

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제1강

Operating System Overview

충남대학교 인공지능학과/컴퓨터융합학부

최 훈



학습 목표

- ❑ 운영체제(Operating System) 정의를 이해한다.
- ❑ 운영체제가 하는 일들을 이해한다.
- ❑ 앞으로 운영체제 과목에서 학습할 내용을 파악한다.

학습 내용

- ❑ 운영체제 정의
- ❑ 운영체제가 하는 일
- ❑ 운영체제 과목에서 학습할 내용



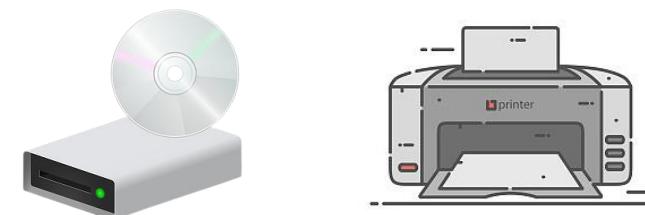


Operating System Overview



What is This Course About?

✓ Operating System, the Brain of Computer



Document
Editing

Music

Messenger

Game

Usage Scenario

✓ Kim wants to use a computer to write a C program

1. He powers on his computer.
2. He edits a source code for a homework.
3. After finishing the programming of the program, he starts the compilation of the source program.
4. While the compilation is under way, he begins the Hangeul program and writes the report.
5. His favorite song is played on a music player program.
6. He runs the program that he wrote.
7. After editing the report, he prints it.
8. Then he shuts down the computer.

Roles of Computer Systems

✓ Booting the computer

- ◎ Make the computer ready after power on

✓ Running the editor, compiler, music player

- ◎ Process management, (main) memory management

» Who dose the above jobs
→ OS

✓ Controlling the multiple programs

- ◎ Process scheduling

» How?
→ You learn it in this course

✓ Receiving the keyboard input

- ◎ I/O device management, file management

✓ Printing an output and controlling the printer

- ◎ I/O device management, file management

✓ Shutting down the computer

What is the Operating System?

✓ A software that controls the execution of application programs

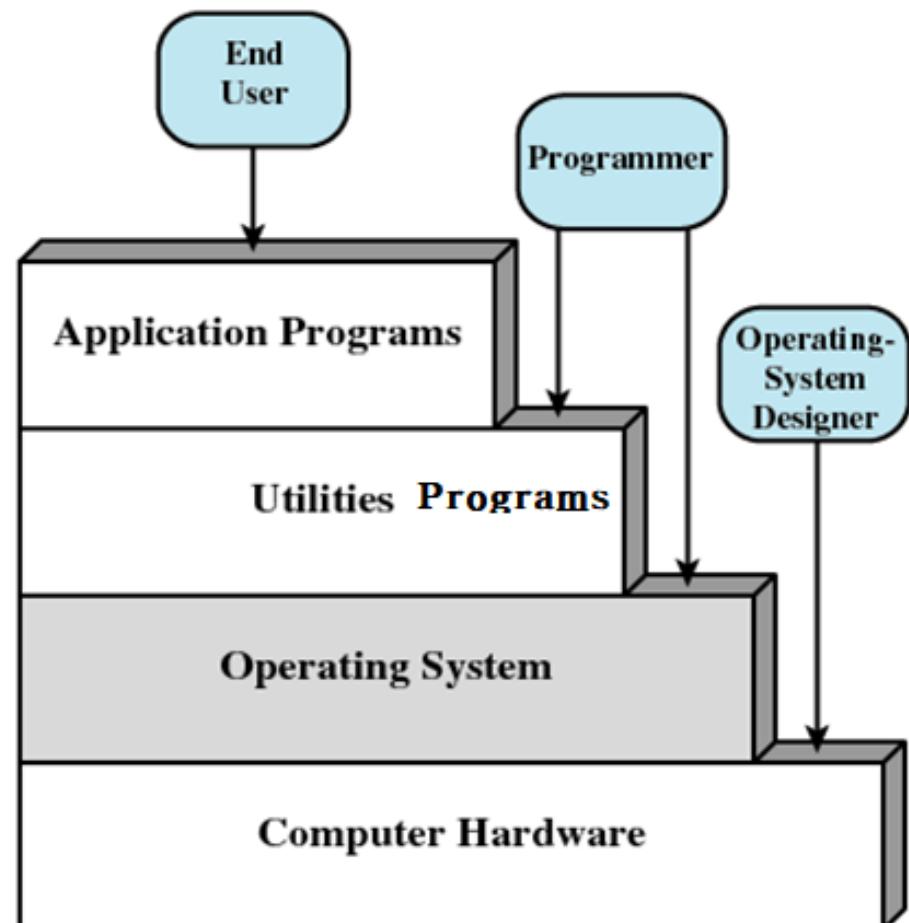
- ◎ It is a set of functions that is executed
- ◎ Windows, UNIX, Linux, Mac OS,

✓ An interface between programs and hardware

- ◎ Responsible for managing resources

✓ Objectives

- ◎ Efficiency
- ◎ Convenience



Services Provided by the Operating

✓ Program execution

✓ Access to computer resources

- ◎ Hardware : I/O devices and other hardware resources
- ◎ Data : files and other data

✓ Error detection and response

- ◎ Internal and external hardware errors
- ◎ Software errors
- ◎ Operating system cannot grant request of application

✓ Accounting

- ◎ Collect usage statistics
- ◎ Monitor performance
- ◎ Used for billing purposes

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제2강

Uni-programming,
Multi-programming

충남대학교 인공지능학과/컴퓨터융합학부

최 혼



학습 목표

- 💬 일괄처리 방식(Batch System) 정의를 이해한다.
- 💬 일괄처리 방식의 유니프로그래밍(Uni-programming) 방식을 이해한다.
- 💬 멀티프로그래밍(Multi-programming) 방식을 이해한다.

학습 내용

- ⌚ 일괄처리 방식 정의
- ⌚ 일괄처리 방식의 유니프로그래밍 방식
- ⌚ 멀티프로그래밍 방식





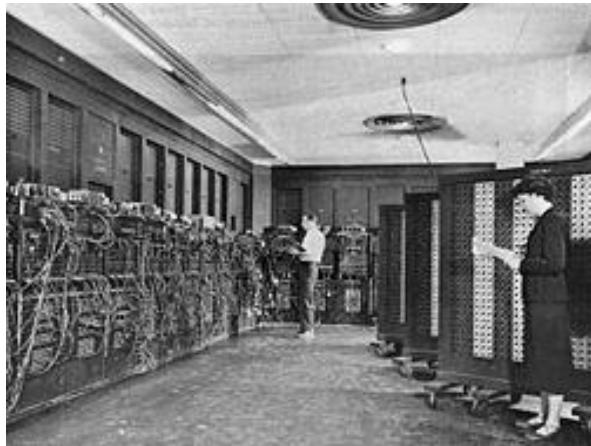
Uni-programming, Multi-programming



Evolution of Operating Systems

✓ Serial Processing (Operation by Human)

- ① No operating system
- ② Machines run from a console with display lights, toggle switches, input device, and printer
- ③ No job scheduling
- ④ Setup time included loading the compiler, source program, saving compiled program, and loading and linking



ENIAC, 1946

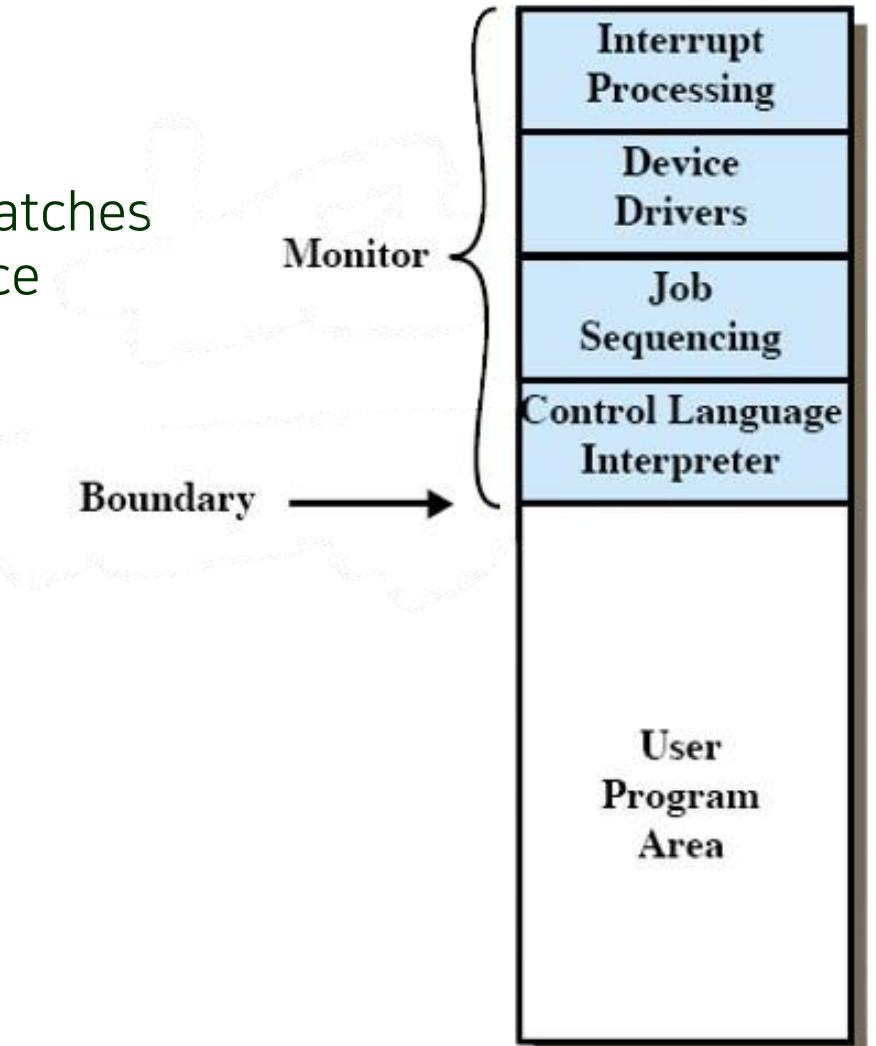
Simple Batch Systems

✓ Simple Batch System

- ④ Evolved from the serial processing system
- ④ Job is submitted to computer operator who batches them together and places them on an input device
- ④ Controlled by the monitor

✓ Monitor

- ④ Software that controls the sequence of events
- ④ Reads in a job and gives control
Job branches back to monitor when finished
- ④ Early version of operating systems



Uni-programming

- Processor must wait for I/O instruction to complete before proceeding



- Bad performance because of slow I/O operation

Read one record from file	15 µs
Execute 100 instructions	1 µs
Write one record to file	15 µs
TOTAL	31 µs

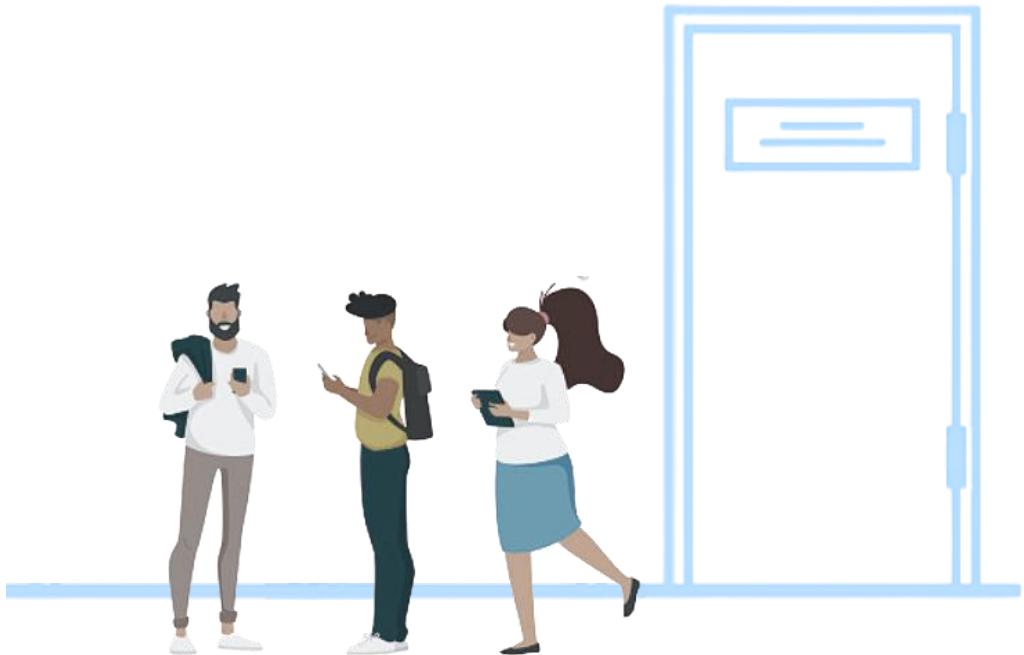
$$\text{Percent CPU Utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

System Utilization Example

Uni-programming

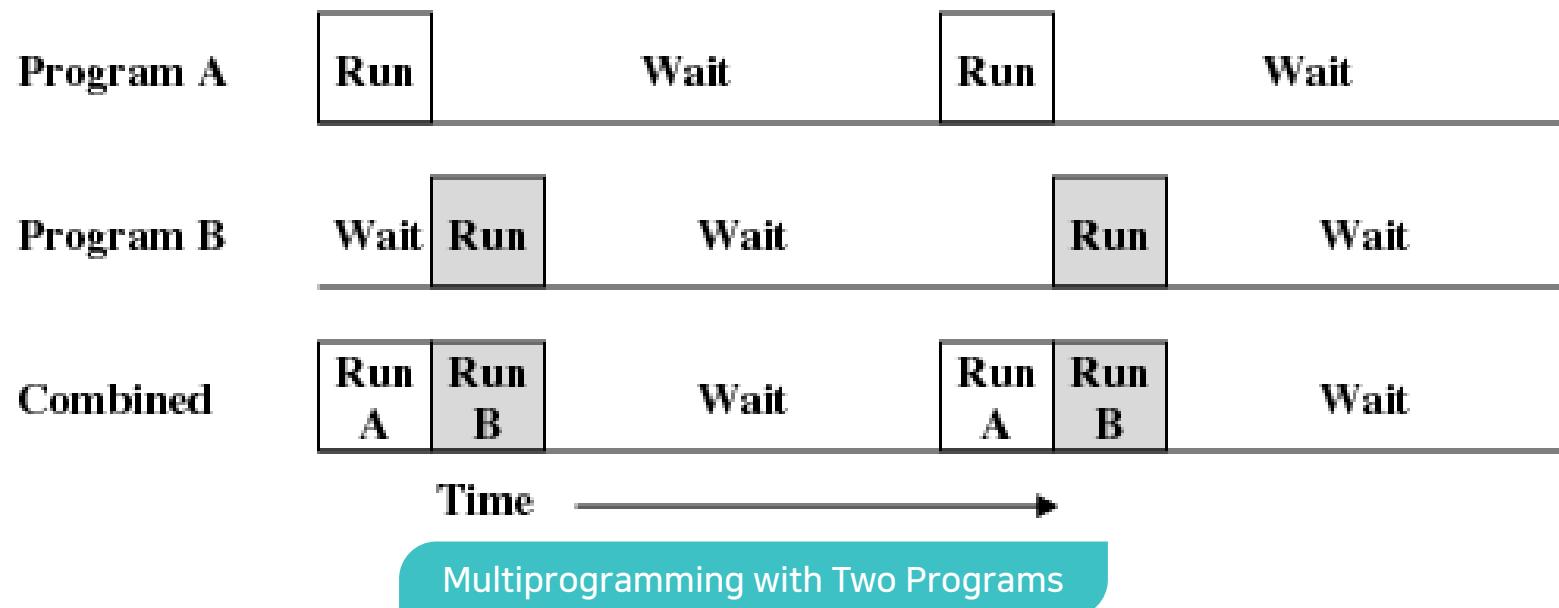


Uni-programming



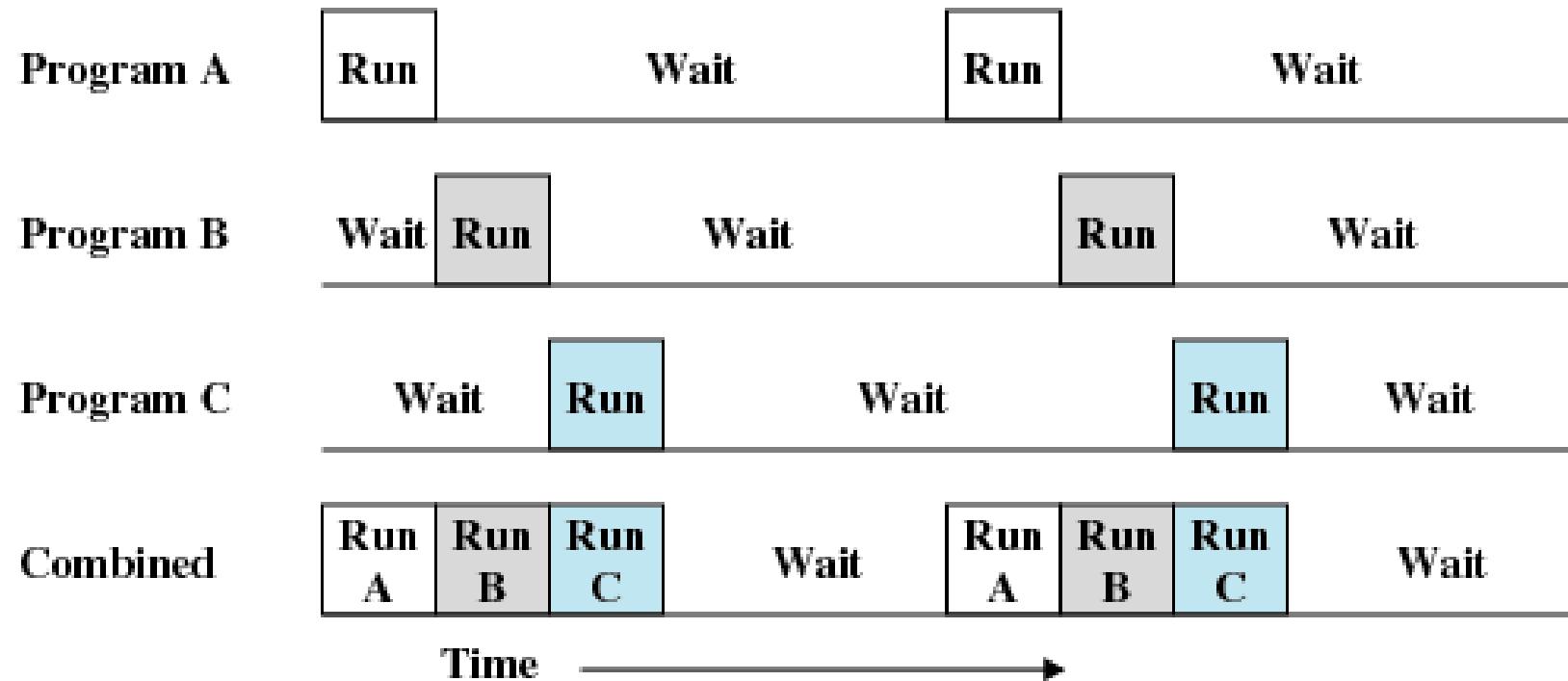
Multiprogramming Systems

- When one job need to wait for I/O, the processor can switch to the other job



- Requirement : Larger memory space

Multiprogramming



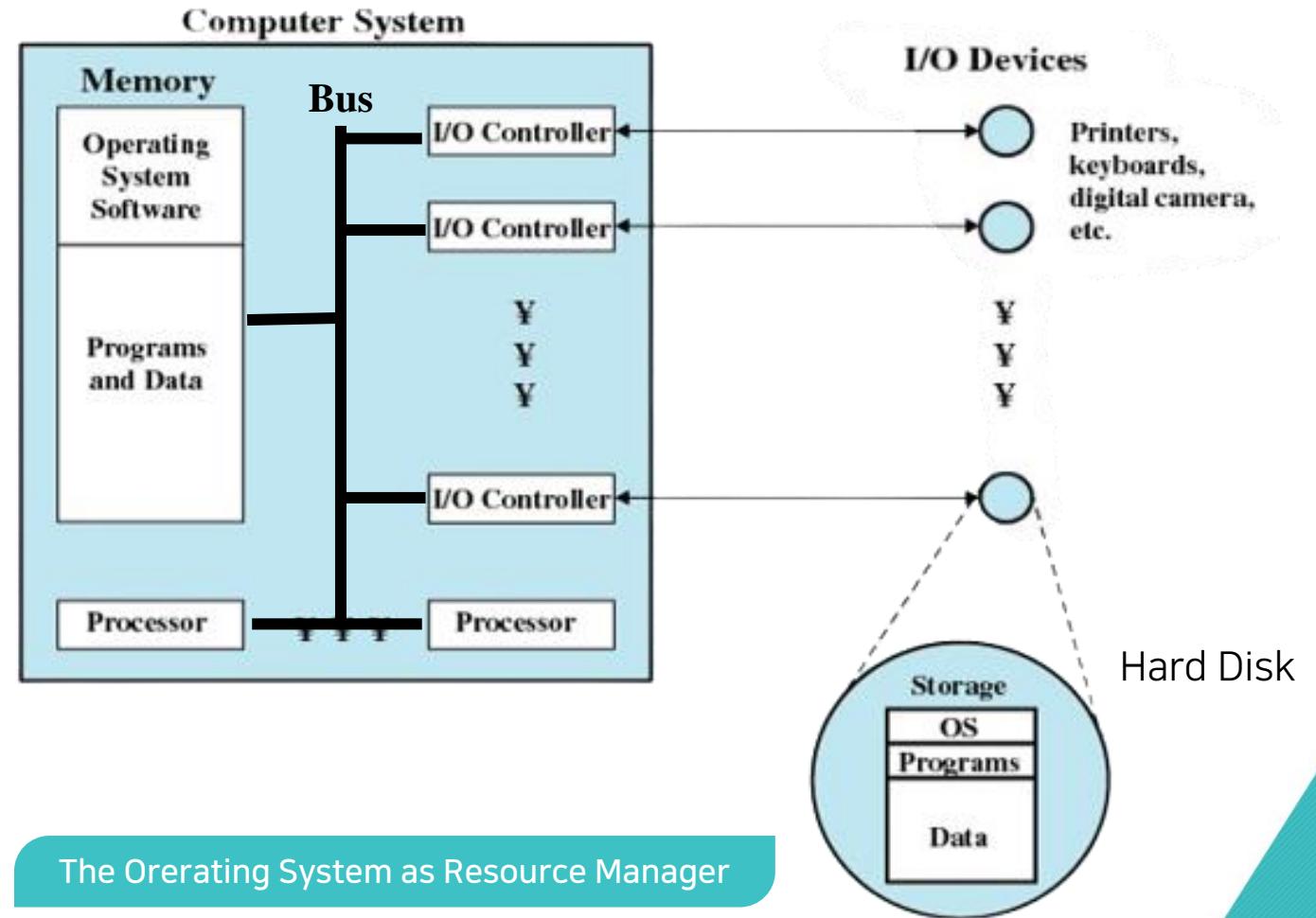
Multiprogramming with Three Programs

Recall
OS Review

Multiprogramming Systems

Kernel

- ① Portion of operating system that resides in the main memory
- ② Contains most frequently used functions



Process Scheduling

Determine which program is next to be executed, in other words,

- ⌚ Which program will take a processor next
- ⌚ Which program the CPU will execute next

Also called CPU scheduling, job scheduling

Process scheduling is performed by the kernel function, scheduler

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제3강

Time Sharing

충남대학교 인공지능학과/컴퓨터융합학부

최 훈



학습 목표

- 💬 타임쉐어링 시스템(Time Sharing System) 정의를 이해한다.
- 💬 타임쉐어링 시스템과 멀티프로그래밍(Multi-programming) 차이를 이해한다.
- 💬 멀티프로세서 시스템의 운영체제 동작 방식을 이해한다.

학습 내용

- ⌚ 타임쉐어링 시스템(Timesharing System) 정의
- ⌚ 타임쉐어링 시스템과 멀티프로그래밍
- ⌚ 멀티프로세서 시스템의 운영체제 동작 방식





Time Sharing



Time Sharing System

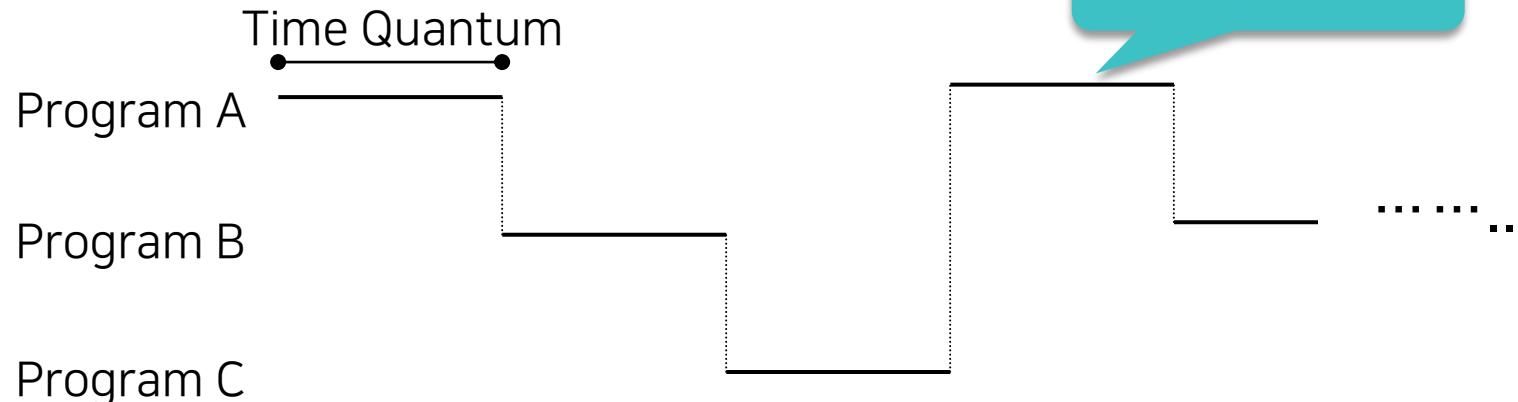
- ✓ Using multiprogramming to handle multiple interactive programs
- ✓ Processor's time is shared among multiple users

- ◎ A time unit called a *time slice* (also called *time quantum*) is given to a program at a time
- ◎ Length of a time slice = 100 msec (1/10 sec)

- ✓ Multiple users concurrently access the system through terminals

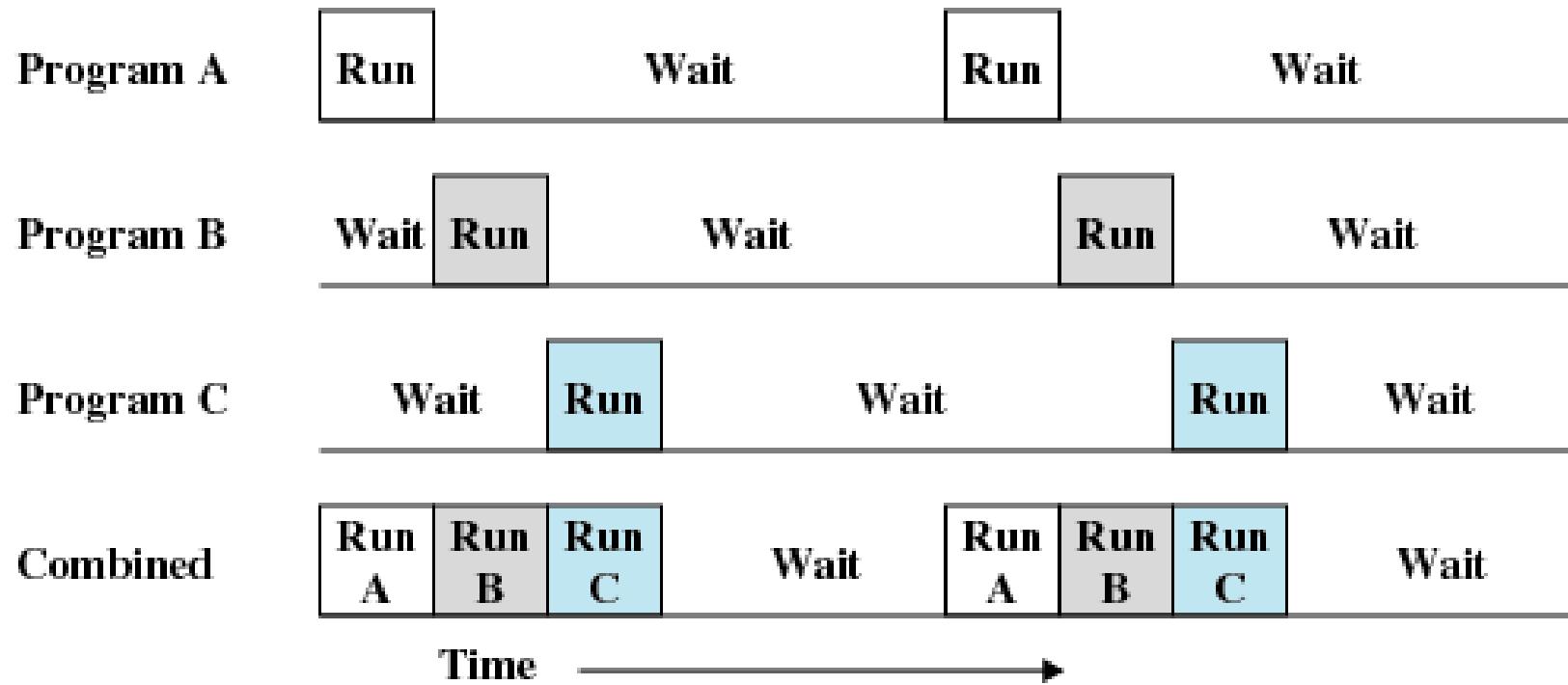
- ◎ Each user thinks he/she has his own CPU

- ✓ Reduces the mean response time



Recall
Uni/Multiprogramming

Multiprogramming

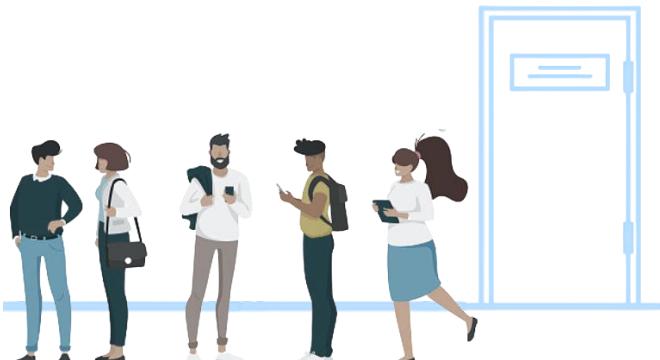


Multiprogramming with Three Programs

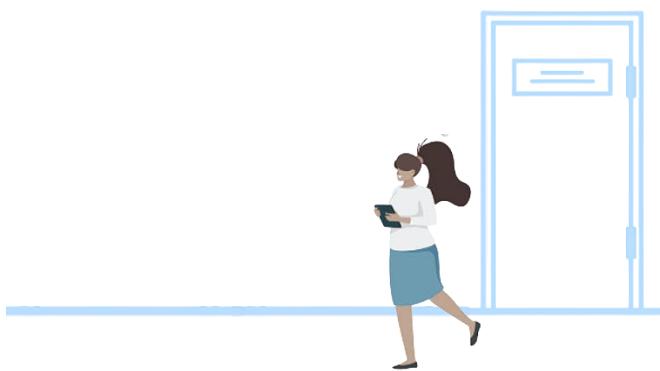
Multiprogramming vs Timesharing

Multiprogramming (a), Timesharing (b)

◎ (a)



◎ (b)



Multiprogramming vs Timesharing

✓ Both technologies are used in modern computers

✓ Multiprogramming

- ◎ Used for batch processing, background processing
- ◎ Less overhead than timesharing
- ◎ Better throughput, better use of processor
 - » Length of a time slice = 100 msec (1/10 sec)

