# 운영체제(공영호)

Lee Jun Hyeok (wnsx0000@gmail.com)

December 19, 2024

# 목차

# 1. Operating System

## 1.1. Operating System

### 1.1.1. Operating System

***1. Operating System***
*운영체제(OS, Operating System)는 program을 실행하고, memory를 관리하고, device 간 상호작용을 도와 computer system이 정확하고 효율적으로 동작하도록 하는 중앙 제어 소프트웨어임.*

*운영체제의 작업은 아래와 같이 2가지로 나누어 생각할 수 있음.*
*1. hardware를 쉽고 효율적으로 사용할 수 있도록 하는 abstraction을 제공함.*
*2. system에 대해 자원의 공유/분배를 위한 policy를 결정함.*

*OS는 process 관리, memory 관리, I/O 관리, network 관리, 보안 등의 기능을 수행함.*

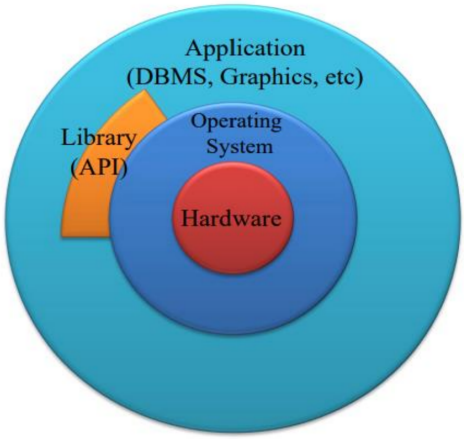
***2. system software***
*software는 system software와 application software로 분류할 수 있음.*

*system software는 computer system의 동작을 위한 software로, OS, compiler, assembler 등이 있음. application software는 특정 목적을 위한 software로, OS 위에서 동작함.*

*OS는 application software와는 달리, 항상 동작하며 자원과 hardware를 관리함. 즉, OS는 kernel (supervisor) mode로 동작하고, application software는 user mode로 동작함.*

***3. Kernel***
*커널(Kernel)은 OS의 핵심 부분으로, OS의 주요 기능들을 처리하는 부분임.*

*OS는 kernel만을 포함하는 개념으로 사용되기도 하고, kernel, GUI, library를 모두 포함하는 개념으로 사용되기도 함.*




### 1.1.2. Abstraction

*추상화(Abstraction)는 복잡한 시스템을 단순화하는 기법으로, software 전반에 걸쳐 사용되는 개념임. OS는 hardware에 대한 abstraction을 제공함.*

*abstraction에 의해 사용자는 해당 시스템을 편리하게 사용할 수 있음. 하지만 abstraction 아래에 존재하는 구체적인 내용을 알아야 성능을 잘 최적화할 수 있음.*

*OS는 CPU는 process로, memory는 address space로, disk는 file로, network는 port로 abstraction함.*

***1. Process***
*프로세스(Process)는 실행 중인 program에 대한 abstraction임.*

*CPU 등의 hardware가 실행 중인 각 program을 구분하고 실행할 수 있도록 함.*

프로그램(program)은 computer의 동작에 대한 명령어 집합으로, disk에 이진수의 형태로 저장되어 있음. 실행 중인 program(disk에서 memory에 올라간 program)을 process라고 함.

process는 program counter(PC), Stack, Data section 등으로 구현됨.

***2. Address space***
Address space는 process가 가지는 memory 공간에 대한 abstraction임.

각 process가 서로의 memory를 침범하지 않도록 하고, memory에 매핑된 device를 사용할 수 있도록 하는 등의 역할을 수행함.

***3. File***
파일(File)은 disk에 대한 abstraction임.

process는 file을 사용해 구체적인 주소나 위치를 몰라도 disk에 대해 읽거나 쓰는 작업을 수행할 수 있음.

***4. Port***
포트(Port)는 network 연결에 대한 abstraction임.

장치는 ip로 구분되고, port는 장치 내부 process의 구분을 위한 communication endpoint로 사용됨.

## 1.2. OS의 발전

### 1.2.1. 초기의 computer

최초의 computer는 2차 세계대전에서 암호 해석, 미사일 탄도 분석 등을 계산하기 위해 등장하였음.

1950년대 초의 computer는 진공관 기반으로(현재는 트렌지스터 기반임.), 굉장히 크고 많은 열을 방출했음. 이때는 Hand-operated System만이 존재하여 기계적 스위치를 직접 조작하여 1bit 단위로 computer에 입력했음.

1950년대 중반의 computer는 플러그 보드에 wire를 적절히 연결하여 동작을 제어(예전처럼 완전 수동은 아님.)했고, 기계어만으로 program을 작성하였음. PL과 OS라는 개념이 존재하지 않았고, 영구저장장치가 존재하지 않았음.

1960년대 초에는 punch card를 사용하는 mainframe이 등장하여 플러그 보드를 대체했음. punch card는 program 동작에 대한 정보를 가지고 있어서 장치에 읽히면 해당 program을 돌릴 수 있었음.

### 1.2.2. Batch System

***1. Batch System***
Batch System은 작업(job)을 일괄적으로 처리하는 아주 단순한 형태의 OS임.

초기 mainframe에서 단순히 punch card를 제출하면 해당하는 job만을 처리하는 system으로 사용되었으며, 한 번에 하나의 job만을 처리할 수 있고 중간에 user와 상호작용할 수 없음. 당연하게도 scheduling이 불가능해서 과거에는 사람이 punch card의 순서를 적절히 제공하는 방식으로 직접 scheduling해야 함.

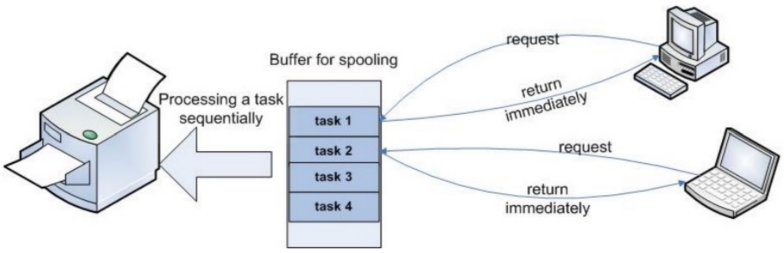
scheduling이 효율적으로 수행되지 못하고, I/O 작업(punch card를 읽거나, 결과를 출력하는 등.)에 의해 cpu가 idle(작업 중이지 않은) 상태로 자주 전환됨.

***2. Automatic Job Sequencing***
batch system에 Automatic Job Sequencing을 적용하여 job을 자동으로 scheduling하도록 할 수 있음.

하지만 여전히 I/O 작업에 의한 cpu의 idle 상태로의 전환은 방지하지 못함. spooling 또는 multiprogramming으로 이를 방지할 수 있음.

***3. Spooling***
Spooling(Simultaneous Peripheral Operation On-line)은 buffer 등을 사용하여 I/O 작업을 효율적으로 관리하는 기법임. job을 처리할 때까지 cpu가 기다리는 대신, data를 buffer에 넣어두고 해당 장치가 buffer에서 data를 꺼내 사용하도록 하는 것.

*spooling을 적용하면 I/O 작업에 의한 cpu의 idle 상태 전환을 방지할 수 있음. 현재에도 프린터 등에 대해서는 spooling batch system을 사용함.*




### 1.2.3. Multiprogramming

***1. Multiprogramming***
*Multiprogramming은 여러 개의 job을 memory에 올려 두고 여러 job을 동시에 처리하도록 하는 기법임. 어떤 job이 I/O을 수행한다면 cpu는 다른 job을 처리하도록 하여 idle 상태로의 전환을 방지함.*

*multiprogramming은 cpu의 idle 시간을 줄이는 기법으로, user가 실행 중인 job에 관여할 수는 없음. timesharing을 적용하여 user가 관여할도록 할 수 있음.*

*multiprogramming에서는 job이 I/O을 처리할 때 다른 job으로 전환되는데, process가 이기적으로 I/O을 처리하지 않도록 설계되어 cpu 사용을 독차지할 수 있는 문제가 있음. 또한 어떤 job은 우선적으로 처리되어야 할 수 있음.*

### 1.2.4. Multitasking

***1. Timesharing***
*Timesharing은 cpu의 실행 시간을 time slice(주로 10ms)로 나누어 사용하는 기법. 각 job이 time slice 동안만 cpu를 사용하고 다른 job에 사용을 양보함.*

*여러 개의 job들이 cpu switching을 통해 실행되고, user는 job의 전환/중단/수정에 비교적 쉽게 관여할 수 있음.*

*현재의 OS는 대체로 timesharing을 사용함. 이는 interrupt를 사용해 구현됨.*

***2. Multitasking***
*Multitasking은 여러 개의 task(job을 분해한 더 작은 단위)들이 cpu 등의 자원을 공유하도록 하는 기법으로, Multiprogramming에 timesharing이 적용된 것으로 생각할 수 있음.*

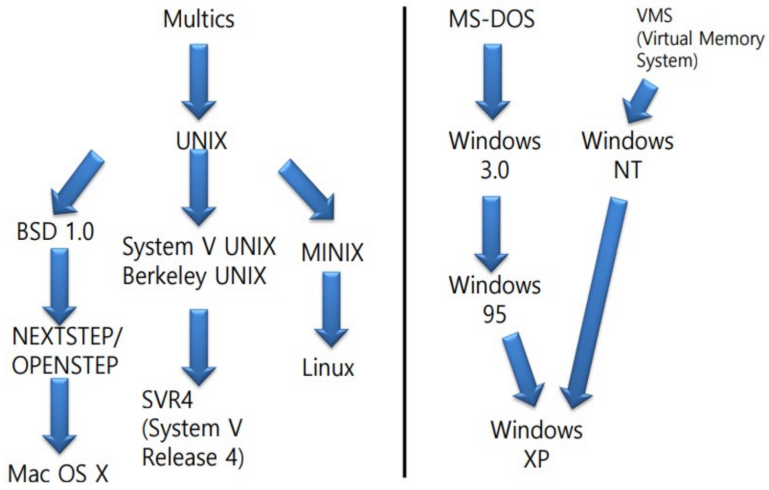
*multitasking에서는 여러 개의 process들이 동시에 수행되고, 각 process들이 child process를 만들어 작업을 수행할 수 있음.*

*multitasking을 사용하면 여러 process들에 대한 memory 공간을 관리/보호해야 하고, 필요에 따라 job을 memory에 올렸다가 disk에 내릴 수 있어야 하고, deadlock(2개 이상의 process가 서로의 자원에 대한 공급을 대기하게 되는 상태.) 등의 문제를 방지해야 하고, 적절한 scheduling이 필요함. 즉, 잘 동작하는 OS가 필요함.*

너무 작은 system이 아니라면 대체로 multitasking을 사용함.

### 1.2.5. Lineage of OS

***1. Lineage of OS***
*OS의 계보(Linage)는 아래와 같음. OS는 batch system으로부터 시작하여 Multics, UNIX, LINUX 그리고 Windows등이 존재하게 되었음.*




***2. 기타 system들***
*각 system에 따라 적절한 OS가 사용됨.*

*현재 대부분의 휴대폰, computer 등은 multiprocessor system을 사용함. 각 processor(core)가 동일한 기능을 수행하는 대칭(Symmetric) multiprocessor와, processor가 각각 주어진 기능을 수행하는 비대칭(Asymmetric) multiprocessor 등이 있음.*

*여러 개의 system이 LAN/WAN에 의해 연동되어 동작하는 분산시스템, cluster system 등도 존재함. server에서만 데이터를 전송하는 client-server model과, 각 system이 서로 데이터를 주고받는 P2P(Peer-to-Peer) model 등을 사용함.*

*hardware에 OS 또는 software가 탑재된 embeded system이 있음.*

*실시간으로 동작하거나 job의 완료 시간(Deadline)이 엄격히 정해져 있는 Real-Time system이 있음. deadline이 비교적 엄격한 것을 hard하다고 하고, 비교적 엄격하지 않은 것을 soft하다고 함.*

## 1.3. OS structure

### 1.3.1. OS design principle

*OS는 규모가 굉장히 크고 복잡하므로, 효율적인 개발/디버깅/유지보수/확장을 위해 OS의 사용 system에 따라 잘 설계하는 것이 중요함.*

*policy와 mechanism을 잘 분리하여 module화하여 설계하는 것이 OS design principle임.*

***1. Policy***
*정책(Policy)은 program이나 시스템의 동작 방식에 대한 정의임.*

*사용되는 system에 따라 안전/성능/보안/전력 소모 등의 측면에서 OS가 적절한 policy를 사용해야 함.*

***2. Mechanism***
*Mechanism은 policy의 구현 방법임.*

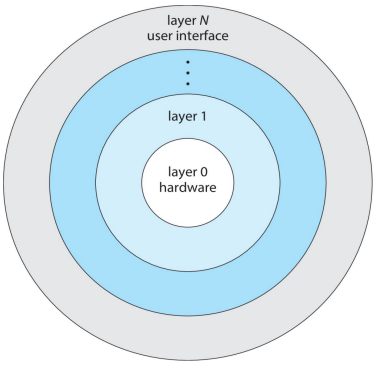
*알고리즘, 자료구조 등을 포함함.*

### 1.3.2. OS design methods

***1. Layering***
*Layering은 software를 여러 layer로 나누고, 각 layer는 인접한 layer와의 통신만 고려하여 독립적으로 관리하는 기법임.*

*OS는 매우 복잡한 software이므로 이를 단순화하기 위해 layering을 사용함.*

*설계와 구현의 복잡도는 낮아지지만, 각 layer 별 처리에 의한 overhead가 발생하거나, layer를 건너뛰어 접근하는 등 layering 원칙을 잘 지키지 않은 경우 보안 상 위험성이 존재할 수 있음.*




***2. Modularity***
*묘듈화(Modularity)는 software를 독립적이고 서로 상호작용하는 module로 분리하여 설계하는 기법임.*

*OS는 매우 복잡한 software이므로 이를 단순화하기 위해 modularity를 사용함.*

*kernel은 signal, file system, CPU scheduling, I/O system, Driver, VM 등 여러 기능들에 대한 module을 가지고 있음.*

### 1.3.3. System call

***1. CPU Execution Mode***
*CPU 실행 모드(CPU Execution Mode)는 CPU가 실행할 수 있는 명령어의 유형과 그 권한에 대한 mode임. 즉, mode에 따라 실행할 수 있는 명령어와 접근 가능한 메모리 등을 제한하여 system을 보호하는 기법임.*

*processor마다 다를 수 있지만, 아래의 2가지 mode로 구분할 수 있음.*

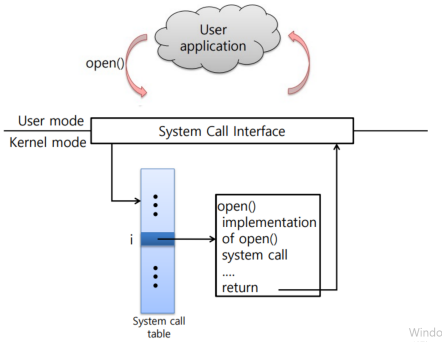
*1. Kernel mode : 모든 권한(root)을 가진 mode로, OS가 실행되는 mode임. Privilege 명령어(system 자원 제어 명령어) 실행, I/O 제어, memory/register 접근, scheduling 지정 등을 처리할 수 있음.*

*2. User mode : kernel mode에 비해 제한된 권한을 가진 mode로, application이 실행되는 mode임. Privilege 명령어를 실행할 수 없음.*

***2. System Call***
*시스템 콜(System Call)은 OS가 user mode에서 실행되는 program에 제공하는 인터페이스로, kernel mode의 권한이 필요한 작업(system 자원 제어 등)을 수행할 수 있도록 함.*

*user mode의 program에서도 kernel mode의 기능이 필요할 때가 있기 때문에 system call이 존재함.*

*kernel에는 system call table이 있고, 각 system call은 고유한 호출 번호를 가지고 있음. user mode에서 system call을 호출하면 해당 번호가 전달되어 적절한 system call이 호출됨.*




*아래와 같은 system call들이 있음.*

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

| | Windows | Unix |
|---|---|---|
| Process Control | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject() | fork()<br>exit()<br>wait() |
| File Management | CreateFile()<br>ReadFile()<br>WriteFile()<br>CloseHandle() | open()<br>read()<br>write()<br>close() |
| Device Manipulation | SetConsoleMode()<br>ReadConsole()<br>WriteConsole() | ioctl()<br>read()<br>write() |
| Information Maintenance | GetCurrentProcessID()<br>SetTimer()<br>Sleep() | getpid()<br>alarm()<br>sleep() |
| Communication | CreatePipe()<br>CreateFileMapping()<br>MapViewOfFile() | pipe()<br>shmget()<br>mmap() |
| Protection | SetFileSecurity()<br>InitializeSecurityDescriptor()<br>SetSecurityDescriptorGroup() | chmod()<br>umask()<br>chown() |

당연하게도 CPU execution mode는 software 뿐만 아니라 hardware(processor 등)적으로도 지원이 되어야 함.

### 1.3.4. Kernel design

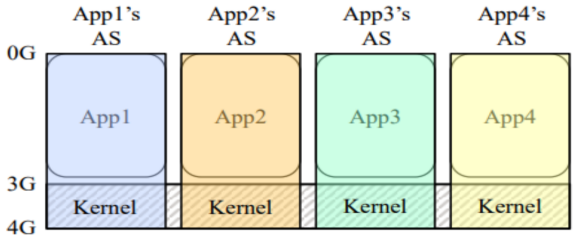
*kernel design 방식에는 아래와 같은 것들이 있음.*

***1. Monolithic kernel***

*Monolithic kernel은 kernel의 각 service들이 하나의 module에 들어 있고, 하나의 process로 실행되어 동일한 memory 공간을 사용하는 kernel임.*

*application이 memory에 올라오면 자신의 memory 공간에 kernel 코드 영역을 매핑해 kernel service를 사용함. 이때 kernel은 hardware에 대한 단일한 인터페이스를 제공하며, 각 application들은 동일한 kernel 코드를 매핑해 사용함.*

*application과 모든 kernel 기능이 같은 memory 공간에 위치하기 때문에 kernel service 이용과 각 service끼리의 호출에서 overhead가 적음. 하지만 kernel 기능 전체가 하나의 module에 들어 있어 유지보수가 어렵고, 중간에 버그가 발생하면 전체 kernel이 죽음.*



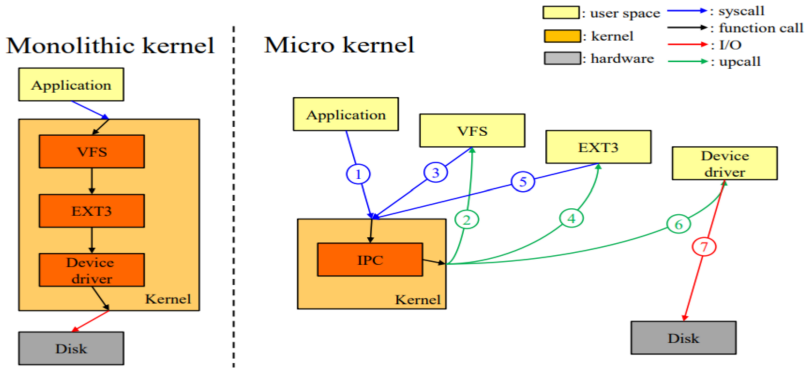
Monolithic 커널에서의 주소공간 매핑
(X86-32bit, ARM-32bit)

***2. Micro kernel***

*Micro kernel은 kernel의 각 service가 module화되어 있고, module별로 독립된 process로 실행되며 다른 memory 공간을 사용하는 kernel임. 이때의 각 모듈을 server라고 함.*

*IPC(Inter-Process Communication)는 각 process간 상호작용을 위한 module 또는 방법론임. 각 process끼리의 상호작용은 중간에 IPC를 거쳐야 함. micro server는 service call, system call 등의 단순한 기능만을 수행하고, 핵심 기능들은 IPC를 활용한 server 간의 통신을 통해 수행함.*

*service별로 module이 존재하기 때문에 개발과 유지보수가 편리하고, 각 service를 선택적으로 실행하고 사용할 수 있으므로 효율적임. 또한 버그 발생 시 해당 process만 종료하면 되므로 이론적으로는 monolithic보다 더 안정적임. 하지만 IPC를 통한 작업 수행 등에 의해 monolithic보다 성능이 비교적 낮음.*




***3. Hypervisor***

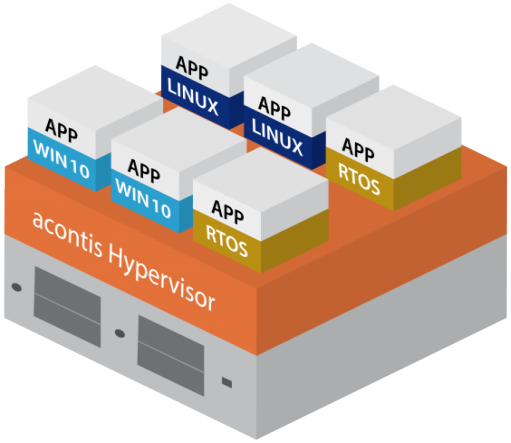
*Hypervisor는 가상화 시에 hardware 자원을 효율적으로 제공하기 위한 관리 계층으로, guest OS와 hardware 사이에 위치하여 각 guest OS에 자원을 분배함.*

*guest OS는 hypervisor에 의한 가상화 환경을 이용하는 OS임. 각 guest OS는 서로 다른 virtual machine에서 수행되며, 독립된 OS인 것처럼 동작함.*

*hypervisor는 주로 cloud service의 server computer 등에서 사용함.*

*하나의 computer에서 여러 guest OS를 독립적으로 실행할 수 있고, 실제 hardware와 호환되지 않는 (ISA가 다른 경우 등.) OS도 hypervisor에 의해 실행할 수 있음. 하지만 당연하게도 hardware를 직접 다루지 않기 때문에 성능이 비교적 떨어짐. 성능 개선을 위한 반가상화 등을 적용하기도 하지만, hardware에 종속적이게 되므로 구현 복잡도가 높음.*

*당연하게도 Hypervisor는 kernel 자체에 대한 design은 아니고, OS 환경에 대한 것이므로 함께 정리함.*



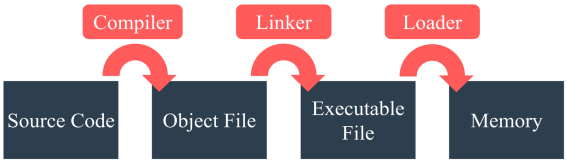

# 2. Process

## 2.1. Process Execution

### 2.1.1. Process Execution

*process는 program을 컴파일, 링크하고 실행하여 memory에 올리는 과정을 거쳐 생성됨.*

1. *source code를 compiler로 컴파일하여 object file을 생성함.*
2. *object file들을 linker로 link하여 executable file을 생성함.*
3. *executable file을 실행하여 loader를 통해 memory에 올림.*




### 2.1.2. Compiler

***1. Compiler***
*컴파일러(Compiler)는 PL(Programming Language)로 작성된 소스 코드로 구성된 소스 파일을 object file로 변환하는 software임.*

***2. Object file***
*목적 파일(Object file)은 cpu가 이해할 수 있는 기계어로 구성된 file임.*

*cpu마다 ISA가 존재하므로 이해할 수 있는 기계어의 종류가 다를 수 있음. 당연하게도 object file의 내용(기계어)은 해당 file을 처리할 cpu에 따라 다름. 즉, compiler는 cpu 종속적임. 다른 cpu에서 실행할 program이라면 cross compiler, translation 등을 사용해야 함.*

*object file은 자체적으로 실행이 불가능함. object file은 memory 주소가 상대 주소(Relocatable Address, Relative Address)로 되어 있고, 라이브러리 등이 포함되어 있지 않음. link를 통해 process로 변환되기 위한 정보가 삽입되어야 함.*

여기서 translation은 특정 cpu가 인식할 수 없는 기계어 집합을 이해할 수 있는 기계어 집합으로의 변환을 말함. 제한된 기능을 가지지만 apple의 Rosetta 등이 이를 수행할 수 있음.

