

리눅스 커널(운영체제) 강의노트[3]



aeharvlee

Nov 18, 2018 · 23 min read

2장에서 설명했던 `fork()` 의 작동 원리에 대해서 이어서 설명한다. `fork()` 뿐만 아니라 이번 3장에서는 다양한 시스템 콜에 대해 학습한다. 또한 데몬(**Daemon**)과 서버(**Server**)에 대해서도 간단히 학습할 것이다.

시작하기 앞서 이번 강의에서 등장할 그림들에 오류가 있다는 점을 언급하고 싶다. 오류가 있는 부분은 별도로 빨간색으로 마크해서 원래 있어야 할 곳으로 표식을 해놓거나 중간 중간 어떤 부분에 오류가 있는지를 언급을 했으니 부디 설명을 읽으면서 헛갈리지 않길 바란다.

1. 주요 시스템 콜 동작 원리

1.1 Fork(2)의 동작 원리

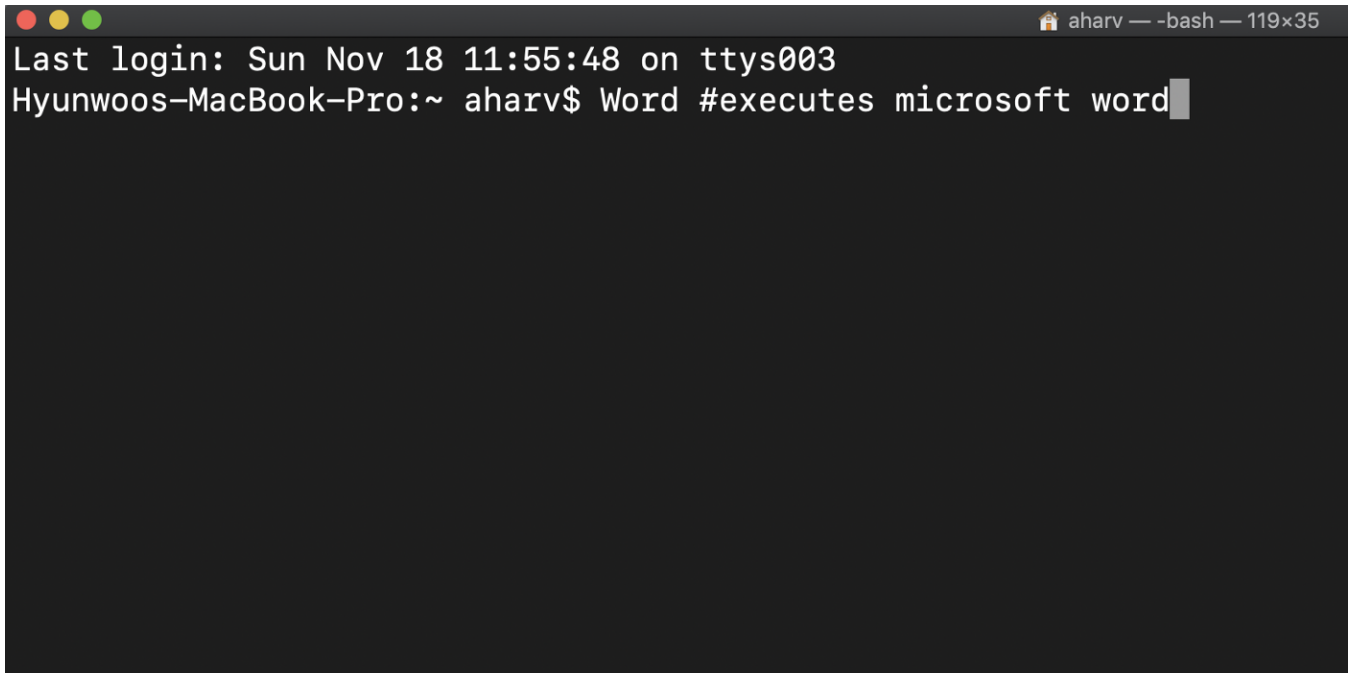
`fork(2)` returns twice (once to parent, once to child)

```
main()
{int pid;
  printf("I am parent!\n");
  pid = fork();
  if (pid == 0)          /* this is child */
    printf("I am child\n");
  else                  /* this is parent */
    /* parent performs other work */
}
```

- Alloc & copy PCB
- Alloc & copy `a.out`

그림에서 수정된 사안이 하나 있는데, **printf("I am parent!\n")** 부분이 **else** 구문에 속해야하는 것이 맞다. 이점을 주의해서 아래 설명을 보자.

위의 소스코드로 동작하고 있는 프로그램이 **셸(Shell)** 프로그램이라고 해보자. 셸 프로그램은 사용자로부터 입력을 기다리고 입력된 명령을 토대로 프로그램을 실행하는 교통 정리 프로그램이라고 우리는 배웠다. 셸이 시작되면 명령어를 입력할 수 있는 터미널 혹은 프롬프트 창이 등장할 것이고 셸은 터미널 혹은 프롬프트 창에 사용자의 명령이 입력되기를 기다리고 있다. 아래와 같은 화면을 생각하면 된다.



```

Last login: Sun Nov 18 11:55:48 on ttys003
Hyunwoos-MacBook-Pro:~ ahurv$ Word #executes microsoft word
  
```

우리가 셸에 Microsoft의 Word 프로그램을 실행시키는 **word**라는 명령을 터미널에 입력했다고 해보자. 입력된 명령어를 받은 셸은 가장먼저 **fork()**를 진행한다. **fork()**를 호출하면 자식 프로세스가 생성되면서 부모 프로세스와 완전히 동일한 소스코드(**image**) 갖게된다. 코드 뿐만 아니라 부모 프로세스의 **PCB(Process Control Block)**도 그대로 물려 받는다. (PCB에 대해서는 2장 강의노트에서 자세히 다뤘으니, 기억이 나지 않는다면 2장 강의노트에서 PCB 키워드로 검색을 해서 확인하길 바란다.)

fork() 는 두번 리턴된다. 한 번의 리턴은 자식 프로세스에게 0값을 리턴하고 나머지 한 번은 부모 프로세스에게 자식 프로세스의 프로세스 아이디값을 리턴한다.

아직은 부모 프로세스가 **CPU**를 점유하고 있기에 **fork()**로부터 리턴된 **pid**값은 자식 프로세스의 **pid**값이고 부모 프로세스는작업 **printf("I am parent!\n")** 을 마저 진행한다. 자식 프로세스의 **pid**값을 리턴 받음으로써 부모 프로세스는 자식 프로세

스를 알고 통제할 수 있는 것이다. 부모 프로세스로부터 복제되어 생성된 자식 프로세스는 현재 **ready queue**에서 **CPU**가 자신에게 할당되기를 기다리는 중이다.

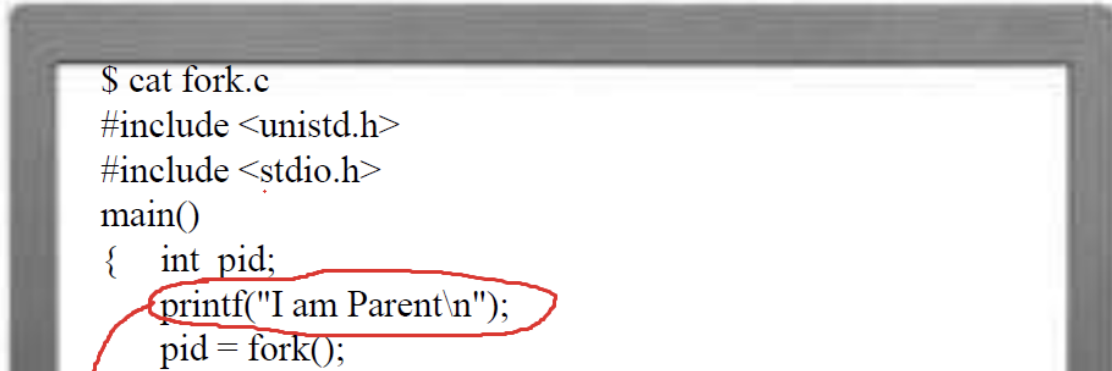
앞서 2장에서 **fork()**는 두 번 리턴된다고 설명한 바 있다. 첫 번째 리턴에서는 자식의 **pid(Process Id)**를 리턴하므로 if 조건문을 건너 띄고 else 구문으로 넘어간다. else 구문으로 넘어가면 **printf("I am parent\n");**가 실행되고 모니터 화면에는 **I am parent**가 나타나게 될 것이다. 작업을 다 마친 부모 프로세스는 종료가 된다.

이후 **CPU**의 점유권은 자식 프로세스에게 넘어가게 된다. 이론상 **ready queue**에 대기하고 있던 다른 프로그램들이 없었다고 가정한다면, 부모 프로세스가 끝남과 동시에 자식 프로세스는 **CPU**를 쥐게 된다.

