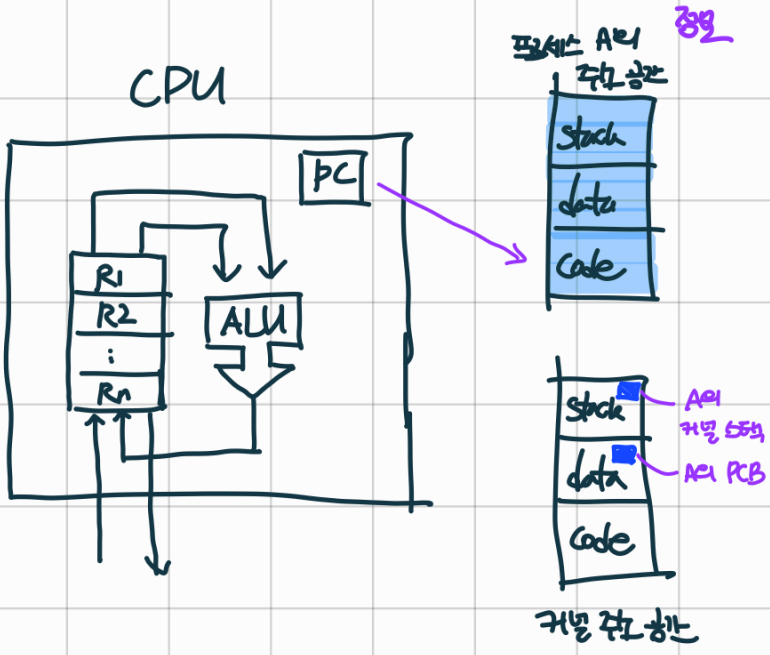
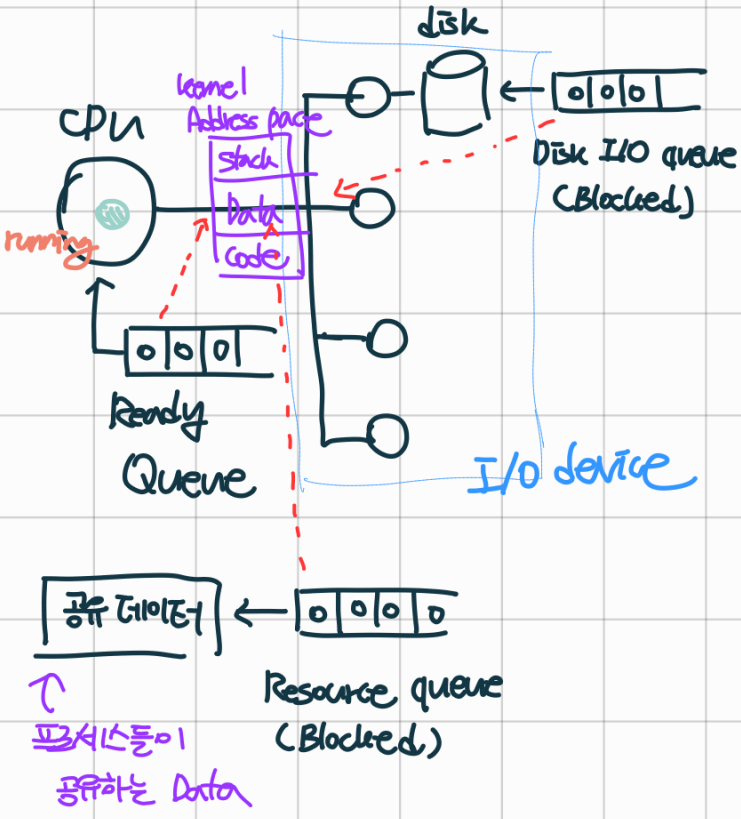
프로세스의 개념

- 프로세스의 문맥 (context)
  - CPU 수행 상태를 나타내는 하드웨어 문맥
    - Program Counter
    - 각종 레지스터
- 프로세스의 주요 공간
  - code, stack, data
- 프로세스 관련 커널 자료 구조
  - PCB (Process Control Block)
  - kernel stack ↳ CPU 사용, Memory 사용 정보