### 2.1.3. Linker

***1. Linker***
*링커(Linker)는 object file들과 라이브러리들을 연결하여 하나의 executable file을 생성하는 software임.*

***2. Executable file***
*실행 파일(Executable file)은 실행하여 memory에 올릴 수 있는 file임.*

*executable file은 header, text, data 등을 포함함. header는 process로의 변환을 위한 정보를 가진 부분이고, text는 수행할 작업에 대한 코드, data는 작업 수행을 위해 필요한 데이터(전역/정적 변수)에 대한 부분임.*

*header와 사용 가능한 라이브러리는 OS에 따라 다름. 즉, linker는 OS 종속적임.*

*executable file에서는 memory 주소가 절대 주소(Absolute Address)로 되어 있음.*

즉, compiler와 linker는 cpu와 OS에 따른 내용을 가지게 됨.

### 2.1.4. Loader

*로더(Loader)는 executable file을 memory에 올리는 역할을 하는 software로, OS의 일부임.*

*loader는 아래와 과정을 거쳐 동작함.*

*1. executable file의 header를 읽어, text와 data에 필요한 memory 공간을 확인함.*
*2. program을 위한 address space를 확보함.*
*3. text와 data에 해당하는 내용을 확보한 address space에 복사함.*
*4. program에 들어온 argument들을 stack에 넣음.*
*5. cpu register를 초기화하고, program을 실행함.*

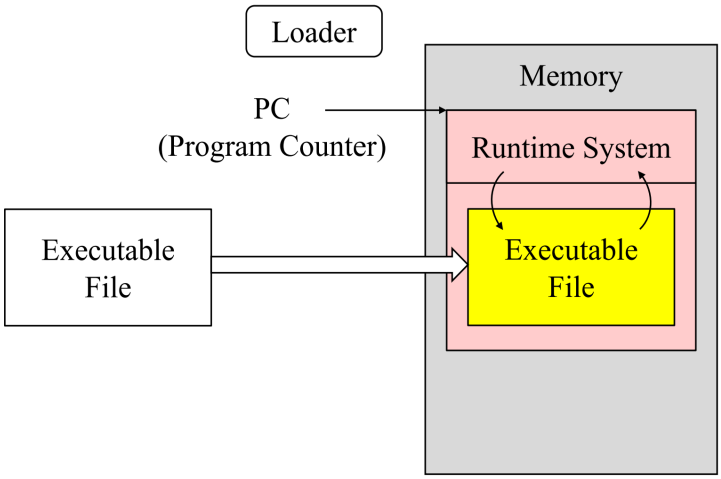
### 2.1.5. Runtime System

*런타임 시스템(Runtime System)은 program을 효율적으로 수행을 위해, program이 실행되었을 때 연*

결되어 상호작용하는 software임.

program이 사용하는 라이브러리 전체를 executable file에 포함시키면 file 크기가 너무 커지므로, run-time system이 메모리에 라이브러리를 올려 놓고 작업에 필요한 경우 동적으로 사용하도록 함.

gcc의 경우 link 시에 Start-up code라는 object file을 추가하고, 라이브러리들을 동적으로 링크함(executable file에 라이브러리 전체를 포함시키는 대신, 라이브러리 사용 관련 정보만을 가지도록 함.). program을 실행하면 _start 함수가 호출되고, _start 함수는 _libc_start_main이라는 라이브러리 등의 초기화 수행 함수를 호출한 뒤 program의 main 함수가 호출됨.




## 2.2. Process

### 2.2.1. Process

***1. Process***
프로세스(Process)는 memory에 올라가 실행 중인 program임.

OS는 process를 execution unit(scheduling의 단위)로 사용하고, 각 process는 protection domain으로 처리되서 각 process가 서로의 영역을 침범하지 못함.

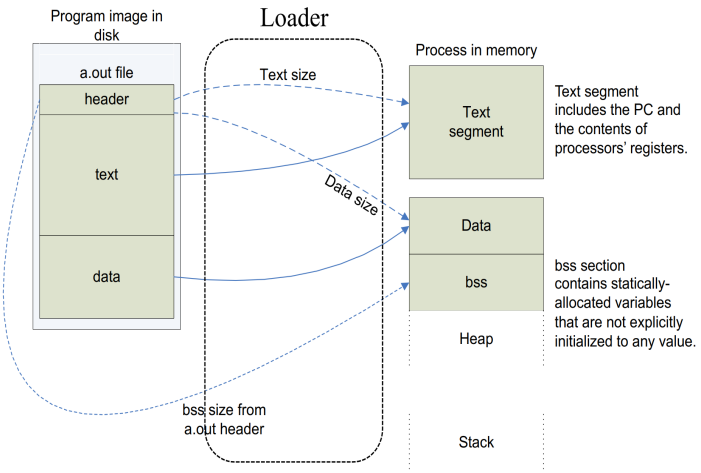
각 process들은 고유한 0이 아닌 정수 ID인 PID를 가짐.

***2. Process memory 구조***
process는 text, data, BSS, heap, stack의 memory 구조를 가짐.

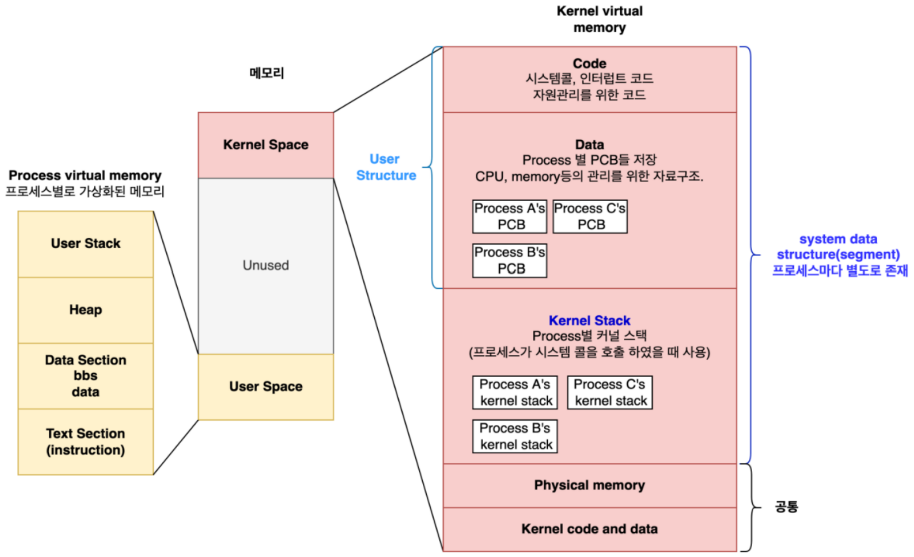
각 부분에 담긴 데이터는 아래와 같음.

text : 수행할 작업에 대한 코드.
data : 초기화된 전역/정적 변수.
BSS : 초기화되지 않은 전역/정적 변수.
heap : 동적 메모리 공간.
stack : 정적 메모리 공간.

data, BSS를 묶어서 data라고도 함.




*참고로 전체 memory 구조는 아래와 같음.*



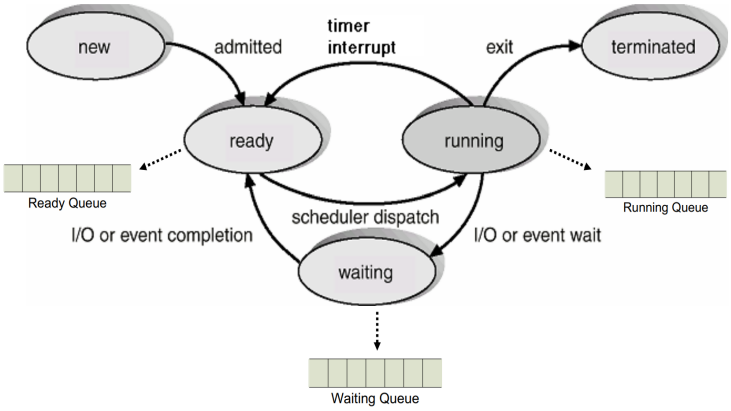

### *3. Process state*

*process는 아래와 같은 state를 가짐.*

*new : process가 생성된 상태.*
*running : process가 cpu에 의해 처리되고 있는 상태.*
*waiting : process가 특정 event의 발생을 기다리고 있는 상태.*
*ready : process가 cpu의 처리를 기다리고 있는 상태.*
*terminated : process의 실행이 끝난 상태.*

*waiting은 I/O 처리나 특정 signal에 의해 대기 중인 상태이고, ready는 모든 준비가 완료되어 cpu의 처리를 기다리는 상태임. 즉, waiting이 끝나도 cpu가 작업 중이라면 바로 실행되는 것이 아니라 ready 상태로 대기함.*

*각 state에 대한 queue가 존재하여 process들을 관리함.*



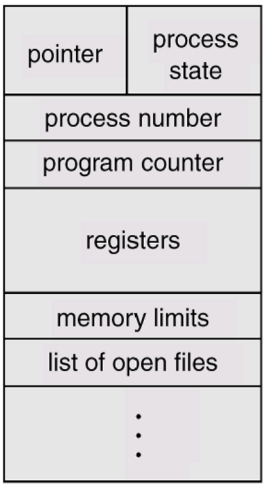

***4. Shared Memory***
*공유 메모리(Shared Memory)는 process들끼리 통신하는 방법 중 하나임. 이때 여러 process가 접근이 가능하기 때문에 적절한 동기화 기법이 적용되어야 함.*

cs에서 image는 데이터를 저장하고 있는 file이나 memory 상태를 나타냄.

### 2.2.2. Context Switching

***1. Process Control Block***
*Process Contorl Block(PCB)는 OS에서 process를 관리하기 위해 사용하는 데이터 구조임. OS는 kernel memory에 PCB를 저장하고 관리함.*

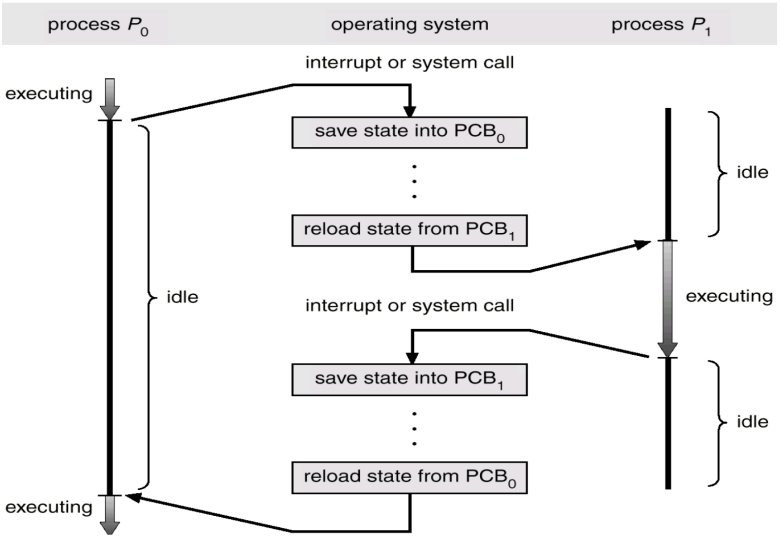
*PCB에는 process state, PC, cpu register, cpu scheduling information, memory management information 등 여러 정보가 저장되어 있음.*




***2. Context Switching***
*컨텍스트 스위칭(Context Switching)은 cpu가 처리할 process 간의 전환으로, 처리 중이던 process 정보를 PCB로 저장하고, 처리할 process 정보를 PCB에서 불러오는 과정임.*

*context switching에 의해 process들은 동시에(concurrently. 물론 cpu 입장에서는 실제로 동시에 처리되는 것은 아님.) 처리될 수 있음.*

*즉, 10ms마다 PCB에 저장하고 PCB로부터 불러오는 작업이 수행됨. PCB를 저장하고 불러오는 작업은 다른 작업과 병행이 불가능해 idle 상태가 발생하고, system call 등의 연산과 비교했을 때 오버헤드가 꽤 큰 연산임. 물론 hardware 성능에 따라 오버헤드 정도는 달라질 수 있음.*




CISC는 복잡한 instruction set을 사용하기 때문에 효율이 높고, 회로가 복잡하여 많은 레지스터를 사용하지 못함. 반면 RISC는 단순한 instruction set을 사용하기 때문에 속도가 빠르고, 물리적인 공간이 여유로워 많은 레지스터를 사용할 수 있음. 이에 따라 RISC에서는 PCB를 읽고 쓰는 작업 대신, register window라는 기법을 적용할 수 있음. 레지스터 자체를 많이 사용해 한 프로세스에 32개의 레지스터가 필요할 때 64개의 레지스터를 놔두고, 스위칭 시에는 그냥 다른 레지스터를 보기만 하면 되도록 구현할 수도 있음. 물론 레지스터는 비싸기 때문에 정말 성능이 중요한 시스템이 아니라면 이렇게는 잘 하지 않음.

## 2.3. Process Management

여기서는 UNIX의 일반적인 방식에 대해 정리함. xv6의 경우 또는 특정 OS의 경우 세부 내용은 다를 수 있음.

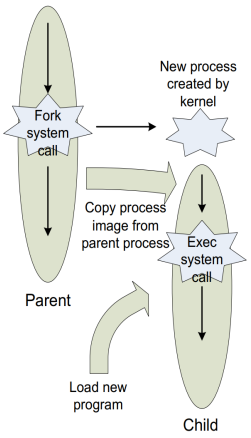
### 2.3.1. Process Creation

***1. Process Creation***
*fork() system call로 process를 생성할 수 있음.*

*fork()를 사용하면 해당 위치에서 parent process와 child process가 분기되어, fork()가 사용된 바로 다음 명령어부터 이어서 수행됨. parent process에서는 생성된 child process의 PID가 반환되고, child process에서는 0이 반환됨. fork() 작업에서 에러가 발생한 경우 0보다 작은 값이 반환됨.*

*exec 계열 system call로 해당 process를 특정 process의 정보로 덮어씌울 수 있음. 이때 process 자체가 (PID가) 변하는 것이 아니라, 해당 process가 다른 작업을 수행하게 되는 것임. 즉, 특정 작업을 부여하는 것.*

*exec 계열 system call들로는 execl(), execv(), execve(), execvp() 등이 있음.*




***2. Parent/Child process***
*fork()를 사용해서 process를 생성했을 때, 기존에 실행 중이던 process를 부모 프로세스(Parent Process), 새로 생성된 process를 자식 프로세스(Child Process)라고 함.*

*child process는 parent process로부터 메모리 공간을 복사함. 이때 데이터의 종류와 사용되는 기법에 따라 전부 공유될 수도 있고, 일부만 공유될 수도 있고, 아예 공유되지 않을 수도 있음.*

*parent process와 child process가 concurrent하게 실행되도록 할 수도 있고, child process가 종료될 때까지 parent process가 대기(waiting)하도록 지정할 수도 있음.*

*root process는 OS에서 최초로 실행되는 process임. OS 내부의 모든 process들은 root process의 child process로, 계층적 구조를 가짐.*

UNIX(xv6에서는 사용하지 않는다고 함.)에서 memory 복사에는 COW(Copy-On-Write)가 주로 적용됨. COW는 child process가 생성되면 child process는 parent process의 memory 공간에 대해 얕은 복사(Shallow Copy)를 해서 사용하고, child process에서 해당 memory 공간을 조작하면 그때 깊은 복사(Deep Copy)를 하는 기법을 말함. 이를 통해 memory를 절약할 수 있음.

### 2.3.2. Process Termination

***1. Process Termination***
*exit() system call로 process를 종료할 수 있음.*

*exit()을 사용하면 OS가 해당 process를 종료하고 해당 메모리 공간을 반납함.*

*wait()을 사용하면 해당 process의 child process가 종료될 때까지 기다리도록(waiting) 할 수 있음.*

*process에서 예기치 않은 에러가 발생했다면 SIGABRT signal을 parent process로 전송함.*

***2. orphan/zombie process***
*child process가 종료되면 parent process에서는 reaping이 수행됨. 수확(reaping)은 child process가 종료된 이후 parent process가 해당 child process의 종료 상태를 전달받는 것을 의미함. 종료 상태를 전달받은 parent process는 해당 child process의 system resource를 반납함. wait() 등을 통해 reaping을 수행할 수 있음.*

*고아 프로세스(Orphan Process)는 parent process가 먼저 종료된 child process임. parent process가 먼저 종료되면 orphan process는 kernel에 의해 root process의 child process로 입양됨.*

*좀비 프로세스(Zombie Process)는 종료되었지만 parent process에 의한 reaping이 수행되지 않은 process임. child proess는 종료 직후 zombie process가 됨.*

참고로, c언어 main 함수의 return은 내부적으로 exit() system call을 사용함.