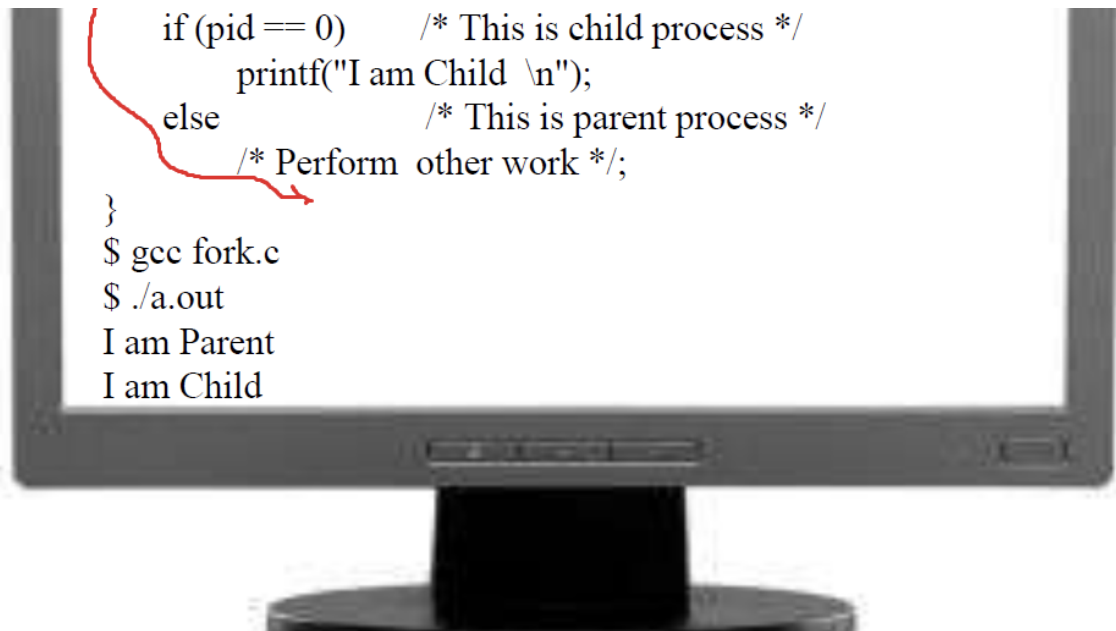
자식 프로세스는 어떻게 동작할까? 위에서 **fork()**가 실행되면서 부모의 코드(이미지) 뿐만 아니라 **PCB**를 통째로 복사했기 때문에 다음에 어디서부터 실행해야 할지 알려주는 **PC(Program Counter)**와 **SP(Stack Pointer)** 등 또한 복사되었다. 즉 **PCB**에 존재하는 **State Vector Save Area** 영역 (이하 **state vector**로 서술함)에 있는 **PC**와 **SP** 등을 복사했기 때문에 자식 프로세스의 코드가 실행될 때는 맨 처음부터 실행되는 것이 아니라 **fork()** 중간에서부터 다시 진행하게 되어 있다.

대부분의 프로그램은 초기 실행될 때 **main()** 부터 시작한다. **PCB**에 그렇게 명시되어 초기화가 되기 때문이다. 하지만 지금 다루고 있는 자식 프로세스의 경우는 **PCB**에서 가리키고 있는 다음 실행주소(**Program Counter**)가 **fork()**에 있었기 때문에, 자식 프로세스는 **fork()** 중간 영역부터 진행한다. (중간 영역이라는 건 **fork()** 함수가 한창 진행중일 때 복사가 일어났으므로, 그 진행중이었던 파트부터 다시 진행된다는 의미로 해석하면 된다.)

자식 프로세스가 **fork()**에서 리턴되면, 자식 프로세스 코드 안의 **pid** 변수는 **0**의 값(자식 프로세스 **pid**는 보통 **0**)을 가지기 때문에 **I am Child \n**이 화면에 출력되게 된다. 지금까지 다룬 내용을 다시한 번 정리하면서 아래 그림을 살펴보자.



```
$ cat fork.c
#include <unistd.h>
#include <stdio.h>
main()
{   int pid;
    printf("I am Parent\n");
    pid = fork();
```



위에서 수정했던 것과 마찬가지로 일단, `printf("I am Parentn")` 는 `else`문에 속해 있어야 하는 것을 염두하고 살펴보면, 첫 번째 출력값인 **I am Parent**는 일단 부모 프로세스가 시행한 작업이다. 그리고 부모프로세스가 끝나면서 자식 프로세스가 CPU를 점유하게 되면서 `fork()` 로부터 리턴 값을 받아 `if`문 조건을 만족하게 되고, `printf("I am Child n")` 를 실행하게 된다. 따라서 **I am Child**는 자식 프로세스가 시행한 작업이라고 할 수 있다. `if`문 끝단에 있는 `execlp` 구문 같은 경우는 바로 아래에서 이어서 설명한다.

1.2 Exec(2) 동작 원리

Example: `exec()`
Try *man exec*

```

main()
{int pid;
    printf("I am parent!\n");
    pid = fork();
    if (pid == 0)          /* this is child */
        {printf("I am child! Now I'll run date \n");
         execlp("/bin/date", "/bin/date", (char *) 0);
        }
    else                   /* this is parent */
        /* parent performs other work */
}
  
```

- Alloc & copy PCB
- Alloc & copy a.out

- load a.out from disk
- init a.out

exec(2) 시스템 콜에 대해 알아보기 전 몇 가지 배경지식에 대해 먼저 짚어보고자 한다. 위 그림을 보면서 함께 설명을 따라가보자. 먼저 `exec()` 에 매개변수를 살펴보면, `/bin` 이 보인다. `/bin` 은 바이너리(**binary**) 파일만 모아둔 폴더(**directory**)를 의미한다. 그 폴더 안에는 바이너리 프로그램들이 수 십개가 존재하고 있는데, 그 바이너리 프로그램 마다 원래는 `a.out` 의 형식으로 되어 있지만 그 이름을 각자의 프로그램 제작사의 입맛에 맞게끔 설정해 놓았다(**ls, cat, hwp, ppt** 등).

코드를 살펴 보면 자식 프로세스 차례가 왔을 때 `I am child!` 부분의 출력문을 출력하고, **execlp(exec 계열 함수)**를 실행하게 되어 있다. `exec` 시스템 콜은 현재 돌아가고 있는 프로세스 위에 자신의 프로세스로 완전히 덮어씌어(**over write**) 버린다. 덮어쓴 후 **exec** 매개변수로 왔던 그 프로그램의 **main()**으로 가는 것이 `exec`의 작동 원리다.

새로운 프로세스가 생기는 것이 아니기 때문에, **pid(Process Id)**는 변하지 않는다. 다만 프로세스를 구성하는 코드(기계어 코드)와 데이터, 힙, 그리고 스택 영역의 값들이 **exec**으로 발생하는 새로운 프로그램의 것으로 바뀌게 된다.

```

$ cat exec.c
#include <unistd.h>
#include <stdio.h>
main()
{
    int pid;
    printf("I am Parent\n");
    pid = fork();
    if (pid == 0)    /* This is child process */
    {printf("I am Child \n");
     execlp("/bin/date", "/bin/date", (char *) 0);
    }
    else /* This is parent process - Perform other work */;
}
$ gcc exec.c
$ ./a.out
I am Parent
I am Child
2010. 08. 21. (i† ) 18:48:15 KST
  
```



설명은 위와 동일하다. **exec**은 자신의 프로세스를 현재 진행 중인 프로세스 위에 덮어 써버린다. 덮어 씌움과 동시에 **date**의 **main()**으로 넘어가는 것이고, 그 쪽에서 날짜를 출력해주는 작업을 진행한다. 그래서 유닉스나 리눅스에서는 프로세스의 생성이 **fork()**하고 **exec()**을 하는 두 스텝으로 존재한다.

