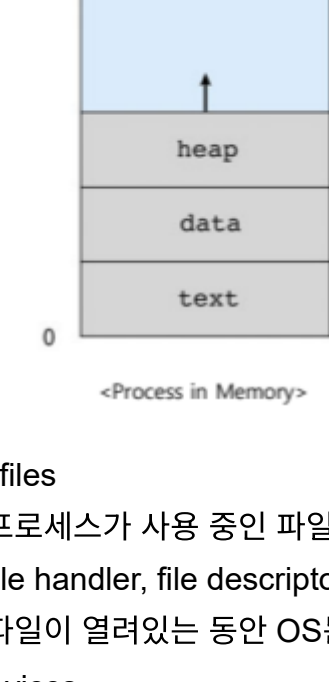


# Process 1

## Process란 무엇인가

- 실행 중인 프로그램

## process에 들어가는 정보

- cpu context
    - 현재 프로세스를 실행하는 데 필요한 정보가 담긴 레지스터들의 집합
    - CPU는 이 레지스터들을 사용하여 계산 수행
  - PC, stack pointer
    - 프로세스가 어디서부터 실행되고 있는지
    - 프로세스가 사용 중인 스택의 현재 위치
  - OS resources
    - 운영체제는 각 프로세스가 사용하는 자원들을 관리
    - 프로세스는 자원들을 요청하고 운영체제가 이를 할당
  - Address Space
    - 프로세스가 사용하는 메모리 공간
- 

The diagram shows a vertical stack of memory segments. From top to bottom: 'stack' (grey), 'heap' (grey), 'data' (grey), and 'text' (grey). A blue rectangular area is positioned between the 'stack' and 'heap' segments. An arrow points downwards from the 'stack' segment into the blue area, and another arrow points upwards from the 'heap' segment into the same blue area. The top of the 'stack' segment is labeled 'max' and the bottom of the 'text' segment is labeled '0'. Below the diagram is the label '<Process in Memory>'.
- Open files
    - 프로세스가 사용 중인 파일들에 대한 정보
    - file handler, file descriptor가 포함
    - 파일이 열려있는 동안 OS는 파일의 상태를 추적, 관리
  - I/O Devices
    - 프로세스가 사용 중인 입출력 장치에 대한 정보 관리
  - Others
    - PID
    - State의 4가지 상태
      - Running : 실행 중
      - Ready : 실행을 기다림
      - Blocked/Waiting : 자원을 기다림
      - Terminated : 종료됨
    - Owner
      - 프로세스의 소유자
      - 해당 프로세스를 실행한 사용자 정보
      - 보안과 접근 권한 관리를 위함
    - Priority
      - OS는 프로세스의 Priority를 기준으로 CPU를 할당
    - Signals
      - 프로세스는 외부로부터 시그널 수신 가능
      - 운영체제나 다른 프로세스가 시그널을 보내서 이 프로세스를 종료하거나 특정 작업 수행 가능

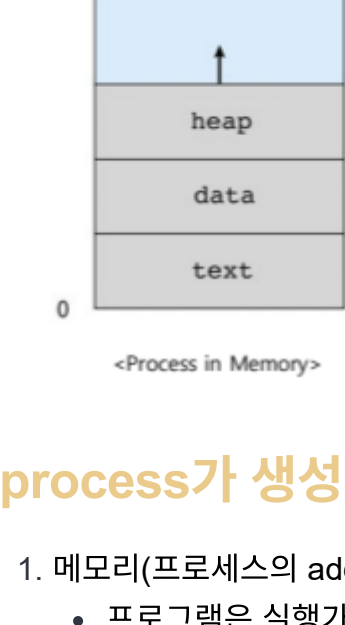
## 프로세스 특징

- protection의 기본 단위
  - protection : 시스템 자원에 대한 접근 권한 제어, 다른 프로세스의 데이터 침범 방지
- Work의 기본 단위
  - 여러 사용자나 프로세스가 동시에 CPU를 공유
  - 멀티태스킹 기반
  - 각 프로세스가 짧은 시간 간격 동안 실행되고 빠르게 전환
  - 시스템에서 자원을 할당받아 실행되는 가장 작은 작업 단위
- 하나의 프로그램이 여러 process가 될 수 있음

