

CDA0017: Operating Systems

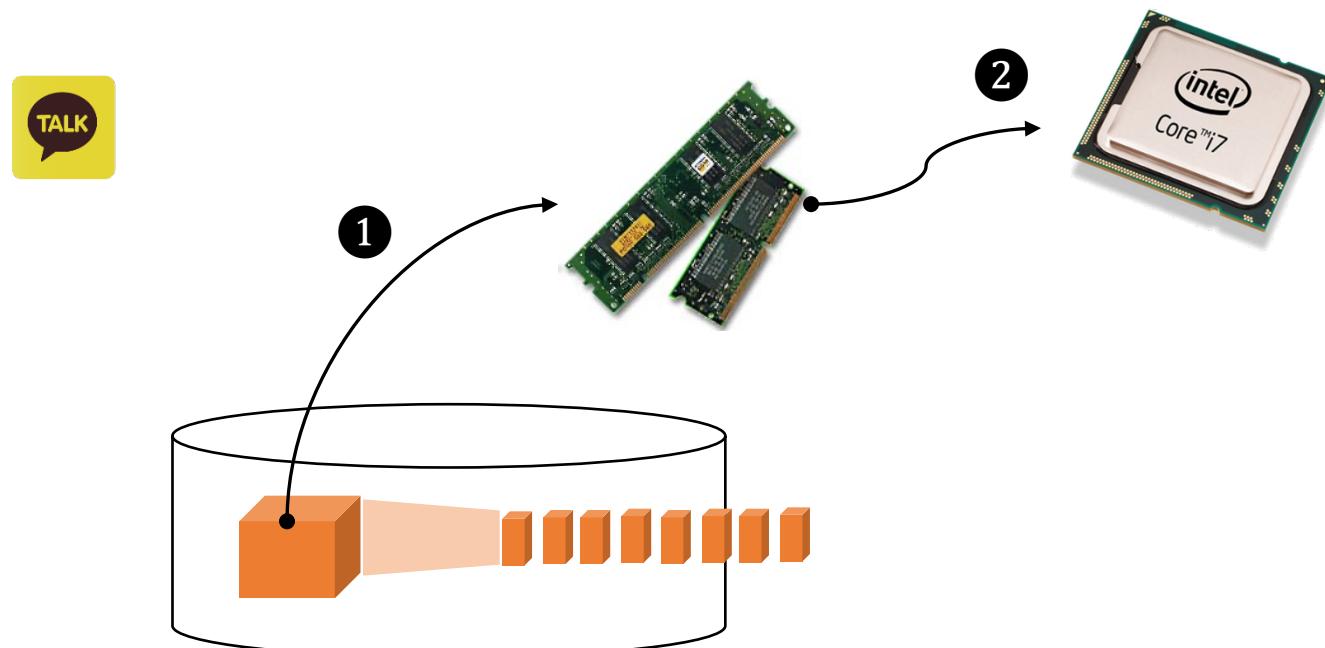
Donghyun Kang (donghyun@changwon.ac.kr)

NOSLab (<https://noslab.github.io>)

