

# 6강

## 질문

Q. 교수님이 답변해주고 계신 질문 내용입니다

1). Time-sharing에서 "사용자는 실행중인 프로그램에 관여가 가능하다"라는 말의 의미가 잘 와닿지 않습니다. 저는 job의 우선순위에 따른 수행에 관한 내용으로 이해했는데, 좀 더 자세한 설명 부탁드립니다.

A. 프로그램을 사용하다가 멈추는 작업을 수행하는 것이 가능하다.

→ multiprogramming까지는 다른 interrupt를 받지 않도록 설계 되어있었다는 의미이다.

→ Time-sharing에서는 주기적으로 interrupt가 들어오기 때문에 interrupt가 들어 왔을 때 그 interrupt가 의미하는 바를 필요에따라서 수행할 수 있게 되는 것이다.

그래서 사용자 관점에서 프로그램 중간에 명령을 내릴 수 있는 것으로 보여주는 것이다.

2) job과 task의 차이가 무엇인가요?

A. 일반적으로 job은 옛날에 batch를 할 때 사용되는 용어이다.

Time-sharing 이후에 task라는 표현이 등장하는 것을 볼 수 있다.

process와 task의 차이점이 뭐냐?

task는 일반적이고 넓은 의미이고, process은 프로그램을 관리하는데 쓰이는 하나의 abstraction이다.

Thread이든 process이든 이것들을 총괄해서 쓰는 말이 task이다.

3) Issues with multitasking system(19p)에서 "적절한 응답시간을 제공 - Job들은 메모리와 디스크로 swap in/out 될 수 있음"이 의미하는 바에 대해 질문드립니다.

A. 적절한 응답시간을 제공할 필요가 있다는 말이다.

???? (좀더 구체적으로)

swap이라는 것은 디스크로 job들이 나갈 수 있기 때문에 적절한 응답시간을 제공하기 어렵다는 것이다. 메모리가 부족한 경우에 그렇게 될 수 있다,

4) 모든 런타임 시스템은 주어진 프로그램을 프로세스로 만들어 주고 메모리에서 내려오나요, 아니면 해당 프로세스가 종료될 때까지 메모리에 올라가 있다가 내려오나요, 아니면 해당 프로세스가 종료되더라도 시스템이 종료될 때까지 메모리에 남아 있나요? 이런 점들이 런타임 시스템마다 다른가요?

A. yes and no

대부분은 yes 대부분의 런타임 시스템은 메모리에 올라가 있다.

그런데 메모리가 아주 타이트해지면 Memory management 시스템이 메모리에서 내려오는 녀이 가능하다

커널의 구성이 나뉘는 이유는 runtime시스템은 스스로가 메모리에 있어야하는지 아닌지의 여부를 알 수 없다. 그 결정은 Memory management가 한다.

런타임 시스템은 주어진 프로그램을 프로세스로 만들어 주고, 메모리에 남아있는지의 여부는 Memory management가 결정한다.

5) 예를 들어 C로 작성했고 실행이 충분히 오래 걸리는 프로그램이 20개가 있습니다. 이때 프로그램들이 각각 다른 버전의 C 런타임 시스템을 요구할 수 있겠지만 그렇지 않고 동일한 버전의 C 런타임 시스템을 요구한다고 가정합니다. 이것들을 거의 동시에 켜올 경우, C 런타임 시스템이 메모리에 20개 올라가서 이것들을 20개의 프로세스로 각각 만드나요.