## 2.4. IPC

### 2.4.1. IPC

*IPC(Inter-Process Communication)은 process들 간에 데이터를 주고받기 위한 기법임.*

*각 process들은 기본적으로 서로 침범할 수 없는 독립적인 memory 공간을 가지므로 process 협력 모델(Cooperating Process Model)을 위해선 IPC를 활용해야 함.*

*데이터를 주고받을 때는 해당 데이터를 memory에 올리는 등으로 처리하는데, 이를 위해서는 IPC는 kernel을 거쳐 수행되어야 함. process들은 system call을 사용하여 kernel을 통해 데이터를 주고받음.*

### 2.4.2. IPC model

*IPC에는 두 가지 모델이 있음.*

***1. Shared Memory***
*Shared Memroy는 여러 process에 대한 공유 memory를 사용하고, 해당 공유 memory에 대해 읽고 씀으로써 통신하는 방식임.*

*공유 memory를 설정할 때만 kernel이 관여하고 이후에 발생하는 통신에는 kernel이 관여하지 않음. 즉, memory 할당이 되었으면 단순히 변수에 읽고 쓰는 것처럼 application 수준에서 통신할 수 있음.*

*추가적인 memory 공간을 사용해야 하고, 공유 memory에 대한 잘못된 접근이 일어난다면 여러 문제가 발생할 수 있음. 하지만 공유 memory 할당 이후에는 kernel의 개입이 없으므로 message passing에 비해 빠르고 자유롭게 통신할 수 있음.*

*application에서 synchronization을 처리해야 함. kernel은 공유 memory 할당 이후부터 통신에 관여하지 않음. locking, semaphore 등으로 처리함.*

*database 등에서 주로 사용하는 방식임.*

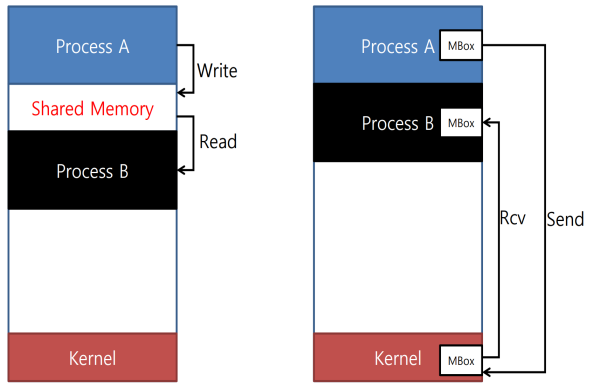
***2. Message Passing***
*Message Passing은 process간에 데이터(message)를 직접 주고받음으로써 통신하는 방식임. 주로 pipe, message queue, socket 등으로 구현됨.*

*어떤 process에서 보낸 message는 항상 kernel을 통해 다른 process에 전달됨. 이때 system call 등을 사용해 message를 주고받게 됨. message는 kernel을 매개로 이동하므로 kernel에서 message들에 대한 buffering 처리 등을 수행할 수 있음. 또한 kernel이 message를 처리한다는 것은 kernel의 특정 부분이 process로서 context switiching되어 cpu에 의해 처리되어야 한다는 것임.*

*매번 kernel을 통해 message를 전달하므로 성능상으로는 shared memory에 비해 떨어질 수 있지만, 보안 상으로는 유리할 수 있음.*

*kernel이 synchronization을 처리하고, application에서는 synchronization를 고려하지 않음. message가 항상 kernel을 거치므로 모든 책임을 kernel이 짐.*

*클라이언트-서버 형식의 시스템 등에서 주로 사용하는 방식임.*




Shared Memory 기법 Message Passing 기법

### 2.4.3. IPC 구현

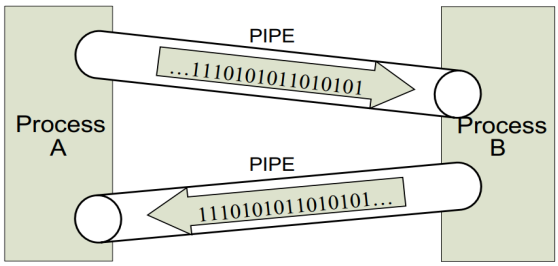
*IPC의 구현 방법으로는 아래와 같은 것들이 있음. 여기서의 코드는 xv6가 아니라 유닉스 계열 os에서 제공하는 sys/*.h 꼴의 헤더 파일을 사용함.*

*socket을 제외한 IPC들은 local system의 process들에 대한 IPC이고, socket은 local/remote system의 process까지 포함하는 IPC임.*

***1. Pipe***

*Pipe는 하나의 process가 다른 process로 데이터를 1대1로 전달하는 방식임.*

*하나의 pipe에서 데이터는 한쪽으로만 이동함. 즉, 하나의 pipe는 한쪽 process에서는 read/receive를, 다른 쪽에서는 write/send를 수행함. 이에 따라 양방향 통신을 하려면 2개의 pipe를 사용해야 함. 이때 데이터는 pipe에 넣은 순서대로 전달되고, 당연하게도 pipe가 꽉 차면 더 전송할 수 없음.*




*아래와 같이 코드로 구현할 수 있음. fd[0]에 대해서는 read/receive를, fd[1]에 대해서는 write/send를 수행함. 안 쓰는 부분은 close()로 닫아서 잘못 송수신하지 않도록 함. 이때 fd는 파일 디스크립터로, 입출력 자원에 접근할 때 사용하는 추상 식별자임. 즉, fd라는 pipe를 생성해 사용한 것임. 당연하게도 두 프로세스가 양방향 통신을 해야 한다면 fd1[2]와 fd2[2] 등으로 또 다른 파이프를 구성해야 함.*

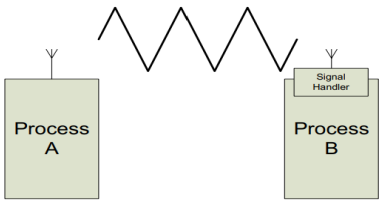
```
int main(void) {
    int fd[2];
    ...
    if((pid == fork()) == 0) {
        close(fd[0]);
        write(fd[1], "hello world!\n", 14);
        ...
    }
    else {
        close(fd[1]);
        read(fd[0], ...);
    }
}
```

***2. Signal***

*Signal은 하나의 process가 kernel을 통해 다른 process로 event를 비동기적으로 전달하는 방식임.*

*송신 process는 signal을 비동기적으로 전송하고, 수신 process는 전달받은 signal에 따라 signal handler에 정의된 동작을 수행함. 물론 수신 process가 해당 signal을 처리하려면 scheduler에게 선택되어야 함.*

*interrupt와 유사하게 처리됨.*




*코드 상에서는 아래와 같이 handler 함수를 정의 및 등록하고, kill()로 signal을 전송할 수 있음.*

```
void SignalHandlerChild(int signo)
{
    printf("signal handler\n");
    fflush(stdout);
}

int main(void)
{
    ... // handler 등록 등 수행
    switch(pid = fork()) {
        case 0 :
            sigpause(SIGUSR1);
        default :
            sleep(3);
            kill(pid, SIGUSR1);
            wait();
    }
}
```

***3. Shared Memory***

*Shared Memory는 두 개 이상의 process들이 하나의 memory 영역을 공유하여 통신하는 방식임. 앞에서 다룬 Shared Memory model을 단순 구현한 것.*

*아래와 같이 코드로 구현할 수 있음. shmget()으로 shared memory를 할당하고, shmat()으로 현재 프*

*로세스와 shared memory를 연결하고, 해당 shared memory를 사용하고, shcdt()로 현재 프로세스와 shared memory를 연결 해제함. 즉, 할당받은 shared memory의 shmId를 process들이 서로 공유하고 있다면 각각 attach하여 통신이 가능함. 물론 shared memory를 안전하게 사용하려면 접근 앞뒤로 lock을 걸어줘야 함.*

```
int shmId;
char *ptrShm;

// shared memeory 할당
shmId = shmget(IPC_PRIVATE, SHM_SIZE, SHM_R | SHM_W);

// shared memory attach
ptrShm = shmat(shmId, 0, 0);

// using shared memory
ptrShm[0] = 11;

// shared memeory detach
shmdt(ptrShm)
```

***4. Message Queue***

*Message Queue는 linked list로 고정된 크기의 message를 저장하여 통신하는 방식임. 이때 주고받는 message의 형태는 process들 사이에서 미리 정의해야 함.*

*여러 process들 간의 소통에 사용되고, 그에 따라 queue의 접근은 동기화 처리가 되어 있어야 함.*

*아래와 같이 코드로 구현할 수 있음. msgget()으로 message queue의 id를 얻을 수 있음. msgsnd()는 해당 id를 사용하여 queue에 message를 넣고, msgrcv()는 해당 id를 사용하여 queue로부터 message를 꺼내 읽음.*

```
typedef struct _MSG {
    long type;
    char message[256];
} MSG, *PMSG, **PPMSG;

int main(void)
{
    pid_t pid;
    key_t msg_id;
    MSG msg;
    msg_id = msgget(IPC_PRIVATE, 0660 | IPC_CREAT);

    switch ( pid = fork() ) {
        case 0:
            msg.type = 1;
            strcpy( msg.message, "Hello, world.");
            msgsnd( msg_id, &msg, MSG_-sizeof(long), 0 );
            return 0;
        default:
            waitpid(pid, 0, 0);
            memset( &msg, 0, MSG_ );
            msgrcv( msg_id, &msg, MSG_-sizeof(long), 1, 0 );
            ...
    }
    return 0;
}
```

***5. Socket***
*Socket은 통신에서의 end-point로, port를 이용하여 통신하는 데에 사용됨.*

*TCP/IP protocol 관점에서 socket은 application layer와 transport layer 사이의 인터페이스임. application에서는 하위 계층 대신 socket만을 고려하여 통신을 할 수 있음. socket에서는 port 번호, ip 주소 등만을 지정하면 통신이 가능함.*

*process의 위치(machine boundary)에 독립적임. 즉, process가 어떤 위치(local/remote)에 있든 주소만 안다면 접근이 가능함. local system의 process와 통신하는 경우 해당 process의 port 번호만으로 통신이 가능하고, remote system의 process와 통신하는 경우 접근하려는 system의 ip 주소와 해당 process의 port 번호가 필요함. local system의 process에 접근하는 상황에서는 port에 접근하는 과정에 의해 다른 IPC 방식보다 오버헤드가 큼.*

*process와의 연결 시에 사용하는 protocol에 따라 연결 방식을 지정할 수도 있음. (ex. TCP vs. UDP)*

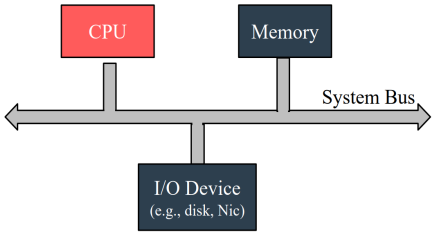
# 3. Computer Architecture

computer architecture 측면에서의 OS. 주로 I/O 처리를 어떻게 하는지에 대해 다룸.

## 3.1. Computer System의 구조

### 3.1.1. 단일 Bus 구조

***1. 단일 Bus 구조***
*cpu, memory, I/O device 등 각 모듈의 처리 속도가 비슷했던 computer system 초창기에는 단일 system Bus에 cpu, memory, I/O device등 여러 모듈이 연결된 방식을 주로 사용했음.*



<단일 버스 구조>

***2. 병목 현상***
*점점 cpu 속도가 memory, I/O device 등에 비해 월등히 빨라지게 되면서 병목 현상이 발생하게 되었음. 빠른 device는 느린 device의 처리를 기다리게(idle) 되어 전체 system 속도는 느린 device의 속도로 제한됨.*

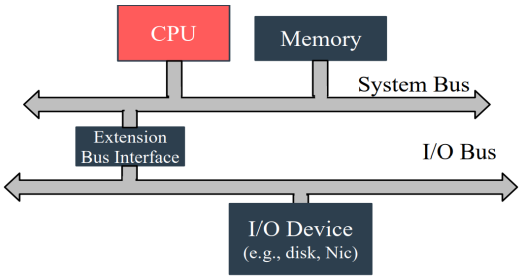
참고로 device 속도를 순서대로 나열해 보면, 레지스터, L1 캐시, L2/L3 캐시, DRAM, SSD임. 레지스터, L1 L3 캐시까지가 cpu에 해당되고, DRAM은 memory, SSD는 disk에 해당됨.

네트워크는 disk보다도 느림.

### 3.1.2. 계층적 Bus 구조

*계층적 Bus 구조에서는 Bus를 세분화(CPU local Bus, Memory Bus, PCI bus 등)하여 병목 현상을 개선하는 등의 성능 향상을 구현함.*

*비교적 빠른 cpu와 memory는 system bus에, 비교적 느린 I/O device(GPU, disk, NIC 등)는 I/O bus에 연결하여 사용하고, cpu, memory는 내부적으로도 bus를 사용함.*



<이중 버스 구조>

## 3.2. Event 처리 기법

device들에서 발생하는 event를 처리하는 기법에는 interrupt와 trap이 있음. computer system에서 event는 주로 문제 상황이나 특정 작업 요청에 의해 발생함.

### 3.2.1. 동기 vs. 비동기

*이해하기 쉽게 함수(system call)의 호출로 설명함.*

*blocking은 호출한 함수가 수행되는 동안 기존 함수(system call)이 block되는 방식이고, non-blocking은 block되지 않는 방식임.*

*동기(Synchronous)는 호출한 함수(system call)의 완료를 확인한 이후 다음 작업을 수행하는 방식으로, blocking으로 동작하고 한 번에 한 작업을 수행함. 비효율적이지만 구현이 편리함.*

*비동기(Asynchronous)는 호출한 함수(system call)의 완료를 확인하지 않고 다음 작업을 수행하는 방식으로, non-block으로 동작하고 여러 작업을 동시에 수행함. 효율적이지만 구현이 복잡함.*

### 3.2.2. Interrupt

***1. Interrupt***
*인터럽트(Interrupt)는 비동기적 event(ex. network packet, I/O 요청 등)를 처리하기 위한 기법임. 주로 process의 작업과는 관련 없는 event이고, cpu 외부의 I/O device에서 전송한 signal 등에 의해 발생함.*

*interrupt handler에 의해 처리됨.*

*OS에는 각 interrupt에 대한 ISR(Interrupt Service Routine)이 정의되어 있음. 특정 interrupt가 들어오면 해당 ISR을 수행함. 정의되어 있지 않은 interrupt가 들어오면 system을 다운시키거나 하지만, 웬만한 경우에 대해서는 정의가 되어있음. 한 interrupt가 들어오면 다른 interrupt의 진입이 막히기 때문에, ISR은 너무 길면 안됨.*

*interrupt 간에는 우선순위가 존재하여, 우선순위가 높은 것부터 처리됨. 실제로는 interrupt 처리 중에 우선순위가 더 높은 interrupt가 들어오면, 해당 interrupt를 우선 수행하고, 이후 다시 원래의 interrupt로 switching하는 식으로 동작함.*

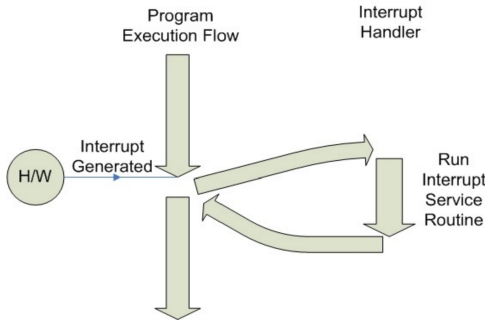
***2. Hardware/Software Interrupt***
*interrupt는 발생 주체에 따라 Hardware/Software로 나눌 수 있음.*

*hardware interrupt로는 timer interrupt, keyboard interrupt, disk(I/O) interrupt 등이 있고, software interrupt로는 system call에 의한 interrupt 등이 있음.*

*timesharing은 hardware interrupt를 사용해 구현됨.*

***3. Interrupt 처리 과정***
*interrupt의 처리는 아래의 과정을 거쳐 수행됨. 비동기적으로 event를 처리하기 때문에 현재 실행 상태(state)를 저장하고 복원함.*

*1. interrupt 발생.*
*2. interrupt 진입 비활성화(disable).*
*3. 현재 실행 상태(state) 저장.*
*4. 해당 interrupt에 대한 ISR 수행.*
*5. 저장한 실행 상태(state) 복원.*
*6. interrupt에 의해 중단된 시점부터 작업 이어서 수행.*



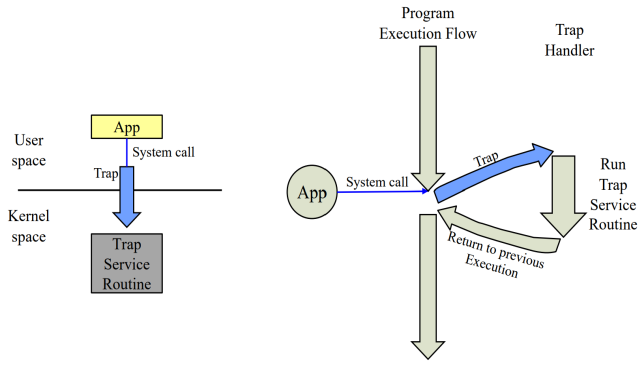

< Flow of handling interrupt >

### 3.2.3. Trap

*트랩(Trap)은 동기적 event(ex. program 에러 등)를 처리하기 위한 기법임. 주로 process의 작업과 관련이 있는 event로, system call이나 에러/예외 상황 처리 등에 사용됨.*

*trap handler에 의해 처리됨.*

*ISR 대신 TSR(Trap Service Routine)이 동작하는 등 interrupt와 유사하게 동작하지만, 동기적으로 event를 처리하기 때문에 현재 실행 상태(state)를 저장하고 복원하지 않음.*




참고로 프로그램 동작 중에는 exception이 발생할 수 있는데, 특징에 따라 abort, trap, fault 등으로 분류할 수 있음.

## 3.3. I/O Device

### 3.3.1. I/O Device

*I/O device는 computer system에서 cpu와 memory 관점에서의 입출력 device를 의미함. 키보드, 마우스, 모니터, 프린터, disk(SSD, HDD), modem, GPU, audio card 등이 있음.*

*각 I/O device들은 device register, I/O controller 등의 부분을 포함하고 있음.*

***1. device register***
*device register는 각 I/O device가 가지고 있는 레지스터로, 아래와 같이 4가지 종류로 구분됨.*

*control register : I/O device의 제어를 위한 레지스터. 특정 값을 넣으면 해당 작업이 수행되는 식으로 동작함.*
*status register : I/O device의 상태를 저장하는 레지스터.*
*input register : I/O device로의 입력값을 저장하는 레지스터.*
*output register : I/O device로부터의 출력값을 저장하는 레지스터.*

***2. I/O controller***
*I/O controller는 cpu의 high level I/O 명령을 device에 대한 low level I/O 명령으로 변환하여, I/O device를 직접 조작하여 작업을 수행하는 부분(회로)임.*

*비교적 복잡한 형태의 I/O device(ex. GPU, SSD 등)에서 주로 사용됨.*

### 3.3.2. I/O 처리 기법

*cpu가 I/O를 처리하는 기법에는 polling과 DMA가 있음.*

***1. Polling***
*Polling은 cpu가 I/O가 처리되었는지(event가 발생하는지)를 계속해서 확인하는 방법으로, event를 동기적으로 처리하는 방식이라고 이해할 수 있음.*

*모든 I/O 작업이 cpu의 처리에 의해 이뤄짐.*

*고전적인 방법으로, I/O device의 속도가 cpu에 비해 느리다면, cpu가 I/O 처리 여부를 계속 확인해야 하므로 오버헤드가 큼. 물론 I/O device의 속도가 충분히 빠른 system에서는 적절한 방법일 수 있음.*

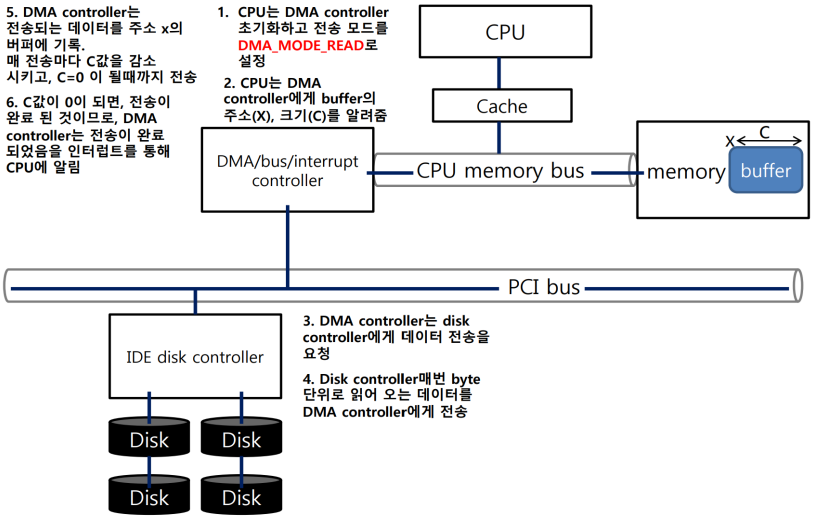
***2. DMA***
*DMA(Direct Memory Access)는 cpu의 개입 없이 memory와 I/O device 간에 데이터를 직접 전송하도록 한 기법임.*

*DMA controller(engine)을 사용하여, cpu가 memroy의 특정 부분(시작/종료 주소 또는 시작 주소와 크기 등)을 DMA controller에 전달하면 DMA controller가 cpu 대신 해당 memory 공간을 가지고 I/O device와의 데이터 전송을 제어하며 작업을 수행함.*

*DMA는 cpu 입장에서는 더 효율적이지만, 추가적인 hardware에 의한 cost는 존재함. 또한 I/O 작업을 DMA에 맡긴 뒤 cpu는 다른 작업을 수행하는 등 DMA를 잘 활용하려면 적절한 병렬화(Parallelism)가 되어 있어야 함.*

*데이터 입력은 아래의 과정에 따라 수행됨.*
*1. cpu는 DMA controller를 초기화하고 read 모드(DMA_MODE_READ)로 지정함.*
*2. cpu는 DMA controller에 memory 시작 주소와 크기를 알려줌. (cpu는 작업 끝.)*
*3. DMA controller는 disk controller에 데이터 전송 요청을 보냄.*
*4. disk controller가 데이터를 DMA controller에 전송함.*
*5. DMA controller는 전달받은 데이터를 memory에 작성함.*
*6. DMA controller는 해당 사이즈만큼 작성이 완료되면 완료 interrupt를 cpu에 전송함.*

*데이터 출력은 아래의 과정에 따라 수행됨.*
*1. cpu는 DMA controller를 초기화하고 write 모드(DMA_MODE_WRITE)로 지정함.*
*2. cpu는 DMA controller에 memory 시작 주소와 크기를 알려줌. (cpu는 작업 끝.)*
*3. DMA controller는 disk controller에 데이터를 전송한다는 신호을 보냄.*
*4. DMA controller는 memory에서 데이터를 읽어 disk controller에 전송함.*
*5. DMA controller는 해당 사이즈만큼 전송이 완료되면 완료 interrupt를 cpu에 전송함.*




### 3.3.3. I/O Device Access 기법

*cpu가 I/O device에 access하는 기법에는 I/O instruction을 사용하는 방법과, memory mapped I/O를 사용하는 방법이 있음.*

***1. I/O Instruction***
*I/O Instruction을 사용하는 방법은 단순히 cpu에서 I/O를 위한 명령어(instruction)을 사용하는 것임. 즉, cpu는 I/O를 위한 instruction을 사용하여 I/O device가 가진 register에 대해 비트 값을 읽거나 써서 I/O 작업을 수행할 수 있음.*

*CISC 기반의 computer architecture에서 주로 사용하는 방식임.*

***2. Memory Mapped I/O***
*Memory Mapped I/O는 device register들을 memory 공간에 매핑하여 사용하는 기법으로, cpu는 단순히 memory를 조작하는 instruction들로 I/O 작업을 수행할 수 있음.*

