 통신을 함. (실제 disk의 file이 아님)
- > disk가 돌아가지 않고, OS가 file인 것처럼 속여줌. (kernel space에 copy해서 들고 있음)

● Signal

- Interrupt handling과 유사한 process간의 알림.
- 특별한 data를 가지고 있는 것이 아니라 단순한 signal.
- 각 process는 signal handler function을 custom하여 사용할 수 있음. custom하지 않으면 default action이 실행됨.
- ex) kill command와 kill() system call -> pid(다른 process)에게 sig를 전달함.

```
#include <sys/types.h>; #include <signal.h>; int kill(pid_t pid, int sig);
```
- sigaction() : custom signal handler를 install.
- \$ man 7 signal : signal에 대한 manual

Signal	Portable number	Default Action	Description
SIGABRT	6	Terminate (core dump)	Process abort signal
SIGALRM	14	Terminate	Alarm clock
SIGBUS	n/a	Terminate (core dump)	Access to an undefined portion of a memory object
SIGCHLD	n/a	Ignore	Child process terminated, stopped, or continued
SIGCONT	n/a	Continue	Continue executing, if stopped
SIGFPE	n/a	Terminate (core dump)	Erroneous arithmetic operation
SIGHUP	1	Terminate	Hanging
SIGILL	n/a	Terminate (core dump)	Illegal instruction
SIGINT	2	Terminate	Terminal interrupt signal
SIGKILL	9	Terminate	Kill (cannot be caught or ignored)
SIGPIPE	n/a	Terminate	Write on a pipe with no one to read it
SIGPOLL	n/a	Terminate	Pollable event
SIGPROF	n/a	Terminate	Profiling timer expired
SIGQUIT	3	Terminate (core dump)	Terminal quit signal
SIGSEGV	n/a	Terminate (core dump)	Invalid memory reference
SIGSTOP	n/a	Stop	Stop executing (cannot be caught or ignored)
SIGSYS	n/a	Terminate (core dump)	Bad system call
SIGTERM	15	Terminate	Termination signal
SIGTRAP	n/a	Terminate (core dump)	Trace/breakpoint trap
SIGTSTP	n/a	Stop	Terminal stop signal
SIGTTIN	n/a	Stop	Background process attempting read
SIGTTOU	n/a	Stop	Background process attempting write
SIGUSR1	n/a	Terminate	User-defined signal 1
SIGUSR2	n/a	Terminate	User-defined signal 2
SIGURG	n/a	Ignore	High bandwidth data is available at a socket
SIGVTALRM	n/a	Terminate	Virtual timer expired
SIGXCPU	n/a	Terminate (core dump)	CPU time limit exceeded
SIGXFSZ	n/a	Terminate (core dump)	File size limit exceeded

- > Portable number : signal마다의 고유 번호
- > Default Action : 취하는 행동
- > SIGINT : Ctrl-C (무시가능) // SIGKILL : \$ kill -9 (무시불가) // SIGTSTP : Ctrl-Z (무시 가능) // SIGUSR1, SIGUSR2 : 원하는 action custom 가능

● System V IPC

- Message queue, Semaphore, Shared memory

- Key를 관리하는 function

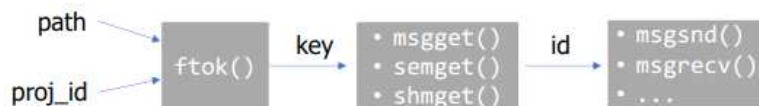
```
#include <sys/types.h>; #include <sys/ipc.h>
```

```
key_t ftok(const char *pathname, int proj_id);
```

-> key를 생성해줌. (Process끼리 key를 공유해야 대화가 가능함)

-> *pathname : 기존에 존재하는 file

-> proj_id : 동일한 path를 사용하여 다른 key를 생성. 최하위 8bit만 사용함.



- ftok() : 해당 경로와 id에 대한 key를 생성함.

- key : 임의의 key를 생성함 (해당 key를 가지고 통신을 위한 특정 id를 만듦)

- id : id를 이용하여 msg 통신. (id 공유 방법 -> key management의 핵심)

- key management command

-> ipcs : showing keys

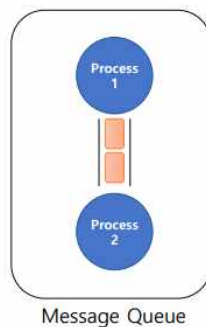
-> ipcrm : removing keys

● Message Queue

- Process 사이에 queue를 생성함.

- Pipe or FIFO와 비슷함.

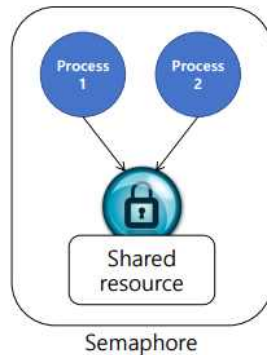
- System Calls : msgget(), msgsnd(), msgrcv(), msgctl()



-> 특별한 Queue를 통해 msg를 전달 (parent-child 관계가 아니어도 괜찮음)

● Semaphore

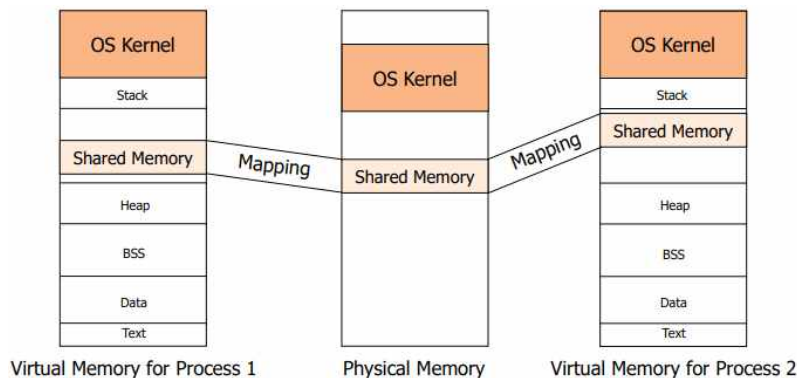
- Process 사이에 sync(동기화)를 해줌.
- System Calls : semget(), semop(), semctl()



- > 두 process가 동시에 shared resource에 접근하지 못하게 sync(locking)
- > ex) Process 1이 접근 시 Process 2가 접근 못하게 lock

● Shared Memory

- Memory area는 각 process space에 mirroring됨.
- copy에 대한 overhead가 없음.
- System Calls : shmget(), shmat(), shmdt(), shmctl()
- 각 process space가 접근 시에 sync가 필요함. (semaphore가 사용)
 - > Round Robin 스케줄링으로 process의 실행이 왔다갔다 하기 때문에 문제 발생.
 - > sync(locking)을 통해 데이터 훼손을 방지함.



● Warning

- IPC mechanisms은 사용 및 검증이 매우 어려움.
- 따라서 실제 산업에서는 ROS, AUROSAR와 같은 middleware 통신을 대신 사용함.
- 대부분의 middleware는 여러 대의 컴퓨터에서 원격으로도 통신이 가능하게 지원해줌.
 - > ex) IBM, kafka, OMQ

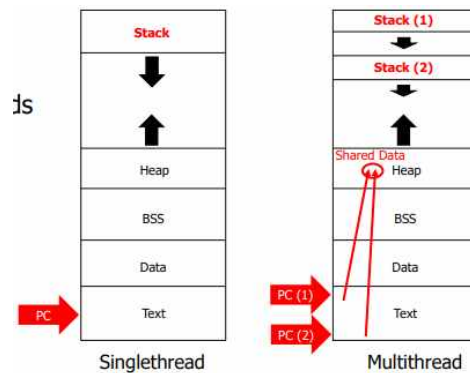
17. Multithreading

● Process만으로 충분한가?

- Process를 사용하는 큰 규모의 system을 만들 때 때때로 문제가 발생함.
 - > IPC의 사용과 검증이 어려움.
 - > 10K problem : 1000개의 process를 사용하면 system이 무조건 down. (OS가 처리할 때 한계)
 - > Process가 많아지면 virtual memory 관리가 어려워짐. (overhead)
- Process는 확장할 수 없으므로 Process가 많아지면 system crash가 일어남.
 - > Process를 create(fork())하는 것은 매우 heavy함. (기존의 process를 copy하는 것이 heavy하여 부하를 느끼게 됨.)
 - > 모든 process에 대한 전용 address space를 유지하는 것도 매우 heavy함.
 - ex) Per-process page tables
 - > Context switching은 virtual memory의 swithing을 유발함. (Context switching을 위해 TLB를 날려야함. Process의 최근 page table을 TLB가 가지고 있는데 process가 바뀌게 되면 TLB가 가지고 있는 것들이 모두 의미가 없어짐. 따라서 page table도 switching을 해야하여 성능이 저하됨.)
- Address space 하나를 두고 해당 address space를 공유하는 경량화된 process를 만들 고자함.

● Multiple Threads in a Process

- Process는 기본적으로 1개의 thread를 가짐. (main thread) -> PC가 1개
- 필요시 더 많은 thread를 만들 수 있음 (다중 PC와 stack)
- Data 영역은 thread들 간에 공유가 가능함. (ex> global variables)
 - > Thread들 간의 memory 보호가 없음. (stack을 제외하면 서로 공유)
- Address copy가 필요하지 않고 기존의 address space에 PC 하나만 추가하면 되기 때문에 thread를 만드는 것이 더 빠름.
- Thread들끼리 하나의 Page Table을 공유함.
- Thread들끼리 TLB도 공유하기 때문에 TLB를 flush하지 않아 overhead가 없음.



- Address space에 process를 몇 개씩 묶어서 할당해줌. (1개의 address space에 여러 개의 thread가 돌아감.)
- Thread들끼리 Stack을 제외한 나머지 부분들을 공유하여 IPC를 사용하지 않아도 됨.
- Thread 하나 당 하나의 Stack을 가지게 되어 Stack에서 수행하는 작업들은 공유되지 않음.