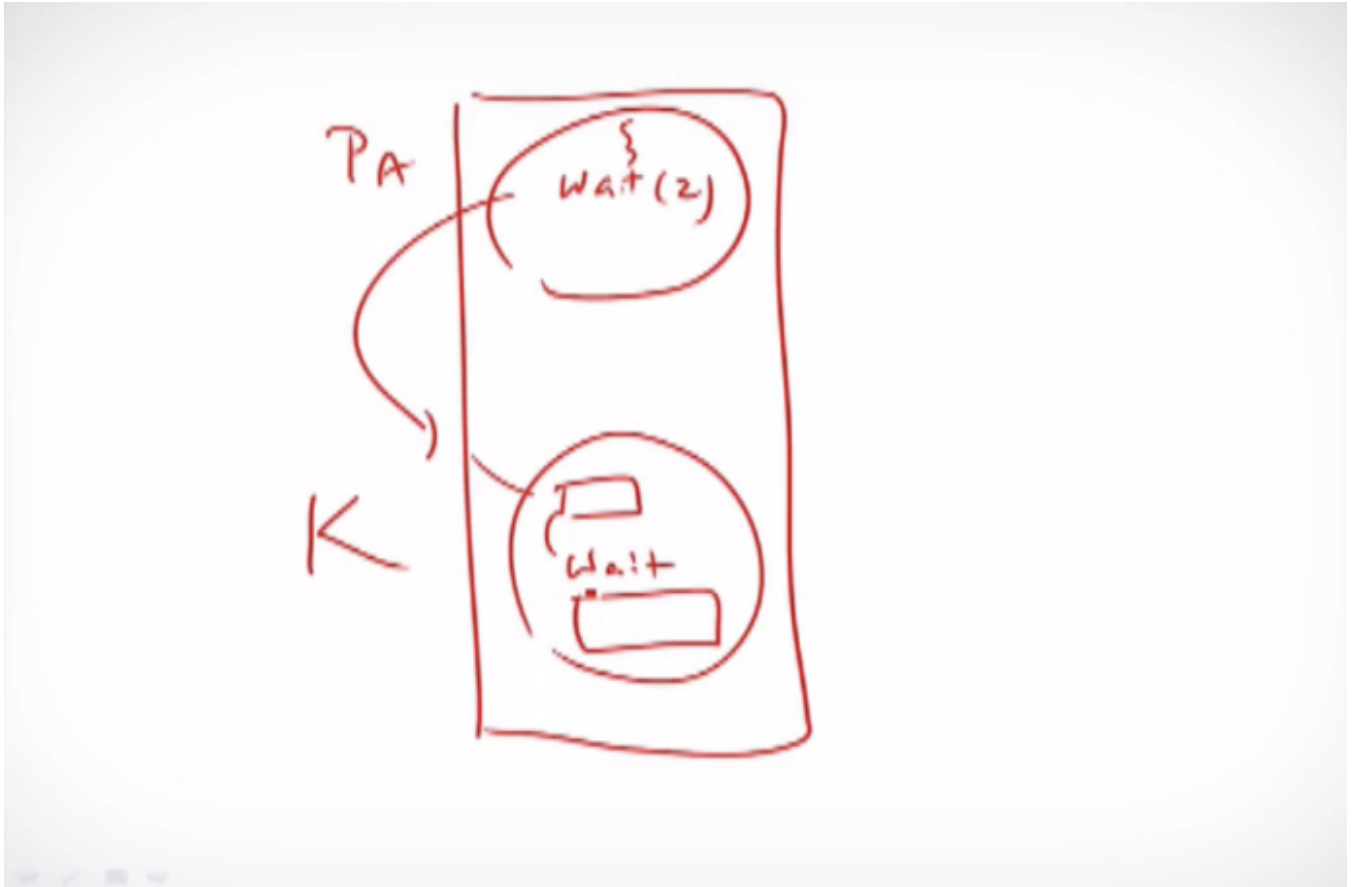
fork()는 **image**(= 소스코드)와 **PCB**를 전부 복사하는데, **exec()**의 경우에는 현재 **image**에 새로운 실행(**execute**)코드를 디스크로부터 바이너리 파일 형태로 가져온 후에 현재 **image**에 덮어 씌우기(**over write**)를 진행하고 자신 프로세스의 **main()**으로 진행하는 것이다. 한마디로 기존의 작업하던 것을 자신의 프로그램으로 갈아 치우고 자신의 프로그램을 가동시키는 행위라고 할 수 있다.

1.3 Wait(2) 동작 원리

wait(2) system call

- If P_A invokes *wait()* → then kernel **blocks** P_A (preempt CPU)
 - until child terminates (waiting for signal from child)
 - child runs (i.e. CPU is given to child)
 - Eventually child terminates, kernel **wakes up** parent
 - kernel puts parent into **ready queue**
 - later, parent is **dispatched** (give CPU)

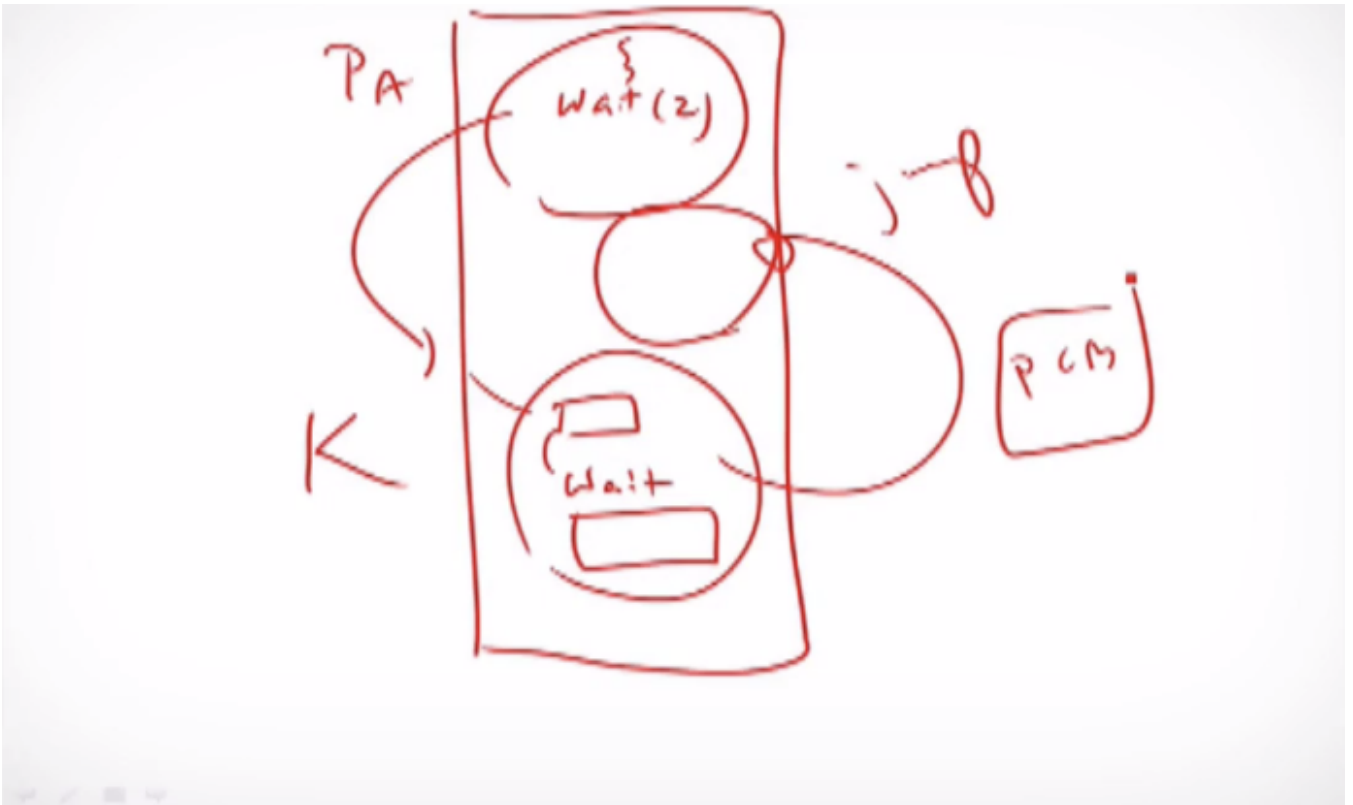
시스템 콜은 결국 커널모드로 진입하는 것을 뜻한다. 위 그림을 보면서 **wait()**에 대해 알아보자. 어떤 프로그램이 `wait()`를 호출하면 해당 프로그램의 **CPU** 사용권한을 박탈한다. 위 그림의 본문 첫 줄에 등장하는 것처럼 프로세스 **P_A**로부터 **CPU** 사용 권한을 박탈한다(**preempt**).



임의의 프로세스 A(위 그림에서 **P_A**라고 표현되어 있음)가 `wait(2)` 시스템 콜을 호출하면 커널모드(**K**)의 트랩 핸들러(**Trap Handler**)에 진입하여 **wait()** 시스템 콜 실행을 하게 되는데, 이때 시스템콜을 호출한 프로세스로부터 **CPU**를 뺏는다(**preempt**).

풀어쓰자면, 커널은 보통 자신의 작업을 다 하고 나면 호출한 프로세스의 유저 모드로 돌아가야 하는데, 유저모드로 돌아가지 않는다.

커널이 아닌 프로그램은 자신의 주소(address)에 한정되서 `read`, `jump` 등을 할 수 있지만 커널은 어디로든 가고 **jmp**(점프)할 수 있기 때문에 `ready queue`에 가서 준비된 프로세스 중 우선순위가 가장 높은 프로그램의 **PCB**를 찾아서 **PC(Program Counter)**를 알아낸 후에 **PC**(프로그램 카운터)가 가리키 쪽으로 가는 것(**jmp**)이다. 이 과정이 **preempt**라 부른다.