Changwon National University

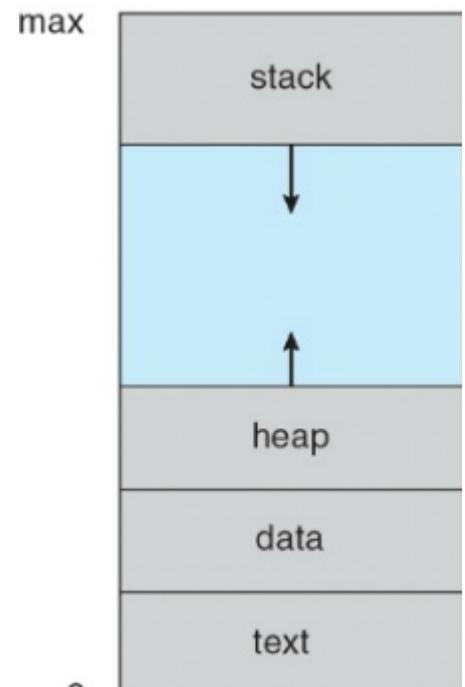
Process

- 프로세스는 **실행중인 프로그램**
 - 스토리지에 저장된 프로그램에게 생명을 넣는 것
- 예: 프로그램 실행



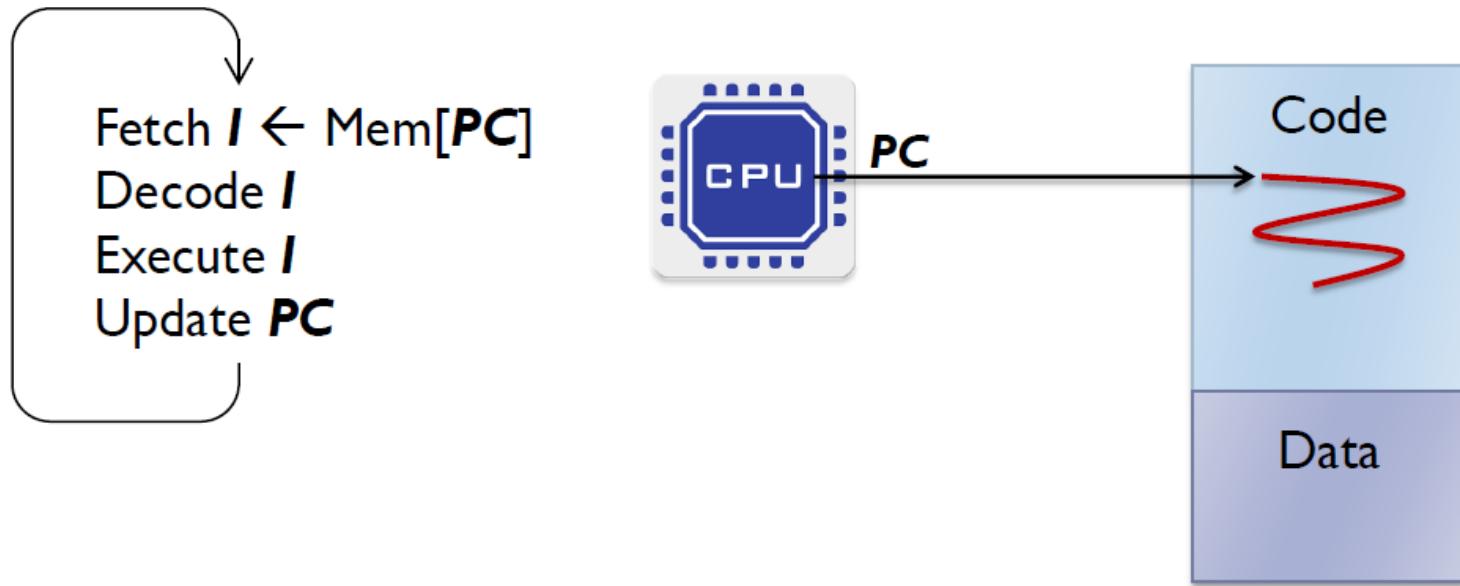
Process structure

- 메모리
 - Text: 프로그램 코드 저장
 - Stack: 함수의 매개변수, 복귀 주소와 로컬 변수 저장
 - Data: 전역변수, static 변수
 - Heap: 실행 중 동적인 메모리 할당
- 레지스터
 - 프로그램 카운터 (program counter)
 - 스택 포인터 (stack pointer)
 - 프레임 포인터 (frame pointer)

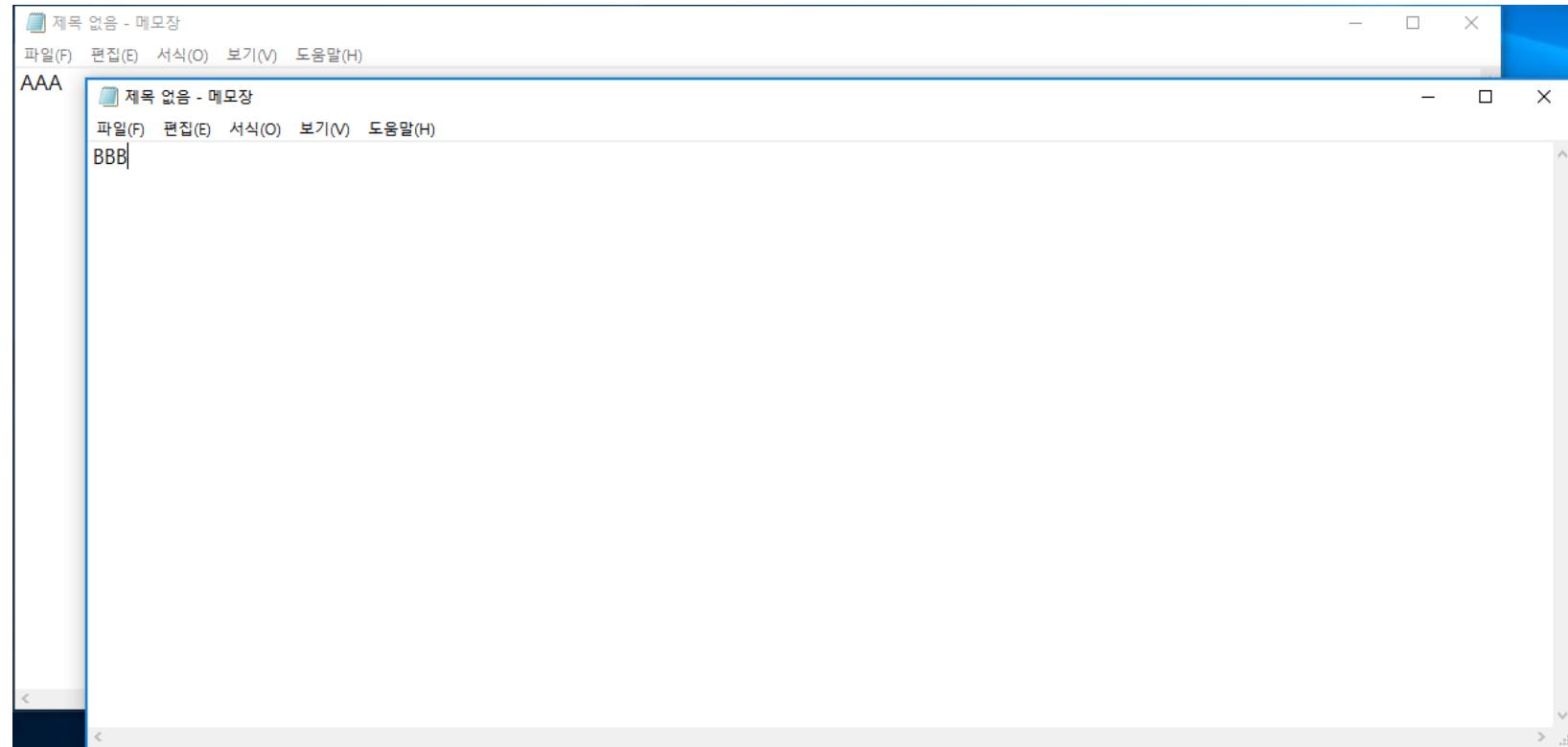


[메모리 상의 프로세스]