* Pthread Library : c언어에서 default로 사용하는 library

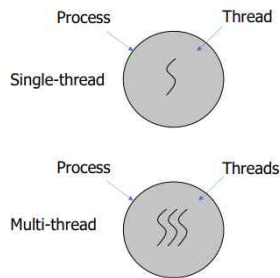
● Creating Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *mythread(void *arg)
{
    printf("%s\n", (char *) arg);
    return NULL;
}
int main(void) => main function은 원래대로 순차적으로 로직 처리.
{
    pthread_t p1, p2;
    printf("main: begin\n");
    pthread_create(&p1, NULL, mythread, "A"); /* p1 thread 생성 */
    pthread_create(&p2, NULL, mythread, "B"); /* p2 thread 생성 */
    // join waits for the threads to finish
    pthread_join(p1, NULL); /* wait until p1 terminates */
    pthread_join(p2, NULL); /* wait until p2 terminates */
    printf("main: end\n"); => main thread는 p1, p2가 끝날 때까지 대기했다가 printf 실행
    return 0;
}
```

=> 각 thread(p1, p2)는 연결되어 있는 함수의 로직만 처리함. p1과 p2중 어떤 thread가 먼저 종료될지 모름.(CPU를 잡는 순서에 따라 달라짐 : sleep을 주어 순서에 영향 줄 수 있음)

-> 그러므로 관리가 굉장히 어려움.

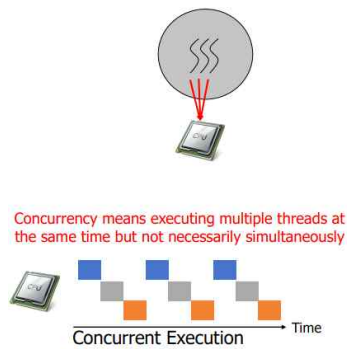
=> compile시 \$ gcc thread.c -lpthread : -lpthread가 붙어야지 thread가 실행됨.



-> main thread에서 thread 두 개를 생성함으로 총 3개의 thread가 동작함.

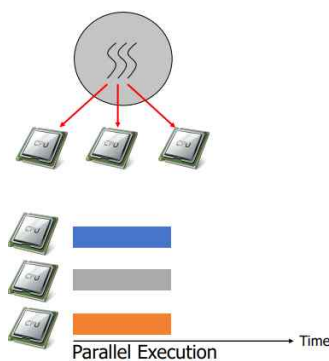
● Concurrency and Parallelism

- Concurrent Execution (동시 실행)



- > Thread가 서로 얹히게 되어 Program이 망가지게 됨.
- > 동시에 실행되는 것과 같은 성질을 가짐. (동시성)
- > CPU가 하나면 Thread들을 CPU 하나로 커버함. (Round Robin으로 스케줄링)
- > 마치 Thread들이 동시에 실행되는 것처럼 illusion. (실제로는 시점에서 하나의 Thread만 동작)

- Parallel Execution (병렬 실행)

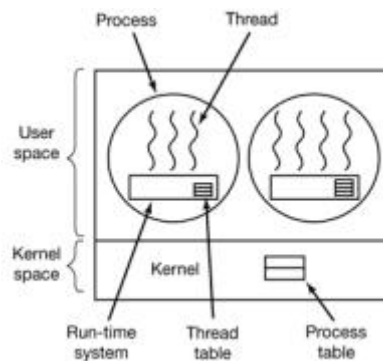


- > 각각의 Thread가 각각의 CPU에서 동작함. (시점에서 모든 Thread가 모두 동작)
- > 각자 하는 일에서 data를 공유하게 되면 문제가 생김.
- > 물리적으로 Thread가 여러개 실행됨.

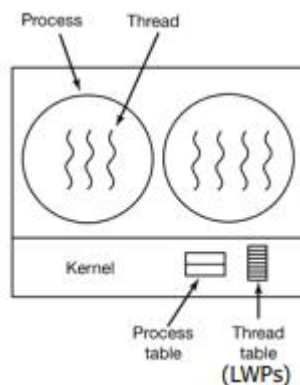
=> Software관점에서 보았을 때 Concurrent Execution과 Parallel Execution은 같음. (물리적으로 다르고, 속도에 차이가 있음)

● Implement Threads (Scheduling)

- User-level Thread Scheduling (1:n) -> (process : 1, thread : n)
 - > kernel은 process를 schedule. (kernel은 thread의 존재를 아예 모름)
 - > Process에게 CPU를 할당하면 process가 알아서 여러 개의 thread를 schedule 하여 time을 나누어서 사용함.
 - > kernel이 thread scheduling에 전혀 필요가 없음.
 - > Run-time system과 Thread table이 모두 process 내부에 존재함.
- * void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);
- * int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
- => User-level thread scheduling (context switching) : 대부분 kernel-level thread처럼 OS가 scheduling하듯이 구현



- Kernel-level Thread Scheduling (1:1) -> (LWP(light-weight process): 1, thread : 1)
 - > Process가 아닌 kernel(OS)이 thread를 직접 schedule. (CPU를 thread에 할당)
 - > LWP(light-weight process)를 참조하여 사용함. (경량 process)
 - > Linux에서 각 thread에 대해 task_struct가 생성됨.
 - > Linux에서 address space를 공유하는 Thread들을 묶어 하나의 process로 봄.



User-level thread	Kernel-level thread
OS does not know the existence of threads	OS manages and schedules threads
Small context switching overhead	Large context switching overhead
If one thread blocks, the entire process stops	Even when a thread blocks, other threads progress
Threads in the same process cannot run in parallel	Threads in the process can run in parallel on different CPUs
Process with 1000 threads and process with one thread are treated equally (process-level fairness)	Each thread takes equal share of CPU (thread-level fairness)

* User-level Threads

- OS(kernel)가 thread의 존재를 알지 못함. (CPU를 process에게 주기 때문)
- Context switching의 overhead가 적음. (OS가 scheduling할 필요가 없기 때문)
- Thread 하나가 block되면 해당 process의 모든 thread가 멈춤.
 - > Thread block시에 OS가 느끼기에는 process를 자발적으로 CPU를 반납했다고 생각하기 때문.
- 같은 Process에서의 thread들은 병렬적으로 동작이 불가능함.
- Process-level로 공평하게 관리하기 때문에 같은 process에 1000개의 thread가 존재하나 1개의 thread가 존재하나 동일하게 관리됨.

* Kernel-level Threads

- OS(kernel)가 직접 thread를 관리하고 scheduling함. (CPU를 kernel에게 줌)
- Context switching의 overhead가 큼. (OS가 직접 thread를 scheduling)
- Thread 하나가 block되어도 다른 thread들은 동작함.
 - > CPU를 반납하여도 OS가 모든 thread를 관리하기 때문에 thread가 block 되어도 나머지 thread가 process가 필요하다는 것을 알아 다시 할당해줌.
- 같은 Process에서의 thread들은 다른 CPU에서 병렬적으로 동작가능.
- Thread-level로 공평하게 관리하기 때문에 thread들은 CPU를 공평하게 사용.
 - > Process-level 공평성의 문제 해결

● Thread는 관리가 어렵고 최적화가 매우 어렵기 때문에 사용을 최대한 기피해야 함.

18. Locking

● Inter-thread Shared Variable Access

```
unsigned int loop_cnt;
```

volatile unsigned int counter = 0; => p1 thread와 p2 thread가 global variable을 공유하여 사용하여 연산함.

```
void *mythread(void *arg)
```

```
{
```

```
    for (i = 0; i < loop_cnt; i++)
```

```
        counter = counter + 1;  => 실제로는 3개의 instruction으로 실행됨.
```

```
    return NULL;
```

```
}
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    loop_cnt = atoi(argv[1]);
```

```
    pthread_t p1, p2;
```

```
    pthread_create(&p1, NULL, mythread, "A");
```

```
    pthread_create(&p2, NULL, mythread, "B");
```

```
    // join waits for the threads to finish
```

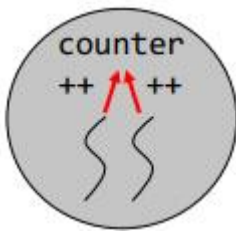
```
    pthread_join(p1, NULL);
```

```
    pthread_join(p2, NULL);
```

```
    printf("%d\n", counter);
```

```
    return 0;
```

```
}
```

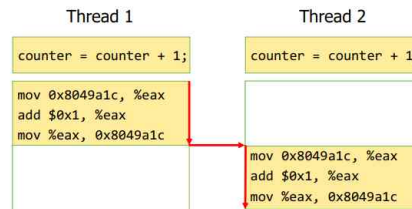


=> global variable을 thread들이 공유하여 연산을 진행함.

=> \$ gcc shared.c -o shared -lpthread \$./shared 1000000으로 실행을 하면 정확히 2배가 된 결과값이 도출되어야 하지만, 정확히 2배가 아닌 이상한 값이 도출됨.

● Race Condition

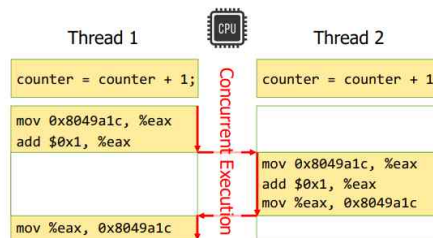
- 연산에 대한 결과가 제어(관리)할 수 없는 Event의 순서나 타이밍에 따라 달라지는 경우.
-> sequence timing에 의해 결과가 바뀌는 경우 (ex> preemptions)



Works as intended

- > 이상적으로(의도한 대로) 실행이 된다면 문제가 발생하지 않음.
- > 하나의 critical section이 모두 실행되면 다음 critical section이 실행.

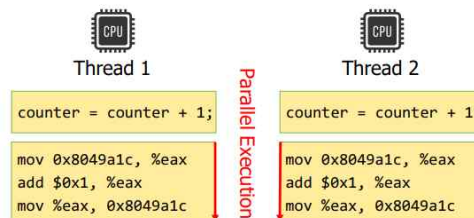
- Concurrent Execution



Functionally incorrect (even unpredictable) result

- > 하나의 CPU에서 Thread가 순서를 지키지 않고 실행되기 때문에 instruction의 순서가 불규칙함.
- > Thread가 강제로 preemption되어 변경됨으로 Thread 1에서 add한 값이 끊어져버림. Thread 2는 Thread 1에서 store하기 전 값으로 연산이 진행되고, Thread 2에서 연산된 값만 최종 store되기 때문에 정상적으로 동작하면 +2가 되어야 하는데 +1이 된 결과가 store됨. (예측할 수 없는 부정확한 결과)

- Parallel Execution



- > 여러 개의 CPU에서 Thread가 병렬적으로 실행되기 때문에 instruction의 순서가 불규칙함.
- > 각자의 CPU에서의 각 Thread가 동일한 critical section을 병렬적으로 동시에 실행하게 되어 서로의 instruction 실행 순서를 알지 못하기 때문에 예측할 수 없는 부정확한 결과가 도출됨.

● Shared Resource

- 여러 User(Thread)가 같은 resource를 사용함. (Thread가 서로 공유하는 data)
- Resource 사용 시에는 같은 resource 공간을 사용 시 홀로 사용해야 함. (상호 배제)
 - > 동시에 같은 resource 공간을 사용하게 되면 연산 시 문제가 생김
- Program의 모든 User(Thread)들은 resource 사용 시 locking을 확인하고 lock을 걸고 사용하는 것에 대한 약속이 되어 있어야 함.
- Computer에서 많은 shared resource가 존재함.
 - > Global or static variables, Files, Shared memory areas, HW registers...
 - > 해당 resource들은 여러 Thread가 공유해서 사용해야 하기 때문에 순서를 sync 해야 함.

● Mutual Exclusion for Critical Section

- Critical Section (instruction이 끊기지 않고 실행되어야 하는 영역)
 - > Shared Resource를 처리하기 위한 code segment
 - > 하나의 shared resource는 resource를 공유하는 다른 thread들에서 많은 critical section을 생성할 수 있다.
 - > 중간에 끊지 않고 instruction이 끝까지 마무리될 때까지 실행해야 함.
 - > lock이 걸려있는 부분을 방해하면 안됨.
- Mutual exclusion (상호 배제)
 - > 단 하나의 thread만 critical section에 존재해야 함.
- Atomicity (원자성)
 - > ex) counter = counter + 1; is not atomic
 - > 하나의 instruction이 아닌, 3개의 instruction으로 구성되어 있음.