그 아래의 그림을 살펴보자. 이번에는 부모 프로세스에 초점을 맞춰서 살펴보자. `fork()` 후에 `if`문을 통과한 후에 `else`문에서 부모 프로세스는 자신의 일을 수행한다. 모든 일을 마친 후 소스코드의 마지막으로 가보니 **`wait()`** 시스템 콜을 호출하고 있다.

```
$ cat wait.c
#include <unistd.h>
#include <stdio.h>
main()
{
    int pid;
    printf("I am Parent\n");
    pid = fork();
    if (pid == 0) /* This is child process */
    {
        printf("I am Child \n");
        execlp("/bin/date", "/bin/date", (char *) 0);
    }
    else /* This is parent process */
    {
        wait(); /* parent sleeps → child run to completion → parent resume */
        /* If wait(2) is omitted, parent runs concurrently with child */
    }
}
```

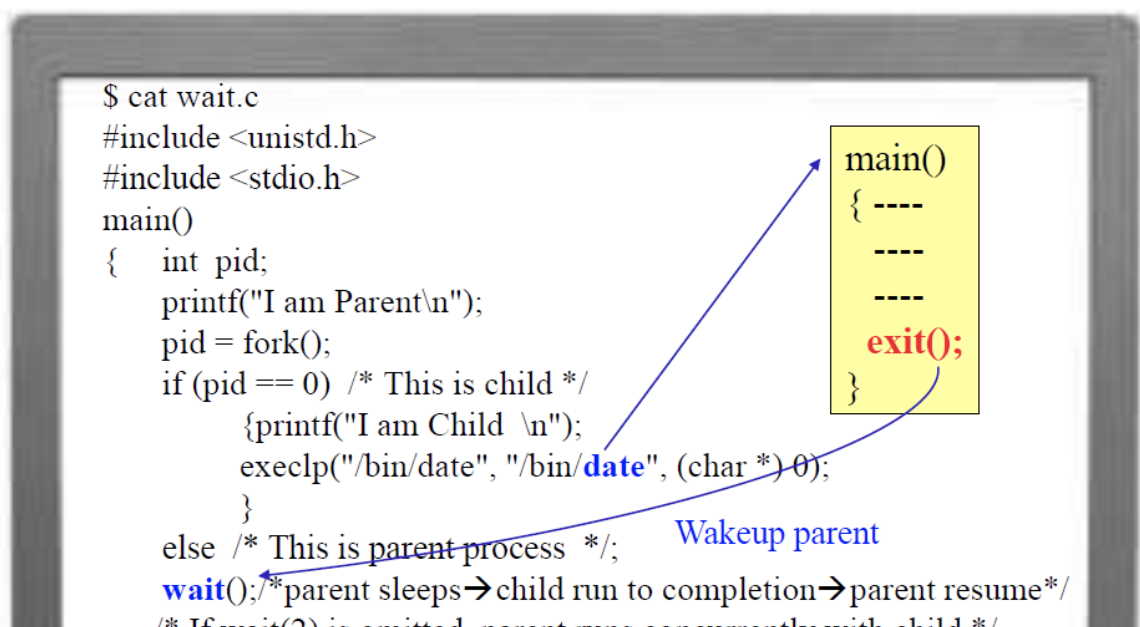

wait() 시스템 콜을 호출하면, 부모 프로세스는 잠들게 된다. 자식 프로세스가 끝날 때까지 잠을 잔다(sleep). CPU는 자식 프로세스에게 넘어가고 자식 프로세스는 자신이 할 일을 수행한다. 자식이 하는 일 중에 `execlp("/bin/date"...)` 라는 명령어가 마지막으로 있으니 해당 명령어를 마지막으로 수행하고 자식 프로세스는 종료한다.

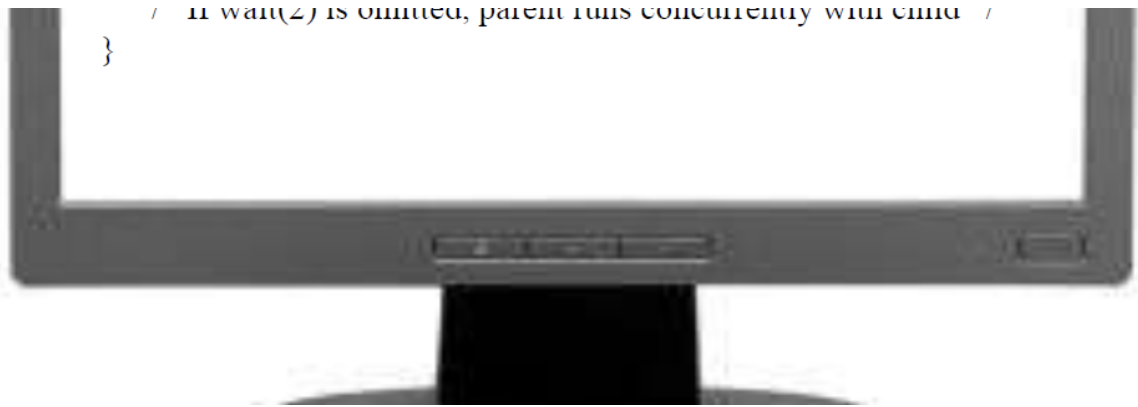
자식 프로세스가 종료했을 때 **CPU**는 자식 프로세스로부터 부모 프로세스를 찾는다. 그 후 CPU는 부모 프로세스를 대기명단(**ready queue**)에 등록시킨다. 이후 부모가 **CPU** 점유권을 받았을 때! 그 때가 바로 **wait()** 시스템 콜이 끝나는 지점이다. 부모는 이후 자신의 남은 일이 있었다면 해당 작업을 진행하게 된다.

비유를 들자면, 메일 프로그램을 들 수 있다. 메일 프로그램을 이용하는 목적은 상대방에게 메일을 보내는 것이므로 우리는 ‘메일 쓰기’를 클릭할 것이고, 곧 텍스트를 입력할 수 있는 에디터가 나타난다. 여기서 메일은 부모 프로세스고 텍스트 에디터는 자식 프로세스라고 할 수 있는데, 우리가 메일 쓰기를 마치면 자식 프로세스(텍스트 에디터)가 종료하면서 부모 프로세스(메일 프로그램)가 다시 등장하게 된다.

1.4 Exit(2) 동작 원리

메인함수 `main()` 가 끝날 때는 반드시 **exit(2)** 시스템 콜이 존재한다. 설령 우리가 소스 프로그램을 작성할 때, `exit()` 을 직접 기입하지 않았더라도 컴파일러가 알아서 **main()** { }의 마지막에 **exit(2)** 시스템 콜을 삽입하게 되어 있다. 아래 그림을 살펴보자.





자식 프로세스(`pid: 0`)의 작업 중 `execlp("/bin/date", ...)` 가 있고 위에서 배웠듯이 `exec(2)` 계열의 시스템 콜(`exec`, `execv`, `execlp` ...)이 실행되면서 현재 있는 프로세스 위에 인자로 주어진 프로세스(**date**)를 덮어 씌어버린다. 그리고 곧장 해당 프로세스의 `main()` 을 실행시키게 된다. 원래 저 노란 박스(**main** 함수가 들어 있는)에는 `exit()` 이라는 소스코드가 존재하지 않았다. 하지만 컴파일러가 컴파일을 할 때 삽입을 해줬고, 실제 만들어진 이진파일(binary file)을 열어 보면, `exit(2)` 에 해당하는 코드가 들어있게 된다.

exit(2) system call

- signals -- ignore them all
- files -- close
- image -- deallocate memory space
- parent -- notify (send signal)
- state -- set it ZOMBIE

- Then kernel (while executing `exit()` system call)
 - takes away the CPU
 - gives CPU to other process
 - `exit()` calls kernel function `schedule()` to do this

위 그림에는 `exit(2)` 의 작동 원리가 좀 더 상세하게 적혀 있다. 이후 들어오는 신호들을 전부 무시해버리고, 파일들이 열려 있다면 파일들을 닫는다. 또한 메모리 영역에서 해당 프로세스가 차지하고 있는 부분(image)을 해제(deallocate) 해버리고, 부모 프로세스에게 통보한다. 그리고 `exit(2)` 을 호출한 프로세스의 상태를 좀비(ZOMBIE)상태로 설정한다. (좀비 상태라는 건 다음 강의에서 다루게 된다.)

커널에서 일어나는 동작으로는 먼저 **exit(2)**을 호출한 프로세스의 **CPU**를 빼앗고, **ready queue**에 있던 다른 프로세스에게 **CPU**를 넘겨준다. 이 과정을 스케줄링(**scheduling**)한다고 표현하는데, 실제로 **exit(2)**을 호출하게 되면 커널 안의 **schedule()** 함수가 호출된다. 스케줄 함수 관련 설명은 글의 마지막 3번 부분에서 다룬다.

2. 시스템 콜 요약 정리 (Summary)

Summary: system calls for process

- `fork` create a child (copy)
- `exec` overlay new image,
 analogous to goto
- `wait` sleep until child is done
- `exit` frees all the resources, notify parent