## 프로세스 종류

- I/O-bound process
  - computation 보단 I/O 하는 데 더 많은 시간을 소비
  - 매우 짧은 CPU 시간
- CPU-bound process
  - computation에 더 많은 시간을 소비
  - 매우 긴 CPU 시간

## Process 생성하기



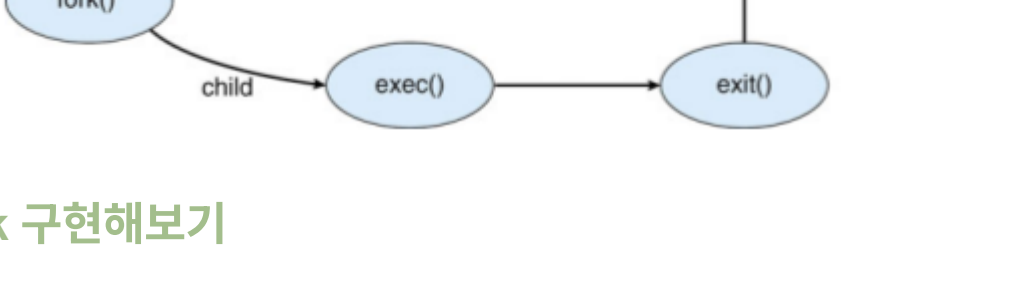
## process가 생성되는 과정

- 메모리(프로세스의 address space)에 프로그램의 코드를 로딩
  - 프로그램은 실행가능한 형태로 디스크에 존재
  - OS는 loading process를 lazily하게 수행
    - lazily : 처음부터 전부 로딩하지 않고 필요할 때마다 필요한 걸 로딩
- 프로그램의 run-time stack 할당
  - local variable, function parameter, return address를 관리할 스택을 사용
  - 인자와 함께 스택 초기화
    - main()에 들어가는 argc, argv[]
- 프로그램의 heap 생성
  - 동적 할당 데이터를 사용할 때 사용
  - malloc()로 call, free()로 해제
- OS가 그 외 초기화 작업 수행
  - I/O setup
    - 각 프로세스는 기본적으로 세 개의 open file descriptor를 가짐
  - standard input / output, error
- main(entry point)부터 프로그램 시작
  - OS가 새로 생긴 프로세스한테 CPU의 제어권 전달

## process 관련 명령어

### fork와 exec

- fork()
  - 시스템 콜
  - 부모 프로세스를 복제해서 새 프로세스를 생성
    - 자식은 부모의 address space를 복사
    - 부모는 대부분의 resource와 privilege를 inherit
      - open files, UID, etc.
  - 부모는 wait()로 자식이 끝나길 기다리거나 병렬로 작업 수행
  - 셸과 GUI가 fork()를 내부적으로 실행
- exec()
  - 새 프로그램을 실행
  - 현재 프로세스 이미지를 새 프로그램으로 대체
  - 기존 프로세스의 address space와 code를 삭제, 새 프로그램 load
  - PID 유지, 기존 프로세스를 종료하고 새 프로세스를 생성하는 것이 아님
  - 윈도우의 createProcess()가 리눅스의 fork(), exec()를 합친 것과 같음
  - 새 프로세스를 생성하는 것이 아니므로 메모리 면에서 효율적



### fork 구현해보기

```
int fork()
```

- PCB 생성 및 초기화
  - process control block ?
    - 각 프로세스의 정보를 관리하는 자료구조
    - PCB가 곧 프로세스를 의미 (queue에 넣어짐)
    - 운영체제가 프로세스를 추적하고 제어하기 위해 사용하는 데이터 저장소
  - 다음과 같은 정보들이 저장됨