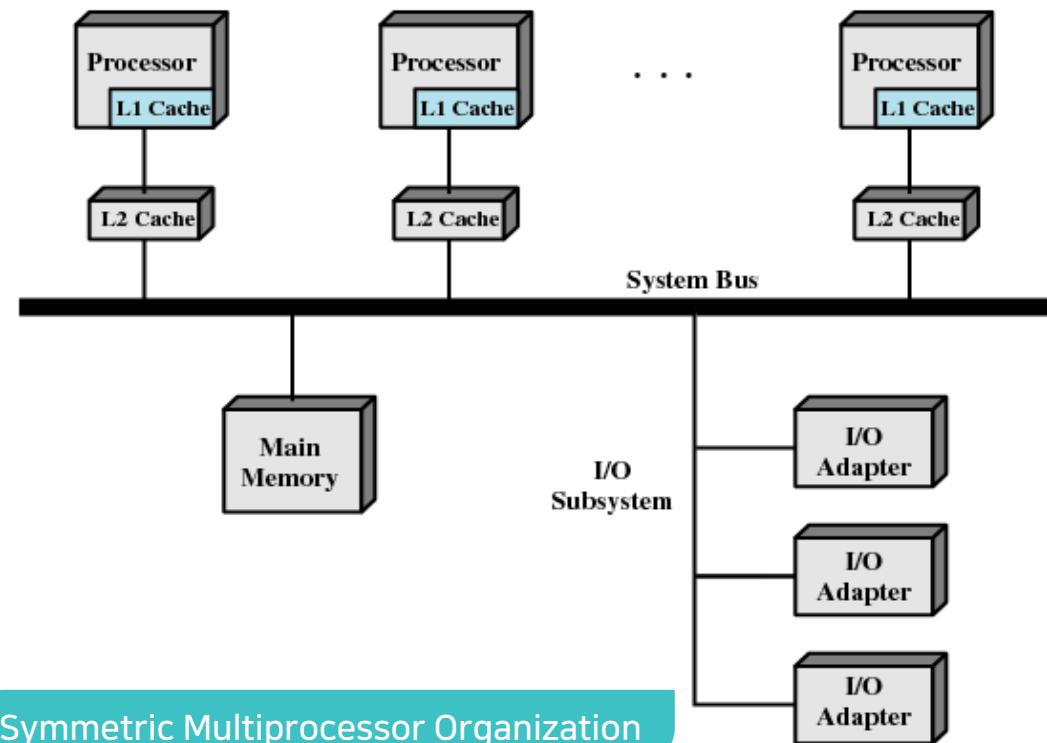
✓ Timesharing

- ◎ Used for interactive processing, foreground processing
- ◎ More overhead than multiprogramming
- ◎ Better(shorter) response time
 - » Response time : time since a user gives a command until the command is begun execution

Recall
Uni/Multiprogramming

Symmetric multiprocessing (SMP)

- ✓ There are multiple processors
- ✓ These processors share same main memory and I/O facilities
- ✓ All processors can perform the same functions

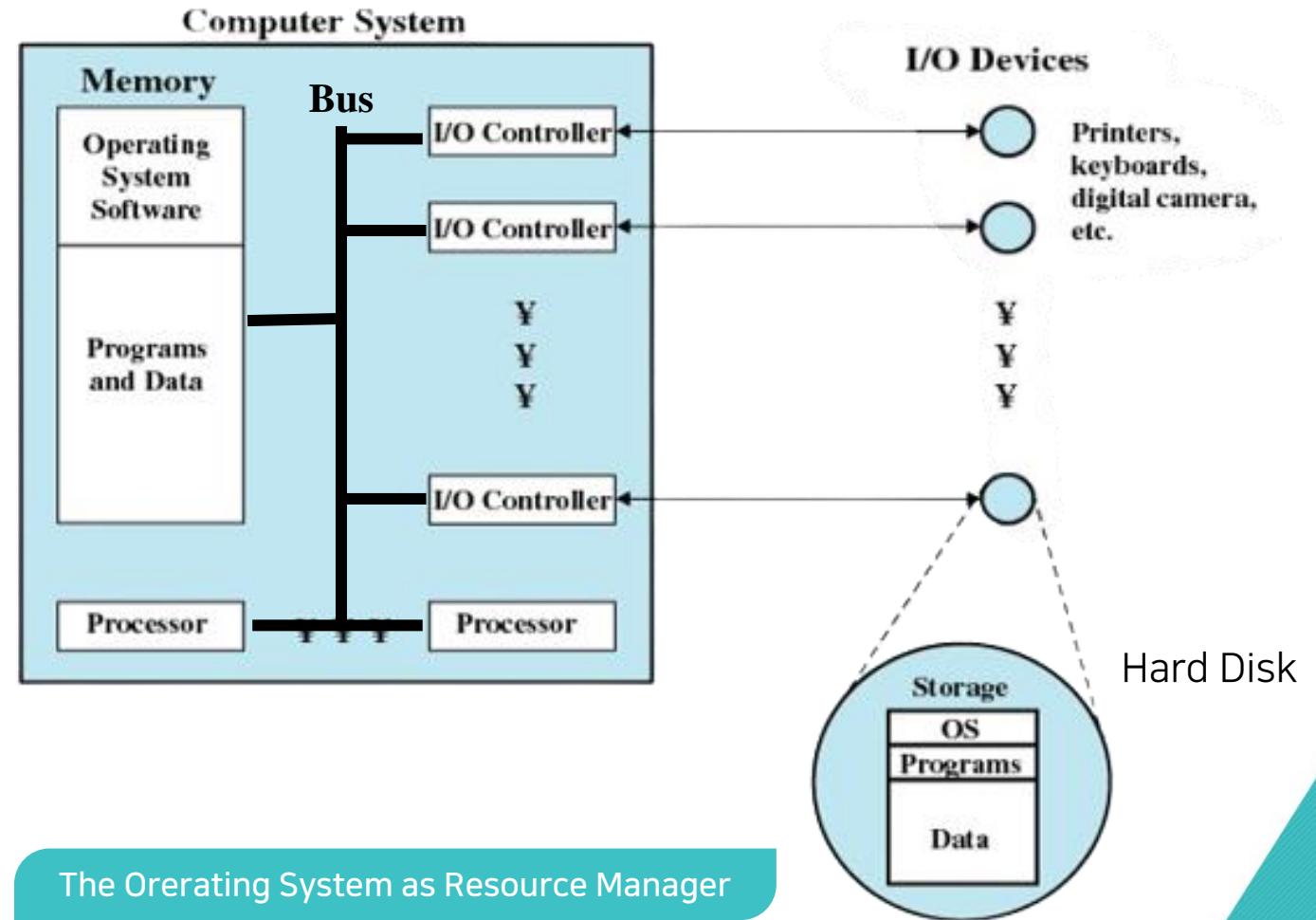


Recall
OS Review

Symmetric multiprocessing (SMP)

Kernel

- ① Portion of operating system that resides in the main memory
- ② Contains most frequently used functions



Multiprogramming vs Timesharing

Time →



a. Interleaving (Multiprogramming, one processor) Also known as Concurrent programming



b. Interleaving and overlapping (Multiprogramming, one processor) Also known as Parallel programming

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제4강

Components of OS

충남대학교 인공지능학과/컴퓨터융합학부

최 훈



학습 목표

- ❑ 운영체제의 구성 기능을 이해한다.
- ❑ 분산시스템의 정의를 이해한다.
- ❑ 운영체제 명령어 해석 방식을 이해한다.

학습 내용

- ❑ 운영체제의 구성 기능
- ❑ 분산시스템의 정의
- ❑ 운영체제 명령어 해석 방식





Components of OS



Components of OS

- Process Management (CPU management, processor management)**
- Main Memory Management**
- File Management**
- I/O System Management**
- Secondary Storage Management**
- Protection and Security Management**
- Networking**
- Command-Interpreter System**

Networking and Distributed Systems

✓ A ***distributed system*** is a collection independent computers

- ◎ Each computer has its own hardware and operating system.
- ◎ *Software for distributed processing* is equipped on top of operating system in each computer.
- ◎ The computers in the distributed system are connected through a communication network.
- ◎ Communication takes place using a *protocol*.

✓ A ***distributed system*** provides user access to various system resources

- ◎ Computation speed-up
- ◎ Increased data availability
- ◎ Enhanced reliability

Command-Interpreter System

The means by which user tells the operating system what to do

- ◎ Command line interface
- ◎ Graphical user interface (GUI) e.g. X Window

Commands are executed by means of system calls :

- ◎ Process creation and management
- ◎ I/O handling, file-system access
- ◎ Main-memory management, secondary-storage management
- ◎ Protection
- ◎ Networking

Command-Interpreter System

- ✓ The means by which **program** tells the operating system what to do

◎ System calls

- ✓ System calls are the interface between a running program and the operating system

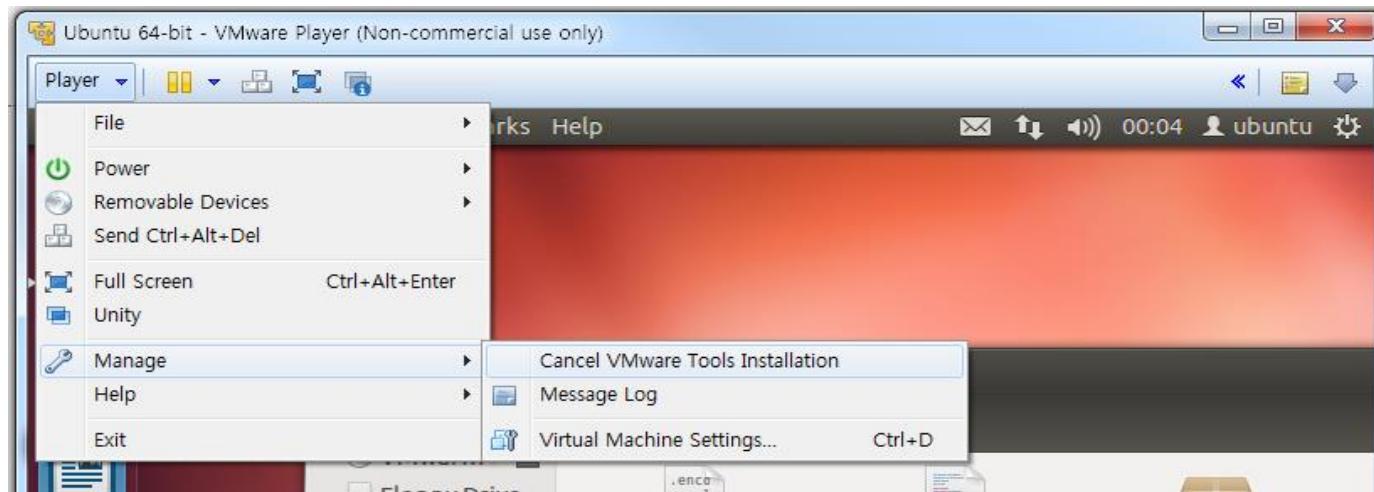
◎ Kernel functions are called by each system call

Command-Interpreter System

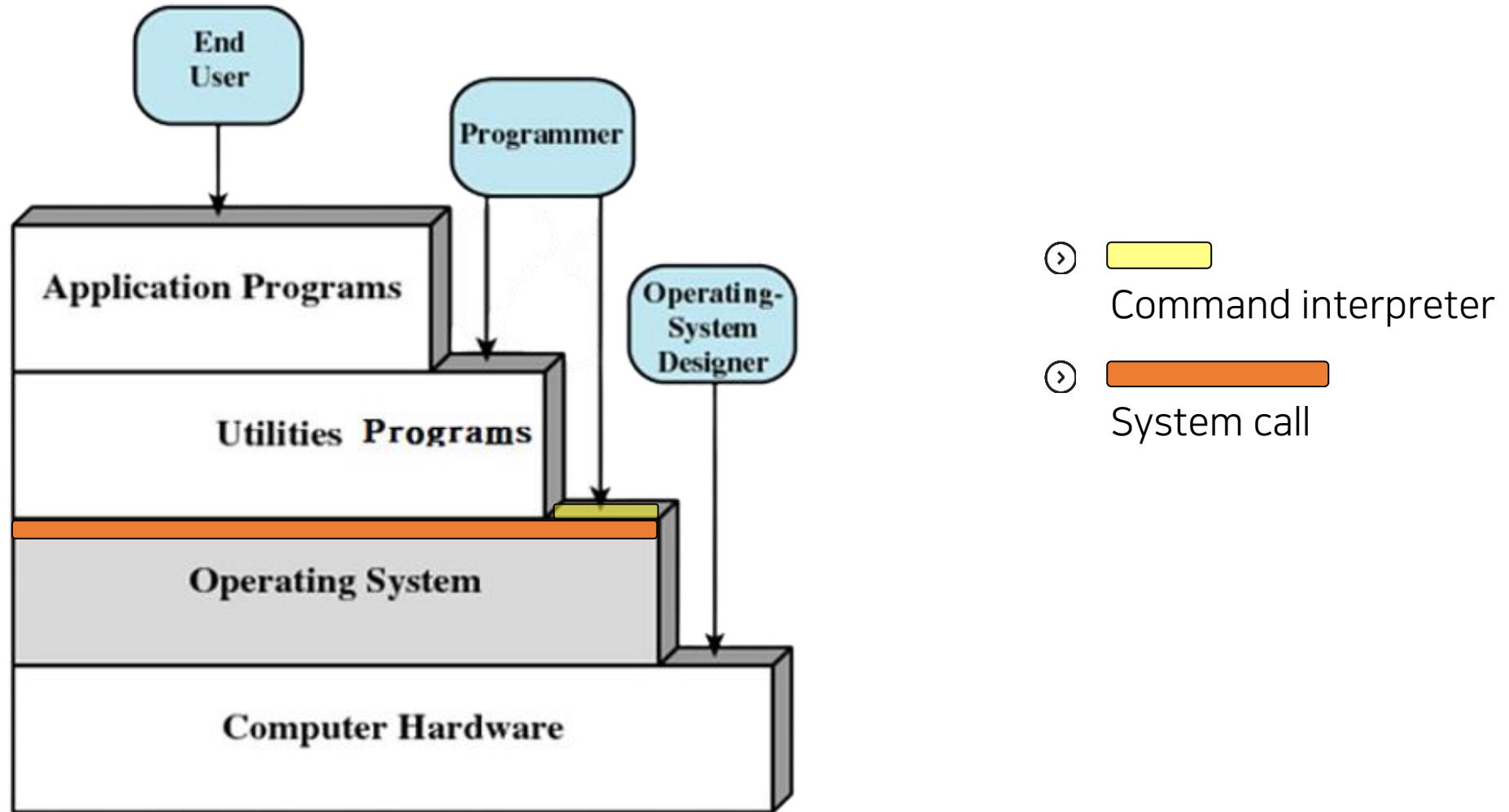
✓ Command Line Interface

```
username@ubuntu:/usr/src/linux$ mkdir proc_list  
username@ubuntu:/usr/src/linux$ cd proc_list  
username@ubuntu:/usr/src/linux/proc_list$ vi proc_list.c
```

✓ Graphical User Interface(GUI)



Command-Interpreter System



Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제5강

OS Architecture :
Monolithic Kernel, Micro Kernel

충남대학교 인공지능학과/컴퓨터융합학부

최 혼



학습 목표

- 💬 운영체제 소프트웨어 구조를 이해한다.
- 💬 모노리식 커널(Monolithic Kernel) 구조를 이해한다.
- 💬 마이크로 커널(Micro Kernel) 구조를 이해한다.
- 💬 운영체제 사례로서 UNIX, Linux, Windows 특징을 이해한다.

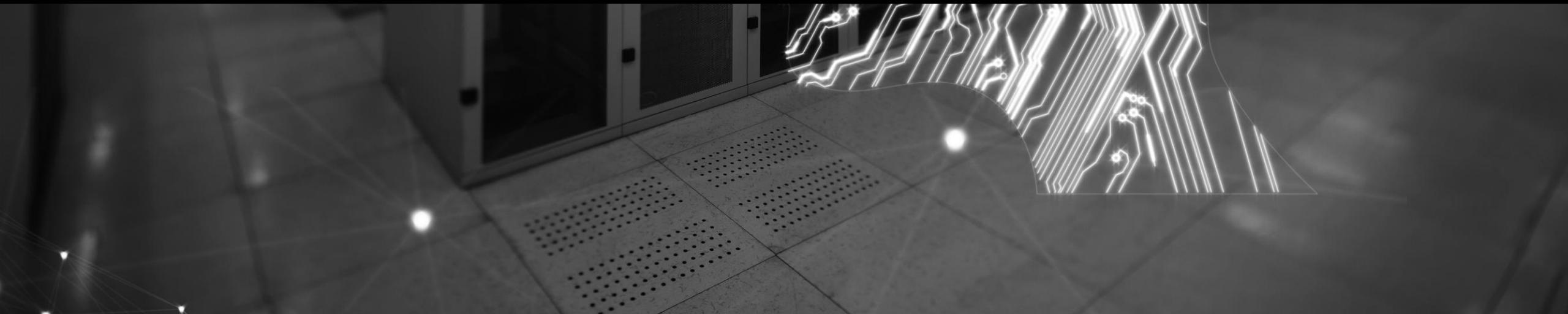
학습 내용

- ⌚ 운영체제 소프트웨어 구조
- ⌚ 모노리식 커널(Monolithic Kernel) 구조
- ⌚ 마이크로 커널(Micro Kernel) 구조
- ⌚ Unix, Linux, Windows





OS Architecture : Monolithic Kernel, Micro Kernel



Operating System Architecture

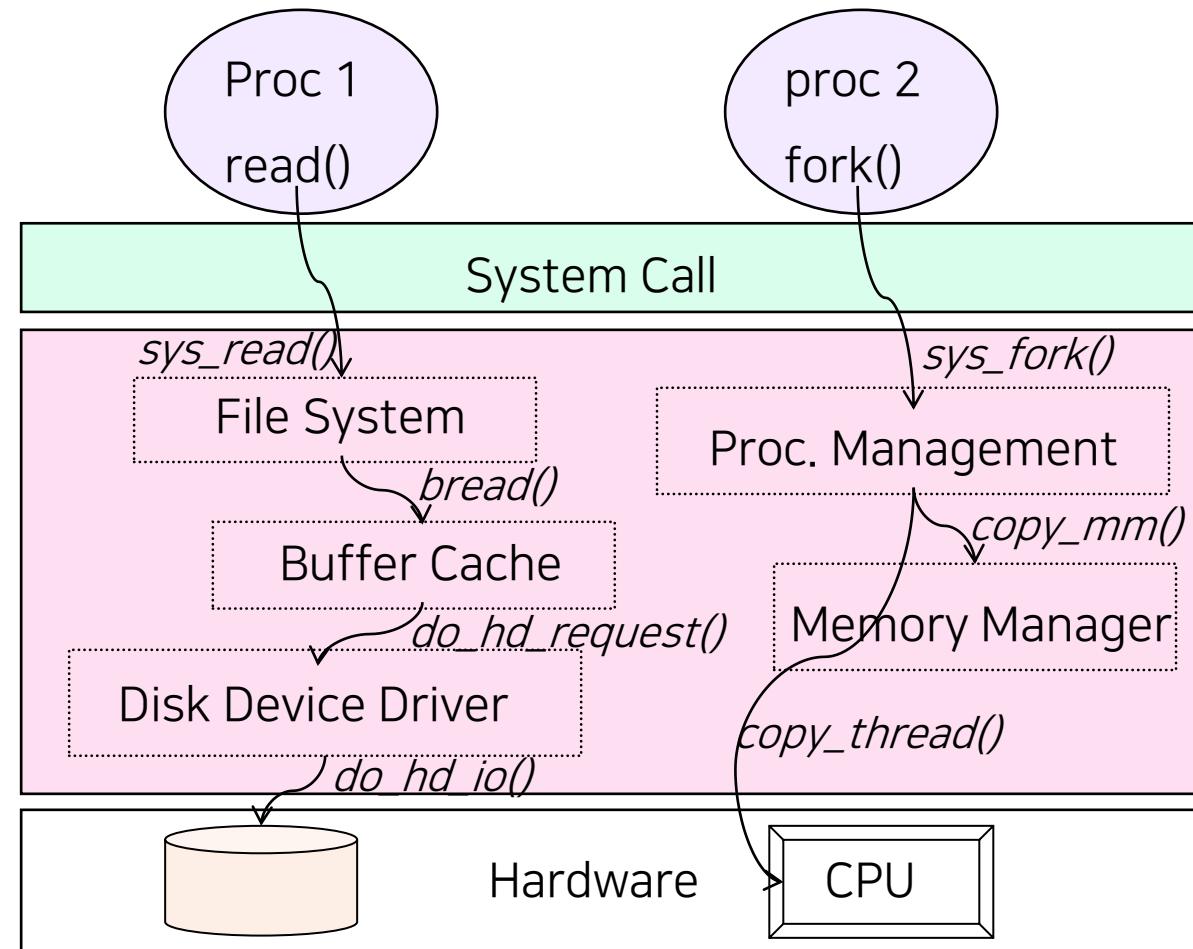
✓ 2 Architectures : Monolithic Kernel and Micro Kernel

✓ Micro Kernels

- ① Small sized kernel
- ① Contains only essential core operating systems functions
 - » Process management (IPC, synchronization, basic scheduling)
 - » Basic memory management capability
- ① Many services traditionally included in the operating system are now external subsystems (user processes)
 - » Device drivers
 - » File systems
 - » Virtual memory manager
 - » Windowing system
 - » Security services

OS Architecture : Monolithic Kernel

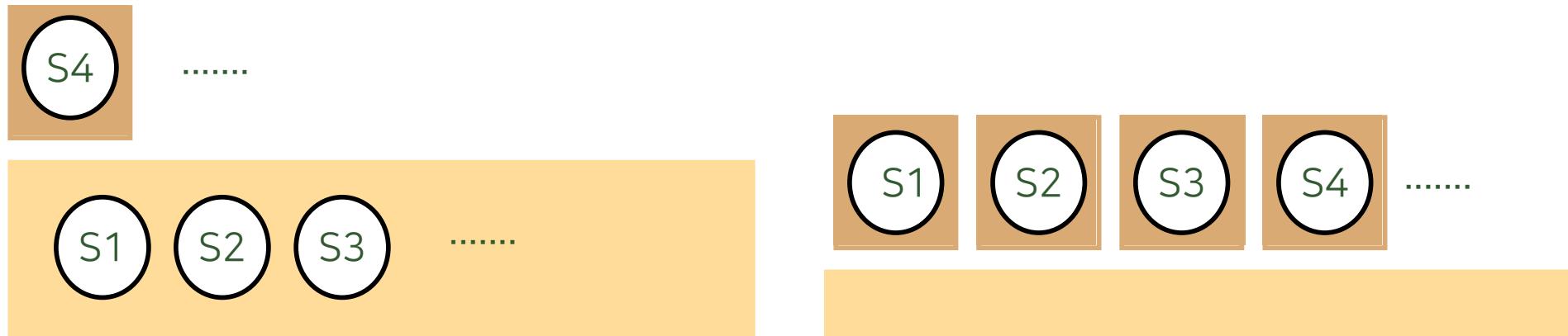
✓ UNIX (SVR4, Solaris), Linux,



OS Architecture : Microkernel

✓ **Assigns only a few essential functions to the kernel such as :**

- ① Address spaces
- ② Interprocess communication (IPC)
- ③ Basic scheduling



Key:

Server: Kernel code and data: Dynamically loaded server program:

Benefits of a Microkernel Organization

Uniform interface on request made by a process

- ◎ Do not distinguish between kernel-level and user-level services
- ◎ All services are provided by means of message passing

Extensibility

- ◎ Allows the addition of new services

Flexibility

- ◎ Features can be added or subtracted

Portability

- ◎ Changes needed to port the system to a new processor is changed in the microkernel - not in the other services

Benefits of a Microkernel Organization

Reliability

- ◎ Small microkernel can be rigorously tested

Distributed System Support

- ◎ Message are sent without knowing what the target machine is

심화

Disadvantages of Microkernel

✓ Performance : Working slower than a monolithic kernel

- ◎ Needs interprocess communication
 - » To build/send a message
 - » To Accept/decode the reply

✓ But the performance gap can be greatly reduced by the careful design of the microkernel

Example Operating Systems

UNIX

- ◎ 4.4BSD
- ◎ Solaris
- ◎ System V Release 4 (SVR4)

Linux

- ◎ Version 2.6, Version 3.3,

Windows

- ◎ Windows 98, 2000, NT, XP, CE, Vista, Windows 7

Operating systems for mobile devices

- ◎ REX, Symbian, Linux, Linux+Android, iOS, Windows Mobile,

Real-time OS

- ◎ VxWorks, RTLinux, QNX, LynxOS,

Operating systems for embedded systems

UNIX

✓ Multics → UNIX → Minix → Linux

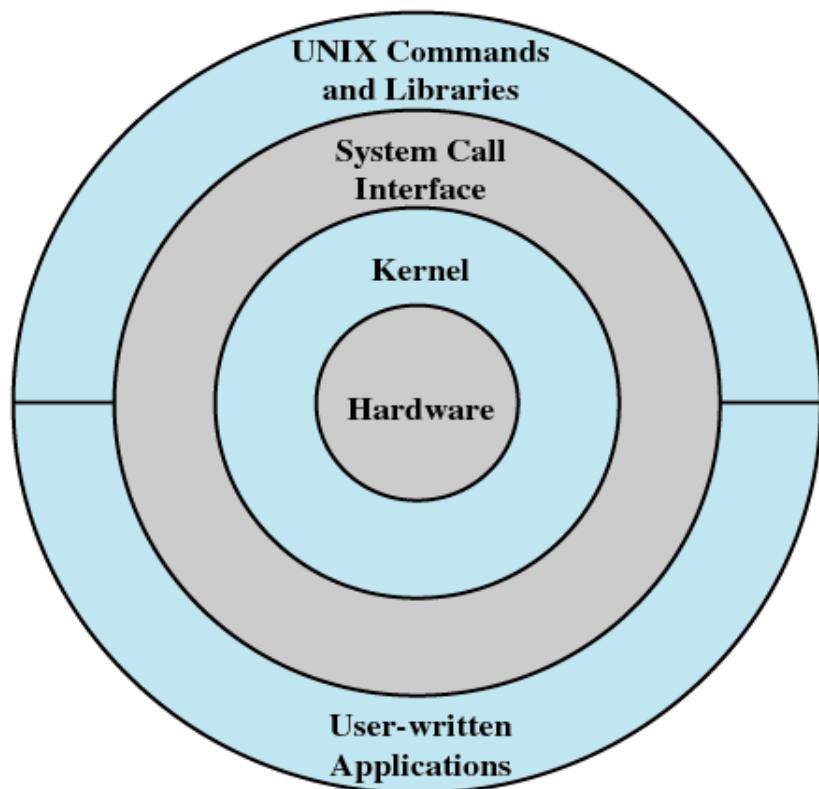
✓ UNIX Operating System

- ④ Released in 1971 by Bell Lab.(Ken Thompson, Dennis Ritchie etc.)
- ④ Rewritten in C language
- ④ First widely available version outside Bell Labs was Version 6 in 1976
- ④ Many versions by many companies, research institutes
 - » BSD(Berkeley Software Distribution) by Univ. of California at Berkeley
 - » System III, System IV(four), System V(five) Release 4 (SVR4) by AT&T
 - » Solaris : Sun's SVR4-based UNIX release
 - Most widely used and most successful commercial UNIX implementation

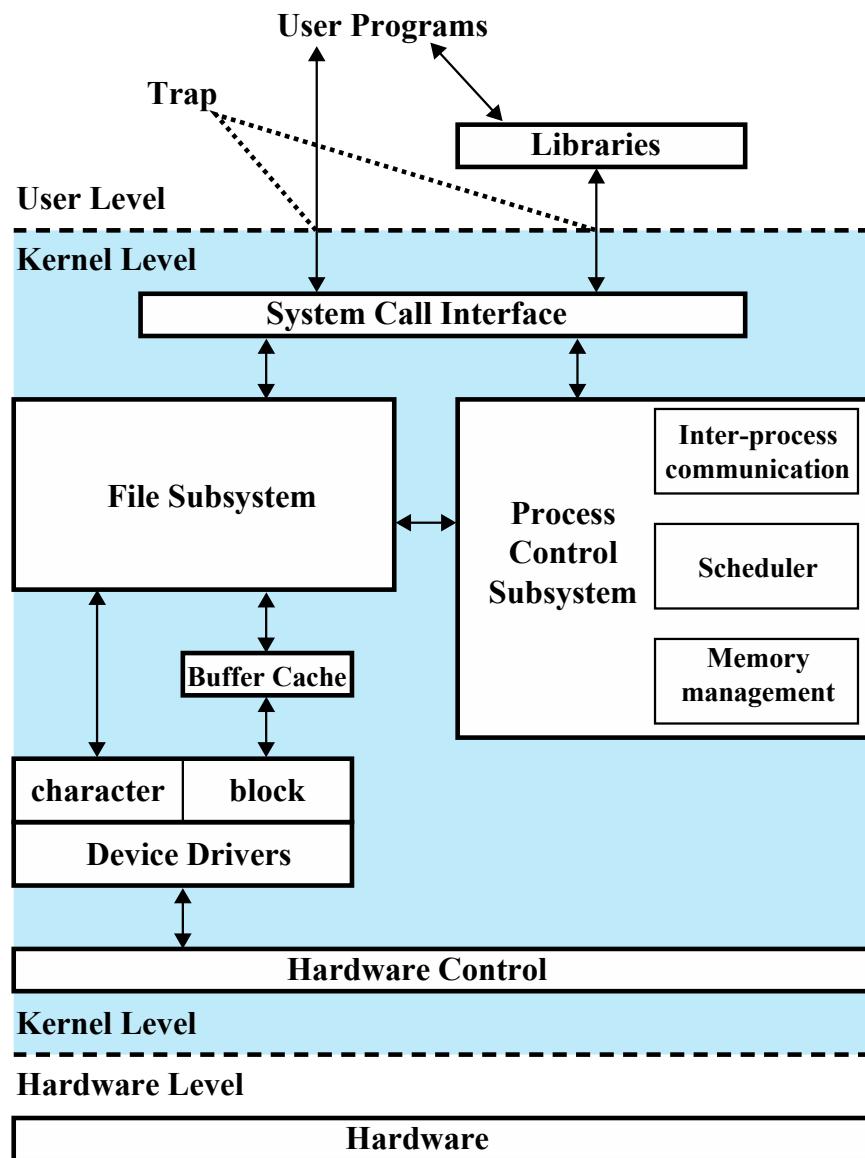
✓ Popular OS for main frame, server computers

UNIX

✓ Traditional UNIX Kernel



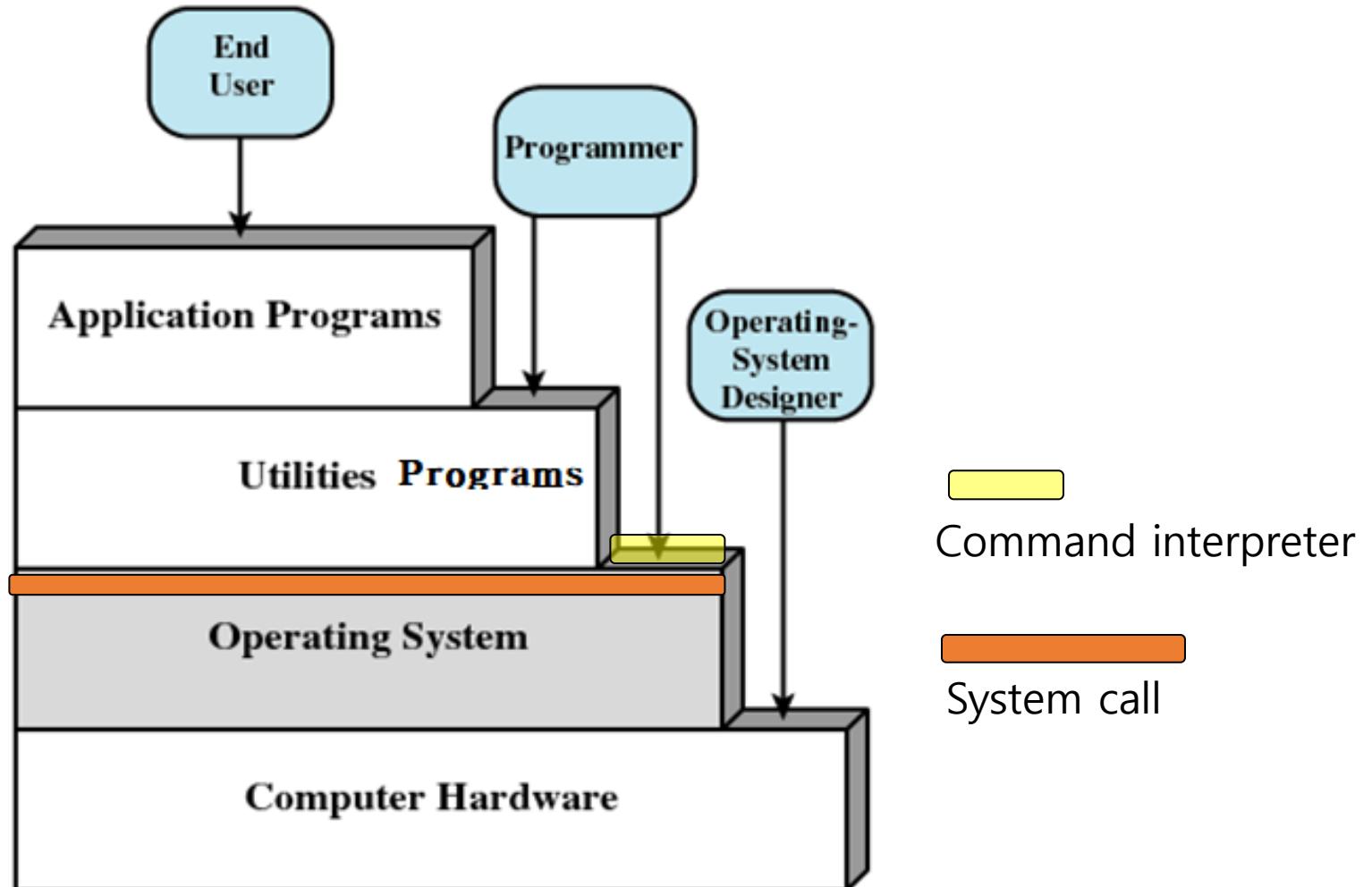
General UNIX Architecture



Traditional UNIX Kernel

Recall
Components of OS

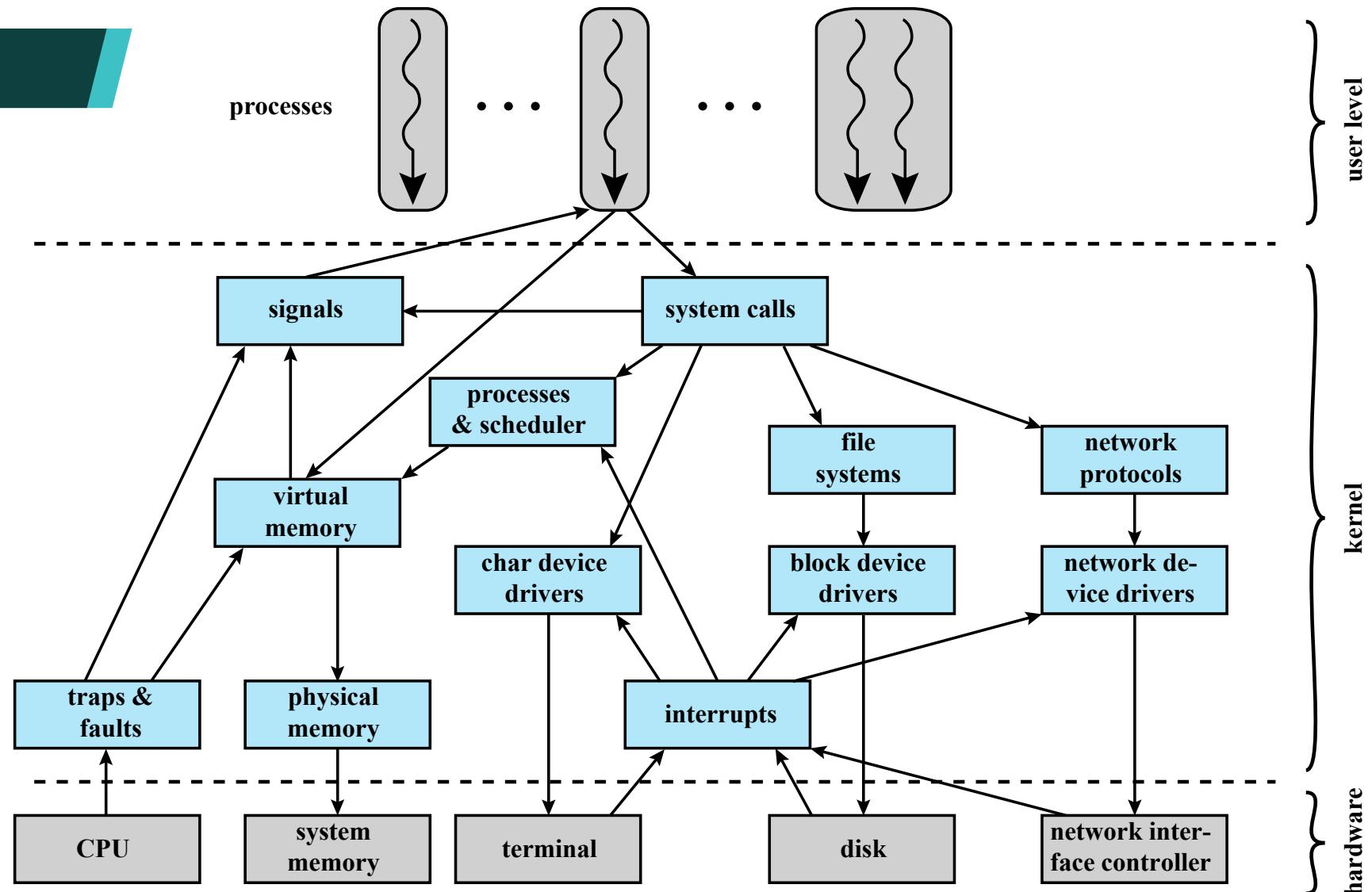
Command-Interpreter System



Linux

- ✓ Started out as a UNIX variant for the IBM PC
- ✓ Linus Torvalds(Finnish student of computer science) wrote the initial version
- ✓ Linux was first posted on the Internet in 1991
- ✓ Today it is a full-featured UNIX system that runs on several platforms
- ✓ Free and the source code is available
 - ◎ The Linux Kernel Archives: <https://www.kernel.org/>
- ✓ Key to success has been the availability of free software packages
- ✓ Highly modular and easily configured

Linux



Linux Kernel Components

Windows

✓ **Consists of the micro-kernel and User-mode processes**

✓ **User-mode Processes**

- ⌚ System support processes
 - » Needed to manage the system, such as the session manager, the authentication subsystem, the service manager, and the logon process.

