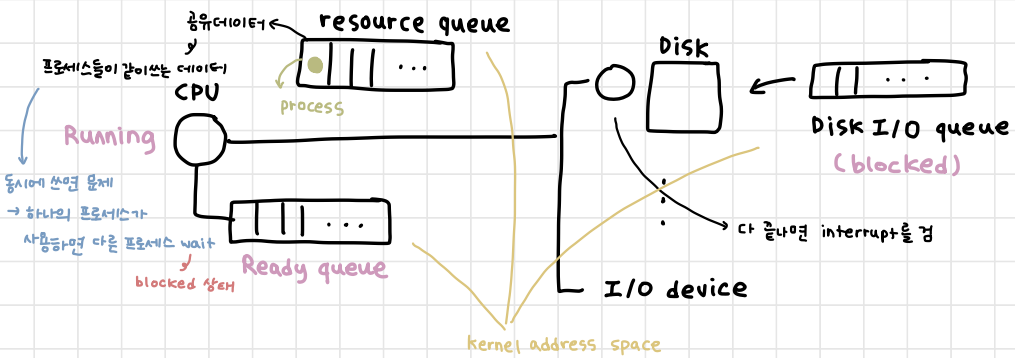
OS_프로세스 관리

Process is a **program in execution**

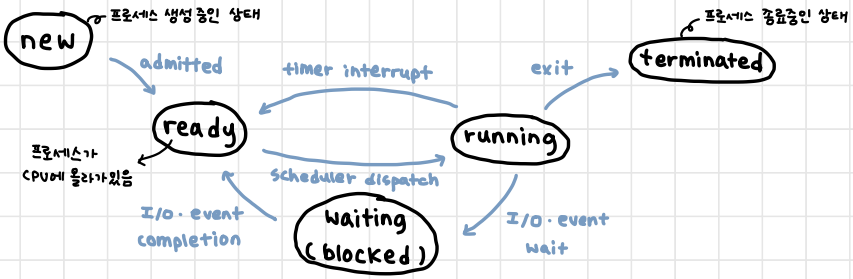
↳ 문맥 (프로세스의 상태를 나타내는)
- CPU 수행 상태를 나타내는 하드웨어 영역 ↬ program counter / 각종 register
- 프로세스의 주소 공간 ↬ code, data, stack
- 프로세스 관련 커널 자료 구조 ↬ PCB(process control block) / kernel stack
  ↓
  프로세스를 운영체제가 관리하면서 가지고있는 자료구조 ⇒ CPU 얼마나 썼는지 / 메모리 어떻게 썼는지 …

프로세스의 상태 → 계속 변경되며 수행됨

- **Running** : CPU 잡고 introduction 수행중인 상태
- **Ready** : CPU 기다리는 상태(메모리 등 다른 조건을 모두 만족하는)
- **Blocked** : CPU를 주어도 당장 introduction 수행할 수 없는 상태
  ↓ wait, sleep
  ↬ Process 자신이 요청한 event가 즉시 만족되지 않아 이를 기다리는 상태



프로세스 상태도



Process Control Block (PCB)

운영체제가 각 프로세스를 관리하기 위해 프로세스 당 유지하는 정보

① OS가 프로세스 관리하는데 필요한 정보

→ process state · process ID

scheduling information · priority (process간 CPU 순서 정하기 위해)

② CPU 수행 관련 하드웨어 값

→ program counter · registers

(문맥교환 위해) CPU를 한 프로세스에서 다른 프로세스로 넘겨주는 과정

↓ CPU 뺏길 땐 어디까지 수행했는지 저장해야 함

⇒ 그 프로세스의 PCB에 저장

CPU는 새롭게 얻는 프로세스 상태 PCB에서 불러옴

③ 메모리 관련

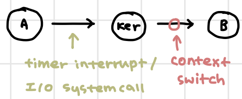
→ code · data · stack의 위치 정보

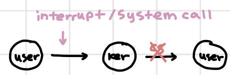
④ 파일 관련

→ open file descriptors

※ system call이나 interrupt 발생시 반드시 context switch가 일어나는게 아님

A에서 결과적으로 다른 프로세스 B로 넘어가면 context switch





↳ A때문에 넘어간게 아님 다시 A로 돌아옴

↓ 이때도 context 일부 PCB에 저장해야 함

Job queue → 현재 시스템 내에 있는 모든 프로세스들의 집합

Ready queue → 현재 메모리 내에 있으면서 CPU 잡아서 실행되기를 기다리는 프로세스의 집합

Device queues → I/O device의 처리를 기다리는 프로세스들의 집합

scheduler (스케줄러)

