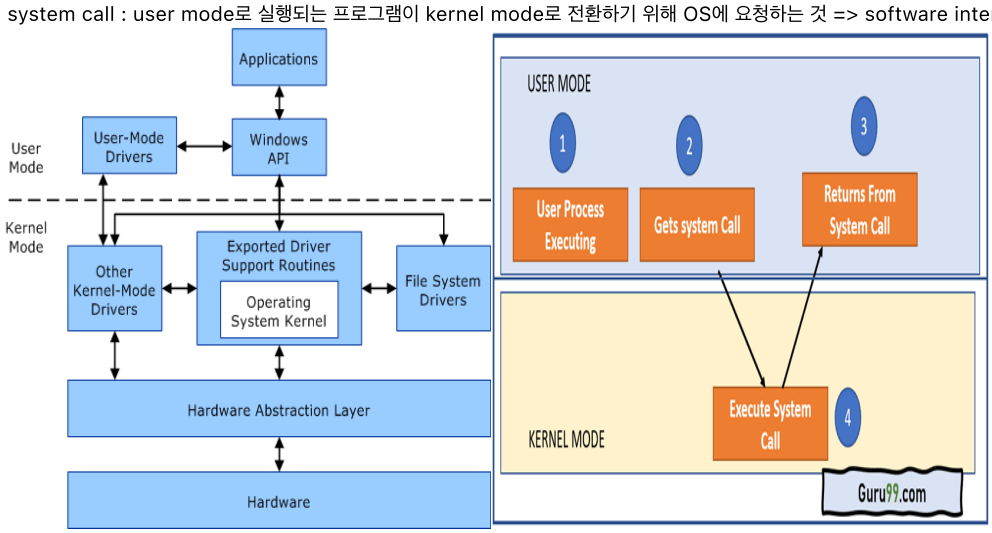
Operating system

Overview of OS

- OS : 실행할 프로그램에 필요한 resource를 할당하고 올바르게 실행되도록 돕는 특별한 프로그램
- OS도 프로그램이므로 memory에 저장된 상태로 실행되어야 하는데 특별히 kernel space에 따로 저장되어 있으며, 이 외의 프로그램은 user space에 저장됨
- resource 관리
  - memory : 실행할 프로그램을 메모리에 적재하고, 더 이상 실행하지 않는 프로그램을 메모리에서 삭제
  - CPU : program의 실행 순서, 실행 시간 등 CPU 자원 분배
- key service
  - process control
    - process : 실행중인 프로그램
  - resource access and allocation
  - file system control