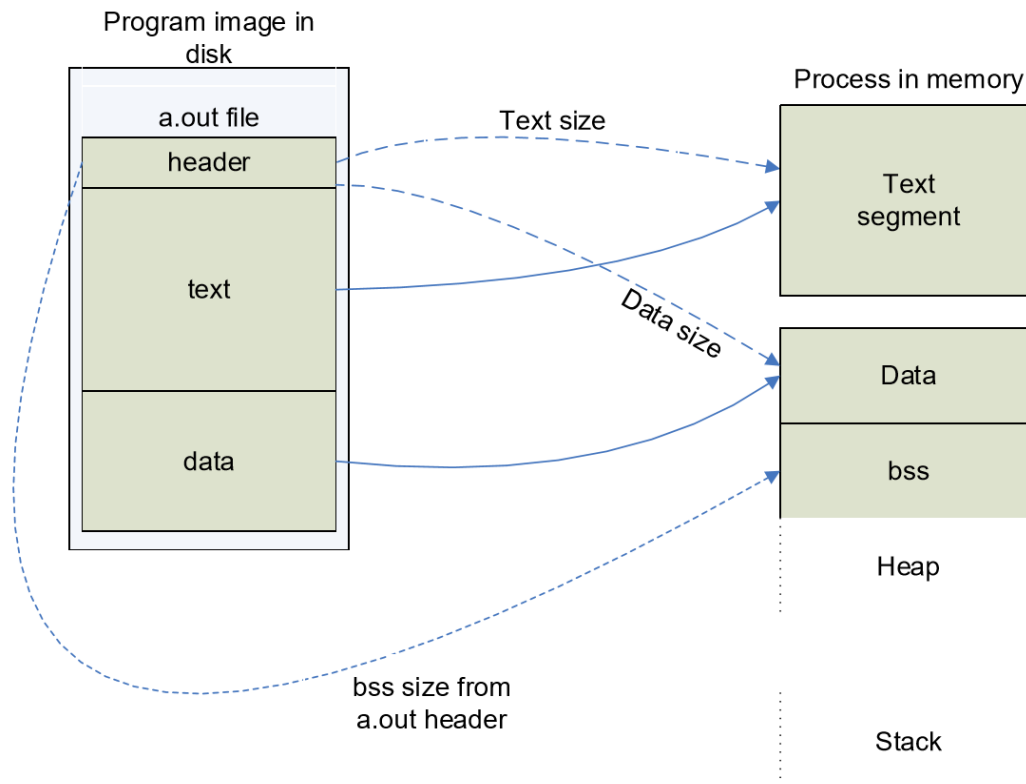
아니면 메모리에 C 런타임 시스템은 하나만 올라가고 그 런타임 시스템이 이것들을 차례 차례 20개의 프로세스로 만들어 줄 뿐인가요? 하나만 올라간다면, 프로그램이 메모리에 올라가고 나서 이것을 프로세스로 만들어줄 수 있는 런타임 시스템이 이미 메모리에 있는지는 무엇이 어떻게 확인하나요? 이런 점들이 런타임 시스템마다 다른가요?

A. yes

C runtime system은 시스템에 하나가 있고 그리고 그 런타임들이 20개의 프로세스를 만들어주고 여기에서 중요해지는 것은 backward computability이다.

새로운 OS가 나오면 거기에 runtime system이 있다. 이 runtime system이 과거에 만들어진 바이너리를 support를 해야한다. backward computability은 새로운 런타임시스템이 과거버전과 똑같은 바이너리를 돌려줄 수 있다는 것이다.

## Process from Program



a.out 포맷을 보면 Header - Text - Data로 구성되어 있다.

Text는 우리가 작성한 code이다. high-level의 데이터가 컴파일되어 저장되어 있는 것이다.

Data는 global 초깃값 변수이다.

이 Header - Text - Data가 Disk에 포맷을 가지고 저장되어 있는 것이다.

Header에 Text size와 Data size가 다 들어 있다. 그리고 그만큼의 메모리를 할당하는 것이다.

RTS가 메모리를 할당한 다음에 Text를 메모리로 복사한다. 그러면 그 text가 있는 메모리 영역이 Text segment가 되는 것이다.

그 다음에 Data를 복사한다.

그 다음에는 이 프로세스의 메모리에 heap을 만든다.

그리고 bss를 만든다.

heap이 뭐냐? 메모리 할당 malloc을 할때의 메모리가 바로 heap영역의 메모리에 할당 되는 것이다.

global data이지만 초기화가 없는 경우에는 bss에 잡히는 것이다.

초기값이 없기에 프로그램에서 위치를 잡고 있지 않는다. 그래서 그냥 bss의 크기만 header에 들어가 있는 것이다.

그렇게 bss를 잡고, heap을 잡고 stack을 만든다.

stack은 가장 하위 메모리에서 내려오고 heap은 메모리 allocation으로부터 올라온다.

