운영체제

운영체제💻

Operating System

운영체제 서론

> 운영체제란 무엇인가?

PC를 구입하면 딸려오는것? Windows, Linux, Mac Os, MS DOS.. 등등

> 그렇다면.. 운영체제가 없는 컴퓨터는 어떻게 될까? 컴퓨터 구조로 생각해보자!

컴퓨터라는 것은 이런 구조로 되어있다.

1 Processor(처리기), Main Memory(메모리) 가 있다. 기본적으로 컴퓨터 전원을 키면, 프로세서가 메모리에 있는 명령을 들고와서 실행을 하고, 그다음 명령을 또 들고와서 실행을 하고.. 하는 식으로 작동을 하는데?

2 ⇒ 곧, 메모리에 명령들(Instruction)을 기록해 둬야 하는데? (명령들의 집합이 프로그램) 만약 운영체제가 없다면 메모리에 자기 멋대로의 임의의 값이 들어있게 된다.

3 메모리는 `휘발성` 메모리이기 때문에? 기본적으로 실행 프로그램들은 하드디스크에 들어있게 되는데, 이걸 메모리에 올리는것도 못함 OS가 없으면.

4 현대 컴퓨터는 또, 여러개의 프로그램을 동시에 올려서 실행을 하는데? 프로세서는 하나뿐이잖아? 이런 역할도 OS가 함.

5 뭘 하드디스크에 저장하거나, 프린터를 연결해서 실행시키거나.. 하는것도 다 OS.

`정리`

1. 성능향상(Performance) = 컴퓨터의 하드웨어를 잘 관리(프로세서, 메모리, 디스크, 주변장치 등등) + 좋은 OS가 있다면 하드웨어 성능도 더 끌어낼 수 있음.
2. 사용자에게 편의성(Convenience) 제공 = 애들도 몇번 해보면 금방 씀. (예전엔 컴 쓰는게 어려워서 컴퓨터를 사용하는 Operator라는 직업도 있었음)

⇒ 결론 : `컴퓨터 하드웨어를 관리하는 프로그램` Control program for computer

> 부팅(Booting)

컴퓨터는 프로세서 + 메인 메모리(주기억장치)가 가장 핵심적인 장치이고, 하드디스크(보조기억장치)를 가지고 있음.

여기서 메인메모리는 2가지로 나뉘는데, 램과 롬임. (근데 메인메모리 대부분은 RAM)

1 RAM(보통 DRAM) → 일반적으로 요즘 작은거라 해봤자 4GB 부터 꼽음

2 ROM(Read Only Memory) → 많아야 수백 kb 정도. 컴퓨터에서 약간의 롬이 필요함. (EX)휴대폰의 플래시 메모리)

근데 이러면 롬은 왜필요하나? ⟹ RAM에 있는 내용은 `휘발성` 메모리라 컴퓨터 껐다 키는 순간 죄다 날아감. 롬은 전원을 꺼도 내용이 유지되고 있음.

> 주의 : 하드디스크도 전원과는 관계 없음. 껐다 킨다고 날아가는게 아님. 그래서 ROM이 하드디스크냐? 이건 아님. 카테고리가 이미 'Main Memory' 니까.

폰이나 PC의 전원을 킨다면? 가장 처음으로 프로세서는 (RAM이 아니고) `ROM`안에 있는 Instruction을 읽어와서 실행하게 됨!! ⇒ 다음과 같은 순서

1. POST(Power-On Self-Test)가 제일먼저 실행 = 전기 키면 셀프테스팅 하는것(제대로 환경설정이 되어있는가, 키보드가 꽂혀 있는가, 메모리는 얼마 있는가 등등 체크)
2. Boot Loader 실행 = 컴퓨터는 보통 `하드디스크`안에 OS를 설치해 두는데, 부트로더가 하는 일은 컴퓨터 켜지면 하드디스크를 뒤져서 OS를 메인 메모리(RAM 영역)로 올려줌.

   ⇒ 이렇게 메인 메모리로 올리는 것을 `Boot` 라고 부름.
3. 2단계 까지 했으면 ROM은 자기 역할을 끝냄.

OS가 Main Memory로 올라오면 바탕화면 뜨고, 우리가 익숙한 그 화면이 보이게 됨. 지금은 OS가 메인 메모리에 상주하면서 이제 받는 명령을 처리할 준비가 되었다는 뜻. (컴퓨터 전원을 끄면 OS도 메인 메모리에서 날아감)

조금 헷갈릴수도 있는 부분이, 컴퓨터를 키고 이것저것 프로그램(게임이나 크롬, 한글등)을 실행할때 그걸 꺼버리면 메인 메모리에서도 내려가는데, OS는 컴퓨터 전원이 들어와 있는 동안은 메모리에 상주함. = 그래서 OS를 `Memory Resident` 라고 함.

OS는 사실 두 부분으로 나뉨. Kernel(핵심) & Shell(껍질)

OS가 하드웨어를 감싸서 관리를 해주고 있는 형태라고 생각하면 되는데. 실제로 OS가 하드웨어를 제어하고 관리하는 부분을 `OS의 커널, Kernel(핵심, 핵)` 이라고 부름.

가령, 프로그램을 실제로 실행하고 싶은 경우 (윈도우즈의 경우 유저 인터페이스는 우리가 익숙한 바탕화면 그거임) 아이콘에 마우스 화살표 갖다 대고 더블클릭해서 실행 하는데? 이건 내가 어떤 프로그램을 실행하라! 라고 명령을 내린거임. 이런 명령을 내릴 수 있도록 만들어 주는 것을 운영체제의 껍데기 부분, `Command Interpreter` 라고 부름. (리눅스는 기본적으로 CLI 라서 $ (달러사인) 오른쪽에 ls 이런거 치거나 함 + who 치면 어떤 유저가 쓰는지, df(disk free)치면 보조기억장치 전체 용량중 사용 용량 이런거 확인 가능)

리눅스로 예를 들면, 이런 사용자 명령어 who 내리면 Shell이 이게 뭔 명령인지 해석하여 동작해서 OS가 일할 수 있도록 기반을 마련.

`정리`

- `커널` = 실제 관리 프로그램 (Cpu, Memory 등 관리)
- `쉘` = 사용자가 명령 내린걸 해석해서 결과를 화면에 보여주는 껍질

그래서 커널은 관리프로그램이라 눈에 거의 안보이는것.

⟹ 만약 누가 너 리눅스 사용법 아냐? 라고 물었을때 $ 옆에 ls 하면 목록 뜨지 ㅇㅇ, 뭐 그런 측면에서 적당히 사용법 알아~ 라고 한다면 이건 커널을 안다는게 아니라 쉘(사용법)을 안다는거임

```
# 운영체제 내용을 배운다고 하면 그러니까 이 강의에서는 "커널" !!! 을 배우는거임.Application[ OS [ Hardware ]  ]  <= 이런 포함관계
```

일반적으로 어플리케이션은 OS 위에서 실행되니까, OS가 달라지면 앱이 실행이 안됨.

뭔뜻이냐면? 윈도우에서 한글 프로그램 파일 긁어다가 맥에 집어넣으면 이거 안됨. 사실 하드웨어는 뭐 인텔 CPU 써서 같을 수 있는데? OS가 다르니까 안되는거.

`어플리케이션은 하드웨어 위에서 도는게 아니라 OS위에서 도는거임`

컴퓨터 구조 수업에서는 컴퓨터를 CPU 메모리 보조기억장치의 세트로 개념도를 그렸다면, 이 수업에서는 Application → Os → Hardware 식으로 그릴 수 있음.

OS는 결국 정부와 비슷하다고 할 수 있는데, 주어진 자원을 활용하여 일을 하는것. 정부에 여러 부처가 있듯, OS에는 `Processor Management`, `Memory Management`, `IO Management`(프린터 키보드 등 관리), `File Management` , `Network Management`, `Security or Protection Management` 등등 여러가지가 있음. 이중 제일 중요한 부처는 Processor Management이고 그다음은 Memory Management임. 이 두개가 수업의 핵심.

OS의 다른 이름. 하드웨어를 자원 이라고 함!

= `자원관리자` (resource manager) = `자원할당자` (resource allocator)

운영체제 역사

역사

1. No O/S

- 1940년대 말 (2차대전 중 개발)
- 거대해서 한 건물안에 설치
    1. 카드리더 : 가장 큰 장치, 입력장치, 천공카드(구멍 뚫린 종이) 입력
    2. 처리기 : processer & memory
    3. 프린터 : 출력장치, output을 line print
- 동작 순서 : card reader -> memory -> proessing -> printer
    1. 프로그램을 천공카드에 표시해 메모리에 적재
    2. 컴파일러를 천공카드에 표시해 메모리에 적재
    3. 컴파일러가 프로그램 번역 -> 기계어
    4. 처리기가 기계어 실행 -> 출력

2. Batch processing system (일괄처리) - 최초의 O/S

- 프로그램을 수행할 때마다 컴파일->링크->로딩 순서를 오퍼레이터가 직접 입력함

  => 이러한 과정을 하나의 프로그램으로 작성해 메모리에 안에 할당해 자동화한 것이 batch processing
- resident monitor : 메모리에 상주하며 일련의 일(컴파일, 링크, 로딩 등)을 하는 프로그램

3. Multiprogramming system (다중 프로그래밍)

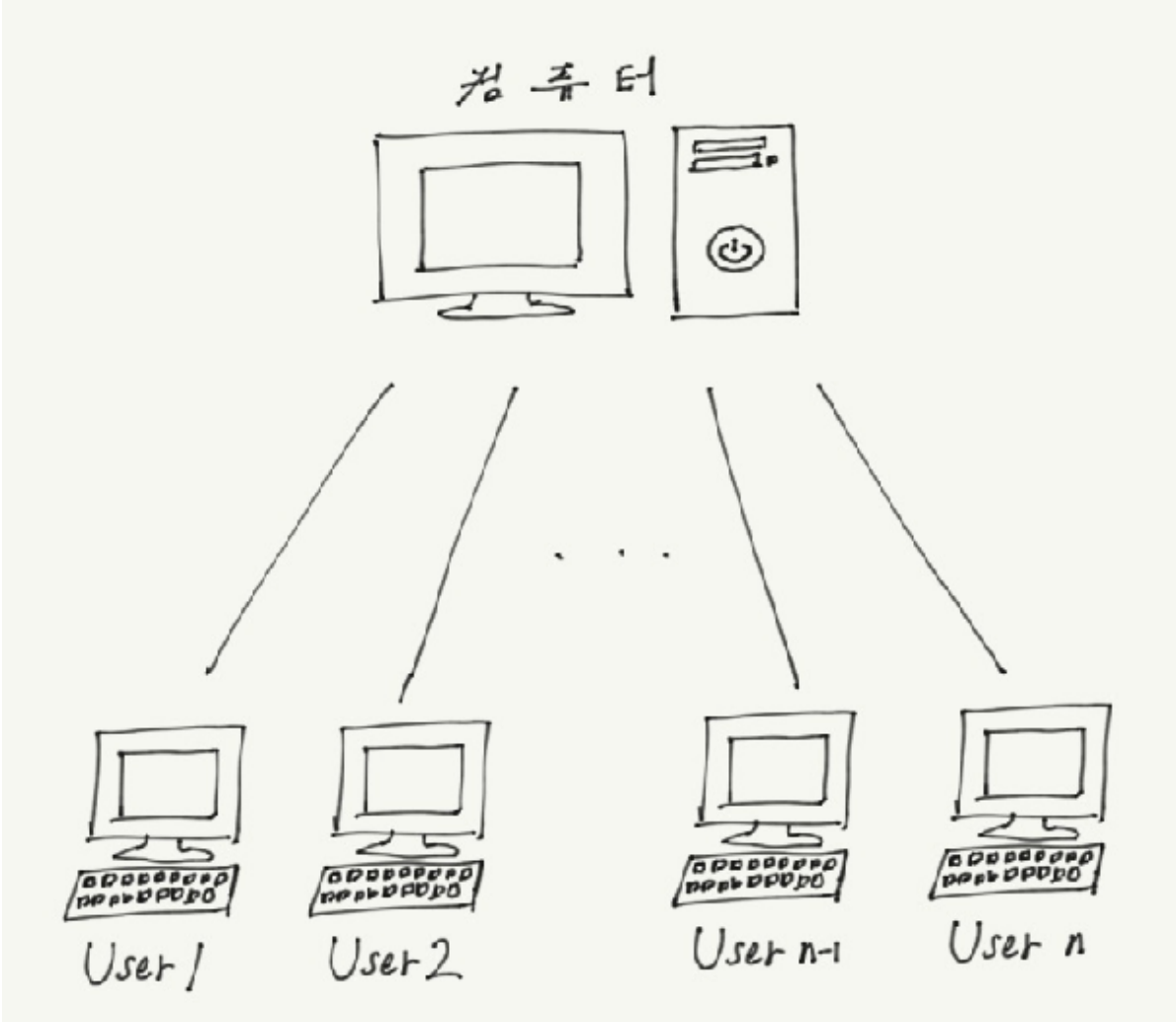
- 1960년대
- 하드웨어의 발달
    1. 메인 메모리 외에도 하드디스크 생산
    2. 메인 메모리 커짐(진공관에서 발전)
    3. 프로세스 속도 빨라짐
- 메모리에서 유저 프로그램 동작 시
    1. CPU는 빠르게 동작하나 I/O가 느림
    2. I/O실행 중 CPU idle (idle : 어떠한 프로그램에서도 사용되지 않는 상태)
- 메모리에 여러개의 프로그램을 올려 작업을 실행 (job1, job2, job3 ...)
    - CPU 유휴 시 다른 프로그램의 연산 작업 수행

