



Opearating Systems

1. OS introduce

운영체제란?

- 시스템 프로그램으로 중개자 역할을 담당
- 커널(Kernel): 항상 실행되는 하나의 프로그램 + 추가적인 프로그램
 - 하드웨어 자원을 프로세스에 분배하고, 메모리와 프로세스를 제어하며, 시스템 콜 등을 수행하는 OS의 코어
- 자원 할당자로 요청 사이에서 결정 및 모든 자원 관리
- 제어 프로그램으로 컴퓨터 오류 및 부적절한 사용 방지하기 위해 실행 제어

운영체제의 목적?

1. 컴퓨터 시스템을 더 편리하게 사용하기 위한 환경을 제공
2. 효율적인 하드웨어 관리 - 컴퓨터 시스템 자원의 효율적 관리
3. 다른 프로그램이 유용한 작업을 할 수 있도록 환경 제공

컴퓨터 시스템의 구성요소

컴퓨터 시스템의 구성요소: 4개의 구성요소로 나누어져 있음

- 1 . 하드웨어: 기본적인 자원을 제공(CPU, Memory, I/O device)
2. operating System: 사용자와 다른 응용프로그램 사이에서 하드웨어 사용 제어 역할
3. Application Programs: 사용자들의 컴퓨터 문제 해결을 위해 프로그램 사용시 자원 사용 방법 등
4. User : people, client, other machine...

운영체제 시스템은 무엇을 하나?

운영체제 환경을 제공

- 단독적으로 기능을 제공하지 않음
- 사용자는 운영체제가 제공하는 환경을 통해 더 쉽게 무언가 실행 가능

시스템 자원을 관리

- 사용자와 프로그램 간 space and time에서 resource를 공유할 수 있게 함
- 사용자가 효과적으로 하드웨어를 사용하게 함

사용자 관점(User View)

OS에 따라 사용되는 interface

- PC 사용자가 자원을 독점하도록 설계, 수행하는 작업을 최대화
- 사용의 용이성, 자원의 이용 효율성에는 신경을 쓰지 않음

System	Major design issues
Single-user system (PC)	Ease of use, Performance, security
Mobile devices	Touch screen, Voice recognition interface, Battery life
Multi-user system (Mainframe, minicomputer)	Resource utilization, Information exchange
Network-based system (workstation/server)	Individual usability vs. resource, utilization
Embedded computers	Little or no user view (numeric keypads + indicator lights)

시스템 관점(system View)

OS는 하드웨어와 가장 밀접한 프로그램이며, 자원할당자(Resource Allocator) CPU 시간, 메모리 공간 등을 효율적으로 사용하기 위한 자원의 관리자 역할을 수행 운영체제는 ‘제어 프로그램(control program)으로 출력 장치의 제어와 작동에 관여 - 서로 엑세스가 침범하지 않도록 통제!

컴퓨터 시스템의 구성(Computer - System Organization)

컴퓨터 시스템 연산(computer-system operating)

공통 버스에 의해 연결된 여러 개의 장치 제어기와 하나 이상의 CPU로 구성

Bootstrap Program(firmware; 펌웨어)

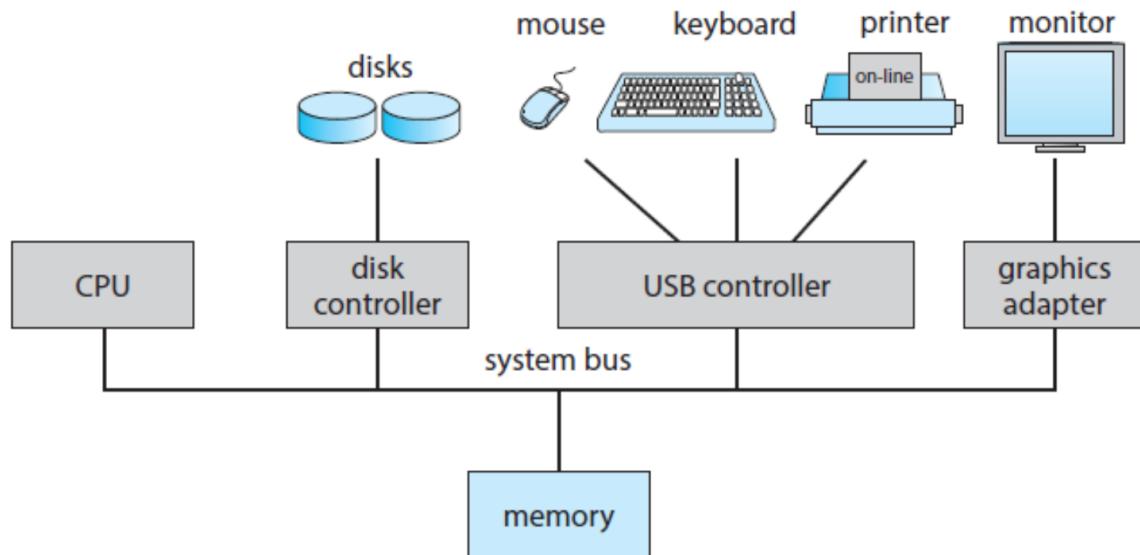
1. Bootstrap Program은 power-up 또는 reboot 할 때 실행하는 프로그램, 모든 시스템 초기화하여 ROM, EEPROM에 저장
2. CPU 레지스터로부터 장치 제어기, 메모리 내용 등을 포함한 시스템 모든 면을 초기화
3. 운영체제를 적재하는 방법 및 수행을 시작하는 방법을 알아야 함
 - OS kernel 로드 및 실행 (Bootstrap Loader)
 - 시스템 진단 및 초기화

OS Kernel (중요)

1. Unix에서는 첫 시스템 프로세스가 init (이 단계가 끝나면 시스템이 완전히 부트 상태)
2. 시스템은 이후에 event가 발생하기를 기다림
3. 하드웨어 또는 소프트웨어로부터 발생한 인터럽트(interrupt)에 의해 신호가 보내짐
4. 하드웨어는 어느 순간이든 시스템 버스를 통해 CPU에 신호를 보내고, 인터럽트를 발생 시킬 수 있음
5. CPU가 인터럽트 되면, CPU는 하던 일을 중단하고 즉시 고정된 위치로 실행 위치를 이동 ⇒ 위치는 일반적 인터럽트를 위한 서비스 루틴이 위치한 시작 주소를 가지고 있음
6. 인터럽트 서비스 루틴이 완료 후, 인터럽트 됐던 연산을 재개 ⇒ 인터럽트 서비스한 후, 저장되어 있던 복귀 주소를 Program Counter에 적재하고 인터럽트에 의해 중단된 연

산이 다시 시작!

7. 현대 연산 시스템 → interrupt driven programs, 그것을 디텍트하여 해결하는 동작



Modern Computer System

- Common bus: 하위 시스템 → 컴퓨터 구성 요소 사이 데이터 전송 시스템
 - Control bus는 명령을 전송
 - Data bus는 데이터를 전송함
- Bus에 비해 I/O device가 느리며, Controller는 device와 Bus 연결해주어, Reading / Writing Operation 발생
- CPU and I/O device 동시에 실행
- 각 device controller는 local buffer를 가지고 있다.
- CPU는 main memory → local buffer → main memory로 데이터 이동
- Reading은 device → local buffer, Writing은 local buffer → device
- device controller는 인터럽트를 발생시켜 CPU 작업을 완료했음을 알림