메모리alloc을 하면 heap을 가리키는 메모리에서 할당을하고 올라가고, 할당을 하고 올라간다.

만약에 우리가 abs[4]라는 변수를 할당하면 stack이라는 변수에서 할당되는 것이다.

즉, local variable들은 디스크에 저장되지 않고 stack에서 동적으로 할당이 되고, 프로그램 사용이 끝나면 사라진다.

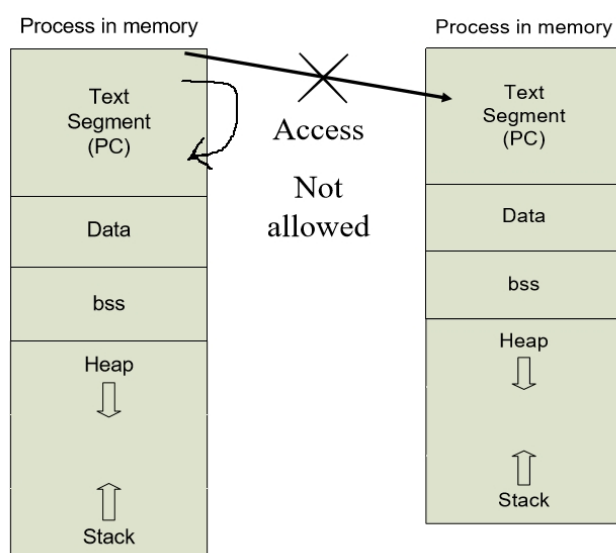
Q. ppt 8페이지 그림에서 점선이 RTS의 역할, 실선이 로더의 역할인가요??

아니다, 점선은 header에서 정보를 알아온다는 의미이다. 해더의 크기 정보를 알아온다는 의미이다.

실선은 복사를 의미한다.

## Process 정의

- Process – abstraction for
  - Execution unit: 스케줄링의 단위 : process단위로 스케줄링이된다는 의미이다.
  - Protection domain: 서로 침범하지 못함



Text segment에서는 다른 프로세스의 text segment를 침범하지 못한다는 임이다.

process란 Execution unit과 Protection domain를 가능하게 하는 abstraction이다.

- Process – implemented with

프로세스는 다음 세가지 레지스터를 구현함으로써 구현된다.

- Program counter
  - 프로세스에서 코드의 어디를 수행하고 있는지를 계산함
- Stack
  - stack pointer이다.
  - stack 부분에서 내가 어디쯤 메모리를 사용하고 있는지를 알려주는 것이다.
- Data section
  - 내 global data의 위치를 알려주는 것이다.

→ 그래서 프로세스의 implement는 CPU의 register이다.

- 프로세스는 디스크에 저장된 프로그램으로부터 변환되어 메모리로 로딩된다
  - 한 마디로 정리하면 이렇게 되는 것이다.

Q. 교수님 bss 다시 한번 설명해주시면 감사하겠습니다

Q. bss 부분이 잘 이해가 가질 않습니다

bss는 초기값이 없는 global variable이다.

초기값을 주면 H-T-D에서 D에 저장되었다가 Data에 올라간다.

Q. 질문이 두가지 있습니다! 1. 초기화되지 않은 데이터를 bss영역에 저장하도록 따로 나눈 이유가 무엇인가요? 2. 힙과 스택이 서로 증가하다가 만나게 될수도 있나요? 혹시 만난다면 어떻게 되나요?

A\_1. 초기값이 있는 것은 저장이 되어있기에 복사만하면 되는데, 초기값이 없으면 메모리를 따로 할당해줘야 하는 등의 일을 해줘야 한다.

A\_2. 만나는 일은 없도록한다. 프로그램 크기와 관련이 있다. Stack이 얼마나 커질 수 있을까?

대부분의 프로그램들은 Text, Data, Stack 모두 그렇게 많지 않다.

생각보다 local variable을 안쓴다. recursive 한 경우에만 Heap이 깊이 들어간다.

Heap과 Stack이 부딪치지 않도록 커널에서 조정한다.