      => Multiprogramming system
- 고려사항
    - 프로그램 동작 순서를 정하는 CPU scheduling 중요
    - 메모리 내에서 유저프로그램의 위치 중요

- 다른 영역의 프로그램 침범을 막기위한 보호 중요

4. Time-sharing system (시공유 시스템)

- 1960년대 말 Unix가 대표적
- 모니터, 키보드의 개발 => interactive system의 구현
- 컴퓨터의 가격이 비쌈 => 컴퓨터 한대에 여러대의 단말기(terminal) 연결
  - 단말기 : 모니터, 키보드만 존재



- CPU가 아주 빠르게 프로그램을 스위칭하며 동작 (Time-sharing)
- 새로운 기술의 발전
  - 유저간 통신 가능해짐
  - 동기화 : 동시에 실행 되므로 프로그램 실행 순서를 정하는 것
  - 가상메모리 : 유저가 많아지면 메인 메모리부족 => 하드디스크의 일부를 메인 메모리처럼 사용하는 기술 등장

OS기술 천이

1. 컴퓨터 규모별 분류

- 1970, 1980 년대

  Supercomputer => Mainframe(단말기 수백대) => Mini(단말기 수십대) => Micro

- 현재

  Supercomputer => Server => Workstation => PC => Handheld => Embedded

2. 고성능 컴퓨터의 O/S기술이 handheld/embeded까지 발전

- Batch processing
- Multiprogramming
- Time-sharing (📌이번 수업 때 중점적으로 배울 것)

3. 고등 운영 체제의 등장

- 추후 배울 예정

3_고등운영체제

운영체제

> 강의 링크

3. 고등 운영체제, 인터럽트 기반 운영체제

A. 고등 운영체제

> 기본 운영체제

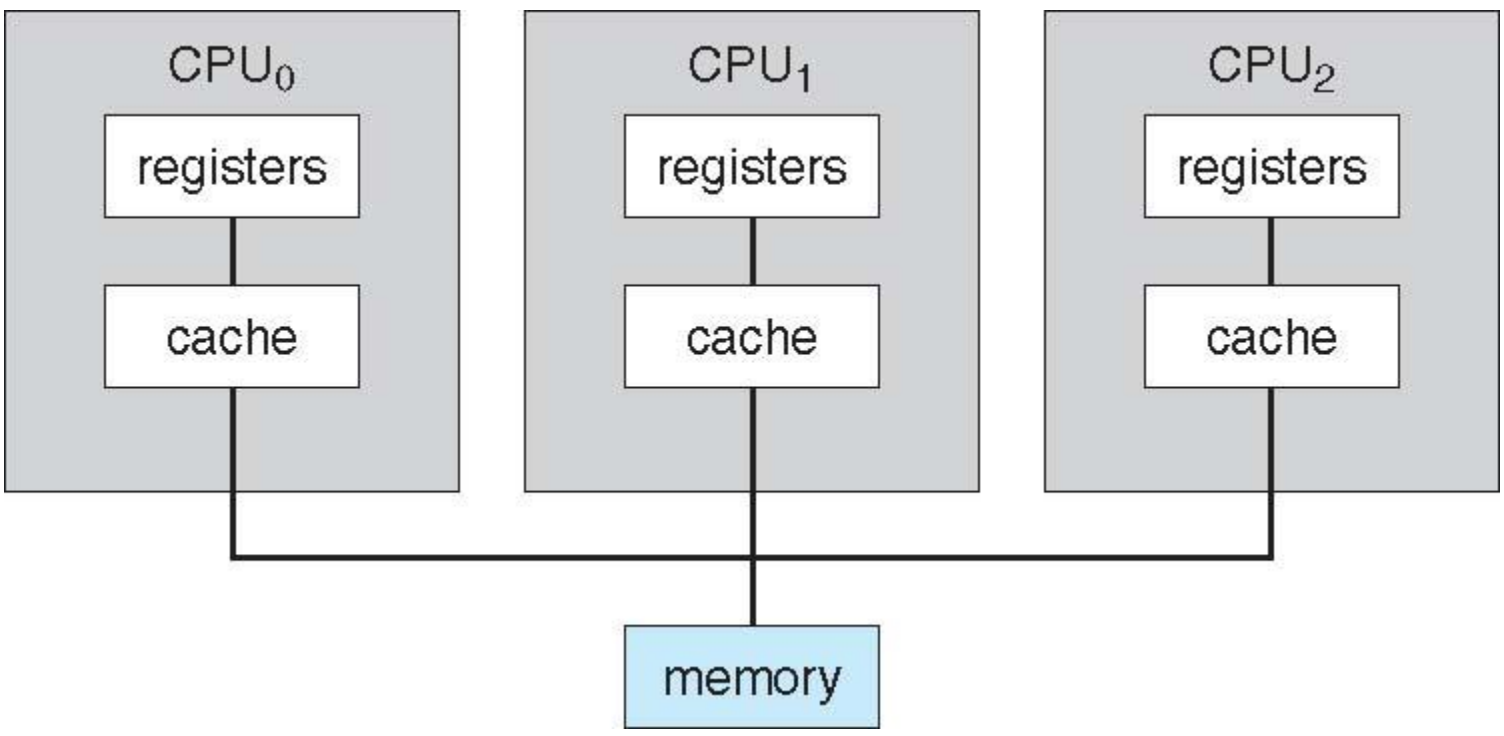
1. 싱글 프로세서 시스템

- 하나의 메모리에 하나의 CPU가 결합된 구조. (메모리는 Bus로 연결)

> 고등 운영체제 - 개요만 다룸

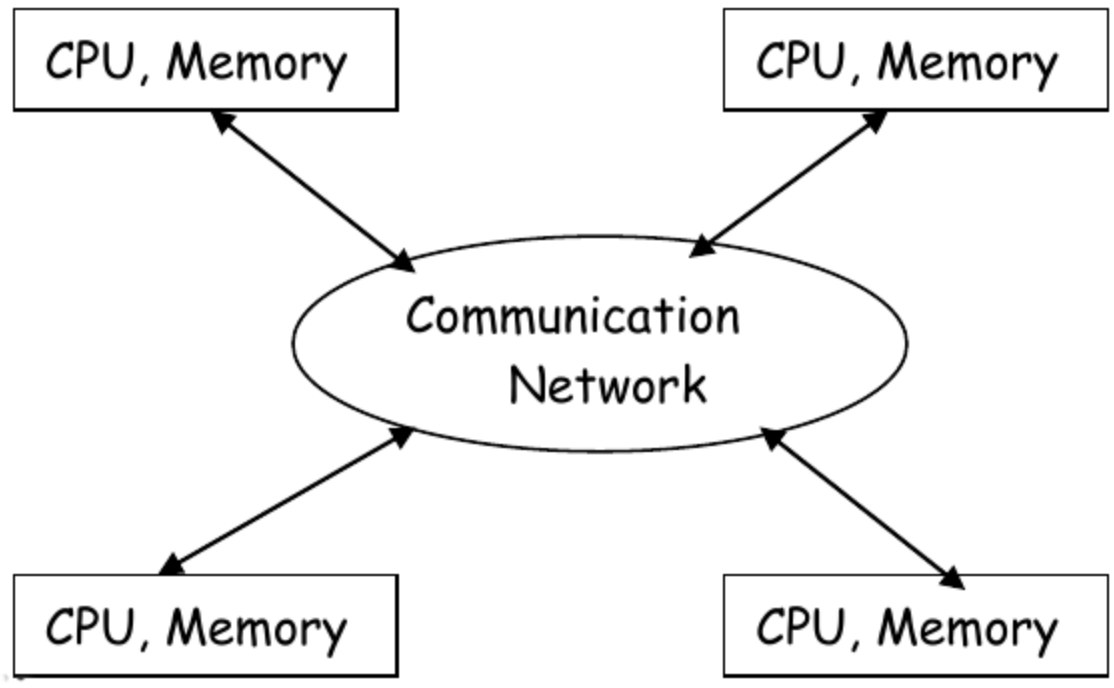
2. 다중 프로세서 시스템(Multiprocessor system)

- 개요
  - `병렬시스템` : 하나의 메모리에 여러개의 CPU가 병렬로 결합된 구조
  - 메인메모리에 결합되어 있기 때문에 강한 결합
  - 멀티코어(quad core 등)랑은 다름. 멀티코어는 1개의 CPU
- 장점
  - Performance : 한번에 많은 계산을 통해 성능 향상
  - Cost : 하나의 좋은 CPU보다 여러개의 싼 CPU를 사용하는게 더 저렴
  - 신뢰성 : 한개의 CPU가 고장나도 다른 CPU가 작업을 대체 할 수 있음



3. 분산 시스템(Distributed system)

- 개요
  - `다중 컴퓨터 시스템` : 컴퓨터 여러대를 연결해 사용하는 개념
  - LAN으로 여러 CPU-Memory 쌍(1:1)을 연결
  - 다중 프로세서 시스템과 구현목적은 같음. 비교적 느슨한 결합(메모리공유 안하다 보니까)
- 예시
  - 일기예보에서 서울, 경기, 대구 등 각 지역에 대한 분석을 각각의 컴퓨터에서 진행
  - 분석 과정에서 일부 중복되는 process는 공유하지만 다중 프로세서 시스템에 비해 느림



- **다중 프로세서 시스템 vs 분산 시스템**

```
다중 프로세서 시스템 : 1대의 컴퓨터(슈퍼컴퓨터), 동시에 여러가지 일을 처리
분산 시스템 : 여러대의 컴퓨터를 네트워크로 연결, 하나의 일을 동시에 여럿이 처리
```

4. 실시간 시스템(Real-time system)

- 개요 : `Real-time OS(RTOS)` 계산이 반드시 어떤 시간내에 끝나야하는 경우
- 예시
- 더하기, 곱하기 연산은 빨리하면 좋다. 그러나 꼭 deadline이 지켜져야하는것은 아니다.
  - 내비게이션 : 교차로 진입전 방향을 결정해야됨. → 실시간 시스템 필수
- 공장자동화, 군사목적 등

참고

```
'리눅스는 분산시스템이고, 윈도우는 다중프로세서 시스템이다'는 틀린 문장이다. 각 OS는 어디 속해있는 개념이 아니다.
리눅스는 분산시스템과 다중프로세서 시스템의 목적으로 각각 사용될 수 있다.
```

B. 인터럽트(가로채기) 기반 시스템

> 현대 운영체제는 인터럽트 기반 시스템

운영체제(OS)

- 운영체제는 상시 메모리에 상주
- 평소에는 사건(event)을 기다리며 대기 상태에 있음
- 어떠한 명령을 지시하는 OS코드(ISR : interrupt service routine)가 내재되어있음. `마우스가 ~하면 어떤 코드를 동작시켜라`
- 운영체제의 위치는 보조기억장치(hard disk 등) 내부임

인터럽트(interrupt)

- 사건을 발생시키는 행위
- 종류
  - 하드웨어 인터럽트 : 마우스 움직임, 클릭, 키보드 타이핑 등을 통해 발생된 전기신호
  - 소프트웨어 인터럽트 : 한글파일 실행, 어셈블리어(add, sub 등)
  - 내부 인터럽트 : 어떤 수를 0으로 나누지 못하므로 이때 에러 발생시키는것, 잘못실행된 프로그램을 강제로 종료하는것
- 유저의 행동에 따른 예제

  @ 한글파일 열기

  1. `처음에 컴퓨터를 킨다`

- 메인메모리는 비워져있음. CPU가 하드디스크를 뒤져 OS를 메인메로리로 가져옴 - 이 과정을 `부팅` 이라고 함.