```
void *mythread(void *arg)
{
    for (i = 0; i < max; i++)
        counter = counter + 1;
    mov 0x8049a1c, %eax
    add $0x1, %eax
    mov %eax, 0x8049a1c
    return NULL;
}
```

Critical Section

● How to protect critical section


1. Critical section에서 scheduling을 강제로 막음.

- > Critical section에 진입 전에 scheduling을 비활성화함.
- > Critical section 모두 실행 후 scheduling을 활성화함.

- Problem

- > User level(일반적인)에서 process는 scheduling을 조작할 수 없음. (특별한 권한이 필요함.)
- > 악성 Program이 scheduling을 조작할 수 있음. (전체 system이 쉽게 손상됨.)
- > 동일한 shared resource에 접근하는 interrupt handler로부터의 보호를 받지 못함. (interrupt handler는 외부에서 HW적으로 받아옴으로 막을 수 없어 근본적인 문제를 해결하지 못함.)

```
void *mythread(void *arg)
{
    for (i = 0; i < max; i++)
        disable_scheduling();
        counter = counter + 1;
        mov 0x8049a1c, %eax
        add $0x1, %eax
        mov %eax, 0x8049a1c
        enable_scheduling();
    return NULL;
}
```



2. Interrupt handling을 비활성화함.

- > Timer interrupts를 비활성화 하면 scheduling은 자연스럽게 비활성화 됨. (Scheduling은 timer interrupt에 의해 동작)

- Problem

- > Process가 외부 이벤트에 대한 반응을 받을 수 없음. (ex> ctrl+c)

```
void *mythread(void *arg)
{
    for (i = 0; i < max; i++)
        disable_interrupts();
        counter = counter + 1;
        mov 0x8049a1c, %eax
        add $0x1, %eax
        mov %eax, 0x8049a1c
        enable_interrupts();
    return NULL;
}
```

3. Lock (Mutex)

- > Thread간의 Lock을 사용하겠다는 상호 합의가 있어야 함.
- > 단 하나의 thread가 lock을 걸고 critical section에 진입함.
- > Lock이 걸려 있으면 다른 thread들은 해당 critical section에 진입 불가.
- Lock Variable (Thread들이 하나의 variable을 공유함.)
 - > Lock state : Locked or not (0 : unlock, 1 : lock)
 - > Lock owner : Lock을 건 thread (Lock을 건 thread가 직접 lock을 풀어야 하기 때문에 owner가 있어야 함.)

```
lock_t mutex;
void *mythread(void *arg)
{
    for (i = 0; i < max; i++)
    {
        lock(&mutex);
        counter = counter + 1;
        unlock(&mutex);
    }
    return NULL;
}
```

● Simple Lock Implementation

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    Must be atomic { while (mutex->flag == 1) // TEST the flag
                    ; // spin-wait (do nothing)
                    mutex->flag = 1; // now SET it! }
    What if a thread is preempted at this point?

    void unlock(lock_t *mutex) {
        mutex->flag = 0;
    }
}
```

Note: lock ownership is not implemented

- 해당 example에는 Lock owner가 빠져 있음.
- 어떠한 global variable에 대한 race condition을 막으려고 lock을 사용하는데, lock variable도 global이기 race condition이 생기게 되어 동시에 lock을 걸어버리거나 동시에 unlock을 하는 문제가 발생함. (상호 배제 불가능)

● Hardware-assisted Method

- 위의 lock variable에 대한 race condition을 해결하기 위하여 ISA에서 근본적인 library를 제공함. (read, write가 하나의 function(instruction)에서 진행.)
 - > 상호 배제 구현을 위한 Special atomic instruction

```
Atomic int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new; // store new into old_ptr
    return old; // return the old value
}
```

- Atomic test-and-set instruction

```
Atomic int compare_and_swap(int *reg, int oldval, int newval) {
    int old_reg_val = *reg;
    if (old_reg_val == oldval)
        *reg = newval;
    return old_reg_val;
}
```

- Atomic compare-and-swap instruction

=> 하나의 Function에서 unlock 체크를 하며 lock을 거는 것이 일어나 owner를 판단하여 이중 locking 등을 방지함.

● What if lock() fails to get the lock? (다른 thread가 Lock을 걸고 있음.)

- Spinlock
 - > 연속적으로 Loop를 돌며 lock이 풀릴 때까지 lock variable을 확인함.
 - > CPU를 계속 소모하게 되어 낭비됨. (Thread가 CPU에 올라가 있어야 함.)
 - > Lock이 풀렸을 때 critical section에 접근 시에 delay가 존재하지 않음.
 - > 짧은 critical section에 유리함. (CPU를 계속 소모하기 때문)
 - > Single process system에서는 의미가 없음. (Spinlock은 context switching을 줄일 수 있는데 single process system에서는 context switching이 필수적임.)
- Mutex (Normal blockig lock)
 - > Thread 호출을 block(CPU를 놓음)하고 waiting queue서 대기함.
 - > CPU를 소모하지 않아 낭비하지 않음.
 - > Unlock() (OS kernel)이 block되어 있는 thread를 깨워줌. 따라서 block이 풀리고 CPU에 올라가는 데 delay가 존재함.
 - > 긴 critical section에 유리함. (CPU를 소모하지 않기 때문)

● Lock Granularity

- Fine-granular method
 - > 각각의 shared resource마다 lock. (one resource one lock)
 - > Lock variable이 많아지면 code의 복잡성도 증가함. (System 효율은 좋음)
 - > Memory 관리가 복잡함. (lock variable도 memory를 차지하기 때문)
- Coarse-granular method
 - > Shared resource를 그룹화하여 lock의 갯수를 줄임. (all resource one lock)
 - > Lock을 걸지 않아도 되는 shared resource도 불필요하게 lock이 걸림.
(다른 Thread가 lock을 걸어야 하는데 걸지 못하게 됨. System이 비효율적임)



● Thread Safety

- Data 구조 또는 Function의 집합이 동시에 thread들에서 안전하게 사용 가능하다면.
 - > Lock system이 내재 되어 있음.
- ex) Standard C lib function인 strtok()는 thread safe 하지 않지만, strtok_r()은 lock system이 내재 되어 있어 thread safe 함.

● Semaphore

- Integer값과 atomic한 P(), V() operation의 객체
- Binary semaphore
 - > 0 (locked) or 1 (unlocked) = Mutex처럼 사용
- Counting semaphore
 - > 0, 1, 2, ... , N (초기에 N개의 state를 지정해주어 Thread N개가 하나의 critical section에 접근할 수 있음.)

Mutex	Semaphore
• Only who locks can unlock	• There is no lock owner concept
• No multiple enters	• Multiple enters with a counter
• Only for mutual exclusion	• Can be used for other synchronization purposes

* Mutex

- lock을 건 owner만 unlock을 할 수 있음.
- 여러 thread가 하나의 critical section에 진입 불가능.
- 상호 배제

* Semaphore

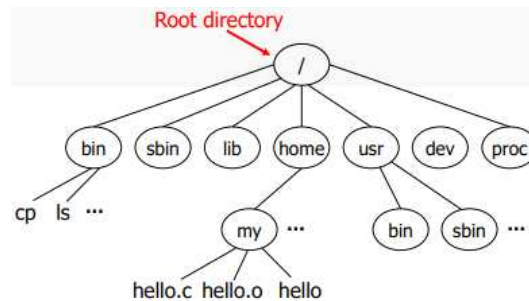
- Owner의 개념이 명확하지 않으므로 lock owner가 없음.
- Race condition에 영향이 없는 thread들은 정해진 갯수 만큼 critical section에 진입 가능.
- 여러 thread가 진입할 수 있기 때문에 다른 synchronization 용도로 사용가능.
 - > 내부에서 thread끼리의 순서를 컨트롤하기가 타이밍 조절 면에서 편함.

19. Filesystem

● Persistent Storages (HDD and SSD), Filesystem Directory Structure

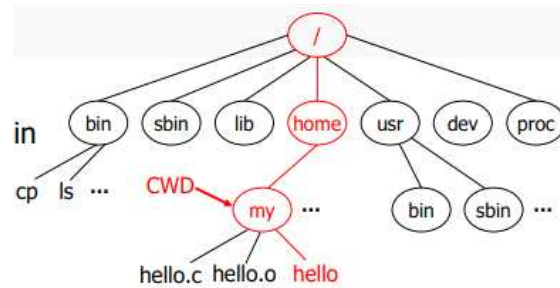
- 대용량 영구저장소에 access시,
 - > Size가 크기 때문에 address space에 맞지 않음.
 - > 많은 양의 data를 구성하는 구조화 된 방법이 필요함.
- => Filesystem : 영구저장소의 계층적 추상화 (Directories and Files)
- OS는 각자 독특한 Directory 구조를 가지고 있음.
 - > Windows = C:
 - > Linux = / (root directory)

● Linux Directory Structure



- / : root directory
- /bin : binary executables (cp, ls ...)
- /sbin : system binary executables (system 관리를 위한 실행파일)
- /lib : library files
- /home : user home directory
- /usr : OS에서 설치한 system-wide read-only files (bin, sbin ...)
 - > 초기 Unix의 hard disk 용량이 작아서 설치 파일들을 모두 root에 못 넣고 필수적인 file들만 bin, sbin에 넣고 덜 필수적인 file들을 따로 disk를 만들어 usr에 붙임. (bin, sbin과 겹치는 file X)
- /dev : device files
- /proc : OS의 정보와 상태를 보여주는 virtual files
 - > CPU info 등 system에 대한 모든 정보를 보여주기 위한 가상 file들

● Path



- Current Working Directory (CWD)
 - > 각 process가 작업 중인 directory (현재 내가 작업 중인 directory)
- Absolute path (절대 경로)
 - > root directory로부터의 path List (ex> /home/my/hello)
- Relative path (상대 경로)
 - > CWD로부터의 path List (ex> ../../bin/cp, ~/hello, ./hello)
 - > ~ : home directory // . : current directory // .. : parent directory

● Linux File and Directory

- File : Sequence of bytes

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	42	4D	7C	00	00	00	00	00	00	00	00	1A	00	00	00	0C
00000010	00	00	04	00	04	00	01	00	18	00	00	00	FF	FF	FF	FF
00000020	00	00	FF	FF	FF	FF	FF	FF	FF	00	00	00	FF	FF	FF	00
00000030	00	00	FF	00	00	FF	FF	FF	FF	00	00	FF	FF	FF	FF	FF
00000040	FF	00	00	00	FF	FF	FF	00	00	00	00	00	00	00	00	00

4 rows 4 columns (U,U) pixel

BM|.....

