

1.1 Introduction to OS

relationship

- user > sw > hw
- sw: application software > operating system
- OS ex. ios, android, windows

role

- performance ↑
 - **resource management, allocation**
 - SW management: manage process, file,,,
 - processor management: manage CPU
 - memory management: manage memory
 - file management: manage HDD
 - IO management: manage keyboard, monitor, printer
 - resource protection ex. prevent from monopolizing CPU
- convenience ↑
 - hw interface
 - user interface
 - ex. GUI

1.2 Architecture of Computer

CPU scheduling (Processor management) Memory management Disk(HDD) scheduling Interrupt 메모리에 여러 프로그램이 올라가면 cpu에서 읽어가서 실행 -> 1cpu이기에 매우 빠른 switching으로 illusion 제공

1.3 internal parts of computer

on motherboard

: register,

1.4 external parts of computer

not on motherboard

I/O devices: monitor, keyboard, mouse

- kernel: resource manager in OS
전원 off 될때까지 메모리에서 계속 상주하는 부분
- Shell = command interpreter
: User 명령 받아서 해석하여 실행

hdd 비휘발성에 보조기억장치 os에 -> memory dram 주기억장치 휘발성 전원이 꺼질때부터 꺼질때

2.1 basic configuration

international units

- (SI) $1\text{km} = 10^3\text{m}$ / (IEC) 2^{10}
 - $\text{K} < \text{M} < \text{G} < \text{T}$
 - $1 < 2 < 3 < 4$
- (SI) $1\text{mm} = 10^{-3}\text{m}$
 - $\text{m} > \mu > \text{n} > \text{p}$
 - $-1 > -2 > -3 > -4$

how digital system works: **clocking**

one cycle = 1-0

- T(clock period): __seconds / one cycle
- F(clock frequency): __cycles / a second

=> $T * F = 1$

2.2.1 CPU

how to execute a source file

1. second memory: compile process
 - $.c \rightarrow (\text{compiler}) \rightarrow .obj \rightarrow (\text{linker-lib}) \rightarrow .exe$
2. load on main memory
 - instruction memory | data memory
3. process the Instruction cycle between main memory <-> CPU

Instruction Cycle

1. **Fetch** the addresses of instructions stored in instruction memory to **Program Counter** of CPU
2. **Decode** the instructions(machine code) by each processors and set the configuration of registers and ALU on **Control Unit** of CPU,
3. **Execute** it on **ALU**
4. **Store** it on GPR or data memory
 - **registers**: faster than memory
 - General Purpose Register: data processing...
 - Special Purpose Register: for PC, IR, SP...

2.2.1 Memory

Types

- short-term memory

- **Dynamic Random Access Memory**: in main memory
- **Static Random Access Memory**: in Cache of CPU
- long-term memory
 - flash memory: in USB
 - Solid State Drive: replacing HDD

Hierarchy

CPU: register > processor > cache > main memory > HDD

-----> price↓ speed↓ size↑

- Processor: gives 8bit(1byte) **address** to memory and the memory gives back **data** by the address to the processor

Why Protection is needed

now time-sharing system => can invade other program's memory: interrupt -> terminate it

Booting Process

: load operating system to memory

- ROM's command to CPU
 1. POST: test
 2. CPU copies boot-strap-code from 0 sector in HDD to RAM
 3. CPU executes boot-strap-code by kernal of RAM which was copied from kernal in HDD by the process of OS

2.3 computer performance improvement technology

Dual mode when CPU works

CPU <-> Program(user+OS)

- Kernel mode by kernal process relevant to the OS (more authority)
 - mode_bit = 0
- User mode by user process
 - mode_bit = 1

Hardware Mechanisum

Polling

: in order(X)

- CPU steadily checks whether the device needs attention

```
void main(void){
    while(1){
        if(button_pressed)
            do_something();
    }
}
```

Interrupt

: in order(O)

- the device notices the CPU that it requires its attention. Interrupt can take place at any time.

```
Interrupt_Service_Routine(BUTTON_PRESS_vect){
    do_something();
}

void main(void){
    setup_button_press_interrupt();
    while(1){
    }
}
```

- request μP to work instantly for the event
- user mode -> kernel mode

Types

- S/W Interrupt** via Kernal of OS
 - system call: the job when program1 requests OS to interrupt -> Program Count moves from program1 to ISR of OS
 - exception: by checking **what mode** of CPU now via mode_bit
- H/W Interrupt** via μC
 - : the job when IO device requests OS to interrupt for program1 -> Program Count moves from the program1 to ISR
- Timer Interrupt**: Time Sharing System by setting timer via time slice(The period of time for which a process is allowed to run in a **preemptive** multitasking system is generally called the *time slice or quantum*)

3.1 Introduction to Process

program vs process

program	process
static state	dynamic state

program	process
on HDD	on memory for execution
before creating	before terminating
process - PCB	program + PCB

1. A **program** on HDD got a **PCB** from OS space of memory => A **process** on user space of memory
2. The **process** returned the **PCB** => program

Architecture

elements of memory

1. user space
 - process A, process B
 - code: program code (RONLY)
 - data: global variables (RDWR)
 - stack: local variables, pointer of program address
2. operating system space
 - **PCB** A, **PCB** B....