항목	설명
PID (Process ID)	프로세스를 식별하는 고유 ID
Process State	현재 프로세스 상태 (Ready, Running, Blocked 등)
Program Counter (PC)	다음에 실행할 명령어의 메모리 주소
CPU Registers	프로세스가 사용하던 CPU 레지스터 값 (문맥 저장)
Memory Management Info	프로세스의 주소 공간, 페이지 테이블 정보 등
Open Files	프로세스가 열어놓은 파일 목록
I/O Information	현재 프로세스가 사용 중인 입출력 장치
Parent & Child Info	부모 프로세스, 자식 프로세스 ID
Priority	스케줄링을 위한 우선순위

- 새 address space 생성 및 초기화 or 부모의 address space를 복사해서 초기화
- 부모가 사용하던 커널 리소스를 참조하기 위해 커널 리소스 초기화
  - 커널 리소스 : file descriptor, socket, pipe, signal handlers, PCB, etc
- PCB를 ready queue에 배치
  - ready queue : CPU를 사용하기 위해 대기 중인 프로세스 목록 (큐)
- 부모에게 자식 PID를 반환, 자식에선 0을 반환
  - 부모는 자식 PID를 알게되고 자식은 0값으로 자신이 자식임을 알 수 있음

### exec 구현해보기

```
int execev(char *prog, char *argv[])
```

- 현재 프로세스 중단
- 프로그램 prog를 현재 프로세스의 address space에 적재
- hardware context와 새 프로그램에 쓰일 args(main)를 초기화
  - hardware context : CPU 레지스터, PC, SP, etc.
- PCB를 ready queue에 배치

- exec()는 새 프로세스를 생성하지 않음
- exec()가 반환하는 값? -1이면 오류?

## process 명령어 사용 예제

```
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int pid;
    if ((pid = fork()) == 0)
        /* child */
        printf ("Child of %d is %d\n",
                getppid(), getpid());
    else
        /* parent */
        printf ("I am %d. My child is %d\n",
                getpid(), pid);
}
```

```
prompt> ./a.out
I am 31098. My child is 31099
Child of 31098 is 31099

prompt> ./a.out
Child of 31100 is 31101
I am 31100. My child is 31101
```

```
int main()
{
    pid_t    pid;
    /* fork another process */

    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork error ...\n");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /*parent process will wait for the child process to complete */
        wait(NULL);
        printf("Child Completed ...\n");
        exit(0);
    }
}

? end main ?
```

## Process Hierarchy



- 부모 프로세스가 자식 프로세스를 생성하면서 트리가 형성
- 보통 pid로 구분

### 3가지 모드

- resource sharing mode
  - 부모와 자식이 모든 자원을 공유
  - 자식은 부모의 일부만 공유 받음
  - 혹은 서로 공유를 아예 안 할 수도
- execution mode
  - 부모와 자식은 동시에 실행
  - 부모는 자식이 terminate될 때 까지 대기
- address space mode
  - 자식은 부모의 복제품
  - 자식은 새 프로그램을 가짐

### 부모-자식 관계

- 하나의 프로세스가 다른 프로세스를 만들
- 유닉스는 이 hierarchy를 process group이라고 부름
- 윈도우는 부모-자식 관계 개념이 없음

### 프로세스 검색

- 유닉스에선 ps로 수행
- 윈도우에선 task manager로 가능

## Process 끊기

### 4가지 termination

- Normal exit (voluntary)
- Error exit (voluntary)
  - 비정상적인 값이 들어오면 exit(1) 하도록 직접 구현
- Fatal error (involuntary)
  - ex) segmentation fault, protection fault, exceed allocated resources, etc.
- killed by another process (involuntary)
  - 시그널을 받으면 수행