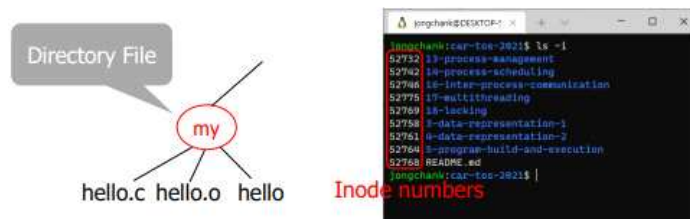
.....yyyy

..yyyyyyyy..yy

..y..yyyy..yyyy

y|..yy.....

- > OS가 보았을 때 내부 구조가 어떻게 생겼는지 알지 못함. (File 유무, size ...)
- Directory : Files name, inode numbers list를 포함하는 file



- > 해당 directory 안에 어떤 것들이 존재하는지, 어디에 위치하는지 정보를 담고 있는 special file. (물리적인 disk안의 공간 X)
- > Inode numbers : 다른 directory의 같은 file name을 구별 가능.

● File Ownership and Permission

* `ls -lh` command는 file의 소유자와 접근 권한을 보여줌.

```
$ ls -lh
total 4.0K
drwxr-xr-x 2 jck student 4.0K Nov 12 07:39 dir/
-rw-r--r-- 1 jck student 0 Nov 12 07:39 file
```

Permission Owner Group Last Modified

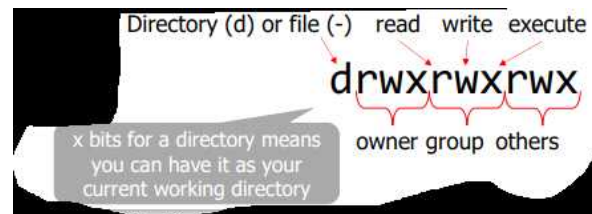
- Permission : 접근 권한, Owner : File의 소유자, Group : File을 소유한 Group

* Representing Permissions

`-rwxrwxrwx` := $111_2 111_2 111_2 \rightarrow 777_8$

`-rw-rw----` := $110_2 110_2 000_2 \rightarrow 660_8$

-> 2진수로 각자 권한을 표현하여 최종 8진수로 합쳐서 표현,



-> owner : File의 주인에 대한 권한

-> group : File을 소유하고 있는 그룹에 대한 권한

-> others : 이외의 모든 외부 사용자에게 대한 권한

-> r : read 권한, w : write 권한, x : 실행권한(CWD로 사용가능 권한)

-> d : directory, - : file

● chown, chgrp, and chmod

- `chown` : File의 주인(owner)을 변경함. (`$ sudo chown newuser filename`)

- `chgrp` : File의 소유 group을 변경함. (`$ sudo chgrp newgroup filename`)

-> 내가 속해 있는 group으로 변경시에는 sudo 필요 X

- `chmod` : File의 권한을 변경함.

```
$ chmod a+rwx file # add read, write, execute permission to all
$ chmod u+x file   # add execute permission to (owner) user
$ chmod g-r file   # delete read permission from group
$ chmod o+w file   # add write permission to others
$ chmod 660 file   # make rw-rw----
```

-> a : all, u : user(owner), g : group, o : others

-> + : add, - : delete

● Important System Calls for File Handling (OS를 통해 File handling)

- open() : open file (존재하지 않으면 create file)
- write() : write to file, read() : read from file
- lseek() : 내가 read하거나 write하고 있는 위치를 변경하여 커서를 움직임. (offset)
- fsync() : Memory에 들고 있는 file들을 disk에 모두 sync시켜줌. (flush buffer cache)
- fstat() : File의 기본정보와 상태를 불러옴.(inode number, permission, size, time ...)
- fcntl() : 기타 다양한 작업을 수행함.(file lock, ...)
- unlink() : remove file

● File Handling Example

```
int main(int argc, char *argv[])
{
    int fd;
    fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror(argv[0]);
        return -1;
    }
    write(fd, "HAHAHA", 6);
    lseek(fd, -2, SEEK_CUR);
    write(fd, "hahaha", 6);
    lseek(fd, 10, SEEK_CUR);
    write(fd, "!!!!!!", 6);
    close(fd);
    return 0;
}
```

fd can be thought as a handle to open a room (file)

Predefined fds

- 0: standard input
- 1: standard output
- 2: standard error

- > fd : File handler
(0 : standard input, 1 : standard output, 2 : standard error)
- > argv[1] : Filename, O_RDWR : File을 read, write
- > O_CREAT : 만약 File이 없으면 create
- > O_TRUNC : 만약 File이 이미 존재하면 내용 초기화
- > 0644 : Permission mask (만약에 File을 새로 만들게 되면 permission 세팅)
(-rw-r--r--)
- > write(fd, "HAHAHA", 6); : 6byte의 HAHAHA를 write.
- > lseek(fd, -2, SEEK_CUR); : 현재 내가 존재한 위치에서 2byte 뒤로 이동.
- > write(fd, "hahaha", 6); : 6byte의 hahaha를 write.
- > lseek(fd, 10, SEEK_CUR); : 현재 내가 존재한 위치에서 10byte 앞으로 이동.
- > write(fd, "!!!!!!", 6); : 6byte의 !!!!!를 write.
- => HAHAhahaha_!!!!!! (_ : 빈칸)