132

NEAOSS MC2.0 2-2-5-I v1.0

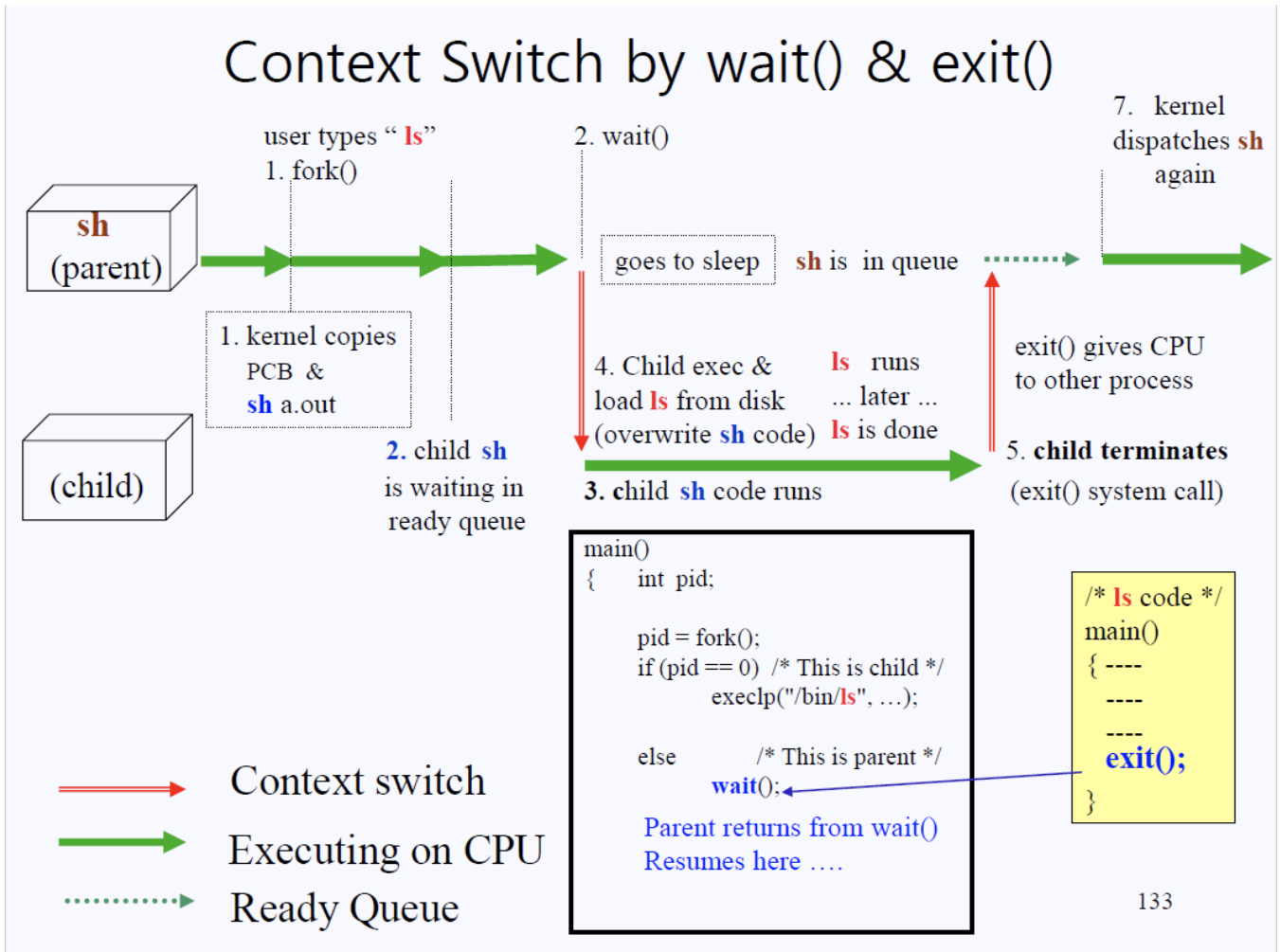
CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

지금까지 우리는 프로세스를 위한 4가지 시스템 콜에 대해 살펴 보았다. `fork()` 는 부모 프로세스와 아주 유사한 자식 프로세스를 만들어 내고, `exec()` 은 진행 중인 프로세스 위에 새로운 프로세스 이미지를 덮어 씌운 후 `main()` 으로 가게 된다. `wait()`

은 이 시스템 콜을 호출한 프로세스를 잠들게 하는 것이고, `exit()` 은 가지고 있던 모든 자원(resource)을 반환하고 부모 프로세스에게 알려주는 역할을 한다.

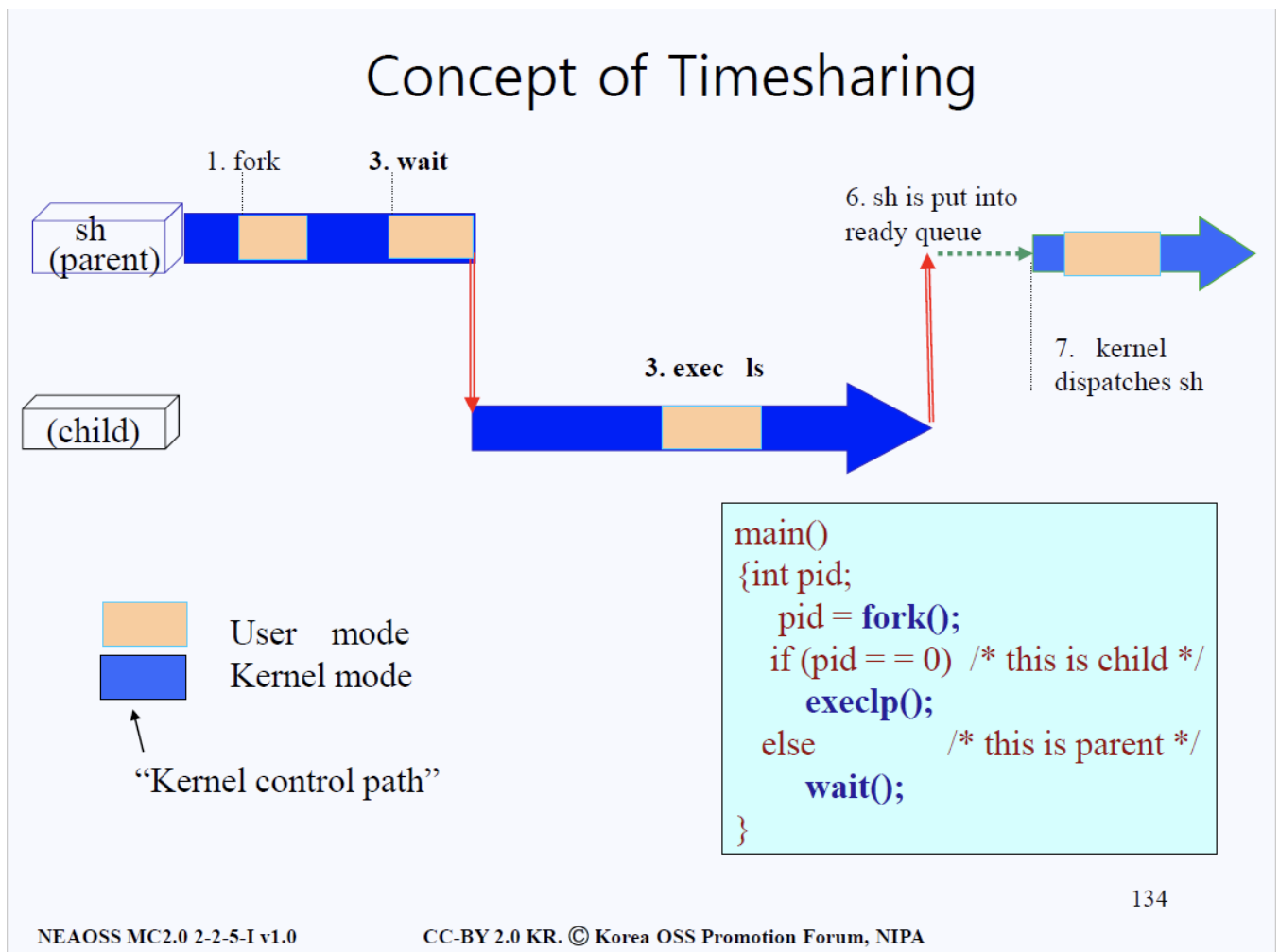
3. Context Switch (유저 모드와 커널 모드 사이의 전환)

지금부터 설명하는 내용은, 설명과 함께 그림을 봐야 이해가 잘되니 이 점 꼭 유의해서 설명을 차근차근 살펴보자.



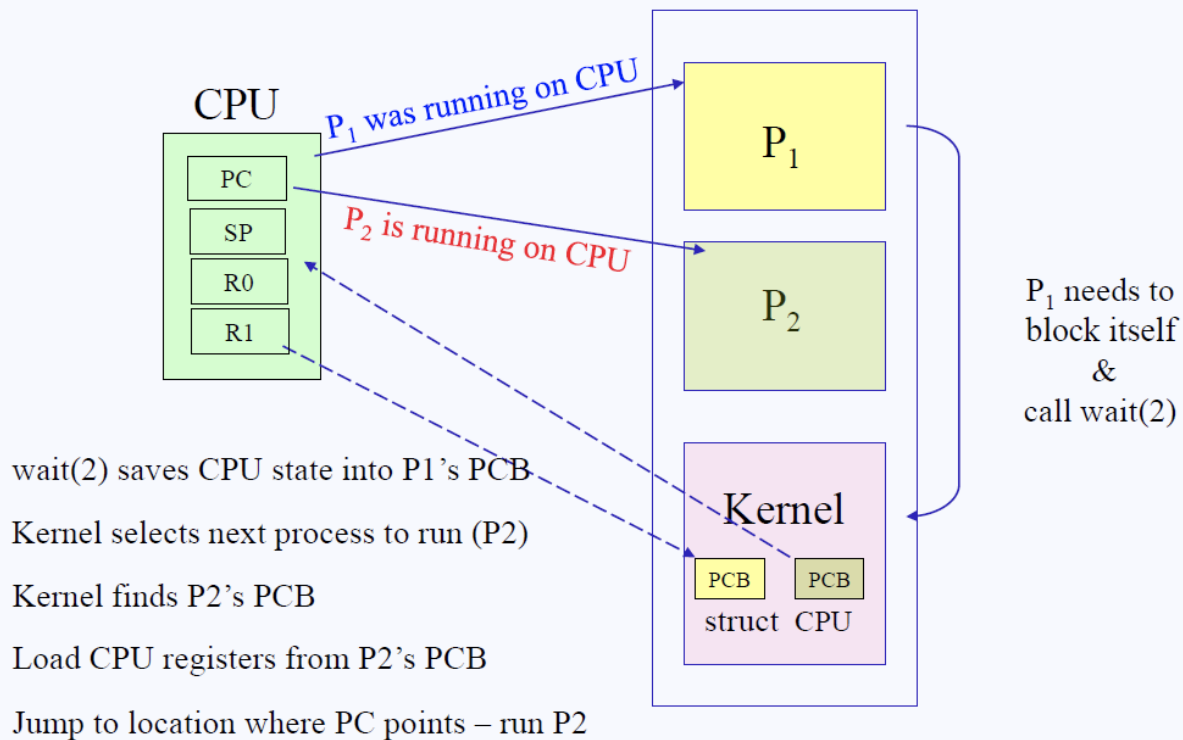
1. 사용자가 셸이 띄어 준 프롬프트에 명령어("`ls`")를 입력한다. 셸 입장에서 이 프로세스를 실행시키기 위해서 `fork()`를 실행한다. 여기서 셸은 부모 프로세스가 되고 새롭게 생기는 프로세스는 자식 프로세스가 된다. `fork()`가 동작하면서 셸의 **PCB**와 셸의 **a.out(코드)**을 그대로 복사한다. 그러나 **CPU**는 아직 셸에게 할당되어 있기 때문에 `ls`가 실행되거나 하진 않는다.
2. 부모 프로세스 셸이 `wait()`을 호출하게 되고 셸은 잠들게 된다. 잠들면서 부모 프로세스는 **CPU**의 대기 리스트(**queue**)에 들어가게 된다.