Interrupt

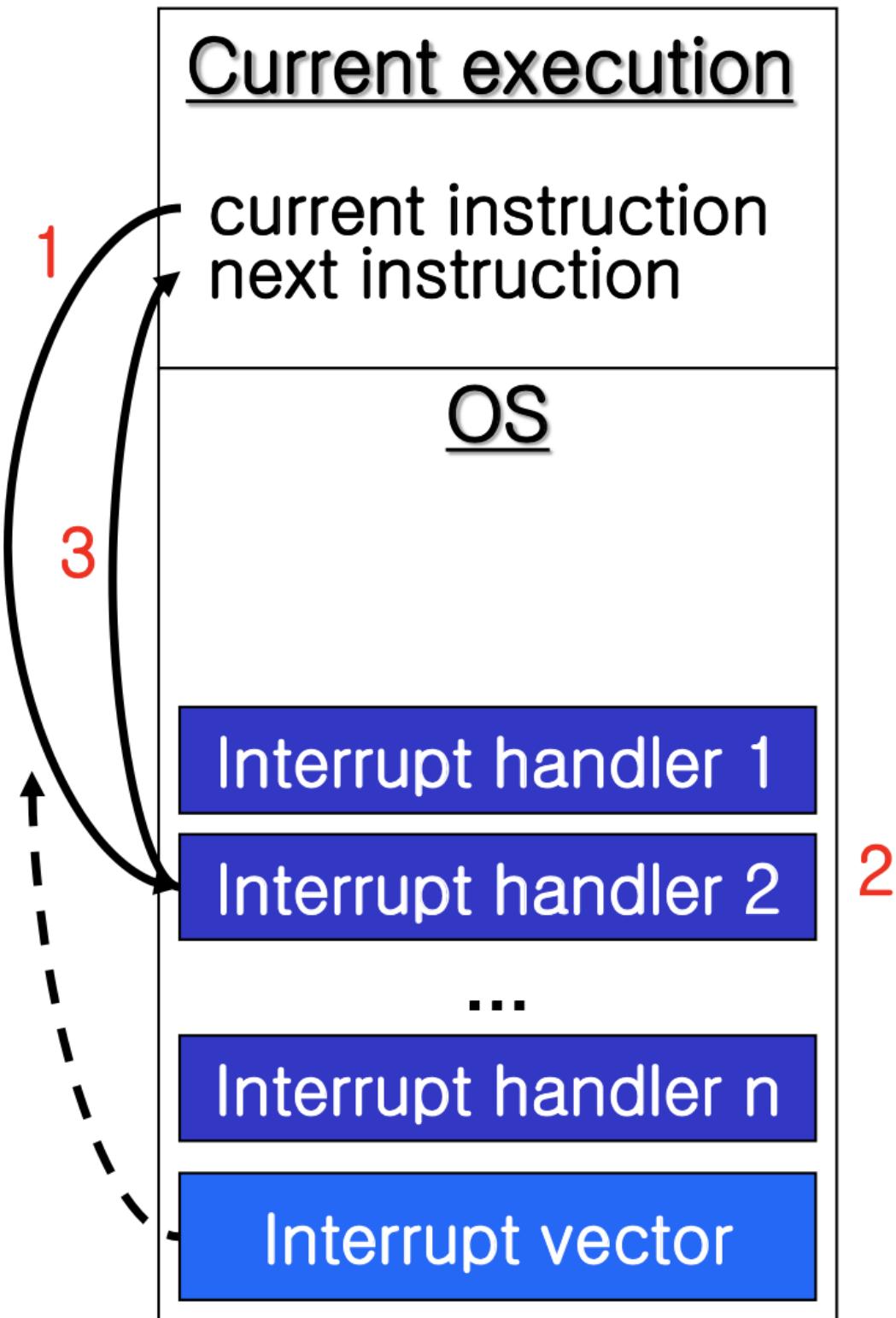


Interrupt : asynchronous signal 하드웨어 또는 소프트웨어 예외처리 필요한 경우 알려주는 것(신호)

- 하드웨어 인터럽트 : 하드웨어가 발생시키는 인터럽트, CPU에 서비스 요청해야 하는 경우 발생
- 소프트웨어 인터럽트 : 소프트웨어가 발생시키는 인터럽트, 소프트웨어가 스스로 인터럽트 라인 세팅 ⇒ 발생 위치나 시간이 정해져 있음
 - 시스템 콜 : 프로그램으로부터 요청
 - 예외상황 : invalid memory access, I/O exception, Divide by zero
- 각 타입의 인터럽트는 **IRQ number★**가 지정되어 있음
- **interrupt handler** : 인터럽트를 처리하기 위한 루틴, **인터럽트 서비스 루틴**이라고도 함
- **interrupt vector** : 인터럽트 발생 시 처리해야 할 **인터럽트 핸들러의 주소**를 인터럽트 별로 보관하고 있는 테이블

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Interrupt Mechanism



- CPU 작업을 멈추고, interrupter handler에 명령을 보냄 → 호출 전에 return address 와 state를 저장 (PCB에 저장)
- interrupt는 해당 handler에 의해 처리됨

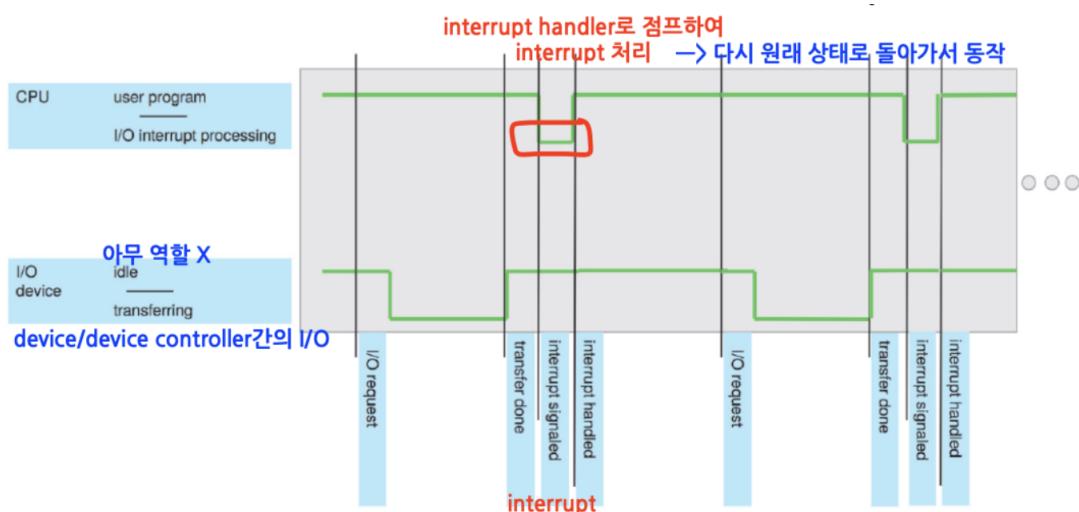
- interrupt된 프로그램으로 돌아감

Interrupt-based I/O (CPU와 I/O가 동시에 실행)

CPU는 요청을 보내면서 현재 프로세스나 다른 일들을 진행

Data 전송이 끝났을 때, I/O device interrupt를 보냄

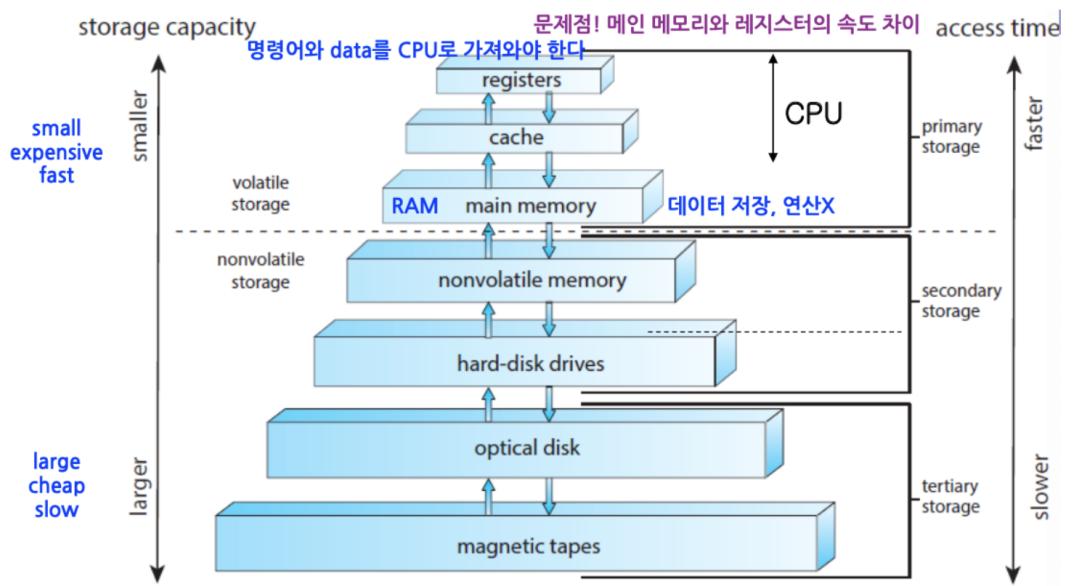
- CPU와 I/O가 동시에 실행, I/O device interrupt 보내면 CPU는 interrupt handler로 점프하여 처리한 후 다시 원래 상태로 돌아가서 동작함. I/O device와 transferring은 device와 device controller간의 입출력 의미



🍓 Storage Structure : 데이터나 명령어를 기억할 수 있는 저장공간을 의하는 저장 구조임.

🍇 Primary storage를 보면 레지스터, 캐시, 메인메모리로 구성되어 있음. 전원 OFF ⇒ 휘발성 메모리 특성, 저장공간이 작지만 빠름, 가격이 비싸다는 특징이 있음.

🍎 메인 메모리는 데이터를 저장하지만, 연산이 일어나지 않음. 연산을 하기 위해 명령어와 데이터를 CPU로 가져와야 하는데, 메모리와 레지스터의 속도 차이가 문제가 됨.



☞ 레지스터와 메인 메모리의 access time을 보면, 레지스터는 매우 빠른 속도로 reading과 writing이 가능한 반면, 메인 메모리는 상대적으로 느린 속도라는 것을 확인 할 수 있음

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

☞ 레지스터에서 메인 메모리를 엑세스하는데 시간이 많이 걸릴수록 전체적인 성능 저하를 가져오게 되므로 중간 역할로 캐시메모리를 사용함.

☞ Cache는 CPU 내에 존재하며 메인 메모리에서 읽어온 데이터를 임시로 저장하여 복사본을 만듦.

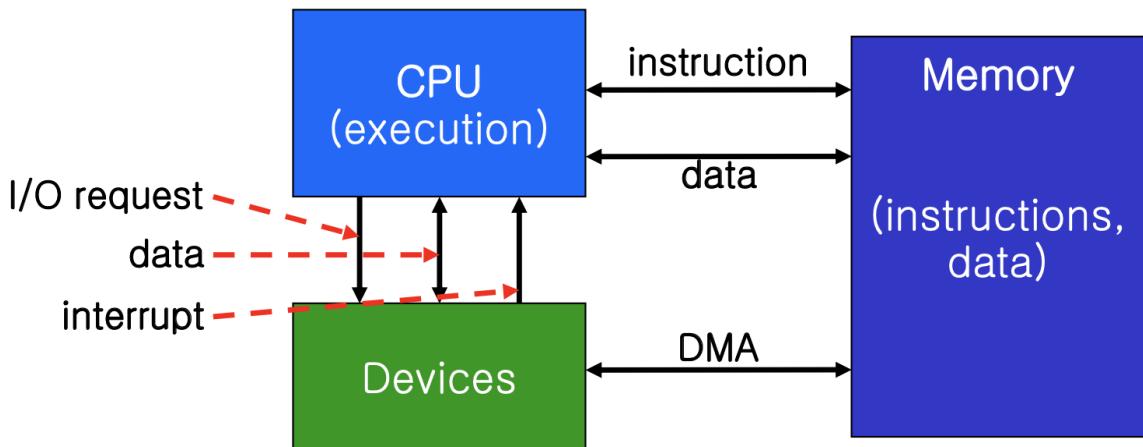
☞ 레지스터가 전에 읽어왔던 데이터와 동일한 데이터를 엑세스 해야 할 경우, 메인 메모리 를 엑세스하지 않고 캐시를 엑세스하면 속도가 개선됨.