3.2 Operation of Process

process hierarchy

0. 'init process' when booting by OS
1. **fork() system call**: starts child process which is a copy of the parent process
 - copy all but not PID, Parent PID, Children PID
 - pid = fork() -> pid = 0: child(o), -1: error
2. program loads to the child process
3. **exec() system call**: replaces the current process image with new one
4. if child exit(PID) -> parent collects its data
 - data collection fail => **Zombie Process**
 - no parent process => **Orphan Process**

3.3 States of a Process

1. new + PCB => **ready**: waiting in the ready queue <-> running: one process executed by CPU => -PCB
=> terminated
2. **running**: during time slice
 - active (CPU)
 - after time slice -> timeout(PID) => ready
 - IO -> **blocked**(PID): connecting PCB by pointers -> wakeup(PID) => ready
 - inactive (swap area)

- expelled, delayed -> swap io -> **suspended ready** -> wakeup -> **suspended blocked** -> swap io => ready

3. terminated: code & data eliminated from memory and -PCB

- normal - exit() / abnormal - core dump

3.4 Process Control Block & Context switching

PCB

- OS relevant: PID, pointer, process state
- CPU relevant: Program Counter, registers
- resource relevant: memory limits, list of open files

Context switching

re-scheduling

why

time-sharing system

when

io/timer interrupt -> ISR or context switching

process context

: HW / memory / kernel context

how to switch

p2 ready, p1 running -> timeout -> store p1 state in PCB 1 -> get p2 state from PCB2 -> dispatch p2 -> p2 running, p1 ready

disadv

overhead => [multi-thread](#) to overhead ↓

3.5 Thread

What is Thread

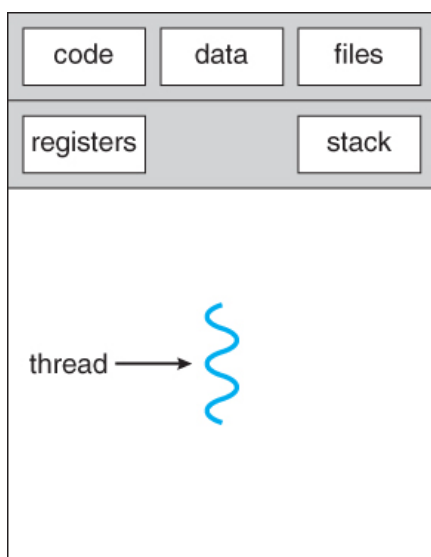
- Lightweight Process: unit of CPU
 - a process > [multiple threads](#)
- Hyper Thread: a set of 2 registers
 - when context switching, overhead ↓

Concurrency

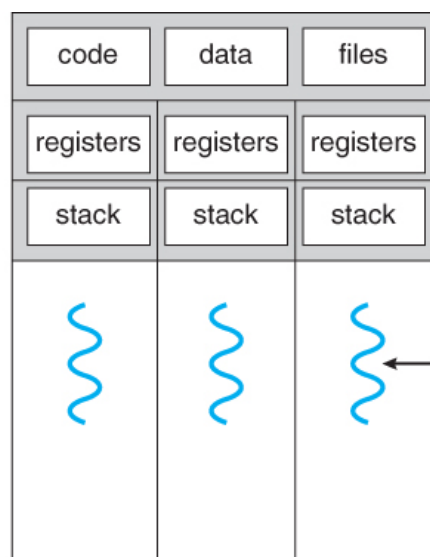
- *one by one process(share)*: address space(Data, Code area), resource(OS) in one process
 - the address space of a Process = **stack**(one each) + *data(share)* + *code(share)*
- **one by one thread**: Stack, Program Counter, Registers
 - Process Control Block = **Program Counter + Registers** > Thread Control Block = thread ID + Program Counter + Stack Pointer

Multi-Thread

- multi-task: multiple processes(single-thread) by fork() -> copied => waste
 - ex. chrome
 - +) security, independency of each webpages
- multi-thread: multiple threads in a process -> shared => no waste
 - ex. explorer
 - +) time, resource, efficiency



single-threaded process



multithreaded process

4.1 Introduction to Scheduling

What is scheduling

the method that allows **when and what process**

cpu resource can be assigned by order of ready queue of *PCB* for more efficiency between the processes whose most jobs are using *CPU VS IO*