✓ **Service Processes**

- ⌚ The printer spooler, the event logger, user-mode components that cooperate with device drivers, various network services
- ⌚ Used by both Microsoft and external software developers to extend system functionality (run background user-mode activity on Windows system).

Windows

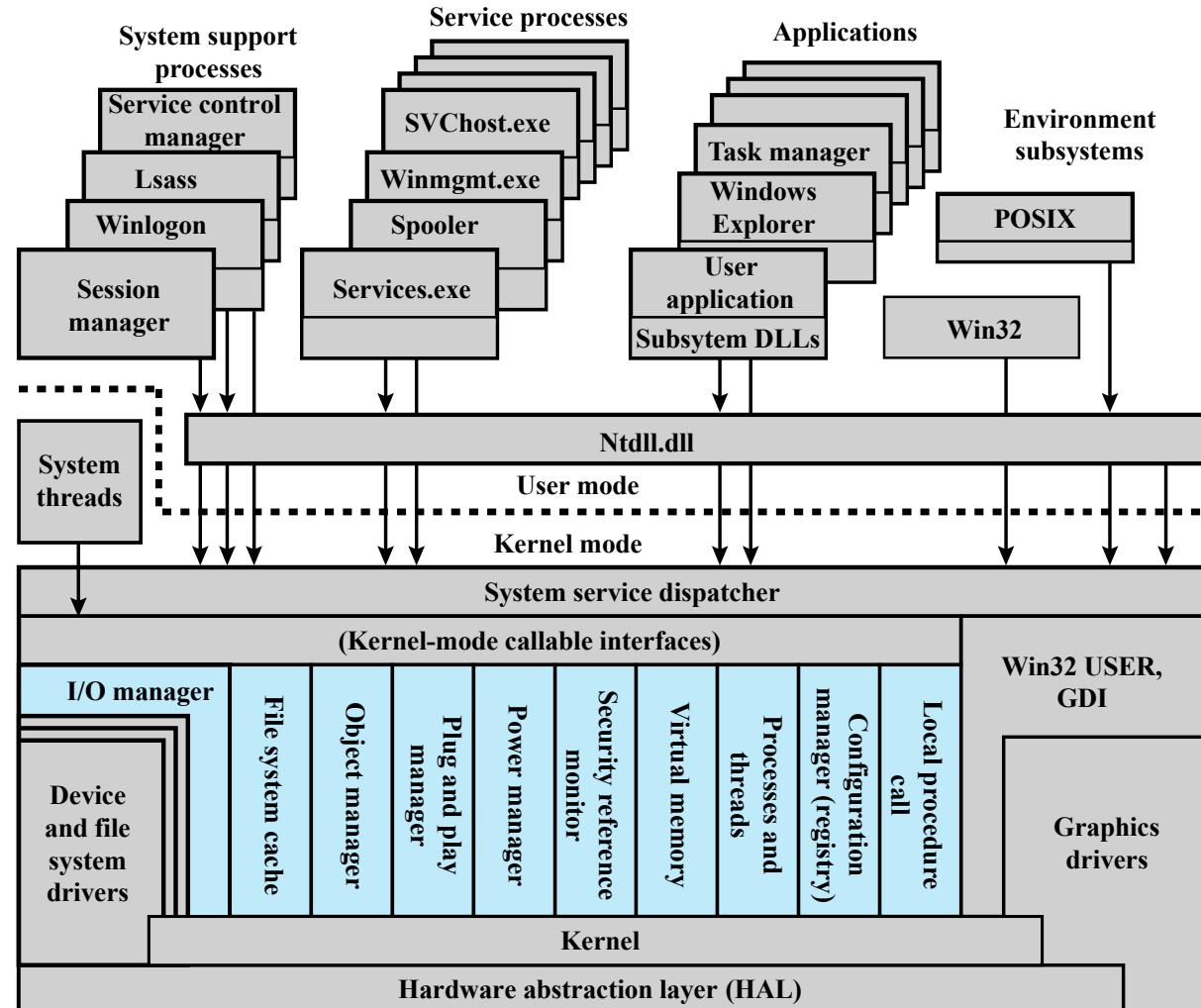
Applications

- ◎ Executables (EXEs) and DLLs that provide the functionality users run to make use of the system

Environment subsystems

- ◎ Provide different OS environments (e.g.: Win32 and POSIX)

Windows



Lsass = local security authentication server
 POSIX = portable operating system interface
 GDI = graphics device interface
 DLL = dynamic link libraries

Colored area indicates Executive

Windows Architecture

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제6강

Definition of a Process
and Its States

충남대학교 인공지능학과/컴퓨터융합학부

최 혼



학습 목표

- ❑ 프로세스 정의를 이해한다.
- ❑ 프로세스 구성 요소를 이해한다.
- ❑ 프로세스 상태를 이해한다.

학습 내용

- ❑ 프로세스 정의
- ❑ 프로세스 구성 요소
- ❑ 프로세스 상태





Definition of a Process and Its States



What is a Process?

✓ Definition

- ◎ A program in execution
- ◎ An instance of a program running on a computer
- ◎ The entity that can be assigned to and executed on a processor

✓ A unit of activity characterized by

- ◎ The execution of a sequence of instructions,
- ◎ A current state, and
- ◎ An associated set of system instructions

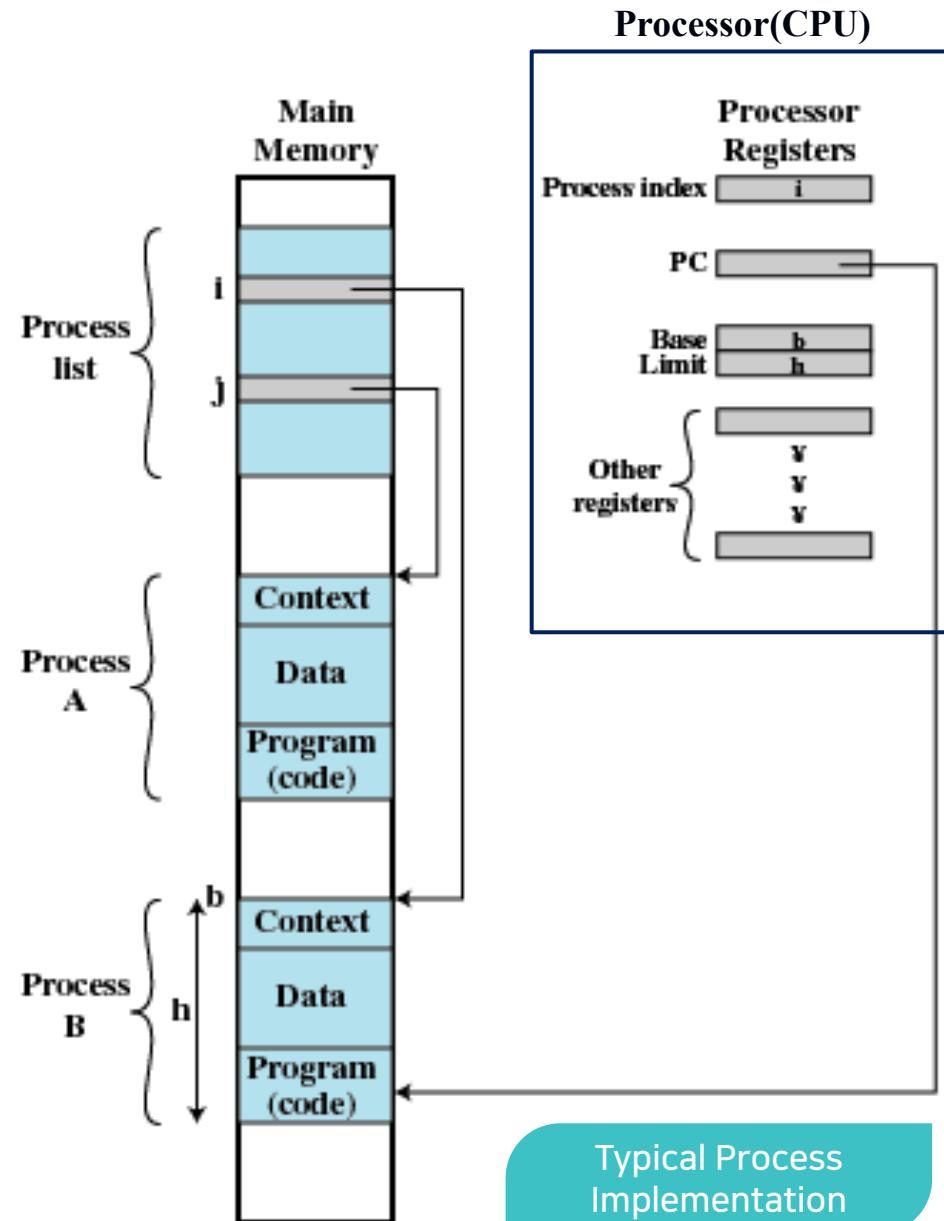
✓ Who creates a process?

- ◎ By a human : Running a program
 - » Graphical User Interface (GUI) or command
- ◎ By a process
 - » Operating system process
 - » User process

Process

✓ Consists of 3 Components

- ① An executable program (code, or text)
- ② Data used in the program
- ③ Execution context of the program
 - » Stack and the values in the registers of the CPU
 - » Information the operating system needs to manage for the process

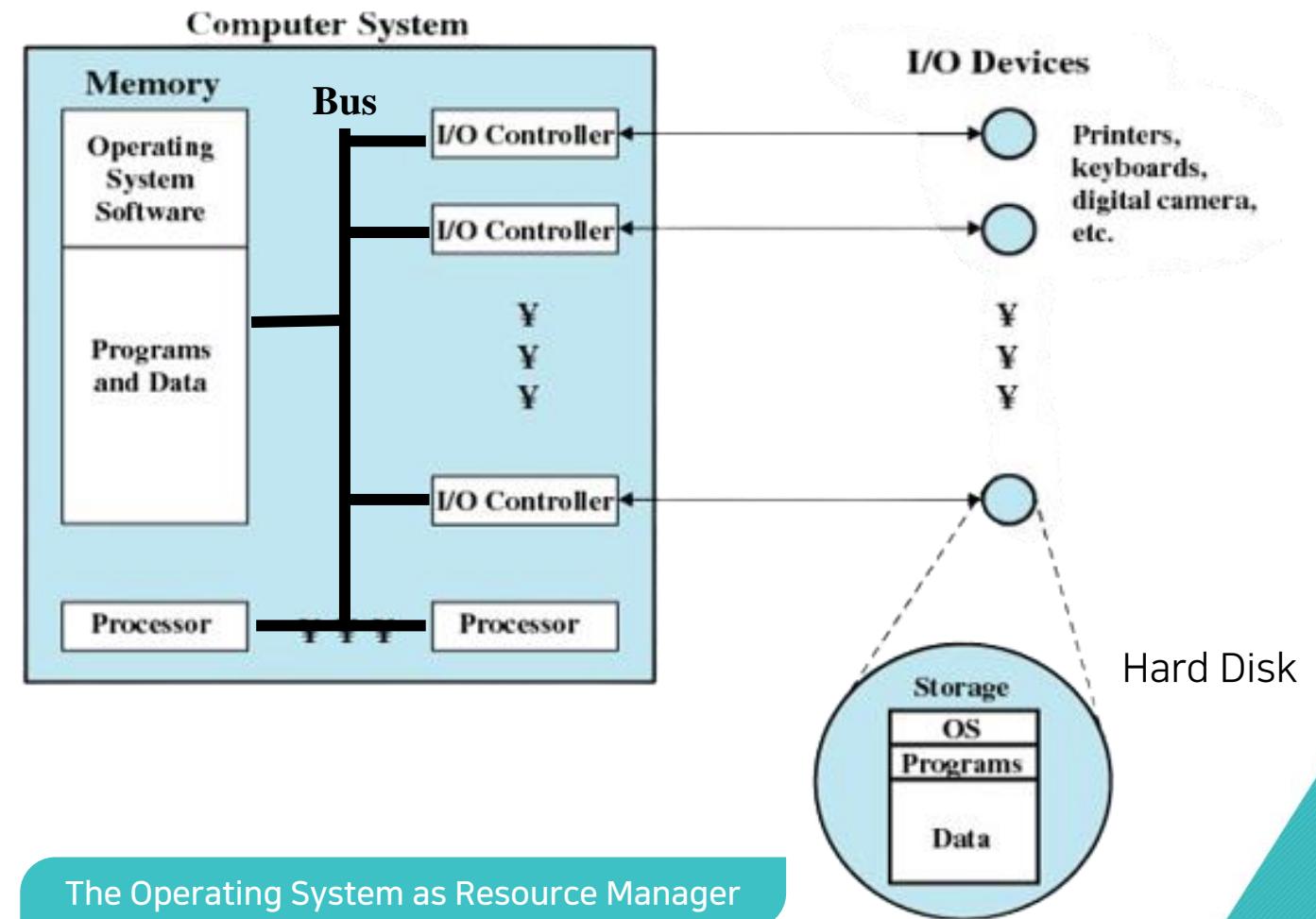


Recall
OS Review

Process

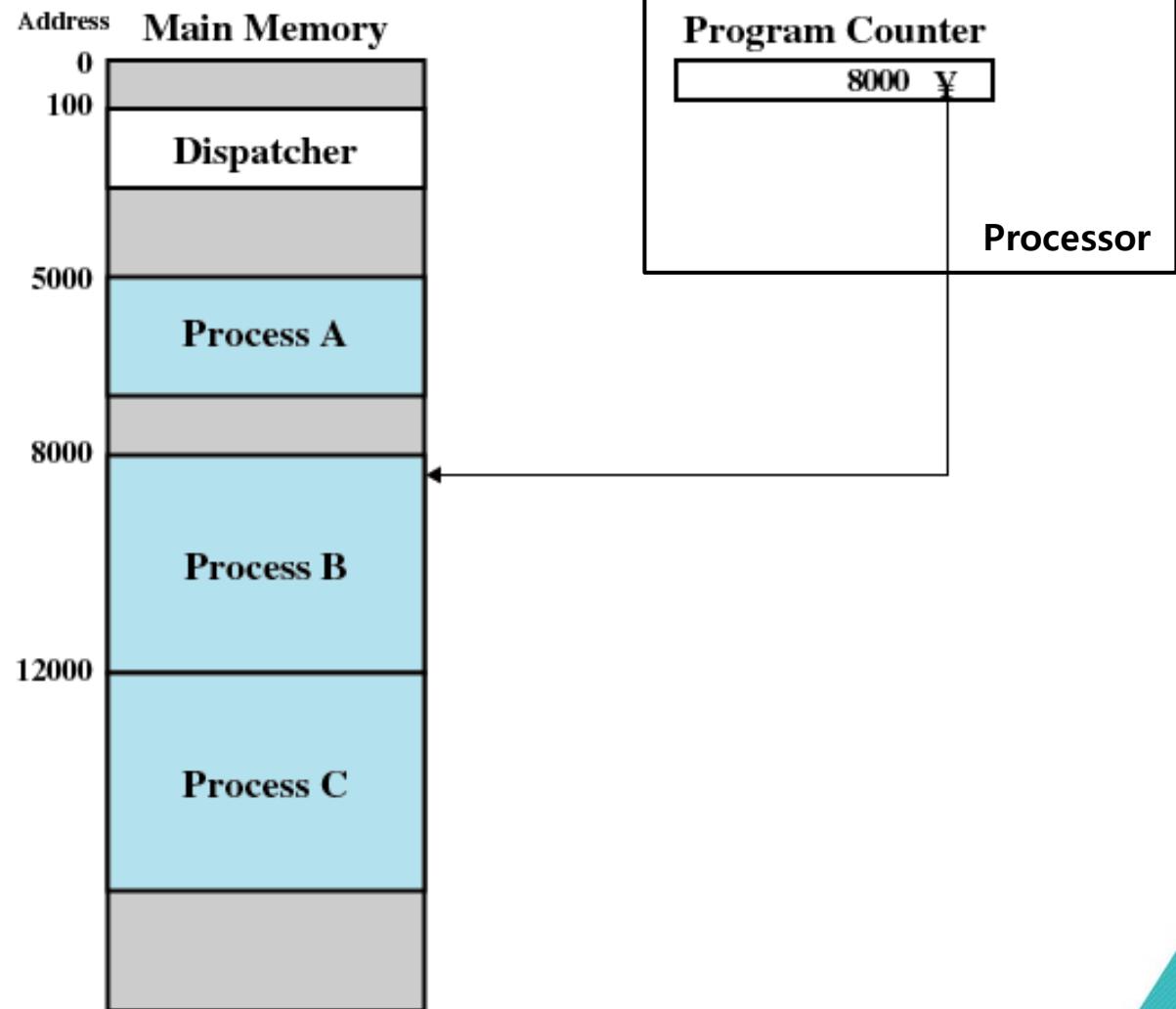
Kernel

- ① Portion of operating system that resides in the main memory
- ② Contains most frequently used functions



Process States

- ✓ Trace of Process : Sequence of instruction that execute for a process
- ✓ Dispatcher (a kernel function) switches the processor from one process to another



Trace of Processes

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A (b) Trace of Process B (c) Trace of Process C

5000=Starting address of program of Process A
8000=Starting address of Program of Process B
12000=Starting address of Program of Process C

Traces of Processes

Trace of Processes

1	5000	27	12004
2	5001	28	12005
3	5002	----- Time out	
4	5003	29	100
5	5004	30	101
6	5005	31	102
----- Time out (interrupt)		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	----- Time out	
16	8003	41	100
----- I/O request (system call)		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
----- Time out			

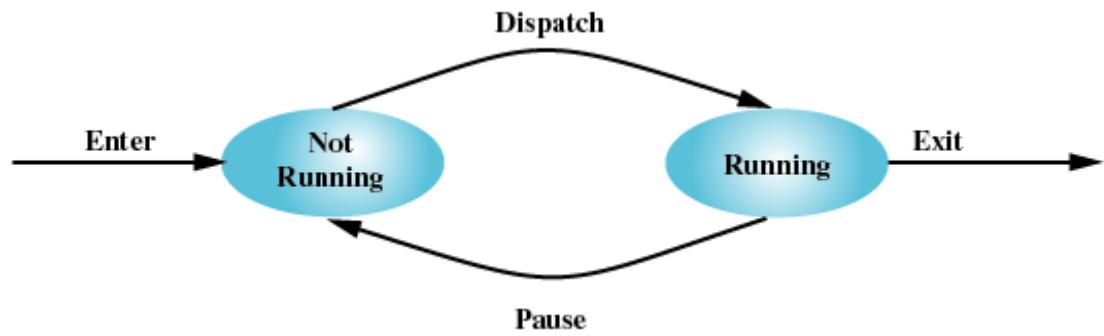
100=Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
First and third columns count instruction cycles;
Second and fourth columns show address of
instruction being executed

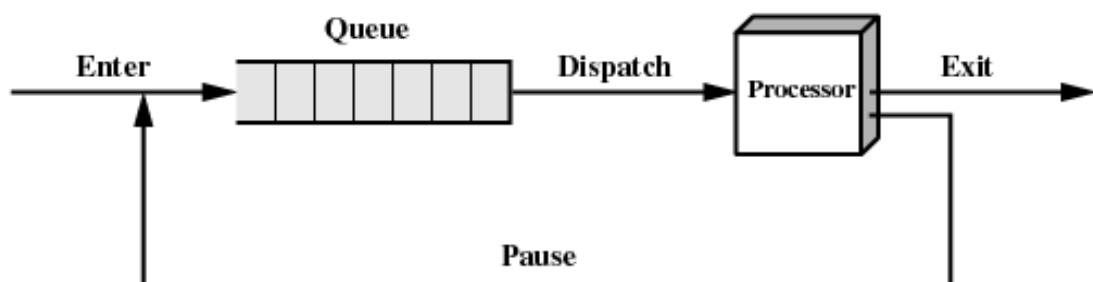
Two-State Process Model

✓ **Process may be in one of two states**

◎ Running or Not-running



State Transition Diagram



Queuing Diagram

Reasons for Process Creation

✓ **Created by OS to provide a service**

- ◎ The operating system can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).

✓ **Created by an existing process**

- ◎ A user program can request OS to create children processes.
- ◎ A user at a terminal logs on to the system.

✓ **Created by user command**

- ◎ Running a program in the foreground (interactive) mode
- ◎ Running a program in the background (batch) mode
 - » In the batch mode, the OS delays the start of execution of programs

Reasons for Process Termination

Normal Completion

- ◎ The process executes an OS system call to inform that it has completed running.

Protection Error

- ◎ The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
- ◎ The process tries to access a memory location that it is not allowed to access.

Parent Request

- ◎ The parent process can request the OS to terminate the children processes.

Arithmetic Error, I/O Failure, Invalid Instruction, etc.

- ◎ The process tries a prohibited computation, such as division by zero

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제7강

Process States Model

충남대학교 인공지능학과/컴퓨터융합학부

최 훈



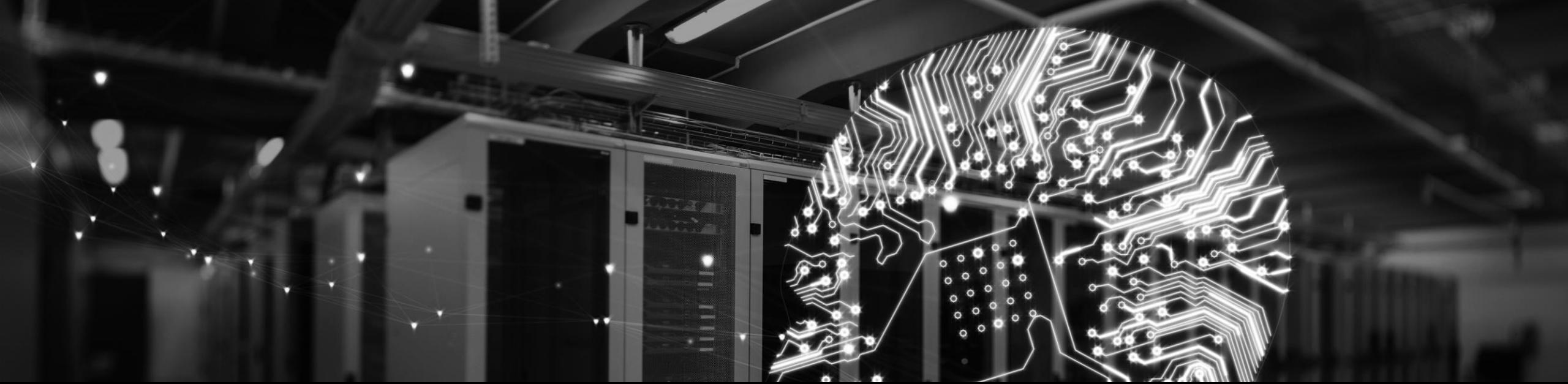
학습 목표

- ❑ 프로세스 상태 모델을 이해한다.
- ❑ 서스펜디드 프로세스(Suspended Process) 정의를 이해한다.
- ❑ 서스펜디드 프로세스를 고려한 프로세스 상태 모델을 이해한다.

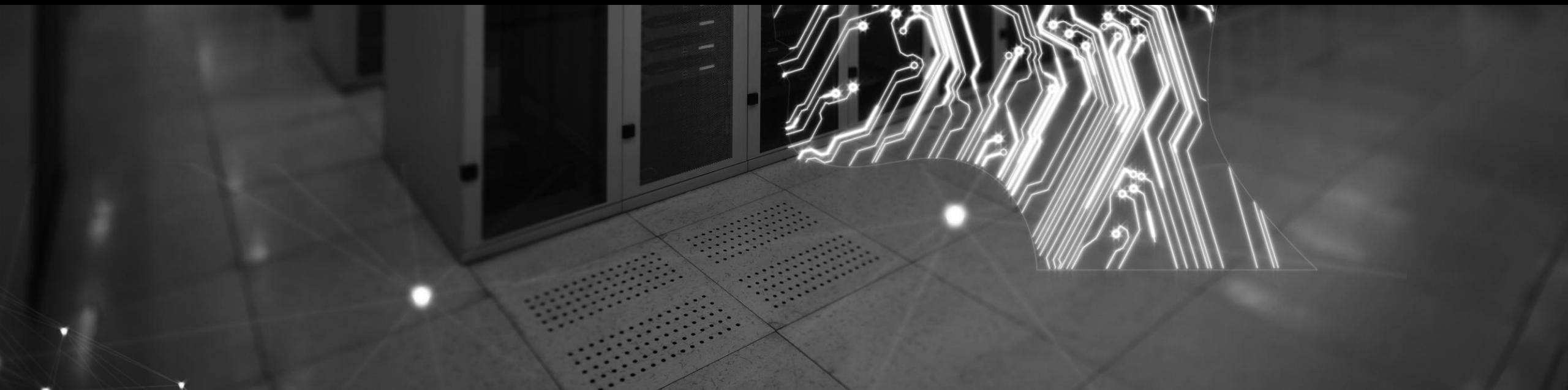
학습 내용

- ❑ 프로세스 상태 모델
- ❑ 서스펜디드 프로세스(Suspended Process) 정의
- ❑ 서스펜디드 프로세스를 고려한 프로세스 상태 모델





Process States Model

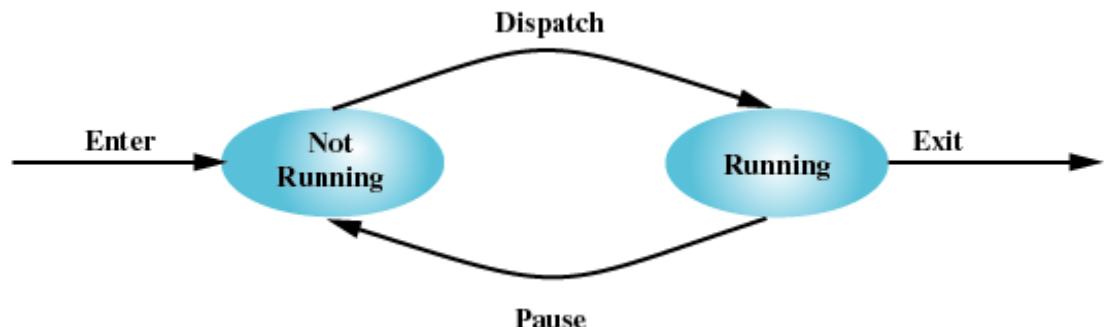


Recall
What is a Process?