● Directory Handling Example

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <dirent.h>

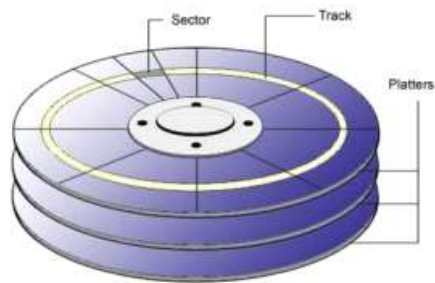
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}
```

- opendir()
- readdir()
- closedir()
- mkdir()
- rmdir()
- ...

Directory has file inode numbers and file names

- > Directory 내부에 '.', '..'이 존재함. (directory 구조)
- > opendir() : Directory를 open, closedir() : Directory를 close
- > readdir() : Directory를 read (not thread safe)
- > mkdir() : Directory를 생성, rmdir() : Directory를 삭제
- > (d->d_ino) : Directory inode number, (d->d_name) : Directory name

● Organization and Usage of HDD



- Disk는 sector로 구성된 platter set (예전 sector는 512bytes, 최근에는 4Kbytes)
- OS는 sector를 일련의 block으로 봄. (Default block size : 4Kbytes)
 - > Block : 논리적 저장영역, Sector : 물리적 disk영역, Platter : 쌓인 원판 하나
- Partitions : Disk는 여러 개의 partition으로 나누어 사용가능함.

● Organization of SSD



- Disk platter 대신에 NAND flash memory를 사용함.
- HDD와 비교했을 때 latency가 작고, 빠르고, 소음이 없고, 작음.
- HDD와 같은 OS interface(CPU가 보았을 때는 SSD와 HDD의 차이 X)

● Formatting and Mounting Disks

- Partition Formatting

-> 새로운 disk(partition) 위에 filesystem을 만듦 (mkfs command)

-> 다른 기능, 성능 특성을 가진 많은 Filesystem.

(Win : FAT, FAT32, NTFS, ...) (Linux : ext2, ext3, ext4, ...)

- Mounting

-> Kernel 초기화할 동안 root filesystem이 mounting됨.

-> mount command를 사용하여 추가 filesystem을 mount

(현재 사용 중인 directotry에 새로운 disk를 만들어 새로운 공간에 접근)



- 새로운 filesystem이 mount point에 붙는다.

- 말단의 directory에 mount를 해야 함. 상단 directory에 mount를 하게 되면 하위 directory가 다른 disk로 함께 넘어가게 되어 접근이 불가능해짐. (shadow)

● File System Structure

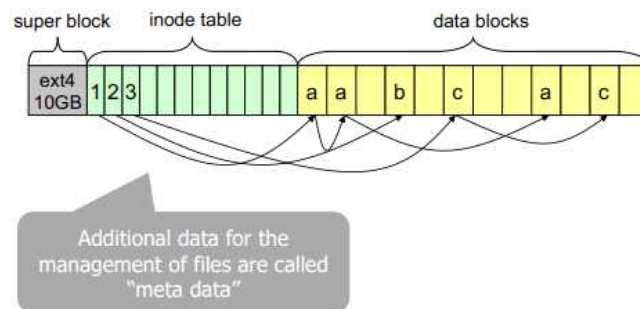
- Super block : File system size, Free area size와 같은 전반적인 정보

- Inode table

-> Inode number로 인덱싱. (Size, Permission, ...) = meta data

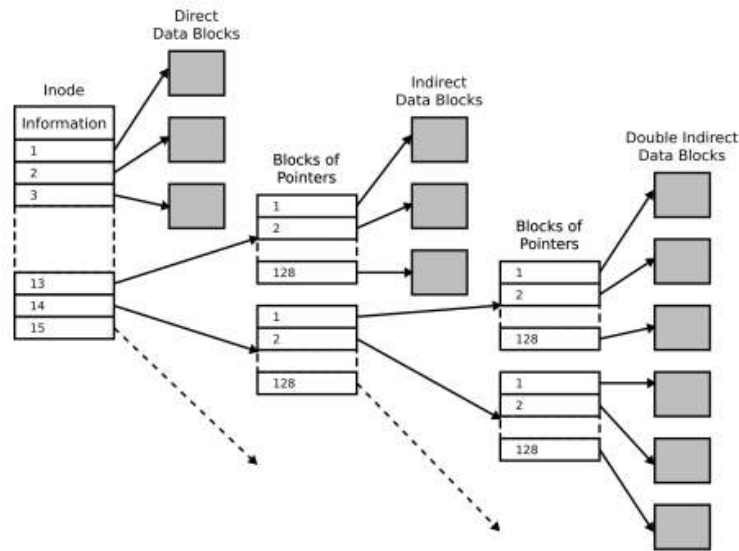
-> 대응되는 data block을 가리키고 있음.

- Data blocks : File의 내용을 담고 있음.



* File의 내용들이 연결되어 있음. (같은 file이지만 다른 block에 존재함)

● From Inode to Data Blocks



- Indirect Data Blocks, Double indirect Data Blocks를 사용하여 용량이 큰 file을 만들 시에 size의 확장이 가능함.
- Small size의 file은 Direct Blocks에만 접근하기 때문에 속도가 빠름. (File size가 커지면 Indirect Data Blocks, Double indirect Data Blocks에 접근으로 속도가 느려짐)
- Inode entry(Information)에는 하나의 File을 구성하는 Inode number가 존재하며, data가 존재하는 address를 담고 있음. Number를 나누어 Direct Data Blocks로 사용할 Inode와 Indirect Data Blocks, Double indirect Data Blocks로 사용할 Inode를 구분하여 사용함.
- Data block은 4KB(block의 기본 단위). Indirect Data Blocks, Double indirect Data Blocks를 위해 Blocks of pointers가 따로 존재함. 따라서 block의 크기와 갯수에 따라서 File의 크기의 제한이 달라짐.
- 예시로 위의 그림과 같이 Inode entry이 Direct Data Blocks를 가리키는 것이 12개, Blocks of Pointers를 가리키는 것이 3개, 그 중에서 Indirect Data Blocks를 가리키는 Block이 1개, 나머지 두개의 Blocks는 Double indirect Data Blocks를 위해 또 다른 Blocks of Pointers를 가리킨다면, $4KB \times 12 + 4KB \times 128 \times 128 \times 2 + 4KB \times 128 = 131.632MB$ 의 File을 저장할 수 있음.

● Hard Links and Symbolic Links

- Hard Link

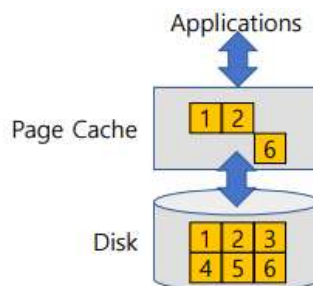
- > 실제 file과 같은 file을 더 만듦. (동일 file이지만 name이 여러개 -> 파일이 연결되어 있음)
- > File에 대한 여러 개의 Hard link가 있을 수 있음.
- > Directory간, Filesystem간의 hard link는 없음.
- > \$ ln target link : target과 link는 같은 file (한 file 수정시 같이 수정됨)

- Symbolic Link

- > 다른 file을 가리키는 작은 file. (Windows의 바로가기)
- > 다른 Filesystem에 속해 있어도 가능, Directory를 가리킬 수도 있음.
- > 어느 file을 실행할지 기록되어 있음.
- > \$ ls -l link : link에 대한 바로가기를 생성

● Buffer Cache and Page Cache

- 최근에 접근한 files 및 disk blocks에 대한 memory 내부에 존재하는 caches.
 - > 다음에 재접근 시에 접근 속도가 향상됨 (LRU, ...)
- OS로부터 관리됨.
- Buffer cache : disk blocks를 caching. (not files - Inode Table, Super block, ...)
- Page cache : Files를 caching.
- 요즘은 buffer cache를 page cache로 통합하여 사용함. (unified page cache)

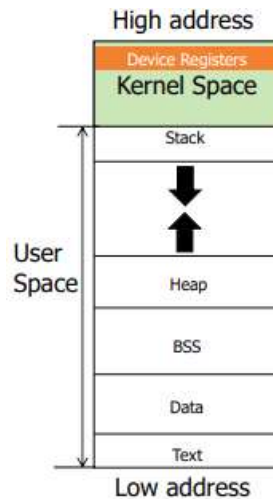


- > Page cache에서 hit가 되면 disk에 접근하지 않고 file을 reading하고 writing할 수 있음.

20. Device Driver

● Device Register Mappings in Linux

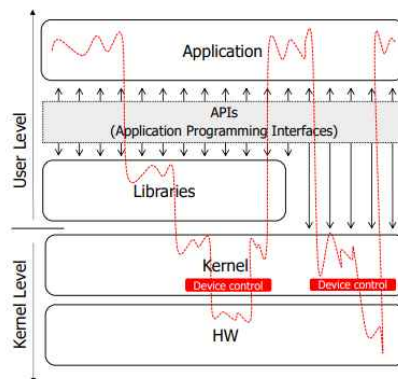
- Device registers는 address space를 통해 접근됨.
 - > Problems : 일반적인 application들은 device registers에 접근할 수 없음.



- Address space는 User space와 Kernel space로 나뉨.
- User code가 실행될 때 process는 kernel space에 접근할 수 없음.
- Device registers는 kernel space에 mapping됨.
 - > Programs는 device registers에 직접 접근이 불가능함.
 - > Program이 system call을 호출하여 kernel mode로 진입하여 kernel code를 동작하여 device register에 접근함. (정해진 interface를 통해 접근)
 - > 모든 device access는 kernel 내부에 구현되어야 함.
 - > User-level에서 device register을 user space에 mapping하여 사용하는 방법이 있으나, 이는 Linux의 standard에 어긋남.

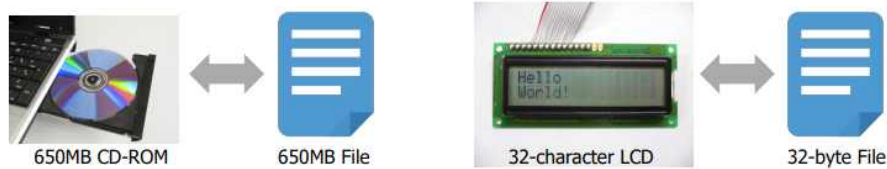
● Interface bet'n Applications and Device Drivers

- Application들은 kernel에 의해 미리 정의된 API를 통해서만 device에 접근함.
- 다양한 새로운 device들마다 새로운 API를 만드는 것은 Linux standard에 어긋나게 되어 OS가 비표준이 됨. (일반적인 API(POSIX Standard)로 모두 대응해야 함.)



● Everything is a file in Linux

- 모든 system의 device들을 모두 file로 추상화함.



- Application들은 system call의 read(), write()를 가지고 각자 device file을 read, write할 수 있음. (모든 application을 실행 가능함.)
- 각자 device마다의 특정한 기능을 수행하는 방법 (ex> CD-ROM ejection)
 - > ioctl() system call을 통해 일반적인 interface로서 각자 device마다 특정 숫자에 대응되는 기능이 구현되어 있는 부분을 실행해줌.

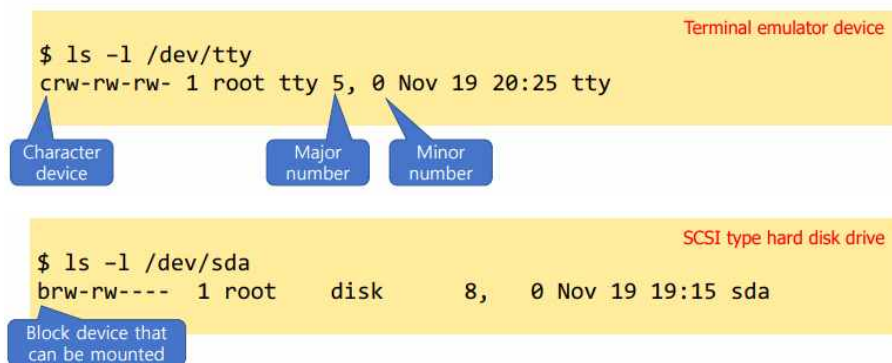
● read(), ioctl(), and write() for device files

- CD-ROM의 처음 1000bytes를 read.
 - > `fd = open("/dev/cdrom", O_RDONLY);` => device file
 - `read(fd, buffer, 1000);`
- Ejection a CD-ROM
 - > `fd = open("/dev/cdrom", O_RDONLY);`
 - `ioctl(fd, CDROMEJECT, 0);` => /usr/include/linux/cdrom.h에 int로 정의
- Writing "Hello" to LCD
 - > `fd = open("/dev/lcd", O_RDWR);` => device file
 - `write(fd, "Hello", 5);`

● Device Special File