CPU-I/O Burst Cycle

how to execute a process

1. **CPU Burst**: when the process is being *executed in the CPU*
 1. Long CPU burst : **CPU bound job** ex. simulation program (complicated)
 - duration ↑ => frequency ↓
 2. Short CPU burst : **I/O bound job** ex. docx program (keyboard)
 - duration ↓ => frequency ↑
 - => *priority* of process ↑
2. **I/O Burst**: when the *CPU is waiting* for I/O for further execution
3. ready queue: the process goes into the ready queue for the next CPU burst

CPU Schedulers

Queuing diagram

how to express process-scheduling

1. Programs on HDD/SDD -> (long-term scheduler: order process) -> A process in the Ready Queue
2. Ready queue -> (short-term scheduler: select one PCB) -> (dispatcher: assign) -> Processor
3. Processor -> blocked, interrupted, paused
 1. (short-term scheduler) IO request -> IO Device Queue: blocked and waiting -> IO Device
 2. (short-term scheduler) time-out for assignment
 3. (short-term scheduler) interrupt
 4. (mid-term scheduler) swap out -> paused process -> swap in
4. -> Ready Queue

5. done

4.2 Considerations for Scheduling

Scheduling Algorithms

Preemptive/Non-Preemptive Scheduling

basis	Preemptive Scheduling	Non-Preemptive Scheduling
when can start a new process	suspended by OS <i>or</i> interrupted	unless <i>termination</i> of the process or I/O
---	response time ↑	throughput ↓ (even IO bound job has to wait for a long time)
e.g.	SRT, RR	FCFS, SJF, HRN

Process Priority

- kernal > general process
- foreground > background process
- interactive > batch process
- IO bound job > CPI bound job

4.3 Multilevel Queue

IO Interrupt

more than 2 io interrupts can happen simultaneously

1. mouse moved, interrupt happened -> current process: paused, out of CPU
2. temporarily store the info of the process
3. **Interrupt Vector Table**: address of ISR -> Interrupt Service Routine: code ->
4. CPU: handle that interrupt -> restart the process

Multilevel Queue

by priority of the multiple ready-queues

- ex. queue 0: HDD -> queue 1: CD-ROM -> ...

Priority

	Static Priority	Dynamic Priority
priority	cannot be changed	can be changed
work	↓	↑
efficiency	↓	↑

State

- ready**: dispatch *one* process
 - priority in PCB: insert process in that specific priority queue
 - CPU scheduler: dispatch the prior process to the CPU
- waiting**: make *multiple* processes be ready
 - check IV table: make the prior process in that specific priority queue be ready

4.4 Scheduling Algorithm

Priority Scheduling

- ex: FCFS, SJF, HRN, [static](#), [dynamic](#)
- : starvation(:convoy effect), overhead(:context-switching)-> low efficiency
- => solution: [aging](#), considering waiting time as well(SRT)

	FCFS	SJF	HRN	SRT
alleviation	First Come First Served	Shortest Job First	Highest Response ratio Next	Shortest Remaining Time
type	non-preemptive			preemptive(time slice O re-scheduling O)
priority↑	waiting_time↓	burst_time↓	(waiting_time/CPU_burst_length)+1↓	remained_burst_time↓
disadv	low efficiency ∴convoy effect(AWT↑)	starvation ∴when the first process takes too long	starvation still	

Round Robin Scheduling

RR: just in order of the ready queue

advantages

response time ↓

disadvantages

- time slice ↑ ≈ FCFS: AWT ↑ -> starvation
- time slice ↓ ≈ SRT: context switching -> low efficiency

Multilevel Queue Scheduling

by static priority

Multilevel Feedback Queue Scheduling

by dynamic priority

- each ready-queue: **priority** ↓ **time slices** ↑ the rate to burst CPU ↓ CPU burst length ↑
- ex: *I/O burst job*: TS ↓* -> CPU burst job*: TS ↑.. -> **FCFS**: TS = ∞ (∵ it has been moved lots.. 😞)
- *aging*: a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- hard to build and implement

Scheduling Criteria

what is better

CPU Utilization ↑ Throughput ↑ Waiting Time ↓ Response Time ↓ Turn-around time ↓

waiting time < response time < turn-around time

- waiting time: ~ start execution of process
- response time: ~ start first output of process
- turn-around time: ~ terminate and turn out all the resources

Algorithm Evaluation

w/ Gantt Chart

Average Waiting Time

- Waiting time is the sum of the periods spent waiting in the ready queue. / num of processes
- time until started - arrival time (- burst time)


Average Turn-around Time

- Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O. / num of processes
- time until turned around - arrival time

problem

► open

process	arrival time	execution time
P1	0	20
P2	1	14
P3	2	8
P4	3	16

 => AWT/ART: SRT < SJF < HRN < FCFS < RR