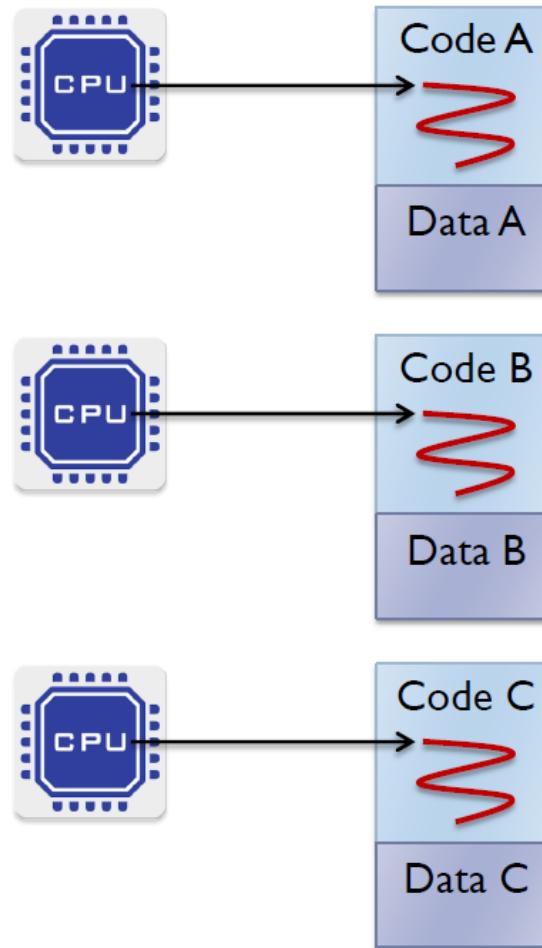
Running a process



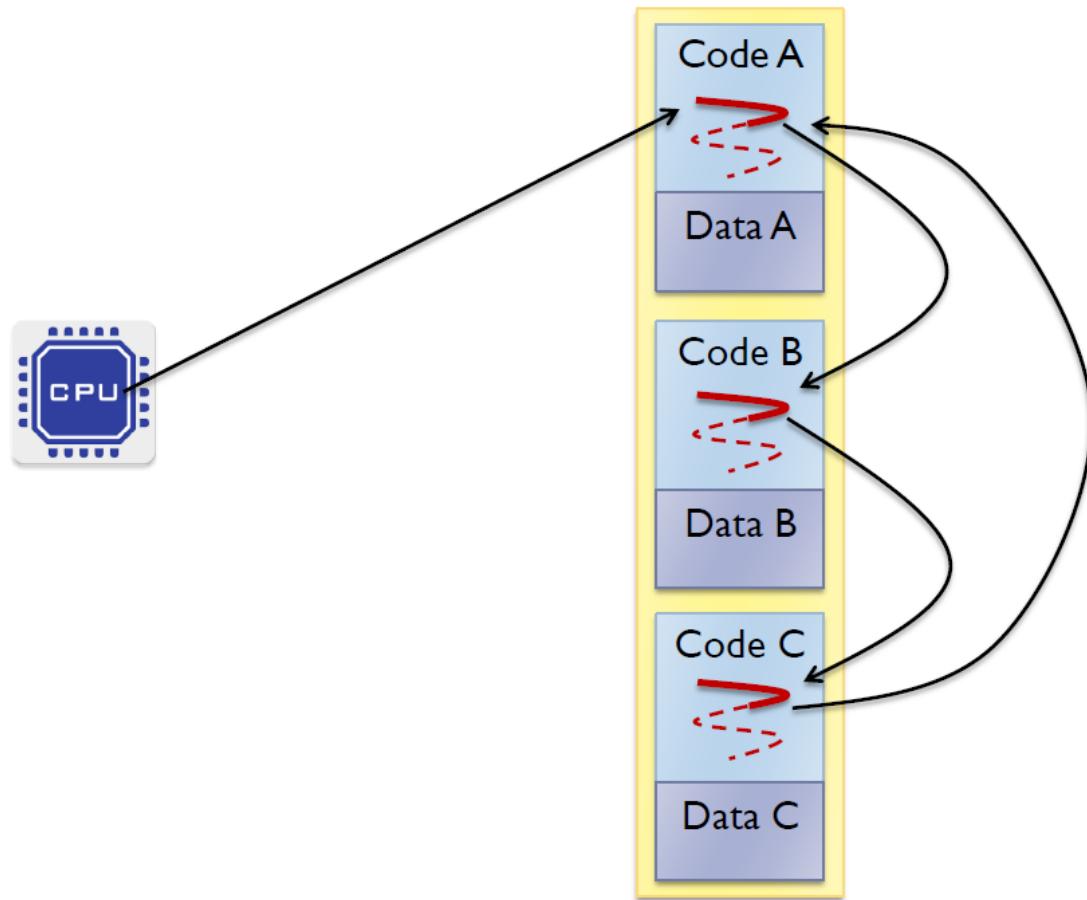
If the same program runs twice?