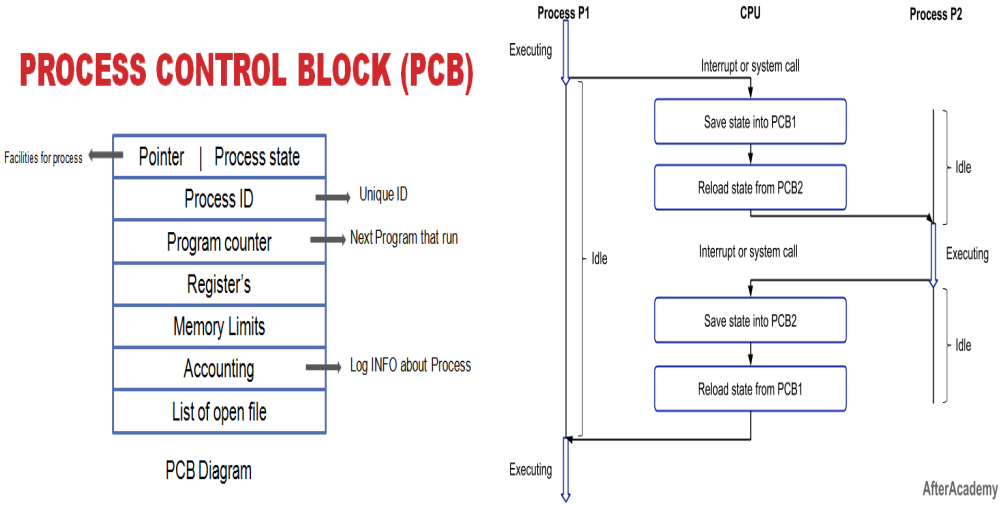
Kernel

- OS의 핵심 기능인 자원에 접근하고 조작하는 일을 담당하는 부분을 kernel이라고 함
- dual mode => CPU가 instruction을 실행하는 모드를 구분해놓은 것 (모든 instruction이 자원에 직접 접근하고 변경하면 위험하므로)
  - user mode : OS service 불가. 즉 kernel 영역의 code을 실행할 수 없음
  - kernel mode : OS service 가능. 즉 kernel 영역의 code를 실행할 수 있음
  - system call : user mode로 실행되는 프로그램이 kernel mode로 전환하기 위해 OS에 요청하는 것 => software interrupt

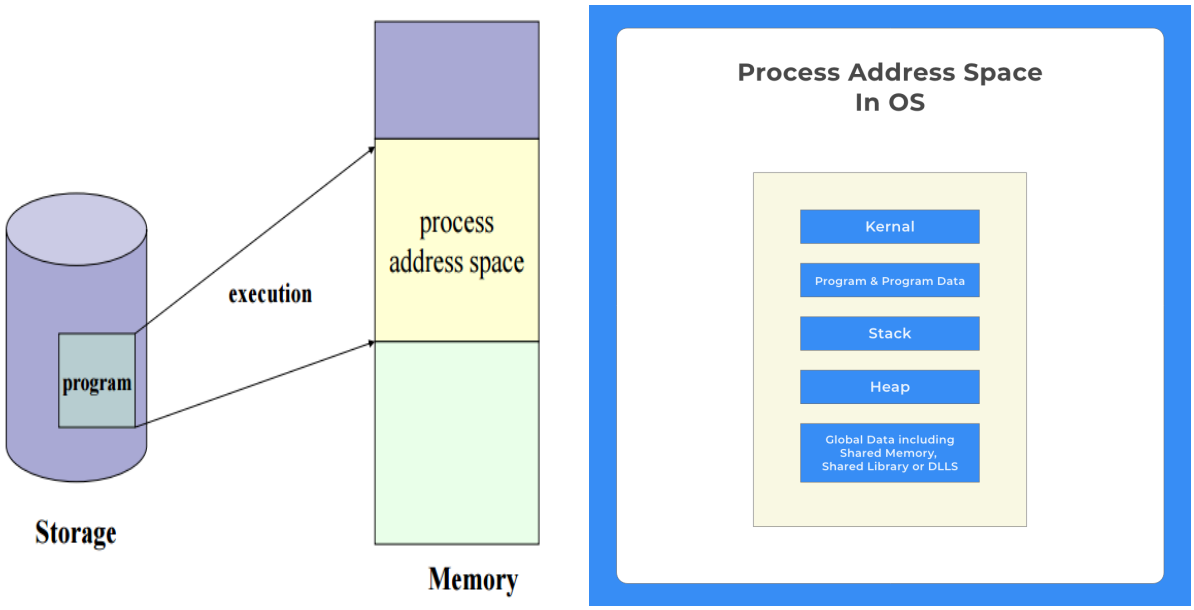


Process & Thread

- program -> set of instruction
- process -> excuted program
- PCB (Process Control Block)
  - CPU는 여러 process들을 번갈아 실행하기 위해 실행 순서와 자원을 조정하는데 이를 위해서는 process와 관련된 정보를 저장하고 있어야 함
  - 이러한 정보를 저장하고 있는 data structure를 PCB라고 하며 kernel 영역에 process 생성 시에 만들어지고 실행이 끝나면 폐기됨
  - PCB에 담기는 정보 => process id, register, process state, cpu scheduling, memory address, page table ...
- Context switching
  - context : 하나의 process 실행을 재개하기 위해 기억해야할 정보
  - 여러 process를 번갈아 실행하려면 반드시 context를 PCB에 저장해야 함
  - A -> B 순으로 process가 switch될 때, A의 context를 PCB에 backup하고, B의 context를 load하는 작업을 context switching이라고 함