```
$ ls /dev
autofs          hwrng          network_latency sr0      tty24  tty44  tty7    ttyS26  vcs2
block           i2c-0          network_throughput stderr  tty25  tty45  tty8    ttyS27  vcs3
bsg             initctl        null          stdin   tty26  tty46  tty9    ttyS28  vcs4
btrfs-control  input          port          stdout  tty27  tty47  ttyS0    ttyS29  vcs5
bus            kmsg           ppp           tty     tty28  tty48  ttyS1    ttyS3   vcs6
cdrom          lightnvm       psaux         tty0    tty29  tty49  ttyS10   ttyS30  vcs7
char           log            ptmx          tty1    tty3   tty5   ttyS11   ttyS31  vcsa
console        loop-control   pts           tty10   tty30  ttyS0   ttyS12   ttyS4   vcsa1
core           loop0          random        tty11   tty31  ttyS1   ttyS13   ttyS5   vcsa2
cpu_dma_latency loop1          rfkill        tty12   tty32  ttyS2   ttyS14   ttyS6   vcsa3
cuse           loop2          rtc           tty13   tty33  ttyS3   ttyS15   ttyS7   vcsa4
disk           loop3          rtc0          tty14   tty34  ttyS4   ttyS16   ttyS8   vcsa5
dri            loop4          sda           tty15   tty35  ttyS5   ttyS17   ttyS9   vcsa6
dvd            loop5          sda1          tty16   tty36  ttyS6   ttyS18   ttyprintk vcsa7
ecryptfs       loop6          sda2          tty17   tty37  ttyS7   ttyS19   uhid     vfio
fb0            loop7          sda5          tty18   tty38  ttyS8   ttyS2    uinput   vga_arbiter
fd             mapper         sg0           tty19   tty39  ttyS9   ttyS20   urandom  vhci
full           mcelog         sg1           tty2    tty4   tty6   ttyS21   userio   vhost-net
fuse           mem            shm           tty20   tty40  tty60   ttyS22   vboxguest zero
hidraw0        memory_bandwidth simple-driver  tty21   tty41  tty61   ttyS23   vboxuser
hpet           mqueue        snapshot      tty22   tty42  tty62   ttyS24   vcs
hugepages     net            snd           tty23   tty43  tty63   ttyS25   vcs1
```

- /dev에 수많은 특수 device files 존재. (device 연결을 위한 virtual files)
- 매우 종류와 갯수가 많이 때문에 device가 인식되면 user에게 띄워줌.



- > Major number : Device의 종류 (이미 system에 결정되어 있음)
(ex> Disk = 8, Terminal device = 5)
- > Minor number : 같은 Major number를 가지는 device끼리 구분을 위함.
(ex> 같은 종류의 device가 2개이면 0, 1) -> System에서 자동으로 결정

● Type of Devices

- Character devices
 - > Device 전체가 file로 추상화됨.
 - > 하나의 device가 하나의 file로 추상화 되기 때문에 directory가 없어 mount가 불가능함.
- Block devices
 - > Device는 mount가 가능한 filesystem으로 추상화할 수 있음.
 - > 방대한 크기의 device는 하나의 file로 추상화하여 사용하기 불편하기 때문에 그 위에 filesystem을 얹어서 사용함. (기본적으로 Character device의 특징)
 - > ex) 1TB hddisk
 - => Disk를 하나의 file로 보고 통째로 read()/write()하면 되지만 비효율적이므로 filesystem을 만들어서 mount를 시켜 새로운 directory를 따라 들어가서 read()/write()
 - => Filesystem을 얹어 올리기 위한 kernel level의 기능이 필요함.
(Block device layer // ex> SSD, Disk, USB, RAM...)

● Writing Device Driver

- Linux에서 device driver에 write하는 것은
- Device를 위한 read(), write(), ioctl() system call을 개발자가 잘 개발해야 함.

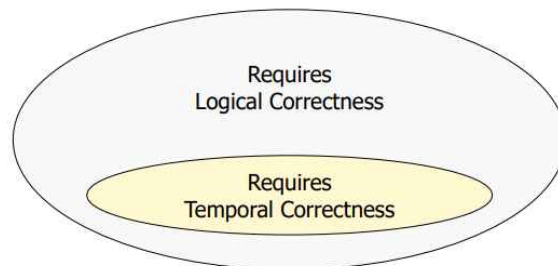
```
struct file_operations {
    ...
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ...
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    ...
}
```

- > 각 device file마다 function pointer을 부여해줌. (function의 name을 저장하여 해당 device 접근 시 system call을 통해 자동으로 kernel 내부에 있는 pointer를 호출하여 실행 시킴.)

21. Real-Time Scheduling (1/2)

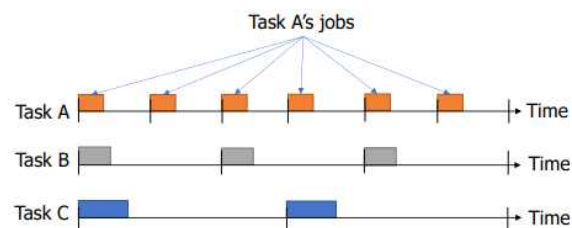
● Real-Time System

- Computing system은 논리적 정확성과 시간적 정확성을 모두 고려함.
- Logical correctness(논리적 정확성) : 올바른 output을 생성.
- Temporal correctness(시간적 정확성) : 정확한 시간에 output을 생성.



-> RTOS는 Temporal Correctness를 가장 중요하게 생각함.

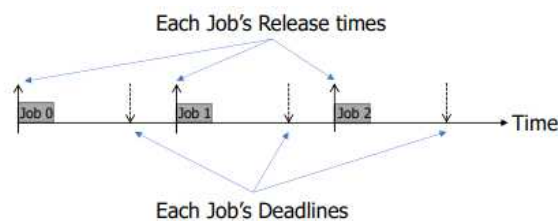
● Jobs and Tasks



- Job : unit of work, Task : 동일한 job의 주기적으로 실행되는 것
- Task A는 주기가 짧고, Task B는 좀 더 길며, Task C는 길.

● Release Time and Deadline

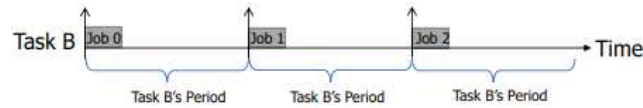
- Release Time : 어느 시점에서 Ready queue에 job을 input할 지.
-> job을 실행할 준비를 마친 시점.
- Deadline : 어느 시점에서 job이 끝나야 하는지. (RTOS에만 존재하는 개념)
-> job이 완료되어야 하는 time instance



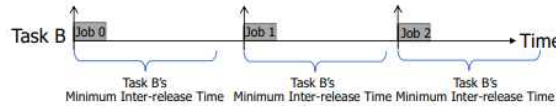
* Release time과 deadline 사이에서 job을 수행해야 함.

● More about release times

- periodic : release가 엄격하게 주기적임. (Task의 간격이 일정함.)

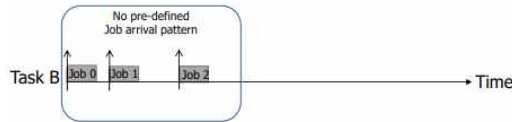


- sporadic : release가 주기적이지는 않지만, 최소한의 정해진 gap이 존재함.



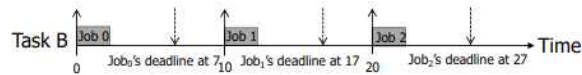
-> 최소한의 gap이 1초라면 해당 gap 사이에 release되면 안됨.

- aperiodic : release에 대한 규칙이 없음. (사용하지 않음)



● More about deadlines

- Absolute deadline (절대 시간) : Deadline을 절대시간으로 지정 (누적 시간)



- Relative deadline (상대 시간) : job이 release되어 있는 time을 time distance로 표현.