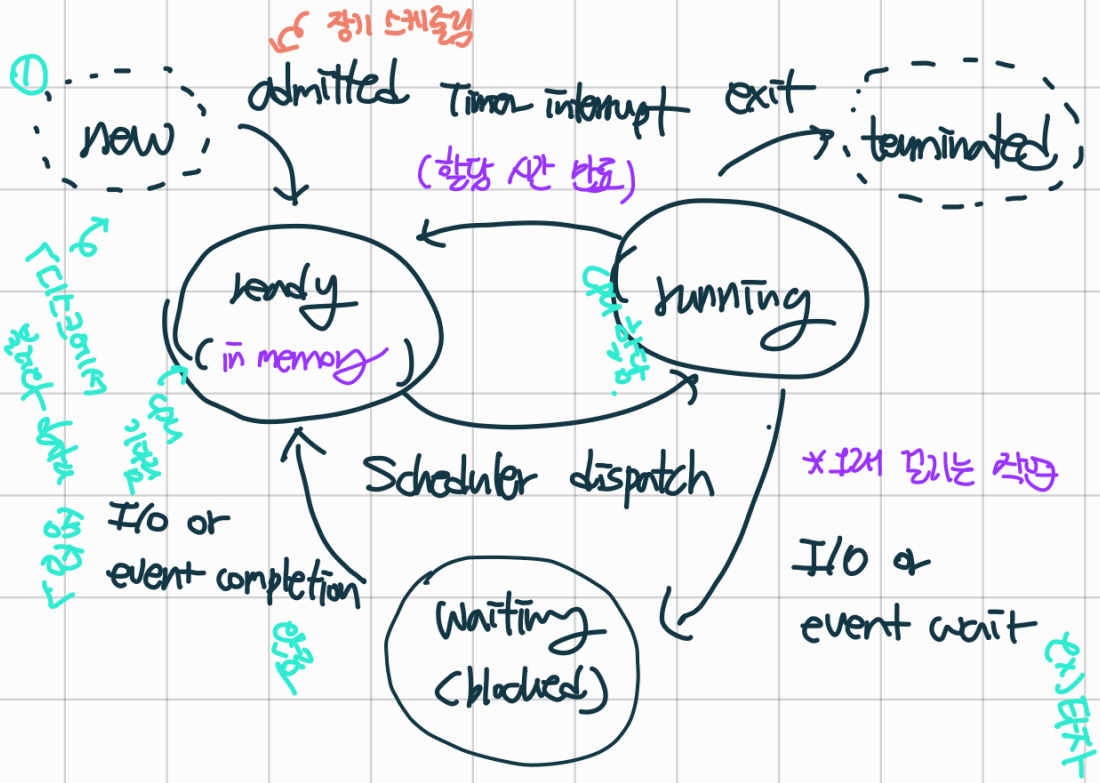


프로세스의 상태 (Process State)



- **Running**
  : CPU를 잡고 instruction을 수행중인 상태
- **Ready**
  : CPU를 기다리는 상태
  (메모리 등 다른 조건 모두 만족)
- **Block (wait, sleep)**
  - CPU를 주어도 당장 instruction을 수행할 수 X
  - Process 자신이 요청한 event (I/O)가 즉시 만족되지 않아 기다리는 상태
    ex) disk에서 file을 읽어와야 하는 경우
- New : 프로세스 생성
- Terminated : 수행 (execution)이 끝난 상태

프로세스 상태도



① time interrupt : CPU를 내놓고 Ready Queue에 줄을 서는 것

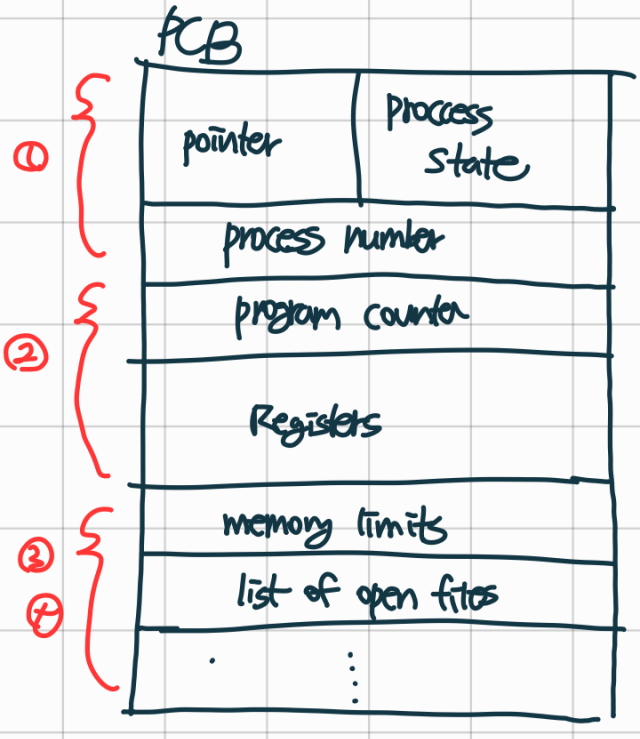
② CPU나 기계에 실행을 하다가 보내지는 작업은 보통 CPU 가 기계에 실행이 안되는 상태