- memory space of process
  - process가 실행되면 storage에 memory space가 할당됨
  - kernel space => PCB 담당
  - user space => stack, heap, data, code
    - code(text) : executable code(machine code)로 이뤄진 instruction이 저장. CPU가 실행할 instruction이 담긴 공간이므로 read-only
    - data : 프로그램이 실행되는 동안 유지할 데이터가 저장되는 공간. global variable이 대표적
    - heap : 프로그래머가 직접 할당할 수 있는 memory space. memory space를 할당하고 반환하지 않으면 memory leak 발생
    - stack : 데이터를 일시적으로 저장하는 공간. local variable, arguments
    - code + data -> static / heap + stack -> dynamic allocation

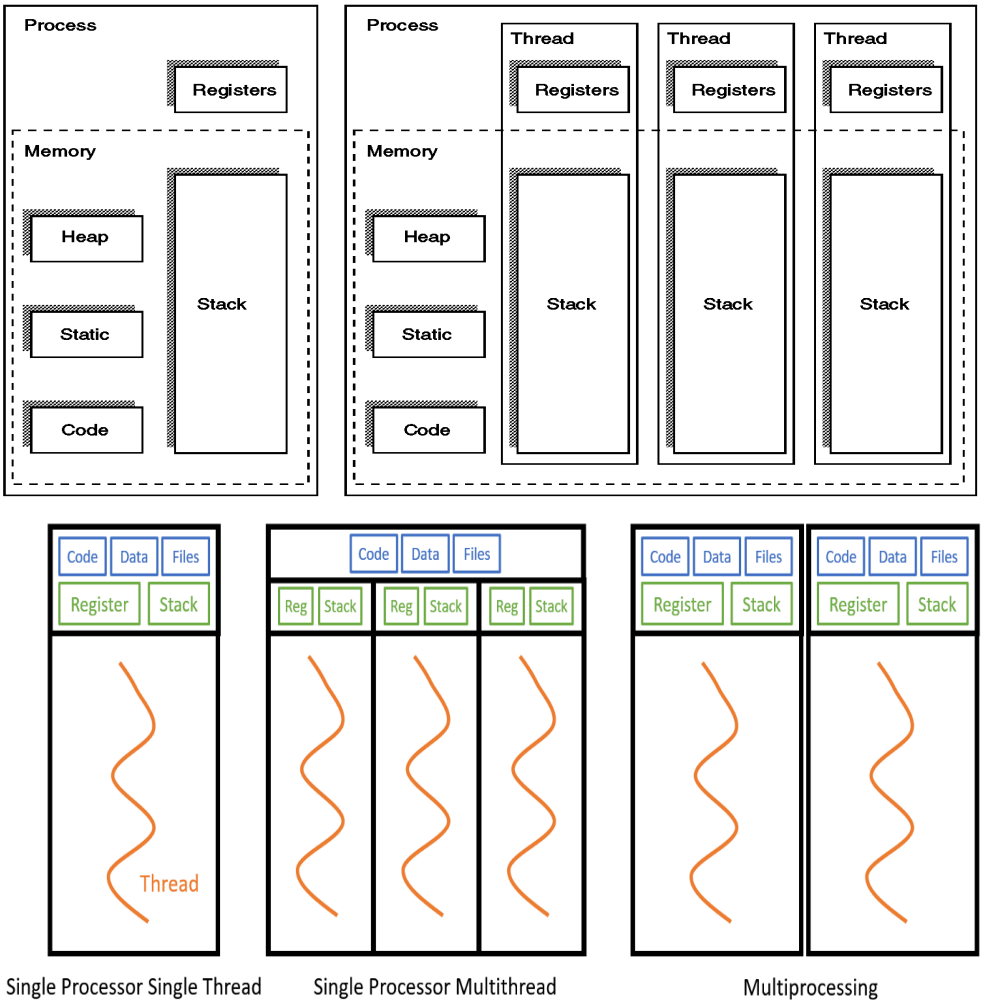


Thread (in detail)