*매핑된 memory 부분에 대해 읽거나 쓰는 작업이 device register에 읽거나 쓰는 작업과 동일하게 처리됨.*

*cpu가 I/O를 더 빠르게 처리할 수 있고, 기본적인 instruction만으로도 처리가 가능하여 RISC 기반의 computer architecture에서 주로 사용하는 방식임.*

# 4. CPU Scheduling

## 4.1. CPU Scheduling

### 4.1.1. CPU Scheduling

***1. CPU Scheduling***
*CPU Scheduling은 ready 상태인 process들 중 어떤 process를 cpu가 수행할지를 결정하는 작업임.*

*cpu scheduling은 아래와 같은 기준으로 설계함. cpu 사용률과 처리량은 높게, 응답/대기 시간은 짧은 것이 이상적임.*

*CPU 사용률(CPU Utilization) : 전체 시스템 가동 시간과 cpu가 사용되는 시간의 비율.*
*처리량(Throughput) : cpu가 단위 시간 당 처리하는 process의 개수.*
*응답 시간(Response Time) : 요청 후 응답이 올 때까지의 시간.*
*대기 시간(Waiting Time) : process가 ready 상태로 대기하는 시간의 총합.*
*작업 시간(Turnaround Time) : process가 시작해서 끝날 때까지 거리리는 시간.*

***2. Burst Cycle***
*Burst Cycle은 process가 cpu 또는 I/O device를 독점적으로 사용하는 연속적인 시간 간격임. process의 burst cycle은 CPU Burst와 I/O Burst로 나눌 수 있고, 일반적인 process는 이 두 burst를 번갈아 수행함.*

*CPU Burst : CPU로 process에 대한 연산을 수행하는 구간. 이 구간에 대한 시간을 cpu burst time이라고 함.*
*I/O Burst : I/O 처리를 위해 process가 기다리는 구간. 이 구간에 대한 시간을 I/O burst time이라고 함.*

*cpu burst와 I/O burst의 분포에 따라 아래와 같이 process를 분류할 수 있고, 어떤 종류의 process가 많이 존재하는지에 부합하는 scheduling 기법을 사용함.*

*CPU-Bound Process : 짧은 주기의 긴 cpu burst를 가지는 process. (cpu 작업 위주.)*
*I/O-Bound Process : 긴 주기의 짧은 cpu burst를 가지는 process. (I/O 작업 위주.)*

*하지만 일반적인 경우에 대해 process들의 cpu burst time을 측정해 보면 대부분이 10ms 이내의 값을 가짐. 즉, I/O-Bound Process들이 대부분임. 그래서 범용적인 OS에서는 cpu 할당의 시간 단위를 10ms로 처리하는 것이 일반적임.*

***3. Scheduling의 종류***
*cpu scheduling의 종류는 어느 시점에 scheduling을 적용하는지에 따라 나눌 수 있음. cpu scheduling이 적용될 수 있는 시점으로는 아래와 같이 4가지가 있음.*

*1. process가 running에서 waiting으로 가는 경우. (I/O 발생)*
*2. process가 running에서 ready로 가는 경우. (scheduling 처리)*
*3. process가 running에서 작업이 완료된 경우. (작업 완료)*
*4. process가 ready에서 running으로 가는 경우.*

*비선점형 스케줄링(Non-preemptive Scheduling) : OS가 강제로 특정 process의 cpu 사용을 해제할 수 없는 방식. 즉, 1번과 4번에 대해서만 scheduling을 함.*

*선점형 스케줄링(Preemptive Scheduling) : OS가 강제로 특정 process의 cpu 사용을 해제할 수 있는 방식. 즉, 2번에 대해서도 scehduling을 함.*

시스템의 용도에 따라 요구사항과 scheduling에 대한 최적화 방향이 달라짐. 당연하게도 범용적으로 사용되는 OS일수록 최적화가 어려움.

### 4.1.2. Scheduling Algorithms

*Scheduling Algorithm에는 아래와 같은 것들이 있음. 여기에서의 scheduling은 processor/core가 1개인 것으로 가정함.*

*각 process 별 대기 시간을 구한 뒤, 이를 전부 더하고 process 개수로 나누어 평균 대기 시간(성능 지표로 활용 가능)을 계산할 수 있음.*

***1. FCFS Scheduling***
*FCFS(First-Come, First-Served) Scheduling은 먼저 cpu 할당을 요청한 process부터 처리하는 방식임. 비선점형으로 구현할 수 있음.*

*queue(ready queue)를 사용하여 간단히 구현할 수 있음. 이때 당연하게도 어떤 process를 먼저 수행하는지에 따라 평균 대기 시간이 달라짐.*

***2. SJF Scheduling***
*SJF(Shortest Job First) Scheduling은 cpu burst time이 짧은 process부터 처리하는 방식으로, 평균 대기 시간을 최소화하기 위한 기법임. 비선점형과 선점형으로 구현할 수 있음.*

*비선점형 SJF scheduling에서는 어떤 process의 처리가 완전히 끝나고 다음으로 처리할 process를 선택할 때 가장 짧은 cpu burst time을 가지는 것을 고름.*

*선점형 SJF scheduling에서는 어떤 process에 대한 처리 중에 다른 process들의 cpu burst time을 계속 확인함. 만약 현재 처리 중인 process의 남은 cpu burst time보다 짧은 값을 가지는 것이 있다면 cpu를 양보함.*

*선점형 SJF scheduling은 SRTF(Shortest Remaining Time First) scheduling이라고도 함.*

***3. Priority Scheduling***
*Priority Scheduling은 우선순위(priority)를 미리 정해두고 높은 priority를 가지는 process부터 처리하는 방식임. 비선점형과 선점형으로 구현할 수 있음.*

*비선점형 priority scheduling에서는 어떤 process의 처리가 완전히 끝나고 다음으로 처리할 process를 선택할 때 가장 높은 priority를 가지는 것을 고름.*

*선점형 priority scheduling에서는 새로 생성된 process가 현재 처리 중인 process보다 높은 priority를 가진다면 cpu를 양보함.*

*priority scheduling에서는 낮은 priority를 가지는 process가 수행되지 못할 수 있는데. 이렇게 특정 process가 계속해서 실행되지 못하는 경우 해당 process가 기아 상태(Starvation)에 있다고 함. priority scheduling에서는 대기 시간이 늘어남에 따라 priority를 증가시켜 starvation을 해결할 수 있는데, 이런 기법을 Aging이라고 함.*

***4. Round Robin Scheduling***
*Round Robin Scheduling은 미리 정해둔 시간 간격(Time Quantum)이 지나면 일단 현재 처리 중인 process를 cpu에서 내리는 방식임. 선점형으로 구현할 수 있음.*

*queue(ready queue)를 사용하여 구현할 수 있음. queue에서 process를 꺼내 처리하고, time quantum이 지나면 이를 queue(맨 뒤)에 넣음. 물론 process의 처리가 완료되었다면 단순히 다음 process를 꺼내 처리함.*

*ready queue 내의 process 개수를 $n$, time quantum을 $q$라 하면 각 process의 다음 처리를 위한 최대 대기 시간은 당연하게도 $(n-1) \times q$임.*

*만약 time quantum이 너무 크다면 FCFS와 동일하게 동작하게 되고, time quantum이 너무 작다면 context switching에 대한 비용이 더 커져서 비효율적임. 적절한 time quantum을 지정하는 것이 중요함.*

*round robin scheduling은 범용적인 OS에서 일반적으로 사용하는 기본 scheduling 기법임. 이때 time quantum은 $10 \sim 100ms$, 주로 10ms로 함.*

***5. Multilevel Queue Scheduling***
*Multilevel Queue Scheduling은 ready queue를 여러 개로 분리하여 각각에 대해 scheduling 기법을 적용하는 방식임.*

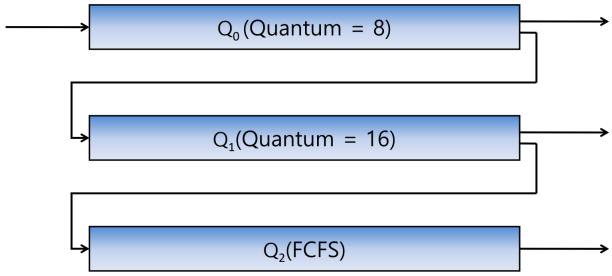
*queue를 적절히 나누어서 scheduling 기법을 적용하고, 각 queue에 대해 cpu를 잘 할당하는 것이 중요함. queue 자체에 대한 priority나 time quantum을 지정하여 사용기도 함.*

*이때 각 queue는 처리하는 process의 종류 등에 따라 분류함. 예를 들어, 사용자 등과의 interactive한 동작이 필요한 process를 처리하는 queue는 foreground queue라고 하고, round robin 기법을 사용함. 또한 cpu 연산 등의 작업을 수행하는 process를 처리하는 queue는 background queue라고 하고, FCFS 기법을 사용함. real-time process, normal process, batch process 등 process의 종류에 따라 queue를 사용하기도 함.*

***6. Multilevel Feedback Queue Scheduling***
*Multilevel Feedback Queue Scheduling은 multilevel queue에서 process들이 다른 queue로 이동할 수 있도록 하는 방식임. 즉, 피드백을 받아서 queue를 옮김.*

*이를 구현하려면 여러 개의 queue, queue별 scheduling 기법, process를 윗 단계로 옮길 조건, process를 아랫 단계로 옮길 조건, process를 다른 queue로 옮길 방법 등을 정의해야 함.*

*주로 cpu burst time이 긴 process를 적절한 queue로 옮겨 더 많은 처리 시간을 제공하는 식으로 구현됨. 즉, aging을 처리를 함. 예를 들어, 아래 그림과 같이 queue를 구성할 수 있음. $Q_0$에서 처리가 완료되지 않으면 $Q_1$로, $Q_1$에서도 처리가 완료되지 않으면 $Q_2$로 보내서 process가 더 많은 cpu time을 할당받을 수 있도록 함.*




범용적으로 사용되는 OS에서는 범용적인 상황에 적합한 scheduling 기법을 사용해야 함. 예를 들어, 리눅스에서는 모든 process를 균등하게 처리하는 CFS(Completely Fair Scheduling) 기법을 사용함.

### 4.1.3. Multiprocessor Scheduling

*processor/core가 여러 개인 경우에는 더 복잡한 scheduling 방식을 사용하게 됨. 특히 각 processor에 서로 다른 I/O 장치가 연결되어 있거나, 각 processor 별로 성능과 ISA가 다를 수 있는데 이때 각 작업을 어떤 processor에 할당할지를 결정해야 함.*

*아래와 같은 기법들을 사용함.*

***1. asymmetric multiprocessing***
*비대칭 멀티프로세싱(Asymmetric Multiprocessing)은 multiprocessor 시스템에서 각 processor가 서로 다른 역할을 수행하는 방식이고, 대칭 멀티프로세싱(Symmetric Multiprocessing)은 각 processor가 전부 같은 역할을 수행하는 방식임.*

*예를 들어, OS에 대한 작업(scheduling, I/O 처리 등)은 성능이 좋은 processor 하나가 우선적으로 전담해서 처리하는 식으로 구현함. 이 경우 processor끼리의 자원 공유를 줄일 수 있어서 효율적임. 대부분의 상용 시스템에서는 asymmetric multiprocessing을 사용함.*

***2. processor affinity***
*processor affinity(친화도)는 이전에 사용했던 processor/core를 계속해서 사용하는 기법임.*

*cache는 processor/core 별로 존재하므로, 특정 processor/core를 반복적으로 사용하면 cache의 효율을 높일 수 있음.*

***3. load balancing***
*load balancing은 여러 processor에 작업을 균등하게 분배하는 기법임.*

*processor마다 ready queue를 두도록 구현할 수도 있고, 하나의 ready queue만을 사용하여 processor에 process를 배정하도록 구현할 수도 있음. 이때 각 processor에서 처리하는 processor는 동적으로 다른 processor에 적절히 배정하여 균등하게 처리함.*

# 5. Thread

## 5.1. Thread

### 5.1.1. Thread

***1. Thread***
*스레드(Thread)는 process 내의 실행 흐름으로, process보다 작은 단위의 execution unit임.*

*thread는 병렬화(parallelism)을 위한 개념으로, 하나의 process에 대해 여러 개의 thread가 존재하는 multi-thread로 사용될 수 있음. 만약 하나의 process가 하나의 실행 흐름만을 가질 수 있다면, 병렬화를 위해서는 매번 새로운 process를 생성(cooperative process)해야 함. 이 경우 각 process끼리의 IPC가 필요하고, 매번 process 간의 context switiching이 수행되므로 오버헤드가 크기 때문에 thread라는 더 작은 단위로 병렬화를 구현함. 물론 이 경우 하나의 process에 대한 thread들은 process의 protection domain이 보장되지 않음.*

*물론 아예 다른 작업의 수행에 대해서는 새로운 process를 생성하는 것이 적절함. thread는 특정 process 내부 작업의 병렬화에 기여함.*

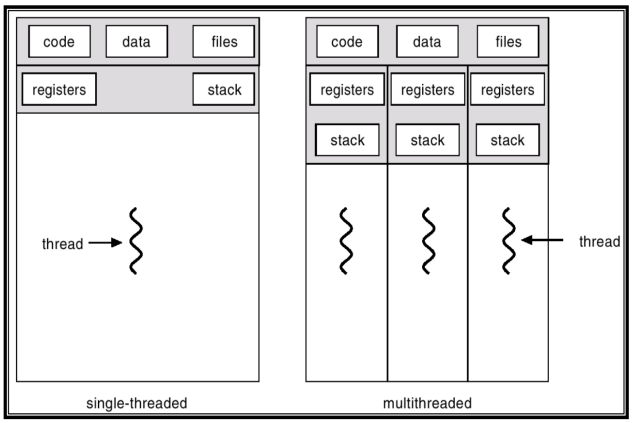
*앞에서는 scheduling과 그에 따른 core 할당을 process 단위로 설명했지만, 사실 multi-thread 환경을 가지는 대부분의 상용 os에서 이는 thread 단위로 수행됨.*

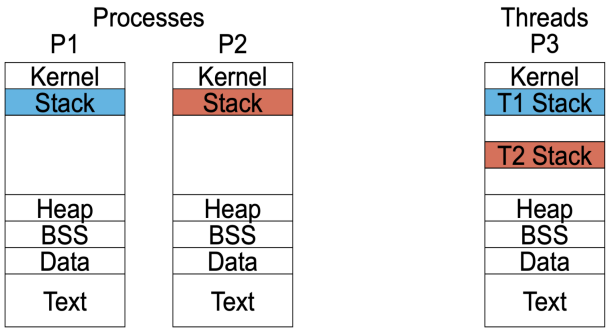
***2. Thread 구성요소***
*각 thread는 아래와 같이 실행과 관련된 정보를 독립적으로 가지고 있어야 함.*

*Thread ID : thread 식별 id.*
*PC(Program Counter) : 실행 중인 instruction 주소. (register의 일종이긴 함.)*
*Register Set : 사용 중인 register의 값들.*
*Stack*

*반면 하나의 process에 속하는 각 thread는 아래와 같은 부분을 공유함.*

*Code section : process의 code section.*
*Data section : process의 data section.*
*File : process와 연결된(열려 있는) file.*






당연하게도 하나의 process는 최소 하나의 thread를 가짐.

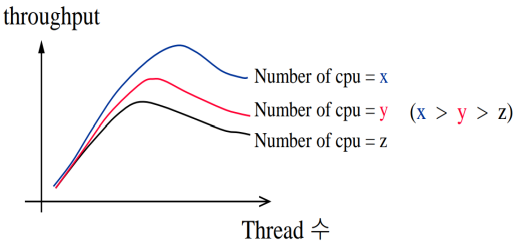
### 5.1.2. Multi-Thread Programming의 이점

*병렬화의 관점에서 multi-thread programming은 아래와 같은 장점이 있음.*

***1. CPU Utilization***
*각 thread는 서로 다른 cpu/core에 할당되어 병렬적으로 처리될 수 있으므로 multi-thread programming은 전체 cpu의 처리량을 높일 수 있음.*

*이때 thread 수를 늘리면 전체 cpu의 처리량은 계속 늘어나다가, thread 수가 임계치를 넘어가면 처리량이 다시 감소함. 이는 thread switching 등이 들어가는 비용이 커지기 때문임. 이때 cpu/core의 개수가 많을수록 임계치가 높아지고, 이에 따라 많은 thread를 사용하는 것이 유리함.*

*참고로, cpu/core의 개수를 엄청나게 늘려서 처리량을 대폭 늘린 것이 GPU임. GPU에서의 cpu/core는 일반적인 cpu보다 훨씬 단순하고 작기 때문에 이런 구현이 가능함.*




***2. Responsiveness***
*process의 일부 thread가 수행하는 작업이 오래 걸리거나 block되더라도, 다른 thread는 돌아가고 있으므로 process가 사용자에게 여전히 interactive함.*

***3. Economy***
*각각 독립된 memory 공간을 가지는 process와는 달리, process 내부의 thread들은 memory 공간을 공유하고 각자가 가지는 공간이 작으므로 switching, communication 비용이 적음. 또한 생성의 측면에서도 thread는 process보다 더 적은 공간을 할당하고 구성하기 때문에 비용이 적음.*

하나의 processor(chip)에 여러 개의 computing core를 가지고 있는 multicore processor에서 특히 multi-thread가 효율적임. 하드웨어적인 측면에서는 당연하게도 각 core에 thread를 할당하여 병렬적으로 처리할 수 있음. 소프트웨어적인 측면에서는 각 core가 processor의 cache를 공유하기 때문에 process의 data/code segment를 cache에 올리면 각 thread에 대한 처리가 빨라짐.

multithread/multiprocessor의 수행은 작업에 대한 동시성(Concurrency)을 확보할 수 있도록 함.

## 5.2. User/Kernal Thread

### 5.2.1. User/Kernel Thread

*thread는 그 관리 주체에 따라 user thread와 kernel thread로 나뉨.*

***1. User Thread***
*User Thread는 사용자 수준에서 관리되는 thread임. kernel의 간섭 없이 지원되며, 주로 사용자 수준의 라이브러리가 thread 생성, scheduling 등을 수행함.*

*kernel을 거치지 않으므로 속도가 빠르지만, user thread만을 사용해서는 실질적인 병렬 처리에는 한계가 있음.*

***2. Kernel Thread***
*Kernel Thread는 kernel(os) 수준에서 관리되는 thread임. kernel이 thread 생성, scheduling 등을 수행함.*

*kernel을 거치므로 user thread에 비해 속도는 느리지만, 실질적인 theading이 수행됨. 즉, 어떤 thread가 blocking되면 kernel은 다른 thread를 처리하도록 함.*

### 5.2.2. Mapping

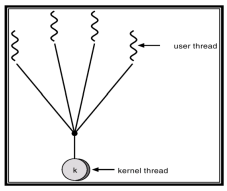
*user thread는 아래와 같은 방식으로 kernel thread와 매핑되어 수행됨.*

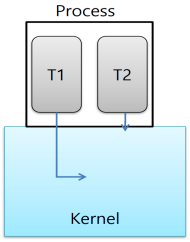
*user thead는 kernel thread와 매핑이 되어야 cpu에 의해 실질적으로 수행될 수 있음.*

***1. Many-to-One***
*여러 개의 user thread를 하나의 kernel thread에 매핑하는 방식. 정확히는 user thread의 라이브러리에서 여러 thread를 scheduling하여 여러 개의 thread가 병렬적으로 처리되는 것처럼 보이게 하는 것임.*

*kernel 입장에서는 하나의 thread를 사용하는 것과 같으므로, 한 번에 하나의 user thread만 kernel에 접근할 수 있음. 이때 하나의 user thread이 system call을 사용하거나 I/O에 의해 block된다면 다른 user thread이 처리되지 못함. 즉, 실질적으로는 thread들이 병렬적으로 처리되지 못함. (당연하게도 multiprocessor인 경우에도 마찬가지임.)*

*실제로는 I/O와 system call을 사용하는 경우가 자주 발생하므로 kernel thread를 지원하지 못하는 시스템에서 many-to-one 방식을 사용하고, 상용 시스템에서는 거의 사용하지 않음.*




T1이 system call을 수행하는 동안
T2는 커널 진입이 불가능

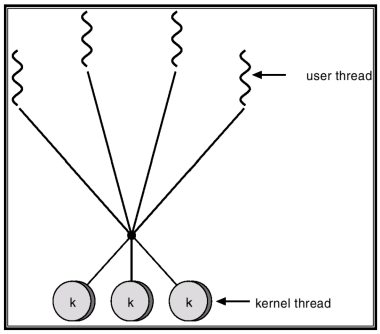
***2. One-to-One***
*하나의 user thread를 하나의 kernel thread에 매핑하는 방식. user thread가 생성되면 대응되는 kernel thread를 생성하는 것임.*

*어떤 user thread가 block되어도 다른 thread들은 계속 수행될 수 있음. 하지만 kernel thread를 항상 user thread 수만큼 생성하는 것은 비효율적일 수 있음.*

***3. Many-to-Many***
*여러 개의 user thread를 여러 개의 kernel thread에 매핑하는 방식.*

*여러 개의 user thread와 같거나 더 적은 수의 kernel thread를 사용하여(당연하게도 user thread보다 많을 필요는 없음.) many-to-one과 one-to-one의 문제점을 해결함. 이때 user thread와 kernel thread 사이의 매핑은 kernel이 처리함.*

*이때 어떤 user thread를 먼저 매핑하여 실행할 것인지는 나름의 scheduling 기법으로 결정됨. 즉, many-to-many에서는 user thread와 kernel thread의 매핑에서 scheduling이 수행되고, 이후 kernel thread의 cpu 할당에 대한 scheduling 또한 수행됨.*