2. `부팅 완료되어 바탕화면 도착`
   - OS는 대기상태
3. `마우스움직이면`
   - 전기신호 발생해서 CPU에 보냄(interupt)
     - > CPU는 하던일 중지하고 OS로 점프
     - > 마우스 움직이는대로 커서를 움직인다(by ISR)
4. `따블클릭`
   - 마찬가지로 interupt 발생, x,y좌표 찾아서 루틴(ISR)대로 함.
5. `한글 파일(hwp) 열기`
   - OS가 하드디스크를 뒤져 hwp 프로그램을 메인 메모리로 올림(소프트웨어 인터럽트)
     - > 한글파일에 저장된 데이터도 같이 불러옴
   - 왜 한글파일을 여는 코드는 한글파일이 아니라 ISR에 저장되어있을까 ?모든 한글파일(x.hwp)마다 이러한 코드를 저장해놓는건 메모리 낭비
6. `바탕화면에서 한글파일로 화면 전환`
   - OS는 다시 event가 생길때 까지 대기상태

운영체제💻

이중모드 하드웨어 보호

> 이중모드?

`1` 한 컴퓨터를 여러 사람이 동시에 사용하는 환경이 있을 수 있다. EX) 수강신청

- 서버 컴퓨터는 하나인데 접속하는 사람은 매우많음.
- 또는 한 사람이 여러 개의 프로그램을 동시에 사용하는 상황도 있을 수 있음.
- 이러한 환경에서 누군가의 고의 혹은 실수로 인해 전체 프로그램에 영향을 줄 수 있음.
    - 특히 서버 컴퓨터가 그렇다고 할 수 있는데, 누군가 악의적으로 서버를 다운시키면 다른 사람들도 모두 못 쓰게 되는 경우. 대표적으로 => STOP(cpu 멈춰!) , HALT ,RESET 명령등이 있음. 중지된 cpu를 다시 깨우려면 껐다 키는 방법밖에 없음.
- 그러므로 사용자 프로그램은 STOP 등 `치명적인 명령들을 내릴 수 없게 해야한다`
    - 사용자 모드 vs 관리자 모드가 그래서 존재하는것.
    - 이것이 이중 모드의 모티브!
    - 관리자 모드는 `시스템 모드` = `모니터 모드` = `특권 모드` 라고 하기도 함.
    - Supervisor, system, monitor, priviliged mode
- 특권 명령 (privilieged instructions)
    - STOP, HALT, RESET, SET_TIMER, SET_HW

특권 명령은 관리자 모드에서만 내릴 수 있는 명령들이다. 타이머 같은 경우, 일반 유저가 막 서버 시간을 자기멋대로 바꿔버리면 안되니까?, SET_HW => 하드웨어의 값을 바꾸는 것.

그렇다면, `이중모드 (dual mode)` 는 어떻게 만드는가?

우선, cpu 안에는 `레지스터` , ALU(산술 연산장치), 제어유닛 등이 있는데,

이중 레지스터는 비트들의 모음인데, (32 비트 컴퓨터라면 비트가 32개)
이 비트를 활용해서 cpu의 상태를 나타내는 비트로 쓸 수 있다.

구체적으로, 이를 `FLAG` 라고 하는데? (어떠한 사건이 일어나면 깃발을 들고 경고한다고 생각)

- carry (자리수가 올라갔을때 씀)
- negative (연산 결과가 음수일 경우)
- zero (연산 결과가 0)
- overflow (연산 결과가 자리범위를 넘어버린 경우)

뭐 특정 비트가 1이면 캐리가 발생했다든가, 이런 식으로 활용할 수 있음. 컴퓨터 구조 시간에 이런 4가지 정도의 flag 들을 배우게 되는데,
OS 에서는 여기 예를들면 비트를 하나 더 할애해서 이중 모드를 나타내는 비트를 추가하는것.

가령, monitor 비트가 1이면 관리자 모드다 뭐 이런식으로 이중 모드를 나타 낼 수 있음.

=> 파워를 키면 부팅이 되고 OS를 주기억장치 영역으로 올리는 등 이 과정에서 모니터 비트는 1을 유지하다가, 게임 프로그램등을 실행할 때에는 OS가 유저모드로 전환시켜줌.

`정리`

- 이중모드=> 레지스터에 모드를 나타내는 플래그를 활용해 on/off 한다.
- OS 서비스 실행될 동안은 `관리자 모드`

- 사용자 프로그램이 실행될 때에는 `사용자 모드`
- 하드웨어 / 소프트웨어 인터럽트가 발생하면 `관리자 모드`
- 운영체제 서비스가 끝나면 다시 `사용자 모드`

게임 프로그램을 쓰다가, 게임 스코어를 하드디스크에 저장하고 싶다 (서버에 기록하고싶다) 라고 할 때, 하드디스크에 저장하는건 OS가 하는것. 게임 프로그램은 OS에게 게임 스코어를 저장해 달라! 라고 부탁을 하게 되고 (소프트웨어 인터럽트) 인터럽트가 걸리게 되면 cpu는 현재 하던일을 중지하고 OS 안에 들어있는 ISR로 점프해서 OS 안의 코드가 동작함.

=> 게임 프로그램이 직접 하드디스크에 접근할 수 없다는 뜻과 같은데, 만약 게임 프로그램이 직접 하드디스크에 접근 가능하게끔 해버리면, 하드 디스크 안에 있는 남의 파일도 뒤질 수 있게 된다는 문제점이 생김. (서버 컴퓨터에는 여러 사용자 파일들이 있으니까)

- 하나의 프로그램을 실행 중이더라도 모드 전환이 많이 일어날 수 있음.

만약, 일반 유저가 특권 명령을 내린다고 해보자. Cpu가 이 명령을 읽어와서 보니 monitor 비트가 꺼져있으면, cpu는 이 명령이 그래서 잘못 내려진 것으로 판단함 (내부 인터럽트가 발생했구나) => 그래서 cpu에셔 이 명령을 실행하지 않고 os의 잘못된 명령의 경우 작동하는 ISR로 점프하고 잘못된 명령을 내린 프로그램을 강제 종료해서 메모리에서 날려버림.

결국, `이중 모드는 보호(protection)와 관련이 있음`

(1) 입출력 장치 보호

- 프린터, 하드디스크 등 => 서버 컴퓨터를 경유하여 누군가 프린트 하고 있는데, 이걸 뭐 리셋시켜버린다든가 그러면 안되니까? 하드디스크에서 다른사람 파일 읽고 쓰면 안되니까?
    - 입출력 명령(IN and OUT)을 특권 명령으로 함. (OS만 이런 명령들 내릴 수 있도록)
    - 그럼 입출력을 하고 싶으면, 유저 모드에서 OS한테 소프트웨어 인터럽트 걸고, OS 안에 해당 인터럽트를 처리하는 루틴으로 간 다음 입출력 완료 후, 다시 사용자 모드로 돌아오는 식으로 하는 것.
    - 다른 사람 파일을 읽으려고 하는 상황을 생각해보면, 이걸 OS 한테 부탁해봤자 OS쪽에서 거부할 수 있게 됨. 곧, OS에게 소프트웨어 인터럽트 걸어봤자 OS의 ISR 에서 정당한 요청인가 확인하는 부분이 있기 때문임. (네이버 N드라이브나 이런데 올려도 다른 유저들이 내걸 뒤져볼 수 없는것)

(2) 메모리 보호

OS 가 상주하긴 하는데, 나머지 MM 영역에 다양한 사용자 프로그램이 올라와 있을텐데?

유저 프로그램 1이 다른 유저 프로그램 2를 읽으려고 하거나, 유저 프로그램 1이 OS의 ISR 코드를 바꿀 수 있거나(이게 해킹) 뭐 이러면 안되니까?
유저 1 프로그램이 돌때는 메모리 안에서 가동범위 영역이 유저 1 프로그램으로 한정되어야 한다는 의미.

그럼 이렇게 한정시키는건 어떻게 하나?

cpu 에서 메모리로 address bus가 가는데 (메모리의 어느 주소를 읽어오겠다) => 그 번지수에 해당하는 내용이 data bus로 오게 되는 구조이다. 애초에 address bus를 잘라버리는 방법이 있을 수 있는데, 애초에 address bus를 잘라버리면 자기 영역에도 못들어 간다는 단점이 생김.

그래서 address bus에 문지기 하나 둬서, 문지기가 유저 1에 할당된 주소로만 bus 보내도록 하는것.

`문지기의 정체` => 레지스터(극히 소량의 데이터나 처리중인 중간 결과를 일시적으로 기억해 두는 고속의 전용 영역)를 둬서 base, limit 설정하고 base ~ limit 사에 있는 주소를 요청할 때에만 통과 시켜주도록 함. 자기 범위를 넘어서는 번지를 읽으려고 하면 문지기가 cpu에게 인터럽트 신호 보내줌. 그럼 cpu에서 OS ISR로 가고, 잘못된 번지를 읽으려고 하는 애를 강제 종료시킴.

다른 사용자 또는 운영체제 영역 메모리에 접근 시도하는 것을 `Segment Violation` 이라고 함.

- MMU (Memory Management Unit = 문지기 이름) 을 둬서, 다른 메모리 영역에 대한 침범을 감시하도록 한다.
- MMU 설정(base limit 값을 바꾸거나)은 특권명령이다! => OS만 바꿀 수 있다.

(3) CPU 보호

- CPU 자체에 대한 공격이 들어올 수도 있으니까!

운영체제 서비스

운영체제 역할

하드웨어 보호

입출력 장치 보호

- 입출력 장치 보호(한 사람의 잘못으로 프린터, 키보드에 악영향을 미쳐서 타인이 피해를 보지 않게)

  how? '특권명령' (입출력에 직접적으로 명령을 못내리고 os를 통해야만 가능하게)

메모리 보호

- 하나의 os에 사용자 각각의 메모리가 독자적으로 존재하나 한 명이 딴 사람거까지 독식하려고 하면??

  how? MMU라는 하드웨어 부품이 CPU가 메모리에 접근하는 것을 제어하며 관리

  MMU란 뭔가요? CPU의 Memory 주소를 속이는 거짓말쟁이(주소를 관리해주는 친구인데 잘은 모르겟습니다 ㅠ)

CPU 보호

- 한 사용자가 CPU시간독점 (While n == 1:)
- 한 사용자만 CPU시간을 독점하고 다른 사람들은 사용하지 못한다는 문제점 발생

  How? Timer를 만들어서 일정 시간 이상이면 인터럽트(중지)되고 다른 프로그램으로 강제 전환

효율적인 관리를 통한 자원 사용

컴퓨터에는 H/W 라는 Resource(자원)이 존재 -> cpu, memory, print , 하드디스크

위의 자원들을 application(game, db, hwp)들이 사용함

효율적으로 사용하기 위해선 OS가 필요한 것(정부기관과 같이 OS도 기관별로 존재)

1. CPU자원을 관리: Process management(프로세스 관리)

   프로세스란? Main memory에서 실행중인 프로그램을 일컫는 말

   즉 Process Management에서는 프로세스와 관련된 작업 담당
2. 주기억장치: main memory management(Application에 메모리 분배)

   Main memory란? 프로그램이 실행되기 위한 공간

   프로세스에게 메모리 공간 할당 및 회수

   즉 메모리를 관리하는 역할을 담당하며 프로세스에게 메모리를 허락해주는 작업 담당
3. 파일관리: File management(하드디스크 내부의 파일 관리)

   디스크는 Track/Sector로 구성되어 있는데 파일이라는 관점으로 os에서 변환해줌

   즉 우리가 파일 및 디렉토리 관리하는게 사실 os에서 변환해서 보여주는 것이었음
4. 보조기억장치관리(secondary storage management)

   대표적으로 하드 디스크, 플래시 메모리로 아무것도 없는 공간 관리

   즉 아무것도 없는(block)들을 관리해주는 역할

5. 입출력장치관리(I/O Device management)

   대표적인 기능으로 Device drivers와 입출력장치 성능향상으로 buffering, caching, spooling등이 있음

6. 시스템 콜(System call)