- process를 구성하는 실행의 flow 단위
- 하나의 process는 여러개의 thread를 가질 수 있음
- 전통적인 의미에서 하나의 process는 한 번에 하나의 일만을 처리할 수 있었음
- thread라는 개념이 도입되면서 **하나의 process가 한 번에 여러 일을 동시에 처리할 수 있게 됨**
- multi-process vs multi-thread
  - multi-process : 여러 process를 동시에 실행
  - multi-thread : 여러 thread로 1개의 process를 동시에 실행
  - 1개 process를 동시에 여러번 실행하는 것과 multi-thread를 여러개 만들어 한번씩 실행하는 것의 차이는?
    -> process끼리는 resource를 공유하지 않지만, thread끼리는 같은 process 내의 resource를 공유함
    -> 아래 그림처럼 여러 개의 process들이 동일한 memory space의 code/data(static)/heap을 공유
    -> process는 공유없이 전부 다 copy해서 실행함

-> process는 resource를 공유하지 않기 때문에 독립적이나, thread는 상호간 communication에 유리함. 단 resource를 공유하기 때문에 그 쪽에 문제가 생기면 다 문제가 생길 수 있음

- IPC (Inter-Process Communication)
  -> process 간에 resource를 공유하고 data를 주고받는 방법
  -> File을 통해서도 가능하고, 서로 공유하는 메모리 영역을 두고 데이터를 주고 받을 수도 있음 (shared memory)



CPU scheduling

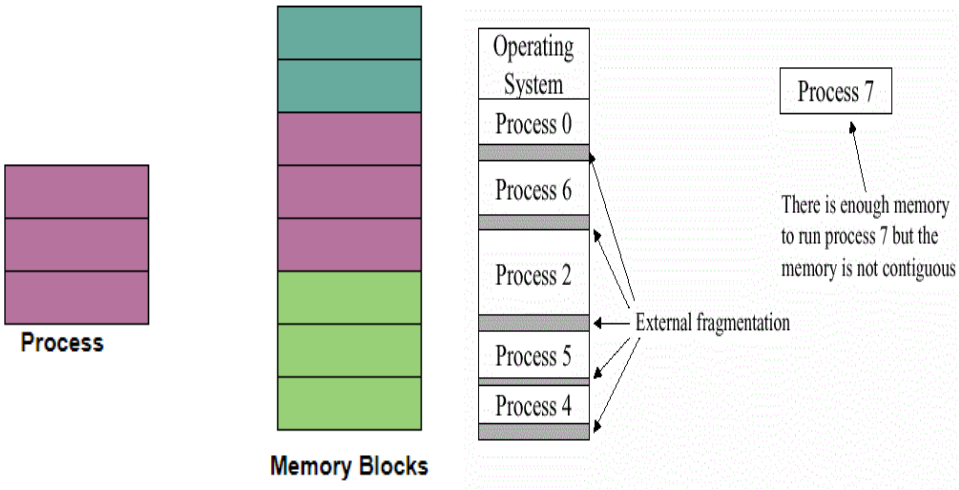
- OS가 process들에게 CPU resource를 배분하는 것
- OS는 PCB에 priority를 명시하고 이에 따라 먼저 처리할 process를 결정함
- process들은 scheduling queue에서 기다림
- scheduling queue
  -> ready queue : CPU를 이용하고 싶은 process들이 서는 줄
  -> waiting queue : I/O 장치를 이용하기 위해 대기 상태에 접어든 process들이 서는 줄
  -> 이름은 queue지만 FIFO(First In, First Out) 방식은 아님. 늦게 줄을 서도 priority가 높다면 먼저 실행될 수 있음
- CPU가 A라는 process를 실행 중인데 B라는 process가 지금 당장 실행되길 요청한다면?
  -> preemptive scheduling : 실행 중이어도 OS가 resource를 빼앗아 B에게 줌
  -> non-preemptive scheduling : 실행 중인 A가 끝나야 B에게 줄 수 있음
  -> 대부분의 CPU는 preemptive scheduling