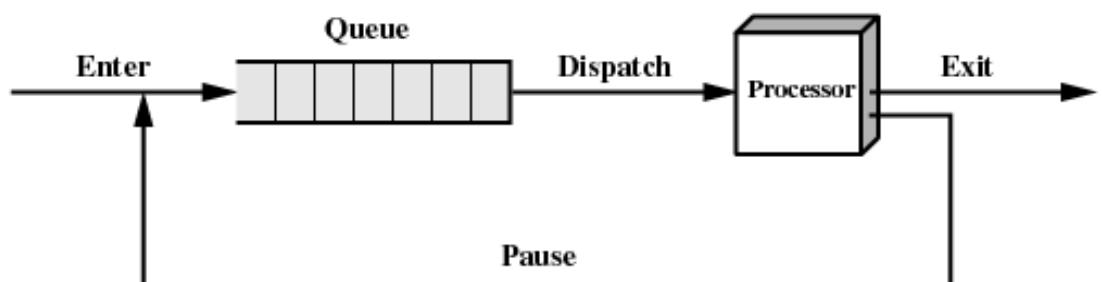
Two-State Process Model

- Process may be in one of two states

① Running or Not-running



State Transition Diagram



Queuing Diagram

심화

Not-running Processes

Ready State

- ◎ Ready to execute
- ◎ Waiting for the processor to dispatch

Blocked State (= Wait state, Sleep State)

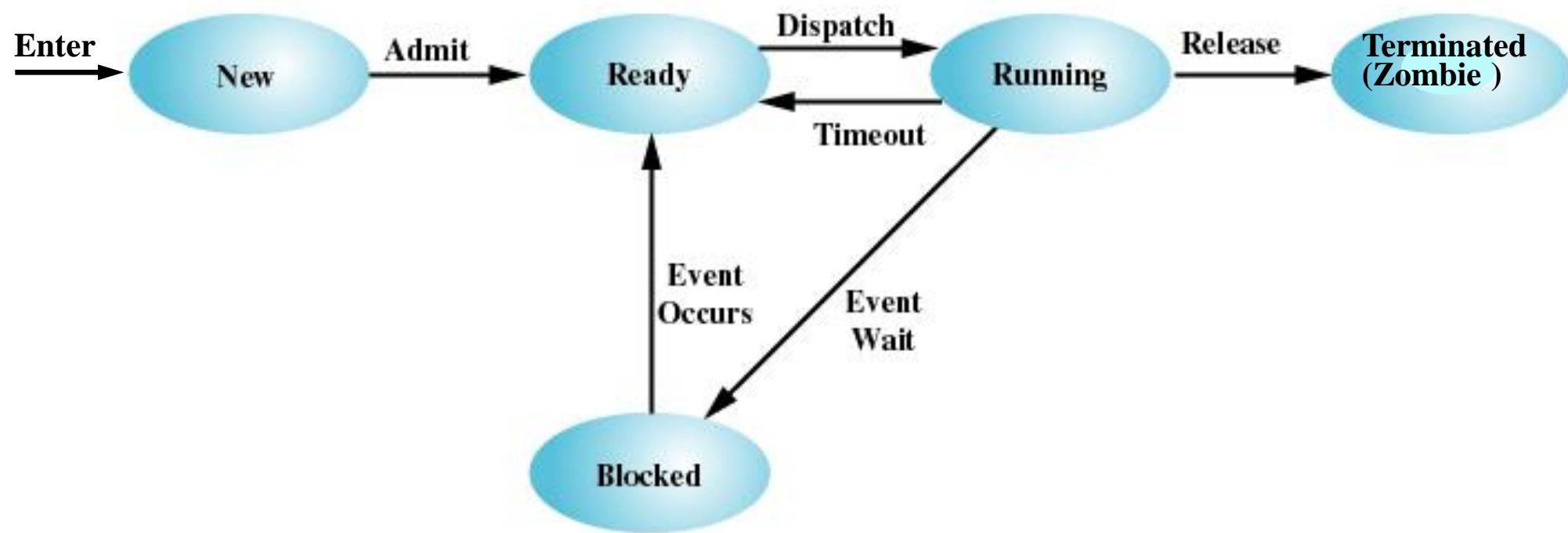
- ◎ Waiting for some event to occur (e.g. I/O completion)

The process that has been in the queue the longest is not always selected by the dispatcher

- ◎ Because it may be blocked

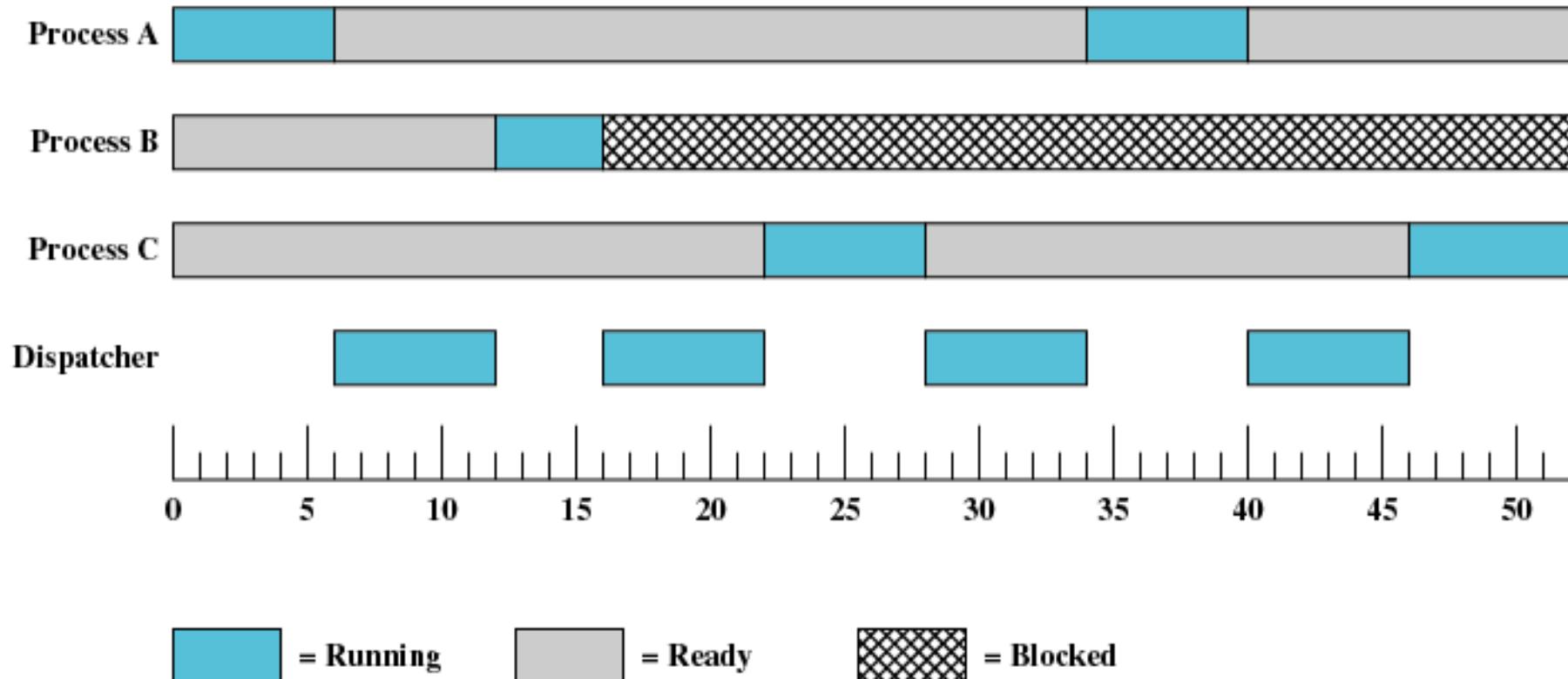
Recall
What is a Process?

Five-State Process Model



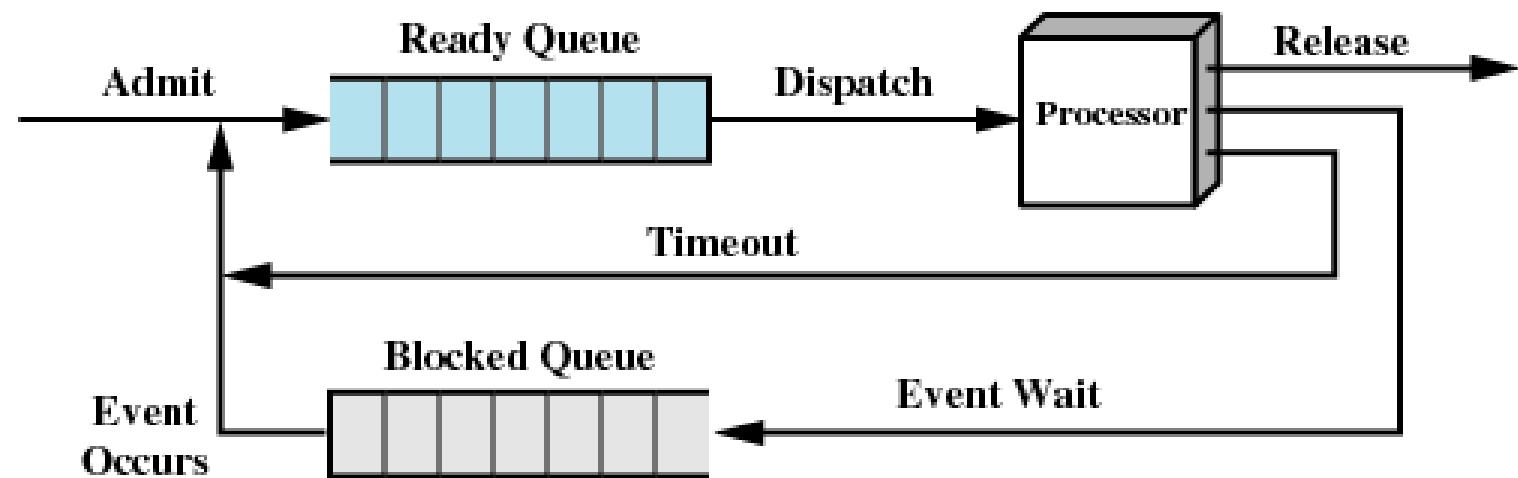
Five-State Process Model

Process States



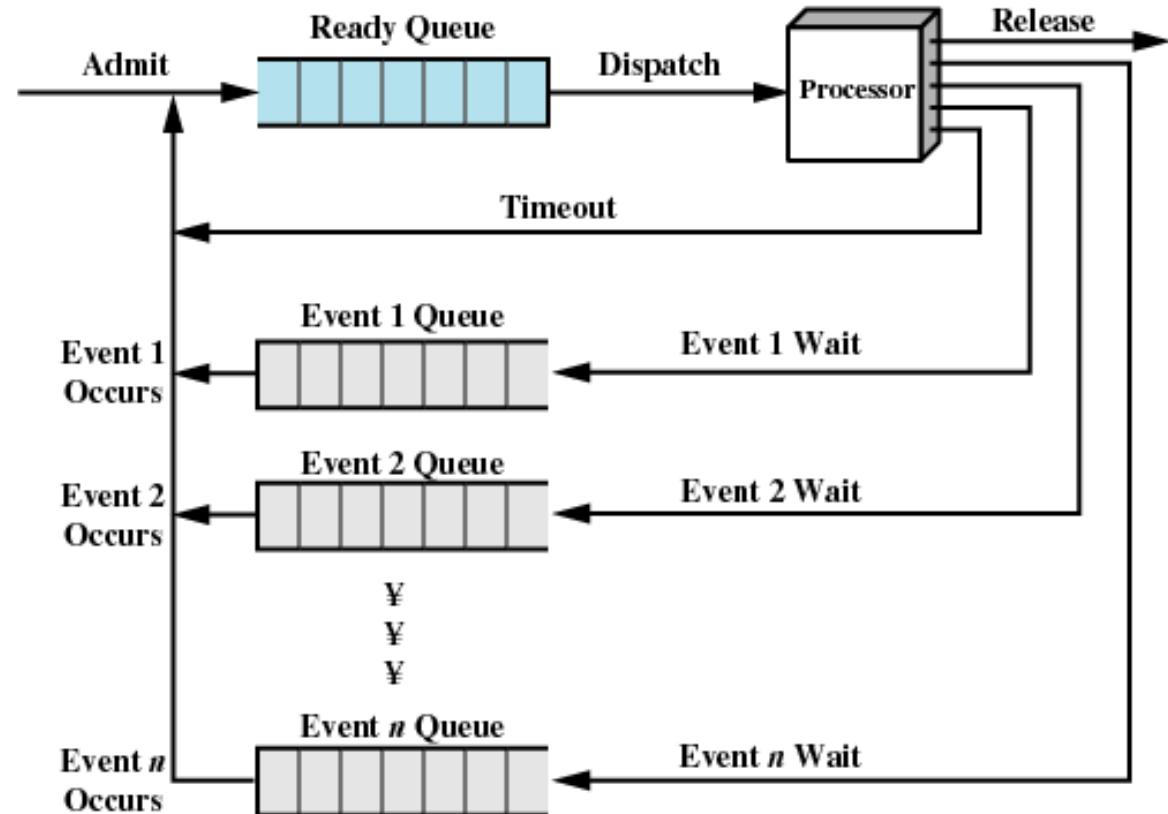
Process States for Trace

Using Two Queues



Single Blocked Queue

Multiple Blocked Queues



Multiple Blocked Queues

Suspended Processes

✓ A process becomes suspend state when it is swapped to disk

◎ User context of a process is moved from main memory to hard disk

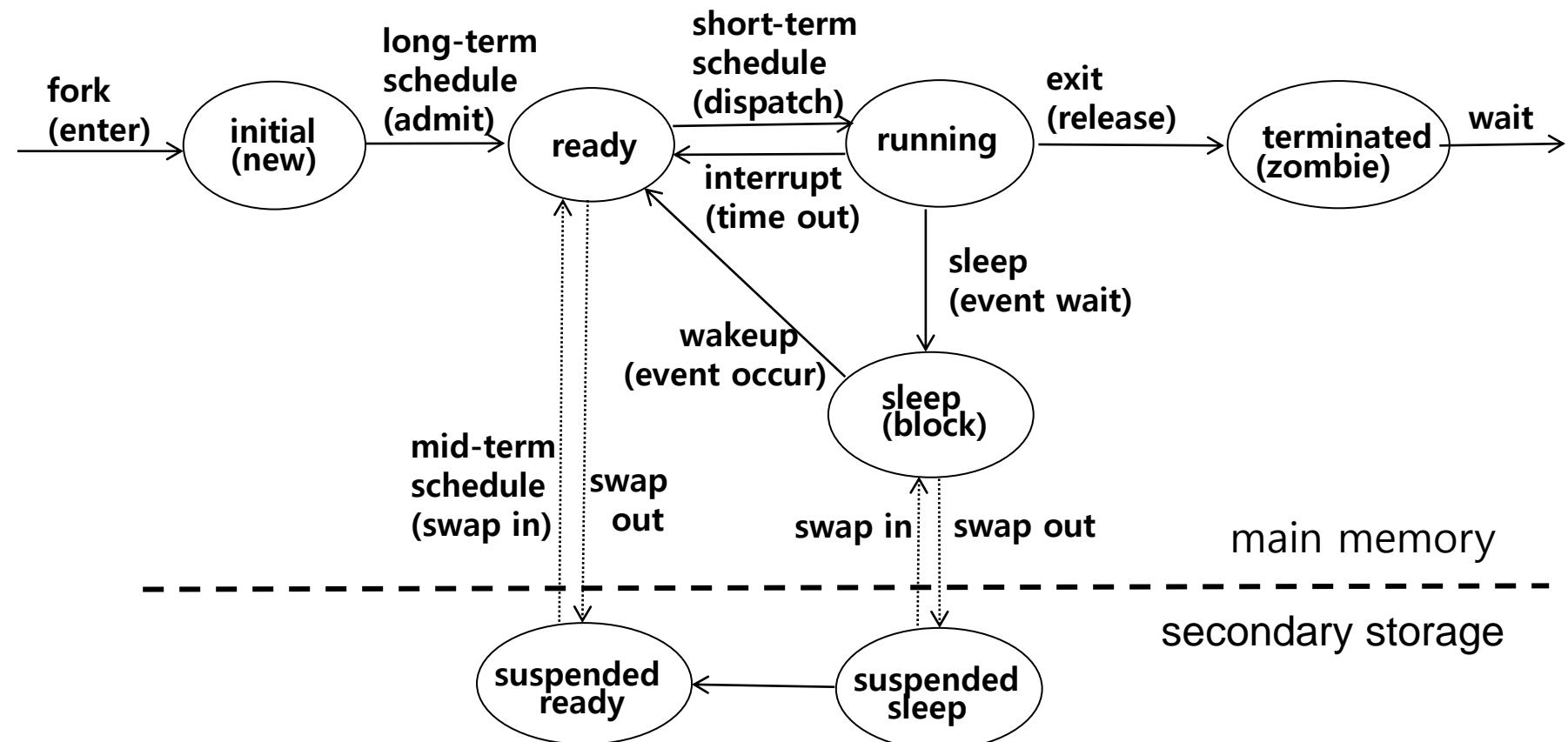
✓ Why?

- ◎ Processor is faster than I/O so all processes could be waiting for I/O
- ◎ Swap these processes out to disk to free up more memory

✓ Two New States

- ◎ Blocked/Suspend (suspended block): blocked state in hard disk
- ◎ Ready/Suspend (suspended ready) : ready state in hard disk

Two Suspend States



Reasons for Process suspension

✓ Swapping

- ◎ The operating system needs to release sufficient main memory to bring in a process that is ready to execute.

✓ Timing

- ◎ A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.

✓ Interactive User Request

- ◎ A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.

Reasons for Process

Parent Process Request

- ① A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

Other OS Reason

- ① The operating system may suspend a background or utility process or a process that is suspected of causing a problem.

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제8강

Process Control Block

충남대학교 인공지능학과/컴퓨터융합학부

최 훈



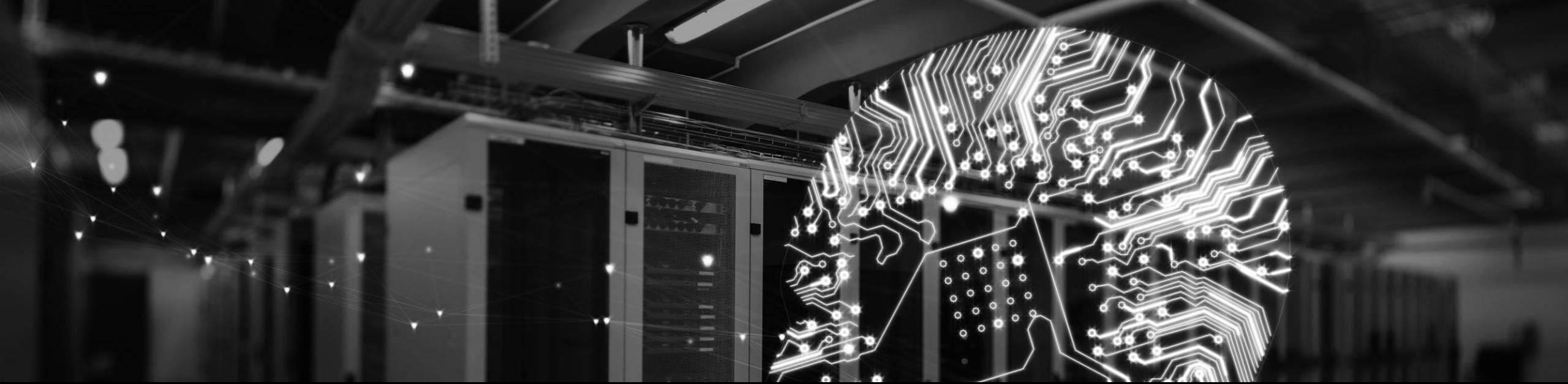
학습 목표

- Process Control Block(PCB) 정의를 이해한다.
- PCB에 저장되는 정보 종류를 파악한다.
- 레지스터(Register) 컨텍스트를 이해한다.

학습 내용

- Process Control Block(PCB) 정의
- PCB에 저장되는 정보의 종류
- 레지스터(Register) 컨텍스트





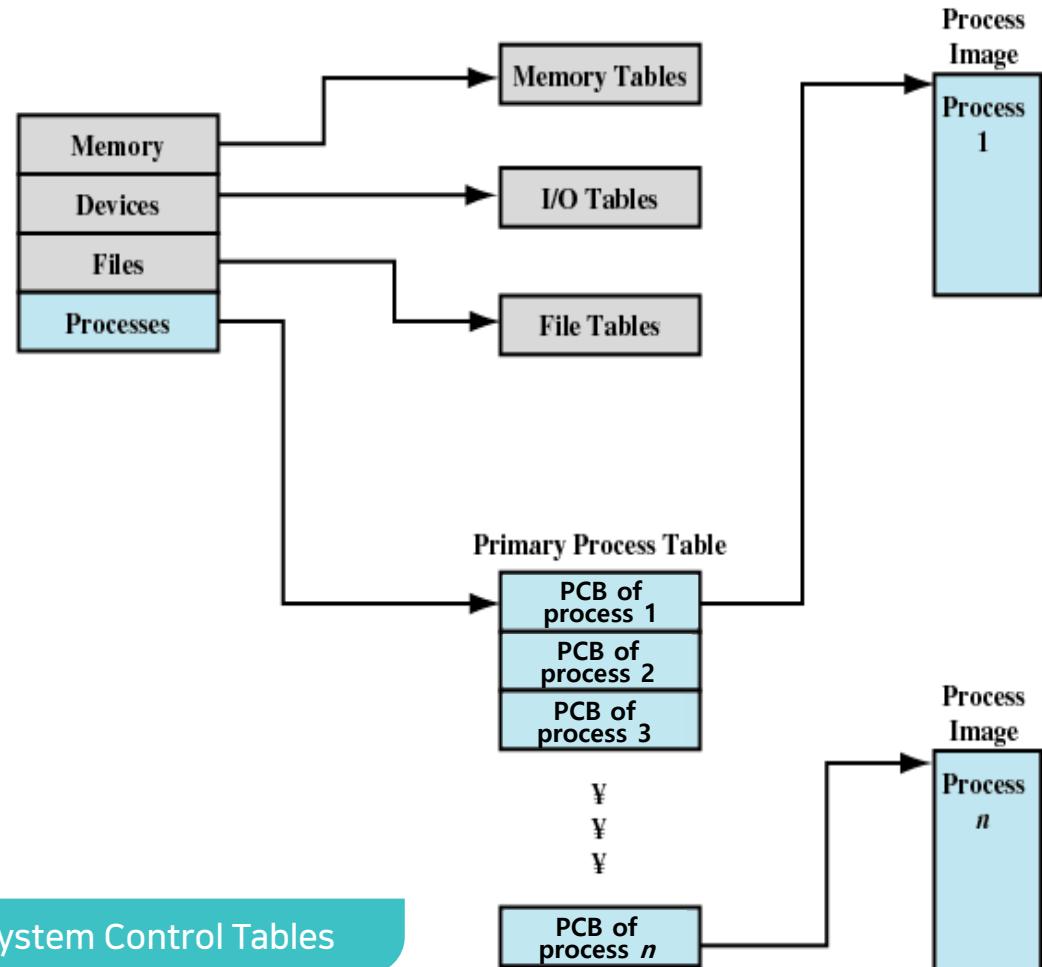
Process Control Block



Process Description

✓ Control Data Structures

- ① Data structures used by OS to control processes
- ② Information about the current status of each process and resource
- ③ Tables are constructed for each entity the operating system manages



Process Control Block

Contains the Process Attributes

- ◎ Identifier
- ◎ State, Priority
- ◎ Program counter
- ◎ Memory pointers
- ◎ Address of memory context (program code, variables)
- ◎ I/O status information
- ◎ Accounting information

Created and managed by the operating system

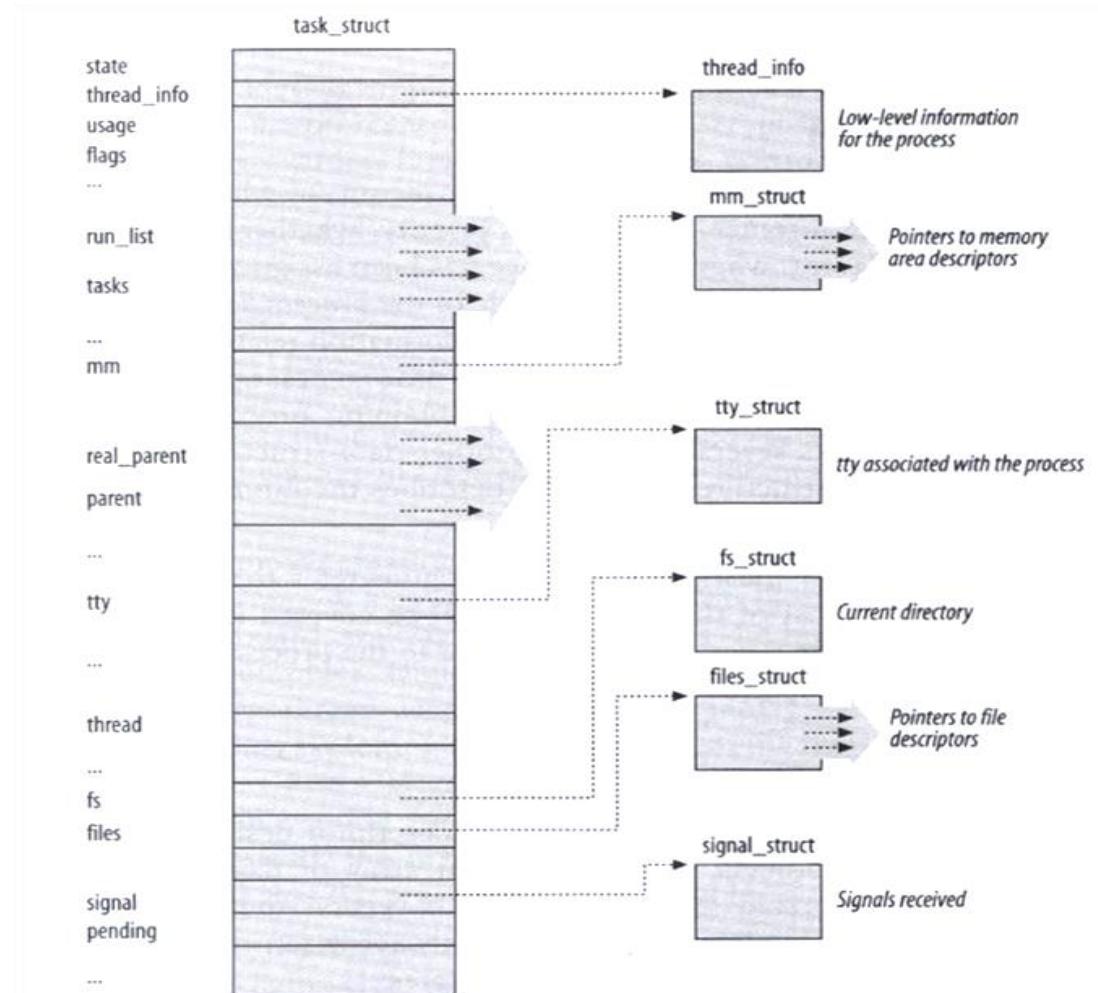
One PCB per Process

Identifier
State
Priority
Program counter
Memory pointers
Context data
I/O status information
Accounting information
⋮
⋮

Simplified Process Control Block

심화

Process Descriptor : PCB of the Linux



심화

Source of the Linux PCB

```
struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    struct thread_info *thread_info;
    ...
               /* Lock depth */

    int prio, static_prio;
    struct list_head run_list;
    struct task_struct *parent; /* parent process */
    /* children/sibling */
    struct list_head children; /* list of my children */
    struct list_head sibling; /* linkage in my parent's
                               children list */
    ...
    struct completion *vfork_done; /* for vfork() */
    int __user *set_child_tid; /* CLONE_CHILD_SETTID */
    int __user *clear_child_tid; /* CLONE_CHILD_CLEARTID */
    unsigned long priority;
    unsigned int time_slice, first_time_slice;
    ...
};
```

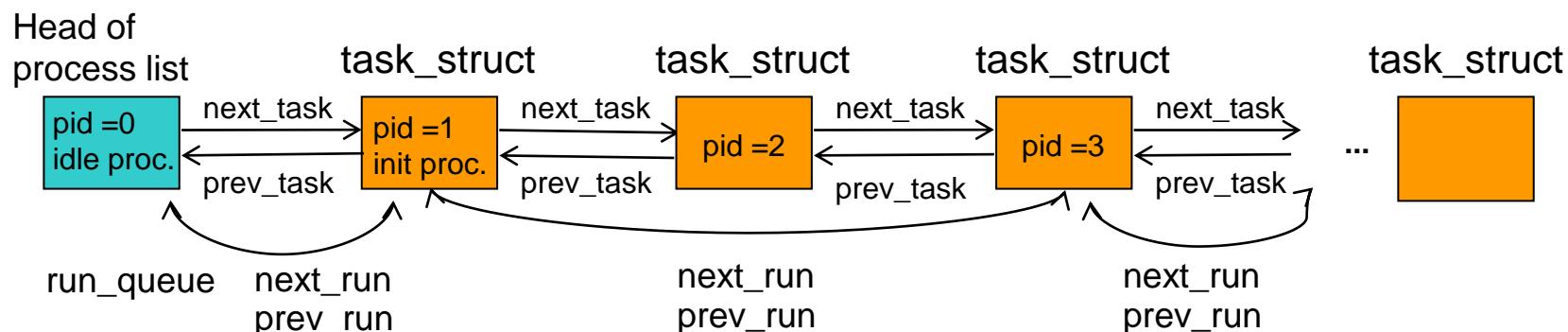
Process Control Block

✓ Process List

- ④ Connects all the process in the computer by double linked list
- ④ Ready queue is made on the process list by linking the processes in the process list whose state is ready state.

✓ Example : Linux Process List

- ④ Head is the process descriptor of the idle(swapper) process → pid =0
- ④ Ready queue of Linux is called the run queue



Process Control Block

Process Identification

◎ Identifiers

- » Numeric identifiers that may be stored with the process control block include
 - Identifier of this process (process identifier)
 - Identifier of the process that created this process (parent process)
 - User identifier

Processor State Information

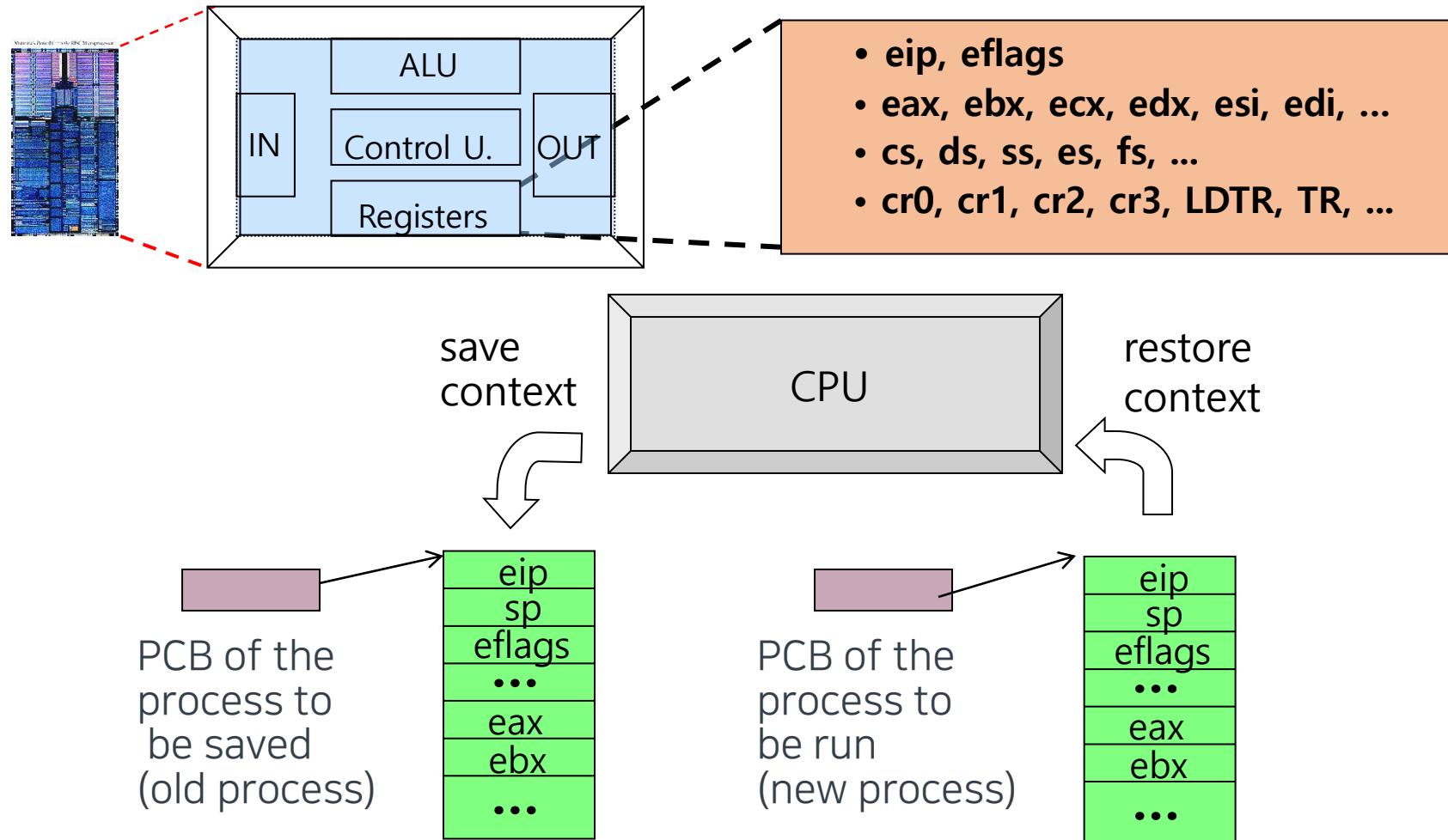
◎ Process state

◎ CPU Registers

- General purpose registers
- Control and Status Registers: Program counter, Condition codes, Status information
- Stack Pointers

Registers

✓ The 80x86 Architecture



Process Control Block

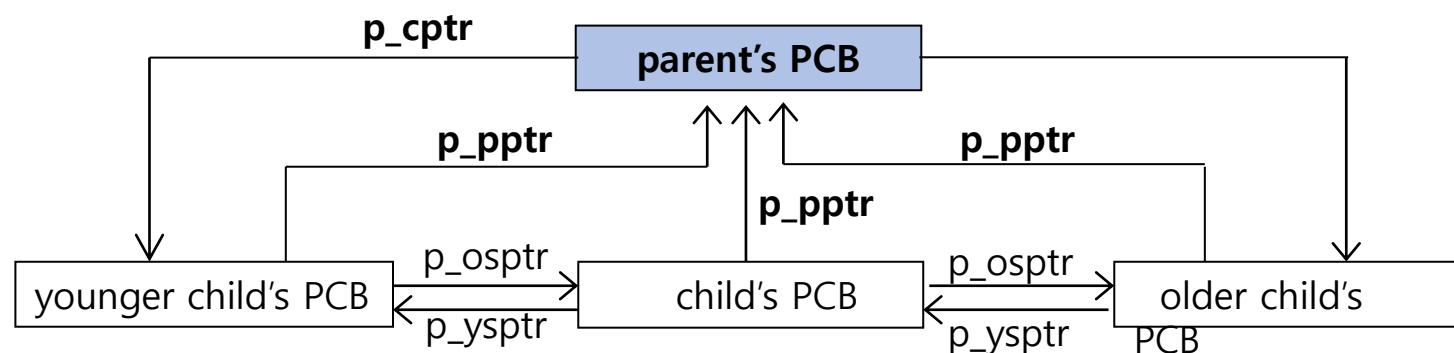
✓ Process Control Information

- ④ Scheduling and State Information: information that is needed by the operating system to perform its scheduling function.
 - » Process state: defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, …).
 - » Priority: to describe the scheduling priority of the process.
 - » Scheduling-related information: depend on the scheduling algorithm used.
 - the amount of time that the process has been waiting
 - the amount of time that the process executed the last time it was running.
 - » Event: Identity of event the process is awaiting before it can be resumed

Process Control Block

✓ Data Structuring

- ① A process may be linked to other process in a queue, ring, or some other structure.
- ② For example, all processes in a waiting state for a particular priority level may be linked in a queue.
 - » A process may exhibit a parent-child (creator-created) relationship with another process.
- ③ The process control block may contain pointers to other processes to support these structures.