## 5.3. Thread Issue

### 5.3.1. Thread Issue

*thread의 사용에 따라 고려해야 하는 사항들은 아래와 같음.*

***1. Creation***
*fork(), exec() 등을 활용한 process의 생성에서 thread를 고려해야 함.*

*process 내의 어떤 thread가 fork()를 호출하면 기존 process의 thread를 전부 가지고 있는 process를 생성할 것인지, fork()를 호출한 thread만을 가지고 있는 process를 생성할 것인지를 정의해야 함. (리눅스에서는 두 가지 버전의 fork()를 지원함.)*

*특히 exec()를 호출하면 기존 process의 데이터는 삭제되므로, exec()를 호출하지 않는다면 전자가, exec()를 호출한다면 후자가 합리적임.*

***2. Cancellation***
*thread cancellation의 처리를 고려해야 함. Thread Cancellation은 thread의 작업이 끝나기 전에 외부에서 작업을 중단시키는 것을 말함.*

*하나의 thread가 cancellation될 때 다른 연관된 thread도 cancellation해야 하는지, cancellation하려는 thread에 할당된 자원을 다른 thread에서 사용하고 있지는 않은지 등을 고려하여 cancellation해야 함.*

***3. Thread pool***
*Thread pool의 사용을 고려해야 함.*

*필요할 때마다 thread를 생성하고 사용이 끝났을 때 제거한다면, 생성과 제거가 빈번한 시스템에서는 오버헤드가 클 수 있음. 또한 한정된 자원에 따라 thread의 개수에는 제한을 둬야 함. 이에 따라 Thread Pool을 사용하여 thread를 미리 생성해 두고, process에 필요한 만큼 할당했다가 다시 반납받는 방식을 사용함. 이때 thread pool에서 관리하는 thread의 개수는 시스템에 따라 정해짐.*

***4. Thread 간 IPC***
*thread 간의 통신을 고려해야 함.*

*동일한 process 내부의 thread끼리는 data 영역을 공유하므로, shared memory 방식으로 통신하는 것이 효율적임. 다른 process의 thread와의 통신은 IPC를 활용해야 하는데, 이런 경우가 빈번하지 않도록 program을 설계해야 함.*

# 6. Synchronization

## 6.1. Synchronization

### 6.1.1. Synchronization

***1. Synchronization***

*동기화(Synchronization)는 공유 데이터에 여러 process/thread가 접근할 때 데이터의 일관성과 무결성을 유지하는 작업임. 즉, race condition을 처리하는 것.*

*한 process 내부 thread끼리의 synchronizatoin을 생각해 보면, 각 thread마다 별도의 register set과 stack을 가지기 때문에 지역변수에 대해서는 고려할 필요가 없음. thread끼리 공유하는 전역 변수(data), 동적 할당된 메모리(heap)에 대해서는 고려해야 함.*

*synchronization 기법을 적용할 때는 consistency가 확보되는지, deadlock 또는 starvation이 발생하는지, concurrency를 얼마나 제공하는지를 확인해야 함.*

***2. Race Condition***

*Race Condition은 공유 데이터에 대해 여러 process/thread가 접근/변경을 시도하는 상황을 말함.*

*race condition을 잘 처리하지 못하면(synchronization 처리를 하지 않으면) program의 결과가 처리 타이밍에 따라 다르게 결정될 수 있고, 이는 데이터에 대한 일관성(Consistency)를 해칠 수 있음.*

*예를 들어, 아래와 같이 balance라는 공유 변수를 거의 동시에 수정하는 두 process가 있다고 하자. 기대되는 결과는 1000이지만, 타이밍/scheduling에 의해 실제로 반영되는 값은 예시처럼 500이거나, 1500이거나, 본래 의도대로 1000일 수 있음.*

| < Process A : incoming> |
|---|
| $Register_1$ = Balance |
| $Register_1$ = $Register_1$ + 500 |
| Balance = $Register_1$ |

| < Process B : outgoing> |
|---|
| $Register_2$ = Balance |
| $Register_2$ = $Register_2$ - 500 |
| Balance = $Register_2$ |

| | | |
|---|---|---|
| $T_0$: A process execute | $Register_1$=Balance | [$Register_1$=1000] |
| $T_1$: A process execute | $Register_1$=$Register_1$ + 500 | [$Register_1$=1500] |
| $T_2$: B process execute | $Register_2$=Balance | [$Register_2$=1000] |
| $T_3$: B process execute | $Register_2$=$Register_2$ - 500 | [$Register_2$= 500] |
| $T_4$: A process execute | Balance =$Register_1$ | [Balance =1500] |
| $T_5$: B process execute | Balance =$Register_2$ | [Balance = 500] |

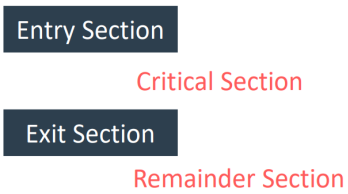
동시성(Concurrency)과 일관성(Consistency)는 tradeoff 관계에 있음. 물론 consistency가 더 중요하므로, consistency를 보장하면서 concurrency를 확보하도록 설계하는 것이 중요함.

### 6.1.2. Critical Section

***1. Critical Section***

*Critical Section은 여러 process/thread가 공유 데이터에 접근하는 code 영역임. 즉, race condition이 발생할 수 있는 부분임.*

*critical section으로 진입하는 부분을 Entry Section, 나오는 부분을 Exit Section이라고 함. 또한 critical section이 아닌 부분을 Remainder Section이라고 함.*




*synchronization은 한 번에 하나의 process/thread만이 critical section에 접근할 수 있도록 하는 것임. 즉, concurrency를 포기하고 consistency를 확보함. 이에 따라 concurrency를 최대한 확보하려면 critical section은 가능하면 짧게 하는 것이 좋음.*

***2. Critical Section의 해결 조건***

*알고리즘이 critical section을 잘 처리하려면 아래의 조건을 만족해야 함.*

*1) 상호 배제(Mutex, Mutual Exclustion)*
*어떤 process/thread가 critical section에 진입해 있다면, 다른 process/thread는 진입할 수 없어야 함.*

*즉, 한 번에 하나만 들어가야 함.*

*2) 진행(Progress)*
*critical section에 진입한 process/thread가 없고, 진입하려는 process/thread가 존재한다면 remainder section에서 어떤 process/thread가 진입할지를 유한한 시간에 결정할 수 있어야 함.*

*즉, 진행이 되어야 함.*

*3) Bounded Waiting*
*process/thread가 진입하는 데에 걸리는 시간에 제한이 존재해야 함.*

*즉, 무한히 기다리는(starvation) 상황이 존재해서는 안 됨.*

## 6.2. Synchronization 처리 방식

peterson solution, atomic instruction, semaphore, monitor 등의 처리 방식이 존재함.

### 6.2.1. Peterson Solution

*공유 변수의 사용에 따른 peterson solution이 synchronization을 어떻게 달성하는지 살펴보자.*

***1. 공유 변수 turn 사용***
*공유 변수 turn을 사용하여 어떤 process(정확히는 process/thread이지만 여기서부턴 process로 작성함.)의 차례인지를 나타냄. 아래와 같이 turn 값이 0일 때는 $P_0$이, 1일 때는 $P_1$이 접근하도록 함.*

```
// 공유 변수
int turn = 0; // 0으로 초기화
```

```
// P0
while(turn != 0) ; // waiting
... // critical section
turn = 1;
... // remainder section
```

```
// P1
while(turn != 1) ; // waiting
... // critical section
turn = 0;
... // remainder section
```

*mutex를 만족하지만, progress, bounded waiting이 처리되지 않음. 생각해 보면 항상 두 process가 번갈아 수행되어야 함. 예를 들어, turn 값이 1인 경우 $P_1$이 수행되기 전까지는 $P_0$이 수행될 수 없음.*

***2. 공유 배열 flag 사용***
*공유 배열 flag를 사용하여 어떤 process가 critical section에 접근하려고 하는지를 나타냄. 아래와 같이 각 process에 대해 접근하려는 상황이면 true를, 그렇지 않으면 false를 지정함.*

```
// 공유 변수
boolean flag[2] = { false, false };
```

```
// P0
flag[0] = true;
while(flag[1]) ; // waiting
... // critical section
false[0] = false;
... // remainder section

// P1
flag[1] = true;
while(flag[0]) ; // waiting
... // critical section
flag[1] = false;
... // remainder section
```

*mutex를 만족하지만, progress, bounded waiting이 처리되지 않음. 당연하게도 두 process 모두 flag가 true이면 양쪽 모두 접근하지 못함(deadlock이 걸림).*

***3. Peterson Solution***

*Peterson Solution은 turn과 flag를 모두 활용하는 방식임. 위에서와 같이 turn은 어떤 process의 차례인지를 나타내고, flag는 어떤 process가 접근하려고 하는지를 나타냄.*

*peterson solution은 공유 데이터에 접하려는 process가 총 2개이고, 한 번에 하나의 process만 접근이 가능한 상황에 대한 알고리즘임.*

*아래와 같이 코드로 구현됨. peterson solution은 flag와 유사한데, turn 값을 먼저 지정한 process가 우선 접근하도록 하여 deadlock을 해결했음. 서로 반대편의 turn을 지정하고 있는데, 마지막에 어떤 값이 배정되는지를 생각해 보면 그렇게 구현한 것을 이해할 수 있음. 정리하면, 상대편 process가 접근하려고 하지 않는 상황이면 flag값에 의해 그냥 접근하게 되고, 상대편 process가 접근하려고 하는 상황이면 turn을 지정하는 코드에 먼저 도달한 process가 접근하게 됨.*

```
// 공유 변수
int turn;
boolean flag[2] = { false, false };
```

```
// P0
flag[0] = true;
turn = 1;
while(flag[1] && (turn == 1)) ; // waiting
... // critical section
false[0] = false;
... // remainder section
```

```
// P1
flag[1] = true;
turn = 0;
while(flag[0] && (turn == 0)) ; // waiting
... // critical section
flag[1] = false;
... // remainder section
```

*mutex, progress, bounded waiting를 모두 만족함.*

*peterson solution은 2개의 process에 대해서는 잘 작동하지만, process가 2개보다 많은 경우는 처리하지 못함.*

***4. Peterson Solution의 확장***
*peterson solution을 2개보다 많은 process에 적용 가능하도록 확장하는 것은 단순하지 않음. 이를 소프트웨어적으로 해결하기에는 scheduling, compile 등 여러 변수가 존재함. 특히, 단순히 c언어로 코드를 작성하면 compiler에 의해 기존 코드가 의도한 것과는 다르게 기계어로 변환될 수 있으므로, 기계어 수준에서 명령을 구성해야 함. 또한 확장한 알고리즘이 모든 경우에 성립하는지에 대한 증명 자체도 까다로울 수 있음.*

*반면에 하드웨어적으로는 interrupt를 차단하거나 전용 instruction을 제공하는 등으로 비교적 간단하게 구현할 수 있음. 이때 application이 interrupt를 조작하는 것이 적절하지 않고, 접근하려는 process의 수가 많아지면 interrupt가 오래 비활성화되는 문제가 생김. 그래서 architecture마다 제공하는 synchronization instruction을 사용함.*

### 6.2.2. Synchronization Instrucstion

***1. Synchronization Instrucstion***
*architecture(cpu)에서 제공하는 Synchronization Instruction을 활용하여 synchronization을 처리할 수 있음.*

*이때의 instruction들은 원자적으로(atmoically) 동작하는 Atomic Instruction임. atomic하게 동작한다는 것은 해당 작업들 사이에 다른 작업이 scheduling 등에 의해 끼어드는 것을 허용하지 않는 것을 말함.*

*synchronization instruction의 사용은 공유 데이터에 접하려는 process가 총 n개이고, 한 번에 하나의 process만 접근이 가능한 상황에 대한 처리 방식임.*

*synchronization instruction의 사용은 mutex만을 보장함. 즉, 나머지 조건은 program에서 알아서 처리할 수 있어야 하는데, 이게 항상 보장되기는 어려움.*

***2. Instruction 예시***
*예시1) 아래와 같은 코드로 구성되어 있고 cpu에 의해 해당 함수 전체가 atomic하게 수행되는 TestAndSet()을 생각할 수 있음. lock이 true이면 계속 true를 반환하고, lock이 false이면 false를 반환하고 lock을 true로 지정함.*

```
// atomic instruction
boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target = true;
    retunr rv;
}

// 공유 변수
boolean lock = false;
```

```
// P_i
while(TestAndSet(&lock)) ;
... // critical section
lock = false;
... // remainder section
```

*이를 circular queue를 사용하여 최대 대기 시간이 전체 개수 만큼의 process가 처리되는 시간이 되도록*

*예시2) cpu에 의해 해당 함수 전체가 atomic하게 수행되는 Swap()을 생각할 수 있음. Swap()은 단순히 인자로 받은 두 값을 바꿔서 배정하고, waiting 배열의 값은 해당 process가 접근을 대기 중임을 나타냄(flag와 동일한 기능). lock이 true이면 waiting 값이 계속 true이고, lock이 false이면 waiting 값이 false가 되고 lock이 true가 되면서 critical section에 접근함.*

```
// 공유 변수
boolean lock = false;
boolean waiting[n]; // # of process
```

```
// P_i
waiting[i] = true;
while(waiting[i] == true) swap(&lock, &(waiting[i]));
... // critical section
lock = false;
... // remainder section
```

#### 6.2.3. Monitor

*Monitor는 공유 데이터(필드)와 해당 데이터로의 접근 코드(매서드)를 캡슐화하는, high-level language에서의 synchronization 방식임.*

*접근하려는 process를 queue에서 대기시켜, 한 번에 하나의 process만이 매서드로 공유 데이터에 접근할 수 있도록 함. 이에 따라 user program에서는 P/V 등의 고려 없이 단순히 메서드를 사용하기만 하면 됨.*

*java 등 여러 high-level language에서 관련 문법을 제공함. java에서는 하나의 class에 대해 synchronized 키워드를 사용하면 monitor로 처리됨.*

## 6.3. Semaphore

semaphore는 내용이 많아 따로 빼서 정리함.

#### 6.3.1. Semaphore

***1. Semaphore***
*세마포어(Semaphore)는 두 개의 atomic operation을 가지는 정수 변수임. 구현 방식에 따라 다를 수 있지만, semaphore 값은 critical section에 현재 접근 가능한 process의 개수를 의미함.*

*그럼 semapore는 한 번에 접근 가능한 process의 개수를 늘린 방식일 뿐, 앞에서 설명한 방식과 마찬가지로 항상 완전한 동기화를 보장하지는 못함. 적절히 구현하지 못하면 starvation 등이 발생할 수 있음.*

*semaphore는 공유 데이터에 접하려는 process가 총 n개이고, 한 번에 여러 개(구현 방식에 따라 개수가 정해짐)의 process가 접근이 가능한 상황에 대한 처리 방식임.*

***2. 연산***
*semaphore 아래와 같은 연산을 가지며 이 연산들로만 접근이 가능함. 이때 각 연산은 atomic하므로, 당연하게도 하나의 process에서 P()나 V()를 호출 중이라면 다른 process에서는 P()나 V()를 호출하지 못함.*

*Wait()/P() : critical section 접근 직전에 호출. semaphore 값을 1 줄임.*
*Signal()/V() : critical section 접근 직후에 호출. semaphore 값을 1 늘림.*

*즉, 코드 상에서 아래와 같이 사용됨.*

```
P();
... // critical section
V();
... // remainder section
```

***3. 단점***
*Semaphore는 잘못 사용할 경우 deadlock이 발생할 수 있고, P()와 V()를 매번 사람이 작성하므로 human error가 발생할 수 있다는 단점이 있음. 이는 high-level language에서 monitor 등을 사용하여 synchronization을 처리하면 해결할 수 있음.*

*만약 P()는 작성했는데 V()를 작성하지 않았다면 lock을 풀어주는 부분이 없으므로 deadlock이 발생하고, V()는 작성했는데 P()를 작성하지 않았다면 lock을 걸어주는 부분이 없으므로 mutex를 만족하지 못함.*

*Deadlock은 두 개 이상의 process들이 수행되지 못하고 계속해서 이벤트 대기 상태에 머무르는 현상을 말함. 예를 들어, 두 process가 서로의 작업이 완료되기를 기다리고 있으면 어느 한 쪽도 작업이 완료되지 못해 deadlock이 걸림.*

### 6.3.2. Semaphore의 구현

*semaphore의 구현 방식에는 두 가지가 있음.*

***1. Busy Waiting***
*Busy Waiting은 critical section에 접근할 수 있는 자리가 없을 때, 자리가 날 때까지 loop를 돌며 계속 기다리도록 하는 방식임.*

*아래와 같이 구현될 수 있음.*

```
P(S) {
    while(S <= 0) ;
    S = S - 1;
}
V(S) {
    S = S + 1;
}
```

*작업 중이 아님에도 반복문에서 비교 연산을 계속해서 수행하므로 cpu에 부담이 감. 또한 다음에 접근할 process가 연산 타이밍에 의해 결정되므로 어떤 process가 접근할지 알 수 없음.*

***2. Sleep Queue***
*Sleep Queue는 critical section에 접근할 수 있는 자리가 없을 때, 자리가 날 때까지 process가 sleep하도록 하고 그 정보를 sleep queue에 저장하는 방식임. 자리가 나면 sleep queue에서 process를 꺼내 접근하도록 함.*

*아래와 같이 구현할 수 있음.*

```
typedef struct {
    int value;              // semaphore value
    struct process *list;   // sleep queue
} semaphore;

P(semaphore *S) {
    S->value--;
    if(S->value < 0) {
        add this process to S->list;
        sleep();
    }
}

V(semaphore *S) {
    S->value++;
    if(S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

*코드는 복잡해지지만 cpu 부담을 덜 수 있음.*

### 6.3.3. Semaphore 종류

*semaphore가 가질 수 있는 값에 따라 semaphore의 종류를 나눌 수 있음.*

***1. Binary Semaphore***
*Binary Semaphore는 semaphore가 0과 1만을 값으로 가질 수 있는 방식임.*

*binary semaphore는 하드웨어에 의존하여 atomic instruction을 사용하여 구현할 수 있음. 아래와 같이 TestAndSet()을 사용하여 구현할 수 있음. S는 semaphore로, critical section에 접근한 process가 있으면 true, 없으면 false를 값으로 가짐.*

```
P(boolean *S) {
    while(TestAndSet(&S)) ;
}
V(boolean *S) {
    S = false;
}
```

*단순히 TestAndSet()을 활용한 synchronization과 다르지 않음.*

***2. Counting Semaphore***
*Counting Semaphore는 semaphore가 가질 수 있는 값이 정해져 있지 않고, 한 번에 접근 가능한 process의 수를 초기값으로 가지는 방식임.*

*counting semaphore는 아래와 같이 binary semaphore를 활용해 구현할 수 있음. counting semaphore C와, binary semaphore S1, S2를 사용함. 여기에서의 binary semaphore는 위의 구현 예시와는 다르게 남아 있는 자리의 개수를 저장함. S1은 mutex를 보장하기 위한 것이고, S2는 자리가 없는 경우 waiting하도록 지정하기 위한 것임. wait()과 signal()은 counting semaphore의 연산이고, P()와 V()는 binary semaphore의 연산임.*

```
c = 5;      // 한 번에 접근 가능한 process의 수 지정
S1 = 1;
S2 = 0;
```

```
Wait() {
    P(S1);
    C--;
    if(C < 0) {
        P(S2);      // S2가 0이므로 접근 가능한 자리가 없다면 waiting
    }
    V(S1);
}

Signal() {
    P(S1)
    C++;
    if(C <= 0) {
        V(S2)
    }
    V(S1)
}
```

*즉, binary semaphore만을 지원하는 시스템에서도 이를 이용하여 counting semaphore를 구현할 수 있음.*

## 6.4. Synchronization 관련 고전 문제

synchronization 관련 고전 문제를 살펴보자.

### 6.4.1. Bounded Buffer Problem

***1. Bounded Buffer Problem***
*Bounded Buffer Problem은 n개의 item을 삽입할 수 있는 buffer로 여러 producer와 consumer가 접근 가능한 상황의 synchronization을 처리하는 문제임.*

*producer는 하나의 item을 생성해 buffer에 저장하려는 대상이고, consumer는 buffer로부터 하나의 item을 가져오려는 대상임. 여러 producer와 consumer가 buffer에 접근하려고 할 때, 한 번에 하나만 접근이 가능하도록 synchronization을 처리해야 함.*

***2. Solution***
*단순히 semaphore를 사용하여 해결할 수 있음. 아래와 같은 3가지 semaphore를 사용함. buffer의 상태 관리를 위한 semaphore임.*

*empty : buffer에서 비어 있는 자리의 개수를 나타냄.*
*full : buffer에서 차 있는 자리의 개수를 나타냄.*
*mutex : mutex 처리를 함.*

*아래와 같이 구현할 수 있음. mutex 외에도 producer에서는 buffer가 꽉 찬 경우에, cosumer에서는 buffer가 비어 있는 경우에 접근을 대기하게 됨.*

```
empty = n;  // buffer 공간의 전체 개수 배정
full = 0;
mutex = 1;  // 한 번에 하나만 접근하도록 함
```

```
// producer
while(1) {
    ...
    P(empty); // buffer에 자리가 있으면 통과
    P(mutex);
    ... // buffer에 item 넣음
    V(mutex);
    V(full);
    ...
}
```

```
// consumer
while(1) {
    ...
    P(full); // buffer에 item이 있으면 통과
    P(mutex)
    ... // buffer에서 item 꺼냄
    V(mutex);
    V(empty);
    ...
}
```

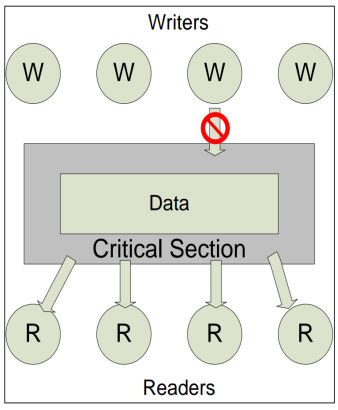
참고로, c에서 pthread.h, semaphore.h 등을 사용해서 구현할 때는 mutex와 empty/full의 사용 함수가 다름. mutex에는 pthread_mutex_init(), pthread_mutex_lock(), pthread_mutex_unlock() 등을 사용하고, emtpy/full에는 sem_init(), sem_wait(), sem_post() 등을 사용함.

### 6.4.2. Readers and Writers Problem

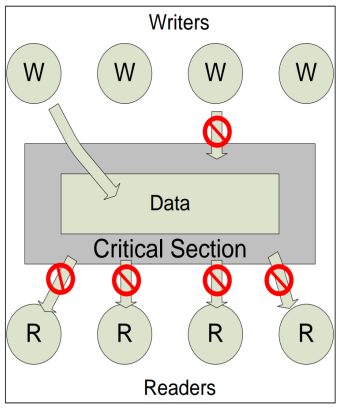
***1. Readers and Writers Problem***

*Readers and Writers Problem은 공유 데이터(buffer 등)로 여러 reader와 writer가 접근 가능한 상황의 synchronization을 처리하는 문제임.*

*reader는 공유 데이터를 read하는데, 데이터를 수정하지 않으므로 동시에 여러 reader가 접근이 가능해야 함. writer는 공유 데이터에 대해 write하는데, 한 번에 하나의 writer만 접근이 가능해야 함. 또한 reader와 writer가 동시에 접근하는 것은 불가능해야 함.*



<여러 reader는 동시에 read가능>



<하나의 writer만 write가능>

***2. Solution1***

*아래와 같이 semaphore를 사용해 해결할 수 있음.*

*readcount : 공유 데이터에 접근 중인 reader의 개수를 나타냄.*

*wrt : writer에 대한 mutex 처리를 함.*
*mutex : reader에 대한 mutex 처리를 함.*

*아래와 같이 구현할 수 있음.*

```
readcount = 0;
wrt = 1;    // 한 번에 하나만 접근하도록 함
mutex = 1;  // 한 번에 하나만 접근하도록 함
```

```
// writer
P(wrt);
... // 공유 데이터에 대해 write
V(wrt);
```

```
// reader
P(mutex);                   // reader에 대한 mutex
    readcount++;
    if(readcount == 1) {    // 최초로 접근하려는 read인 경우.
        P(wt);              // writer에 대한 mutex
    }
V(mutex);

... // 공유 데이터 read. 다중 접근을 위해 P/V로 감싸지 않음.