Running Multiple Processes

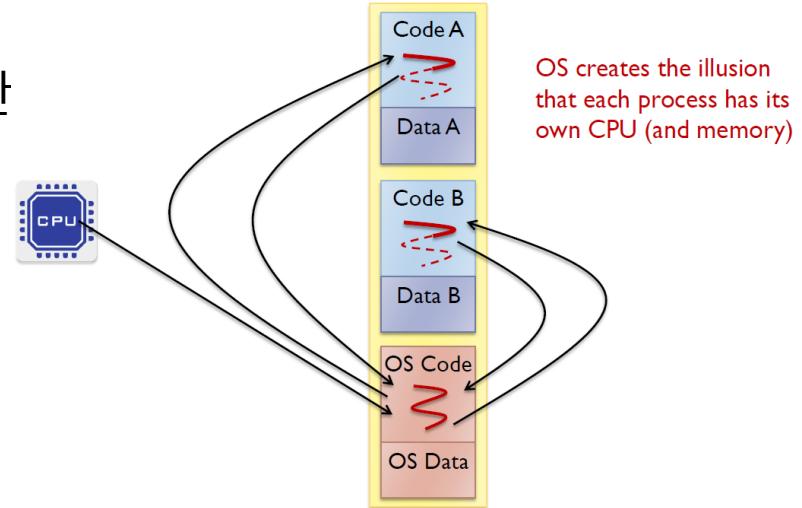


Interleaving Multiple Processes



Virtualizing the CPU

- 시분할(time sharing) 기법
 - 하나의 프로세스를 실행시키고, 얼마 후 중단시키고 다른 프로세스를 실행하는 작업의 반복
 - CPU를 공유하기 때문에, 각 프로세스의 성능이 저하됨
- 문맥 교환 (context switching)
 - CPU에서 프로그램 실행을 잠시 중단





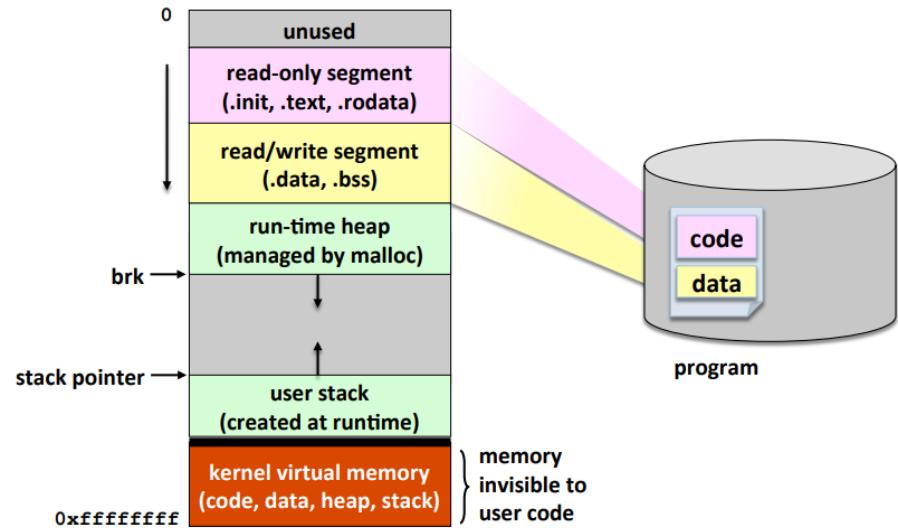
최신 폴더블 폰을 100대를 관리하려면?

API for Process

- **생성** (Create): 새로운 프로세스 생성을 위한 API
- **제거** (Destroy): 실행 중인 프로세스의 제거를 위한 API
 - 프로세스 스스로 종료
 - API를 통해 프로세스 종료
- **대기** (Wait): 어떤 프로세스의 실행 중지를 기다릴 필요가 있는 경우를 위한 API
- **각종 제어** (Miscellaneous Control): 프로세스를 일시정지하거나 재개하기 위한 API
- **상태** (Status): 프로세스 상태 정보를 얻기 위한 API

어떻게 프로그램을 준비하고 실행시키는가??

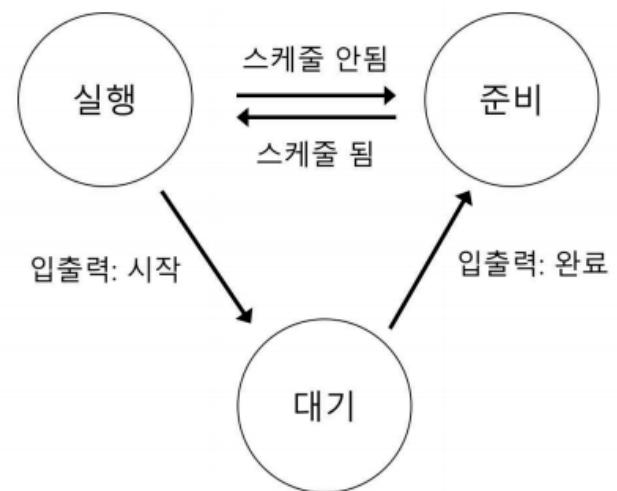
- 프로그램 코드와 정적 데이터를 메모리, **프로세스의 주소 공간에 Load**
 - On-demand memory load:** 프로그램을 실행하면서 코드나 데이터가 필요할 때 필요한 부분만 메모리에 탑재
- 스택 초기화
 - argc, argv 배열
- 힙 영역 생성 및 초기화
- 입출력 파일 디스크립터 셋업
 - STDIN, STDOUT, and STDERR
- main() 루틴으로 분기



[프로세스 Load]

Process State

- 실행 (Running)
 - 프로세서에서 실행 중인 프로세스의 상태
 - 명령어 fetch-decode-execute 가 진행되는 프로세스
- 준비 (Ready)
 - 실행할 준비가 되어 있지만 운영체제가 다른 프로세스를 실행하고 있는 등의 이유로 대기 중인 상태
- 대기 (Blocked)
 - 프로세스가 다른 사건을 기다리는 동안 프로세스의 수행을 중단시키는 연산



〈그림 7.2〉 프로세스: 상태 전이

Example

- 예제 (CPU만 이용할 때)

시간	Process0	Process1	비고
1	실행	준비	
2	실행	준비	
3	실행	준비	
4	실행	준비	Process0 종료
5		실행	
6		실행	
7		실행	
8		실행	Process1 종료