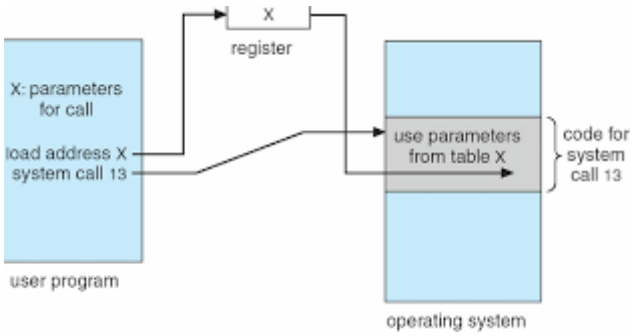
   유저 프로세스에서 운영체제 서비스를 필요할 때 받기 위한 호출

   ex) 임의의 프로세스가 타 프로세스의 함수를 호출(사용)할 수 있을까? == hwp가 excel의 sum함수를 사용할 수 있을까

   기본적으로는 불가능함 why? 운영체제는 프로세스가 각자의 영역에서만 제한을 두기 때문에

   But 어떤 프로세스가 OS의 함수를 사용하고 싶은 상황에서 사용하는게 System call

   실제사용법 참조링크: https://www.youtube.com/watch?v=PsXXjNL_ogc&t=130s

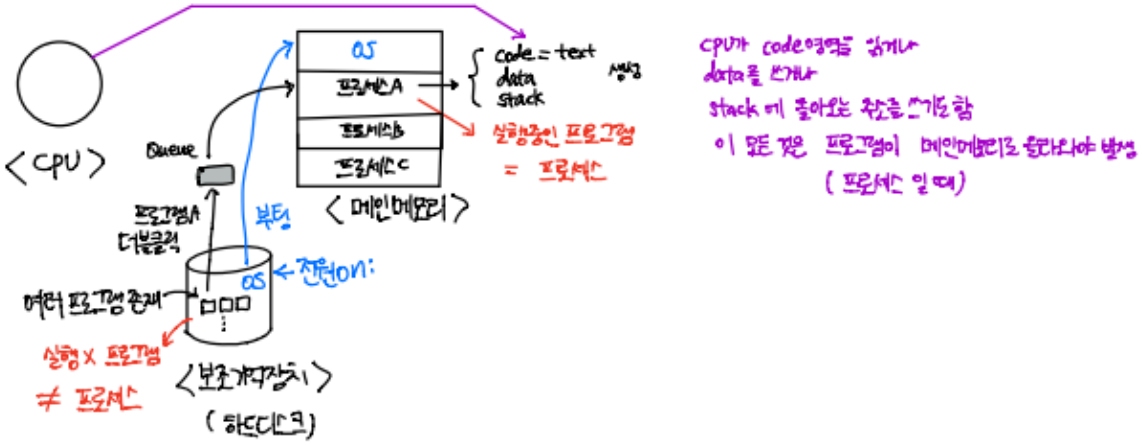


OS_03_프로세스 관리

프로세스 관리 Process Management

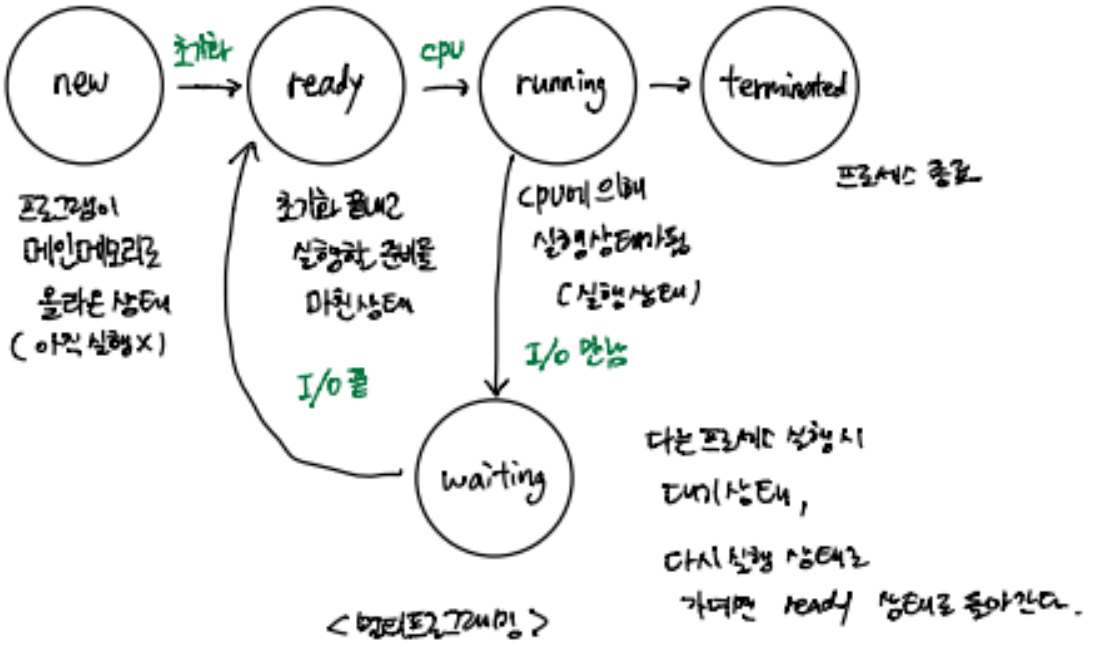
> CPU 자원을 어떻게 효과적으로 각 프로세스에게 나누어 줄 것인지
>
> 프로세스 관리 = CPU 관리

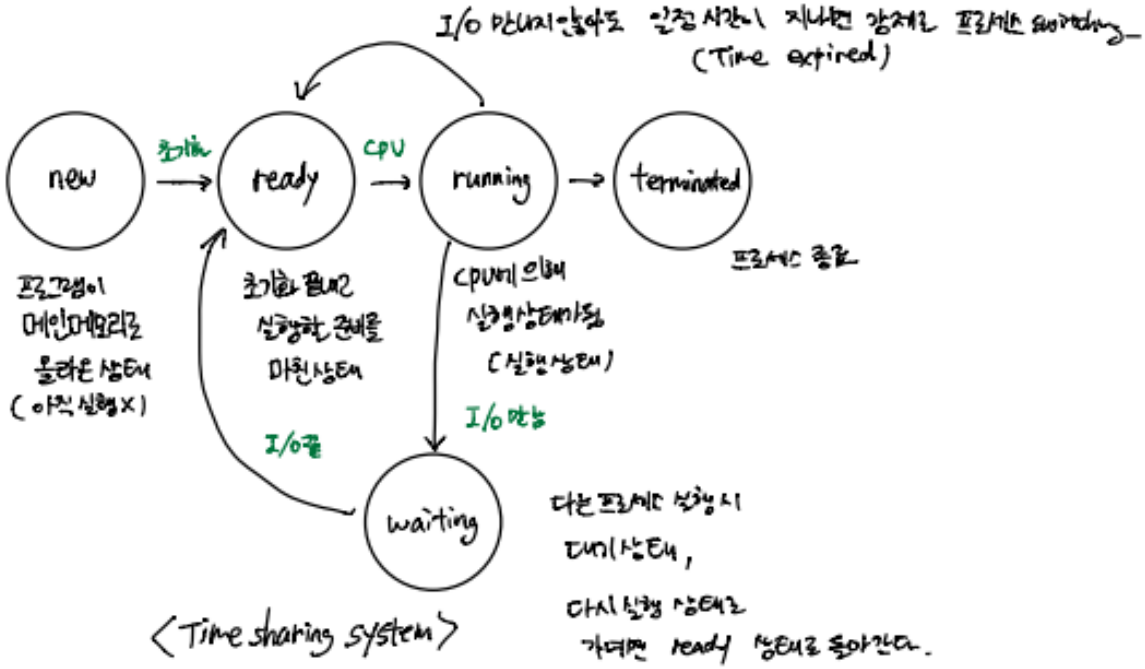
프로그램 VS 프로세스



- **프로세스(process, task, job...)**: program in execution => **실행 중인 프로그램**을 프로세스라고 함 = **메모리 위에 있는 프로그램**
- 프로그램: 하드디스크 안에 있는 프로그램
- 프로그램이 하드디스크에 있을 땐 아무 일도 발생 X, 프로그램 더블 클릭 시 메인 메모리로 올라감 (실행)
- 프로그램이 실행(프로세스)되면 내부적으로 pc(program counter), sp(stack pointer), register 등이 바뀐다
  - program counter: 명령어 주소 레지스터 (몇 번지를 실행할 것인지, 다음 실행할 명령어 주소 저장)
  - stack pointer: 스택 프레임의 최상단 주소를 가리키는 레지스터
  - register: 고속 메모리, 임시 저장장치
- 프로세스 구조
  - code(text): 컴파일된 소스 코드가 저장되는 영역
  - data: 전역 변수/초기화 된 데이터가 저장되는 영역
  - stack: 임시 데이터(함수 호출, 로컬 변수 등)가 저장되는 영역
  - heap: 코드에서 동적으로 생성되는 데이터가 저장되는 영역
  - 스택 프레임: 함수를 실행하기 위해 스택을 이용하여 만든 구조
- 다중 프로그래밍: CPU가 여러 프로세스를 관리 => CPU 할당하는 시간을 관리 (Time sharing system)

프로세스 상태

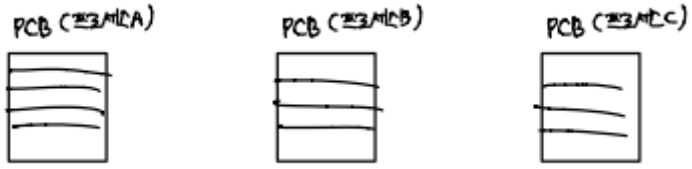




- **new**: 프로그램이 메인 메모리로 올라온 상태 (아직 실행 X)
- **ready**: 초기화 끝내고 실행할 준비를 마친 상태
- **running**: CPU에 의한 실행 상태
- **waiting**: 다른 프로세스 실행 시 대기 상태, 다시 실행 상태로 돌아갈 때 ready 상태로 돌아간다
- **terminated**: 프로세스 종료
- TSS(Time Sharing System)에서는 I/O 만나지 않아도 일정 시간이 지나면 강제로 프로세스 switching 발생(Time expired) => ready 상태로 돌아감

PCB (Process Control Block)

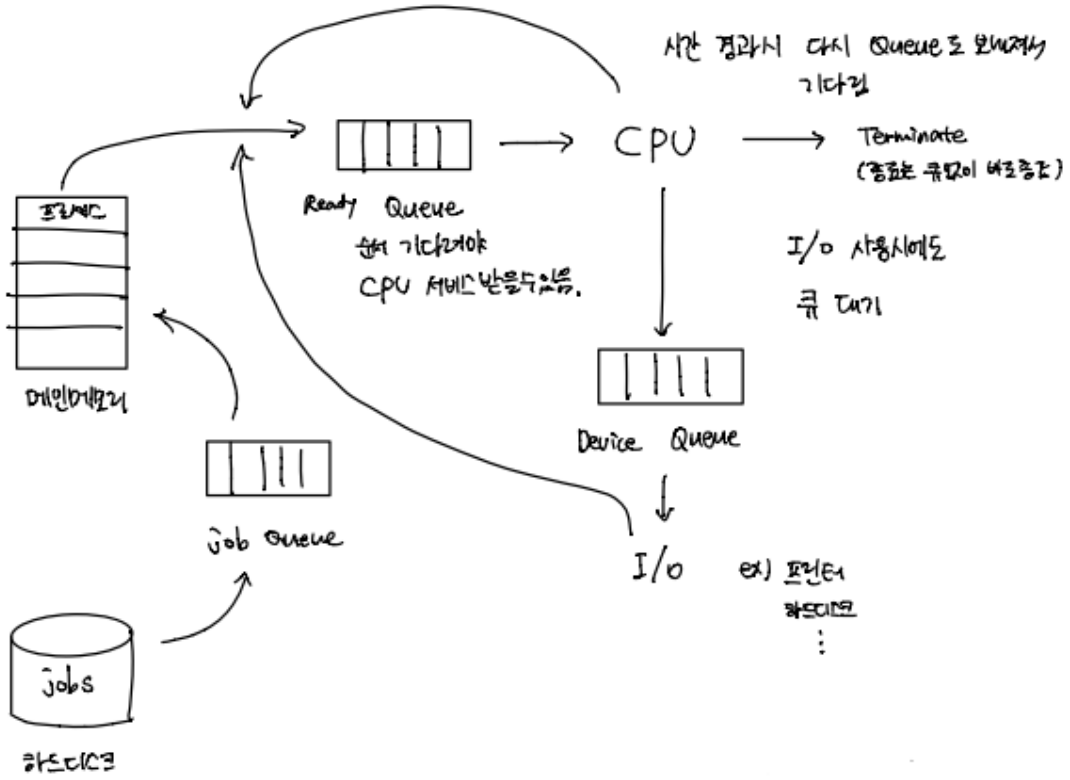
> = task control Block (TCB)