Process Control Block

✓ Process Control Information

◎ Inter-process Communication

- » Various flags, signals, and messages may be associated with communication between two independent processes.

◎ Process Privileges

◎ Memory Management

- » May include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

◎ Resource Ownership and Utilization

- » Resources controlled by the process may be indicated, such as opened files.
- » A history of utilization of the processor or other resources may also be included.
- » This information needed by the scheduler

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제9강

Process Context

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



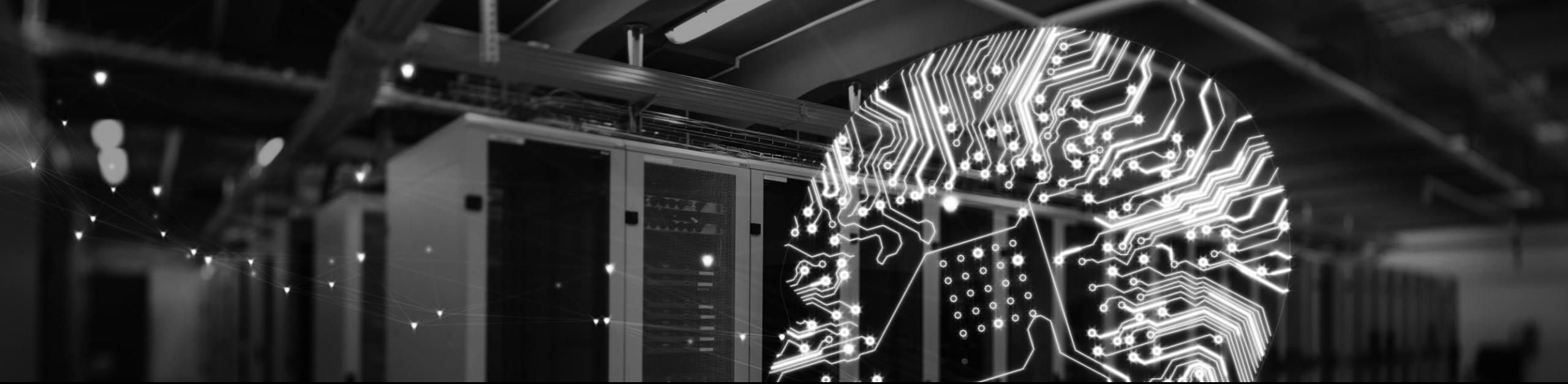
학습 목표

- ❑ 프로세스 컨텍스트(Process Context) 의미를 이해한다.
- ❑ 유저 컨텍스(User Context), 시스템 컨텍스(System Context)를 이해한다.
- ❑ 프로세스 컨텍스트 저장 방법을 이해한다.

학습 내용

- ❑ 프로세스 컨텍스트의 의미
- ❑ 유저 컨텍스, 시스템 컨텍스
- ❑ 프로세스 컨텍스트 저장 방법





Process Context



Process Context

✓ Context : The Execution Environment of a Program

✓ User Context

- ④ Code : User program code (instructions) to be executed
- ④ Data : Global variable of the process
- ④ User Stack
 - » Last-In-First-Out (LIFO) data structure
 - » Used to store the information to invoke user functions
 - *Local variables of the called function, parameters of function, register values (return address)*

✓ System Context

- ④ Kernel Stack (System Stack)
 - Used to store the information to invoke kernel functions
- ④ Process Control Block (PCB)
 - » Data needed by the operating system to control the process

User Part of Process Context

```
/* test.c */

int      glob = 6;
char    buffer[2048];
char    buf[] = "a write to stdout\n"; Uninitialized global variable : bss area

int main(void)
{
    int var;
    pid_t pid; Local Variable : Allocated in a stack area

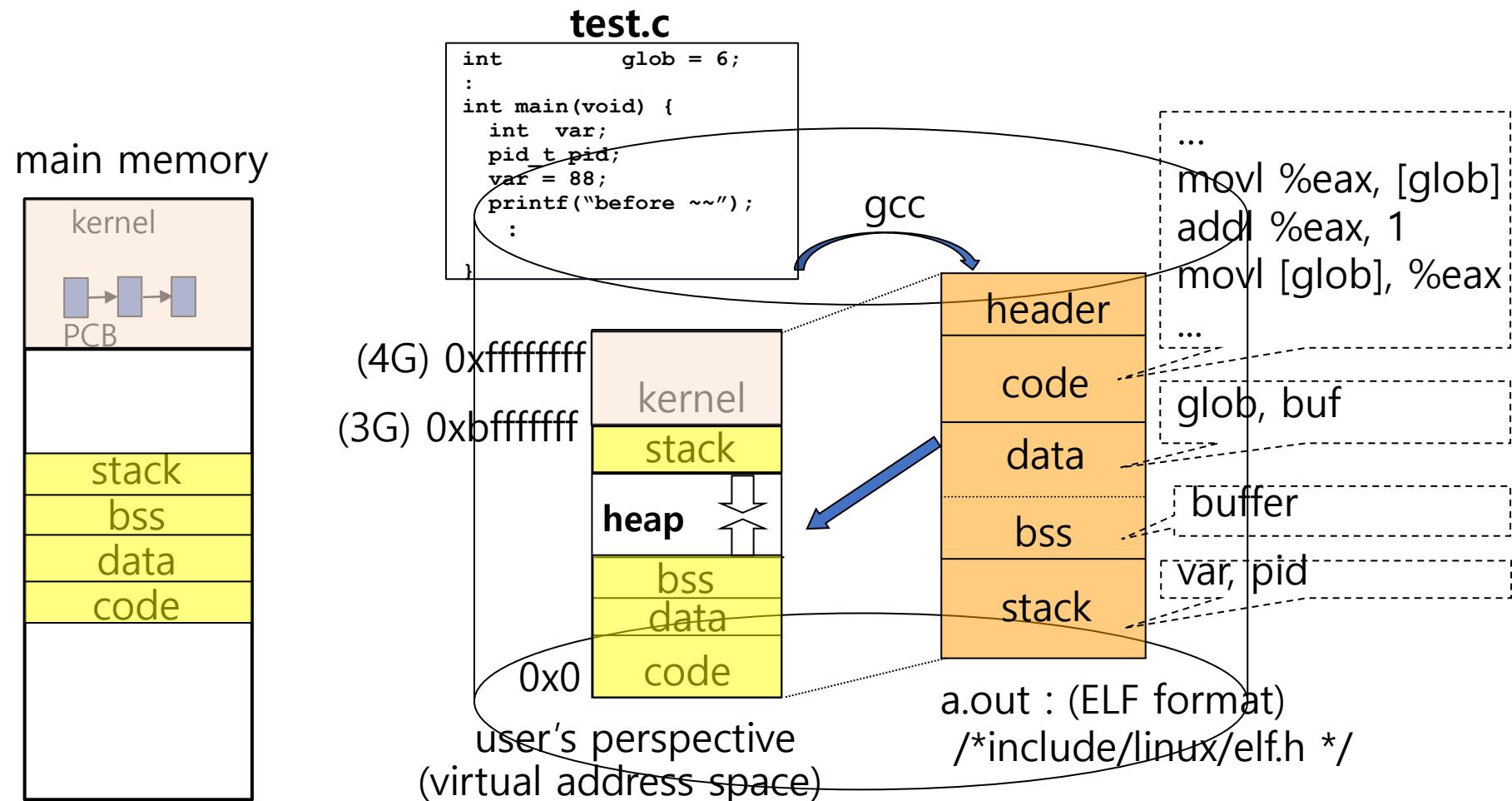
    var = 88;
    printf("before fork\n");

    if ((pid = fork()) == 0) { /* child */
        glob++; var++;
    } else
        sleep(2); /* parent */

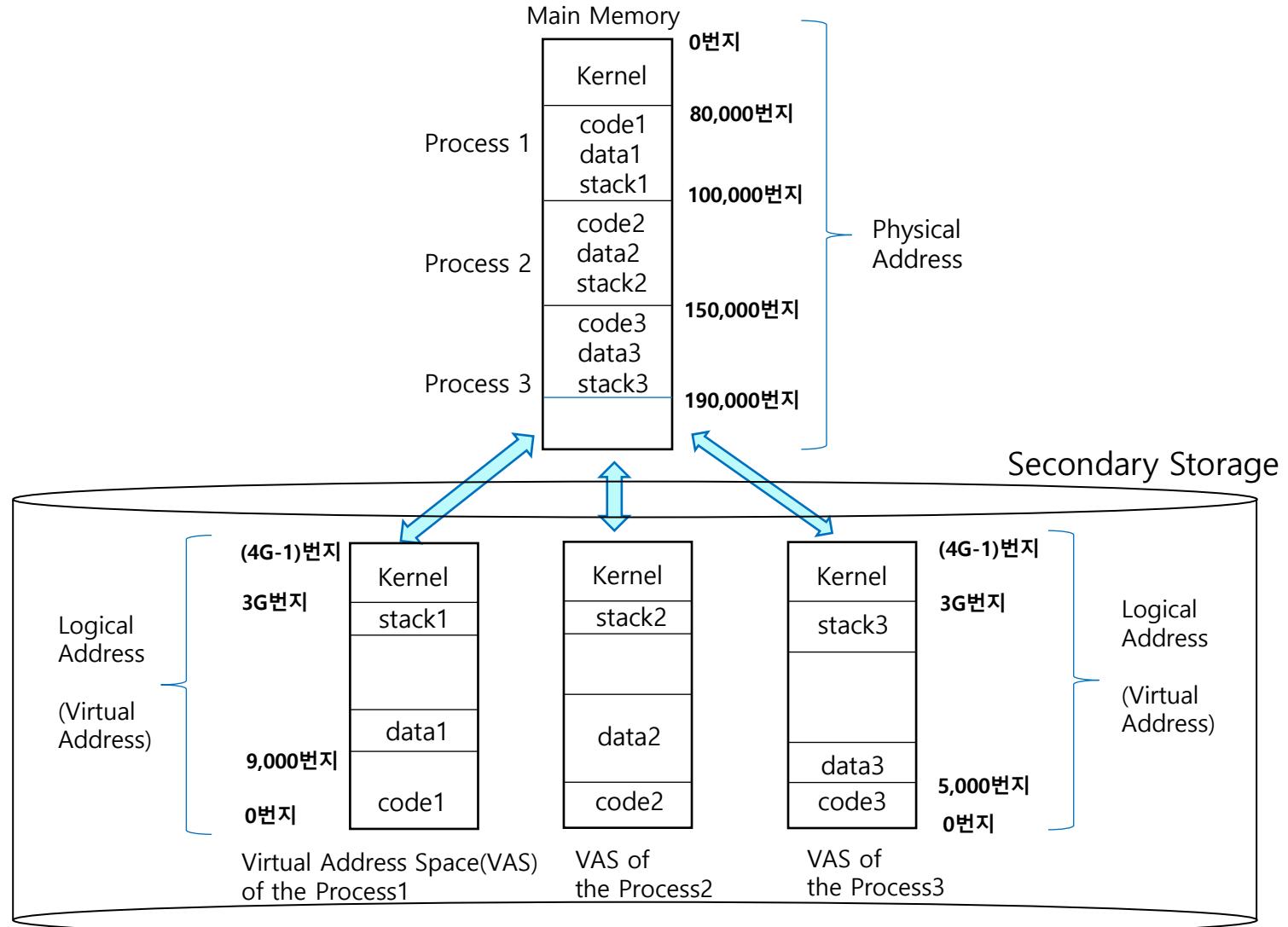
    printf("pid=%d,glob=%d,var=%d\n", getpid(), glob, var);
    exit (0);
} (Source : Adv. programming in the UNIX Env., pgm 8.1)
```

User Part of Process Context

Process Creation : Creation of the virtual addr. space from execution file



Main Memory, Virtual Address Space



Context Saving in the User Stack(1)

✓ Contents of the user stack when executing line 2~5 of the main()

◎ When this program begins, stack layer 1 is created for main() function.

```
① int glob=10;  
  
② int main(argc,argv) {  
③     int var;  
④     var = 33;  
⑤     ...  
⑥     glob=func1(var,50);  
⑦     printf(glob, var);  
⑧ }  
  
⑨ int func1(int x,int y) {  
⑩     int w=0;  
⑪     w = x + y;  
⑫     func2(w+10);  
⑬     return(w);  
⑭ }  
  
⑮ int func2(int z){  
⑯     int a=20;  
⑰     printf(a,z,glob);  
⑱ }
```

execution
of
main()
function

register values, return address
local variables : var=33
arguments : argv[0], argv[1]..., argv[argc-1]

User
Stack
Layer 1

User stack

Context Saving in the User Stack(2)

✓ Contents of the user stack when executing line 6, 9~11

◎ When func1() is called, layer 2 is pushed in the stack.

```

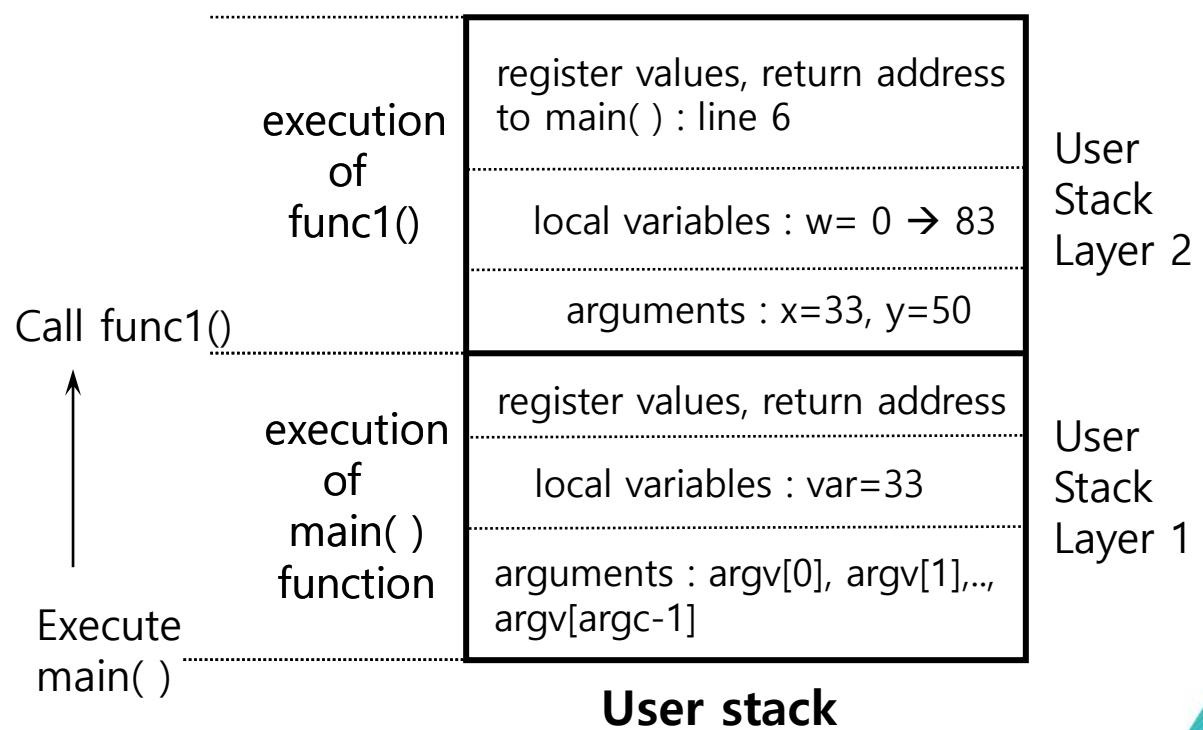
① int glob=10;

② int main(argc,argv) {
③     int var;
④     var = 33;
⑤     ...
⑥     glob=func1(var,50);
⑦     printf(glob, var);
⑧ }

⑨ int func1(int x,int y) {
⑩     int w=0;
⑪     w = x + y;
⑫     func2(w+10);
⑬     return(w);
⑭ }

⑮ int func2(int z){
⑯     int a=20;
⑰     printf(a,z,glob);
⑱ }

```



Context Saving in the User Stack(3)

✓ Contents of the user stack when executing line 12, 15~18

```

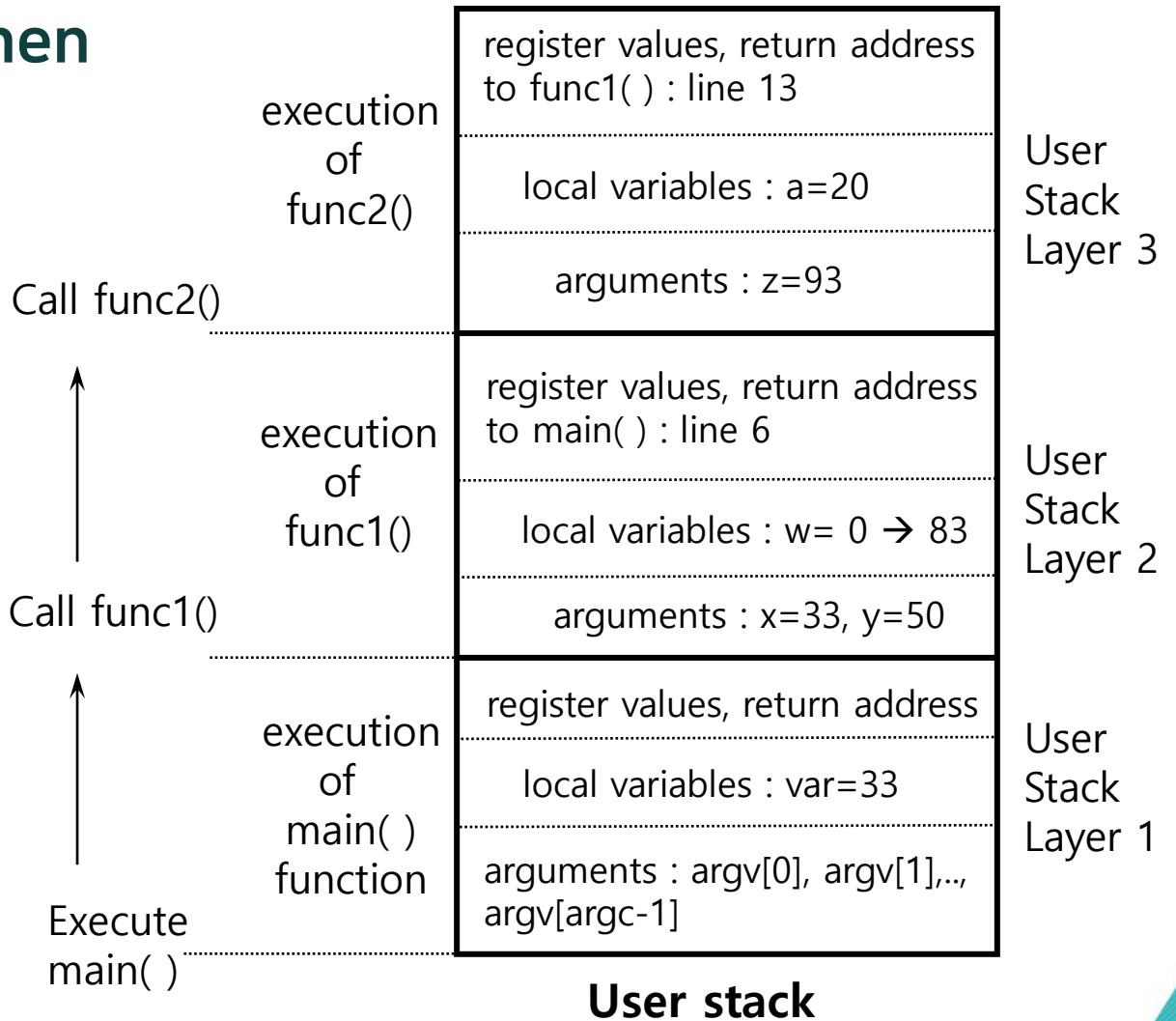
① int glob=10;

② int main(argc,argv) {
③     int var;
④     var = 33;
⑤     ...
⑥     glob=func1(var,50);
⑦     printf(glob, var);
⑧ }

⑨ int func1(int x,int y) {
⑩     int w=0;
⑪     w = x + y;
⑫     func2(w+10);
⑬     return(w);
⑭ }

⑮ int func2(int z){
⑯     int a=20;
⑰     printf(a,z,glob);
⑱ }

```



Context Saving in the User Stack(4)

✓ User stack when executing line 13,14 after returning from func2()

④ Accessing a or z in main() or func1() is not possible. They do not exist.

```

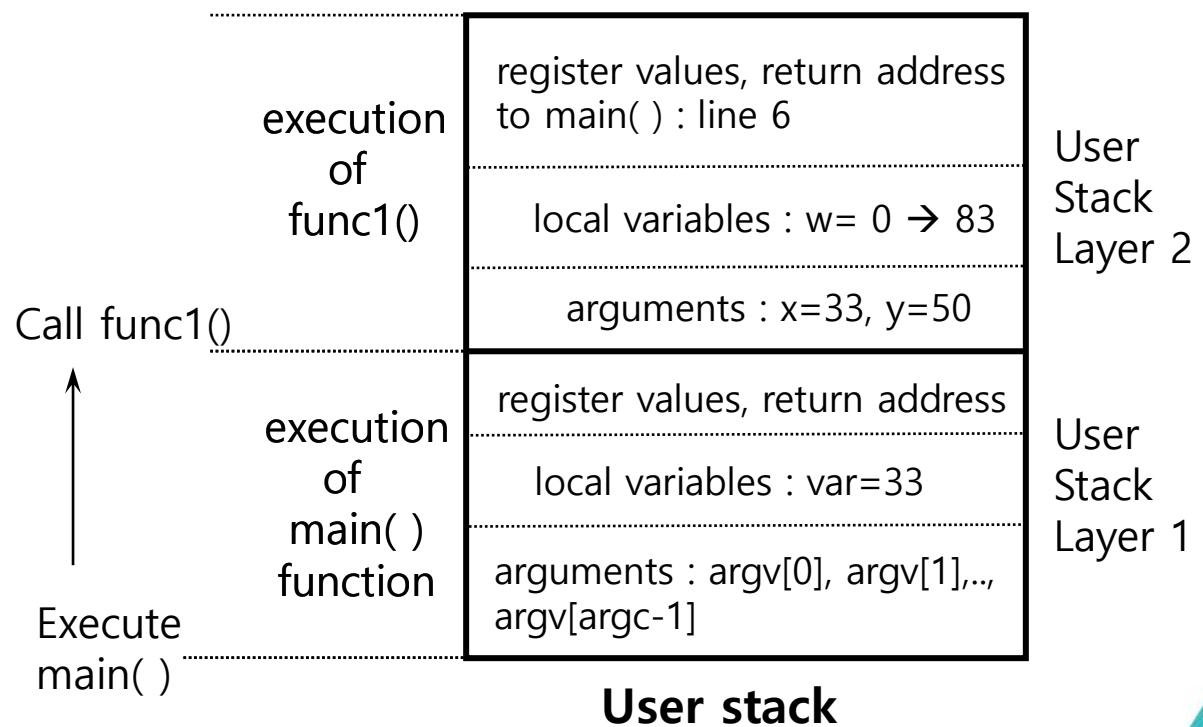
① int glob=10;

② int main(argc,argv) {
③     int var;
④     var = 33;
⑤     ...
⑥     glob=func1(var,50);
⑦     printf(glob, var);
⑧ }

⑨ int func1(int x,int y) {
⑩     int w=0;
⑪     w = x + y;
⑫     func2(w+10);
⑬     return(w);
⑭ }

⑮ int func2(int z){
⑯     int a=20;
⑰     printf(a,z,glob);
⑱ }

```



Context Saving in the User Stack(5)

✓ User stack when executing line 6~8 after returning from func1()

④ Accessing w in main() is not possible. It does not exist now.

```
① int glob=10;  
  
② int main(argc,argv) {  
③     int var;  
④     var = 33;  
⑤     ...  
⑥     glob=func1(var,50);  
⑦     printf(glob, var);  
⑧ }  
  
⑨ int func1(int x,int y) {  
⑩     int w=0;  
⑪     w = x + y;  
⑫     func2(w+10);  
⑬     return(w);  
⑭ }  
  
⑮ int func2(int z){  
⑯     int a=20;  
⑰     printf(a,z,glob);  
⑱ }
```

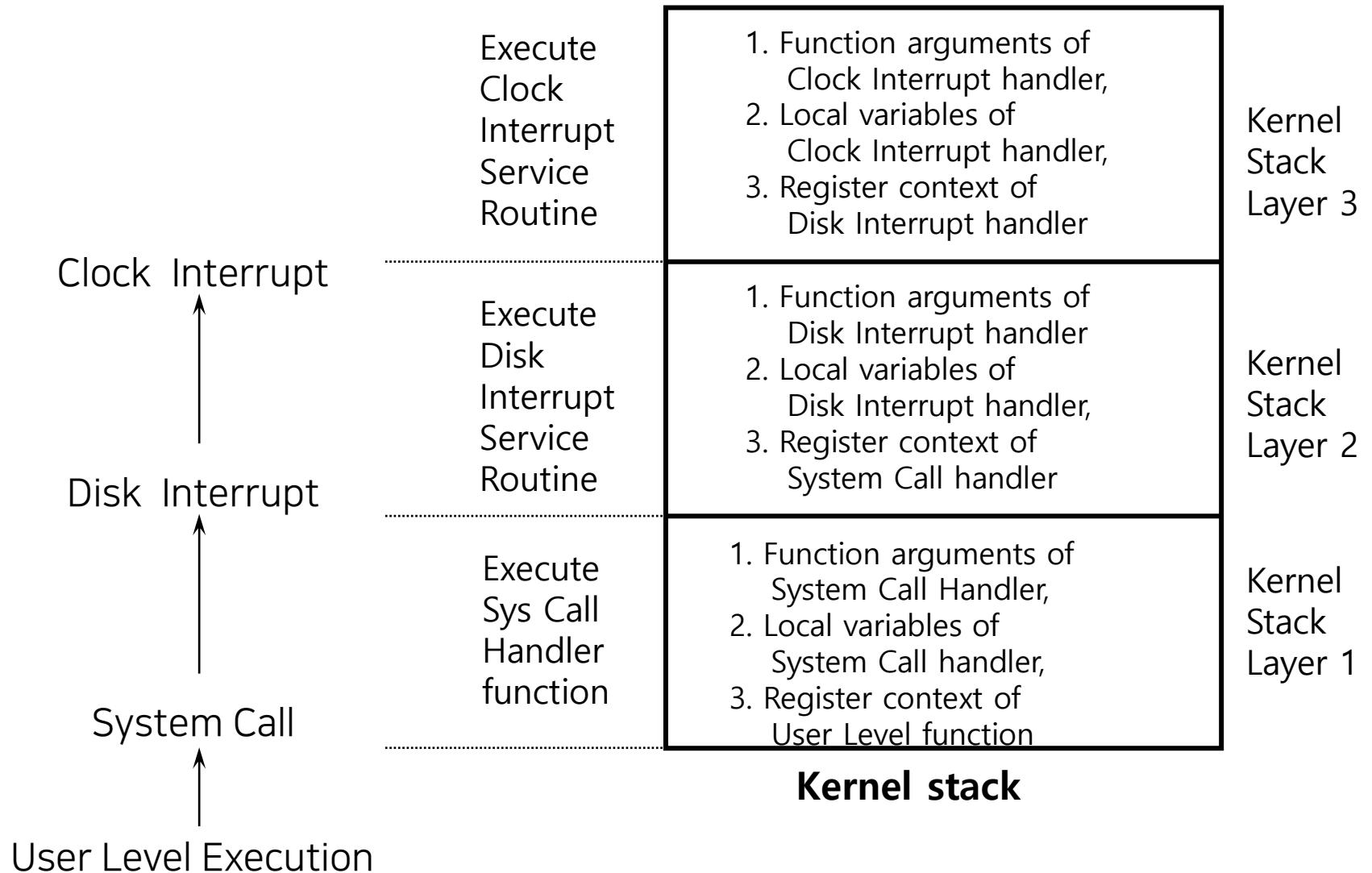
execution
of
main()
function

register values, return address
local variables : var=33
arguments : argv[0], argv[1]...,
argv[argc-1]

User
Stack
Layer 1

User stack

Context Saving in the Kernel Stack



Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제10강

Process Control : Creation

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



학습 목표

- 💬 프로세스 제어 개념을 이해 이해한다.
- 💬 프로세스 생성 절차를 이해한다.
- 💬 프로세스의 유저 컨텍스트(User Context) 생성 방법을 이해한다.
- 💬 부모/자식 프로세스 관계를 이해한다.