〈그림 7.3〉 프로세스 상태 추이 : CPU만 이용할 때

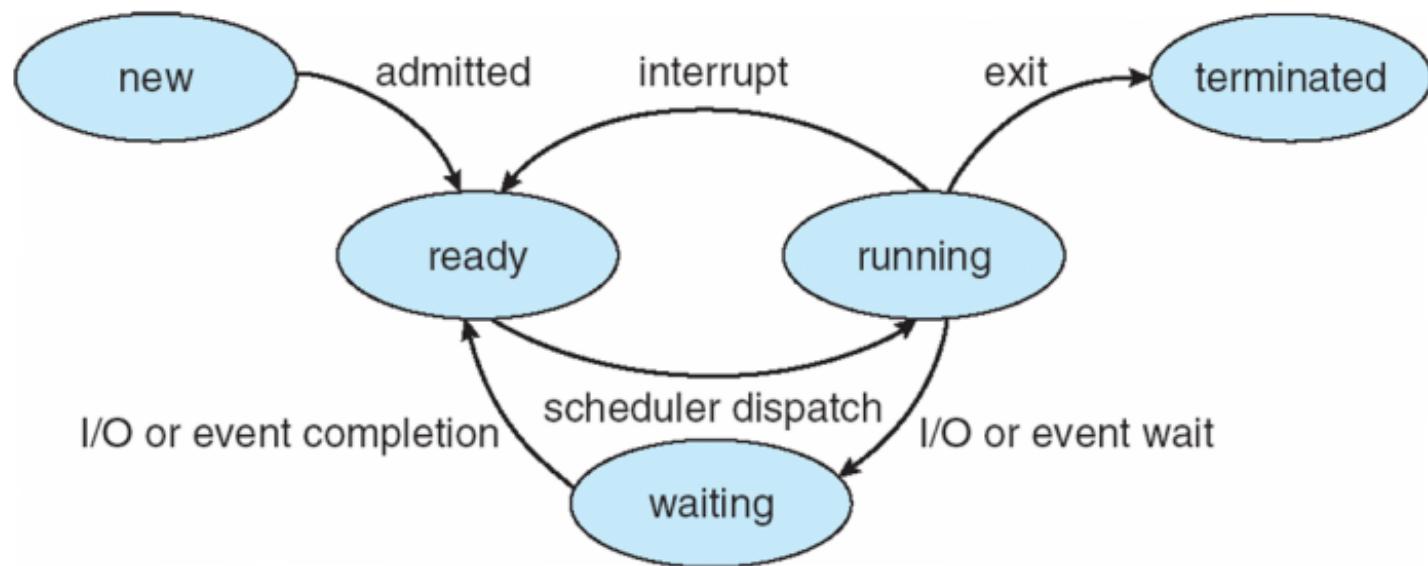
Example

- 예제 (CPU & 입출력을 이용할 때)

시간	Process0	Process1	비고
1	실행	준비	
2	실행	준비	
3	실행	준비	Process0이 입출력을 시작한다.
4	대기	실행	Process0은 대기 상태가 되고,
5	대기	실행	Process1이 실행된다.
6	대기	실행	
7	준비	실행	Process0 입출력 종료
8	준비	실행	Process1 종료
9	실행	-	
10	실행	-	Process0 종료

〈그림 7.4〉 프로세스 상태 추이: CPU 이용과 입출력 작업을 할 때

Extended Process State



Example in Ubuntu

- **R:** Running
- **S:** Sleeping
- **T:** Traced or Stopped
- **D:** Uninterruptable Sleep
- **Z:** Zombie
- **<:** High-priority task
- **N:** Low-priority task
- **S:** Session leader
- **+::** In the foreground process group
- **I:** Multi-threaded

```
7574 0.2 0.0 506696 85784 pts/23 Sl+ 8월30 35:43 /opt/google/chrome/chrome --type=utility --field-trial-h  
7663 0.0 0.0 714012 79324 pts/23 Sl+ 8월30 0:07 /opt/google/chrome/chrome --type=renderer --field-trial-h  
7805 0.0 0.0 714012 78568 pts/23 Sl+ 8월30 0:07 /opt/google/chrome/chrome --type=renderer --field-trial-h  
7815 0.0 0.1 828932 168852 pts/23 Sl+ 8월30 9:16 /opt/google/chrome/chrome --type=renderer --field-trial-h  
7901 0.0 0.0 714616 81264 pts/23 Sl+ 8월30 0:06 /opt/google/chrome/chrome --type=renderer --field-trial-h  
8096 0.3 0.4 1287712 567804 pts/23 Sl+ 8월30 49:56 /opt/google/chrome/chrome --type=renderer --field-trial-h  
8254 0.2 0.1 781296 148148 pts/23 Sl+ 8월30 37:21 /opt/google/chrome/chrome --type=renderer --field-trial-h  
8268 0.0 0.0 723320 86548 pts/23 Sl+ 8월30 0:05 /opt/google/chrome/chrome --type=renderer --field-trial-h  
8282 0.2 0.1 780784 142696 pts/23 Sl+ 8월30 37:24 /opt/google/chrome/chrome --type=renderer --field-trial-h  
8420 0.0 0.1 799376 158960 pts/23 Sl+ 8월30 7:37 /opt/google/chrome/chrome --type=renderer --field-trial-h  
8531 0.0 0.1 776308 137448 pts/23 Sl+ 8월30 2:39 /opt/google/chrome/chrome --type=renderer --field-trial-h  
8576 0.0 0.1 775672 136780 pts/23 Sl+ 8월30 0:10 /opt/google/chrome/chrome --type=renderer --field-trial-h  
8592 0.0 0.0 629100 61740 pts/23 Sl+ 8월30 0:04 /opt/google/chrome/chrome --type=ppapi --field-trial-han  
12007 0.0 0.0 667920 49876 pts/23 Sl+ 06:13 0:00 /opt/google/chrome/chrome --type=renderer --field-trial-ha  
15337 0.0 0.0 28324 6004 pts/9 Ss 09:21 0:00 bash  
15366 0.0 0.0 23872 2952 pts/9 R+ 09:23 0:00 ps -ux  
22648 0.0 0.0 651176 114224 ? SNl 9월08 0:03 /usr/bin/python3 /usr/bin/update-manager --no-update --n  
24155 0.1 0.1 798120 153164 pts/23 Sl+ 8월31 13:33 /opt/google/chrome/chrome --type=renderer --field-trial-h  
@dhkangd-X10DRi:~$
```