-> 간격으로 deadline을 표현함 (Release부터 deadline까지의 time이 7)

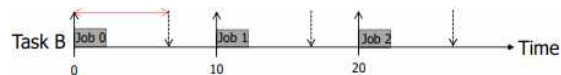
- Implicit deadline : Deadline = period -> 다음 job release time=현재 job deadline



-> period만 알려주면 deadline도 알 수 있음.

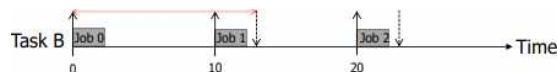
- Constrained deadline : Deadline <= period (구현이 복잡)

-> 현재 job deadline <= 다음 job release time



- Arbitrary deadline : Deadline >= period (구현이 복잡)

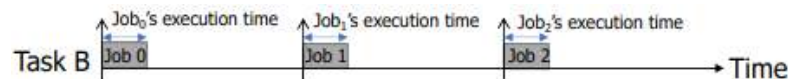
-> 현재 job deadline >= 다음 job release time



- Hard deadlines (ex> Airbag, LKAS)
 - > Deadline을 어기게 되면 제어된 환경에 치명적인 결과를 초래하는 Application 오류로 간주되는 경우. (Deadline을 어기면 큰일 남.)
 - > System을 deploy하기 전에 deadline이 100% 보장되는지를 미리 검증.
- Soft deadlines (ex> media players)
 - > Late completion으로 심각한 피해 없이 성능이 저하되는 경우. (Deadline을 어기면 큰일 나지는 않음.)
 - > System을 deploy하기 전에 deadline이 100% 보장되는지를 미리 검증 불가능. (확신을 할 수 없음)

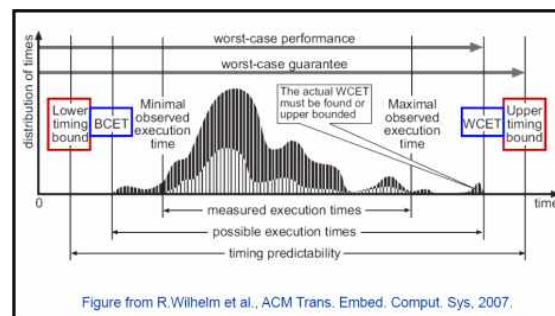
● Execution Times (Preemption 되지 않고 CPU를 온전히 다 사용한다는 가정에서)

- Each job's execution time : Application마다 input과 system state(ex> if-else)가 다르기 때문에 동일한 job의 execution time은 다를 수 있음. (항상 같다는 보장 X)
- Each task's execution time : job execution time에 따라 task execution time이 달라짐. 또한 cache와 TLB등의 영향으로 달라짐.



● More on execution times

- Stochastic(확률적) execution time
 - > Task execution time이 일정하지 않아 확률적 분포로 표현될 수 있음.
- Deterministic(결정적) execution time
 - > 일반적으로, WCET(worst-case execution time)만 유의미함.



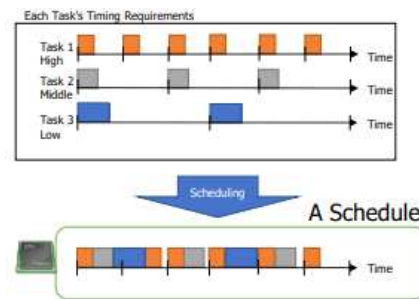
- > 흰색 빗금 부분 : measured task execution times
- > 검정색 빗금 부분 : possible task execution times (알아내는 것이 불가능)
 - BCET : Best-case execution time
 - WCET : Worst-case execution time
- > WCET를 기준으로 하여 system의 용량과 안정성을 선정하는데, 이를 알아내는 것이 불가능하여 예측하여 선정함.

● Periodic Task Model

- N개의 periodic task가 존재함. $\{T1, T2, \dots, TN\}$
- 각 T_i 는 세 개의 tuple로 표현됨. (p_i, e_i, d_i)
 - > p_i : period
 - > e_i : WCET(Worst-Case Execution Time)
 - > d_i : hard relative deadline
- T_i 가 implicit-deadline task라면, Deadline = period이므로, $p_i = e_i$
 - > (p_i, e_i) 만 존재하면 표현 가능.

● Scheduling

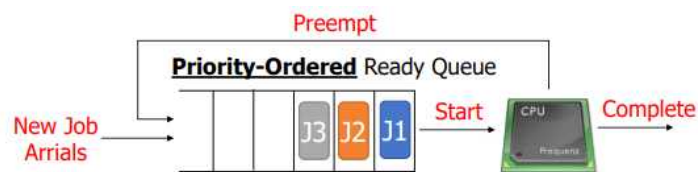
- Schedule : Time domain에서 주어진 resource에 job을 할당.
- Scheduler : Scheduling algorithm을 구현하는 module.



-> Single core에서의 scheduling

● Priority-Driven Scheduling

- 각자의 job은 고정된 우선순위를 가짐.
- Ready queue는 우선순위에 따라 job들이 정렬됨. (높은 우선순위의 job이 head)
- 항상 가장 높은 우선순위를 가진 job을 선택함.
- Scheduling은 job이 release될 때, job이 모두 끝나서 CPU에서 빠질 때만 수행하면 됨.
 - > 현재 돌아가는 job이 우선순위가 낮아질 때까지 계속 CPU에서 동작하며 우선순위는 fix되어 있기 때문에 기존 것들에 대한 우선순위를 계속 체크하여 scheduling할 필요가 없음.

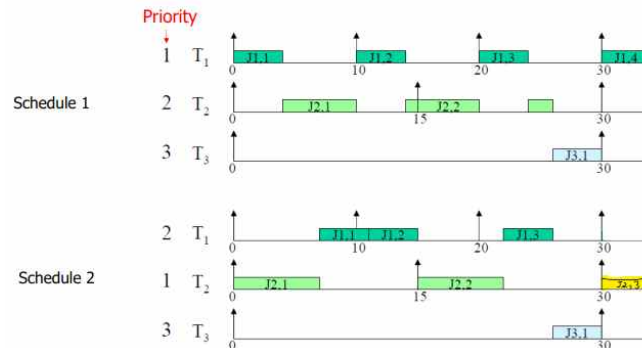


=> 우선순위가 더 높은 job이 arrivals되면 기존에 CPU에 있던 job을 preempt.

● Example Fixed-Priority Schedules (implicit-deadline)

- $\{T_1 = (p_1 = 10, e_1 = 4), T_2 = (p_2 = 15, e_2 = 7), T_3 = (p_3 = 30, e_3 = 4)\}$

-> workload



-> Schedule 1에서 T_1 이 우선 순위가 높음. T_1 이 끝나는 deadline에서 T_2 가 period되고, T_1 의 period가 10이기 때문에 T_2 가 time 7이 아닌 time 6까지 work하는 중에 T_1 이 preempt하여 T_2 의 time 1을 뺏음. 이후 scheduling이 계속 되다가 T_2 에서 마저 실행하지 못한 time 2를 T_1 이 종료된 후 실행하고, T_2 의 time 2가 모두 실행되면 T_3 가 실행됨. period가 가장 긴 T_3 가 모두 실행되면 이후 주기에서는 해당 순서가 반복됨. (time을 기준으로 우선순위를 줌.)

-> Schedule 2에서 T_2 가 우선 순위가 높음. T_2 이 끝나는 deadline에서 T_1 이 period됨. T_1 의 period가 10인데 T_2 의 period가 15이므로 T_1 은 우선순위에 따라 job이 실행되게 되며 time 2만큼의 차이로 deadline을 지키지 못해 deadline miss가 남. period가 가장 긴 T_3 가 모두 실행되면 이후 주기에서는 해당 순서가 반복됨. (work의 종류에 따라 우선순위를 줌.) => Deadline miss로 문제가 발생.

* Task의 우선순위 결정 방법

- Criticality ordered

-> T_1 is a brake control task

-> T_2 is a speed display task

=> Job 내부의 내용을 기준으로 우선순위 결정.

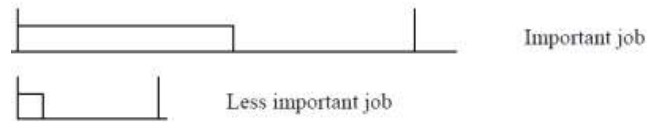
- Urgency ordered

-> T_1 (100ms, 2ms)

-> T_2 (1000ms, 15ms)

=> 시간(급한 순서대로)을 기준으로 우선순위 결정. (주로 사용)

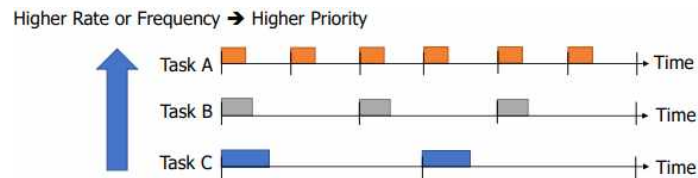
● Priority vs. Criticality



- Criticality : 기능이 우선적으로 중요함.
- RTOS는 Importance와 무관하게 deadline을 만족시켜야 하기 때문에 실행 시간에 따라 우선순위를 줌. (기능으로 우선순위를 주면 Time이 급한 job 실행을 못할 수 있음.)

● Optimal Priority Assignment

- RM(Rate Monotonic) : implicit deadlines에서 periodic task들이 고정된 우선순위를 가질 때 최적.
- > 주기가 빠를수록 높은 우선순위를 줌.

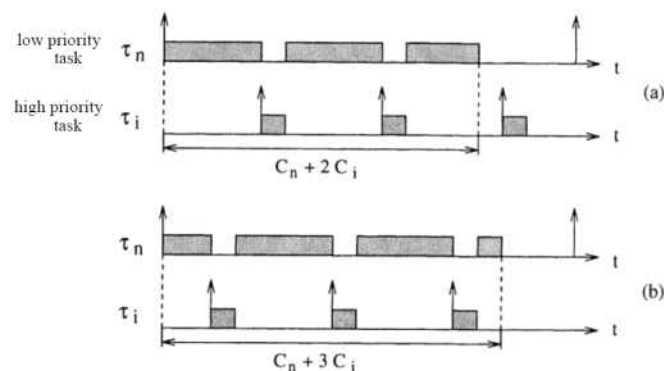


● RM Optimality

- RM으로 scheduling할 수 없으면 어떠한 것들로도 scheduling할 수 없음.
- > RM으로 scheduling하고 RM으로 돌려 보았을 때 Deadline miss가 난다면 어떠한 것들로도 scheduling할 수 없음.
- * RM은 우선순위가 고정인 domain에서 implicit-deadline일 경우 optimal함. 우선순위가 동적인 경우 EDF(Earliest-Deadline First)가 RM보다 optimal함.

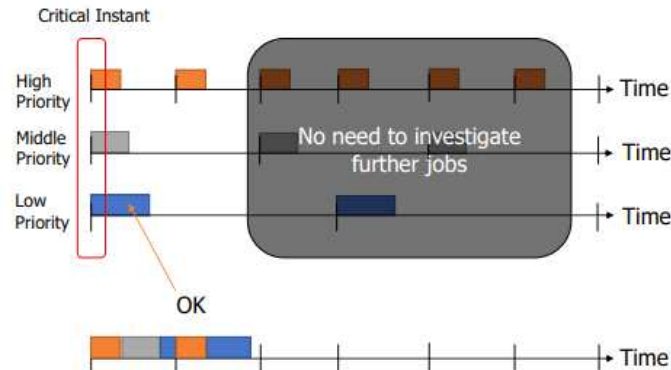
● Critical Instant Theorem

- 가장 최악의 Scenario는 모든 Task가 동시에 release될 때임.



- > 기존에 preemption이 두 번 일어나던 scenario에서 우선순위가 높은 task의 주기를 뺏기면 preemption이 세 번 일어나게 됨. (동시에 모든 Task가 시작되면 preemption은 최악.)

● Critical Instant Theorem