Cache는 register와 main memory 사이의 속도 차로 인한 병목 현상을 막기 위해 사용함.

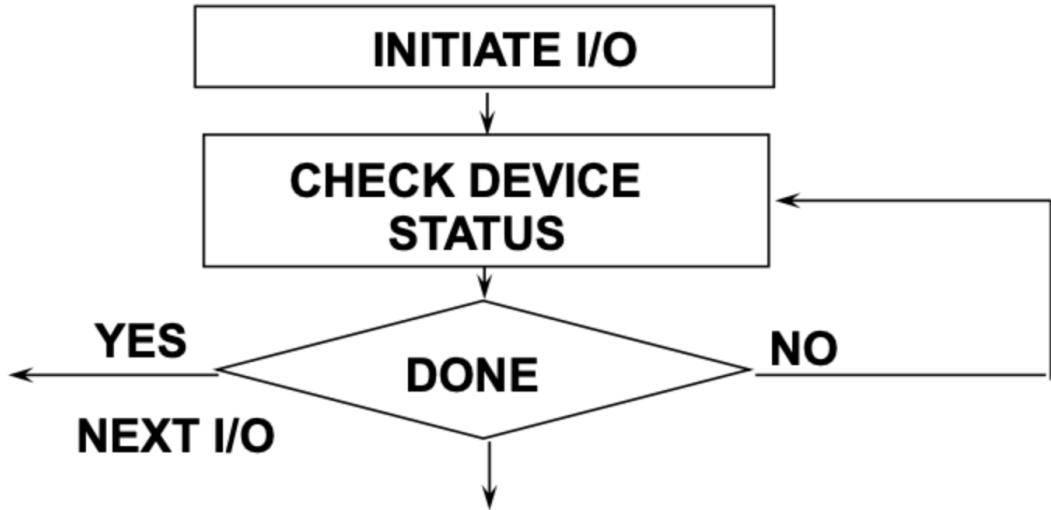
I/O Device Access

Old systems (Busy waiting)

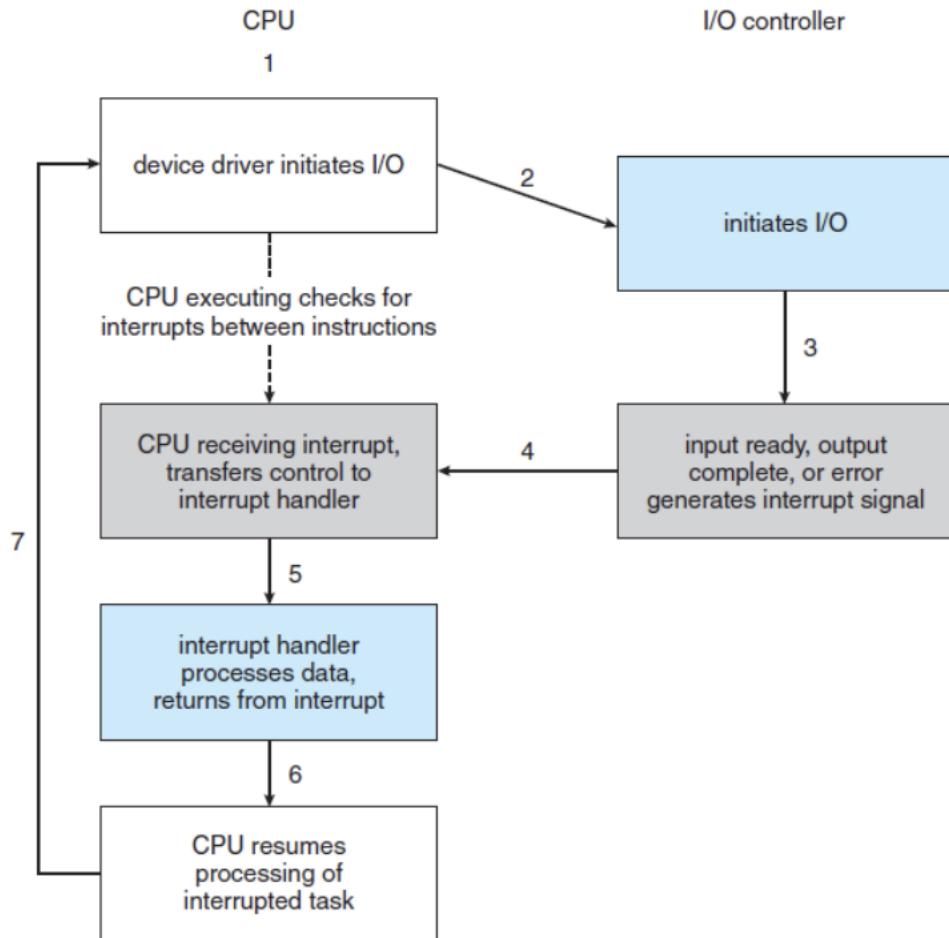
Modern systems (Interrupt-driven I/O, DMA = direct memory access → for large bulk of data)



- Busy waiting 시스템 : CPU가 I/O operation 요청 후, 주기적인 체크를 하며, operation이 끝나야 다음 루틴을 수행함 → CPU가 다른 작업을 하지 못하고 계속 기다려야 한다는 문제가 발생



- **Interrupt-driven I/O** : CPU가 Device Controller에게 동작을 요청했을 때, device Controller가 동작을 마친 후, interrupt controller에게 인터럽트 요청 → interrupt controller가 cpu에게 인터럽트를 보냄
 - **device controller**가 디바이스로부터 어떤 데이터를 읽어왔을 때, 이 데이터는 메모리에 저장되어야 함. CPU는 device controller의 **local buffer**에 있는 메모리를 writing 해야 함.
 - 디스크 같은 경우는 한 번에 많은 데이터를 전송해야 함. 읽어온 데이터가 큰 용량이라면, CPU가 하나의 메모리에 writing을 하는 것은 성능을 저하시킴.



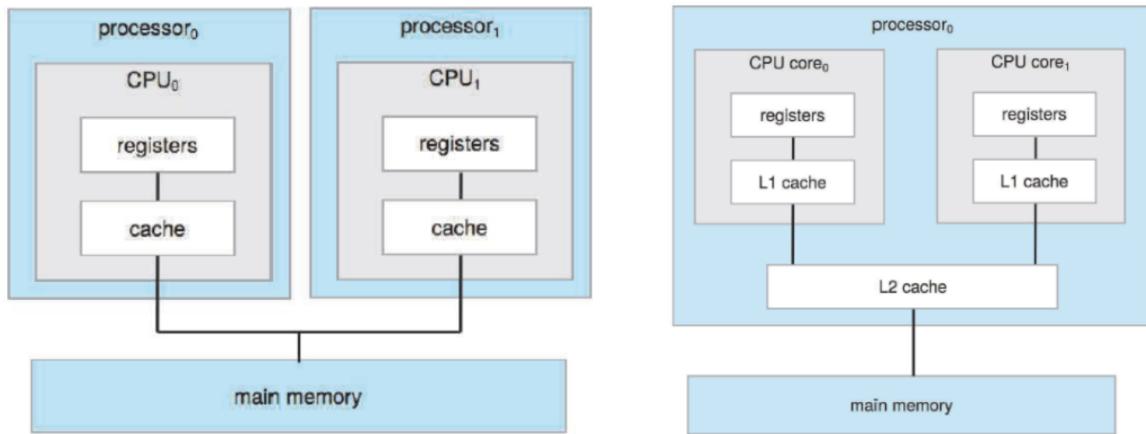
- **DMA(Direct Memory Access)** : CPU가 직접 접근하여 writing하지 않고, device controller가 직접 데이터를 메인 메모리에 저장시켜 줄 수 있음. CPU가 직접 local buffer에 있는 데이터를 옮길 필요없이 DMA를 통해 전송시킴 → CPU가 단순한 transfer 작업에 사용되지 않음
 - 전송해야 할 데이터의 양이 많을 때 성능 개선 효과가 높음

Computer system architecture

Single Processor, Multi Processor, Clustered Systems이 있음

- **Single-Processor systems** : 일반적 싱글 코어, 하나의 CPU가 존재, 간단하며 저렴하다는 장점이 있음

- Multi-Processor systems : 컴퓨터는 1대, **프로세스는 2개 이상**인 것을 의미함. 시스템의 컴퓨팅 속도를 향상시키기 위해, CPU가 추가됨. Tightly-coupled system으로 Bus, Clock , Memory, peripheral 등을 공유함.
- 멀티프로세싱의 종류는 Symmetric multiprocessing(SMP, 대칭형 멀티프로세싱)과 Asymmetric multiprocessing(비대칭형 멀티프로세싱)이 있음.
- SMP는 여러개의 CPU가 대등한 위치에서 협업을 함.