3. 자식 프로세스는 부모 프로세스와 똑같은 코드 및 상태를 가지고 있으므로 **fork()** 중간에서 동작하게 된다. **fork()** 로부터 리턴된 값은 자식 프로세스를 뜻하는 **pid** 값 0으로 자식프로세스는 **execlp("/bin/ls" ...)**를 실행하게 된다.
4. 디스크로부터 **ls** 를 로드한다. 자식 프로세스가 기존의 부모 프로세스(셸)로부터 그대로 복사해왔던 이미지 위에 그대로 덮어씌운다(overwrite). 덮어 씌운 후 **ls** 의 메인 코드로 가서 코드를 실행한다(**ls** 가 실행된다).
5. **ls** 가 끝나면 **exit(2)** 시스템 콜을 하게 되어 있고, **exit(2)** 을 호출함으로써 다시 커널모드로 들어와서 커널은 **CPU**를 다른 프로세스에게 할당하게 된다. 이 때 **wait(2)** 시스템 콜이 끝난 것으로 인지를 하게 된다.
6. (그림에는 7번으로 되어 있음) 높은 우선순위를 가지고 기다리고 있던 프로세스가 없다면, 기존의 부모 프로세스(셸)는 다시 동작하게 된다.



위 그림은 셸의 유저모드와 커널모드를 왔다 갔다 하는 것을 시간 순서로 도식화 해놓은 것이다. 위에 해당하는 부분은 가볍게 훑어보는 것으로 아래의 그림으로 넘어가 보자.

Context Switch -- CPU & PCB



135

NEAOSS MC2.0 2-2-5-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

Kernel의 경우 하드웨어를 위한 자료구조, 즉 테이블이 하나 존재한다. 그 자료구조를 위 그림에서는 **struct CPU**라고 표현하고 있다. 위 그림의 상황을 보자면, 먼저 CPU가 P₁을 실행시키고 있다(파란 글씨로 **P₁ was running on CPU**). 그리고 P₁이 **wait(2)** 시스템 콜을 호출한다. 시스템 콜을 호출하면서 커널은 **CPU state vector(PC, SP 등)**를 P₁의 **PCB**에 저장한다.

이렇게 상태 값을 기억하는 이유는 **wait(2)** 시스템 콜이 끝났을 때 **wait(2)**을 호출한 프로세스가 다시 정상적으로 작업을 원활하게 진행하기 위해서다. 보다시피 P₁과 P₂의 **PCB**는 커널 코드 안에 있다. 위 그림의 Kernel 파트를 보면, 커널에는 2종류의 자료구조가 존재하고 있다. P₁과 P₂에 해당하는 **PCB**들을 각각 하나씩 가지고 있는데, 커널 안에는 기본적으로 각 하드웨어 자원들마다(**for each hardware resource**) 자료구조가 존재하고 또한 각 유저 프로세스마다(**for each user processs**) 자료구조가 존재한다.

P₁은 자신의 **state vector**에 해당하는 값들을 P₁에 대응되는 **PCB**에 써주고(저장하고), CPU는 이제 그 다음 실행해야 할 프로세스에게 자신을 넘겨줘야 한다. CPU는 ready queue를 따라가서 CPU를 쓰겠다고 줄을 서 있는 프로세스들의 **PCB**를 살펴보고 우선순위가 제일 높은 프로세스를 선택한다. 그 프로세스가 동작하기 위해

서는 그 프로세스에 해당하는 **PCB**로부터 레지스터 값들을 가져와서 자신이 가지고 있는 **PC, SP** 등에 저장해야 한다. **CPU**안에 있는 **PC(Program Counter)**가 **P2**의 **PC**로 바뀌었기 때문에 **P2**의 **PC**가 가리키고 있는 곳부터 실행(**run**) 된다.

Context Switch – *schedule()*

- Kernel internal function (not known outside a.out)
- Following system calls may invoke *schedule()*
 - *read()*, *wait()*, *exit()*
- Selects **new process to run** & calls *context_switch()*
- *context_switch()* calls
 - *switch_to()* - CPU switching
 - **save current CPU state** → PCB (retiring process)
 - mark this process SLEEP (in itsPCB)
 - **load CPU registers** ← PCB (arising process)
 - : *switch_mm()*: virtual memory mapping
 -
- Goto arising process (i.e. fetch next instruction - PC)

136

NEAOSS MC2.0 2-2-5-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

다음으로 **Context Switch**에서 중요한 역할을 맡고 있는 **schedule** 함수에 대해 살펴보자. *schedule()* 은 **internal** 함수이고, Kernel a.out 에 알려지지 않는 함수다. **internal** 함수란 정 반대되는 성격을 가진 것이 바로 시스템 콜이며, 시스템 콜은 커널 **a.out**에 알려지고 커널 밖에서 부를 수 있다. (커널이 금단의 영역이라면, 시스템 콜은 그 영역에 접근할 수 있는 유일한 방법이다.) 반면에 **schedule()**은 커널 안에서만 부를 수 있는 함수이다. 즉 유저모드(커널 밖)에서는 요청조차 할 수 없다.

우선 이 *schedule()* 은 다음에 실행될 프로세스를 찾아 선택한다. 그리고 **context_switch()**라는 함수를 호출한다. *schedule()* 은 *read()*, *wait()*, *exit()* 과 같은 함수가 호출한다. *read()* 의 경우를 생각해 보면, 사실 디스크로부터 데이터를 읽어와 달라는 요청은 CPU 입장에서 몇 억년 걸리는 일이다. 디스크에 간다고 해서 바로 정보를 읽어올 수 있는 확률은 매우 적기 때문에(다른 프로세스에