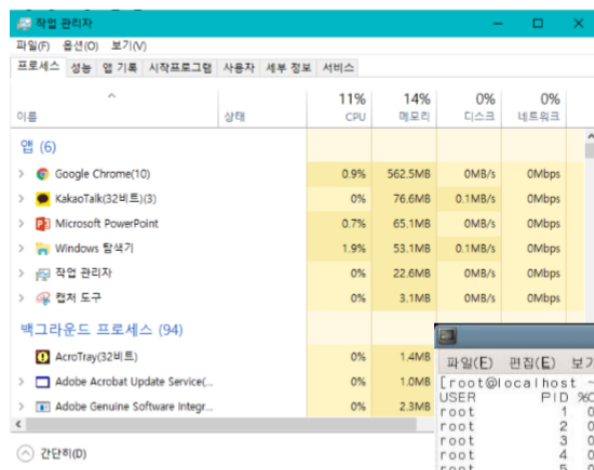
이것은 메모리 management의 영역이다.

해킹의 기본적인 아이디어는 process의 Protection domain을 깨는 것이다.

Q. bss나 heap의 위치는 따로 저장되는 레지스터가 없나요??

A. 이것은 CPU마다 다르다. CPU에 레지스터가 많으면 따로 할당해서 줄 수도 있다. 아니면 레지스터가 없으면 data크기를 계산을 해서 indirect하게 접근한다.

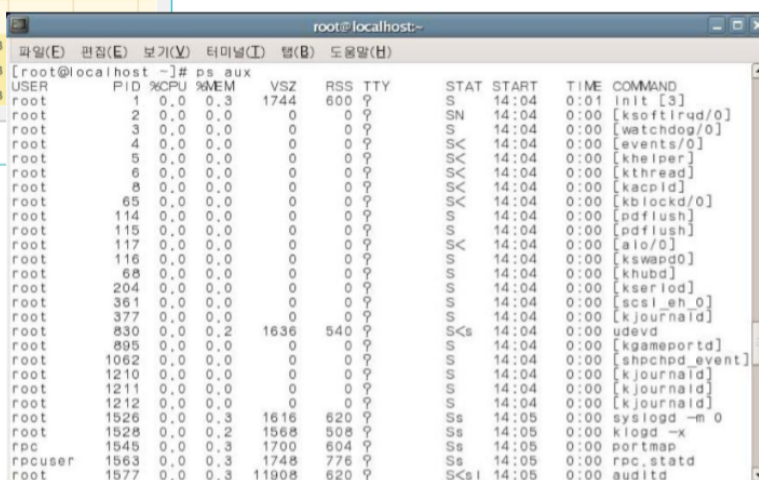
## Process from Program



이름	상태	11% CPU	14% 메모리	0% 디스크	0% 네트워크
<b>업 (6)</b>					
Google Chrome(10)		0.9%	562.5MB	0MB/s	0Mbps
KakaoTalk(32비트)(3)		0%	76.6MB	0.1MB/s	0Mbps
Microsoft PowerPoint		0.7%	65.1MB	0MB/s	0Mbps
Windows 탐색기		1.9%	53.1MB	0.1MB/s	0Mbps
작업 관리자		0%	22.6MB	0MB/s	0Mbps
컬쳐 도구		0%	3.1MB	0MB/s	0Mbps
<b>백그라운드 프로세스 (94)</b>					
AcroTray(32비트)		0%	1.4MB		
Adobe Acrobat Update Service(...)		0%	1.0MB		
Adobe Genuine Software Integr...		0%	2.3MB		