All processors are peers

- Asymmetric multiprocessing 하나의 CPU가 master, 나머지 slave로 구성됨. 두 개 이상의 각각의 프로세서가 자신만의 다른 특정 기능을 수행하는 아키텍처를 의미함.

Multi-Processor Systems

장점

- Increased throughput : 일정 시간동안 얼마나 많은 프로그램을 끝낼 수 있는가를 의미하는데, N개의 프로세서가 실행되면 N time만큼 속도가 빨라짐
- 병렬로 실행 가능하여 performance가 improve됨
- 저렴

Clustered Systems

Multiple systems working together

- loosely coupled system
- storage-area network(SAN) 통하여 스토리지 공유
 - SAN : 데이터를 서버와 연결
 - lock manager (DLM) → 분산을 풀어줌으로서 충돌을 피하기 위함.

목적

- high-availability service → failures
 - Asymmetric clustering은 하나의 머신으로 hot-standby mode
 - Symmetric clustering은 multiple nodes로 running applications, 각각을 모니터링하고 있음
 - Increased reliability
 - graceful degradation : ability to continue service proportional to the level of surviving 하드웨어 😊
 - fault tolerant : 하나의 프로세서에서 문제가 발생하여도 나머지 시스템에서는 계속 올바르게 작동하는 특성
- High-performance computing (HPC)
 - parallelization(평행화) 사용

Operating System Structure

- Motivation : single user가 CPU 및 I/O 디바이스를 항상 사용 중인 상태로 유지할 수 없음
- Multi-programming

- jobs을 관리함. 그래서 CPU는 항상 하나를 실행함
 - 시스템의 총 작업 중 일부가 메모리에 저장됨
 - job은 scheduling을 통해 선택하고 실행함
 - OS에서 jobs switch함 (기다려야 할 경우)
 - Timesharing (multitasking) : CPU job을 자주 switch, 실행 중에 상호 작용할 수 있도록 하여 인터랙티브 컴퓨팅 만들
 - response time < 1으로 해야 함
 - 각 user는 적어도 하나의 프로그램을 실행
 - CPU 스케줄러는 이미 실행한 job을 선택함.



Virtual Memory

Main memory는 모든 프로세스를 수용하기 충분하지 않음

Remedies

- swapping은 메모리 내용을 in / out으로 이동하여 실행함.
 - 가상 메모리를 통해 메모리에 완전히 저장되지 않은 프로세스를 실행할 수 있음.

2. Operating System Structures

operating system service

OS는 사용자에게 실행 환경을 제공

- User interface
- Program execution : 메모리에 프로그램 로드 및 실행이 가능해야 함
- I/O operation
- File-system manipulation
- Communications
- Error detection : 오류 시 찾고 알려줘야 함 (예외처리)

시스템 자체 효율적인 함수

- Resource allocation : CPU cycle, main memory, 파일 storage, I/O device 등
다수의 작업을 할 때 리소스를 서로 써서 효율성 보장함.
- Logging
- accounting : 사용자가 얼만큼의 리소스를 사용하는지 알아야 해서 리소스 사용량
추적
- Protection and security

OS User Interface (UI)

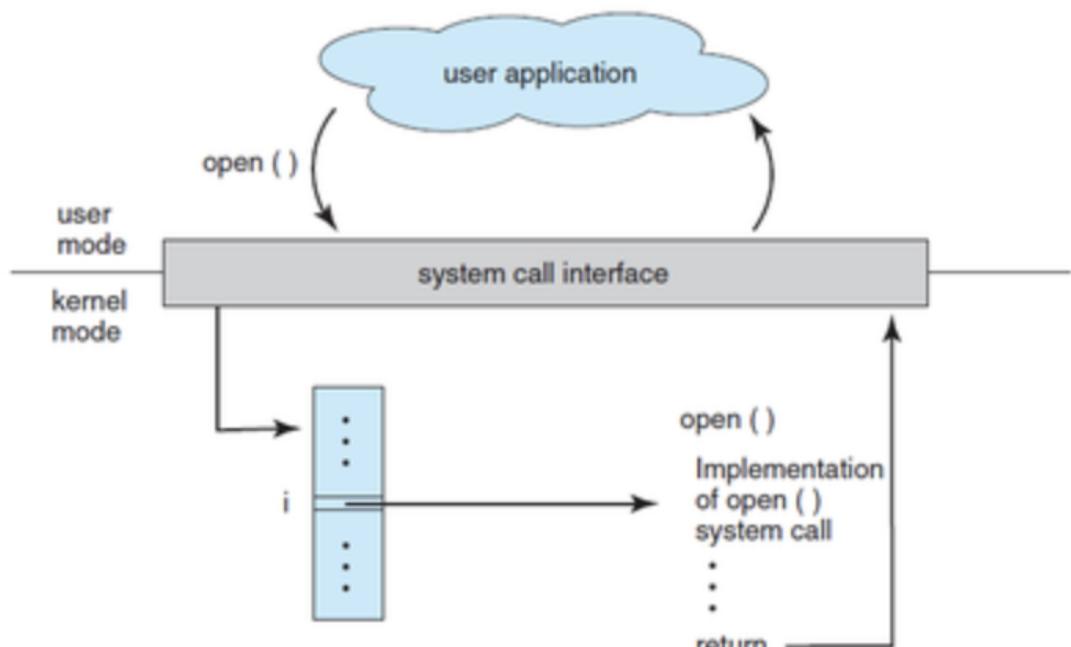
- CLI - Command line interpreter
 - 쉘, 커널, 시스템 프로그램으로 구현

- 사용자 기반
- GUI
 - I/O Device 아이콘
 - 마우스로 인터럽트 발생해서 실행함

Programming Interfaces

System call

OS에서 제공하는 프로그래밍 인터페이스이며, API를 통해 프로그램에서 주로 액세스함.
OS의 kernel로부터 application 프로그램을 요청함.



- system call open()

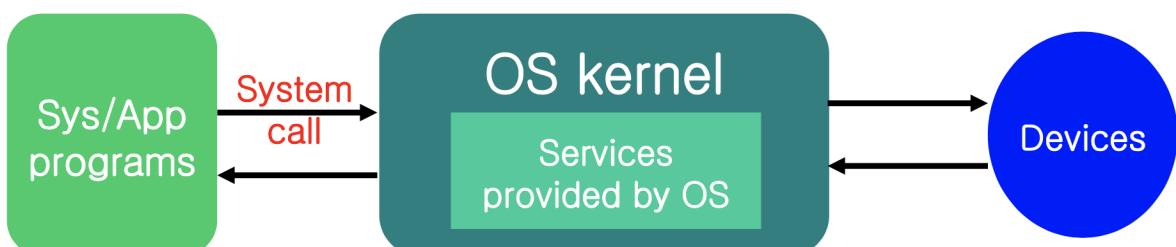


POSIX I/O system calls ⇒ [unistd.h](#)

```
| int open(const char *pathname, int flags, mode_t mode);  
| int close(int fd);  
| ssize_t read(int fd, void *buf, size_t count);  
| ssize_t write(int fd, const void *buf, size_t count)  
▼ size_t: unsigned int, ssize_t: signed int
```



POSIX system calls: [open](#), [close](#), [read](#), [write](#), [fork](#), [kill](#), [wait](#)



Dual Mode Operation

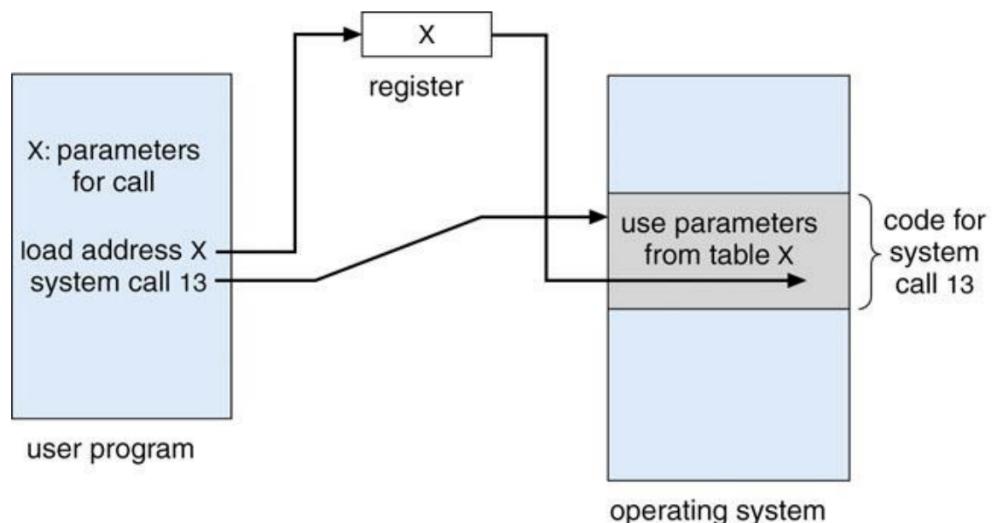
- User mode : user defined code
- Kernel mode : OS code
 - Privileged instructions are permitted

Interrupt Mechanism

Parameter Passing in System call

system call은 interrupt를 통해서 서비스 됨

- Parameter Passing methods
 - 레지스터
 - 메모리 내의 블록이나 테이블에 저장, 블록의 주소 레지스터 내 저장
 - 블록이나 스택에 저장함 (parameter의 길이 제한하지 않음)



Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communication



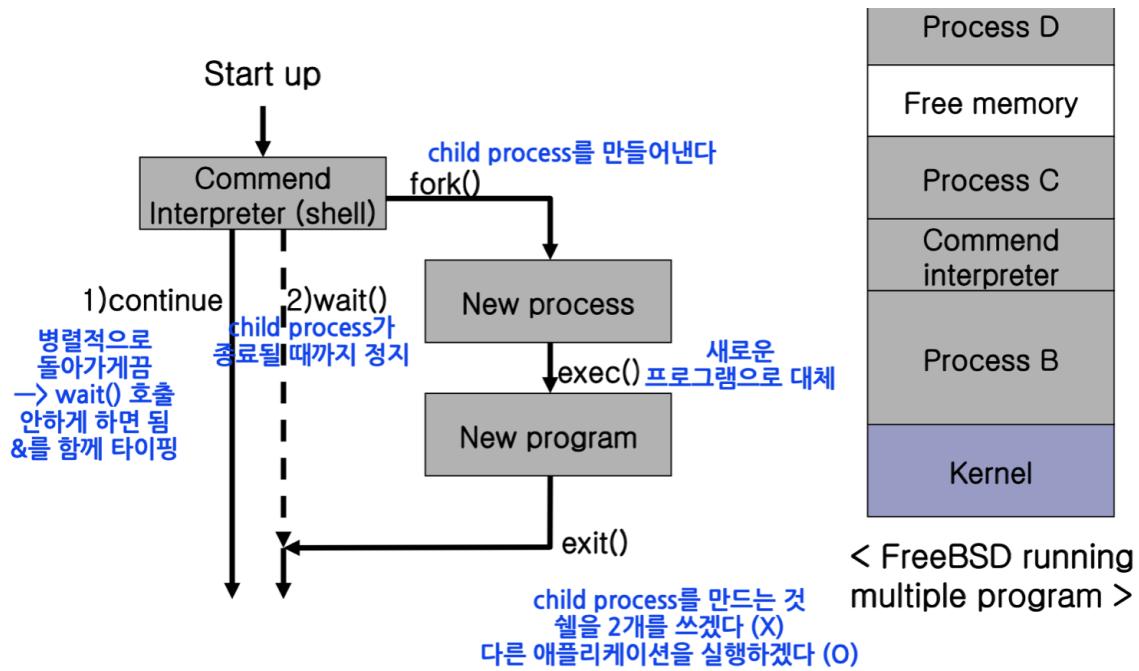
System Call interface

system call example

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

Process Control : Load / Execution

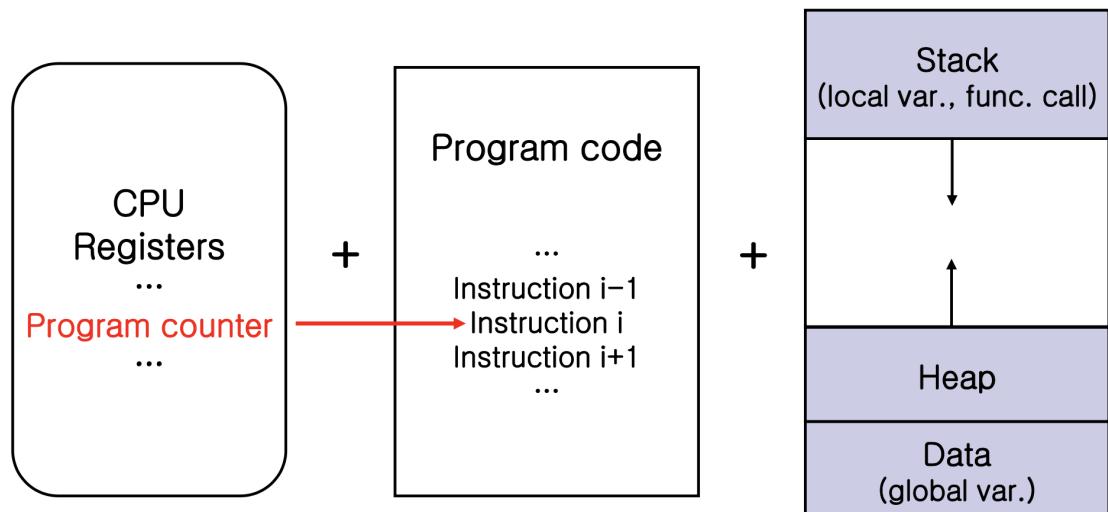
- 또 다른 프로그램을 load / execution



3. Processes 프로세스○○○○○○○ → 하기 싫은 느낌

빨리 정리할께유 😊😊

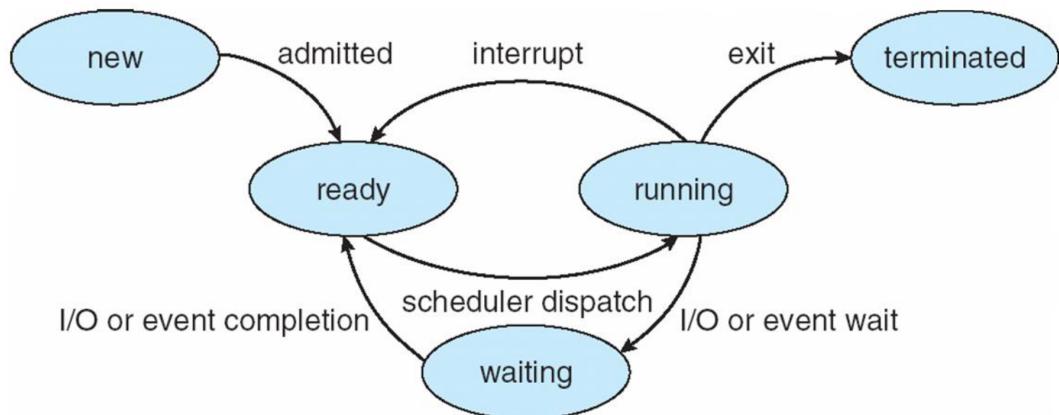
process는 프로그램 안에서 실행 + 리소스



Process State

프로그램이 메모리에 할당되어 실행될 때 프로세스의 상태가 5개의 상태 중 1개로 변하게 됨

- new : 프로세스 생성 상태
- running : 실행 상태
- waiting : 이벤트 발생으로 프로세스 대기 상태
- ready : 프로세서에 할당하기 위해 프로세스 대기 상태
- terminated : 실행종료



PCB(Process Control Block)

Process State	프로세스 상태 값
Process counter	다음 실행 명령어의 위치값
Process number	pid (process ID)
CPU register	프로세스 중심의 레지스터 내용
CPU scheduling	위선순위 정보, 스케줄링 큐포인터 등
Memory-management info	프로세스의 할당된 메모리
I/O status info	프로세스에 할당된 I/O 장치, 파일목록
Accounting info	CPU 시작이후 경과된 시간, 시간 제한 정보 등

Scheduling : 리소스(resources)에 할당된 일

Process scheduling : CPU에 실행되는 프로세스를 선택

- 한 번에 하나의 프로세스만 실행
- 다른 프로세스들은 대기(wait)

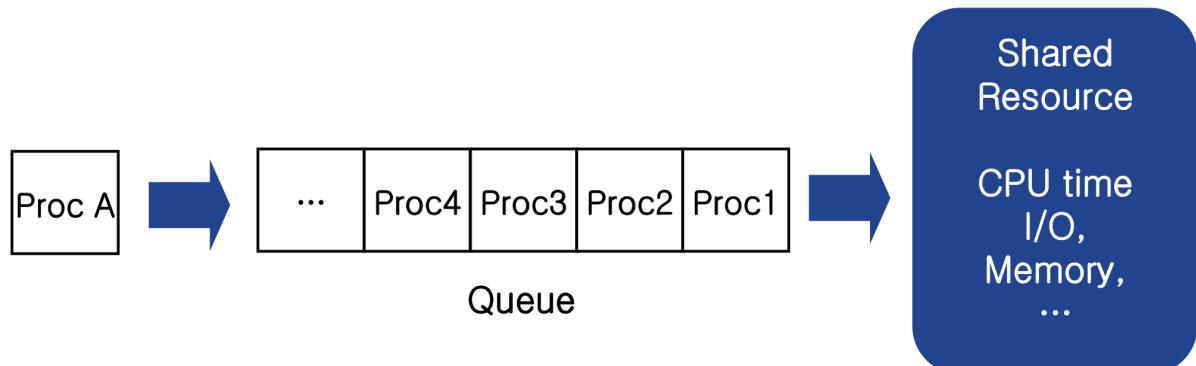
스케줄링의 목적 → CPU utilization 최대화, 각 각의 프로그램과 상호작용

- 어떤 시간대에 어떤 process가 실행될지 결정한다. (지금이 기준이다.)
- Process scheduler는 CPU에서 다음에 실행될 수 있는 프로세스들 중 선택을 한다.
- 각 queue에는 PCB가 들어가 있다.
- Job queue - 모든 프로세스의 집합
- Ready queue
- Device queue
 - 각 I/O device별로 device를 관리한다.