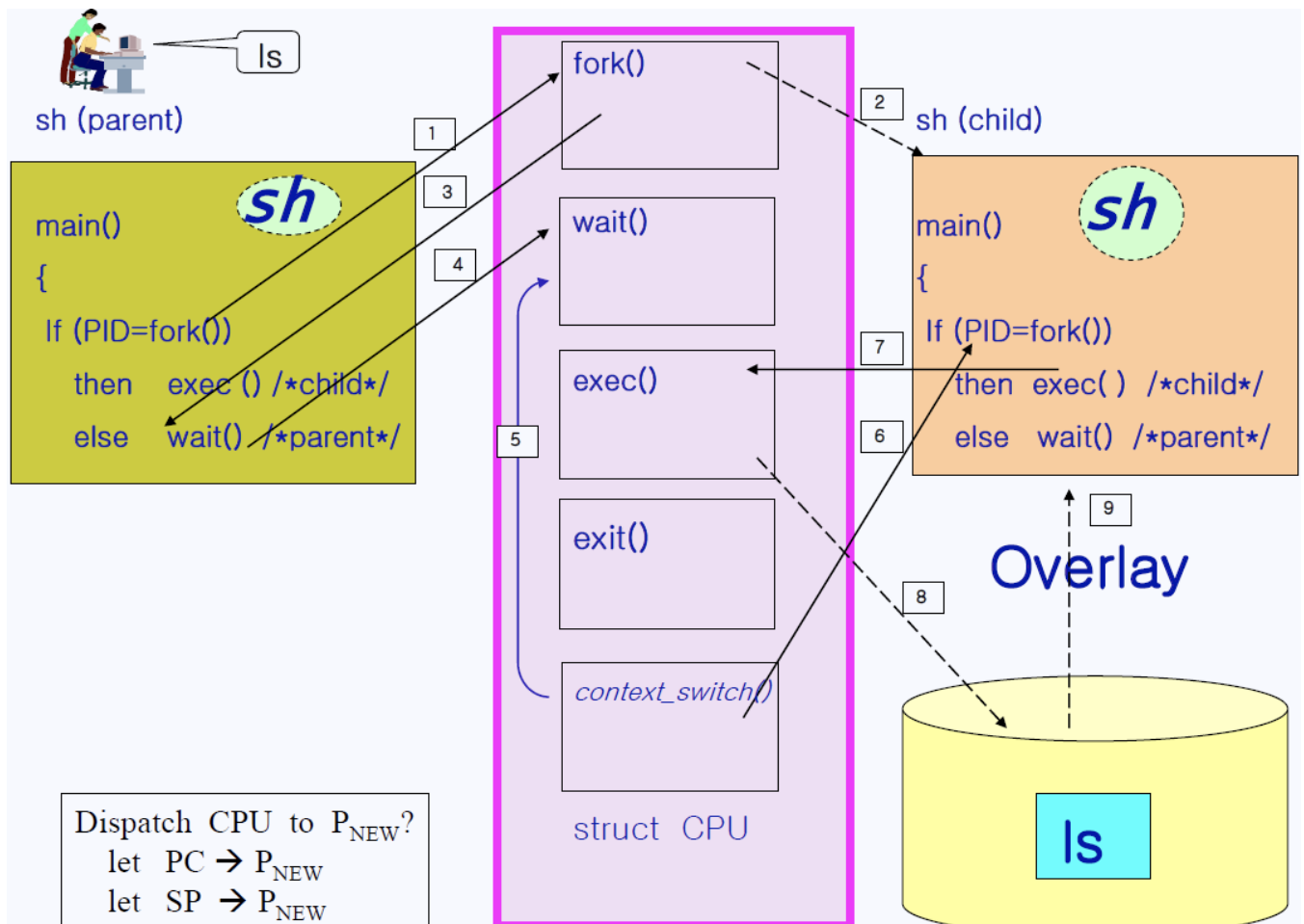
서도 디스크를 사용중일 수 있기 때문에) 필연적으로 대기하는 시간이 생기게 되는 데 이 시간 동안 CPU가 가만히 있을리 없다. 모든 자원은 제때 제때 효율적으로 사용이 되어야 하기 때문에 CPU를 다른 프로세스에 할당 해주어야만 한다. 그래서 `read()` 에서도 `switch()` 호출이 일어나는 것이다.

context_switch()를 부르면, 현재 **CPU state vector**를 은퇴하는 프로세스의 **PCB**에 쓰고, 새로 등장(**arising**)하는 프로세스로부터 **PCB**를 로드하고, 해당 **PCB**의 **PC**로부터 다시 프로그램을 진행하는 작업을 해준다.

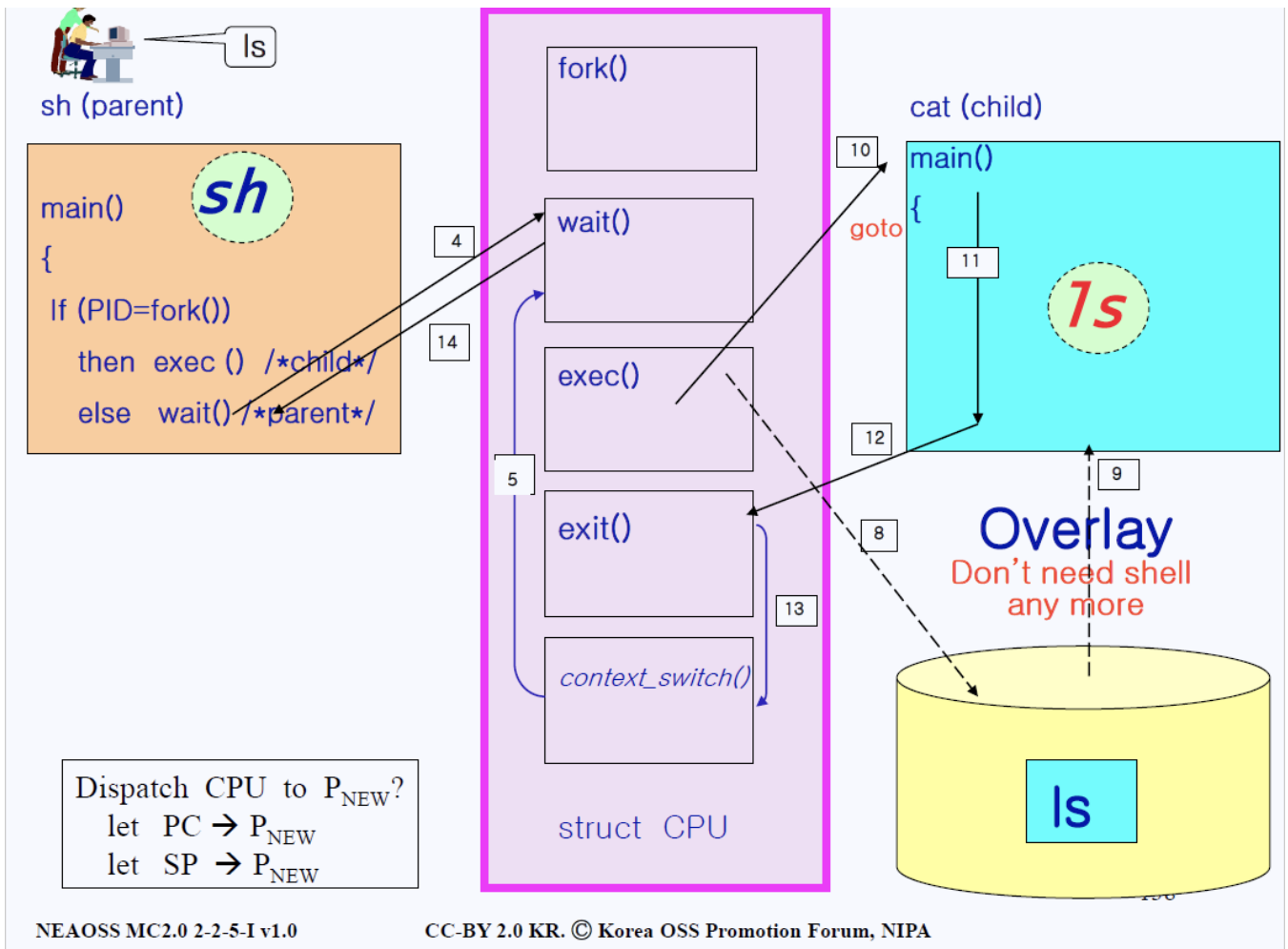
즉 `schedule()` 은 **CPU**의 임자가 바뀌어야 할 때(**read()**, **wait()**, **exit()**)마다 불리고, 새로운 임자에게 할당해주기 위한 내부 작업을 진행한다.

4. 총정리

지금까지 다뤘던 내용들을 총 엮어서 설명을 진행한다. 꽤나 복잡한 그림이 엮여 나오니 설명과 함께 따라오도록 노력해보자. 일단 아래 그림에 분홍색 구간은 커널이다. 커널 안에는 여러가지 시스템 콜이 존재하고 있다. 그리고 이 시스템 콜들은 `context_switch()` 와 같은 내부함수와 연관이 있으며 각 하드웨어 자원마다 자료구조가 존재(**struct CPU**)한다. 아래 그림 또한 그림에 오류가 있는 부분이 있는데, 오류가 난 부분은 설명하면서 함께 나오니 너무 걱정할 필요는 없다.



1. `fork()` 를 진행하면 커널로 진입한다. 커널에서 **`fork()`**는 부모 프로세스와 똑같은 **image**를 생성한다.
2. 점선으로 표시된 이유는 아직 **CPU** 제어가 부모 프로세스에 있기 때문에 자식 프로세스로 향하는 선은 점선으로 표시가 되어 있다.
3. 그 다음 `fork()` 작업이 끝나고 리턴한다. 앞서 언급했듯 부모 프로세스에서 `fork()` 를 실행했을 때의 결과값과 자식 프로세스가 실행했을 때의 결과값은 다르다고 했다. 일단 첫번째로 리턴되는 건 부모 프로세스의 **PID**가 리턴되므로 **`else`**문으로 가서 **`wait()`** 시스템 콜을 호출한다.
4. `wait()` 시스템 콜의 요청을 처리하기 위해 또 다시 커널모드로 진입한다. **`wait()`**은 **CPU**를 잠시 포기하겠다는 의미이기 때문에 **`context_switch()`** 함수를 실행한다.
5. 그림을 정정해야 한다. `wait()` 에서 `context_switch()` 로 가는 것이기 때문에 **5번** 화살표의 방향은 반대가 되어야한다. `context_switch()` 함수가 실행되면서, 먼저 **CPU**에 있던 **state vector** 영역에 해당하는 정보를 부모 프로세스의 **PCB**에 덮어 쓴다(저장한다). 이렇게 저장을 해야 후에 자식 프로세스의 작업이 끝나고 돌아왔을 때, 부모 프로세스의 **PCB**에 저장되어 있는 상태값들을 보고 후에 다시 부모 프로세스로 돌아가서 남은 작업들을 원활하게 처리할 수 있다.
6. 그런데 자식 프로세스가 생겨날 때 애초에 부모프로세스에서 `fork()` 가 일어난 시점에 형성된 것이므로, 자식 프로세스의 **PC(Program Counter)** 는 `fork()` 중간을 가리키고 있었을 것이다. 따라서 제어흐름은 **6번** 화살표를 따라 `fork()` 로 가게 되고, 자식 프로세스의 시작은 **`fork()`**에서 시작되는 것이다.
7. 자식 프로세스에서 실행되고 있는 `fork()` 의 리턴 값은 당연히 자식 프로세스의 **PID**일 것이다. 따라서 자식 프로세스가 실행하기로 되어 있는 `exec()` 이 호출된다.
8. **`exec()`**이 해주는 작업은 하드 디스크에 저장되어 있는 프로그램 코드(유저가 **`exec`** 시스템 콜의 매개변수로 준 프로그램)를 불러들여 현재 진행되고 있었던 프로세스 이미지 위에 덮어 씌우는 작업이다.
9. 따라서 디스크에 유저가 `exec()` 시스템 콜에 매개변수로 넘긴 `ls`에 해당하는 프로그램이 현재 진행중이었던 셸(자식 프로세스) 위에 덮어 씌어지게 된다.