P(mutex)
    readcount--;
    if(readcount == 0) {
        V(wt);
    }
V(mutex)
```

*나름 잘 작동하지만, reader가 계속해서 접근하려고 하면 writer는 계속 대기하게 됨. 즉, starvation이 발생할 수 있음.*

***3. Solution2***
*solution2는 writer를 우선 처리하여 solution1의 starvation을 해결함. 이때 wrtpending이 없어도 동일하게 동작하기는 하지만, 실제로는 write보다 read가 많은 것이 일반적이기 때문에 write에 기회를 더 부여하기 위한 구현임. 아래와 같이 구현할 수 있음.*

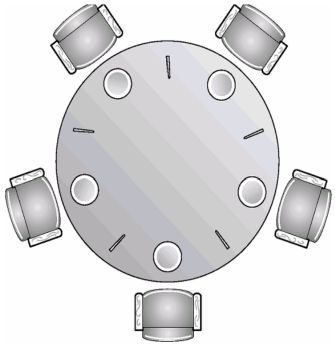
```
readcount = 0
rd = 1
s1 = 1
writecount = 0
wrt = 1
s2 = 1
wrtpending = 1
```

```
// writer
P(s2);
writecount++;
if(writecount == 1) {
    P(rd);
}
V(s2)

P(wrtpending)
P(wrt);
... // 공유 데이터에 대해 write
V(wrt);
V(wrtpending)

P(s2);
writecount--;
if(writecount == 0) {
    V(rd);
}
V(s2);
```

```
// reader
P(writepending);
V(writepending);

P(rd);
P(s1);
readcount++;
if(readcount == 1) {
    P(wrt);
}
V(s1);
V(rd);

... // 공유 데이터 read

P(s1);
readount--;
if(readcount == 0) {
    V(wrt);
}
V(s1);
```

### 6.4.3. Dining Philosophers Problem

***1. Dining Philosophers Problem***
*Dining Philosopher problem은 5명의 철학자가 식사를 하는데 젓가락이 사람 사이에 하나씩 총 5개만 존재하여, 각 철학자가 적절히 식사할 수 있도록 synchronization을 처리하는 문제임. 이때 각 철학자는 자신의 바로 옆 젓가락만 집을 수 있고, 두 젓가락이 모두 있어야 식사할 수 있고, 젓가락을 집었으면 식사를 한 이후에 내려놓을 수 있음.*



*실제로는 컴퓨터 자원의 사용 등에 대한 문제 해결에 적용할 수 있음.*

***2. Solution1***
*아래와 같이 각 젓가락에 대한 semaphore를 사용하여 단순히 젓가락에 대한 synchronization을 생각할 수 있음.*

```
// philosopher
...
P(chopstick[i])              // 철학자 위치에 따른 특정 젓가락 잡기
P(chopstick[(i + 1) % 5])    // 그 옆 젓가락 잡기
... // eat
P(chopstick[(i + 1) % 5])
P(chopstick[i])
...
```

*이 경우 각 철학자가 젓가락을 하나씩 모두 잡은 상황에 deadlock이 걸림. 이에 따라 아래와 같은 개선 방법을 생각해 볼 수 있음.*

*1. 철학자의 수 자체를 줄임.*
*2. 철학자에 번호를 부여하여 짝수 번째는 오른쪽을, 홀수 번째는 왼쪽을 집게 함.*
*3. 집기 전에 검사를 해서 양쪽의 젓가락이 모두 사용 가능한 경우에만 집게 함.*

*물론 이렇게 deadlock은 해결이 되지만 starvation은 발생할 수 있음. 이는 aging 등 개별적인 구현에 따라 해결이 가능함.*

***3. Solution2***
*집기 전에 검사를 해서 양쪽의 젓가락이 모두 사용 가능한 경우에만 집게 함.*

*이를 위해 철학자의 상태(THINKING, HUNGRY, EATING)를 저장하는 배열 state, 젓가락 조작에 대한 semaphore mutex, 젓가락 2개를 모두 잡은 상태에 대한 semaphore 배열 self를 사용함.*

*아래와 같이 구현됨.*

```
state[5] = {THINKING, THINKING, THINKING, THINKING, THINKING};
mutex = 1;
self[5] = { 0 }
```

```
// philosopher
... // thinking
take_chopsticks(i)  // i번째 철학자에 대한 작업
... // eating
put_chopsticks(i)   // i번째 철학자에 대한 작업
... // thinking
```

```
take_chopsticks(int i) {
    P(mutex);
    state[i] = HUNGRY;  // 상태가 HUNGRY이면 대기 상태인 것
    test(i);            // 양쪽이 모두 존재하는지 검증 및 self[i] 조정
    V(mutex);
    P(self[i]);         // test()에서 높인 값을 깎으며 들어감
}

put_chopsticks(int i) {
    P(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    V(mutex);
}