③ 입출력 요청 대해서 종료

PCB

kernel 이 관리하는 data에 접근

- OS가 프로세스를 관리하기 위해 프로세스당 유지하는 정보
- 구성요소
  - ① OS가 관리상 사용하는 정보 :
    - Process state, Process ID
    - scheduling information, priority
  - ② CPU 수행 관련 하드웨어 값
    - Program Counter, registers
  - ③ 메모리 관련
    - code, data, stack의 위치정보
  - ④ 파일 관련
    - Open file description

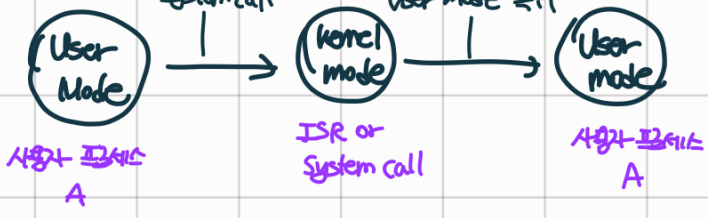


Context Switch (문맥 교환)

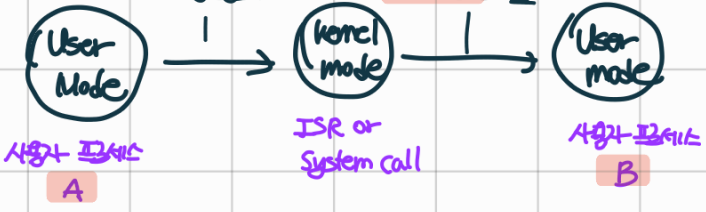
- CPU를 한 프로세스에서 다른 프로세스로 넘겨주는 과정
  - PCB를 내어주는 프로세스의 상태를 그 프로세스의 PCB에 저장
  - CPU를 새롭게 얻는 프로세스의 상태를 PCB에서 읽어옴

vs context switch X

① Interrupt or System call → 문맥 교환 없이 user mode 복귀



② timer Interrupt or I/O 요청 System call → 문맥 교환 일어남



* System call이나 Interrupt 발생 시 반드시 context switch가 일어나는 것은 X

프로세스를 스케줄링하기 위한 큐

- Job queue : 현재 시스템 내에 있는 프로세스 집합
- Ready queue : 메모리 내에 있으면서 CPU를 잡아서 실행되기 기다리는 프로세스 집합
- Device queue : I/O device 처리를 기다리는 프로세스 집합

Scheduler

- Long-term scheduler
(Job scheduler)

- 시작 프로세스 중 어떤 것들을 ready queue로 보낼지 결정
- 프로세스에 memory(및 각종자원)을 주는 문제 ↪ 메모리에 올라가는 것 제어
- degree of Multiprogramming을 제어
- time sharing system에는 보통 장기 스케줄러가 없음 (무조건 Ready)

☆ - Short-term Scheduler
(CPU scheduler) ↪ 자주 호출

- 어떤 프로세스를 다음번에 running 시킬지 결정
- 프로세스에 CPU를 주는 문제
- 충분히 빨라야 함 (millisecond 단위)

- Medium-Term Scheduler (Swapper)