< Windows >

< Linux >



```
root@localhost:~# ps aux
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.3  1744    600 ?        Ss   14:04   0:01 init [3]
root           2  0.0  0.0      0     0 ?        S    14:04   0:00 [ksoftirqd/0]
root           3  0.0  0.0      0     0 ?        S    14:04   0:00 [watchdog/0]
root           4  0.0  0.0      0     0 ?        S<   14:04   0:00 [events/0]
root           5  0.0  0.0      0     0 ?        S<   14:04   0:00 [khelper]
root           6  0.0  0.0      0     0 ?        S<   14:04   0:00 [kthread]
root           8  0.0  0.0      0     0 ?        S<   14:04   0:00 [kacpid]
root           65  0.0  0.0      0     0 ?        S<   14:04   0:00 [kblockd/0]
root          114  0.0  0.0      0     0 ?        S    14:04   0:00 [pdflush]
root          115  0.0  0.0      0     0 ?        S    14:04   0:00 [pdflush]
root          117  0.0  0.0      0     0 ?        S<   14:04   0:00 [aio/0]
root          116  0.0  0.0      0     0 ?        S    14:04   0:00 [kswapd0]
root           68  0.0  0.0      0     0 ?        S    14:04   0:00 [khud]
root          204  0.0  0.0      0     0 ?        S    14:04   0:00 [kseriod]
root          361  0.0  0.0      0     0 ?        S    14:04   0:00 [scsi_eh_0]
root          377  0.0  0.0      0     0 ?        S    14:04   0:00 [kjournald]
root          830  0.0  0.2   1636   540 ?        S<   14:04   0:00 udevd
root          895  0.0  0.0      0     0 ?        S    14:04   0:00 [kgameportd]
root          1062  0.0  0.0      0     0 ?        S    14:04   0:00 [shchedd_event]
root          1210  0.0  0.0      0     0 ?        S    14:04   0:00 [kjournald]
root          1211  0.0  0.0      0     0 ?        S    14:04   0:00 [kjournald]
root          1212  0.0  0.0      0     0 ?        S    14:04   0:00 [kjournald]
root          1526  0.0  0.3   1616   620 ?        Ss   14:05   0:00 syslogd -m 0
root          1528  0.0  0.2   1568   508 ?        Ss   14:05   0:00 klogd -x
root          1545  0.0  0.3   1700   604 ?        Ss   14:05   0:00 portmap
root          1563  0.0  0.3   1748   776 ?        Ss   14:05   0:00 rpc.statd
root          1577  0.0  0.3   11908   620 ?        S<   14:05   0:00 auditd
```

11

리눅스에서의 pid는 프로세스 아이디이다.

## 문맥전환(context switch)

- CPU에서 수행되는 프로세스가 바뀌는 것  
→ 프로세스1이 cpu에서 돌아가다가 프로세스2가 돌아가는 것이다.  
프로세스가 바뀌는 것을 context가 바뀌는 것이라고 말한다.

- 언제 발생하나
  - Time quantum expires
  - I/O 호출
- 프로세스 old 가 위의 조건에 해당하면 old 의 동작을 멈추고, 프로세스 new를수행한다

- 커널에서 문맥 교환을 담당하는 부분을 dispatcher라고 한다

Q. dispatcher의 일이 뭐가 있겠는가? dispatcher의 역할이 뭐겠는가?

A. 디스패처가 앞서서 말한 프로세스를 가르키는 레지스터들의 값을 바꿔줘야 할 것 같습니다

프로세서 1에서 사용되는 레지스터의 값을 프로세서2로 바꾸는 것이다.

1. 레지스터를 교환한다.
2. 레지스터를 save한다.
3. 레지스터를 reload한다.
4. p2를 결정해줘야한다.

그것을 위해서는 cpu안의 레지스터들을 하나씩 다 저장을 하고 새로운 p2의 레지스터를 가져와야 한다. 그리고 디스패처가 프로세서2를 결정해줄 필요성 또한 있다.

A. 메모리 공간이 부족하다면, disk와 메모리에서 swapping이 발생해야 할 것 같습니다.

→ 아직 메모리 이야기는 하지 말자.

→ 디스패처와 swapping은 관계가 없다. 디스페처는 메모리에 관여하지 않는다.