long-term scheduler

→ memory scheduler

admitted 주는게 장기스케줄러

new → ready

프로세스 들어올 때 메모리에 올라가도록 허락

- 시작 프로세스 중 어떤 것들을 ready queue로 보낼지 결정
- 프로세스에 memory를 주는 문제
- degree of Multiprogramming을 제어 (메모리에 올라가는걸 허락하는 역할이라 몇개를 메모리에 올릴지 정함)
- time sharing system에는 보통 장기 스케줄러가 없음

short-term scheduler

→ CPU scheduler

CPU를 누구에게 · 얼마나 줄지 결정

⇒ 자주 호출됨

- 어떤 프로세스를 다음번에 running 시킬지 결정
- 프로세스에 CPU 주는 문제
- 충분히 빨라야 함

time sharing system이 장기 스케줄러 대신 두는 것

medium-term scheduler

→ swapper

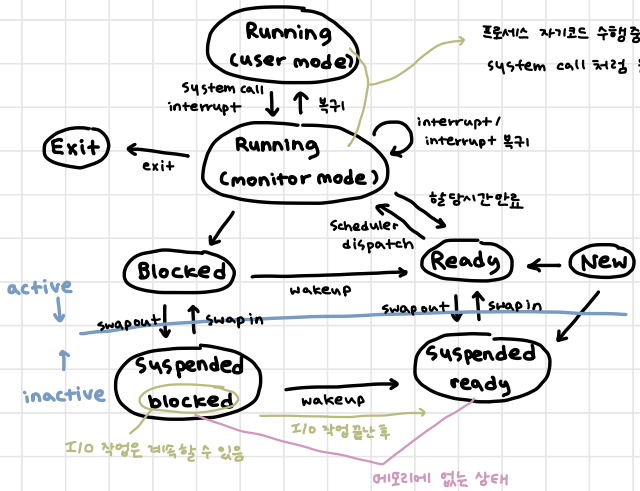
- (장기스케줄러 없어서 모두 메모리에 올라옴 → 메모리 부족할 때) 여유공간 마련을 위해 프로세스 통째로 메모리에서 디스크로 쫓아냄
- 프로세스에게서 memory 뺏는 문제
- degree of multiprogramming 제어

↓ 프로세스 상태 추가됨

Suspended (stopped)

- 외부적인 이유로 프로세스 수행이 중단된 상태
- 프로세스가 메모리에서 쫓겨난 상태

↓ 외부에서 resume 해 주어야 active



프로세스 자기코드 수행중이면 user mode에서 running

system call 처럼 운영체제 코드가 수행중이면 kernel mode에서 running

↳ kernel이 running하는게 아님

Thread

같은 code지만 실행하는 함수의 위치가 다 다름 → 각각 process로 만드는 것 비효율적

→ 여러 개의 동일한 프로그램 띄우더도 하나의 프로세스만 생성

stack은 thread 별로 따로 둠

→ program counter (어디 실행하고 있는지) · register는 따로 둠

↳ CPU 수행부분만 별도로 가지고 있음

↳ CPU 환경 부분만 따로 빼서 관리

↳ context switch의 overhead 필요없음

→ program counter · register set · stack space만 별도로 가지고 있고 code section · data section · OS resource는 공유

⊕ 다중 스레드인 경우 하나의 서버 스레드가 blocked인 경우에도 동일한 태스크 내의 다른 스레드 실행 가능

Responsiveness · Resource sharing · Economy · Utilization of MP architectures

병렬성 - CPU 여러 개일때

Kernel Threads → OS가 thread 알고 구현하는 것

User Threads → OS가 thread 존재 모름

프로세스 생성

→ 부모 프로세스가 자식 프로세스 생성 ~ 프로세스의 트리 형성 (동일한)

↳ systemcall로 OS에게 만들어달라고 요청 ⇒ fork() system call

프로세스 자원 필요로 함 → 운영체제로부터 받는다 / 부모와 공유

(부모와 자식 공존하며 수행되는 모델

자식이 종료될 때까지 부모가 기다리는 모델

↳ blocked 상태

자식은 부모의 주소 공간 복사함 → 여기에 새로운 프로그램 올림

↳ 똑같은 위치의 program counter로 동일한 위치에서 시작됨

fork() → exec() 를 통해 새로운 프로그램 덮어씌움

exit으로 종료된 것을 OS에게 알림 + 부모에게 종료됨을 알림

부모가 자식 프로세스 수행 종료시킴 → abort

fork() ⇒ 어떤 프로세스가 부모 프로세스인지 자식 프로세스인지 모름 ⇒ pid

↳ fork의 return value

자식의 번호 → 부모 받음 · 자식은 0 받음