Scheduling Queue

resource 또는 CPU time에 대한 프로세스의 waiting list

- Ready queue, job queue, device queue



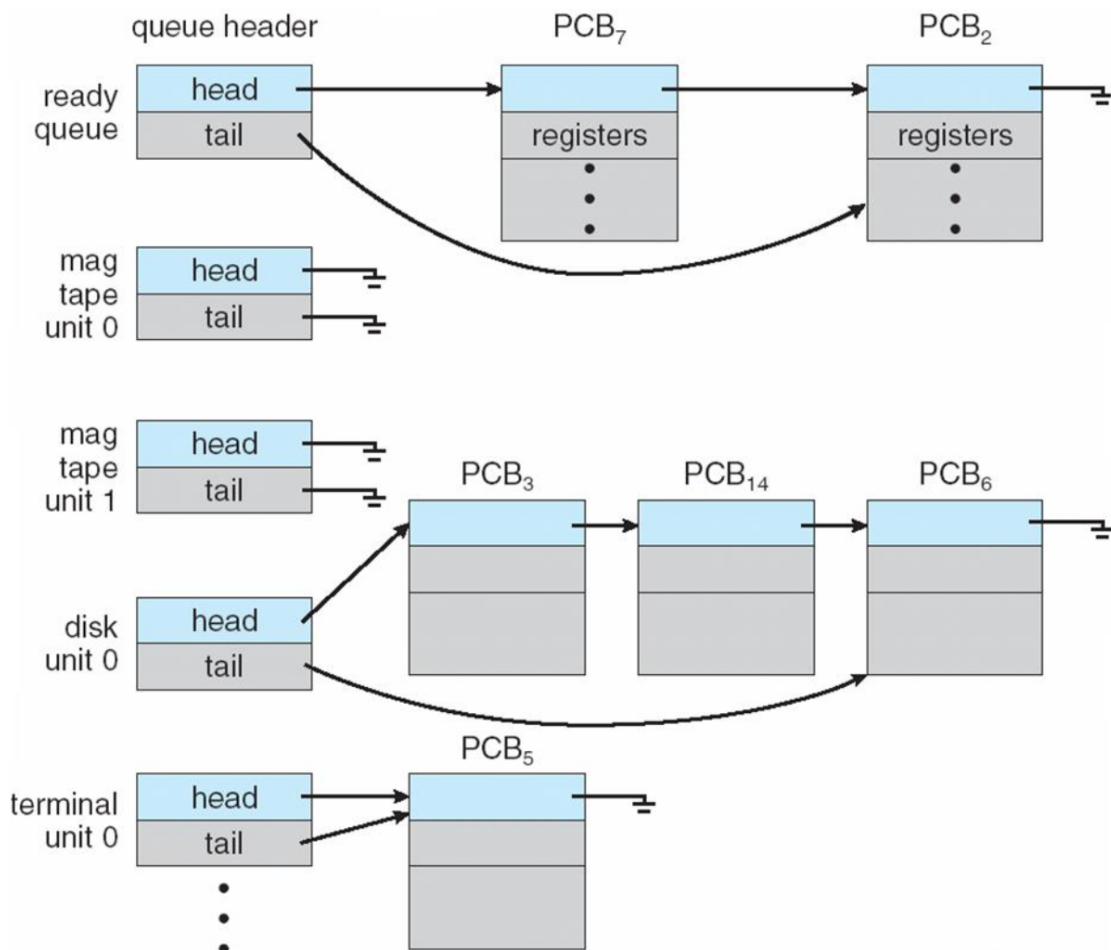
```

pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */

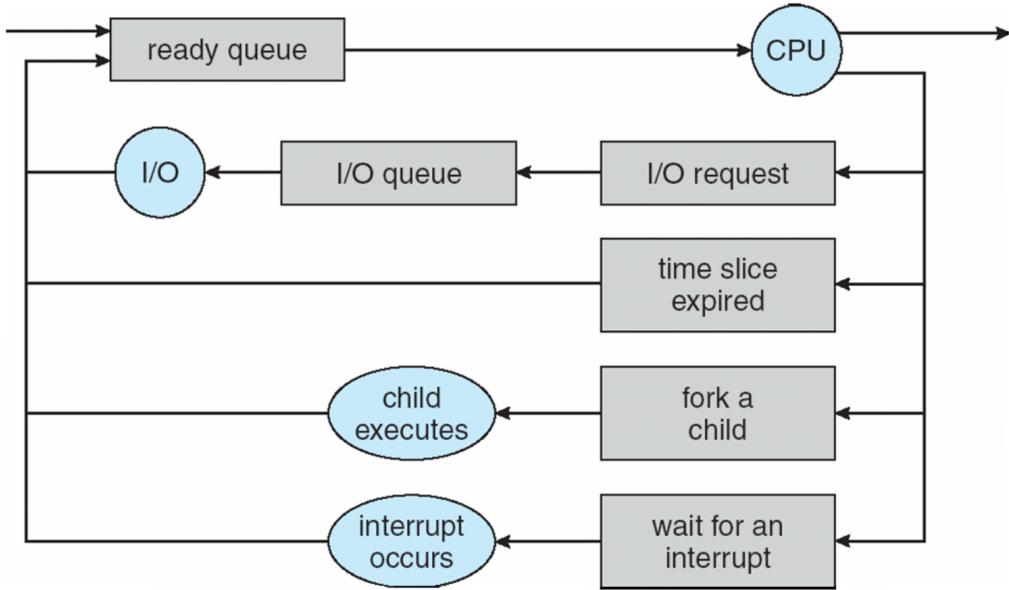
```



각 queue는 PCB들의 linked list로 표현됨



Queueing Diagram



Schedulers

queue로부터 processes 선택함

- Short-term scheduler(CPU scheduler or process scheduler)
 - 굉장히 자주 돌며 누가 실행되어야 하는지(CPU를 사용하는지) 확인해주는 스케줄러
 - 시간을 엄청 많이 나누어 그 순간마다 여러 프로세스를 실행시키는 즉, 멀티테스킹을 하게 해주는 스케줄러
 - ready queue에서 빼내어 실행시킴



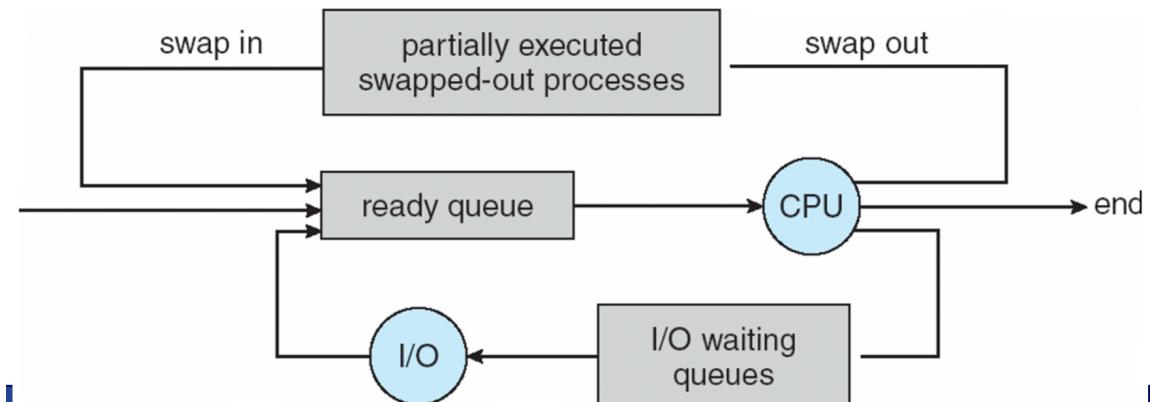
- Long-term scheduler (job scheduler)
 - ready queue에 넣을 수 있는 프로세스를 확인하고 넣어준다.
 - 일을 안하면 ready queue에 프로세스를 잘 안넣어주게 되는데 이렇게 되면 short-term scheduler는 있는 것만으로 실행되게 하므로, 멀티테스킹이 힘들어진다. 그래

서 long term은 멀티프로그래밍의 정도를 제어한다고 볼 수 있다.

- I/O bound 와 CPU bound process를 확인하여 전반적인 시스템 유ти리티가 높아 지도록 밸런스를 조정해주어야 한다.



- Medium-term scheduler → time sharing system
 - 한정적인 메모리에서 실행될 process와 아직 실행하지 않아도 되는 process를 실행시키기 위해 아직 실행하지 않아도 되는 process를 메모리에서 제거하고 disk에 두고 그 자리에 실행될 process를 넣어줌. 이를 **swapping**이라 하며 medium-term scheduler가 담당함.
 - Performance(속도)가 느려지지만 여러 프로세스를 실행시키기 위해서는 꼭 필요 한 작업임.



Context Switch

여러 프로세스가 CPU 리소스를 공유할 수 있도록 CPU의 상태(컨텍스트)를 저장하고 복원하는 프로세스

- interrupt가 들어오는 순간 발생한다.
- PCB에 data를 저장, 새로운 PCB에서 data를 load한다.