Q. 그럼 이전의 CPU 레지스터 값들은 어디에 저장되나요?

A. bss와 stack은 아니다. 왜냐하면 여기에는 프로세스의 특정한 목적이 있기 때문이다.

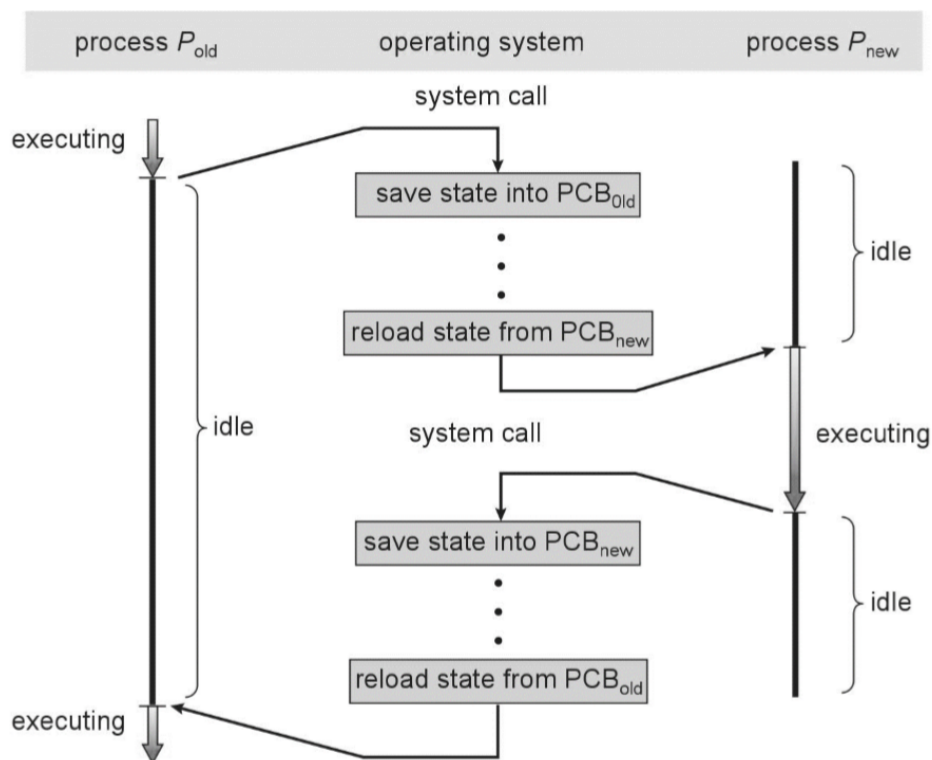
그렇기에 커널 내부의 어딘가에 저장을 한다.

어딘지는 나중에 말해준다.

Q. 혹시 dispatcher에서 새로운 process를 결정할 때 현재 process가 priority가 높아서 연속적으로 실행되게 결정한다면 state를 저장하고 restore하는 과정은 생략되게 되나요?

A. 만약 연속적으로 수행하게 결정된다면 state를 저장하고 restore하는 과정에서 overhead가 발생하기에 생략되는 것이 성능적인 관점에서 더 좋고, 실제로도 그렇게 된다.

## Context switch



interrupt와 비슷하다. 차이점은 interrupt은 돌아가지만, Context switch는 안 돌아 간다.

저장되고 reload되는 어딘가는 커널 안의 PCB(process control block)라는 곳이다.

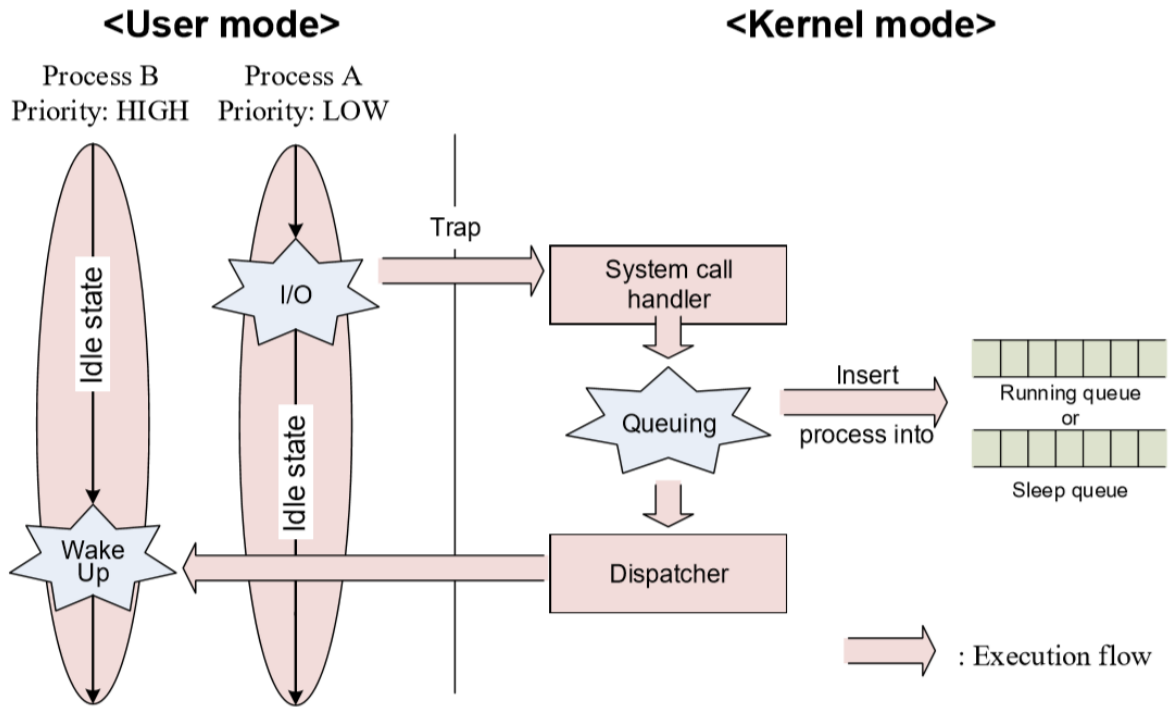
여기의 system call은 잘못된 표현이다. Context switch라고 봐야한다.