Virtual memory

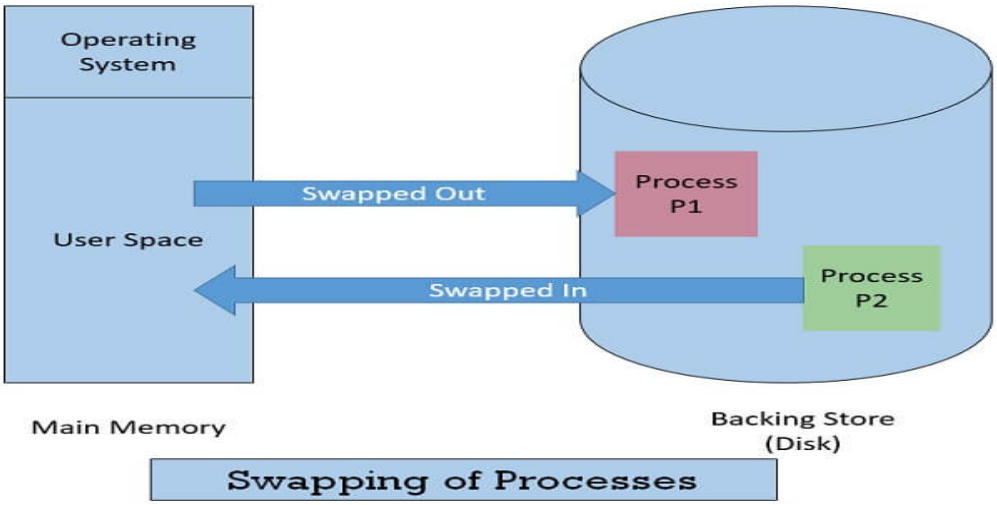
1. contiguous memory allocation : process를 연속적인 memory space에 할당하는 방식 (아래 그림 왼쪽)
   - 문제점 : external fragmentation (아래 그림 오른쪽)
   - process들이 연속적으로 allocation되어 있다가 실행이 끝나면 그 자리가 비어있게 되는 데, 총 memory의 용량은 충분하지만 간격이 애매하게 있어서 새로운 process를 실행할 수 없는 상태 (아래 그림에서도 검정색 부분을 다 합치면 process 7보다 크지만, external fragmentation으로 인해 loading 불가)
   - compaction : external fragmentation을 해결하기 위해 memory의 빈공간을 하나로 모아 큰 memory space를 만드는 것. 단, 이를 위해서 OS는 하던 일을 멈추고 작은 공간들을 탐색해야 하고, memory의 내용들을 옮기는 것 자체가 큰 overhead를 야기하는 비효율적인 작업. 어떤 순서로, 어떤 방식

으로 compaction을 해야 가장 효율적인 지에 대한 알고리즘도 없음

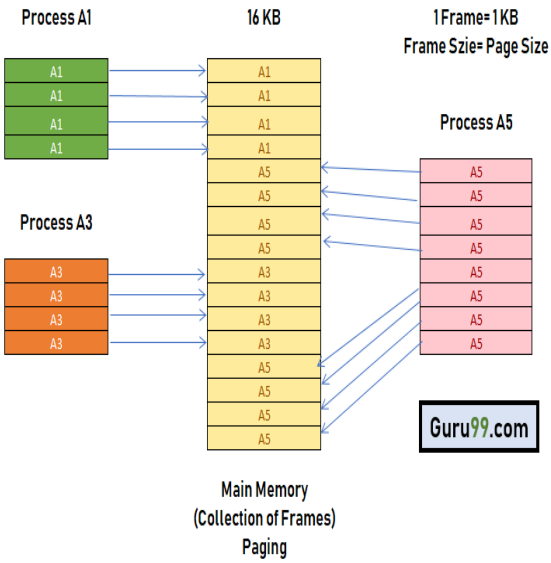
- 이를 효율적으로 해결하기 위한 방법이 ***virtual memory와 paging***