NEAOSS MC2.0 2-2-5-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

10. 덮어씌어진 후에 **ls** 프로그램의 **main()**으로 흐름이 넘어간다.
11. **ls**의 코드가 전부 실행된 후 **exit()**이 호출되면서 흐름은 **12**번으로 넘어간다.
12. 소스코드 상에 **exit()**이 존재하지 않아도 컴파일러가 알아서 삽입을 해주기에, **exit()**을 무사히 실행할 수 있다. **exit()**은 지금까지 진행중이었던 프로세스로부터 **CPU**를 뺏고 다른 프로세스에게 재할당해 주는 과정이 있기에 마찬가지로 **context_switch()**를 호출하게 된다.
13. 자신을 호출한 프로세스로부터 **CPU**를 뺏고, **ready queue**에 가서 **CPU**를 기다리고 있던 프로세스 중 우선순위가 높은 프로세스를 골라서 해당 프로세스의 **PCB** 안의 상태값들을 현재 **CPU**의 레지스터에 복사 붙여넣기(복붙) 한다.
14. **ready queue**에 부모 프로세스만 남아있다고 가정한다면, 부모 프로세스가 선택되어 실행될 것이고 부모 프로세스는 **wait()**을 진행하고 있었기 때문에 **wait()** 중간부터 다시 실행된다.
15. 14번까지의 작업이 끝났다면 셸은 다시 사용자로부터 또다른 명령을 기다리고 있게된다.

4.1 용어 정리

총정리인 만큼, 기존의 프로그램과 프로세스 차이에 대해서도 한 번 짚어보고 가도록 한다.

Process(프로세스).

Concept of a "Process"

- Program in execution
- a.out (private address space)
- main()
- unit of scheduling
- protection domain (page table, files, ...)
- resource allocation
- can run at user mode/kernel mode (OS kernel, system call)

139

NEAOSS MC2.0 2-2-5-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

프로그램이 실행중일 때 우리는 프로그램을 프로세스라 부른다. a.out 형식을 가지고 main() 함수부터 시작하게 되어 있다. 스케줄링과 보호의 단위이고, 자원을 할당받는 과정을 수반하고 유저모드와 커널모드를 왔다갔다 하면서 진행된다.

Context

What constitutes a Process "Context"

[1] user space

– text / data / bss / heap / stack [argv, envp]

[2] kernel space

– *user / proc / stack*

[3] HW

– state vector (PC, SP, flags, reg0, ...)

Note:

data: initialized part. Space allocated in a.out `int A[]={1, 2};`
bss: uninitialized part. No space allocated in a.out `int B[100];`

140

NEAOSS MC2.0 2-2-5-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

유저 영역(user space)의 **text**는 **instruction(명령문)**을 의미한다. *data* 와 *bss* 에 대한 설명은 위 그림의 Note파트에 서술되어 있다. 먼저, 두 개의 배열(array)이 존재한다. A라는 배열은 초기값을 할당해줬고 B라는 배열에는 초기값을 주지 않았다. 만약 배열의 크기가 100만 정도에 전역변수로 선언되어 있다면? A 배열처럼 초기값을 할당 해줬다면 디스크에서 백만 개의 셀을 갖고 있어야 한다(사전에 자원이 지급됨). 만약 초기값을 주지 않았다면 디스크에 실제로 존재하진 않고 해당 배열이 실행 중 로드 될 때만 할당되게 된다.

초기에 값이 할당된 부분을 **data**라고 하며 초기에 할당되지 않은 데이터 부분을 **bss**라고 한다. **heap**은 동적 메모리 할당에 쓰여지는 데이터 영역이며 **stack**은 함수 호출 등에 사용되는 자료구조다.

커널 영역(kernel space)에는 **PCB**와 **stack**이 존재한다. HW(CPU) 쪽에서는 **state vector**가 존재한다. 이런 것들을 합쳐서 우리는 **context**라고 부른다.

Daemon (데몬) 또는 Server

서버(혹은 데몬)는 무엇일까? 근본적으로 서버는 **a.out(실행 파일)**이다. 다만 조금 특이한 알고리즘을 가지고 있을 뿐이다. 아래 그림을 살펴보자.

"Daemon" (or "Server") Process

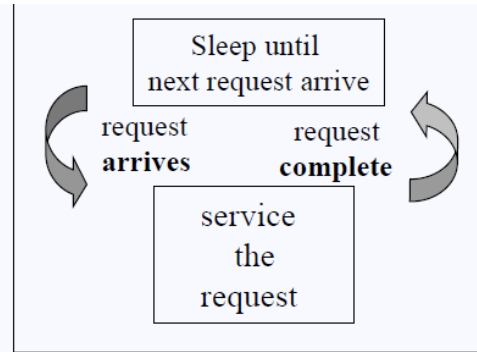
- Format: a.out
- common algorithm

Start a.out at
boot time



- **always waiting (loop)**
- **service incoming request**

- usually
 - user id = system
 - created during boot
 - runs forever
- mission
 - helps system (print, network, paging, ...)



141

NEAOSS MC2.0 2-2-5-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

맨 처음 서버 혹은 데몬이 시작되는 건 부팅 될 때(**boot time**)다. 부팅하고 나서 대부분의 시간은 잠들어 있다. 요청이 올 때만 해당 요청을 서비스 해주고 서비스가 끝나면 또 잠들게 된다. 이런 프로그램을 우리는 데몬 또는 서버라고 부른다. 만약 프린트 서버가 존재한다고 하면, 프린트 서버는 말 그대로 프린트 요청이 올 때만 프린트를 해주고 그 이외에는 잠든다. 네트워크 서버 또한 네트워크 요청(연결, 해제 등)이 올 때만 처리하고 그 이외에는 잠든다.

서버라는 것은 하드웨어의 개념이 아니라 소프트웨어의 개념인 것이다. 항상 **incoming request**가 오는지 안 오는지 지켜보고 있으며 서비스가 올 때만 서비스를 해주게 되어 있다.

• Exercise

- Try *ps -e*
- Different vendors, different daemons
- What does each daemon do?:

Nemeth, et. al., UNIX System Administration Handbook, 2nd. ed.,
Prentice Hall PTR.

• Daemon programming requires special skills

- must not use standard I/O → all I/O should be logged
- its working directory?
- its parent? Who forks daemon? Environment?

For more:

Chapter 13, Stevens, Advanced Programming in UNIX, Addison Wesley.

142

NEAOSS MC2.0 2-2-5-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

리눅스 시스템에서 사용되는 명령어 **ps(Process State)**를 살펴보자. 현재 기기에 어떤 프로세스가 작동하고 있는지를 나타낸다. `-e` 옵션의 경우 시스템 프로세스까지 전부 보여주는 명령어다. 웹서버나 네트워크서버 등의 모든 시스템 프로세스의 상태를 보여주는 명령어다.

- Typical daemons

- *httpd* web server
- *ftpd* ftp server
- *lpd* lineprinter spooler daemon
- *pagedaemon* paging

- Why not put this into kernel?

- Size
- flexible

- Micro-kernel

143

NEAOSS MC2.0 2-2-5-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

보통 데몬이나 서버 프로그램의 경우 이름 뒤에 `d` 자가 붙는다. `httpd`는 웹에서의 통신에 사용되는 데몬이고, `ftpd`는 파일전송 서버를 나타내는 등 다양한 서버와 데몬이 존재하고 있다.

5. 마치며

생각만큼 크게 어렵지 않았던 3강이다. 결국 모든 프로그램은 알고리즘을 이해하는 것이 전부가 아닐까 하는 생각이 든다. 애초에 프로그램이란 건 논리의 집합이고, 해당 논리대로 작업을 하는 것이니 그 논리만 파악하고 있으면 그 프로그램을 아는 것이니까. 그나저나 강의 노트를 작성하는 건 생각만큼 쉬운 일이 아니라는 걸 다시한번 체감한다. 내가 듣고 이해하는 것과 다시 누군가에게 풀어서 설명하는 건 천지차이니까.

Linux Linux Kernel Operating Systems