- 하나의 프로세스에 대해 하나의 PCB 할당
- 프로세스에 대한 모든 정보가 들어있다
  - PID: Process ID
  - 프로세스의 상태 정보: new, ready, running 등
  - pc(process counter): 해당 프로세스가 다음에 실행할 명령어 주소
  - MMU(Mermory Management Unit) info: base, limit 정보
    - base: 프로세스의 시작 주소
    - limit: 프로세스의 끝 주소
  - CPU Time: CPU 사용시간 => 많이 쓰면 다음 스위칭 때 적게 사용하게끔 조절하기도 한다
    - 예시) 슈퍼 컴퓨터 사용 시 CPU Time에 따라 비용을 지불한다
  - list of open files: 어떤 파일들을 사용하고 있는 지
- 이러한 정보를 담아두는 이유
  - CPU가 여러 프로세스를 왔다 갔다 스위칭하는데 다른 프로세스로 넘어가기 전 정보를 담아서 다시 돌아왔을 때 **넘어가기 직전 상태를 유지하기 위해서**
- PCB는 OS의 Process Manager에 존재

Queue

> 하드디스크에는 많은 프로그램 존재 but 메인 메모리 공간은 한정적
>
> 처리를 위한 대기줄



Queue 종류

- Job Queue: 하드디스크에서 메인 메모리로 올릴 때의 대기줄
- Ready Queue: 메인 메모리에서 CPU 서비스를 받기 위한 대기줄
- Device Queue: 프린터, 하드디스크와 같은 I/O Device를 사용하기 위한 대기줄

스케줄러 Scheduler

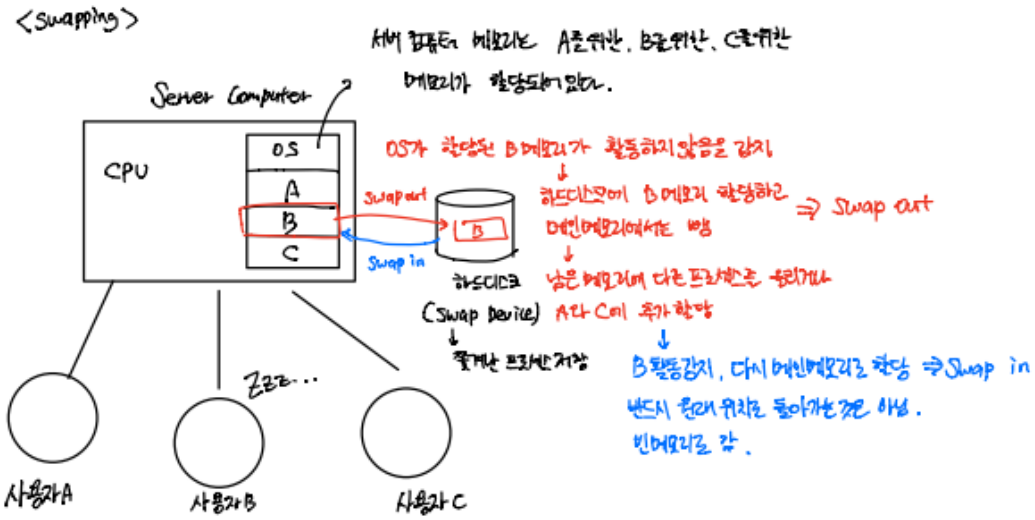
> 큐에는 프로세스들이 줄서서 대기하고 있음
>
> 스케줄러는 큐에서 어떤 프로세스를 먼저 처리할지 결정하는 것
>
> 반드시 큐 순서대로 처리 하는 것은 아니다
>
> OS의 Process Manager에 존재

- Job Scheduler
  - 하드디스크의 작업(job queue의 jobs) 중 어느 것을 먼저 메인 메모리로 올릴지 결정하는 프로그램
  - Long-term scheduler
    - 메인 메모리 가득 차면 스케줄링은 발생하지 않음
    - 한 프로세스가 끝나 메모리가 비워져 자리가 생기면 스케줄링 발생 => 자주 일어 나지 않음
- **CPU Scheduler**
  - Ready Queue의 프로세스 중 어떤 것을 먼저 서비스할지 결정
  - Short-term scheduler
    - 프로세스 간 스위칭이 빠르게 돌아야 동시에 실행하는 것처럼 보임

        - 1초에 수십번 이상 발생
- Device Scheduler
    - Device 큐 중에 어떤 프로세스를 먼저 서비스할지 결정

멀티 프로그래밍 Multiprogramming

> 메인 메모리에 여러 프로세스를 올리는 것

- Degree of Multiprogramming: 메인 메모리에 프로세스 몇 개 올라가 있는지 정도
- 프로세스 종류
    - I/O-bound process: 주로 I/O 관련 작업하는 프로세스 (워드, ppt 등 워드프로세스)
    - CPU-bound process: 주로 CPU 사용하는 프로세스 => 계산량 많음 (날씨예보 프로세스)
- CPU와 I/O 중 한 쪽이 놀지 않도록 job 스케줄러가 메인 메모리에 올릴 때 적절히 밸런스 맞춰서 올린다
- **Medium-term scheduler**: 메모리에 있는 프로세스 중 사용되고 있지 않은 어떤 프로세스를 swap device로 보낼 지 결정
    - 대화형 시스템(Time Sharing System)의 swapping
    - Swapping은 Long-term 스케줄링보다는 자주 Short-term 스케줄링보다는 덜 발생한다
    - Swap out: 메인 메모리에서 Swap Device로 쫓아냄
    - Swap in: Swap Device에서 메인 메모리로 복귀
    - 예시
    - 

        - 사용자 B가 휴식 중이면 B 메모리 사용 X
        - OS가 메모리에 B가 할당되어 있지만 활동하지 않음 감지
        - 하드디스크로 B 메모리 할당 후 메인 메모리에서 제거 (Swap out)
        - 남은 메모리 공간에 다른 프로세스를 올리거나 A 나 C에 추가 할당
        - B가 다시 활동하면 메인 메모리로 다시 올림 (Swap in)
        - swap in 할 때 반드시 원래 위치로 돌아가는 것은 아니다 빈 메모리에 위치

OS - 4. CPU 스케줄링 알고리즘(1)

CPU 스케줄링

- 현재 CPU에서 실행중인 프로세스가 끝나고 나면 어느 프로세스를 실행시킬 지 결정하는 것

🐰 Preemptive vs Non-preemptive

🥕 Preemptive(선점) : 미리 점유

--> CPU가 어떤 프로세스를 실행하고 있을 때 아직 CPU가 끝나지도 않았고, IO를 만난 것도 아닌데 강제로 쫓아내고 새로운 프로세스가 들어갈 수 있도록 허용하는 스케줄링

--> 예) 병원에서 환자가 진료받고 있는데 응급 환자가 와서 진료 중이던 환자를 내보내고 응급환자를 먼저 받도록 허용하는 것

🥕 Non-preemptive(비선점) : 미리 점유하지 않음

--> 이미 CPU가 실행중이면, CPU가 끝나거나 IO를 만나기 전까지는 절대 스케줄링이 일어나지 않는다.

--> 예) 병원에서 환자가 진료를 받고 있는 동안에는 중간에 환자를 내보내고 다른 사람이 진료를 받을 수 없다.

🐰 Scheduling criteria(스케줄링 비교척도 --> 이 스케줄링으로 하니 이런 점이 좋고, 저 스케줄링으로 하니 저런 점이 좋다는 걸 비교하는 척도!)

- CPU Utilization(CPU 이용률) : CPU를 얼마나 활용해 작업을 빨리 끝내는가? (높을수록!👍)
- Throughput(처리율) : 주어진 시간에 몇 개의 작업이 끝났는가? (높을수록!👍)
- Turnaround time(반환시간) : 작업을 시작한 순간부터 다 끝내고 나오는 시간 --> 어떤 작업이 Ready Queue에서 기다리다가 CPU에서 실행 후 다시 Ready Queue로 돌아가고 다시 반복... 하다가 어느 순간 완전히 작업이 끝났을 때! (짧을수록!👍)
- Waiting time(대기시간) : CPU서비스를 받기 위해서 Ready Queue에서 얼마나 기다렸는가? --> 단위는 sec (짧을수록!👍)
- Response time(응답시간) : 명령을 내리면 처음 응답이 나올 때까지 걸리는 시간(ex) 사이트 로딩 시간) (짧을수록!👍)

--> 병원으로 예를 들자면..

- CPU 이용률 : 병원장 입장에서 의사가 돈을 많이 받으니 받은 만큼 진료를 많이 하기를 원한다.
- 처리율 : 하루에 몇 명의 환자를 받는가?
- 반환시간 : 환자가 병원에 발을 들이는 순간부터 진료를 끝내고 나오는 시간
- 대기시간 : 의사를 만나기까지 환자가 대기하는 시간

CPU Scheduling Algorithms

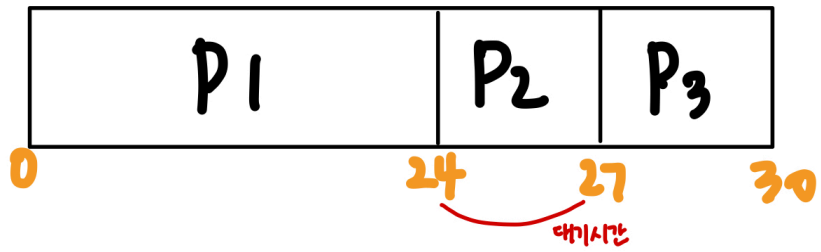
- First-Come, First-Served(FCFS) : 먼저 온 것을 먼저 서비스한다.
- Shortest-Job-First(SJF) : 작업시간이 짧은 것을 먼저 서비스한다.
- Priority : 우선순위가 높은 것을 먼저 서비스한다.
- Round-Robin(RR) : 빙빙 돌면서 순서대로 서비스한다.
- Multilevel Queue : Queue를 여러 개 둔다.
- Multilevel Feedback Queue : Queue를 여러 개 둔다.

First-Come, First-Served

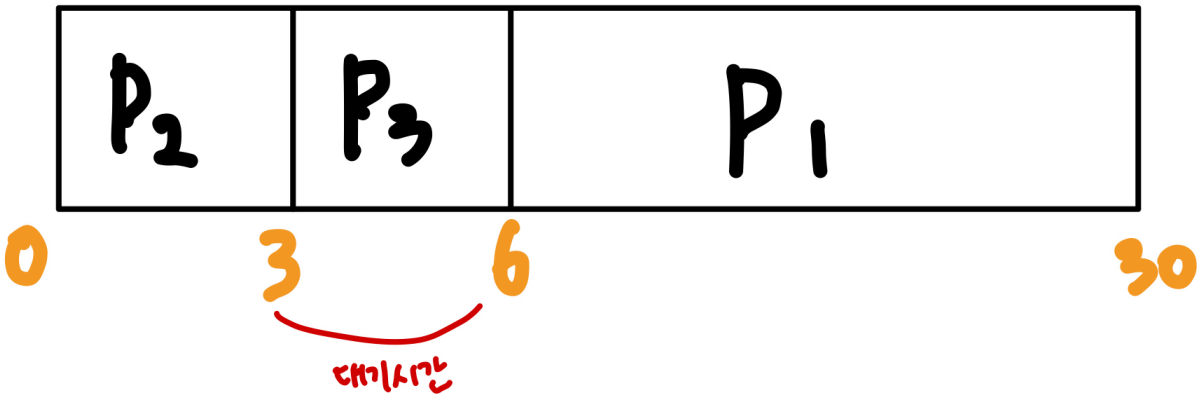
- 병원 & 은행 등 보통 이러한 방식이다.
- Simple & Fair : 가장 간단하고 일반적으로 가장 공정한 방법 --> 그러나 이 방법이 가장 좋다고는 할 수 없다.
- 예) Find Average Waiting Time - 대기 시간의 평균을 통해 효율적인지를 확인해볼 수 있다.
    - AWT = (0+24+27)/3 = 17 msec
- Gantt Chart : 아래 그림과 같이 표현한 것(Gantt라는 사람의 이름을 따왔다)

Process	Burst Time (msec) - 사용하는 시간
P1	24
P2	3
P3	3

--> 예를 들자면, 은행에 P1, P2, P3 동시에 도착했는데 번호표를 P1-P2-P3 순으로 뽑게되었다.