idle은 sleep을 의미한다.

그리고 우리 수업은 cpu가 하나라는 가정을 둔다.

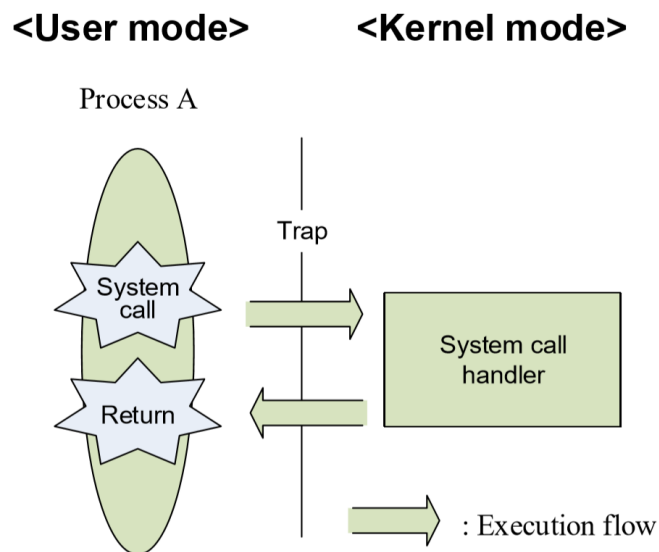
## User mode and Kernel mode





여러개의 프로세스가 처리되려면 queue가 필요하다는 것이다. queue로 처리를 한다.  
Higher Priority Process는 Running queue로, Lower Priority Process는 Sleep queue로 들어간다.

## 시스템콜-문맥교환없음



그냥 system call은 Context switch을 일으키지 않는다. I/O가 아니다.  
오직 시간을 다쓰거나, I/O를 할 때만 변화가 일어난다.

## 프로세서 구조에 따른 문맥전환의 차이

- CISC

HW로 많은 처리를 하려는 의도이다.

- 복잡한 명령어 셋 구성 → 효율 높임, 클럭 속도 저하

HW logic이 복잡해지더라, 회로가 복잡하니 효율이 오히려 떨어지고 클럭이 떨어지더라.

- 복잡한 회로 → 물리적인 공간 차지 → 레지스터 용량 저하
- Ex) Intel pentium processor

- RISC

- 간단한 명령어 셋 구성 → 클럭 속도 높임 → 빠른 수행 속도
- 절약된 물리적 공간에 보다 많은 레지스터 장착
  - 문맥 전환 시 레지스터 내용 변경에 보다 큰 오버헤드가 생김
    - CISC에서는 레지스터의 갯수가 적었다. 저장할 것이 적다.
    - RISC에서는 레지스터의 갯수가 많아졌다. 큰 오버헤드가 생겼다.
- 예) Sparc, Arm processor
- Register Window (Berkeley RISC design) → 오버헤드를 줄이기 위함
  - 두 프로세서간의 레지스터를 공유, 오버랩하게 만드는 것이다.
  - HW die에 공유가능한 레지스터를 만들어 오버헤드를 줄인다.