[“ps – ux” 명령의 실행 결과]

The xv6 kernel

- 운영체제도 일종의 프로그램
 - 즉, 다른 프로그램들과 같이 다양한 정보를 유지하기 위한 자료 구조가 필요함
- 자료구조

```
// 프로세스를 중단하고 이후에 재개하기 위해
// xv6가 저장하고 복원하는 레지스터
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

// 가능한 프로세스 상태
enum proc_state { UNUSED, EMBRYO, SLEEPING,
    RUNNABLE, RUNNING, ZOMBIE }, };

// 레지스터 맵과 상태를 포함하여
// 각 프로세스에 대하여 xv6가 추적하는 정보
struct proc {
    char *mem;           // 프로세스 메모리 시작 주소
    uint sz;             // 프로세스 메모리의 크기
    char *kstack;        // 이 프로세스의 커널 스택의 바닥 주소
    enum proc_state state; // 프로세스 상태
    int pid;             // 프로세스 ID
    struct proc *parent; // 부모 프로세스
    void *chan;          // 0이 아니면, chan에서 수면
    int killed;          // 0이 아니면 종료됨
    struct file *ofile[NOFILE]; // 열린 파일
    struct inode *cwd;   // 현재 디렉터리
    struct context context; // 프로세스를 실행시키려면 여기로 교환
    struct trapframe *tf; // 현재 인터럽트에 해당하는 트랩 프레임
};
```

The ubuntu kernel

- Process Control Block (PCB)
 - Process state: 상태 정보 저장 (e.g., running, waiting, etc.)
 - Program counter: 다음 실행 명령어 위치 저장
 - CPU registers: CPU 연산 정보 저장
 - 누산기, 상태 코드, 인덱스, 스택, 범용 레지스터
 - CPU scheduling information: 현재 프로세스 우선순위 저장
 - Memory-management information: 메모리 정보 저장
 - Accounting information: CPU 사용 시간 등의 정보 저장
 - I/O status information: 프로세스에 할당된 입/출력 장치들과 열린 파일의 목록 저장

process state
process number
program counter
registers
memory limits
list of open files
...