--> 만약 순서가 P2-P3-P1이라면?



대기시간은 0msec, 3msec, 6msec가 된다. AWT를 계산하면, (0+3+6) / 3 = 3msec가 된다.

- Convoy Effect (호위효과) : 뒤에 붙어서 호위하는 것 --> P1 같이 시간이 오래 걸리는 것이 앞에 있으면, 시간이 짧게 걸리는 것들이 뒤에서 계속 기다리면서 마치 호위하듯이 따라다니는 것

- Nonpreemptive scheduling : 비선점 스케줄링이다.

❗ 들어온 순서대로 진행하면 대기 시간 면에서 별로 좋지 않다는 것을 알 수 있다.

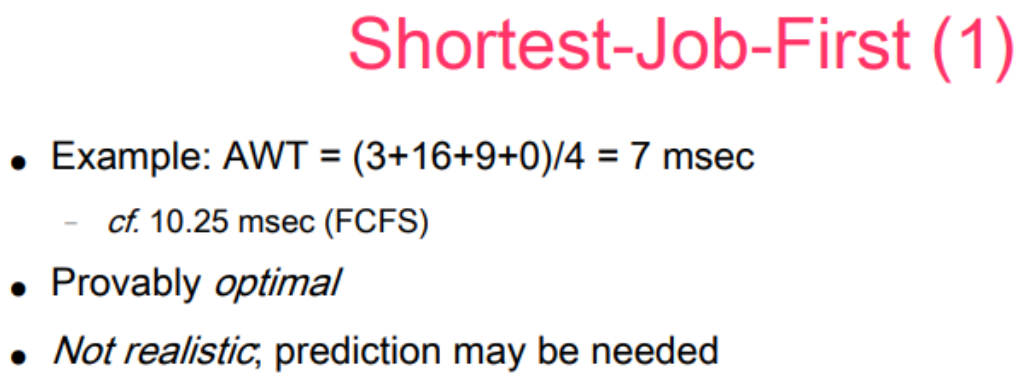
CPU 스케쥴링 알고리즘 (2)

> SJF, Priority, RR 스케쥴링

SJF (Shortest Job First)

`SJF` 는 말 그대로 가장 시간이 적게 걸릴 애를 잡아가지고 걔먼저 처리하는 것이다.

간트 차트를 한번 보자,



Process	Burst Time (msec)
P_1	6
P_2	8
P_3	7
P_4	3

보니까 P_4 가 가장 짧은데? 짧은 순으로 넣게 되면 4 => 1 => 3 => 2 로 넣게 되고,
AWT(평균 대기시간)는 7이 된다. (P_2 입장에서는 자기가 제일 기니까 앞의 3 7 6 다 기다려야해서 16이 됨.)

`Provably Optimal` ? : 증명되어진 최적이다.

다른 모든 예제를 가져 와도 `대기 시간` 의 측면에서는 FCFS보다 항상 나음.

그럼 근데 이 방법의 문제점은 뭐냐? = CPU 시간 어떤 프로세스가 얼마나 사용할지 어떻게 아냐는 문제가 있음. == `얼마나 쓸지 예측할 수 밖에 없는데 이건 비현실적임`

- 예측을 그나마 할 수는 있는데, 어떤 방식이냐면,
- 프로세스는 죽기(Terminate)전까지 큐에 갔다가, CPU 서비스 받다가, 다시 Device Queue 갔다가 다시 Ready Queue 왔다가 하는 방식으로 빙글빙글 돌다가 종료되는데?
- 반복할때마다 이번엔 CPU 시간을 얼마 쓰는가 뭐 이런식으로 일일히 기록해 뒀다가 OS가 조사한 걸 바탕으로 합리적 예측을 할 수는 있겠다만, 그러려면 뭔가 일일히 기록하고 저장해야 한다는 측면에서 Overhead(부담) 발생.

> SJF는 Preemptive / Nonpreemptive 두가지 방식으로 만들어 볼 수 있음.

Shortest-Job-First (2)

- Preemptive or Nonpreemtive
  - *cf.* Shortest-Remaining-Time-First (최소잔여시간 우선)
- Example
  - Preemptive: AWT = (9+0+15+2)/4 = 26/4 = 6.5 msec
  - Nonpreemptive: 7.75 msec

Process	Arrival Time	Burst Time (msec)
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

1 Nonpreemptive 방식은 현재 프로세스가 다 끝나고 난 다음에야 다음게 실행 되는거.

간트 차트를 보면, 위의 차트랑 다른건 위의 차트에서는 (SJF의 경우)그냥 도착시간이 똑같은 상황을 상정한거고, 여기 차트에서는 도착 시간이 다름.

우선, P_1 은 큐에 들어와보니 지밖에 없음(물론 나중에 미래 상황까지 상정해 보면 Burst Time이 더 적은 프로세스들이 존재하기는 하지만?) 그래서 실행 바로 됨.

P_1이 실행되고 1미리초가 지나면 P_2 가 들어오고, 뭐 이런식으로 진행될거임.

애초에 근데 `P₁ 이 8미리초나 하기 때문에` 얘가 한 중간쯤 실행될 시점엔 나머지 애들도 다 큐에 들어와서 기다리고 있음. 그러면 이제 큐에 P2 , P3, P4 가 들어와 있기 때문에 얘들 중 Burst Time이 짧은 애를 선택해가지고 그다음을 실행하고 이런 식임.

그러면 AWT = 7.75 라는건 어떻게 나왔냐? 우선 P1 -> P2 -> P4 -> P3 순으로 끝날거.

(1) 기본적으로 P1 이 젤 먼저 실행돼서 얜 기다린 시간이 없으니까 0 이고,

(2) P2 는 그다음 간택될건데(2 3 4 중에 제일 짧아서) 들어온 시간이 1이고, 앞에 실행이 되던 P1 이 다 끝나야만(Nonpreemptive) P2가 실행될거니까 7초 후 실행,

(3) P4 는 17-2 = 15

(4) P3 는 12-3 = 9

그래서 0 + 7 + 15 + 9 / 4 = 7.75

2 Preemptive 방식은 응급실 같은건데, 지금 진료중 환자(현재 실행중 프로세스)가 있어도 더 응급한 애가 있으면 지금 하고있는애보다 우선해서 진료(프로세스 처리) 가능한 거.

==> `Shortest Remaining Time First` 라고 부르기도 함.

이 방식은 A 프로세스가 실행되고 있는 상황에서 B 프로세스가 큐에 들어왔다면,
A 프로세스의 잔여 실행 시간과 B 프로세스의 전체적인 Burst TIme과 비교해 더 짧은애를 일단 실행시키는 방식으로 이루어짐.

대기실에 박혀있는 시간 다 합해보면 => 6.5 가 나온다는 뜻.

Priority

우선순위 : 누가 더 우선인 것인가?

일반적으로 우선순위를 정수값으로 나타냄. (숫자가 작으면 우선순위가 높다)

Priority Scheduling (1)

- Priority (우선순위): typically an integer number
  - Low number represents high priority in general (Unix/Linux)
- Example
  - AWT = 8.2 msec

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

간트 차트를 보면, Time 0 에 5개가 줄서서 기다리고 있는데? 그냥 우선순위대로 넣음.

P2 가 가장 높은데? 얘부터 들어감.

== 6 + 0 + 16 + 18 + 1 / 5 = 8.2

Priority Scheduling (2)

- Priority
  - Internal: time limit, memory requirement, i/o to CPU burst, …
  - External: amount of funds being paid, political factors, …
- Preemptive or Nonpreemptive
- Problem
  - Indefinite blocking: *starvation* (기아)
  - Solution: againg

- 우선순위는 그럼 어떻게 정하나? 내부적 요소 vs 외부적 요소
  - 내부적? : 타임 리미트 (짧은애를 먼저 해라)
  - memory requirement (메모리 적게 차지하는애 먼저 해라)..etc

    - 외부적? : 어느 학과가 서비스 이용하는데 돈 많이 내냐..? 뭐 이런 외부적 요인들이 기준이 될 수 있음.
- 기아? 굶어 죽는건데.. 어떤 우선순위 낮은 애는? 이 5개 말고도 막 새로 굴러들어오는 애가 기다리는애보다 우선순위 높으면 얜 영영 기다려야함.
- 에이징 : 오래 기다리면 우선순위 좀 올려주는 방식으로 해결 가능.

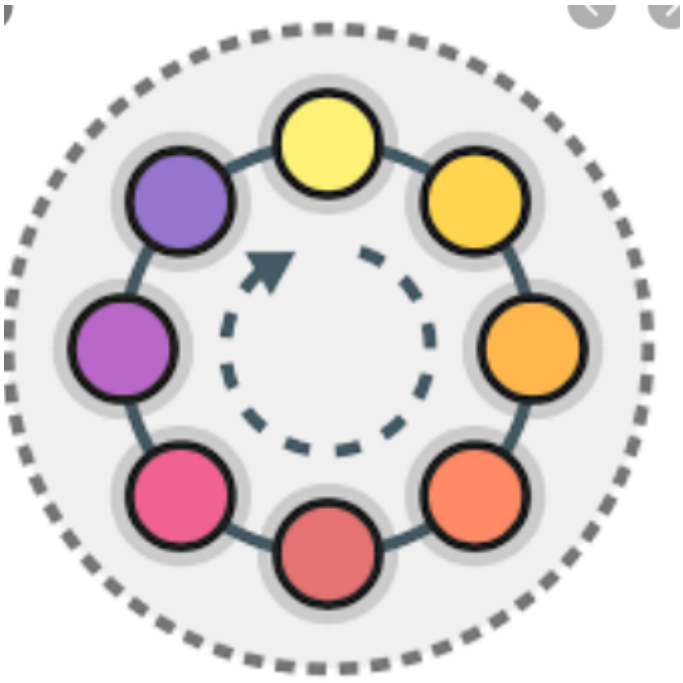
Round Robin

Round-Robin (1)

- Time-sharing system (시분할/시공유 시스템)
- Time *quantum* 시간양자 = time *slice* (10 ~ 100msec)
- Preemptive scheduling
- Example
  - Time Quantum = 4msec
  - AWT = 17/3 = 5.66 msec

Process	Burst Time (msec)
P_1	24
P_2	3
P_3	3

> 뱅글 뱅글 돌면서 스케쥴링



Time Sharing System 에서 많이 사용되는데,

P1 P2 P3 / P1 P2 P3 / P1 P2 P3 => 이런 식으로 하니까 당연히 preemptive 가 됨.

타임 퀀텀을 두고 (4 밀리초라고 할때?) => 걍 스위치 시간이 4 밀리초라고 생각.

P1 이 실행되고, 4밀리초 지나면 스위칭 될건데, 4밀리초 꼭 다 쓸필요는 없고 남은 Burst Time 이 끝나면 스위칭 됨.

뭐 타임퀀텀 (델타)를 짧게 바꾸면 AWT같은 척도가 달라지게 될 것.

=> 그래서 `타임 퀀텀`에 굉장히 의존적이다!!

- 만약 타임 퀀텀을 무한대로 둔다면??? => 스위칭이 안됨 프로세스 끝날때까지.

=> 그래서 그냥 선착순 FCFS랑 똑같게 됨.

- 타임 퀀텀을 0으로 두면? 프로세스 쉐어링 이라고 하는데 스위칭이 하도 빈번해서 3개가 한꺼번에 도는 것처럼 보이게 됨. 0으로 수렴시키면 근데 context switching overhead 발생!!
- 너무 빈번하게 스위치 하면 dispatcher가 PCB 값 바꾸고 하는 역할을 하는데 이게 너무 자주 발생해서 부담이 증가하게 된다는 뜻.

아래 예제는 타임 퀀텀에 따라 성능이 달라지는다는 것을 계산해 둔 표.