- Context switch time은 overhead
- 상태를 저장할 수 있는 공간이 제한적이라 필요
- 조금 더 많은 registers를 제공하면 time을 줄일 수도 있음

Operations on Processes

Process Creation

- Process Tree : OS 부팅 과정에서 첫 process(init)가 생성된다. OS가 동작하기 위한 process들을 생성해준다. 생성된 process들은 또 다른 process들을 생성해주는 형태로 진행된다.
- creation-process system call : 부모로부터 자식 프로세스 생성 (프로세스 id)
- 각 process에는 각기 다른 id를 갖고 있다.
- 자원 관리 - OS 정책에 따라 달라진다.
- 실행 관리 - 부모 자식 동시에 실행, 자식이 끝나면 부모 실행
- Linux에서는 Process를 생성한다는 것을 fork()라고 함 → pid return
 - 부모 프로세스에서 자식의 pid를 return
 - 자식 프로세스에서 zero return
 - Parent 복사본 or 독립적인 프로그램 2가지 유형으로 생성된다.
 - fork()는 하나의 process를 두 개로 분리시키는 작업이다.
 - fork()를 하면 부모 process와 똑같이 만들어진다.
 - 주소와 id만 다르다.
 - exec()계열 함수를 사용하면 fork()한 process의 contents를 변경해준다.
 - exec()할 때 메모리를 더 필요로 하면 OS에게 요청을 해준다.
 - Thread가 3개, 그 중 하나가 fork()를 호출한다면 3개의 thread를 생성하는가 아니면 process를 생성하는가 ? OS에 따라 다르다.
 - Linux에서는 process 단위로 fork()를 한다.
- 그렇다면 parent와 child 중 누가 먼저 실행될까 ? -> 모른다.
 - scheduler 마음이다. (wait를 걸지 않았기 때문에 경쟁적으로 실행된다.) ready queue에 둘 다 넣어지고 이는 scheduler가 선택한다.

- wait가 걸리면 child가 끝날 때까지 parent가 기다린다.



parent가 wait()하고 child가 exec()를 하도록 구분해주어야 한다.

- fork()의 return 값이 0이면 child, 0이 아니면 parent이다. parent인 경우 return 값이 방금 생성시킨 child의 id이다.

To creation process options

Resource	자식 프로세스가 직접 자신의 리소스 요청, 부모 프로세스 리소스 공유
Execution	동시 실행, 부모 프로세스 자식 끝날때까지 기다림
Address space	프로그램 코드와 데이터 공유, 자식 프로세스는 새로운 프로그램 로드

■ Functions in **exec** family (declared in <unistd.h>)

- int **execl**(const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/);
- int **execv**(const char *path, char *const argv[]);
- int **execlp**(const char *file, const char *arg0, ..., const char *argn, char * /*NULL*/);
- int **execvp**(const char *file, char *const argv[]);
- **execle()**, **execve()**

cf. Argument names

- path : path name that identifies the new process image file
- file : the new process image file. If '/' is not included, the corresponding file is identified through directories in PATH environment variable

| pid_t wait(int *stat_loc)

- stat_loc : integer 포인터

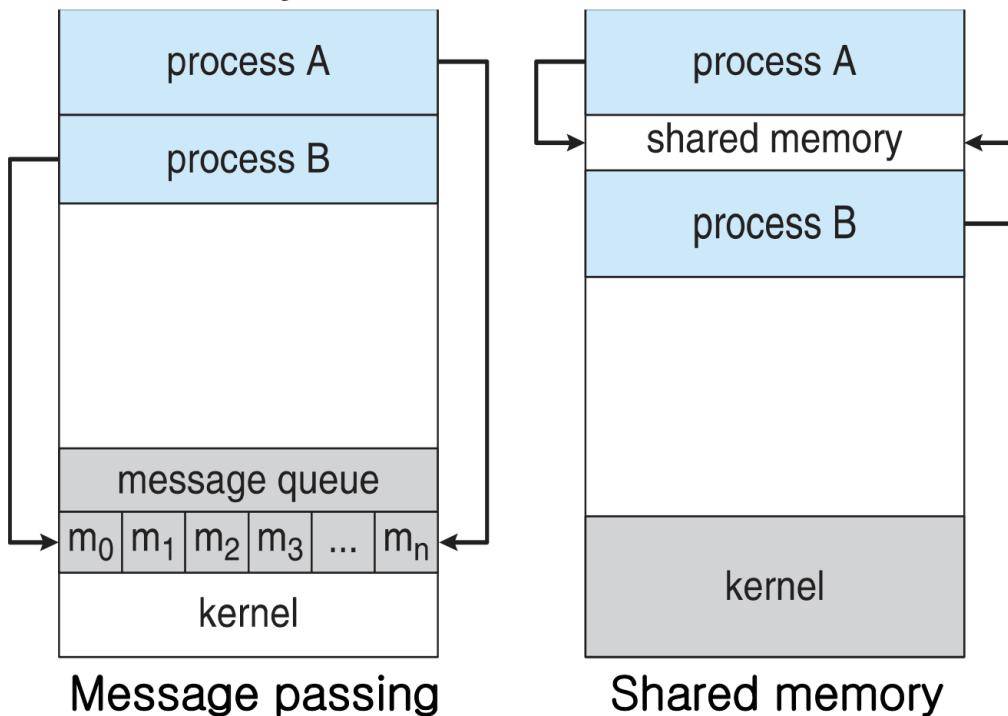
- if `stat_loc == NULL` 이면, 무시
- 다른 경우 : 자식 프로세스로부터 상태 정보를 받음
 - 부모 프로세스 → `wait(&stat);`
 - 자식 프로세스 → `exit(code);`
 - `code == (stat >> 8) & 0xff`
- `wait`의 return value
 - alive인 자식 프로세스의 pid
 - -1은 자식 프로세스가 존재하지 않음

Process Termination

- 프로그램이 종료될 때 `exit()`이라는 시스템 콜이 호출이 된다.
- Resource를 deallocate !
- `wait()`는 단순히 기다리는게 아닌 resource allocate & deallocate를 담당하는 프로세스에게 init을 시키기 위함
- `abort()`, child에게 resource를 allocate를 했는데 문제가 발생할 때 사용
- cascading termination
 - 부모 프로세스가 종료되면 모든 자식 프로세스를 `abort()`한다. (OS가 부모 프로세스가 종료되는걸 발견하자마자 자식 프로세스들은 `abort()`)
 - orphan process 부모 프로세스가 없는 경우로, `init()`에 가져다 붙인다. 부모 혼자 `wait()` 호출 없이 종료된 경우 부모 프로세스가 자식 프로세스보다 먼저 종료된 경우
 - zombie process 종료될 때까지도 `wait()`를 호출하는 parent가 없는 경우로, parent가 메모리 회수 작업을 해줘야 하는데 해주지 않는 경우 부모 프로세스가 있고 자식 프로세스는 종료되었는데 `wait()`를 통해 메모리 회수를 해주지 않는 경우
 - 메모리 관리 측면에서 두 프로세스를 신경써야 함
 - OS가 background에서 프로세스를 관리하는 것 또한 프로세스이기에 두 프로세스를 찾아서 대처를 해야하는 방안이 있어야 함

- wait를 한다고 꼭 해결되는 것은 아니다. cascading termination이 아니라면? 고아 프로세스가 생길 수 있음.

Inter-Process Communication (IPC)



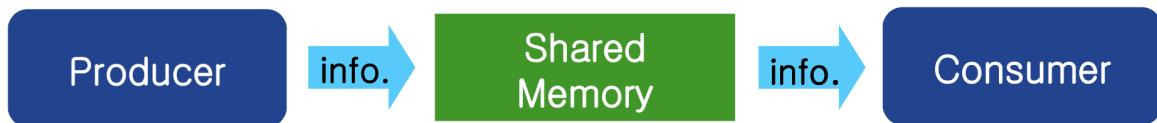
Shared-Memory Systems

- shared memory segment

프로세스간 통신을 하기위한 공유메모리 영역은 OS영역이 아닌 사용자 프로세스의 제어

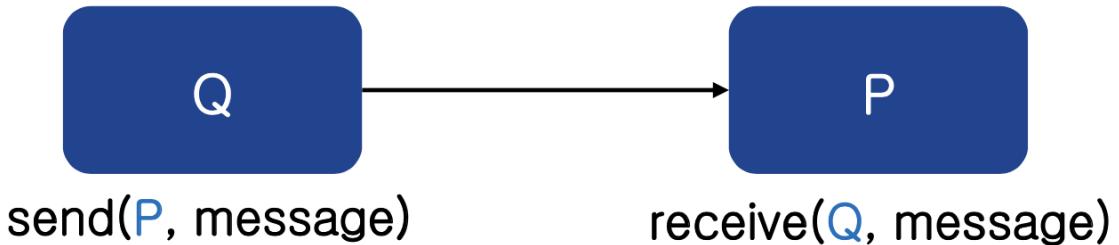
여기서 중요한점은 공유메모리에 사용자가 접근시, 두 스레드 혹은 프로세스가 동기화 할수 있는 방법을 제공해야한다.