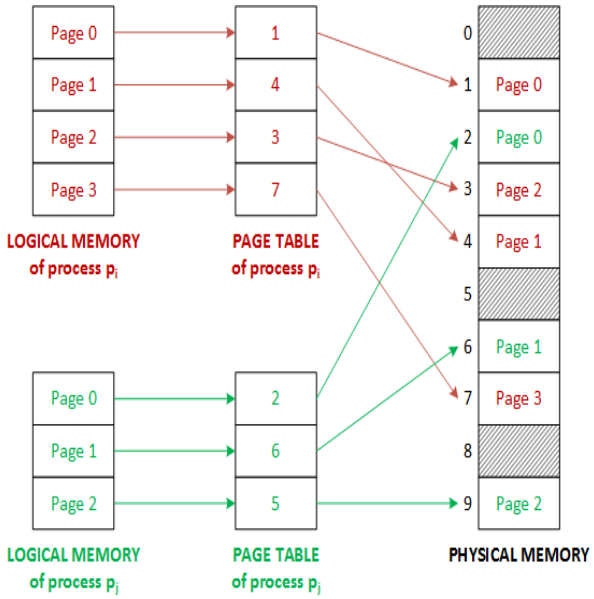


2. swapping : I/O 장치의 요구로 wait 상태가 된 process/오랫동안 사용되지 않은 process들을 보조기억장치(disk)로 쫓아내고, 그렇게 해서 생긴 memory space에 다른 process를 추가해서 실행하는 방식. 쫓겨나는 것을 swap 아웃, 쫓겨난 disk 영역을 swap space, swap space에서 다시 memory로 옮겨오는 것을 swap in이라고 함. swap out 후에 swap in을 하면 같은 process여도 다른 memory address에 load될 수 있음
   - swapping을 이용하면 process들이 요구하는 총 memory가 실제 memory보다 크더라도 동시에 실행할 수 있음
     -> (ex) A, B, C, D process가 요구하는 memory가 20, 30, 40, 50이고 실제 memory가 100이면, 140 > 100이므로 A,B,C,D를 동시에 실행할 수 없지만, memory swapping을 이용해서 A loading, B loading, C loading, A swap out, B swap out, D loading을 하면(20+30+40-20-30+50) 가능 (대신에 A,B가 swapping이 가능한 상태여야 함!)



3. virtual memory & paging
   - virtual memory : 실행하고자 하는 program을 일부만 memory에 loading하여 실제 physical memory보다 더 큰 process를 실행할 수 있게 하는 기술 (즉 어떤 실질적인 memory를 얘기하는 것이 아니다)
   - 이를 가능케 하는 기법에는 *paging과 segmentation* 이 있는데 대부분 paging기법 사용 (paging 사용하면 external fragmentation도 해결 가능)
   - paging
     (1) memory와 process를 일정한 단위로 자르고, 이를 memory에 non-contiguous하게 할당
     (2) process의 logical address space를 page라는 일정한 단위로 자르고, memory physical address space를 frame이라는 page와 동일한 크기의 일정한 단위로 자른 뒤 page를 frame에 할당하는 기법
     (3) swapping도 process가 아닌 page 단위로 swapping할 수 있음
     (4) 문제점 : process를 잘게 쪼개놓다 보니, 어디서 시작해서 어디서 끝나는 지 CPU가 알기가 힘듦 -> page table 활용
     (5) page table : page number + frame number => 어떤 page가 어떤 frame에 loading되었는 지에 대한 정보를 담고 있음. 따라서 CPU는 logical memory에 따라서 page들을 순차적으로 실행하면 됨
     (6) page table에서 frame을 찾아갈 때 발생하는 overhead를 해결하기 위해 TLB(Translation Lookaside Buffer)라는 cache memory에 저장해서 빠르게 접근할 수 있도록 함


Process A1
16 KB
1 Frame = 1 KB
Frame Szie = Page Size
Process A5
Process A3
Guru99.com
Main Memory
(Collection of Frames)
Paging


Page 0
Page 1
Page 2
Page 3
LOGICAL MEMORY
of process p_i
PAGE TABLE
of process p_i
Page 0
Page 1
Page 2
LOGICAL MEMORY
of process p_j
PAGE TABLE
of process p_j
PHYSICAL MEMORY