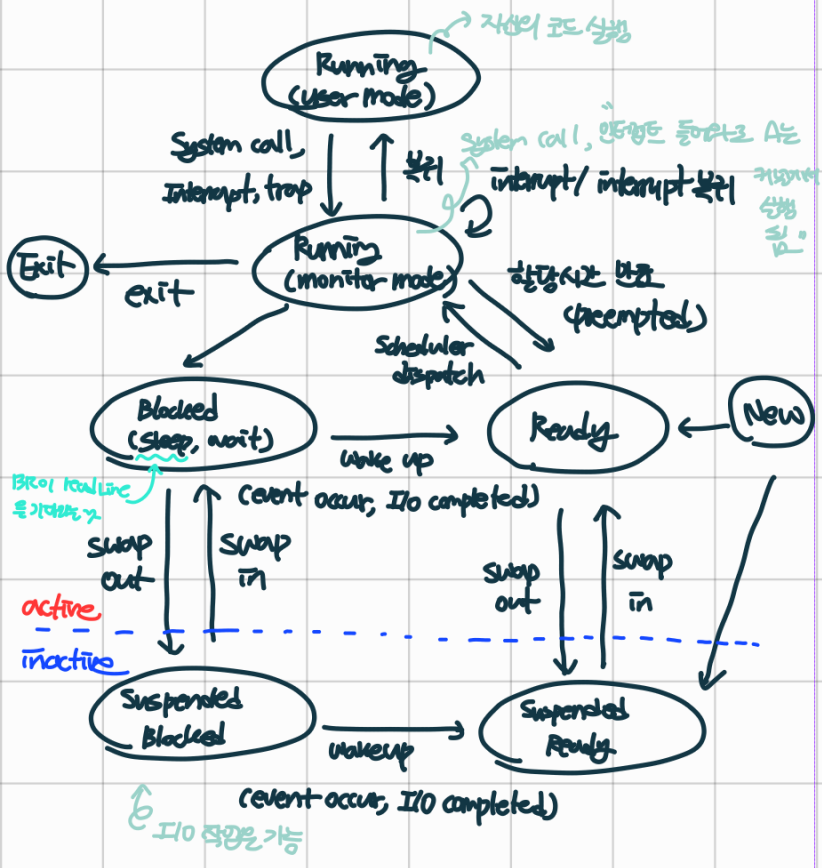
- 여유공간 마련을 위해 프로세스를 통째로 메모리에서 디스크로 쫓아냄
- 프로세스에게서 memory를 뺏는 문제
- degree of Multiprogramming을 제어 ↪ 메모리에 올라간 프로그램의 수

프로세스의 상태

- Running : CPU를 잡고 instruction을 수행중인 상태
- Ready : CPU를 기다리는 상태 (메모리 등 다른 모든 자원은 얻음)
- Blocked (wait, sleep) — 수행 중인 상태
  - I/O 등의 event를 (스스로) 기다리는 상태
  - ex) disk에서 파일을 읽어와야 하는 경우
- Suspend (stopped) — 정지된 상태 ↪ 대부분의 이유 프로세스 메모리를 통째로 빼앗김
  - 외부적인 이유로 프로세스의 수행이 정지된 상태
  - 프로세스는 통째로 디스크에 swap out 된다
  - ex) 사용자가 프로그램을 일시 정지시킨 경우 (Break Key)
    시스템이 여러 이유로 프로세스를 잠시 중단시킴 (메모리에 너무 많은 프로세스가 올라와 있을 때).

* Blocked : 자신이 요청한 event가 만족되면 ready
* Suspended : 외부에서 resume해 주어야 active

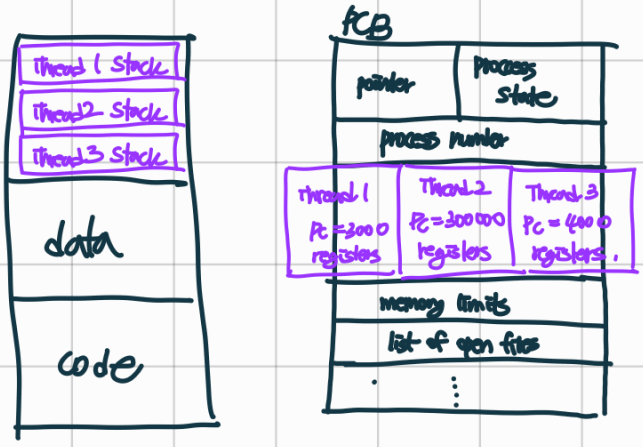
프로세스 상태도



Thread (cpu의 실행 단위, light weight process)

- Thread 구성
  - program counter
  - register set
  - stack space : 각자 분리
- Thread가 동료 thread와 공유하는 부분 (=task)
  - code section
  - data section
  - OS resources
- 전통적인 개념의 heavy weight process는 하나의 thread를 가지고 있는 task로 볼 수 있다.