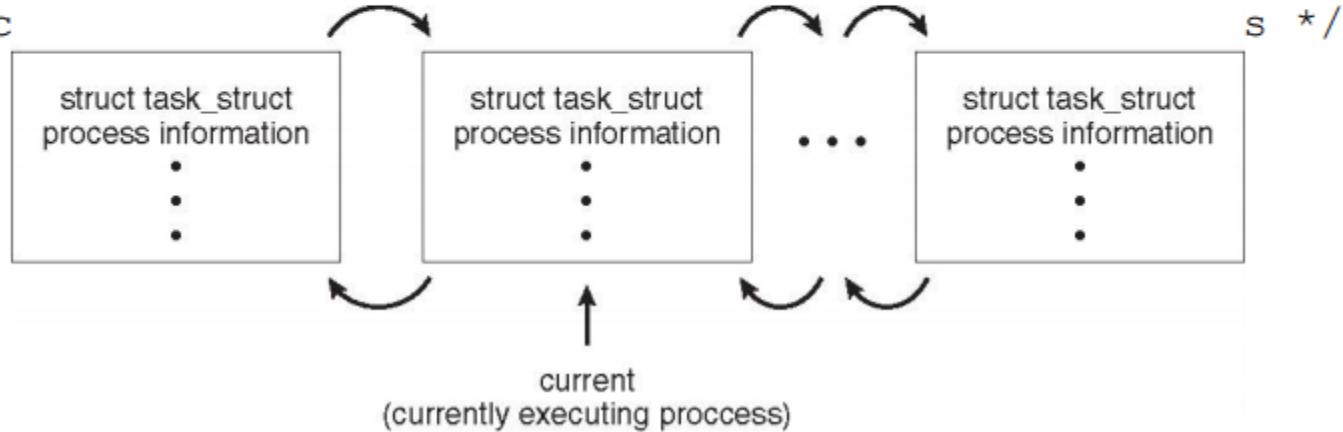
The ubuntu kernel

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

C implementation

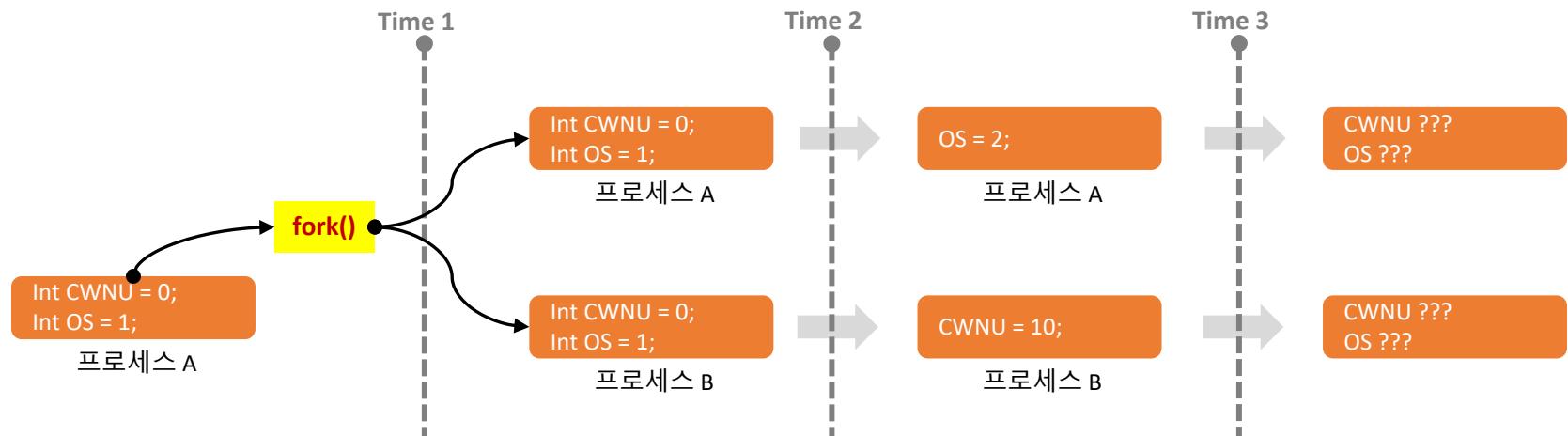
- **task_struct** 구조체

```
pid t_pid; /* process identifier */  
long state; /* state of the process */  
unsigned int time_slice /* scheduling information */  
struct task_struct *parent; /* this process's parent */  
struct list_head children; /* this process's children */  
struct files_struct *files; /* list of open files */  
struc
```



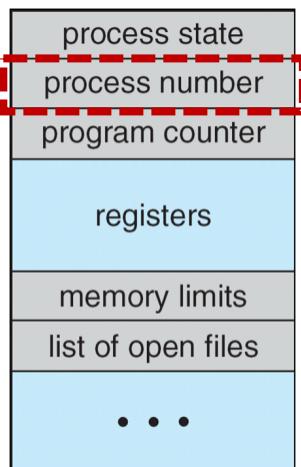
Create process

- **fork()** 자신을 복사하여 새로운 자식 프로세스를 생성하는 시스템 콜



Create process

- 그러나, 부모와 자식은 서로 다른 PCB을 가짐



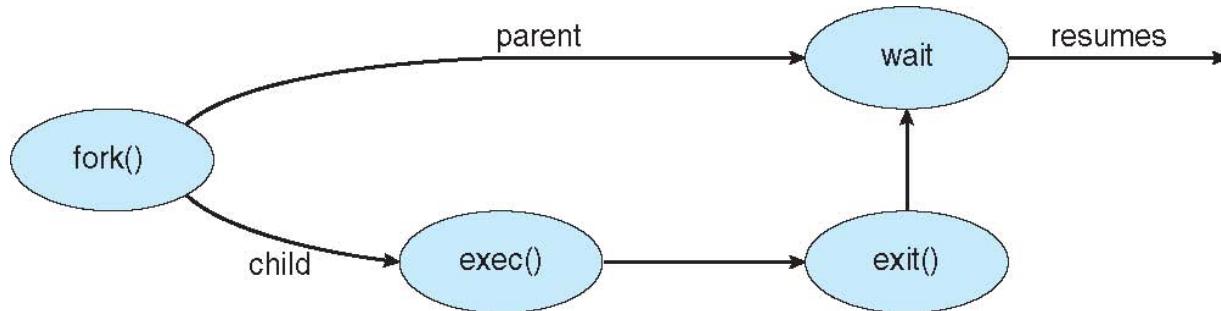
- Process state: 상태 정보 저장 (e.g., running, waiting, etc.)
- Program counter: 다음 실행 명령어 위치 저장
- CPU registers: CPU 연산 정보 저장
- 누산기, 상태 코드, 인덱스, 스택, 범용 레지스터
- CPU scheduling information: 현재 프로세스 우선순위 저장
- Memory-management information: 메모리 정보 저장
- Accounting information: CPU 사용 시간 등의 정보 저장
- I/O status information: 프로세스에 할당된 입/출력 장치들과 열린 파일의 목록 저장

C implementation for Crate process

- **do_fork()** <kernel/fork.c>
 - Allocate a new PID
 - **copy_process()**
 - **dup_task_struct()** to create a new kernel stack, **thread_info**, **task_struct**
 - Copy resources: **copy_files()**, **copy_fs()**, **copy_mm()**, ...
 - **copy_thread()** to initialize the kernel stack
 - **sched_fork()** to ready for the new task to schedule
 - **wake_up_new_task()**
 - Schedule the child task (run the child first)

Parent vs. Child

- 부모 프로세스: fork() 함수의 리턴으로 **0보다 큰 값을** 가짐
- 자식 프로세스: fork() 함수의 리턴으로 **0 값을** 가짐



Example

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int
6 main(int argc, char *argv[])
7 {
8     printf("hello world (pid:%d)\n", (int) getpid());
9     int rc = fork();
10    if (rc < 0) {           // fork 실패; 종료
11        fprintf(stderr, "fork failed\n");
12        exit(1);
13    } else if (rc == 0) {    // 자식(새 프로세스)
14        printf("hello, I am child (pid:%d)\n", (int) getpid());
15    } else {                // 부모 프로세스는 이 경로를 따라 실행한다(main)
16        printf("hello, I am parent of %d (pid:%d)\n",
17               rc, (int) getpid());
18    }
19    return 0;
20 }
```

〈그림 8.1〉 `fork()` 호출 (p1.c)

Example

- 실행 화면

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

Q1: 자식 프로세스는 어디서부터 시작하는가?



Q2: 항상 부모 프로세스가 먼저 시작하나?

Wait for child

- **wait()** 자식 프로세스가 종료되기를 기다리는 시스템 콜

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 int
6 main(int argc, char *argv[])
7 {
8     printf("hello world (pid:%d)\n", (int) getpid());
9     int rc = fork();
10    if (rc < 0) {           // fork 실패: 종료
11        fprintf(stderr, "fork failed\n");
12        exit(1);
13    } else if (rc == 0) {    // 자식(새 프로세스)
14        printf("hello, I am child (pid:%d)\n", (int) getpid());
15    } else {                // 부모 프로세스는 이 경로를 따라 실행한다 (main)
16        int wc = wait(NULL); // ← 여기서 문제 발생!
17        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
18               rc, wc, (int) getpid());
19    }
20    return 0;
21 }
```