학습 내용

- ⌚ 프로세스 제어 개념
- ⌚ 프로세스 생성 절차
- ⌚ 프로세스의 유저 컨텍스트 생성 방법
- ⌚ 부모/자식 프로세스 관계





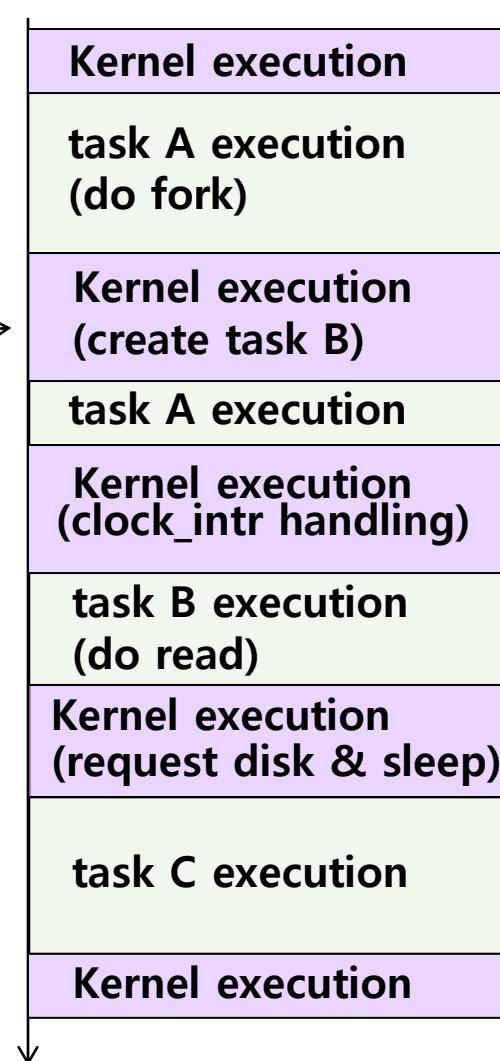
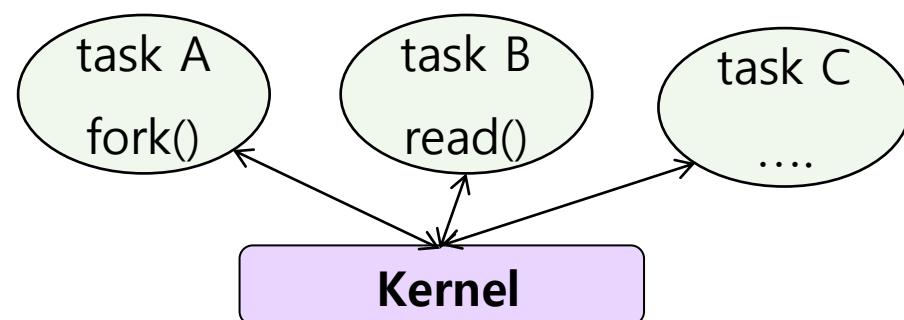
Process Control : Creation

Process Control

✓ Modes of Execution

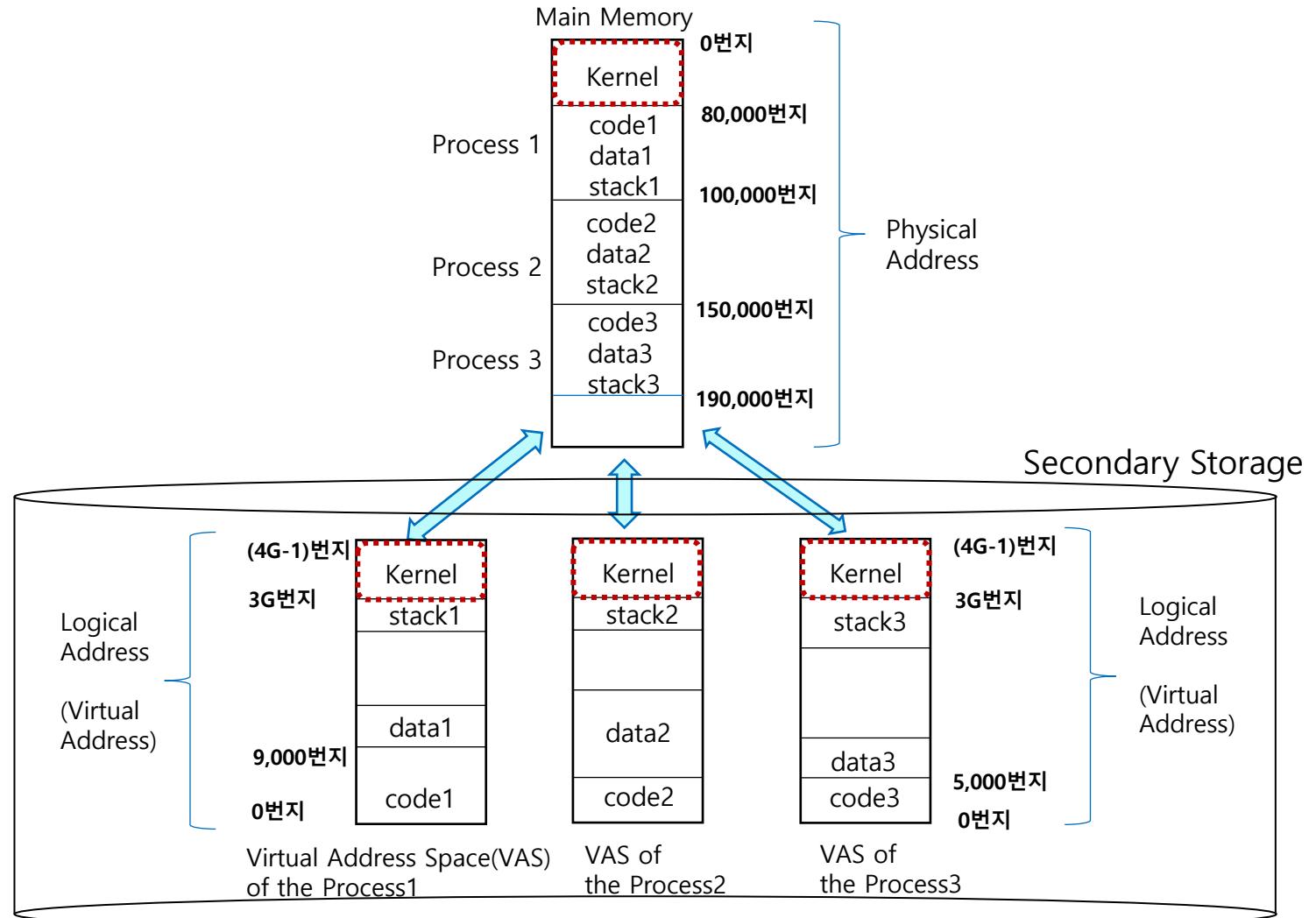
- ① User mode →
 - » Less-privileged mode
 - » User programs execute in this mode
- ② System mode (= Kernel mode, Supervisor Mode) →
 - » More-privileged mode
 - » Kernel of the operating system executes in this mode

✓ Mode Change : User Mode ↔ System Mode



Recall
Process Context

Main Memory, Virtual Address Space



Recall
 Definition Of Process

Main Memory, Virtual Address Space

1	5000	27	12004
2	5001	28	12005
3	5002	<hr/> Time out	
4	5003	29	100
5	5004	30	101
6	5005	31	102
<hr/> Time out		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	<hr/> Time out	
16	8003	41	100
<hr/> I/O request		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
<hr/> Time out			

100=Starting address of dispatcher program

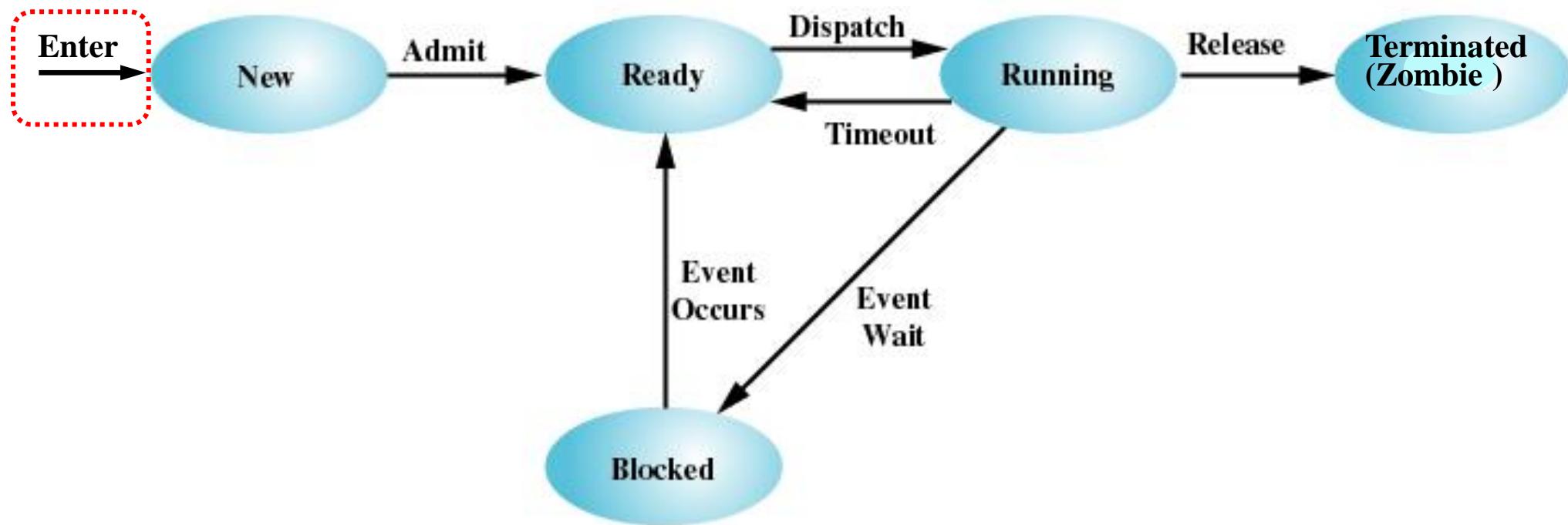
Shaded areas indicate execution of dispatcher process;
 First and third columns count instruction cycles;
 Second and fourth columns show address of
 instruction being executed

Steps for Process Creation

1. Create a PCB data structure for the new process
2. Assign a unique process identifier (PID) to the child process
3. Set values of the process control block
4. Set up appropriate linkages
 - Ex : Add new process to linked list of parent, sibling processes
5. Create or expand other data structures
 - Ex : Increments counters for any files owned by parents, to reflect that an additional process now also owns those files
6. Create the user context : time consuming job
 - Copy the user context of the parent process
7. Assigns the child process to the Ready state and insert it to a ready queue
8. Returns the process identifier of the child to the parent process, and a 0 value to the child process

Recall
Process State Model

Five-State Process Model



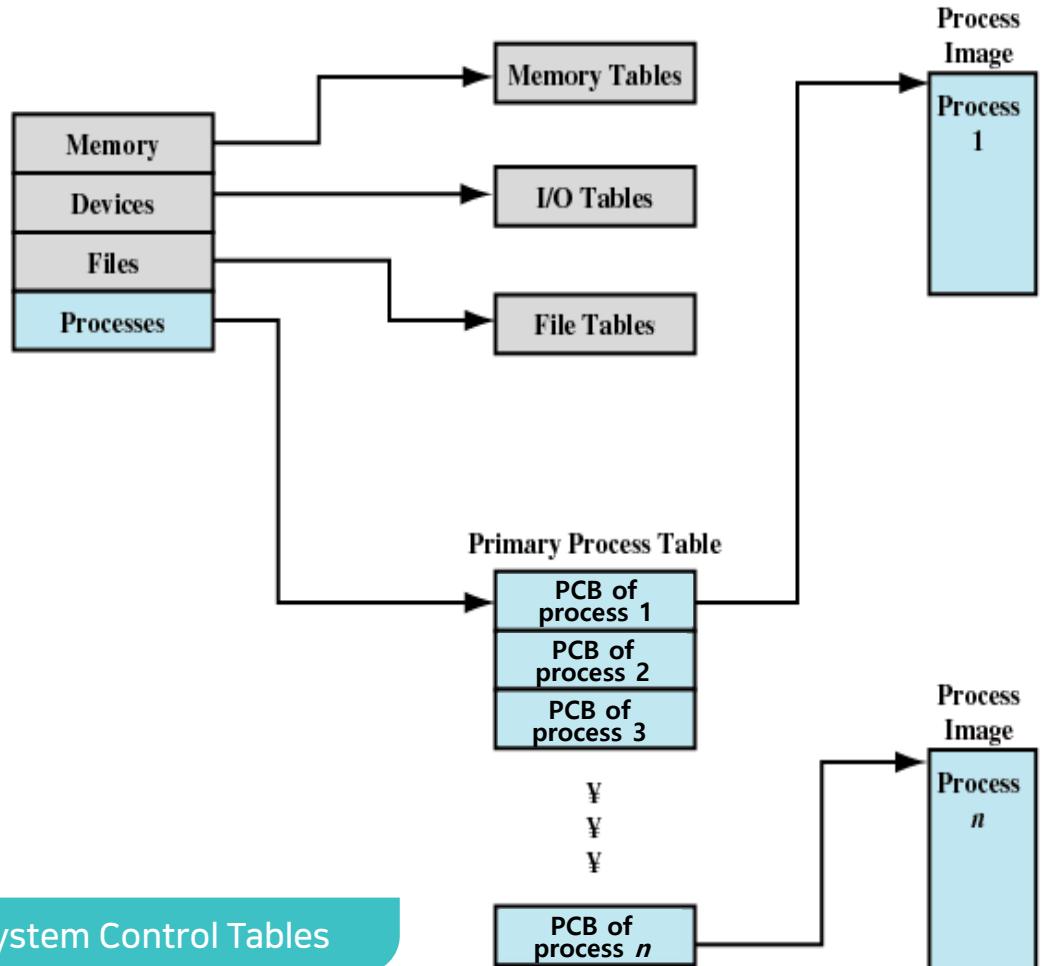
Five-State Process Model

Recall
What is a Process?

Process Description

✓ Control Data Structures

- ① Data structures used by OS to control processes
- ② Information about the current status of each process and resource
- ③ Tables are constructed for each entity the operating system manages



General Structure of Operating System Control Tables

Recall
PCB

Process Control Block (PCB)

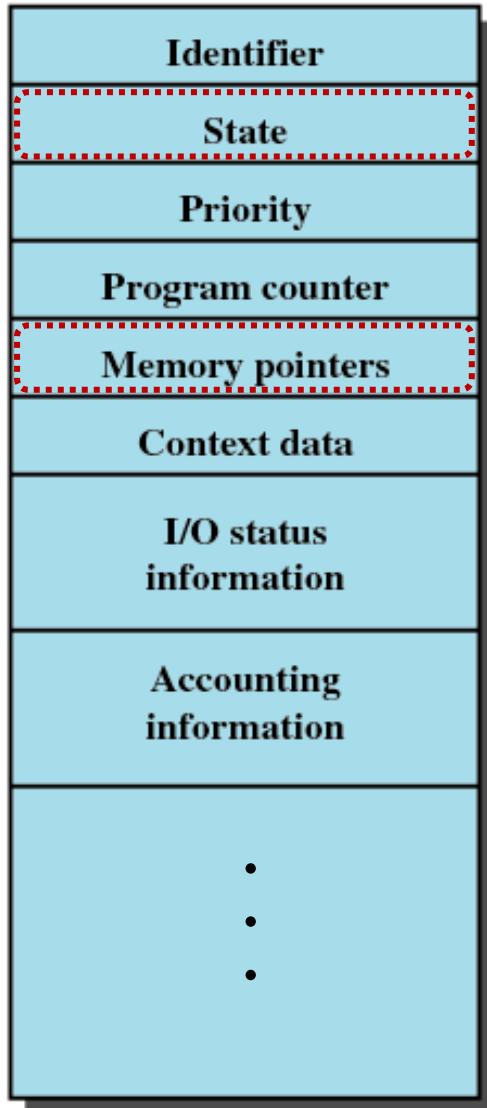
✓ Contains the process attributes

- ◎ Identifier
- ◎ State, Priority
- ◎ Program counter
- ◎ Memory pointers
- ◎ Address of memory context (program code, variables)
- ◎ I/O status information
- ◎ Accounting information

✓ Created and managed by the operating system

✓ One PCB per process

Simplified Process Control Block



Recall
PCB

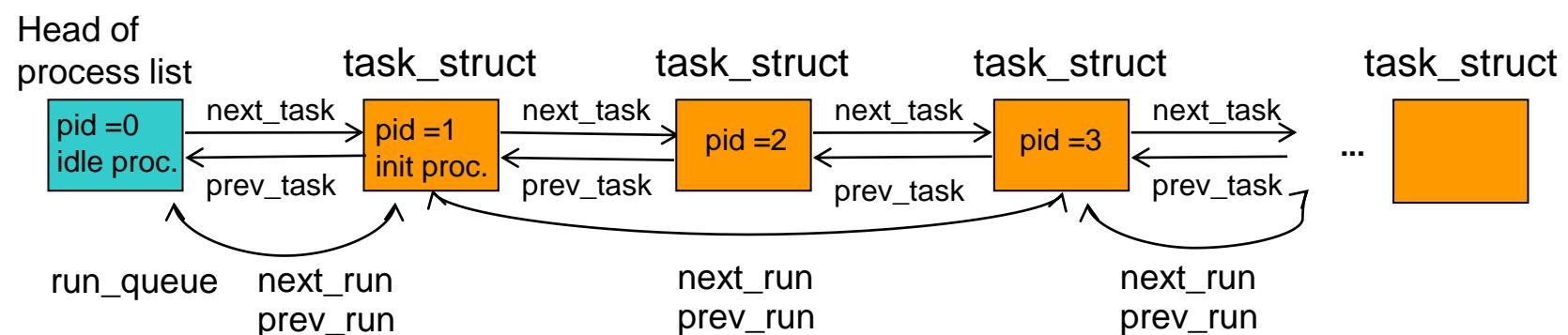
Process Control Block

✓ Process list

- ④ Connects all the process in the computer by double linked list
- ④ Ready queue is made on the process list by linking the processes in the process list whose state is ready state.

✓ Example : Linux process list

- ④ Head is the process descriptor of the idle(swapper) process → pid =0
- ④ Ready queue of Linux is called the run queue

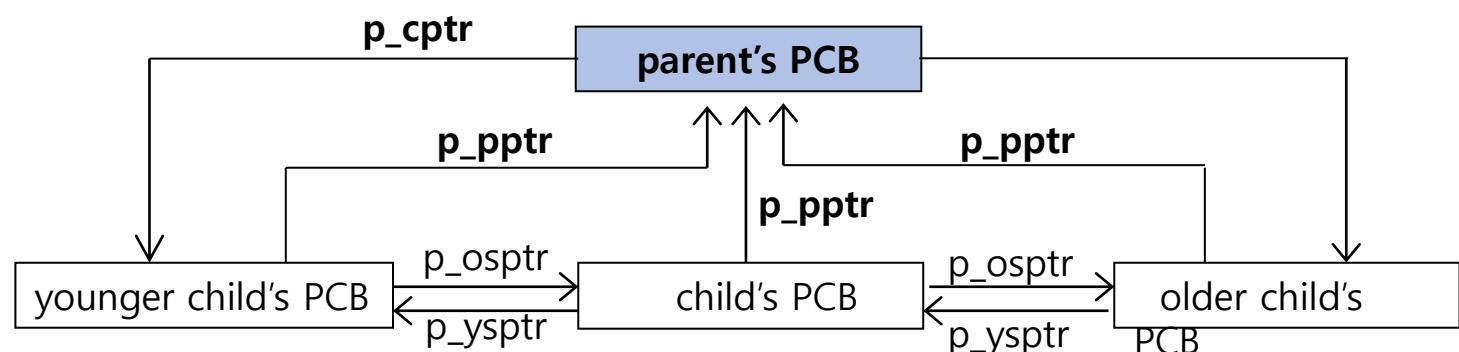


Recall
PCB

Process Control Block

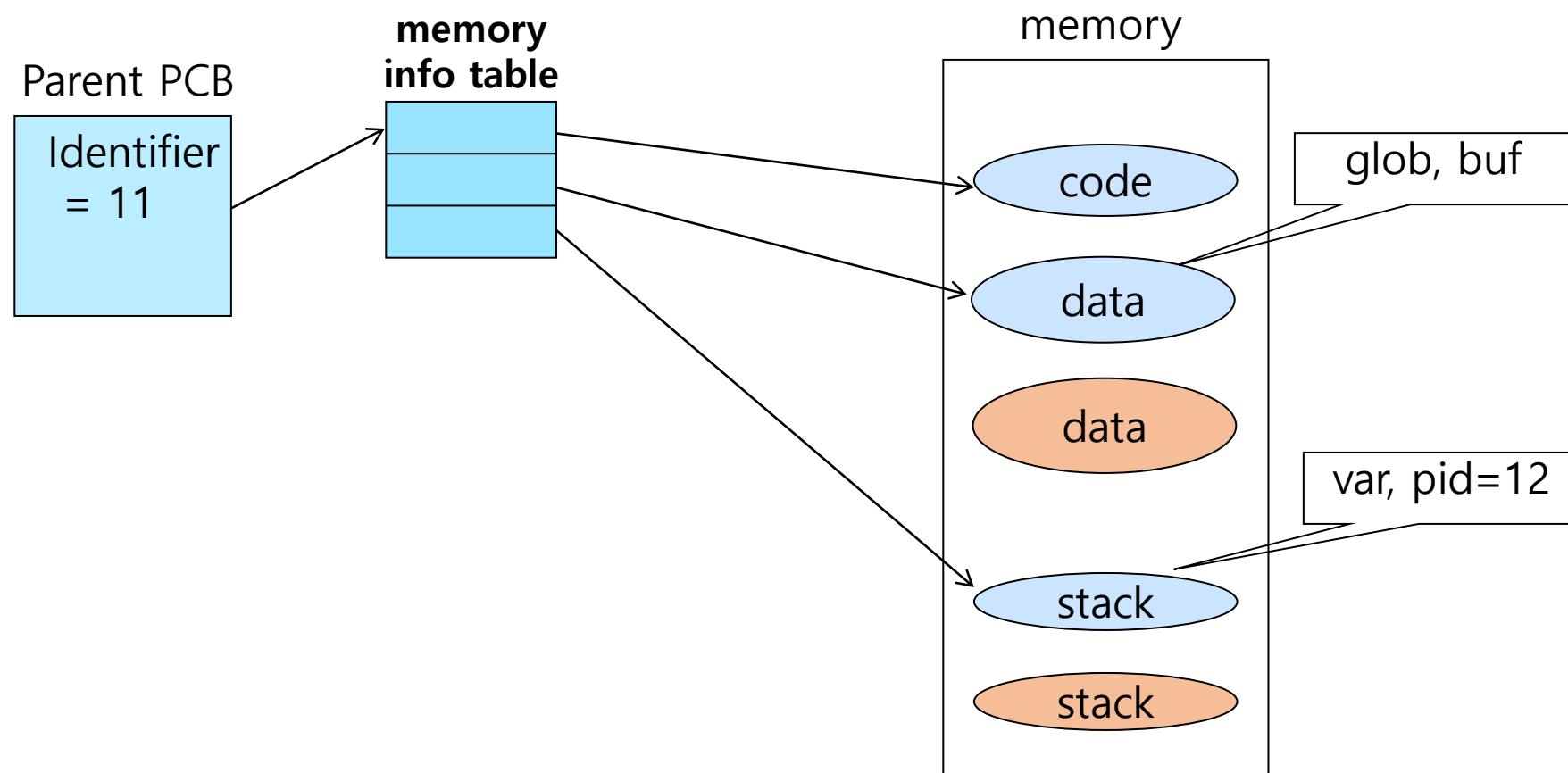
✓ Data Structuring

- ④ A process may be linked to other process in a queue, ring, or some other structure.
- ④ For example, all processes in a waiting state for a particular priority level may be linked in a queue.
 - » A process may exhibit a parent-child(creator-created) relationship with another process.
- ④ The process control block may contain pointers to other processes to support these structures.



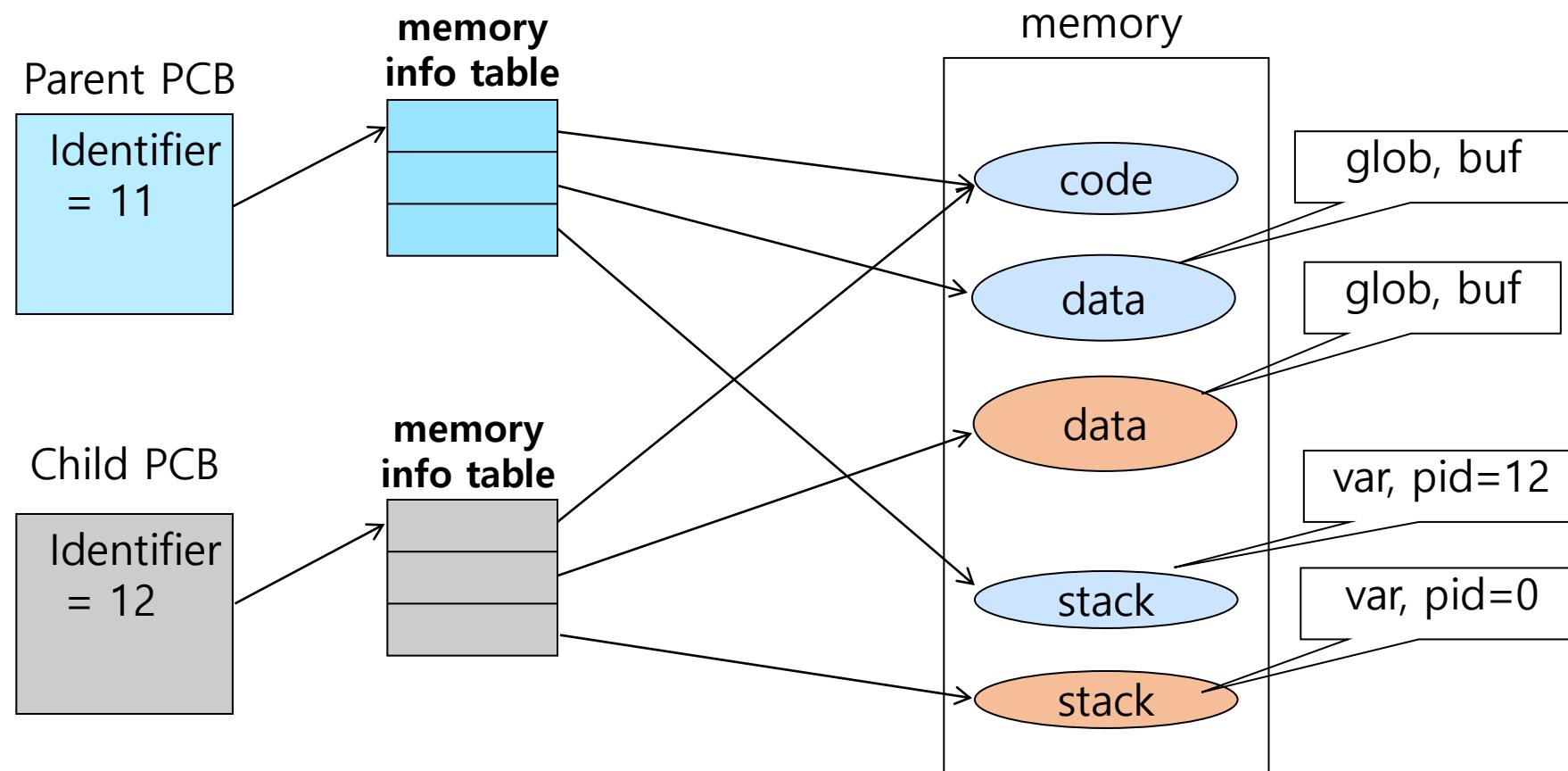
Process Creation : Relation with Parent(1)

✓ How 'fork()' Works : Before Fork (After run a.out)



Process Creation : Relation with Parent(2)

- After Fork() : Information(PCB, Resources(file, ...) and memory context) for child process is copied from that of parent process



Recall
Process Context

User Part of Process Context

```
/* test.c */
int    glob = 6;
char   buffer[2048];
char   buf[] = "a write to stdout\n";
int main(void)
{
    int var;
    pid_t pid;

    var = 88;
    printf("before fork\n");

    if ((pid = fork()) == 0) { /* child */
        glob++; var++;
    } else
        sleep(2); /* parent */

    printf("pid=%d,glob=%d,var=%d\n", getpid(), glob, var);
    exit (0);
}
```

Uninitialized global variable : bss area

Initialized global Variable : data area

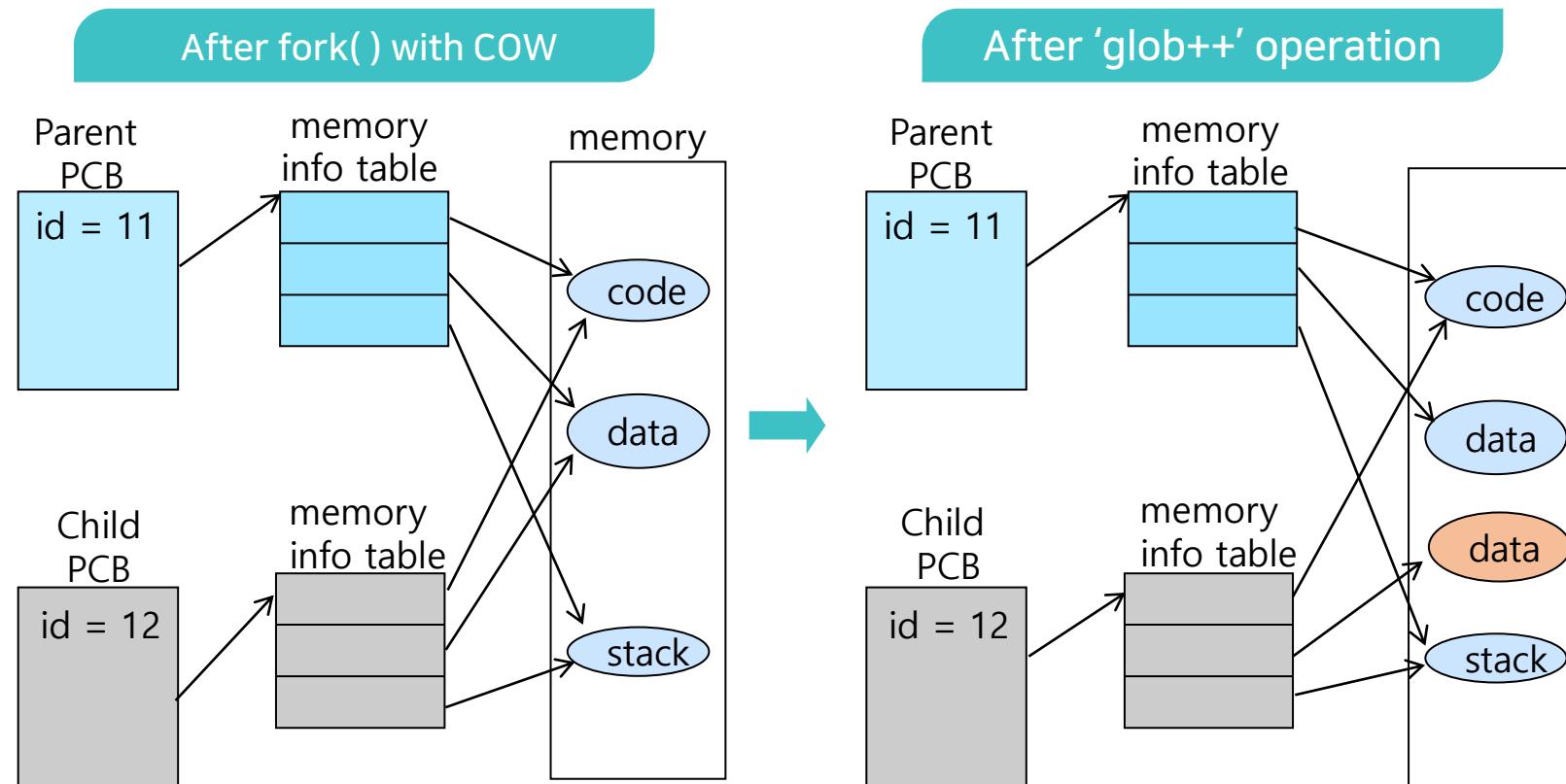
Local Variable : Allocated in a stack area

(Source : Adv. programming in the UNIX Env., pgm 8.1)

Process Creation : Relation with Parent(3)

✓ How 'fork()' works with COW(Copy on Write) mechanism

- ◎ COW allows parent, child processes to initially share the same pages
- ◎ If either process modifies a shared page, only then is the page copied



심화

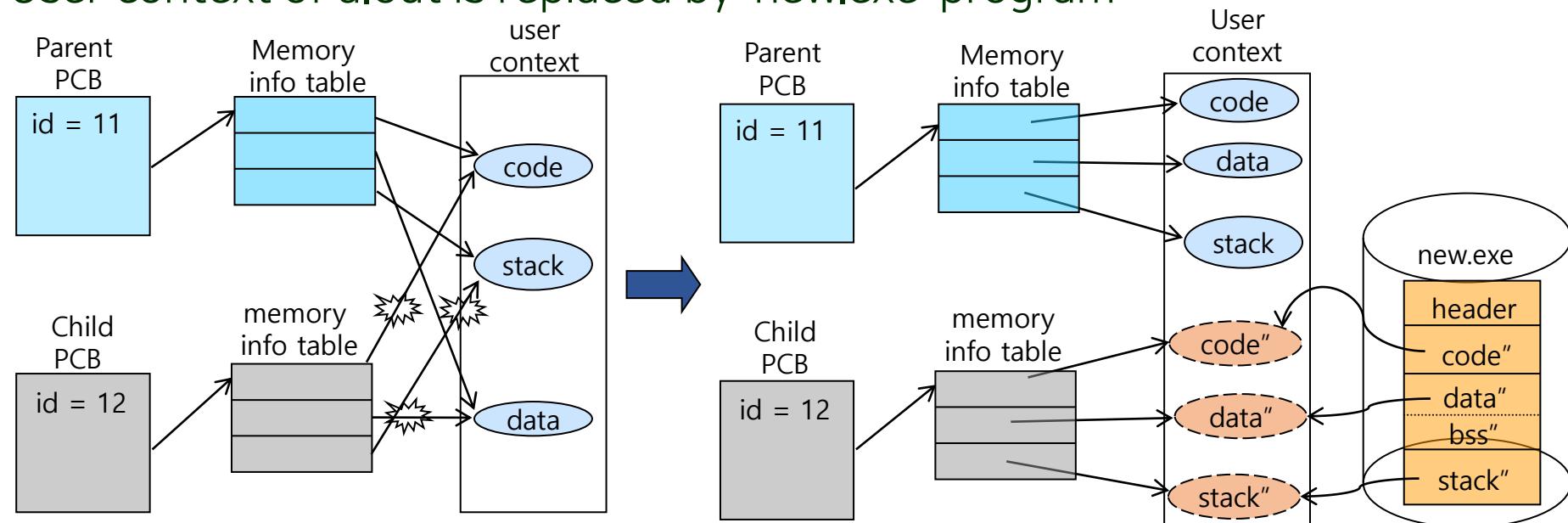
Process Creation : Relation with Parent(4)

✓ How to create a child process which has a different code?

- ④(e.g) parent process is tcsh, child process is pstree
- ④tcsh calls fork() to create tcshchild , tcshchild calls exec() to replace itself with pstree

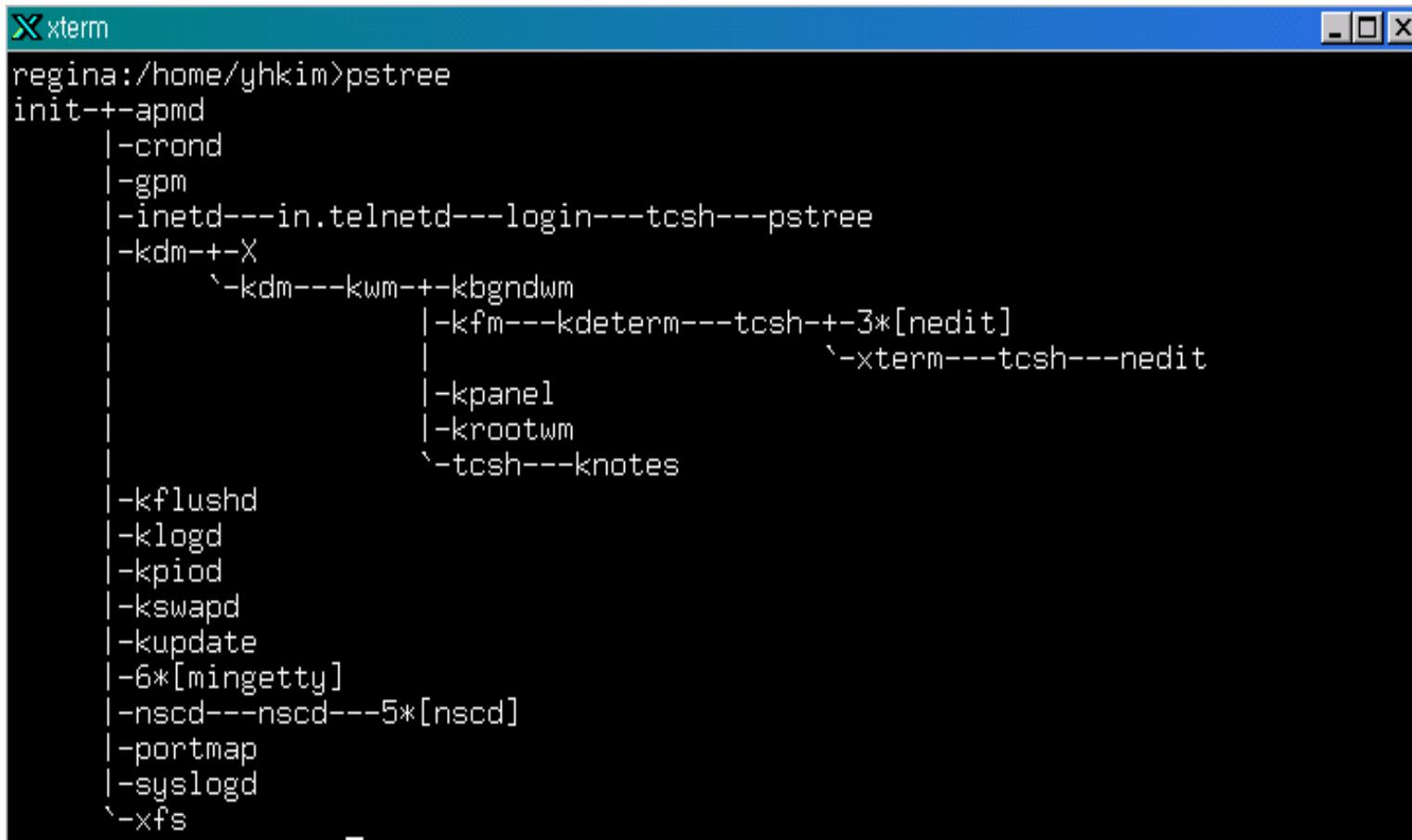
✓ How 'exec("/bin/new.exe",parameters)' works

- ④User context of a.out is replaced by 'new.exe' program



Process Creation : Relation with Parent

- ✓ Parent process create children processes, which, in turn create other processes, forming a tree of processes.