Producer-consumer problem

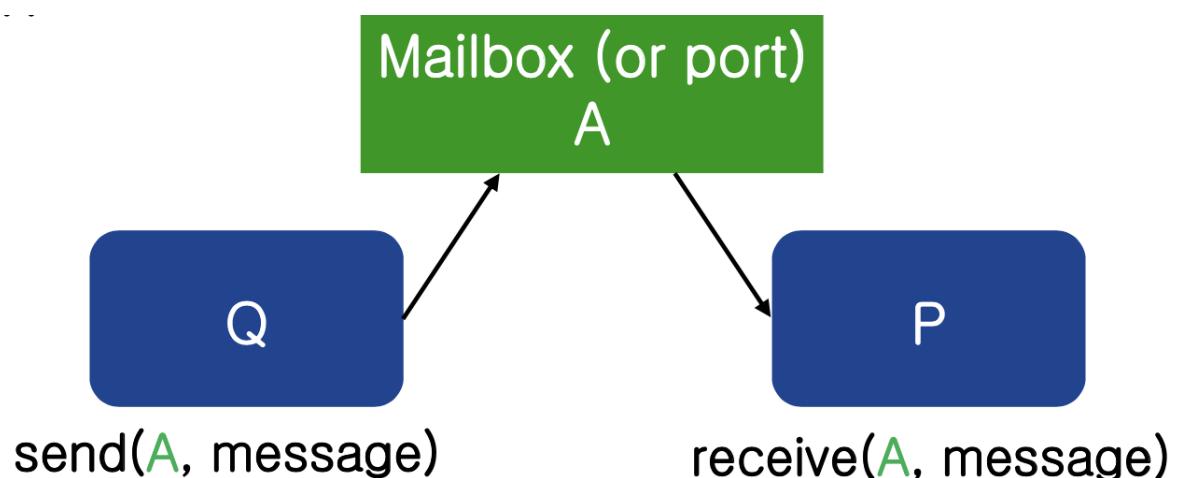


two types buffer이 있음

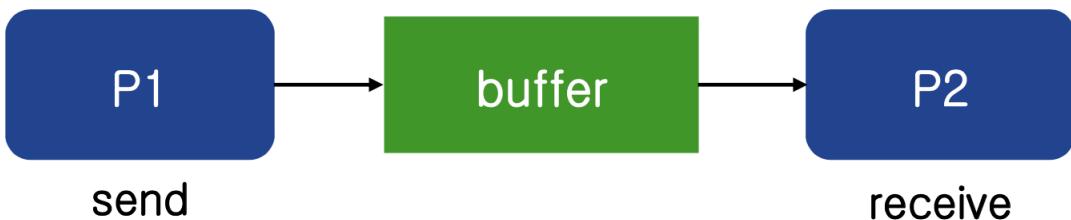
- unbounded-buffer는 무제한 크기의 버퍼
- bounded-buffer는 고정된 크기의 버퍼
- consumer 입장에서는 data를 underflow하지 않다면 읽는, producer 입장에서는 data를 overflow하지 않다면 데이터를 넣는 작업을 한다. **shared memory solution**
- OS는 메모리만 할당, 프로세스들이 독립적으로 communication을 관리한다.
- 완벽은 아니지만 synchronization을 어느정도 제공한다. **message passing**
- 메시지의 크기가 동적인경우
 - 정적으로 정한 크기보다 작을 때, 크기보다 클 때 유용하다. 작으면 메모리 낭비를 막을 수 있고, 클 때는 굳이 나누지 않고 한번에 보낼 수 있다.
- 메시지의 크기가 정적인경우
 - 크기가 정해져 있기에 메시지 파싱이 쉽고 빠르다.
- 분산환경에서 유용하게 쓰인다.
- communication link를 만들어야 한다.
- Direct Communication (주소를 명시)
 - Q라는 process가 P라는 process에게 받고싶으면 receive(받고 싶다는) 메시지를 보내고 보내고 싶으면 send(보낸다)메시지를 보냄.
 - 한 쌍에 하나의 link가 생긴다. 보통 양방향
 - Symmetric links



- Indirect(mailbox인 message queue를 사용)
 - Mailbox가 존재하여 이곳에 message를 넣으면 누군가 가져가는 방식 (1:1 pair link가 아니고 explicit하게 명시하지 않음.)
 - message sharing의 문제가 생긴다.
 - mailbox에 있는 message를 누가 먼저 가져갈까 ?OS가 정해준다. 최대 2명만 mailbox를 공유하도록, 오직 한 명만 메시지를 가져가도록 등...



Buffering



- rendezvous 방식이면 임시저장공간이 필요 없다. (zero capacity 또는 no buffer)

- non-blocking에서는 bounded capacity를 사용한다.
- unbounded capacity 서버가 기다릴 필요 없이 계속 받아들인다. 이론적으로 overflow 가 없고 실제로 존재하지 않음(메모리가 한정되어 있으니까). 모델링에만 쓰임

Three Types

- zero capacity : blocking send 가능함
- Bounded capacity : buffer는 제한된 길이
 - buffer 가득참, sender는 blocked
- Unbounded capacity : buffer는 무한함.
 - sender는 결코 blocks하지 않음

shared-Memory API

 **shmget()** ⇒ Create shared-memory

| **int shmget(key_t key, int size, int shmflg);**

key : Key of shared memory segment

- key는 ftok()함수를 통해서 생성된 값이나, 또는 임의의 숫자를 사용한다.
- IPC_PRIVATE : key를 IPC_PRIVATE로 설정하면, key값은 중복되지 않는 임의의 값으로 자동으로 생성된다.

size : size of shared memory segment

- 할당할 메모리의 byte단위 크기
- 주로 Buffer size가 들어간다.

shmflg : flags

- shared memory에 대한 설정값으로 IPC_CREAT, IPC_EXCL, access 권한 9 bit의 bit or 연산으로 설정한다.

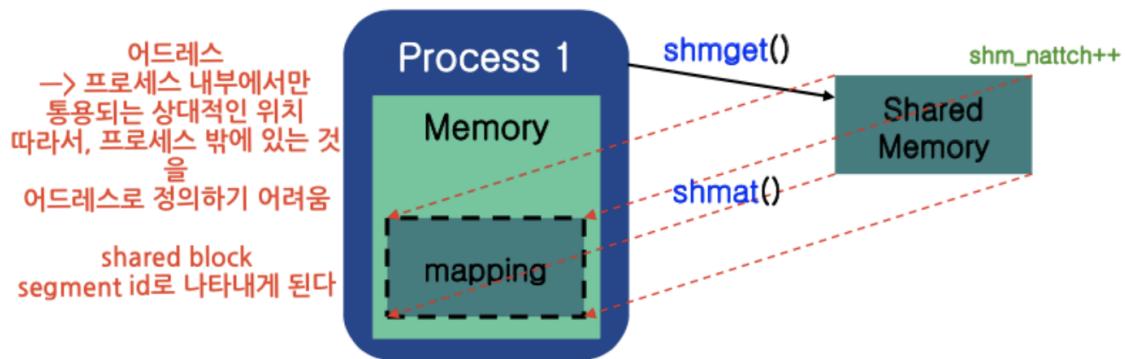
- S_IRUSR : 사용자 읽기 (사용자가 파일과 디렉토리 항목들을 읽을 수 있다.)
- S_IWUSR : 사용자 쓰기 (사용자가 파일을 기록, 제거, 생성할 수 있다.)
- IPC_CREAT : shared memory를 생성한다.
- IPC_EXCL : key로 생성된 shared memory segment가 없는 경우에만 생성하고, 이미 같은 key로 shared memory가 생성되어 있다면 오류가 발생한다.
- ? 둘다 포함되어 있지 않은 경우 ? 이미 만들어진 shared memory에 대한 shm id를 return한다.

```
shm_id = shmget(IPC_PRIVATE, sizeof(Buffer), S_IRUSE | S_IWUSR);
shm_id = shmget((key_t)1234, sizeof(Buffer), IPC_CREAT | 0666);
```

shmat() ⇒ 프로세스의 주소 공간에 shared-memory

| **void shmat(int shmid, char *shmaddr, int shmflg);**

Ex) shared_mem = (char *) shmat(seg_id, NULL, 0)



- **shmid** : shmget() 리턴으로 얻은 세그먼트 ID
- **shmaddr** : attach 되도록 기대되는 메모리 주소 (보통 NULL)
- **shmflg** : attach flags



공유메모리는 attach된 주소부터 쓰어지게 됨

```
shared_mem = (char *)shmat(seg_id, NULL, 0);
sprintf(shared_mem, "writing to shared memory");
```

shmdt() - detach shared memory from address space of process

```
| void shmdt(char *shmaddr);
```



shmctl() - deallocating a shared memory block

```
| shmctl(shmid, IPC_RMID, NULL);
```



shm_attach가 0이 되었을 때, shared memory block을 deallocate함

POSIX Shared Memory

memory-mapped file

- ✓ **shm_open()** - process creates or open shared memory segment

| **shm_id = shm_open(name, O_CREAT | O_RDWR, 0666);**

```
O_CREA : creat if it does not yet exist ; 아직 존재하지 않으면 만들어라  
O_RDWR : open for reading and writing
```

- ✓ **ftuncate()** - set the size of the object

| **ftuncate(shm_id, 4096);**

- ✓ **mmap()** - map shared memory segment to process address space

| **mmap(0, 4096, PROT_WRITE, MAP_SHARED, shm_id, 0)**



메모리를 매핑은 파일을 프로세스의 메모리에 매핑

```
Buffer *buffer = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_id, 0);
```

```
MAP_SHARED : 다른 유저와 데이터의 변경 내용을 공유한다.  
PROT_READ : 매핑된 파일을 읽기만 허용한다.  
PROT_WRITE : 매핑된 파일에 쓰기를 허용한다.  
PROT_EXEC : 매핑된 파일을 실행할 수 있다.  
PRO_NONE : 매핑된 파일에 접근할 수 없다.
```

▼ 매핑이 성공하면, mmap은 공유메모리의 포인터를 리턴한다.

munmap(), shm_unlink()

공유메모리 사용을 마쳤으면 위의 2개의 함수로 unmapping과 unlink를 하여야 한다.

만약, 수행하지 않는다면 프로세스는 종료되어도 메모리 영역에 공유메모리가 제거되지 않아 비정상적인 작동으로 이어질 수 있다.

- Procedure call을 원격으로 보낸다. ex 이런 (함수)계산을 해줘
- stubs, 사전에 동의가 되어 있는 상태에서 서로의 PC 사이에 communication을 위해 통역을 해주는 역할을 한다.(marshalling이라 한다.) endian(big-endian little-endian) 이 다를 때 marshalling 작업 필요