Thread



- Thread 사용 X
  : OS 내에서 Program A → Program B 로 context switch 일어나서 overhead가 많이 일어나는 작업
  save, load
- Thread 사용 O
  : 동일 thread 안에서 A 에서 B로 CPU 변경 시 Overhead 없음

Thread

- 다중스레드로 구성된 태스크 구조에서는 하나의 서버 스레드가 blocked (waiting) 상태인 동안에도 동일한 태스크 내의 다른 스레드 실행 (Running)을 통해 빠른 처리를 할 수 있다.
- 동일한 일을 수행하는 다중 스레드가 협력하여 높은 처리율 (throughput)과 성능 향상을 꾀할 수 있다.
- 스레드를 사용하면 병렬성을 높일 수 있다.

- Benefits

- Responsiveness
- Resource Sharing : 하나의 스레드 코드, data, 소스 공유
- Economy
- Utilization of MP Architectures
  : 병렬 작업

Implementation of Threads

- Some are supported by kernel → kernel thread
  - Windows, Solaris, Digital Unix
- Others are supported by library → User Threads
  - POSIX Pthread, Mach C-threads...
- Some are real-time threads

운영체제는 thread의 존재를 모른다

프로세스 생성

- parent process가 children process 생성
- 프로세스의 트리 (계층 구조) 형성
- 프로세스는 자원을 필요로 함
  - 운영체제로 부터 받는다
  - 부모와 공유한다
- 자원의 공유
  - 부모와 자식이 모든 자원을 공유하는 모델
  - 일부를 공유하는 모델
  - 공유하지 않는 모델
- 수행 (Execution)
  - 부모와 자식은 공존하며 수행되는 모델
  - 자식이 종료 (terminate) 될 때까지 부모가 기다리는 (wait) 모델
- 주소 공간 (Address space)
  - 자식은 부모의 공간을 복사함 (binary and OS data)
  - 자식은 그 공간에 새로 프로그램을 올림
- 유닉스의 예
  - fork() 시스템 콜이 새로운 프로세스 생성
    - 부모를 그대로 복사 (OS data except PID + binary)
    - 주소 공간 할당
  - fork 다음에 이어지는 exec() 시스템 콜을 통해 새로운 프로그램을 메모리에 올림

프로세스 종료 (Termination)

- 프로세스가 마지막 명령을 수행한 후 OS에게 이를 알려줌 (exit) ~> 자발적 종료 (자식 → 부모 알림)
  - 자식이 부모에게 output data를 보냄 (via wait)
  - 프로세스의 각종 자원들이 운영체제에게 반납됨
- 부모 프로세스가 자식의 수행을 종료시킴 (abort) ← 강제적 종료
  - 자식이 할당 자원의 ① 한계치를 넘어섬
  - 자식에게 할당된 ② 태스크가 더 이상 필요하지 않음
  - 부모가 종료 (exit)하는 경우
    - OS는 부모 프로세스가 종료하는 경우 자식이 더 이상 수행되도록 두지 않는다.
    - 단계적인 종료

fork() 시스템 콜 ~> 자식 생성

- 프로세스는 fork System Call로 생성되었다
  - 새로운 Address space를 복제하며 생성

```
ex) int pid;
    pid = fork();
    if (pid == 0)        ~> 자식
        printf("this is child");
    else if (pid > 0)    ~> 부모
        printf("this is parent");
```

Exec() 시스템 콜

- exec() 시스템콜로 인해 다른 프로세스를 실행할 수 있음
  - 새로운 프로그램이 caller의 메모리 이미지 교체

```
ex) int pid;
    pid = fork();
    if (pid == 0)
        printf("this is child");
새로운 프로세스 → execlp("bin/data", "bin/data", (char*)0);
생성
    else if (pid > 0)
        printf("this is parent");
```

현재 부모의 실행 순서 보다 자식이 수행