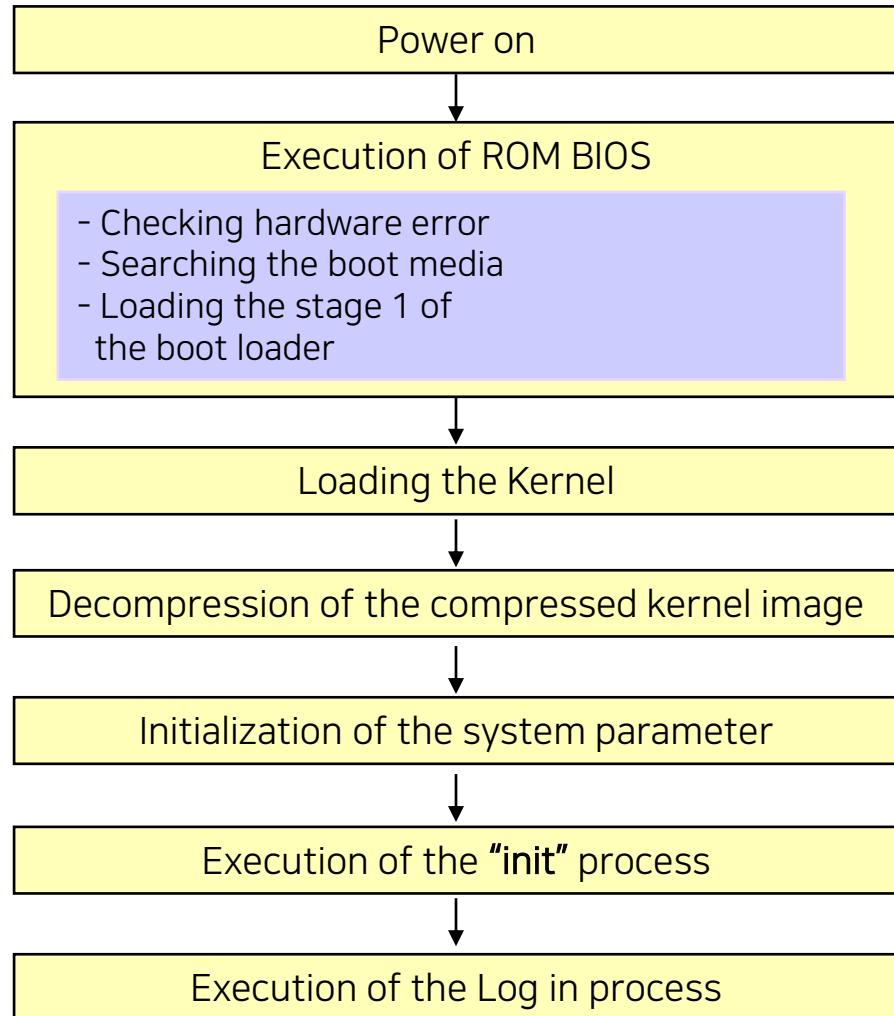
```
xterm
regina:/home/yhkim>pstree
init+-apmd
|-crond
|-gpm
|-inetd---in.telnetd---login---tcsh---pstree
|-kdm+-X
|`-kdm---kwm---kbgrndwm
|   |-kfm---kdeterm---tcsh---+3*[nedit]
|   |           `xterm---tcsh---nedit
|   |-kpanel
|   |-krootwm
|   `tcsh---knotes
|-kflushd
|-klogd
|-kpiod
|-kswapd
|-kupdate
|-6*[mingetty]
|-nscd---nscd---5*[nscd]
|-portmap
|-syslogd
`-xfs
```

심화

Booting a Computer

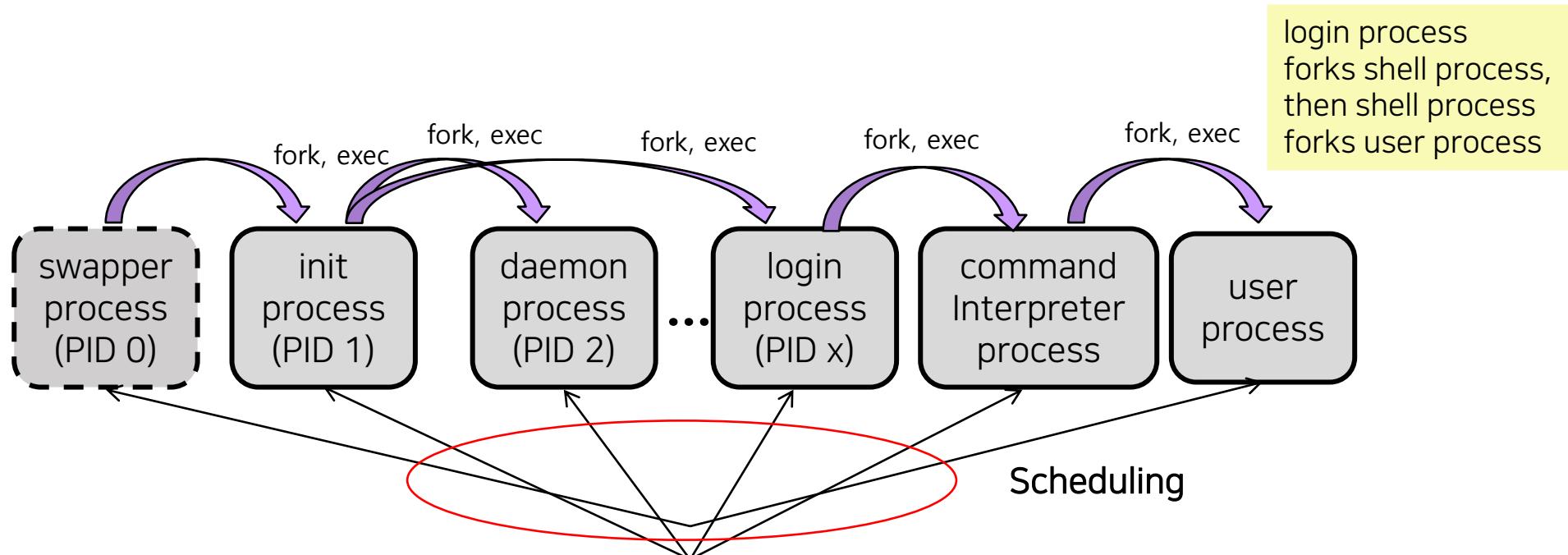
✓ Example : Linux booting process

- ④ Boot loader is located in the boot block of a hard disk
- ④ Boot loader is the first program which is loaded into the main memory and executed by the processor
- ④ Boot loader loads the OS into the main memory
- ④ Then the OS executes the "init" process



심화

Process Creation



Process made manually by OS during the system initialization time



Process made by OS using the fork() system call

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제11강

Process Control : Process Switch

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



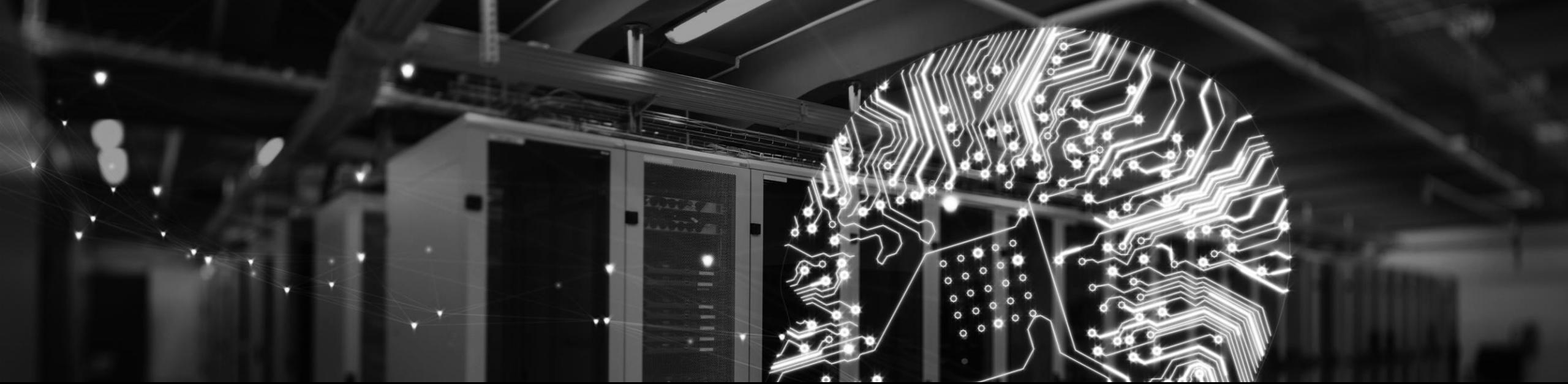
학습 목표

- ▶ 프로세스 스위치(Process Switch)의 의미를 이해한다.
- ▶ 프로세스 스위치가 일어나는 시기를 이해한다.
- ▶ 프로세스 스위치 절차를 이해한다.

학습 내용

- ▶ 프로세스 스위치(Process Switch)의 의미
- ▶ 프로세스 스위치가 일어나는 시기
- ▶ 프로세스 스위치 절차





Process Control : Process Switch



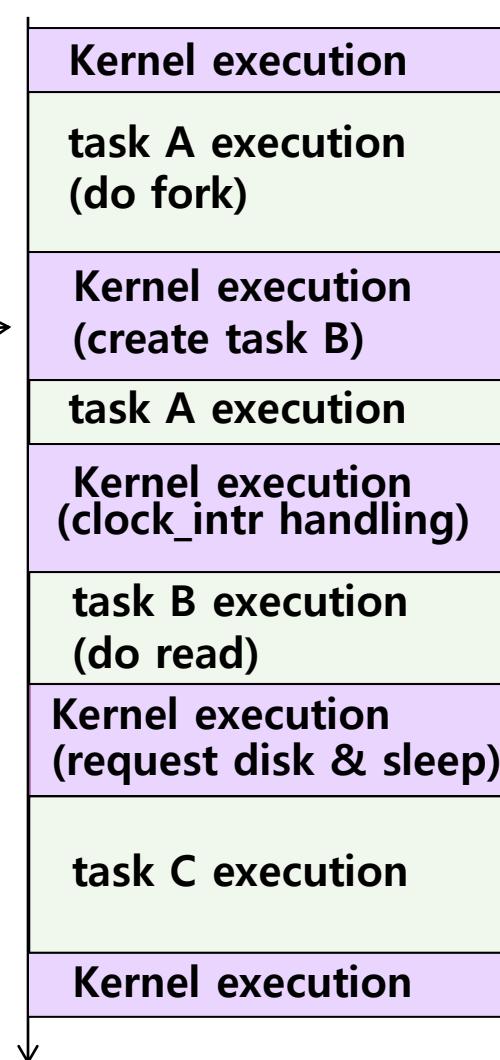
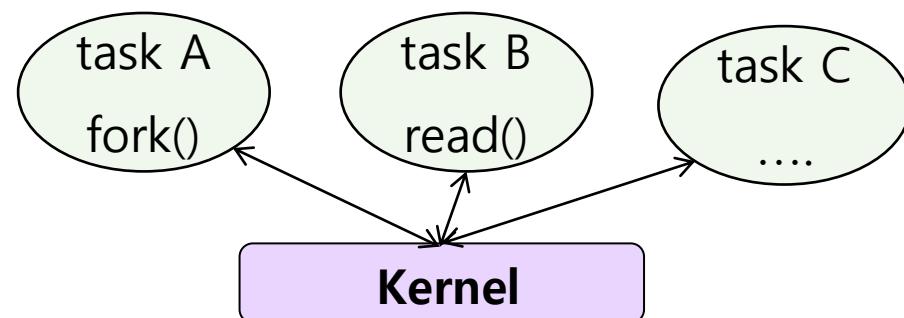
Recall
Process Creation

Process Control

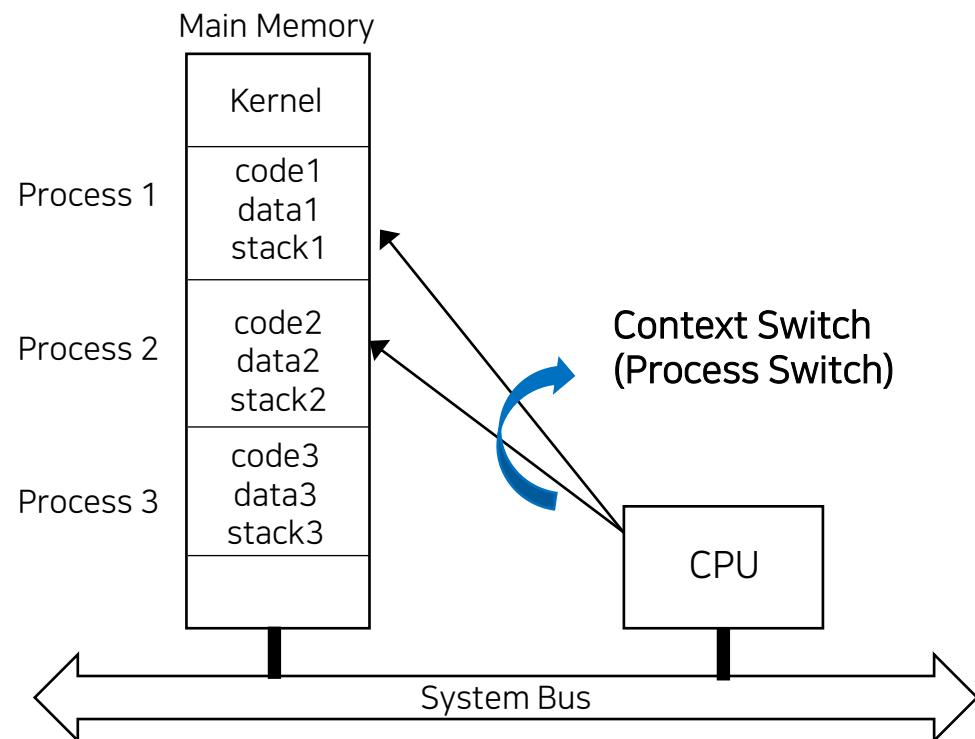
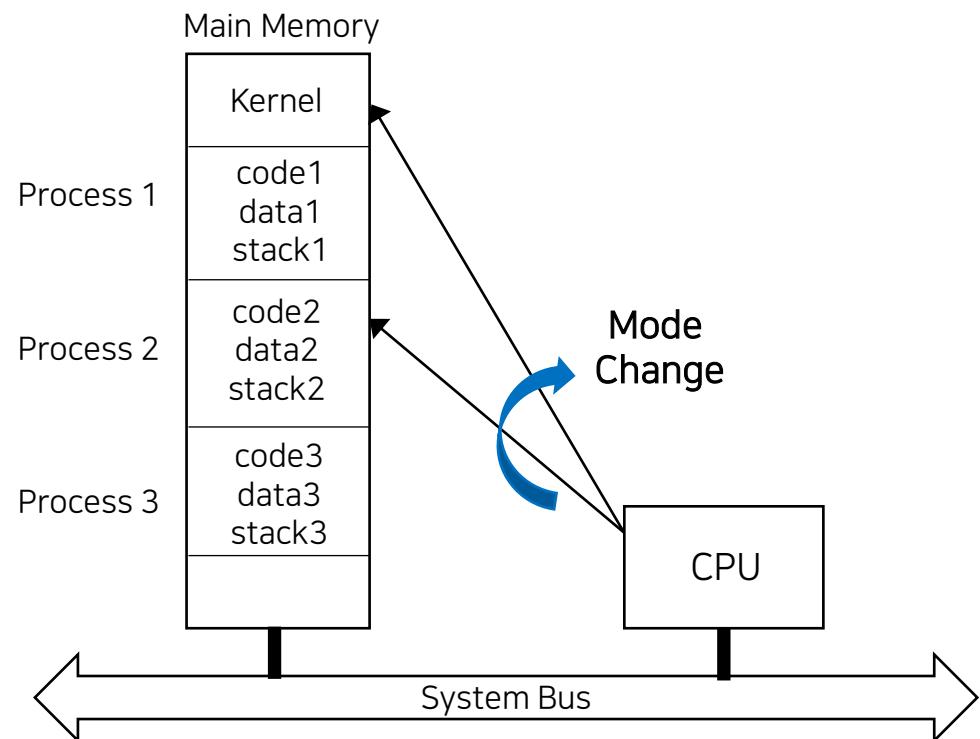
✓ Modes of Execution

- ① User mode →
 - » Less-privileged mode
 - » User programs execute in this mode
- ② System mode (= Kernel mode, Supervisor Mode) →
 - » More-privileged mode
 - » Kernel of the operating system executes in this mode

✓ Mode Change : User Mode ↔ System Mode



Mode Change vs. Context Switch



When to Switch a Process(Context Switch)

✓ Termination of a Process

- ◎ Error/exception occurred or normal termination of a process
- ◎ The process goes to terminated state and a new process is dispatched

✓ Blocking System Calls(Also called supervisor call)

- ◎ Ex: Some system calls such as I/O request, file open
- ◎ Page fault
 - » Memory address is in virtual memory so it must be brought into main memory

✓ Expiration of Time Slice

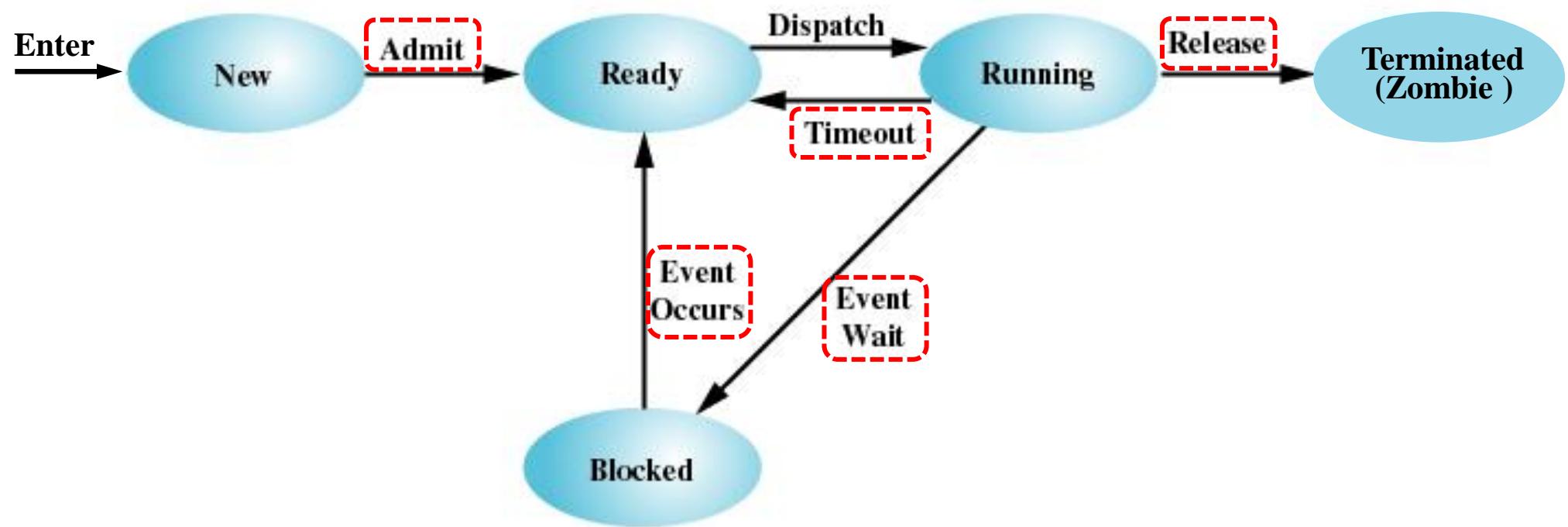
- ◎ Clock interrupt occurs and the running process has executed for the maximum allowable time slice

✓ I/O Interrupt

- ◎ Completion of an I/O changes blocked processes into ready state
- ◎ This 'ready' process MAY take the CPU

Recall
Process States

When a Context Switch May Occur



Five-State Process Model

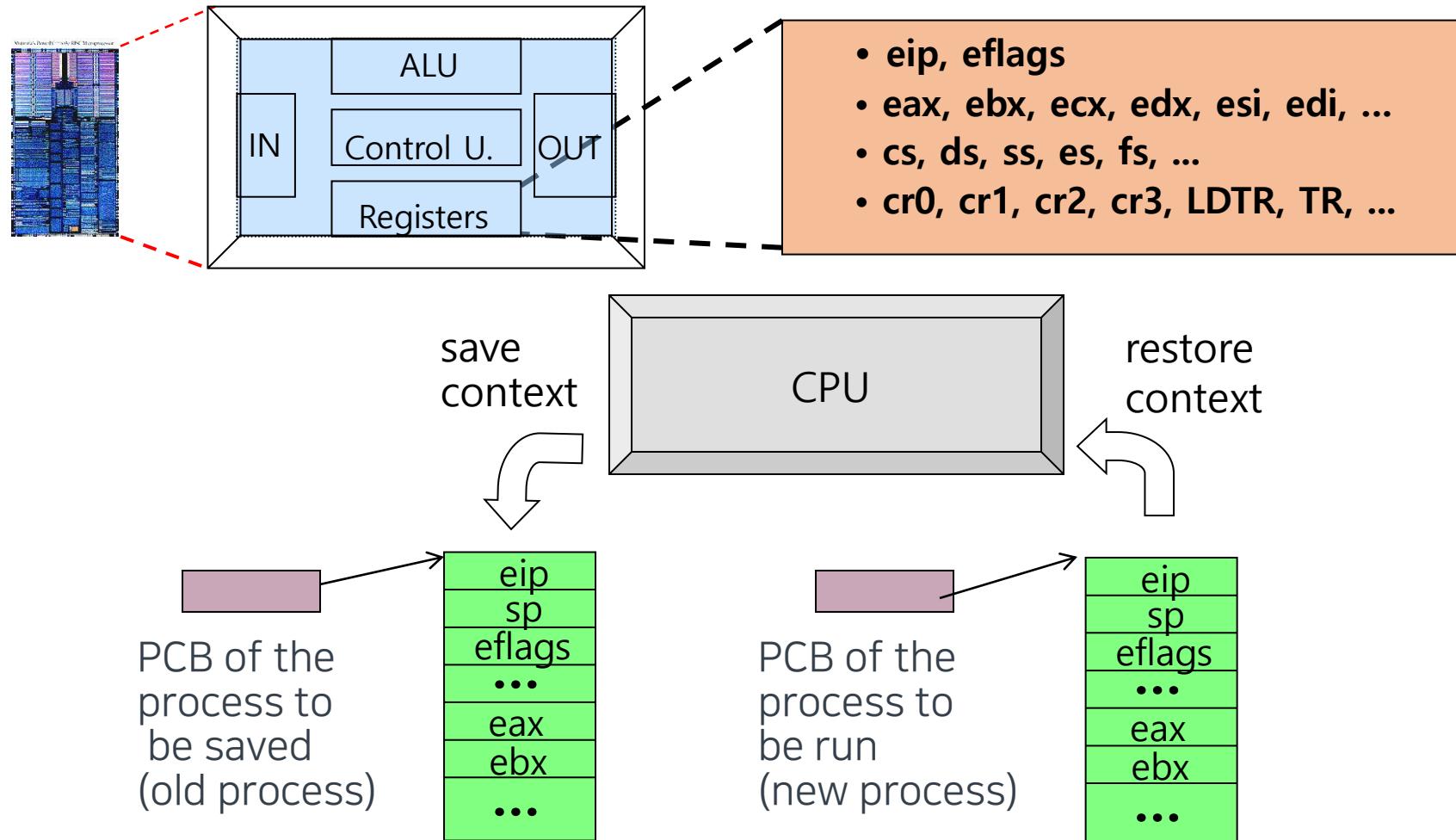
Steps of Context Switch

1. Save context of processor including program counter and other registers
2. Update the process control block of the process that is currently in the Running state
3. Move process control block to appropriate queue – ready or blocked or ready/suspend
4. Select another process for execution
5. Update the process control block of the process selected
6. Update memory-management data structures
7. Restore context of the selected process

Recall
PCB

Registers

✓ The 80x86 Architecture



Recall
PCB

Process Control Block(PCB)

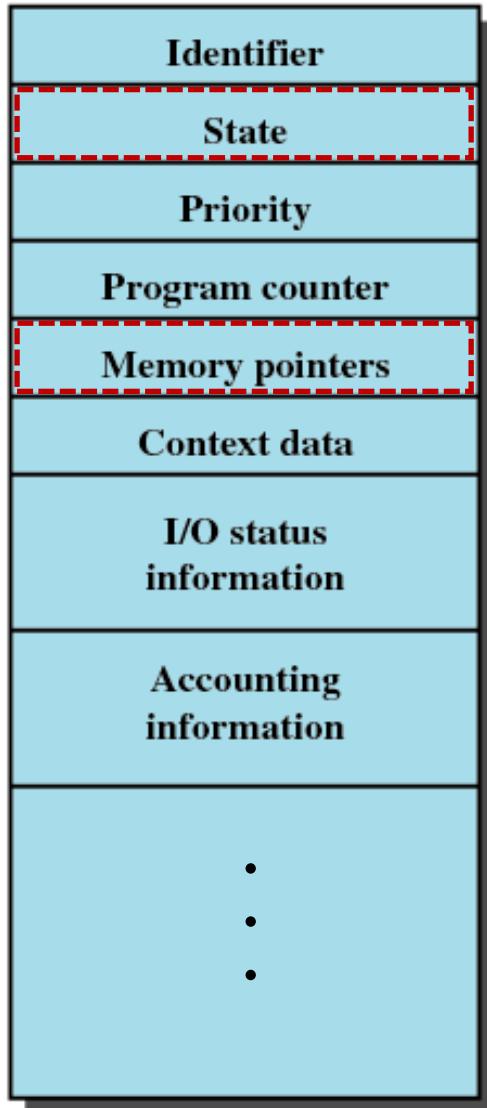
✓ Contains the Process Attributes

- ◎ Identifier
- ◎ State, Priority
- ◎ Program counter
- ◎ Memory pointers
- ◎ Address of memory context (program code, variables)
- ◎ I/O status information
- ◎ Accounting information

✓ Created and managed by the operating system

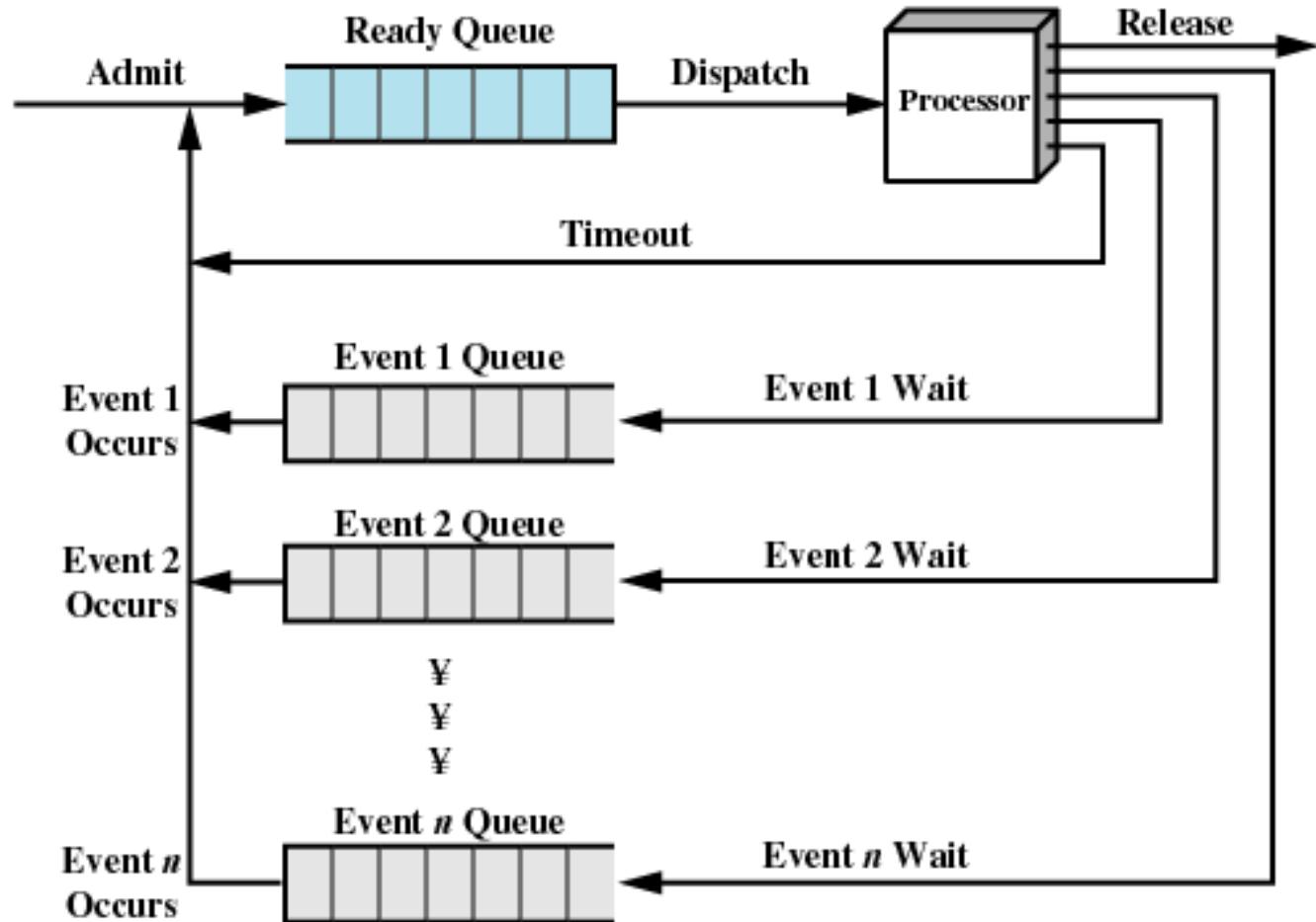
✓ One PCB per process

Simplified Process Control Block



Recall
Process States

Multiple Blocked Queues



Multiple Blocked Queues

Recall
PCB

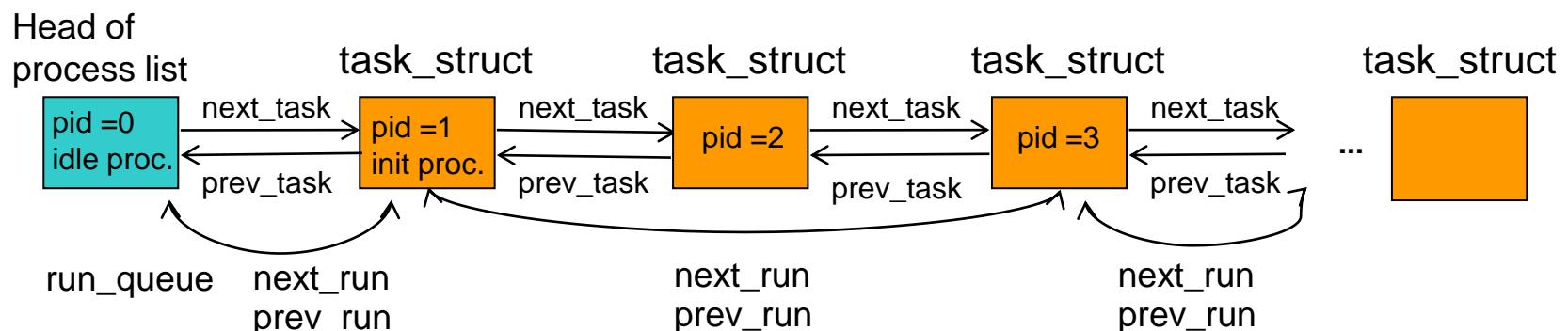
Process Control Block

✓ Process list

- ④ Connects all the process in the computer by double linked list
- ④ Ready queue is made on the process list by linking the processes in the process list whose state is ready state.