<그림 8.2> fork() 와 wait() 호출 (p2.c)



Run process by fork()

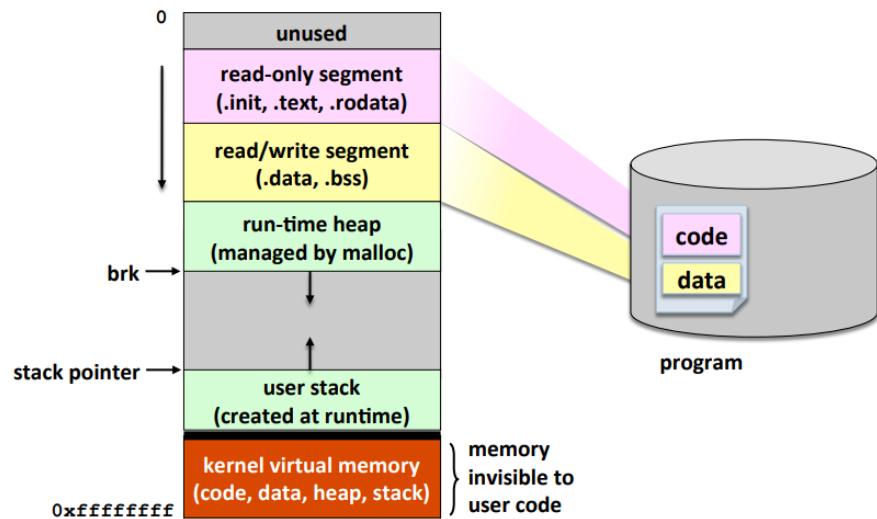
- **exec()** 다른 프로그램을 실행하기 위한 시스템 콜



현재 프로세스가 다른 프로세스를 어떻게 실행하는가??

Run process by fork()

- 현재 프로세스 재활용
 - 코드, 데이터 세그먼트 덮어쓰기
 - 힙, 스택, 기타 영역 초기화
- 실행된 프로세스는 return 불가

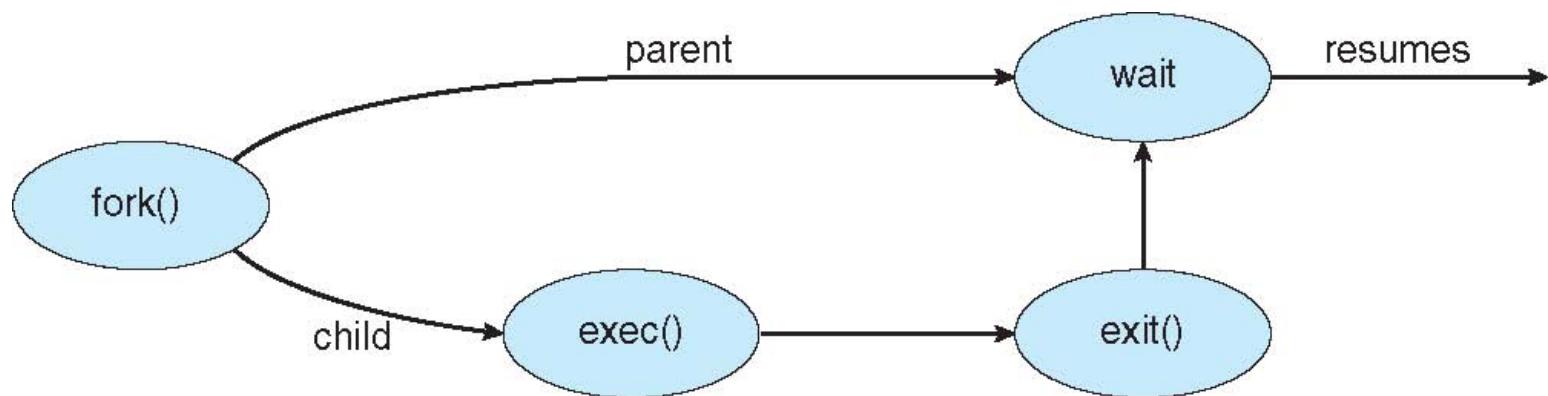


Example

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <sys/wait.h>
6 int
7 main(int argc, char *argv[])
8 {
9     printf("hello world (pid:%d)\n", (int) getpid());
10    int rc = fork();
11    if (rc < 0) { // fork 실패함: exit
12        fprintf(stderr, "fork failed\n");
13        exit(1);
14    } else if (rc == 0) { // 자식(새 프로세스)
15        printf("hello, I am child (pid:%d)\n", (int) getpid());
16        char *myargs[3];
17        myargs[0] = strdup("wc"); // 프로그램: "wc"(단어 세기)
18        myargs[1] = strdup("p3.c"); // 인자: 단어 셀 파일
19        myargs[2] = NULL; // 배열의 끝 표시
20        execvp(myargs[0], myargs); // "wc" 실행
21        printf("this shouldn't print out");
22    } else { // 부모 프로세스는 이 경로를 따라 실행한다 (main)
23        int wc = wait(NULL);
24        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
25               rc, wc, (int) getpid());
26    }
27    return 0;
28 }
```

〈그림 8.3〉 `fork()`, `wait()`, 및 `exec()` 호출하기 (p3.c)

Why does OS call fork and exec in a separate way?



Q&A