test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING
                          && state[RIGHT] != EATING) {
        state[i] = EATING;
        V(self[i]);
    }
}
```

*먹으려는 철학자는 상태가 HUNGRY로 지정됨. 좌우를 검사하여 모두 EATING이 아니면 해당 철학자가 EATING이 되고, 둘 중 하나라도 EATING이면 HUNGRY 상태로 P(self[i]);에서 대기함. 해당 철학자는 EATING이던 철학자가 자신의 작업이 끝난 뒤 젓가락을 건내주면 작업을 수행할 수 있음.*

*철학자가 젓가락을 넘길 때에는 좌우를 살펴서 좌우의 철학자 중 하나가 대기 중(HUNGRY)이고 좌우가 먹고 있지 않으면 젓가락을 넘겨줌. 젓가락을 내려놓을 때 이 작업을 수행하지 않으면 대기 상태의 철학자가 젓가락을 잡지 못함.*

# 7. Memory Management

## 7.1. Virtual Address

### 7.1.1. Virtual Address

***1. Address Space***
*주소 공간(Address Space)은 process에서 참조할 수 있는 주소의 범위 또는 집합으로, process와 1대1로 대응됨.*

*address space의 크기는 cpu의 주소 버스(address bus)에 의해 결정됨. cpu는 address bus를 통해 주소를 주고 받으므로, address bus가 n bit라면 해당 cpu는 총 $2^n$개의 주소를 식별할 수 있음. 이때 시스템이 addressing할 수 있는 전체 memory 공간은 당연하게도 address space의 크기와 각 주소가 가지는 memory 크기를 곱해서 구할 수 있음. 예를 들어, cpu의 address bus가 32bit이고 주소 하나 당 1byte의 크기를 가진다면, 해당 시스템이 활용할 수 있는 전체 memory 크기는 $2^{32}B = 4GB$임.*

*32bit/64bit cpu라고 이야기할 때 32bit/64bit는 해당 bus의 크기임. 더 많은 memory 공간을 위해 64bit 시스템이 등장했지만, 지금의 상황에서는 64bit의 주소 전부를 다 쓰는 상황은 발생하지 않음.*

***2. Physical Address vs. Virtual Address***
*물리 주소(PA, Physical Address)는 memory에서 실질적으로 사용되는 실제 주소이고, 가상 주소(VA, Virtual/Logical Address)는 process 관점에서 사용하는 주소임.*

*VA는 logical address임. 각 process에서는 VA로 주소에 접근하여, 마치 해당 process 혼자서 memory 전체를 사용하고 있는 것처럼 동작함. 즉, process별로 사용 가능한 memory 영역을 정의할 필요가 없고, 여러 개의 process가 서로 충돌할 걱정 없이 동시에 실행될 수 있음.*

*초창기 시스템에서는 각 program은 compile time에 PA가 결정되었음. 이 방식으로는 여러 program을 동시에 memory에 load하는데에 한계가 있었는데, 초기에 사용되었던 MS-DOS 같은 os는 CLI 기반이어서 큰 문제가 없었음. windows 등 GUI가 등장하면서 multiprogramming의 필요성이 생기자 VM을 사용하기 시작했음.*

***3. VM의 처리 과정***
*소스 코드는 VM의 사용을 가정하고 작성됨.*

*compile time에서 compiler는 소스 코드로 symbol table(소스 코드의 식별자들과 그에 대한 주소 등을 저장하는 테이블)을 생성함. 이때 table의 symbol들은 table에 대해 상대적인 주소를 가지게 되고, table(object file)은 주소가 0부터 시작함.*

*link time에서 linker는 object 파일들과 라이브러리들을 묶어서 하나의 table을 생성함. 이때 생성된 table(executable file) 또한 주소가 0부터 시작함.*

*load time에서 loader는 executable file을 memory에 올리는데, 이때 os가 table과 PA 사이의 매핑을 생성함.*

*실행 도중에 발생하는 주소 관련 처리는 MMU라는 하드웨어를 사용함.*

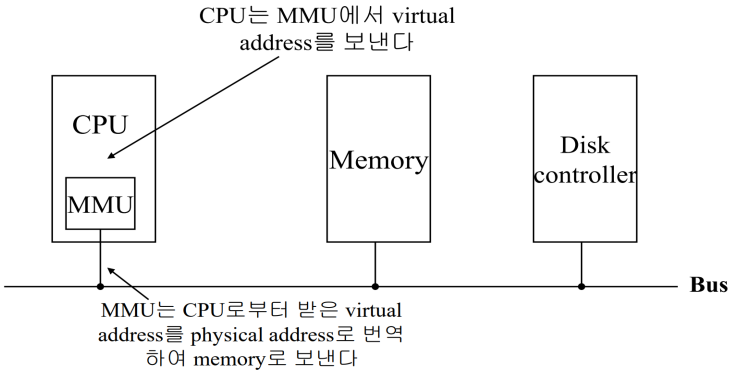
memory 모델에 따라 PA만을 사용할 수도 있고 VA를 사용할 수도 있겠지만, 대부분의 상용 시스템에서는 VA를 사용함.

### 7.1.2. MMU

*MMU(Memory Mangement Unit)는 VA과 PA 간의 변환(translation)을 수행하는 하드웨어 장치임. MMU는 cpu 안에 위치함.*

*MMU는 뒤에서 설명할 page table을 통해 translation을 수행함.*

*memory 접근은 빈번히 발생하는 작업이므로 MMU는 성능에 미치는 영향이 큼.*




## 7.2. Virtual Memory

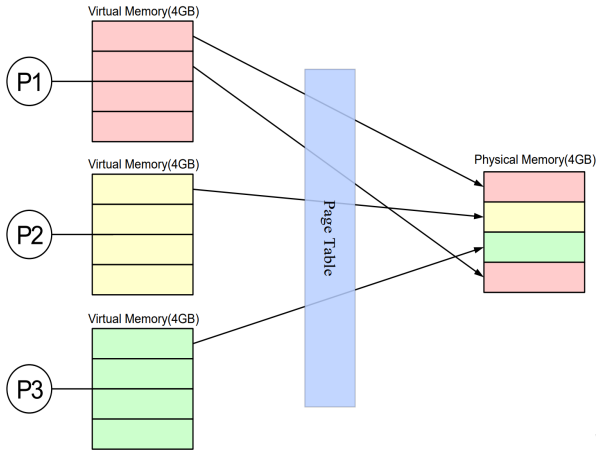
### 7.2.1. Virtual Memory

***1. Virtual Memroy***
*가상 메모리(VM, Virtual/logical Memory)는 실제로 존재하지는 않지만, 사용자에게 memory로서의 역할을 하는 memory임. 반면 물리 메모리(PM, Physical Memory)는 실제 하드웨어적으로 존재하는 memory임.*

*VM은 PM을 disk 공간까지 확장함. 즉, VM은 VA 활용에 의한 가상의 memory이기도 하고, 현재 사용하는 부분만을 memory에 올려 사용 공간을 늘리면서, 또한 효율적으로 사용하기 위한 개념이기도*

*함. 이는 뒤에서 설명할 page table과 demand paging으로 구현됨.*

*이에 따라 VM은 PM보다 큰 것이 일반적임. VA가 가지는 주소 값의 범위를 생각해봐도 당연함. VM이 PM보다 크므로 VM의 여러 page들은 같은 frame에 매핑될 수 있음.*




***2. VM을 사용한 memory 접근***
*VM을 사용한 memory 접근을 간단히 요약하면 아래와 같음.*

*각 process는 memory 접근 시에 VM에 접근하는 것으로 가정하고 동작함. VA를 사용해 VM에 접근하려고 하면, MMU는 page table을 사용해 해당 VA와 매핑된 부분이 PM에 올라와 있는지를 확인함. PM에 올라와 있으면 translation 수행 후 접근하고, 올라와 있지 않으면 해당 부분을 PM에 올린 뒤 translation하여 접근함.*

# 7.3. Paging

### 7.3.1. Page

***1. Page***
*Page(virtual page)는 VM에서 address space를 관리하는 단위임. 반면에 Frame(physical page)은 PM에서 address space를 관리하는 단위임.*

*즉, VM과 PM은 각각 page와 frame이라는 고정된 단위로 memory를 나누어 관리함.*

*일반적인 상용 시스템에서 page와 frame의 크기는 4KB임. 이때 page와 frame, 그리고 disk 관리 단위의 크기는 동일함. (동일하지 않다면 구현이 복잡해짐.)*

*당연하게도 page에 대응되는 실제 데이터는 PM에 올라와 있을 수도 있고(page에 frame이 할당된 상태라고 함.), disk에 있을 수도 있음.*

***2. Page Number/Address***
*Page Number는 process가 가진 page 각각에 부여된 번호로, 각 page를 구분하기 위한 번호임. Page Address(Page Offset)는 특정 page 내부에서의 구분을 위해 사용하는 주소임.*

*cpu가 관리하는 모든 주소는 page number와 page address로 나뉨. MMU는 VA를 page number와 page address로 분리하여 PA로 변환함.*

*page의 크기가 4KB이고 주소 하나가 1byte이면 총 $2^{12}$개의 주소가 있으므로, page 내부의 구분을 위해서 page address로는 12bit가 필요함. 하위 12bit를 제외한 나머지 bit는 page number로 사용됨.*

*예를 들어, page의 크기가 4KB이고 주소 하나가 1byte인 32bit system에서는 page number에 20bit가 사용되고, page address에 12bit가 사용됨.*

### 7.3.2. Page Table

***1. Page Table***
*Page Table은 각 process 별 page 정보를 저장하는 table임. process들은 각각 개별적인 page table을 가짐.*

*page table의 인덱스는 page number이고, 그 값은 PTE임.*

***2. Page Table Entry***
*Page Table Entry(PTE)는 page table의 레코드(열)임. PTE는 page base address와 flag bit 등의 항목들을 포함함.*

*Page Base Address(Frame Number)는 해당 page에 할당된 frame의 시작 주소임. page base address의 bit 수는 PA의 전체 길이에서 page address만큼을 뺀 것임. 이때 당연하게도 PA의 전체 길이는 PM의 크기에 의해 정해지고, VA의 길이(32bit 등)와는 다를 수 있음.*

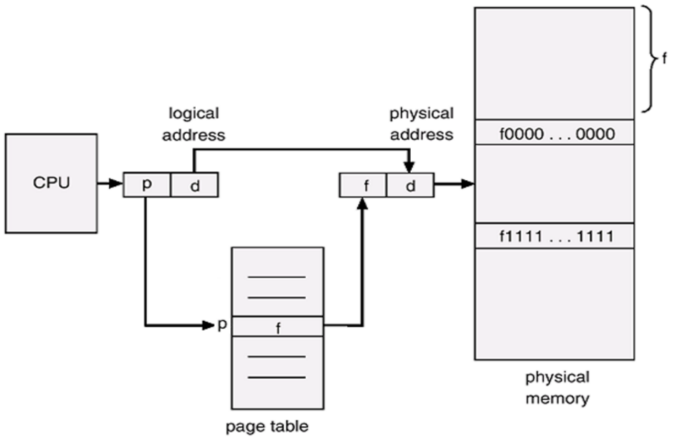
*Flag Bit는 해당 page의 상태를 나타내는 bit로, 아래와 같은 것들이 있음.*
*Accessed Bit : page에 대한 접근 발생 여부.*
*Dirty Bit : page 내용의 변경 여부.*
*Present Bit : page에 할당된 frame의 존재 여부.*
*Read/Write Bit : read/write 관련 권한 존재 여부.*

***3. PTBR/PTLR***
*page table은 그 자체도 꽤 큰 공간을 차지하므로 PM에 위치해 있음. page table의 주소에 VM을 사용할 수는 없으므로, page table에 대한 PA를 저장하는 register를 사용함.*

*아래와 같은 register를 사용함. 물론 register 이름은 architecture마다 다를 수 있음.*

*PTBR(Page Table Base Register) : page table에 대한 PA를 저장하는 register.*
*PTLR(Page Table Length Register) : page table의 size를 저장하는 register.*

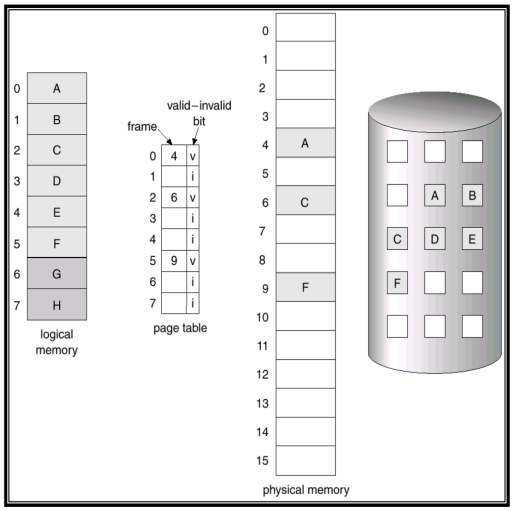
***4. 동작 과정***
*MMU는 VA를 page number와 page address로 분리하고, PTBR에 저장된 주소로 page table에 접근한 뒤 page number를 인덱스로 PTE를 찾음. 이후 PTE의 flag bit 등을 확인하고 해당 page base address를 page address와 결합하여 PA를 반환함.*




### 7.3.3. Demand Paging

***1. Demand Paging***
*Demand Paging은 process 실행을 위한 모든 page를 memory에 올리는 대신, 일부는 disk에 저장해 두었다가 page에 대한 사용 요청이 발생했을 때 memory에 올리는 기법임.*

*demand paging은 valid bit 등을 사용하여 구현됨. 즉, 해당 page에 대한 부분이 PM에 올라와 있는지 등을 검토함.*



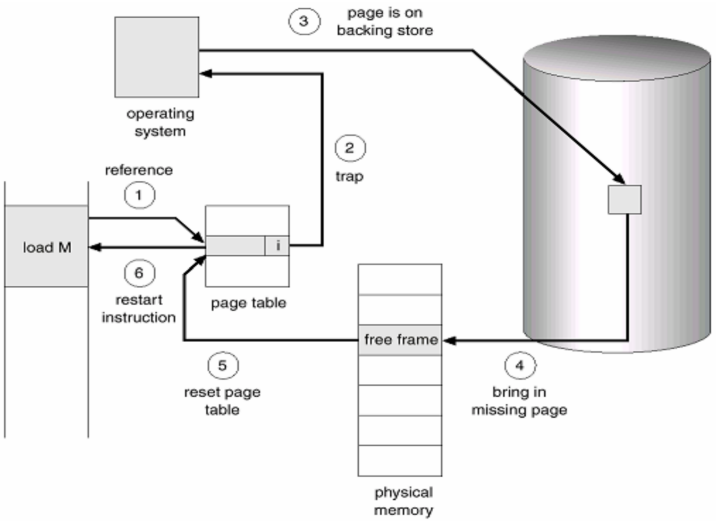

*demand paging은 process 실행을 위한 PM 구성 시간을 단축시키고, PM의 크기 한계를 disk를 활용하여 극복할 수 있도록 함.*

*demand paging을 사용하는 경우 page에 해당하는 부분이 PM에 존재하지 않는 경우가 발생하는데(page fault), 이에 대한 처리 또한 구현해야 함.*

***2. Page Fault***

*Page Fault는 process가 page를 참조했을 때 해당 page가 할당 받은 frame이 없는 경우를 말함. 즉, page에 해당하는 부분이 PM에 올라와 있지 않은 것임.*

*page fault가 발생하면 MMU는 os에 event 발생을 알리고, os는 page fault handler를 호출하여 아래의 작업을 수행함.*
*1. disk에서 page에 해당되는 내용을 찾음.*
*2. 해당 page에 free frame을 할당함.*
*2-1. free frame이 있다면 해당 frame을 사용함.*
*2-2. free frame이 없다면 page replacement를 수행함. 즉, victim frame을 PM에서 disk로 내리고(swap out), page table에서 해당 frame을 할당받았던 page를 invalid 처리함.*
*3. 요구 page에 해당되는 내용을 disk에서 empty frame에 올림(swap in).*
*4. page table에서 요구 page를 valid 처리함.*
*5. process의 작업을 재시작함.*




***3. Working Set***

*Working Set은 특정 시간 간격 동안 참조한 page number를 모은 집합임. (집합이므로 중복 원소를 가지지 않음.)*

*working set은 locality를 나타냄. 어떤 시간 간격의 working set이 가지는 원소의 개수가 적을수록 locality가 잘 활용되어 page fault가 덜 발생하였고, 반대로 원소가 많으면 page fault가 많이 발생한 것으로 이해할 수 있음. 후자의 경우 thrashing이 발생했을 수 있음.*

***4. Thrashing***
*Thrashing은 process의 전체 실행 시간에서 작업 시간보다 page fault 처리 시간이 더 긴 상황을 말함.*

당연하게도 PM의 크기와 page fault 발생 횟수는 대체로 반비례함. 각각의 process가 사용하는 memory가 크지 않더라도, PM이 크면 여러 process에 대한 부분을 많이 올릴 수 있으므로 page fault가 덜 발생함.

application에 따라 page fault의 발생 양상에 차이가 있을 수 있음. 일반적인 application의 경우 한 번 page fault가 발생하여 frame이 할당되면, 이후에는 locality에 의해 그 주위의 데이터를 사용하므로 fault 발생이 줄어듦. 비디오, 오디오 관련 application의 경우 어떤 부분을 계속 보는 게 아니므로 page fault가 일정하게 발생함. 그래서 이런 application은 몇십 초 뒤까지 미리 PM에 올려두는 식으로 동작함.

물론 page replacement에 따른 page table 수정 시에는 TLB 또한 수정해야 할 수 있음.

### 7.3.4. Paging의 필요성

*초창기 시스템의 memory 사용 형태와 이를 통해 paging의 필요성을 알아보자.*

***1. Contiguous Memory Allocation***
*memory 사용에 대해 고전적인 방식부터 살펴보면 아래와 같음. contiguous memory allocation에는 signal parition allocation과 multiple partition allocation이 있음.*

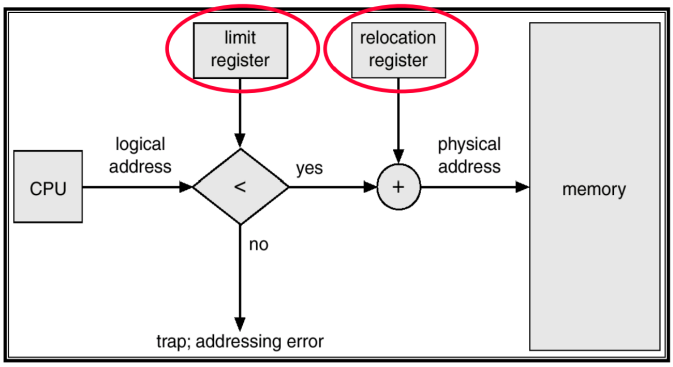
*1) Single Partition Allocation : user program 영역을 한 번에 하나의 user process가 연속적으로 사용하는 방식.*

*2) Multiple Partition Allocation : user program 영역을 여러 개의 user process가 각각 연속적으로 사용하는 방식. multiprogramming이 등장하며 사용한 방식임.*

*이 경우 특정 process 영역을 어떤 공간에 배치할 것인지를 적절히 결정할 수 있어야 함. 즉, allocation problem이 존재함. 가장 먼저 발견한 곳에 배치하는 것을 first-fit, 가장 작은 공간에 배치하는 것을 best-fit, 가장 큰 공간에 배치하는 것을 worst-fit이라고 함. 어떤 식으로 배치해도 항상 fragmentation의 발생을 완전히 배제하지 못함.*

*이 경우 MMU는 protection과 relocation을 수행함. Protection은 kernel memory 영역과 user memory 영역이 서로 침범하지 못하도록 보호하는 것을 말하고, limit register라는 하드웨어로 구현됨. Relocation은 user program을 재배치 가능한 주소로 표현하여 실제 memory 위치에 상관없이 내부 내용에 접근이 가능하도록 하는 것을 말하고, relocation register라는 하드웨어로 구현됨.*

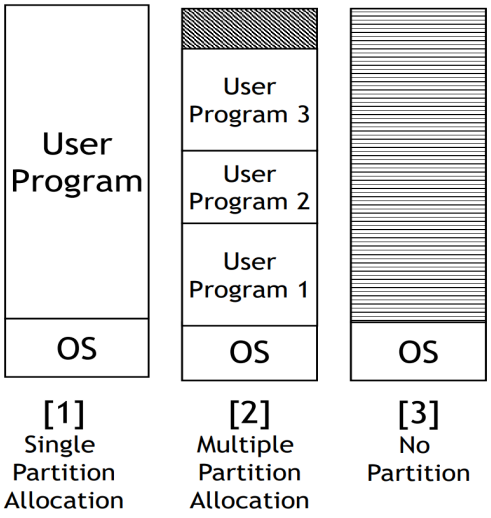
*Limit Register는 참조가 허용되는 주소의 최댓값을 저장하고, Relocation Register는 program이 차지하는 memory 영역 중 첫 번째 주소를 저장함.*




*3) No Partition : paging을 사용하여 각 process가 필요에 따라 연속적이지 않게 user program 영역을*

*사용하는 현대의 방식.*

*external fragmentation을 해결함.*




***2. Fragmentation***

*Fragmentation은 사용 가능한 memory가 존재하는데도 이를 효율적으로 활용하지 못해 낭비되는 현상임. external/internal fragmentation이 있음.*

*1) External Fragmentation : 남은 memory 공간이 충분하지만 그 공간이 연속적이지 않아 사용하지 못하는 경우. memory를 partition하여 사용할 때 발생하고, paging으로 해결이 가능함.*

*2) Internal Fragmentation : 할당된 memory의 크기가 요청된 크기보다 커서 할당에는 성공했지만, 남는 공간을 다른 곳에서 사용할 수 없는 경우. paging에 의해 개선될 수는 있지만, 완전히 해결되지는 못함.*

*예를 들어, 요청된 공간이 3900B이고 page의 크기가 4KB이면 100B의 internal fragmentation이 발생하게 됨.*

## 7.4. Page Replacement

### 7.4.1. Page Replacement

***1. Page Replacement***

*Page Replacement는 기존 frame(Victim Frame)의 내용을 disk에 저장하고, 요구된 page가 frame(Free Frame)을 할당 받도록 하는 것을 말함.*

*memory의 과다 할당(Over Allocation)은 모든 user process가 사용하려는 page의 개수보다 실제 frame의 개수가 적은 상황임. page fault 처리 중에 page replacement를 수행하여 이를 해결할 수 있음.*

***2. 관련 알고리즘***

*page replacement은 아래와 같은 알고리즘에 의해 동작함.*

*Frame Allocation Algorithm : 각 process에게 어떤 기준으로 frame을 분배할 것인지에 대한 알고리즘.*
*Page Replacement Algorithm : 어떤 frame을 victim frame으로 선정할 지에 대한 알고리즘.*

*이 알고리즘들은 결국 I/O 작업(disk 접근)을 최소화 하기 위한 것들임.*

### 7.4.2. Page Replacement Algorithm

*Page Replacement Algorithm으로는 아래와 같은 것들이 있음.*

*당연하게도 page fault 발생(I/O 작업 수행)이 적을수록 효율적인 알고리즘임.*

***1. Optimal Algorithm***
*Optimal Algorithm은 앞으로 가장 오랫동안 사용되지 않을 page부터 먼저 교체하는 방식임.*

*모든 알고리즘 중 page fault가 가장 적게 발생하는 최적의 알고리즘이지만, 미래의 page 사용을 알 수는 없기 때문에 당연하게도 구현이 불가능함.*

***2. FIFO***
*FIFO(First In First Out)은 frame이 먼저 할당된 page부터 먼저 교체하는 방식임.*

*가장 단순하고, queue를 통해 간단히 구현이 가능함. 어떤 데이터가 한 번 사용됐으면 다시 등장하지 않는 데이터 집합(ex. 멀티미디어 데이터)에 대해서는 유리함.*

***3. SCR***
*SCR(Second Chance Replacement)는 page마다 참조 bit를 사용하여 자주 사용되는 page의 교체를 방지하는 방식으로, FIFO를 개선한 것임.*

*아래와 같이 참조 bit를 사용함.*
*1. 처음 frame을 할당받았을 때와 page가 참조되었을 때 참조 bit 값을 1로 함.*
*2. 참조 bit는 일정 주기마다 0으로 초기화됨.*
*3. victim frame으로 지정될 때. 참조 bit가 1인 경우에는 victime frame으로 지정되는 대신 참조 bit를 0으로 함.*

*참조 bit를 사용하는 대신, 참조된 page에 할당된 frame은 queue의 맨 끝으로 옮기는 방식으로 구현하기도 함.*

***4. Clock***
*Clock은 SCR과 동일한데, queue 대신 circular queue를 사용하는 방식임.*

*hand라는 포인터를 사용하여 ciruclar queue를 돌며 victim을 지정함.*

*자주 쓰이는 방식은 아님.*

***5. LFU***
*LFU(Least Frequently Userd)는 사용 빈도가 가장 적은 page를 먼저 교체하는 방식임. 즉, 해당 시점까지 가장 적게 참조된 page를 victim으로 선택함.*

*특정 시점에 많이 사용된 page는 이후에 사용되지 않는 경우에도 frame을 계속 차지하는 문제가 있음.*

***6. NRU***
*NRU(Not Recently Used)는 최근에 사용되지 않은 page를 먼저 교체하는 방식임.*

*참조 bit와 변형 bit를 아래와 같이 사용함. 변형이 발생했으면 다시 쓰일 것이는 경우가 많으므로 변형 bit를 사용함.*

*참조 bit.*
*1. 처음 frame을 할당받았을 때와 page가 참조되었을 때 참조 bit 값을 1로 함.*
*2. 참조 bit는 일정 주기마다 0으로 초기화됨.*

*변형 bit.*
*1. 처음 frame을 할당받았을 때 변형 bit 값을 0으로 함.*
*2. page의 내용이 수정되었으면 1로 함.*

*아래의 순서대로 victim을 결정함.*
*1. 참조 0, 변형 0*
*2. 참조 1, 변형 0*
*3. 참조 0, 변형 1*
*4. 참조 1, 변형 1*

***7. LRU***
*LRU(Least Recently Used)는 가장 오랜 시간 참조되지 않은 page를 먼저 교체하는 방식임.*

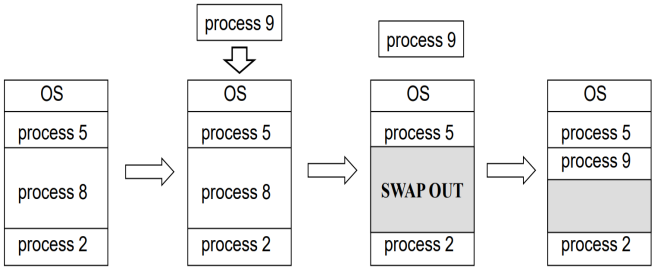
*temporal locality를 고려한 방식으로, 일반적인 경우에 대해 거의 optimal algorithm에 준할 정도로 성능이 좋음. 일반적인 상용 시스템에서 주로 사용하는 방식임.*

구현 방식으로는 아래와 같이 2가지가 있음.
1. Counter의 사용 : page가 참조된 시간을 저장하고 비교함.
2. Queue의 사용 : page가 참조되었으면 queue의 가장 위로 올림.

### 7.4.3. Swapping

***1. Swapping***
Swapping은 page replacement로 memory 부족을 해결할 수 없을 때 process 전체를 disk로 swap out 하는 것임.

***2. Swap 영역***
swap 영역은 page replacement 또는 swapping 시에 해당 데이터를 저장하는 disk 영역임. swap 영역을 구성하는 방법은 os마다 다름.




## 7.5. Page table 개선

### 7.5.1. Page Table 개선

단순 page table은 아래와 같이 2번의 PM 접근과 memory 낭비가 발생함. 이를 해결하는 기법을 알아보자.

***1. 2번의 PM 접근***
page table은 PM에 존재하므로 어떤 memory에 접근하려면 실질적으로 2번의 접근이 필요함. main memory에 대해서도 cache를 두는 것을 생각해 보면, page table 확인을 위해 매번 main memory에 접근하는 것은 비효율적임. 이에 따라 page table에도 TLB라는 하드웨어를 사용하여 caching함.