Round-Robin (2)

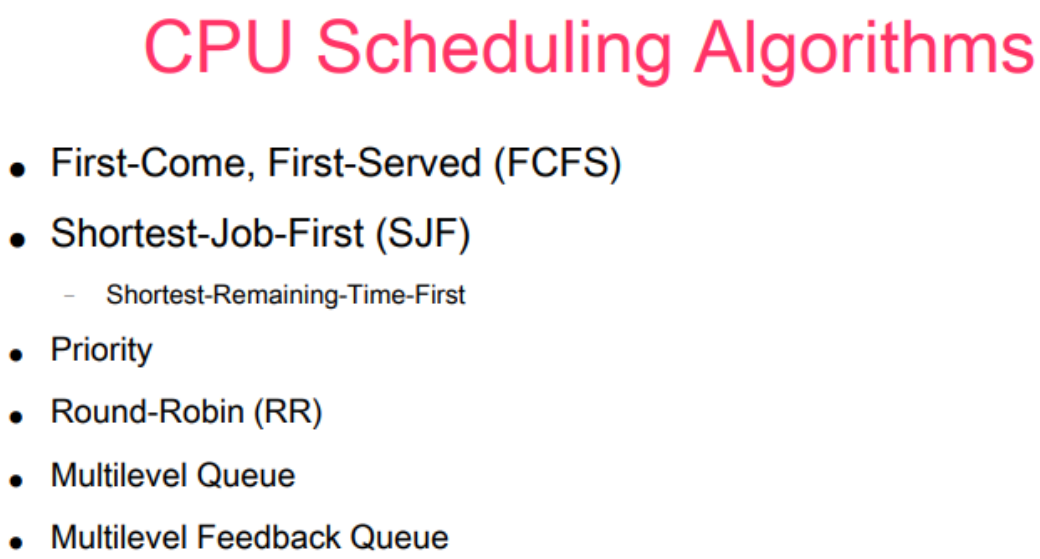
- Performance depends on the size of the time quantum
  - $\Delta \to \infty$ FCFS
  - $\Delta \to 0$ Processor sharing (* context switching overhead)
- Example: *Average turnaround time (ATT)*
  - ATT = 11.0 msec (Δ = 1), 12.25 msec (Δ = 5)

Process	Burst Time (msec)
P_1	0
P_2	3
P_3	1
P_4	7

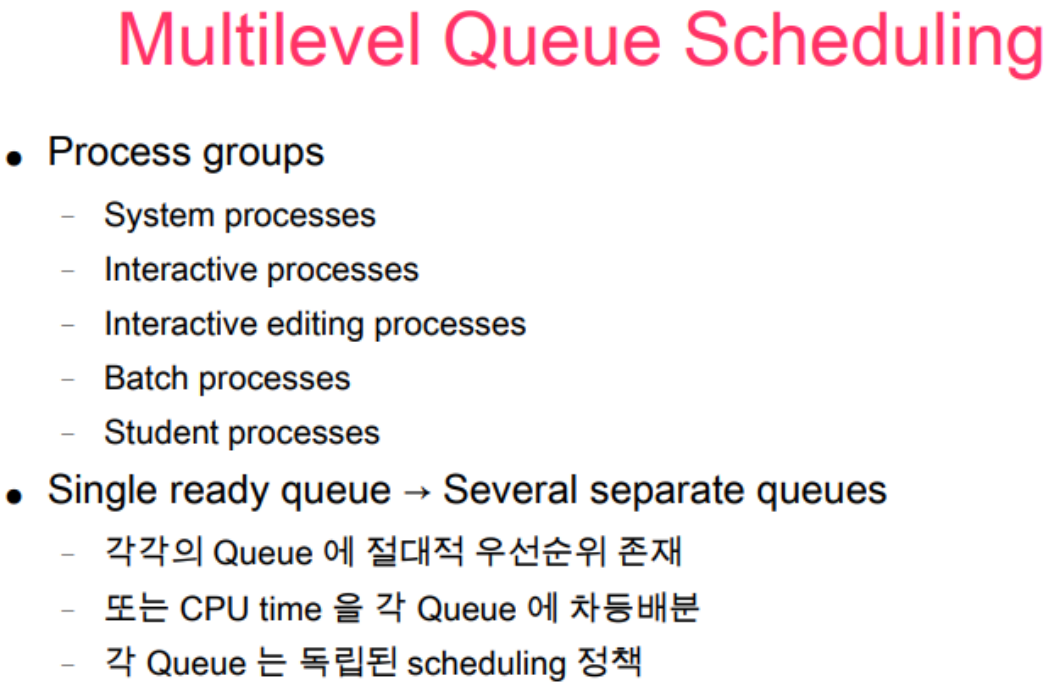
CPU 스케쥴링 알고리즘(3)

> 경성대학교 양희재 교수님의 강의 자료를 정리한 것입니다.

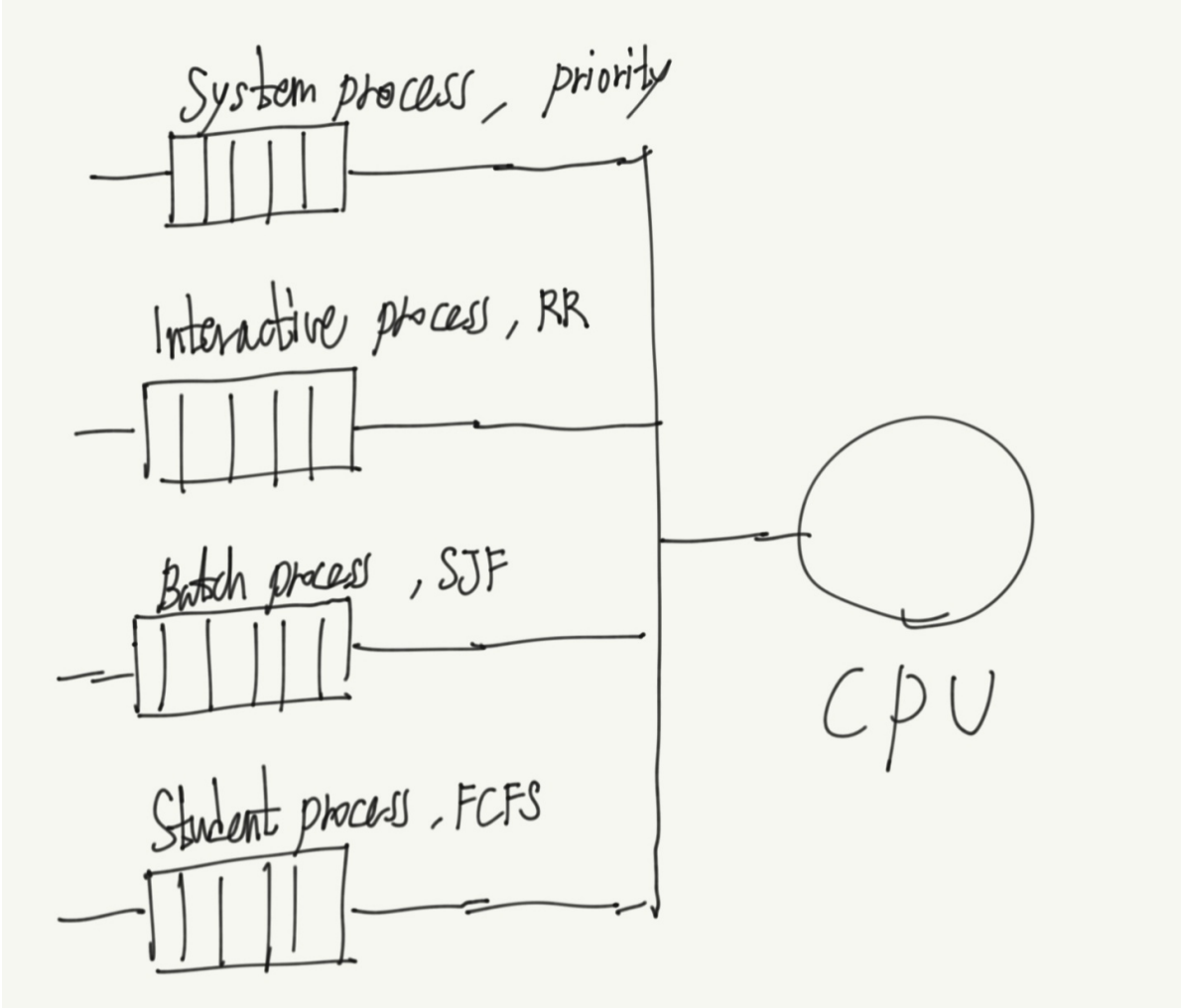
CPU Scheduling Algorithms



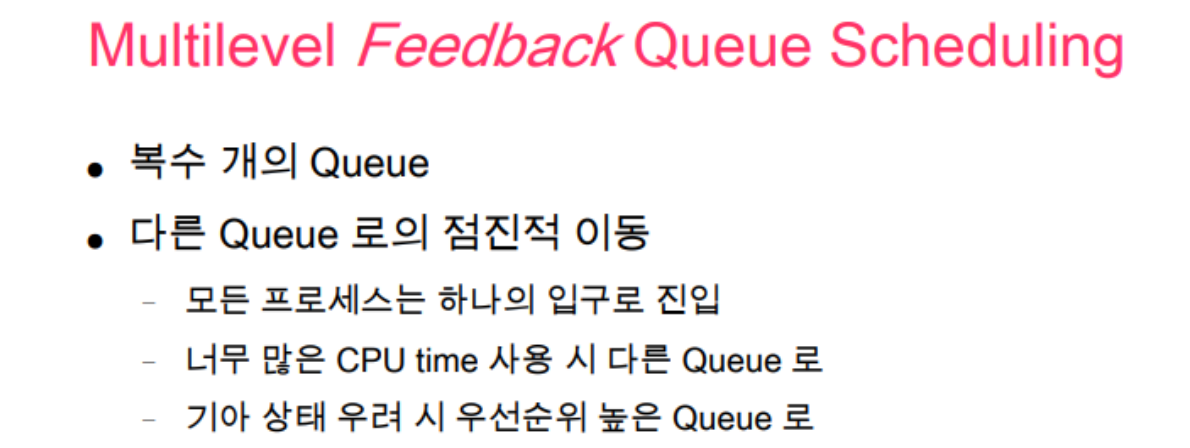
Multilevel Queue



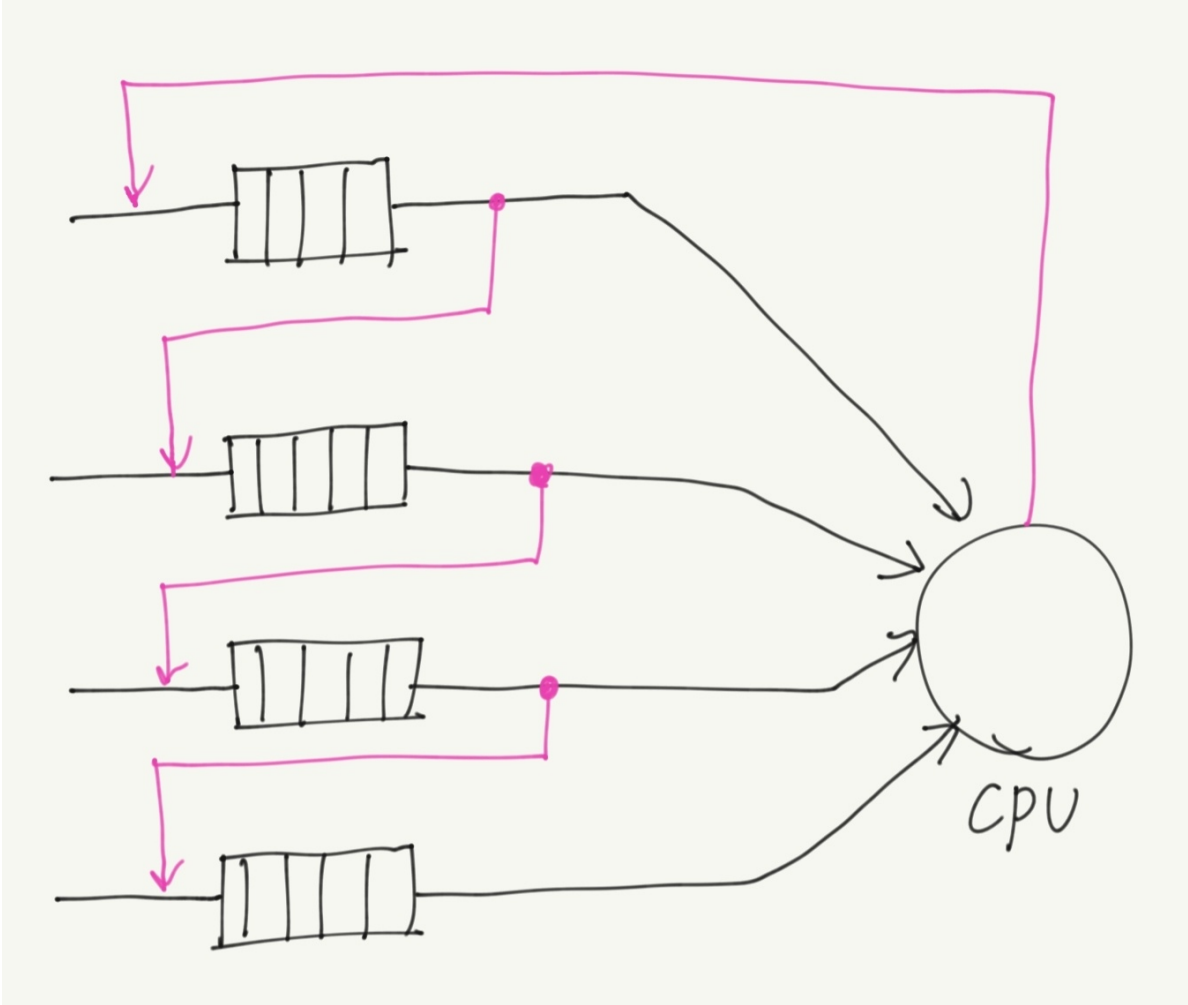
- System processes : OS안에서 OS 나름의 작업을 하는 것 (가상메모리 읽기, 파일 매핑 등)
- Interactive processes : 사용자와 대화하는 것 / 마우스,키보드 등을 이용 (게임 등)
- Interactive edting processes : Interactive processes 중 대표적인 것으로 편집하는 것 (워드프로세서 등)
- Batch processes : 대화형X, 꾸러미를 컴퓨터가 일괄처리, interaction을 안하므로 좀 느리게 처리되어도 됨
- Student processes : 학생 정보 처리