### process termination 추가 설명

- 프로세스는 exit()를 통해 마지막 명령문을 실행하고 OS에게 삭제해달라고 요청
  - wait()를 통해 자식 프로세스의 output data가 부모 프로세스로 전달
  - OS가 프로세스 자원을 deallocate
- 부모는 abort()로 자식 프로세스 실행을 terminate
  - 자식에 자원 할당이 초과됐을 때
  - 자식이 더 이상 필요하지 않을 때
  - 만약 부모가 exit 해서 terminate 됐으면 OS는 자식도 없앴 (cascaded termination)
- 부모는 wait()로 자식 프로세스가 termination 되길 기다림.
  - wait()는 중단된 프로세스의 pid와 status의 정보를 반환
  - 만약 부모가 어떤 프로세스를 wait()하지 않고 계속 남아있으면 그 프로세스는 zombie
  - 만약 부모가 wait하지 않고 terminated 되면 남은 프로세스는 orphan

### process code

- cmdline을 입력받아 실행
- getcmd()로 사용자 입력 받기
- parsecmd()로 명령어와 인자를 argv 배열로 변환
- builtin\_command(argv) (내장명령어) 가 아니면 fork()로 자식 프로세스 생성
- 자식 프로세스에서 execv()를 호출하여 명령 실행
- 부모는 waitpid()로 자식 프로세스가 중단될 때까지 기다림

```
int main(void)
{
    char cmdline[MAXLINE];
    char *argv[MAXARGS];
    pid_t pid;
    int status;
    while (getcmd(cmdline, sizeof(buf)) >= 0) {
        parsecmd(cmdline, argv);
        if (!builtin_command(argv)) {
            if ((pid = fork()) == 0) {
                if (execv(argv[0], argv) < 0) {
                    printf("%s: command not found\n", argv[0]);
                    exit(0);
                }
            }
            waitpid(pid, &status, 0);
        }
    }
}
```

## Process State에 대하여

- 프로세스가 실행되면서 state도 계속 바뀜
  - new : 프로세스가 새로 생김
  - running : instruction이 실행되는 중임
  - waiting : 프로세스가 어떤 이벤트가 발생하길 기다리는 중임
  - ready : 프로세스가 프로세서에 할당되길 기다림
  - terminated : 프로세스가 끝남
- tracing process state : CPU only vs CPU and I/O

Time	Process <sub>0</sub>	Process <sub>1</sub>	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process <sub>0</sub> now done
5	-	Running	
6	-	Running	
7	-	Running	
8	-	Running	Process <sub>1</sub> now done

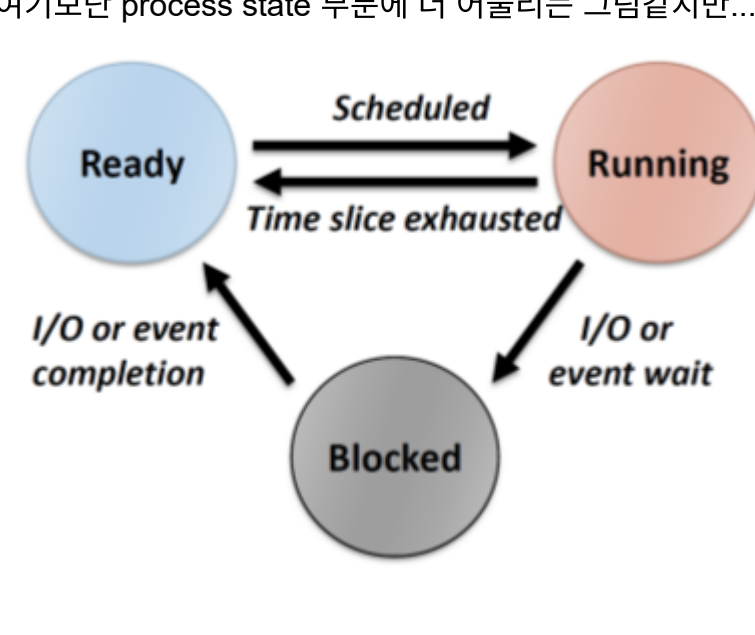
Time	Process <sub>0</sub>	Process <sub>1</sub>	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process <sub>0</sub> initiates I/O
4	Blocked	Running	Process <sub>0</sub> is blocked, so Process <sub>1</sub> runs
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process <sub>1</sub> now done
9	Running	-	
10	Running	-	Process <sub>0</sub> now done