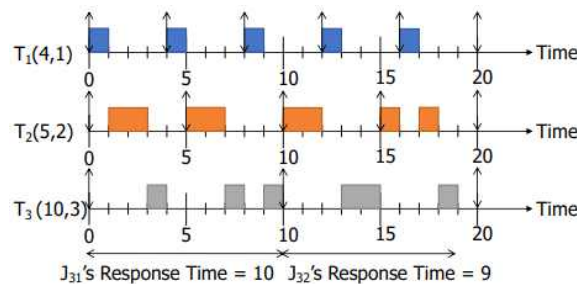


- > 모든 Task가 동시에 release되는 최악의 경우를 만들어 deadline을 검증.
- > 최악의 경우에서 Deadline에 문제가 없다면 뒤의 경우는 확인하지 않아도 됨.

● Response Time

- A job's response time

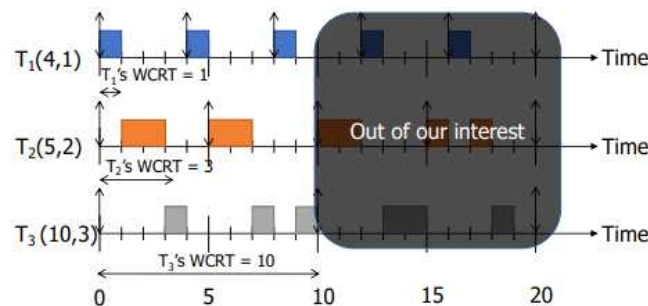
- > Release time부터 completion time까지의 time (Delay로 인한 time 포함)



- > 처음에 동시에 task들을 실행시켰을 때 가장 deadline이 긴 T3가 범위 내에 들어옴으로 문제가 없음.
- > T3의 J31은 time 3가 time 10까지 주기로 실행되기 때문에 response time=10
- > T3의 J32는 time 3가 time 9까지 주기로 실행되기 때문에 response time=9

● WCRT (Worst-Case Response Time)

- WCRT : Task가 가질 수 있는 가장 긴 response time.
- Critical instant에서의 response time은 WCRT.



● Schedulability Check (Response Time Analysis)

- 모든 Task마다 T_i 의 WCRT인 r_i 를 계산함.
- 모든 $r_i \leq p_i = d_i$ 의 경우 RM으로 모두 scheduling이 가능함.

● Utilization Bound

- Utilization : Periodic한 task들에 대해 utilization U 는 다음 수식을 이용해 계산.

$$U = \sum_{1 \leq i \leq N} \frac{e_i}{p_i}$$

-> (WCET / period) = 주기 내에서 얼마나 CPU를 demand하는가.

-> 전체 System이 몇 %의 CPU를 사용하는가.

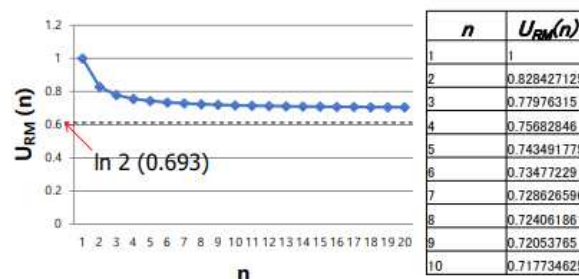
- Scheduling이 가능한 utilization bound : U_{bound}
 - > $U \leq U_{\text{bound}}$: task들은 scheduling이 보장됨.
 - > $U > 100\%$: 어떠한 방법을 사용해도 scheduling이 되지 않음.
 - > 또한 $U \leq U_{\text{bound}}$ 여도 동시에 task가 몰리게 되면 deadline miss가 발생.

● RM Utilization Bound (aka L&L bound)

- implicit-deadlines인 n 개의 task에서 RM의 utilization bound는 다음과 같음.

$$U_{RM}(n) = n(2^{1/n} - 1)$$

- Period와 execution time을 사용하지 않고 계산함.



- > 69.3%에 수렴함. 최종 $U_{RM}(n)$ 이 69.3%보다 크면 scheduling이 될 수 있을지 알 수 없음.

● Utilization Bound Check

- 충분조건
 - > $U \leq U_{RM}$: Scheduling이 가능함.
 - > $U > U_{RM}$: Scheduling이 될지 안 될지 알 수 없음.
 - > 만약 후자의 경우 그림을 그려서 WCRT를 확인하여 결정해야 함.
- Response time 분석과 같이 정확하지 않지만 단순한 one-shot 해결책이고, online admin 시험(채용)에 적합하기 때문에 유용함.

22. Real-Time Scheduling (2/2)

● Two Problems Caused by Locks

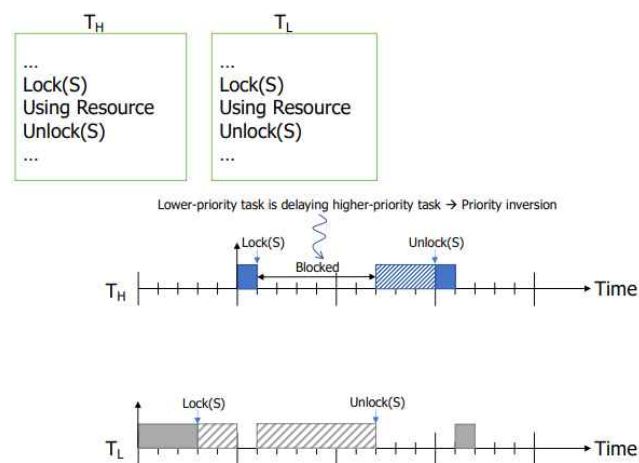
- Priority Inversion

-> 우선순위가 높은 Task가 우선순위가 낮은 Task에 의해 block됨.

- Deadlock

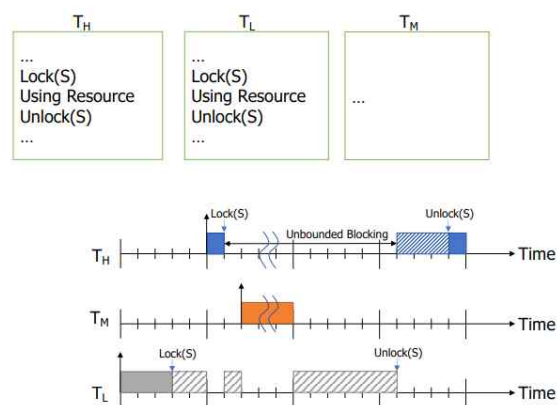
-> 두 개 또는 많은 Task가 각자의 lock에 대한 unlock이 될 때까지 기다림.

● Priority Inversion



-> 우선순위가 높은 Task가 우선순위가 낮은 Task에 의해 lock을 걸지 못하게 되어 delay됨 (Blocked).

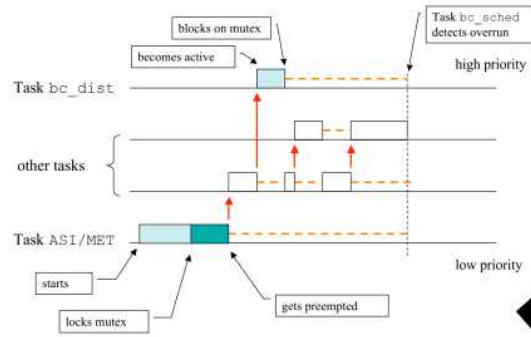
● Unbounded Priority Inversion



-> Priority Inversion이 될 때 우선순위가 High와 Low의 중간의 Task가 많이 존재하면 문제가 크게 작용됨.

-> 많은 Middle priority task가 High Task를 Delay시킴. (High와 Middle은 lock에 대한 아무 연관이 없는데 Middle에 의해 High Task가 간섭을 받기 때문에 문제가 생김. -> High priority Task가 계속 Unbounded Blocking됨.)

Priority Inversion on Mars Pathfinder

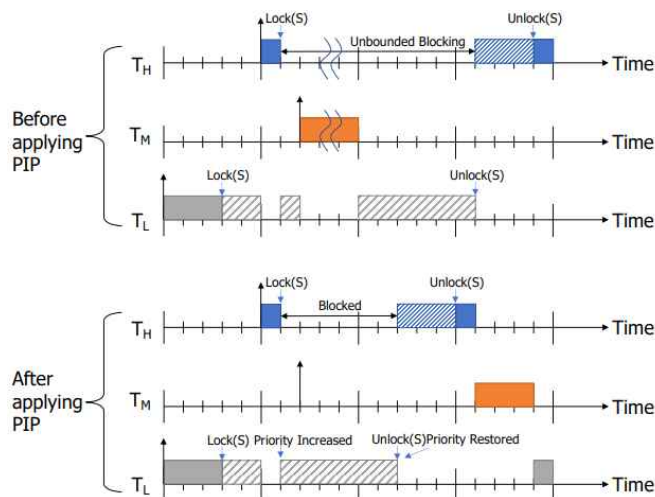


-> Middle Task가 High Task를 block시킴

=> 결론적으로 middle task가 low task이 unlock되지 않았을 때 preempt하여 high task가 low task가 걸은 lock을 풀지 못하여 block되어 middle task가 계속 실행이 됨.

● Priority Inheritance Protocol(PIP)

- 우선순위를 상속시킴.

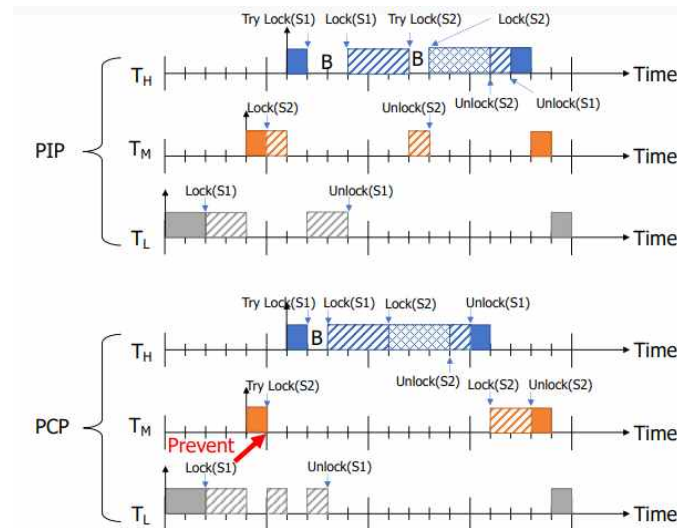


-> High priority task가 실행되려고 했는데 lock을 low priority task가 걸어두어 실행이 되지 못하는 시점에 low priority task에게 high priority를 상속하여 low priority의 priority를 증가시켜 middle priority task가 우선순위에 따라 실행되지 않도록 함. (on demand시 고려함.)

-> Chained Blocking : lock의 종류가 많아지게 되면 blocking이 많이 발생하여 delay가 계속해서 생김.

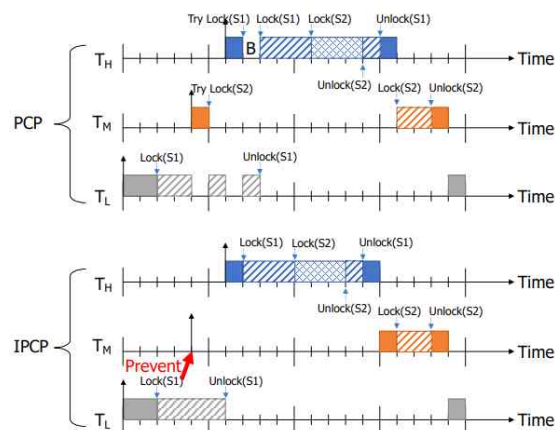
-> Deadlock : 많은 Task가 wait하게 되어 deadlock이 생김.

● Priority Ceiling Protocol (PCP) -> AUTOSAR의 standard (실제 구현은 IPCP)



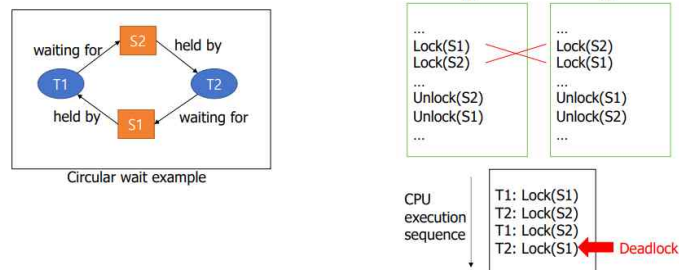
- > Low priority task가 lock이 걸린 시점부터 lock을 걸 수 있는 priority의 ceiling(우선순위 기준)을 lock을 사용하는 Task 중 high priority보다 더 큰 priority로 설정하여 해당 priority보다 priority가 높지 않은 모든 task들이 lock을 걸지 못하게 함.
- > Ceiling 밑의 우선순위를 가진 Task에서의 모든 locking을 막음. (Lock을 사용하는 Task들을 알고 있어야 함.)
- > Low priority task에서 S_1 Lock이 걸리고 Middle priority task가 실행되고 S_2 Lock을 걸려고 해도 ceiling이 높게 변경되었기 때문에 prevent됨. 따라서 high priority task가 우선순위에 따라 긴 delay없이 실행되게 됨.
- > 어떤 Task가 lock을 사용하는지 미리 알고 있어야 하기 때문에 system 설계 시 고려하고 runtime에는 OS가 자동으로 판단해줌.

● Immediate Priority Ceiling Protocol (IPCP)



- > PCP와 동일한 방법으로 ceiling을 높이지만 IPCP에서는 middle priority task의 실행 자체를 막아버려 더 강력하게 대응함.

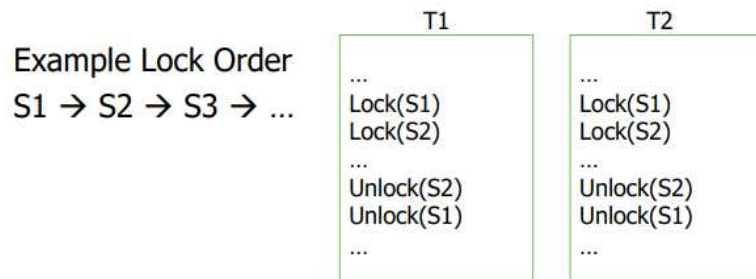
● Deadlock



- > T1이 S1 lock을 걸고 T2가 S2 lock을 건 상태에서 T1이 S2 lock을 걸려고 시도를 하여 T1이 S2 lock이 풀릴 때까지 대기 상태에 들어가 있는 동안 T2가 S1 lock을 걸려고 시도를 하여 T2도 S1 lock이 풀릴 때까지 대기 상태에 들어가 서로 lock이 풀리는 것을 기다리게 됨. (중첩하여 lock을 걸려고 서로 시도함.)
- > Task의 실행이 되지 않아 system이 멈춤.

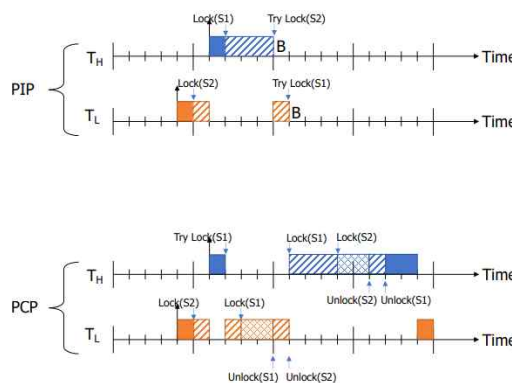
● Lock Ordering

- 중첩하여 lock을 걸려고 시도 (nested locks)할 때 미리 정해진 lock ordering을 따라 미리 lock의 순서를 정해주어 Deadlock을 방지함.



- > System적인 해결 방안이 필요함 : lock의 수가 많아지면 모두 우선순위를 주는 것이 힘들.

● PCP for Deadlock Prevention



- > PCP를 사용하면 자동으로 Deadlock이 해결됨. low priority task가 lock이 걸리면서 Ceiling이 높아지며 OS가 high priority task가 S1에 lock하려는 작업을 막기 때문.