Multilevel Feedback Queue



앞의 정책의 프로세스로 실행하다 되지 않으면 다음 정책의 프로세스로도 실행해 봄



실제 계산이 필요한 앞의 4가지 프로세스가 더 중요함!

프로세스 생성과 종료

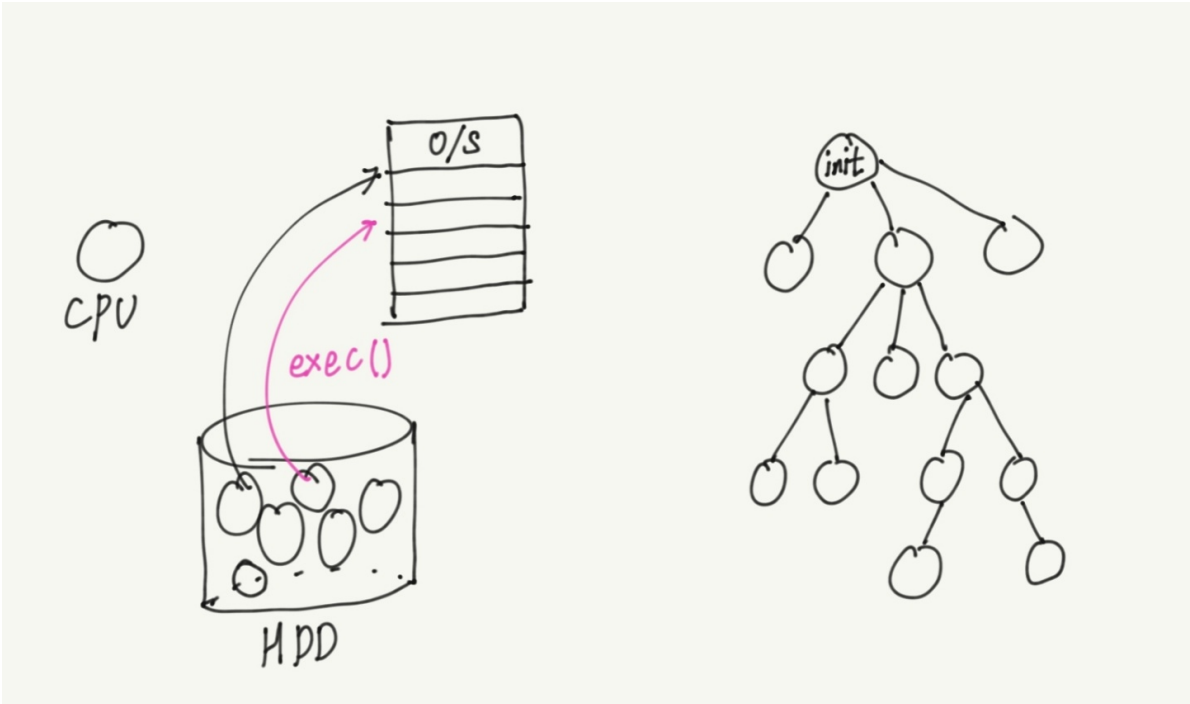
Process Creation

Process Creation

- 프로세스는 프로세스에 의해 만들어진다!
  - 부모 프로세스 (Parent process)
  - 자식 프로세스 (Child process)
    - *cf.* Sibling processes
  - 프로세스 트리 (process tree)
- Process Identifier (PID)
  - Typically an integer number
  - *cf.* PPID
- 프로세스 생성
  - *fork()* system call – 부모 프로세스 복사
  - exec() – 실행파일을 메모리로 가져오기

1. HDD 에 여러 프로그램과 OS 존재
2. 가장 처음 올라간 OS가 첫번째 프로세스를 만듬 (Unix의 경우 init이라는 프로세스 생성)
3. 프로세스가 자식 프로세스들을 생성

> Sibling porocesses : 부모가 같은 프로세스
>
> PID : 프로세스 고유의 ID, 양수의 정수값, 중복X
>
> PPID : 부모의 PID

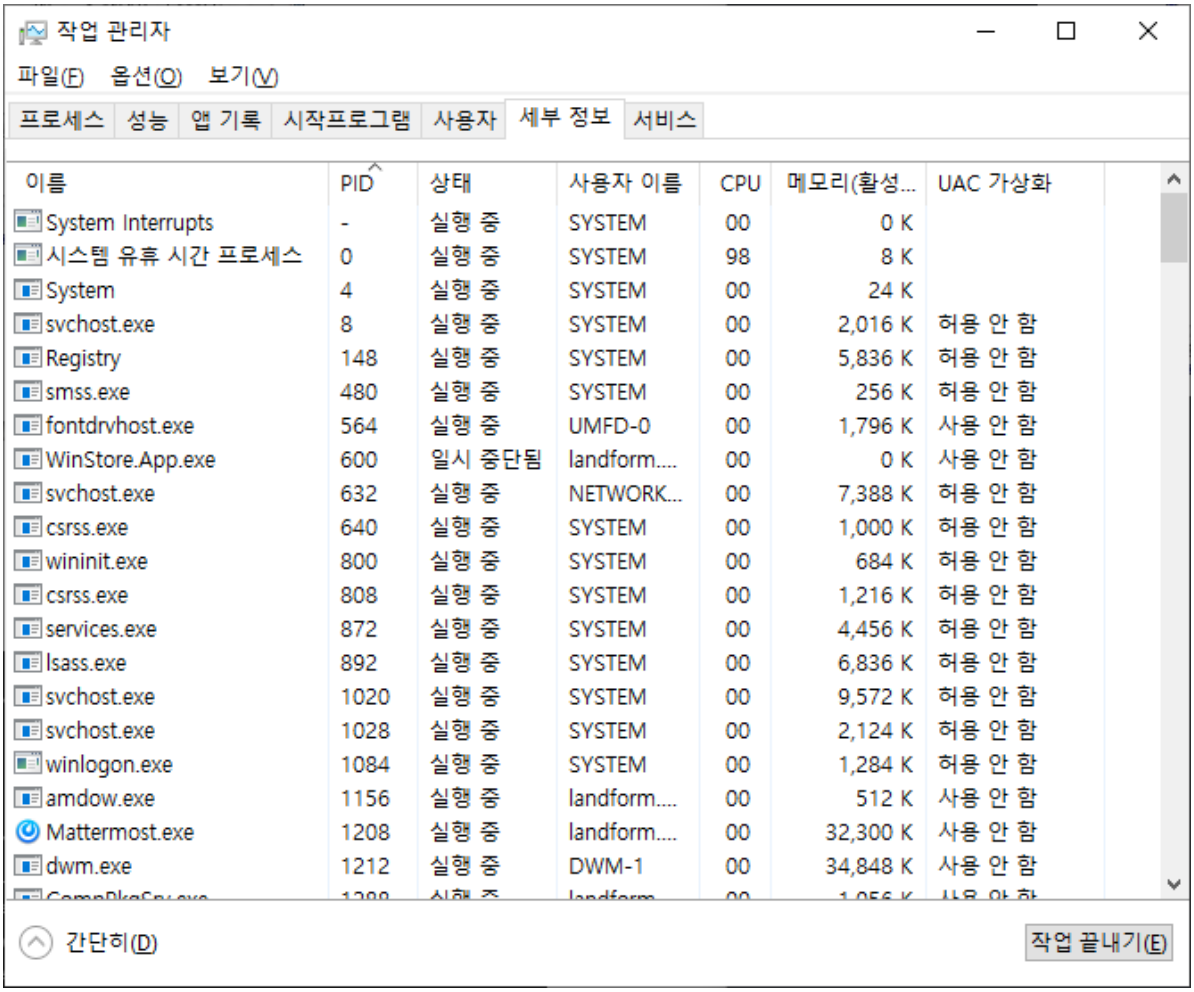


- 프로세스 생성
  1. fork() : fork() system call을 이용해 새로운 프로세스 생성

2. exec() : 실제로 일을 시키기 위해서 실행파일을 하드디스크에서 메모리로 가져오는 것

예제: Windows 7 프로세스 나열

WIndow10의 경우



예제: Ubuntu Linux 프로세스 나열

```
hjyang@rm303:~$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2197  2189  0  80   0 -  1799 wait   pts/0    00:00:00 bash
0 R  1000  2368  2197  0  80   0 -  1177 -      pts/0    00:00:00 ps
hjyang@rm303:~$ ps -axl
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4     0     1     0  20   0   3536  1948 poll_s Ss   ?          0:00 /sbin/init
1     0     2     0  20   0      0     0 kthrea S    ?          0:00 [kthreadd]
1     0     3     2  20   0      0     0 run_ks S    ?          0:00 [ksoftirqd/0]
1     0     4     2  20   0      0     0 worker S    ?          0:00 [kworker/0:0]
5     0     5     2  20   0      0     0 worker S    ?          0:00 [kworker/u:0]
......
1  1000  1820     1  20   0  55944  3992 poll_s Sl   ?          0:00 /usr/bin/gnome-...
4  1000  1831  1658  20   0  50924  9096 poll_s Ssl  ?          0:00 gnome-session --sessio...
0  1000  2196  2189  20   0   2404   724 unix_s S    ?          0:00 gnome-pty-helper
0  1000  2197  2189  20   0   7196  3572 wait   Ss   pts/0      0:00 bash
0  1000  2370  2197  20   0   4708   708 -      R+   pts/0      0:00 ps -axl
hjyang@rm303:~$
```

> ps : process status, 현재 내컴퓨터에서 돌아가는 process 보여줌
>
> UID : user id
>
> PRI : 우선순위

> SZ : 사용중인 메모리 사이즈
>
> TIME : 사용시간

ProcessTermination

Process Termination

- 프로세스 종료
  - *exit()* system call
  - 해당 프로세스가 가졌던 모든 자원은 O/S 에게 반환
    (메모리, 파일, 입출력장치 등)

쓰레드(Thread)

Thread?

- 쓰레드 (Thread)
  - 프로그램 내부의 흐름, 맥

```
class Test {
 public static void main(String[] args) {
    int n = 0;
    int m = 6;
    System.out.println(n+m);
    while (n < m)
     n++;
    System.out.println("Bye");
 }
```

Multithreads

> concurrent : 빠른 스위칭으로 동시에 진행되는 것 처럼 보이는 것
>
> simultaneous : 진짜로 한 순간에 두 개의 일을 하는 것 (CPU가 두 개 이상인 경우)

Thread vs Process

한 프로세스 내에서 code나 data는 공유하나 stack은 쓰레드 별로 따로 사용!

- 옛날 : CPU가 프로세스 단위로 일을 수행함
- 요즘 : 프로세스 내에서도 쓰레드가 존재해서 일을 처리