- process state의 diagram
  - running에서 IO event 발생 시 IO가 끝날 때 까지 기다리는 waiting으로 전호나
  - running에서 interrupt 발생 시 잠깐 CPU 제어권을 뺏기므로 레디 큐 뒤로 밀려남 그래서 다시 ready 가 됨



## Process Scheduling 맛보기

- 다음엔 어떤 프로세스를 실행시킬지 결정하는 방법
  - 자주 발생하므로 빨리 처리하는 방법이어야 함
- basic approachs
  1. non-preemptive(비-선점형) scheduling
    - 프로세스가 스스로 CPU를 양보하도록 기다림
    - 프로세스들이 cooperative 해야함
  2. preemptive scheduling
    1. 모든 현대 스케줄러는 preemptive
    2. 스케줄러는 프로세스를 인터럽트하고 강제로 context switch가 가능
- 여기보단 process state 부분에 더 어울리는 그림같지만...



## Process 구현해보기

- PCB : process control block / process descriptor
  - PCB는 다음과 같은 정보들을 저장하는 곳
- 각 프로세스와 관련된 정보, task control block
  - **process state** : running, waiting, etc
  - **program counter** : 다음에 실행될 instruction의 위치
  - **CPU register** : 모든 process-centric(프로세스 관련) 레지스터
  - **CPU scheduling information** : priorities와 scheduling queue pointers
  - **memory management information** : 프로세스에 할당된 메모리
  - **process accounting information** : CPU 사용 시간, 경과된 clock time, ...
  - **I/O status information** : 프로세스에 할당된 IO, open files 목록

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

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip; // Index pointer register
    int esp; // Stack pointer register
    int ebx; // Called the base register
    int ecx; // Called the counter register
    int edx; // Called the data register
    int esi; // Source index register
    int edi; // Destination index register
    int ebp; // Stack base pointer register
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
    RUNNABLE, RUNNING, ZOMBIE };
```

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem; // Start of process memory
    uint sz; // Size of process memory
    char *kstack; // Bottom of kernel stack
                // for this process

    enum proc_state state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    struct context ctx; // Switch here to run process
    struct trapframe *tf; // Trap frame for the
                        // current interrupt
};
```

## Context Switch에 대하여

- 프로세스의 CPU를 다른 프로세스로 전환시키는 것
- administrative overhead
  - 레지스터와 메모리 맴을 save하고 restore
  - 메모리 캐시를 flush하고 reload
  - various table과 list들을 update
  - ...
- 오버헤드는 hardware support에 달렸다
  - advanced memory management technique은 다른 context로 전환하기 위해 추가 데이터를 필요로 함
- 보통 초당 100번, 1000번 정도 스위칭이 일어남
- 프로세스의 context는 PCB로 표현
- CPU가 다른 프로세스로 switch할 때 context switch를 통해 현재 프로세스 state를 저장하고 새 프로세스의 saved state를 load해야함.
- context switch time 자체가 오버헤드. 스위칭 동안 유용한 작업을 하지 않음
  - OS와 PCB가 복잡할수록 context switch 시간이 길어짐
  - hw support에 영향을 많이 받음
    - 어떤 하드웨어는 CPU마다 multiple sets of registers를 제공해서 여러 context가 한 번에 load 됨
- context switching time 실제 예시

#### ❖ Example

- Total uptime: 101,197.65 sec (/proc/uptime)
- Total 135,098,709 context switches (/proc/stat)
- Average 1,335 context switches / sec (for all 8 CPUs)
- Roughly 167 context switches / sec / CPU