***2. Page Table의 memory 낭비***
process마다 page table이 존재하고, 각 process가 해당 table 전체를 사용하는 경우는 잘 없기 때문에, 기존의 page table의 사용은 memory의 낭비를 야기할 수 있음.

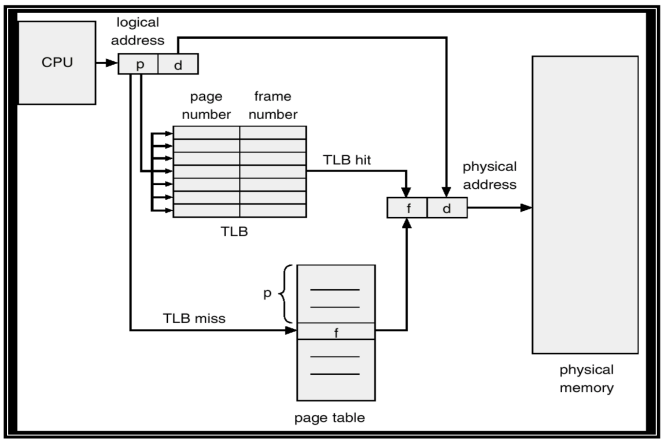
기존의 단일 page table을 생각해 보자. PTE는 $2^{20}$이고 각 entry 별로 4B(32bit)라고 하면 process 당 page table의 크기가 4MB임. 100개의 process가 존재한다면 400MB이므로 page table 자체가 사용하는 memory가 큼. 심지어 64bit 시스템이 등장하면서 그 낭비는 더욱 커짐.

이에 따라 page table의 크기를 개선하는 기법들이 존재함. multilevel page table과 inverted page table이 대표적인데, 현재 대부분의 상용 시스템에서는 multilevel page table을 사용함.

### 7.5.2. TLB

***1. TLB***
변환 색인 버퍼(TLB, Translation Look-aside Buffer)는 page table을 caching하는 register임. TLB는 MMU 내부에 위치함.

TLB에는 page table을 사용하여 수행된 translation 정보를 저장함. 즉, VA의 page number에 대응되는 PA의 frame number를 저장함. MMU는 page table에 접근하기 전에 TLB를 확인하여 대응되는 PA가 존재하는지를 확인함.




***2. TLB hit ratio***
*TLB hit ratio는 말 그대로 TLB에서 hit가 발생한 비율임. 이 값이 높을수록 접근 속도가 향상됨.*

*TLB는 register이므로 그 크기가 굉장히 작음(cache보다도 작음.). 하지만 TLB를 작게 둬도 locality에 의해 대체로 hit ratio가 높음. 생각해 보면 page가 4KB이므로 locality가 잘 고려된 application이라면 사용하는 page에 큰 변화가 자주 있지는 않을 것임.*

### 7.5.3. Multilevel Page Table

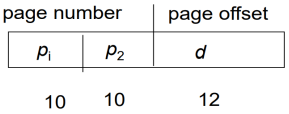
***1. Multilevel Page Table***
*Multilevel Page Table은 page table도 paging된 공간에 저장하는 기법임. 즉, page table을 여러 개의 작은 page table로 나누고, level을 부여하여 계층 구조로 구성함.*

*multilevel page table에서는 각 page table에 대한 PM을 필요할 때마다 할당하여 memory 사용을 최적화함. multilevel이 아닌 page table에서는 page table 전체를 PM에 올림. 반면 multilevel page table에서는 level 1 page table만을 PM에 올려 두고, 필요할 때마다 page table을 만듦. 구체적인 내용은 아래에 있는 2-level page table에서 정리함.*

*현재 대부분의 상용 시스템에서는 5개의 level을 사용함. 물론 level이 많아지면 속도가 느려지므로, TLB의 역할이 중요함.*

***2. 2-Level Page Table***
*2-Level Page Table은 2개의 level을 가지는 계층 구조의 page table임. 2-level page table에는 level 1 page table(page directory, outer page table)과 level 2 page table이 있음. 이 방식은 주로 32bit 시스템에서 사용함. 아래의 설명은 32bit 시스템과 4KB의 page 크기를 가정한 것임.*

*2-level page table에서는 VA의 page number(20bit)를 10bit씩 나누어 각각 level 1과 level 2에 대한 인덱스로 사용함. 각각을 page directory index, page table index라고도 함.*




*level 1 page table은 전체가 PM에 올라와 있음. 이 table은 VA의 첫 번째 10bit 값을 인덱스로 하고, level 2 page table의 frame number를 값(구현 방식에 따라 다를 수 있음.)으로 함. 각 크기가 4B(PA)인 $2^{10}$개의 레코드를 가지므로 총 4KB의 크기를 가짐. 이 크기는 frame 하나의 크기와 일치하므로 효율적임.*

*level 2 page table은 필요할 때마다 PM에 생성됨. 이 table은 VA의 두 번째 10bit 값을 인덱스로 하고, 구하려는 PA의 frame number를 값으로 함. 이 table도 마찬가지로 각 크기가 4B인 $2^{10}$개의 레코드를 가지므로 총 4KB의 크기를 가짐.*

*MMU는 VA를 받으면 TLB를 검사하고, 매핑이 없으면 PTBR에 저장된 PA를 사용해 level 1 page table에 접근함. 첫 번째 10bit를 인덱스로 level 1 page table에서 level 2 page table의 PA를 얻음. 이후 두*

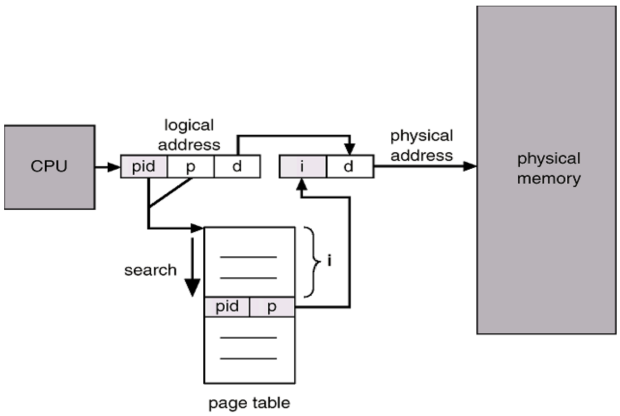
*번째 10bit를 인덱스로 level 2 page table에서 frame number를 얻음. 이 frame number와 page address를 합쳐서 PA를 도출함.*

#### 7.5.4. Inverted Page Table

*역 페이지 테이블(Inverted Page Table)은 page를 중심으로 frame을 찾는 기존의 방식과는 달리, frame을 중심으로 구성한 page table임.*

*inverted page table은 PM에 하나만 존재함. 즉, 기존 page table처럼 process별로 page table이 존재하는 게 아니라, 모든 process가 하나의 page table을 사용함. 이때 각 레코드는 frame 별로 존재하므로 page table의 크기는 PM의 크기에 따라 정해짐.*

*inverted page table은 frame number를 인덱스로 하고, PID와 page number를 값으로 함. 즉, 특정 process가 VA를 MMU에 전달하면 MMU는 해당 process의 PID와 VA의 page number로 page table을 탐색하여 일치하는 인덱스 값을 frame number로 PA를 구성함.*




*기존의 page table은 process마다 존재하는데, 각 process가 table의 모든 부분을 사용하지는 않음. inverted page table은 PM에 따른 고정 table 하나만을 사용하여 memory 사용을 줄임.*

*기존의 page table보다 적은 memory를 사용하기는 하지만, 매번 table을 순회하며 원하는 값을 찾아야 하므로 검색에 시간이 많이 든다는 단점이 있음.*

# 8. File System

## 8.1. File

#### 8.1.1. File

***1. File***
*파일(File)은 disk 사용에 대한 abstraction으로, 실질적으로는 단순히 byte의 나열임.*

*address space와 비교하자면, address space는 크기가 가변적이고, 저장공간에 연속적인 형태로 휘발성으로 저장됨. 반면 file은 기본적으로 고정된 크기를 가지며, 저장공간에 비연속적인 형태로 반영구적으로 저장됨. 물론 둘 다 process 간 공유가 가능함.*

*file도 page처럼 Disk Block(Data Block)이라는 단위로 나뉘어 있음. 앞에서 정리한 것처럼 page, frame, 그리고 disk block은 그 크기가 맞춰져 있는 것이 일반적임.*

***2. File Attribute***
*file은 아래와 같은 attribute, metadata를 가짐.*

*name : 사람이 알아볼 수 있는 기호화된 file 이름.*

*type : os가 지원할 수 있도록 구분한 file의 종류.*
*location : file이 disk에 저장된 위치.*
*size : file의 크기.*
*protection : file에 대한 권한 정보.*

*os는 file header에 적힌 type 정보로 파일 종류를 판단하여 관련 작업을 수행함. 윈도우에서는 확장자로 type을 확인할 수 있도록 하지만, 실질적으로는 header에 type이 명시되어 있음.*

### 8.1.2. File Operation

***1. File Operation***
*file에 일반적으로 적용되는 operation들은 아래와 같음.*

*create : file을 disk에 저장하기 위해 공간을 확보하고, 해당 공간에 대한 entry를 생성함.*
*write : file에 내용을 write함.*
*read : file 내용을 read함.*
*file seek : file의 cp를 특정 위치로 이동시킴.*
*delete : file을 삭제함.*
*open : disk의 directory structure를 확인하여 memory에 올림.*
*close : memory에 올라와 있는 directory structure를 disk에 저장함.*

*current-file-position pointer(cp)는 어떤 process가 해당 file에 대해 작업하고 있는 위치를 나타내는 포인터임. 여러 process가 하나의 file에 대해 동시에 작업할 수 있기 때문에 cp는 process 별로 존재함. 또한 주로 write/read를 수행하면 바로 다음 요소를 가리키도록 되어 있음.*

***2. Open File Table***
*여러 process가 동시에 하나의 file에 접근할 수 있기 때문에, open과 close 등에 대한 작업을 Open File Table로 관리함. open file table로는 아래와 같이 2가지 table이 있음.*

*1) Per Process Table : process 별 정보를 저장하는 table. process마다 존재함. cp 값 등을 저장함.*

*2) System-wide Table : file에 대한 정보를 저장하는 table. 시스템에 하나 존재함. 접근 시간, file 위치, open count 등을 저장함.*

*open count는 지금 몇 개의 process에게 열려 있는지를 나타냄. process가 file을 open하면 open count 값이 하나 증가되고, close하면 하나 감소됨. delete 연산은 open count를 검사하여 접근 중인 process가 없을 때 수행됨.*

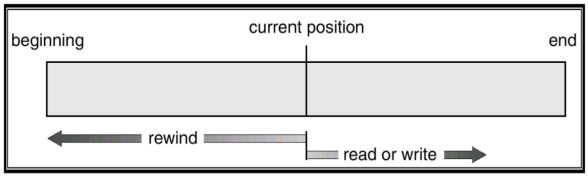
*물론 여러 process가 동시에 접근하는 상황에 대한 적절한 동기화도 적용되어야 함.*

### 8.1.3. File Access Method

*file의 특정 위치에 접근하는 방법은 아래와 같음.*

***1. 순차 접근***
*file의 특정 위치에 대한 접근을 맨 앞부터 순차적으로 수행하는 방식. 아래와 같은 operation을 사용함.*

*read next : cp에 해당하는 부분을 read함.*
*write next : cp에 해당하는 부분에 write함.*
*reset : 맨 앞으로 돌아감.*




*고전적인 방식으로, 현재에는 잘 사용되지 않음. 과거에는 tape 형태의 storage를 사용했기 때문에 이런 방식을 채택했음.*

***2. 임의 접근***
*file의 특정 위치에 대한 접근을 임의로 수행하는 방식. 즉, 원하는 위치에 바로 접근함. 아래와 같은 operation을 사용함.*

*read n : 인자로 작성한 block number에 해당되는 위치를 read.*
*write n : 인자로 작성한 block number에 해당되는 위치에 write.*

## 8.2. File System

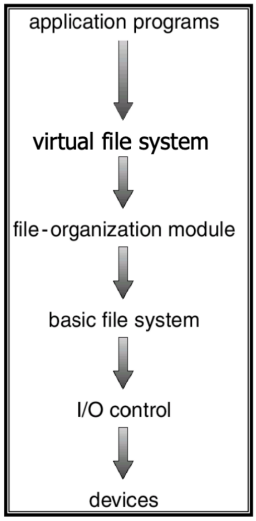
### 8.2.1. File System

***1. File System***
*파일시스템(File System)은 file의 file block과 physical disk의 disk block에 대한 매핑을 제공하는 시스템임. 즉, 어떤 file을 disk의 어떤 위치에 저장할 것인지를 결정함. 또는 disk에 존재하는 file의 집합 전체를 의미하기도 함.*

*사용자는 file system을 사용하여 disk에 대한 실질적인 구현과 물리적인 위치를 몰라도 접근하여 사용이 가능함.*

*disk에 대한 매핑과 접근 방식은 file system에 정의된 바에 따라 다름.*

***2. 계층화***
*일반적으로 file system은 아래와 같이 계층화되어 있음.*




*각 disk마다 다른 file system을 사용할 수 있는데. 계층화는 하나의 시스템에서 여러 file system과 호환될 수 있도록 하고, file system 선택에 대한 유연성을 제공함.*

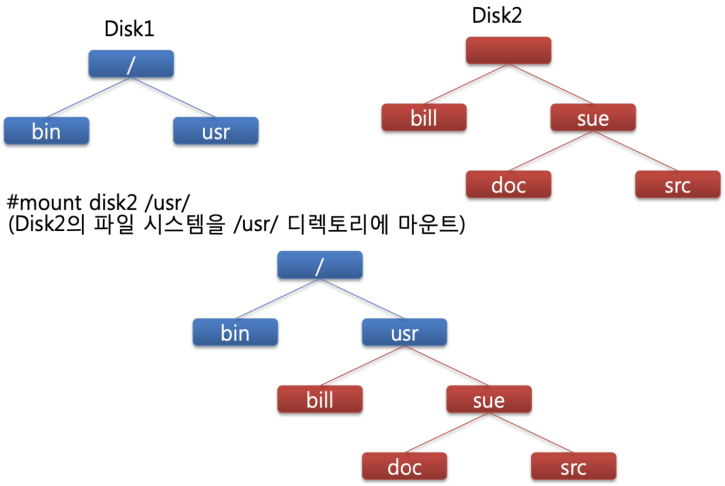
*특히 VFS(Virtual File System)은 각 file system에 대한 인터페이스로서 기능함. 즉, VFS에 호환되는 file system이면 해당 시스템에서 사용이 가능함.*

참고로 windows에는 Diskmgmt.msc, linux에서는 Fdisk라는 소프트웨어를 사용해 disk 및 file system 정보를 확인하고 관리할 수 있음.

### 8.2.2. Mount

*Mount는 각 disk device를 directory에 매핑하는 것으로, device와 directory를 분리하여 볼 수 있도록 하는 개념임. 주로 linux 계열 os에서 사용함.*

*어떤 device를 os가 지원하는 file system을 통해 사용하려면 mount를 해서 어떤 directory와 연결해야 함.*




*mount를 사용하면 distributed file system으로도 확장이 유리함.*

### 8.2.3. Protection

*Protection은 file system에 존재하는 file로의 부적절한 접근을 막는 것임. 즉, 권한 관리를 하는 것.*

*Unix 기반의 os가 사용하는 file system에서는, 권한을 총 9bit로 read, write, execute 작업에 대해 owner, group, public 사용자의 권한을 각각 지정한 뒤, 이를 3자리의 8진수로 변환하여 저장함. file과 directory에 대해 권한을 지정할 수 있음(리눅스에서는 chmod 명령어 사용.).*

*group은 사용자에 대한 group임. 특정 file에 대한 group을 지정할 수 있음(리눅스에서는 chgrp 명령어 사용.).*

*예를 들어, owner는 read/write/execute이 가능하고, group은 read/write만, public은 execute만 가능하도록 권한을 지정한다고 하자. 111 110 001이므로 761을 권한 정보로 저장함.*

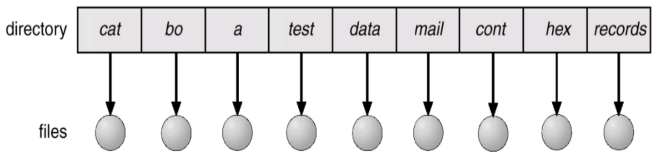
## 8.3. Directory

### 8.3.1. Directory

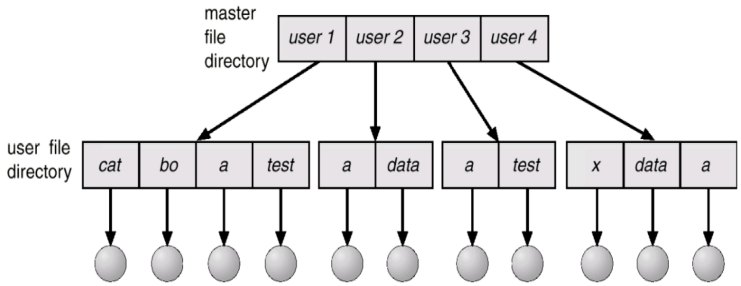
*디렉토리(Directory)는 여러 file을 묶는 abstraction으로, file들에 대한 정보를 가지고 있는 file임.*

*Directory Structure는 disk 상에서 file과 directory를 계층적으로 배치하여 저장하는 구조임. file system은 directory structure로 file들을 관리함. 아래와 같은 directory structure들이 있음. 현재 일반적인 시스템에서는 tree 구조에 link 등을 사용함.*

***1. Single-level Directory***
*모든 사용자에 대해서 하나의 단일 directory만을 사용하는 방식.*




***2. Two-level Directory***
*각 사용자 별로 단일 directory를 사용하는 방식.*




*상위 directory에는 각 사용자와, 매핑된 directory를 저장함. 사용자 별 하위 directory에는 해당 사용자의 file 정보를 저장함.*

***3. Tree-Structed Directory***
*Tree 구조로 directory를 구성하는 방식.*

*grouping을 통한 효율적인 탐색과 분류를 할 수 있음.*

***4. Acyclic-Graph Directory***
*tree 구조에서 aliasing으로 file이나 directory를 공유하는 방식.*

*Aliasing은 동일한 데이터를 서로 다른 이름으로 접근하는 것을 말함. unix 계열 os에서는 link로 구현됨.*

*aliasing을 적용 중인 대상을 삭제했을 때 Dangling Pointer(가리키는 대상이 없는 포인터)가 발생할 수 있음.*

***5. General Graph Directory***
*일반적인 graph 구조로 directory를 구성하는 방식.*

*이때 cycle이 발생하지 않도록 file만이 link를 할 수 있도록 하거나, link 생성 시마다 Cycle Detection Algorithm을 사용하여 cycle의 생성 여부를 검토함.*

## 8.4. File/Directory의 구현

### 8.4.1. File의 구현

*file의 구현에서는 file의 내용을 담고 있는 data block의 위치 정보를 어떻게 저장할지를 고려해야 함. 이에 따라 아래와 같은 구현 방법들이 있음.*

***1. Contiguous Allocation***
*Contiguous Allocation은 file을 물리적으로 연속된 data block에 저장하는 방식임.*

*구현이 간단하고, file 전체를 한 번에 읽어야 하는 상황에서 유리함. 반면에 file이 한 번에 처음부터 끝까지 작성되어야 하고, external fragmentation이 발생할 수 있음.*

*frame에서의 contiguous memory allocation과 유사함. 주로 tape를 disk로 사용하던 시기에 사용하던 방식임.*

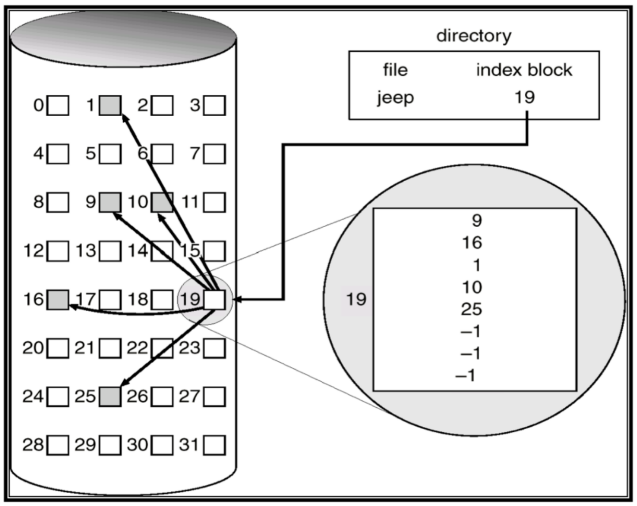
***2. Linked List Allocation***
*Linked List Allocation은 file의 각 data block을 linked list로 연결해 저장하는 방식임.*

*이때 포인터가 차지하는 크기는 disk의 전체 크기와 block size 등에 의해 결정됨.*

*file의 data block이 연속된 공간에 구성될 필요가 없고, external fragmentation을 방지함. 반면에 특정 지점의 데이터에 접근하려면 linked list의 처음부터 순회해야 함(random access 불가.). 또한 pointer 사용에 따라 추가적인 공간을 사용해야 하고, 저장 공간이 딱 떨어지지 않을 수 있음.*

*즉, 공간 낭비 줄이는 거 빼고는 안 좋은 게 많음.*

***3. Linked List Allocation using Index***
*Linked List Allocation using Index는 file의 data block 중 하나를 Index Block으로 지정하여, 나머지 data block에 대한 위치 정보 전부를 index block에 저장하는 방식임.*




*linked list allocation을 random access가 가능하도록 개선함. 하지만 index block에 저장할 수 있는 위치 정보에는 한계가 있으므로, index block의 크기에 따라 file 하나의 크기가 작게 제한됨.*

*이 경우 크기가 큰 데이터에 대해서는 여러 개의 file을 사용해야 함.*

***4. Inode***

*Inode는 file에 대한 data block index를 table 또는 multilevel table로 관리하는 방식임.*

*inode에는 direct index, single/double/triple indirect 방식이 존재함.*

*1) Direct : data block을 직접 가리키는 방식으로, data block에 대한 주소를 직접 저장함. inode 구조체에서는 direct가 주로 12개의 entry를 가짐.*

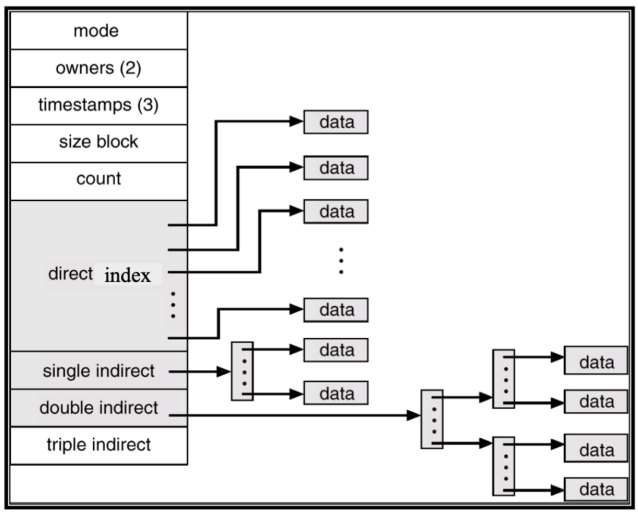
*direct index가 12개이므로, data block 하나를 4KB라고 하면 총 48KB 크기의 file에 대한 위치 정보를 저장할 수 있음.*

*2) Single/Double/Triple Indirect : index block을 사용하여 data block을 multilevel로 가리키는 방식임. 사용하는 index block의 계층 수에 따라 single/double/triple으로 구분됨.*

*single indirect를 생각해 보자. data block의 크기를 4KB, data block의 주소 하나를 4B로 한다면, 하나의 index block은 총 $2^{10} = 1024$개의 주소를 가리킬 수 있음. 즉, single indirect는 총 $1024 \times 4 = 4MB$에 대한 위치 정보를 저장할 수 있음. double indirect는 single indirect가 $2^{10}$개, triple indirect는 double indirect가 $2^{10}$개 있는 것이므로 각각 4GB, 4TB에 대한 위치 정보를 저장할 수 있음.*

*inode는 각 file마다 존재하여 metadata를 저장하는 구조체임. inode는 아래의 그림과 같이 file에 대한 attribute를 저장하는 field와, direct에 대한 index를 저장하는 부분, single/double/triple indirect에 대한 주소를 저장하는 부분으로 구성됨.*

*각 file에는 inode number가 부여되고, 해당 번호로 inode에 접근할 수 있음.*




*page table을 multilevel로 한 것처럼, index block을 multilevel로 한 것임. 현재 대부분의 상용 시스템은 I-node 방식을 주로 사용함.*

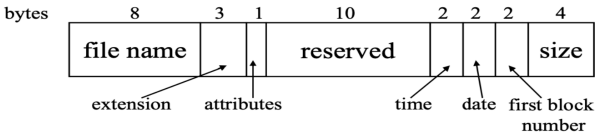
### 8.4.2. Directory의 구현

*Directory Entry는 directory를 표현하기 위한 자료구조임. directory는 자신이 포함하는 file에 대한 정보를 담고 있는 directory entry로 구성됨.*

*directory entry는 file의 구현 방식에 따라 가지는 field가 달라짐. 아래와 같이 MS-DOS의 방식과 Unix의 방식으로 나누어 볼 수 있음.*

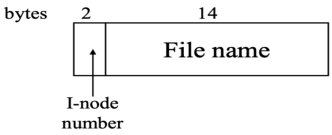
***1. MS-DOS의 방식***

*MS-DOS는 linked list allocation으로 file을 구현함. 이는 고전적인 방식으로, 아래와 같은 directory entry를 가짐. linked list에서 첫 번째 block에 대한 정보를 first block number field에 저장하고 있는 것을 알 수 있음.*

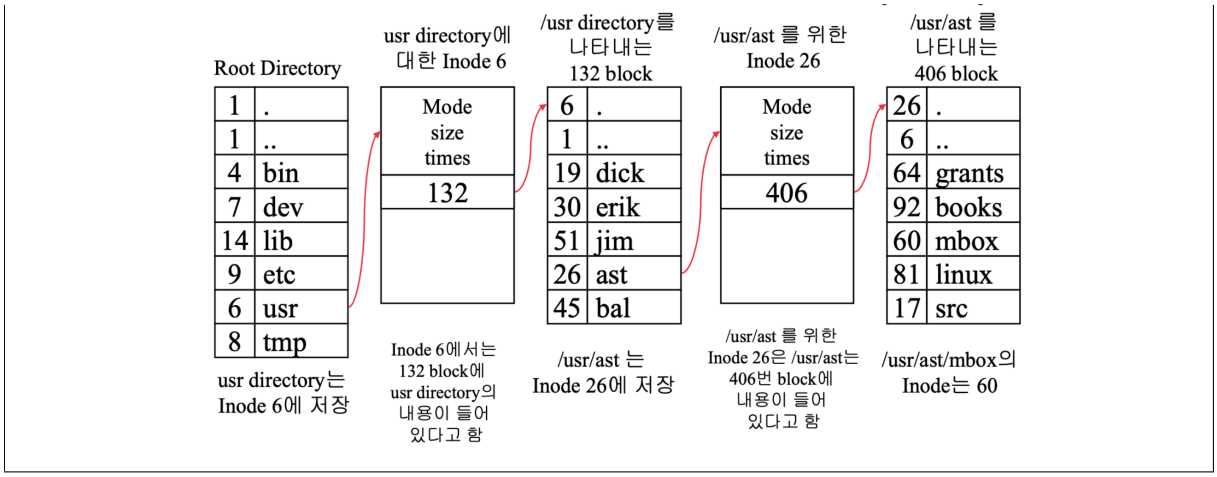



***2. Unix의 방식***

*Unix는 inode로 file을 구현하고, directory entry가 file의 name과 inode number로 구성됨. 해당 file에 대한 정보는 inode에 포함되어 있으므로 inode number만 있으면 됨.*




*아래와 같이 특정 위치에 있는 file에 접근이 가능함.*


Root Directory
1 .
1 ..
4 bin
7 dev
14 lib
9 etc
6 usr
8 tmp
usr directory는 Inode 6에 저장
usr directory에 대한 Inode 6
Mode
size
times
132
Inode 6에서는 132 block에 usr directory의 내용이 들어 있다고 함
/usr directory를 나타내는 132 block
6 .
1 ..
19 dick
30 erik
51 jim
26 ast
45 bal
/usr/ast 는 Inode 26에 저장
/usr/ast 를 위한 Inode 26
Mode
size
times
406
/usr/ast 를 위한 Inode 26은 /usr/ast는 406번 block에 내용이 들어 있다고 함
/usr/ast 를 나타내는 406 block
26 .
6 ..
64 grants
92 books
60 mbox
81 linux
17 src
/usr/ast/mbox의 Inode는 60