## Process control block

- Process control block (PCB)
  - Each process is represented by PCB.
    - 하나의 프로세스당 하나의 PCB가 존재한다.

- Located in kernel memory  
→ PCB는 커널 메모리에 존재한다.

- Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
  - priority도 들어 있다.
- Memory-management information
- Accounting information
- I/O status information

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

Q. PCB가 아까 말씀하신 queue로 관리된다고 보면 되는걸까요?

A. yes. 오른쪽 그림의 Pointer가 그런 것을 나타내는 queue이다.

Running queue → CPU에 올라갈 큐이다.

Sleep queue

Q. 시스템 콜에서 문맥교환이 없다는 것은 trap이 발생할 시 cpu가 IDLE 상태로 유지된다고 생각하면 되나요

A. 프로세스는 idle하나, cpu는 idle하지 않는다.

Q. 현재 강의와 관련없지만 궁금한 점이 생겨 질문드립니다! 한 컴퓨터에 두개의 NIC가 있고 모두 네트워크에 연결되어 있을때 어떤 NIC를 통해 패킷을 주고 받을지는 어디서 결정하나요? 응용프로그램이 결정하나요, OS가 결정하나요..?

A. 2학기에 나와야 할 질문이다.

NIC가 2개 있다. 그렇다면 무엇이 어디로 받을지 결정하겠는가?

OS 아니겠는가?

A. 응용프로그램이 우선순위를 정할 수 있되 디폴트는 OS가 결정할 것 같습니다.

→ OS의 기본 철학은 그렇지 않다.

응용프로그램이 우선 순위를 결정가능하면 스스로의 priority를 올리려고 하지 않겠는가?

→ 사용자가 이렇게 하는 것을 허락하지 않으려고 한다.

이것은 운영체제 디자인의 원칙이다.

일차적으로 connection에 의해서 달라진다.

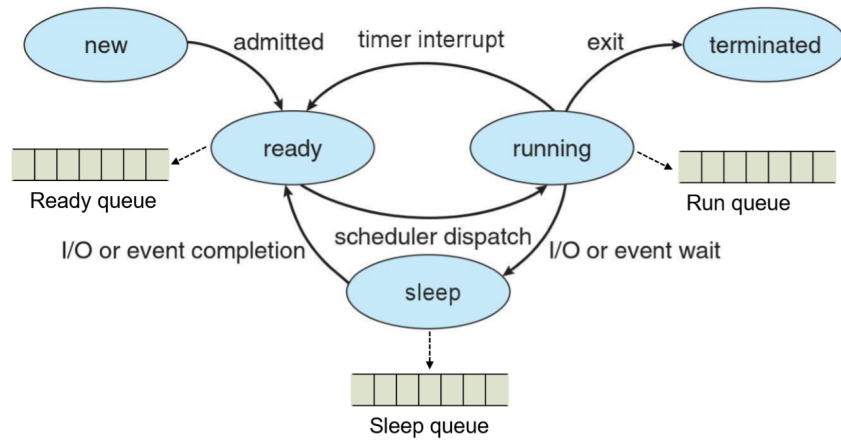
아마 왔다갔다 하게 만들 것이다.

## Process State

- New: process is created.
- Running: process is being executed.
- Sleep: process is waiting for some event to occur such as I/O completion
- Ready: process is waiting to be dispatched to a processor
- Terminated: process has finished execution
- 각 상태 마다 큐가 있음
  - ready queue, sleep queue, run queue

이것은 프로세스의 순환주기를 말하는 것이다.

## Transition of process state



ready 상태인 프로세스는 여러개일 수 있다.

프로세스가 만들어지면 ready로 갔다가

dispatcher가 스케줄링을 하면 running으로 간다.

카카오톡은 ready → running → sleep → ready가 계속 반복된다.

엄청나게 빠른 속도로 사이클이 돈다.

sleep queue에 들어가면 context 스위치가 일어난다.