✓ Example : Linux process list

- ④ Head is the process descriptor of the idle(swapper) process → pid =0
- ④ Ready queue of Linux is called the run queue



Recall
Uni/Multiprogramming

Process Scheduling

Determine which program is next to be executed, in other words,

- ⌚ Which program will take a processor next
- ⌚ Which program the CPU will execute next

Also called CPU scheduling, job scheduling

Process scheduling is performed by the kernel function, scheduler

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제12강

Process Control :
Termination, Communication

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



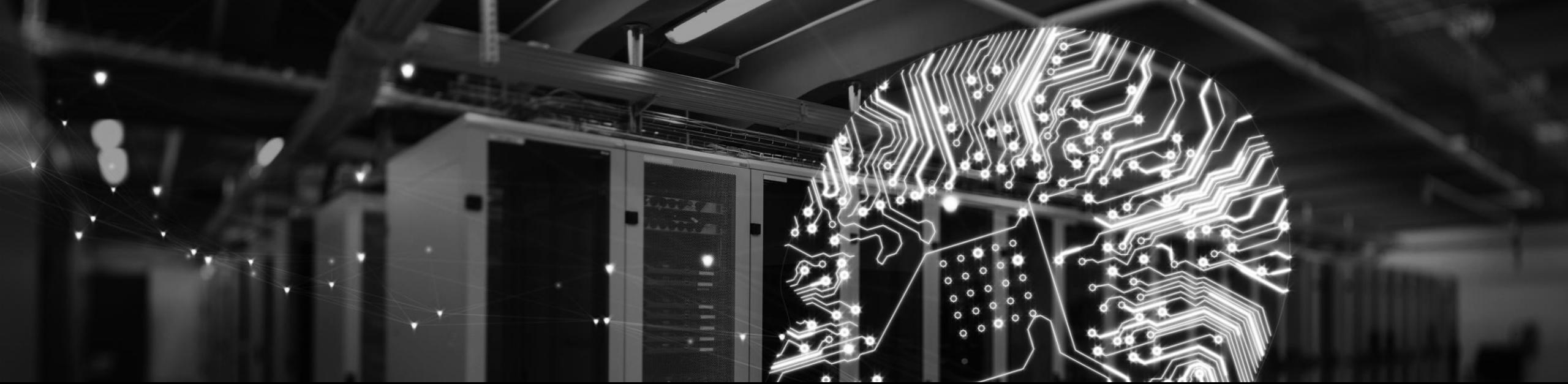
학습 목표

- 💬 프로세스 종료 절차를 이해한다.
- 💬 종료 시 자식 프로세스 처리 방법을 이해한다.
- 💬 프로세스 간 통신(Interprocess Communication)의 의미를 이해한다.
- 💬 프로세스 통신 방법을 이해한다.

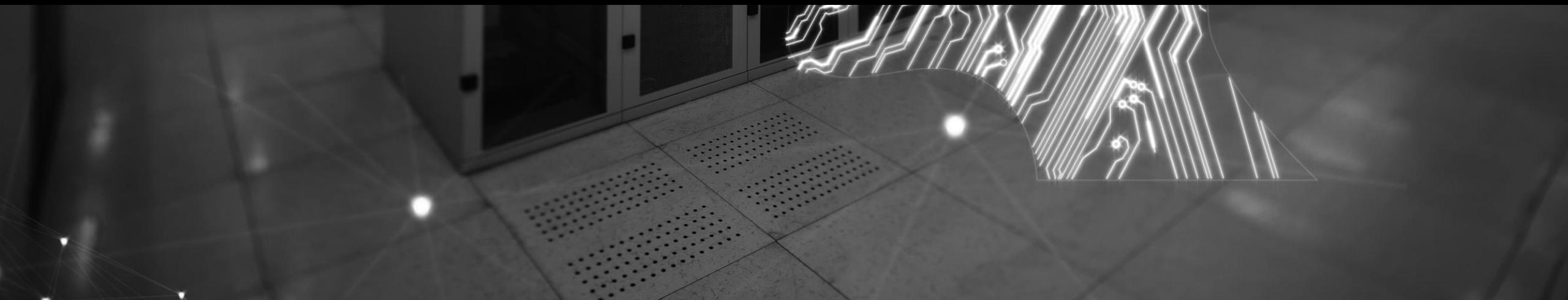
학습 내용

- ⌚ 프로세스 종료 절차
- ⌚ 종료 시 자식 프로세스 처리 방법
- ⌚ 프로세스 간 통신의 의미
- ⌚ 프로세스 통신 방법





Process Control : Termination, Communication



Process Termination

When

- ◎ The process invokes `exit()` system call, or
- ◎ The process receives a signal that it cannot handle, or
- ◎ An unrecoverable CPU exception has been raised in Kernel Mode while the kernel was running on behalf of the task
- ◎ Parent process kill the child because the child is no longer required

Process executes last statement and asks the operating system (via `exit()` system call) to do the followings.

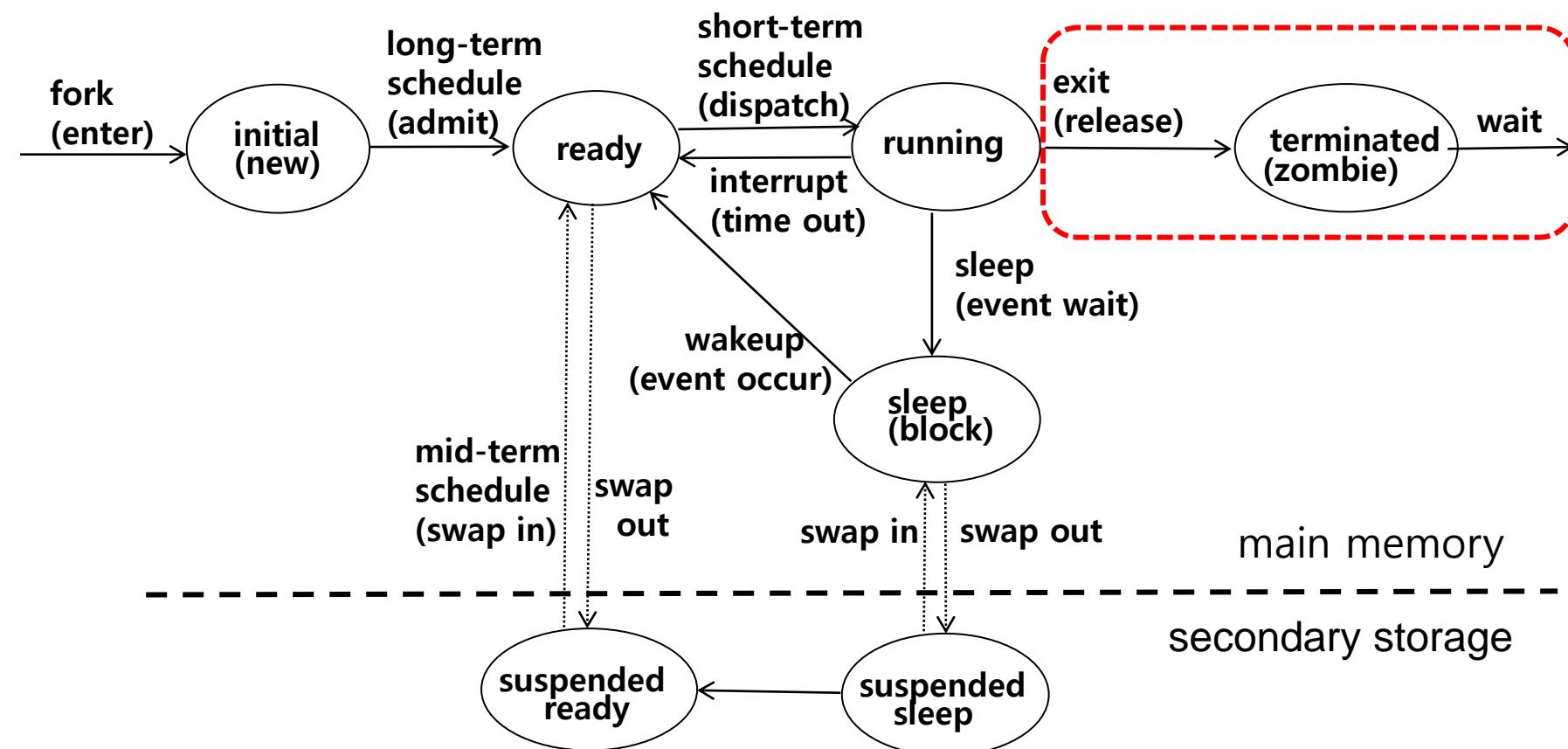
- ◎ Process' resources are released by operating system.
- ◎ Sends a signal (death-of-child signal) to the parent process
- ◎ Operating system puts this process in the terminated state.
- ◎ Wait in the terminated state until the parent process calls `wait()` system call to collect output data from child.
- ◎ Then, operating system releases the PCB of the ending process.

Process Termination

- ✓ If a terminating process has children processes, the *init* process becomes the new parent of the children processes in UNIX/Linux.

Recall
Process States

Process States



Recall
Process Context

User Part of Process Context

```
/* test.c */

int glob = 6;           Uninitialized global variable : bss area
char buffer[2048];
char buf[] = "a write to stdout\n";  (Initialized global
                                         Variable : data area)

int main(void)
{
    int var;
    pid_t pid;          Local Variable : Allocated in a stack area

    var = 88;
    printf("before fork\n");

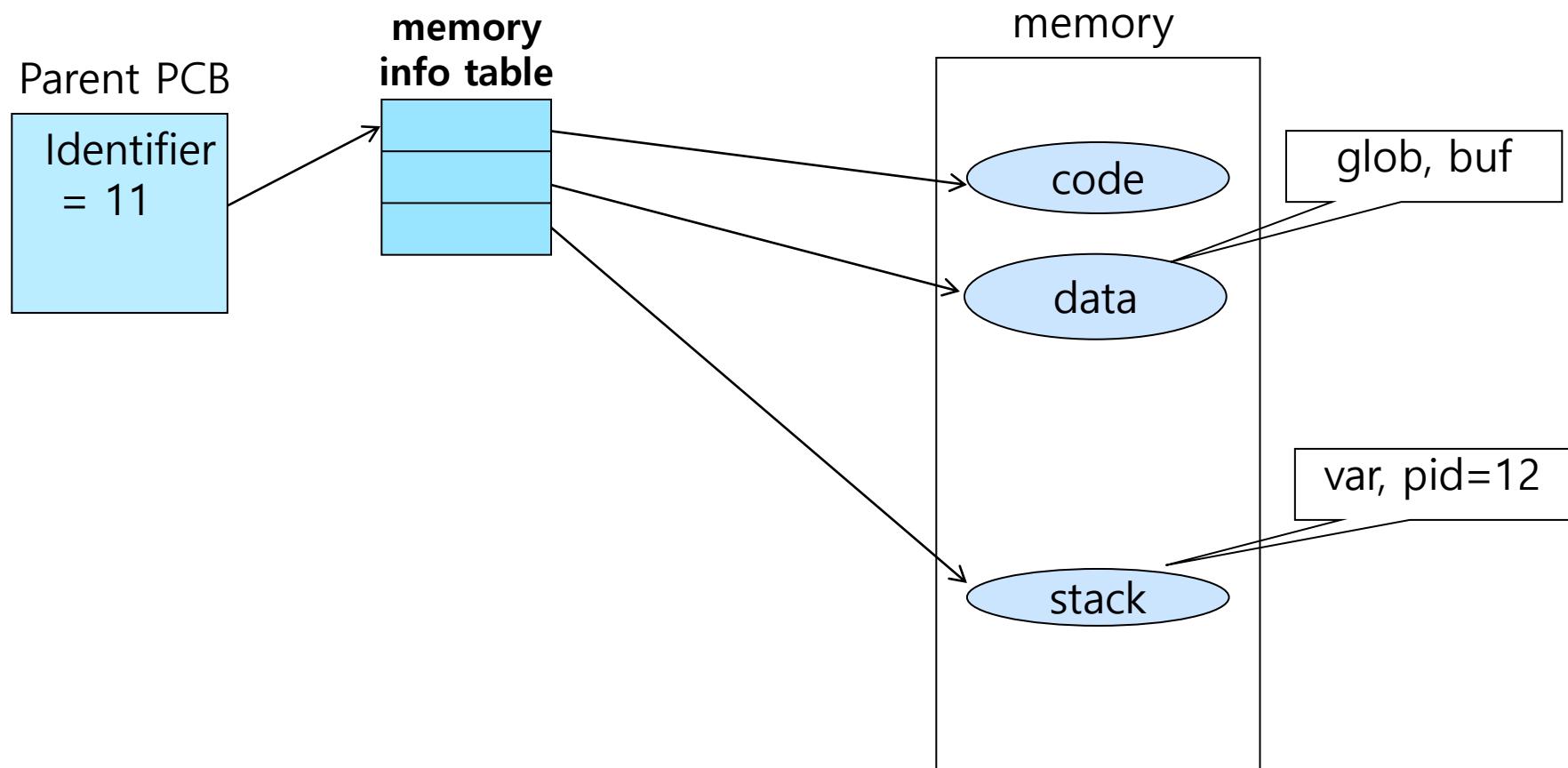
    if ((pid = fork()) == 0) {      /* child */
        glob++; var++;
    } else
        sleep(2);                  /* parent */

    printf("pid=%d,glob=%d,var=%d\n", getpid(), glob, var);
    exit (0);
}
```

Recall
Process Creation

Process Creation : Relation with Parent(1)

- ✓ How 'fork()' works : before fork (after run a.out)



Process Control Block(PCB)

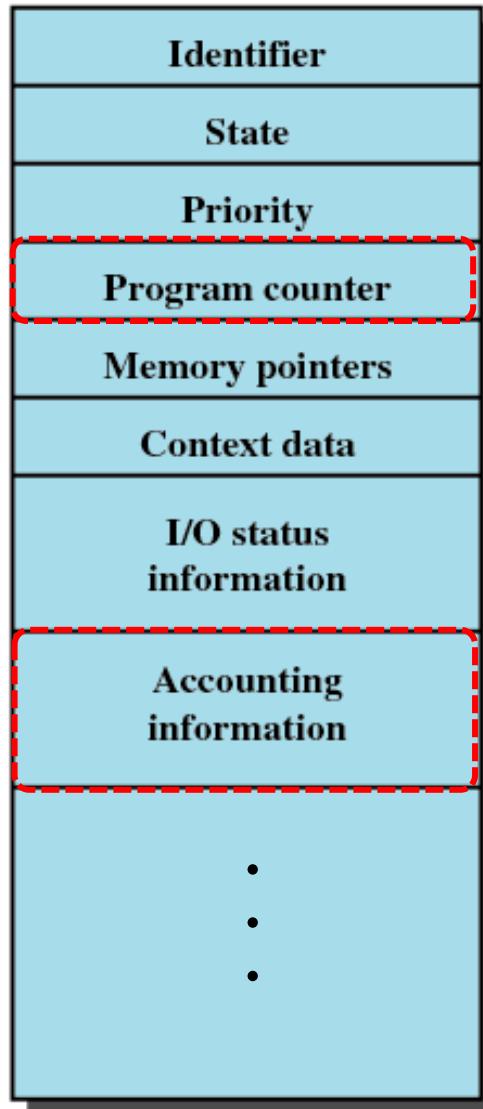
✓ Contains the process attributes

- ◎ Identifier
- ◎ State, Priority
- ◎ Program counter
- ◎ Memory pointers
- ◎ Address of memory context (program code, variables)
- ◎ I/O status information
- ◎ Accounting information

✓ Created and managed by the operating system

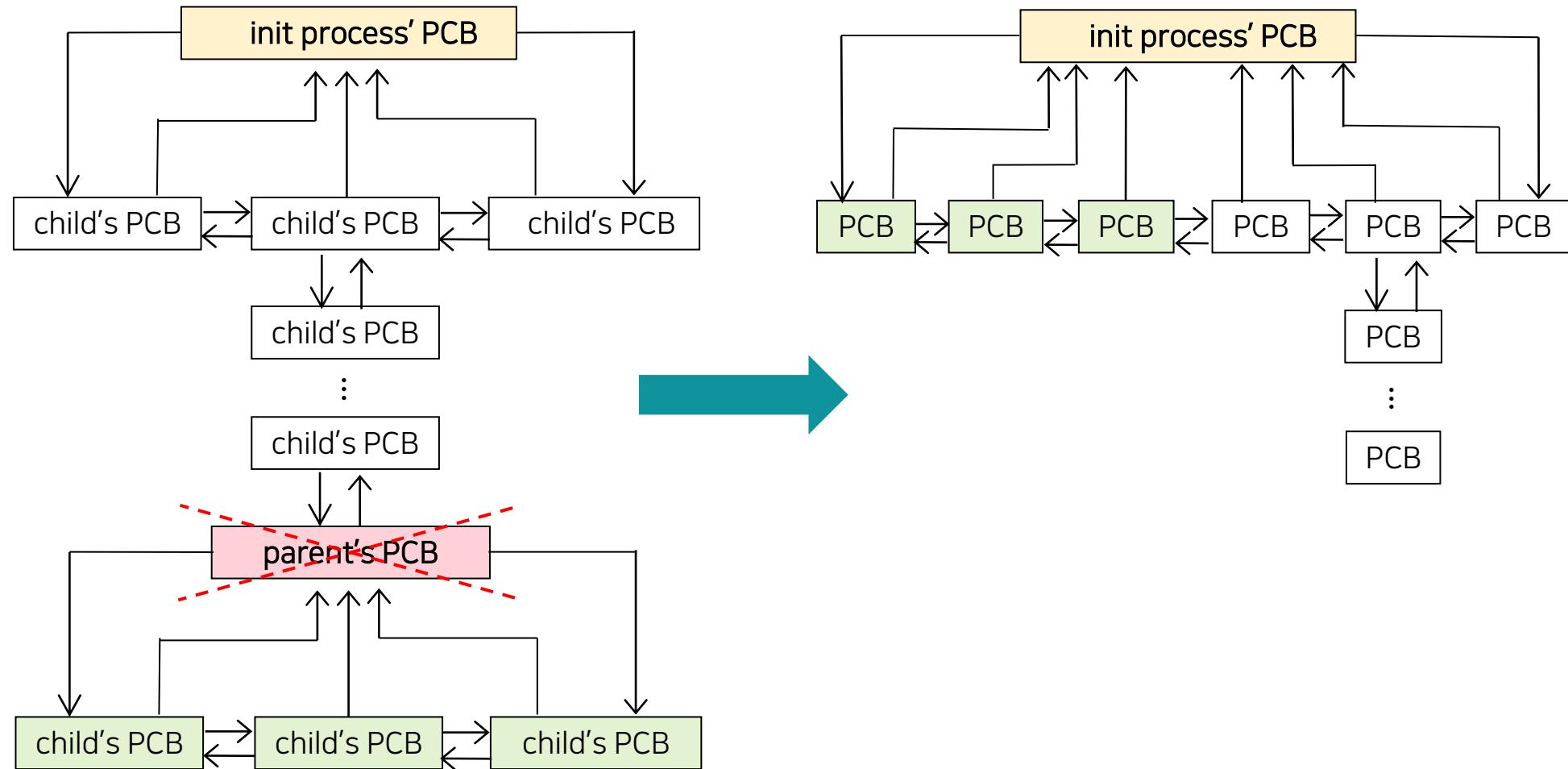
✓ One PCB per process

Zombie

Recall
PCB

Simplified Process Control Block

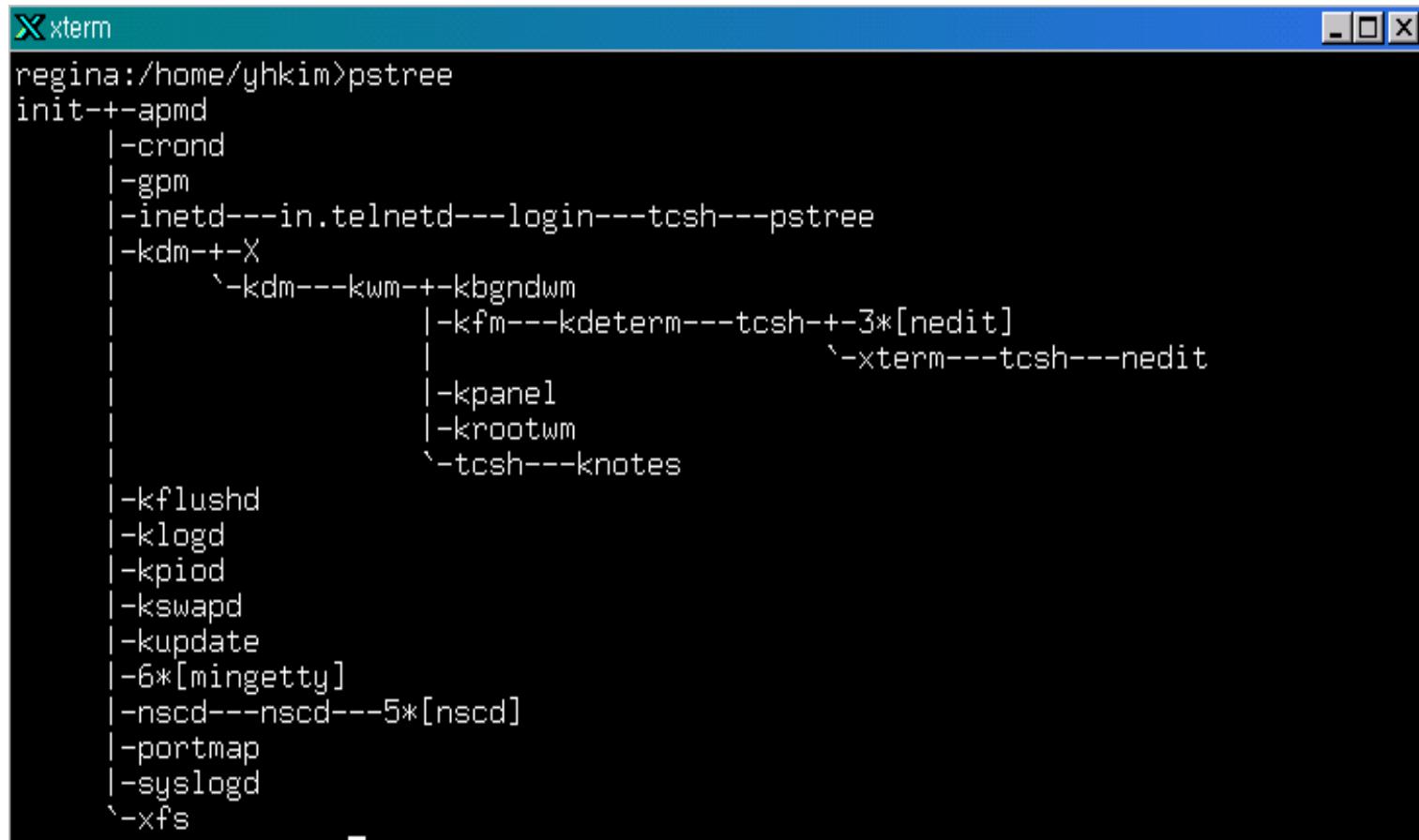
Children Processes of a Terminating Process



Recall
Process Creation

Process Creation: Relation with Parent

- ✓ Parent process create children processes, which, in turn create other processes, forming a tree of processes.

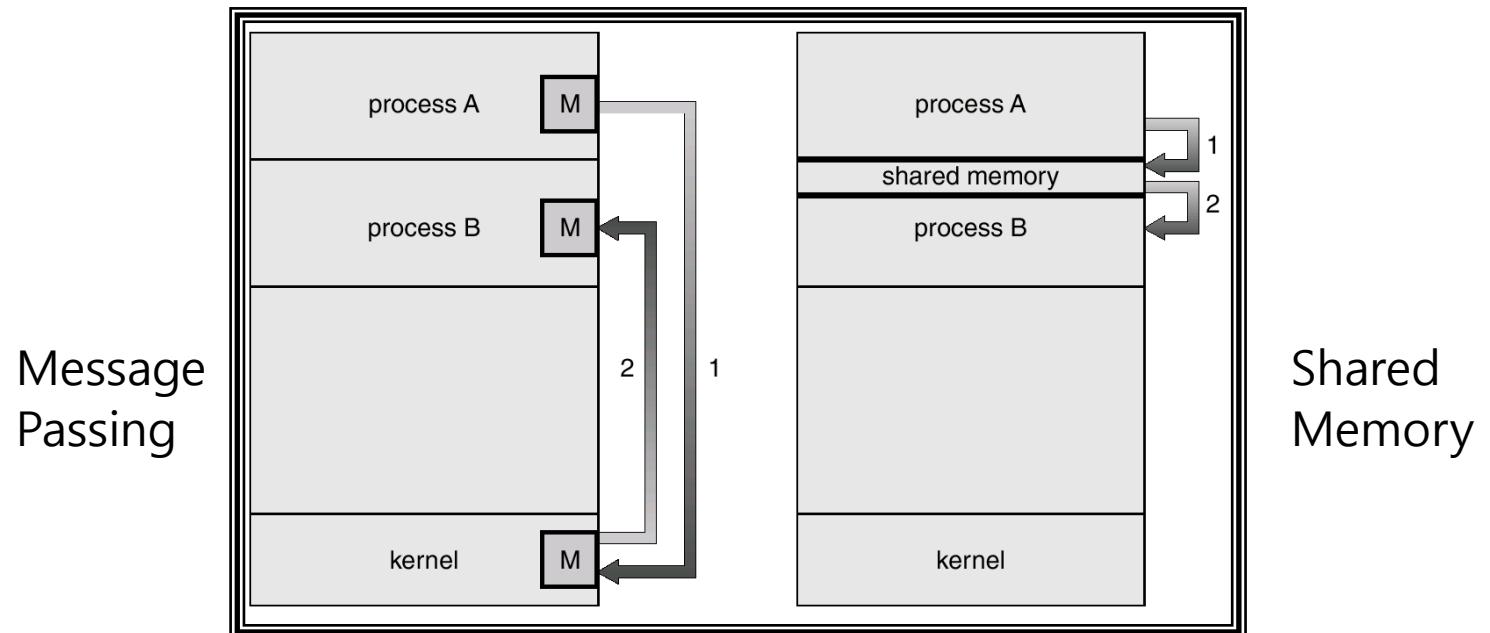


```
xterm
regina:/home/yhkim>pstree
init+-apmd
|-crond
|-gpm
|-inetd---in.telnetd---login---tcsh---pstree
|-kdm+-X
|  `--kdm---kwm---kbgrndwm
|     |-kfm---kdeterm---tcsh---3*[nedit]
|           |
|           `--xterm---tcsh---nedit
|     |-kpanel
|     |-krootwm
|     `--tcsh---knotes
|-kflushd
|-klogd
|-kpiod
|-kswapd
|-kupdate
|-6*[mingetty]
|-nscd---nscd---5*[nscd]
|-portmap
|-syslogd
`-xfs
```

Inter-Process Communication (IPC)

- ✓ **Independent process** : not affect or be affected by another process
- ✓ **Cooperating process** : can affect or be affected by the execution of another process. → distributed system
- ✓ **Interprocess Communication(IPC) for the information sharing**

◎ Message passing, shared memory, Remote Procedure Call



Message Passing

```
int msqqid;

msgqid = msgget(key, IPC_CREAT); /* create a message
queue with the key */

pid_parent = getpid();
pid_child = fork();
if(pid_child != 0) // in parent process
{
    struct message sndmsg;
    send(msqid, &msg); /* parent sends a message
    to the queue */
}
```

Interprocess Communication

✓ Advantages of process cooperation

- ◎ Information sharing, computation speed-up, reliability/availability

✓ Concurrent execution of cooperating processes requires a synchronization mechanism.

- ◎ Signal, Semaphore, Lock : types of Interprocess Communication

✓ Signal : a mechanism for a process to inform another process of the occurrence of an event

- ◎ SIGKILL, SIGINT, SIGBUS, SIGUSR1, (64 signals in Linux)
- ◎ When a signal is received, one of 4 actions occurs depending on the signal.
 1. The process may be terminated or
 2. The process may be stopped, or
 3. The signal may be ignored or
 4. The user level catch function may be executed

Comparisons of Signal, Interrupt

- ✓ **Signal** : a mechanism for a process to inform another process of the occurrence of an event
- ✓ **Interrupt** : a mechanism for the OS to inform a process of the occurrence of an event
- ✓ **System call** : a mechanism for a process to inform the OS of the occurrence of an event (requesting a service)

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제13강

Difference between a Process
and a Threads

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



학습 목표

- ☞ 쓰레드(Thread)의 정의를 이해한다.
- ☞ 프로세스와 쓰레드의 차이를 이해한다.
- ☞ 쓰레드의 장점을 이해한다.

학습 내용

- ☞ 쓰레드(Thread)의 정의
- ☞ 프로세스와 쓰레드의 차이
- ☞ 쓰레드의 장점





Process Control : Termination, Communication

Processes and Threads

✓ Characteristics of a process

- ① Resource ownership - process includes a virtual address space to hold the process image
- ② Execution and scheduling - follows an execution path that may be interleaved with other processes
- ③ These two characteristics are treated independently by the operating system

✓ Definition of a thread

- ④ A unit of execution of in a process (by a group of instructions)
- ⑤ Has an execution state (running, ready, etc.)

✓ CPU dispatching is referred to as a thread or lightweight process.

✓ Resource of ownership is referred to as a process or task.

Resources of Process and Thread

✓ Process

- ◎ Have a virtual address space which holds the process image
- ◎ Protected access to processors, other processes, files, I/O resources

✓ Process consists of

- ◎ Code
- ◎ Data
- ◎ Stack
- ◎ Memory for registers context (program counter, registers etc.)

✓ Thread consists of

- ◎ Stack
- ◎ Memory for registers context (program counter, registers etc.) : thread control block

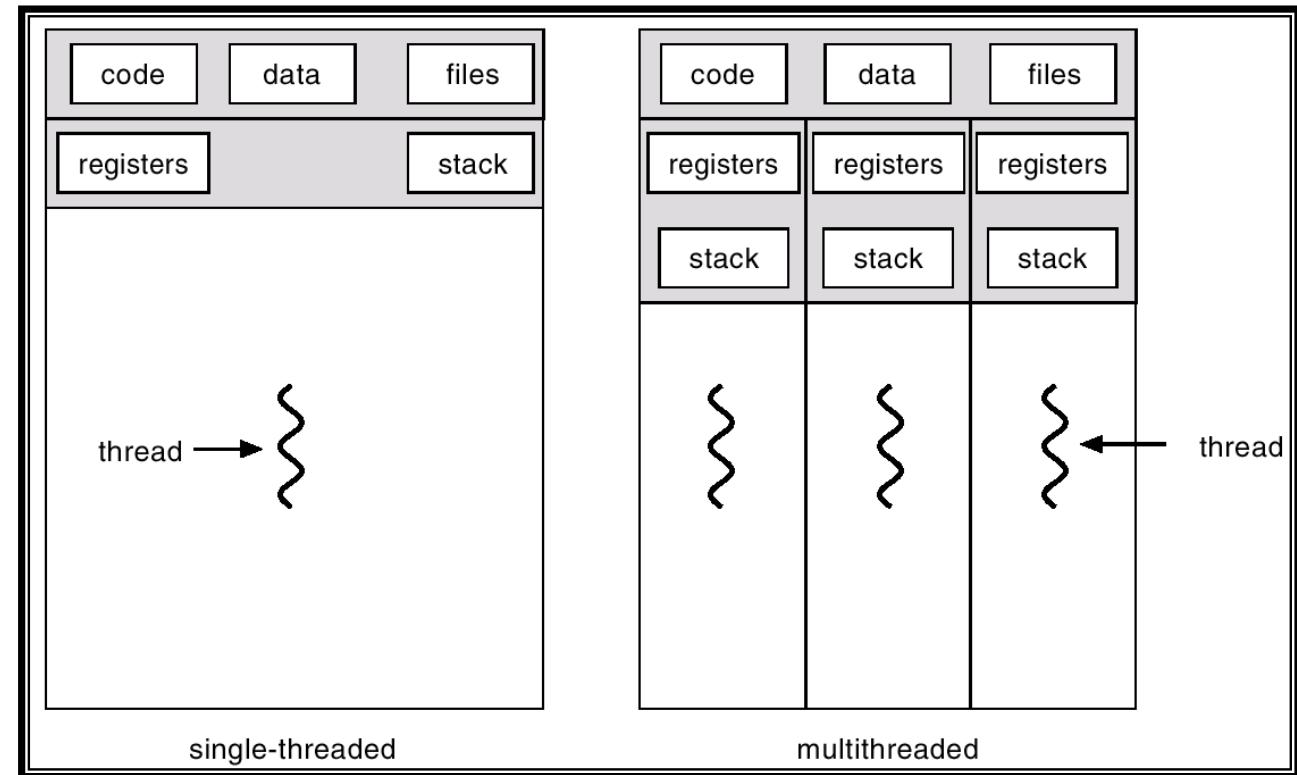
✓ Thread

- ◎ Access to the memory and resources of its process
 - » All threads of a process share this
- ◎ Some thread supports per-thread static storage for local variables

Resources of Process and Thread

✓ Process is divided into threads that can run concurrently

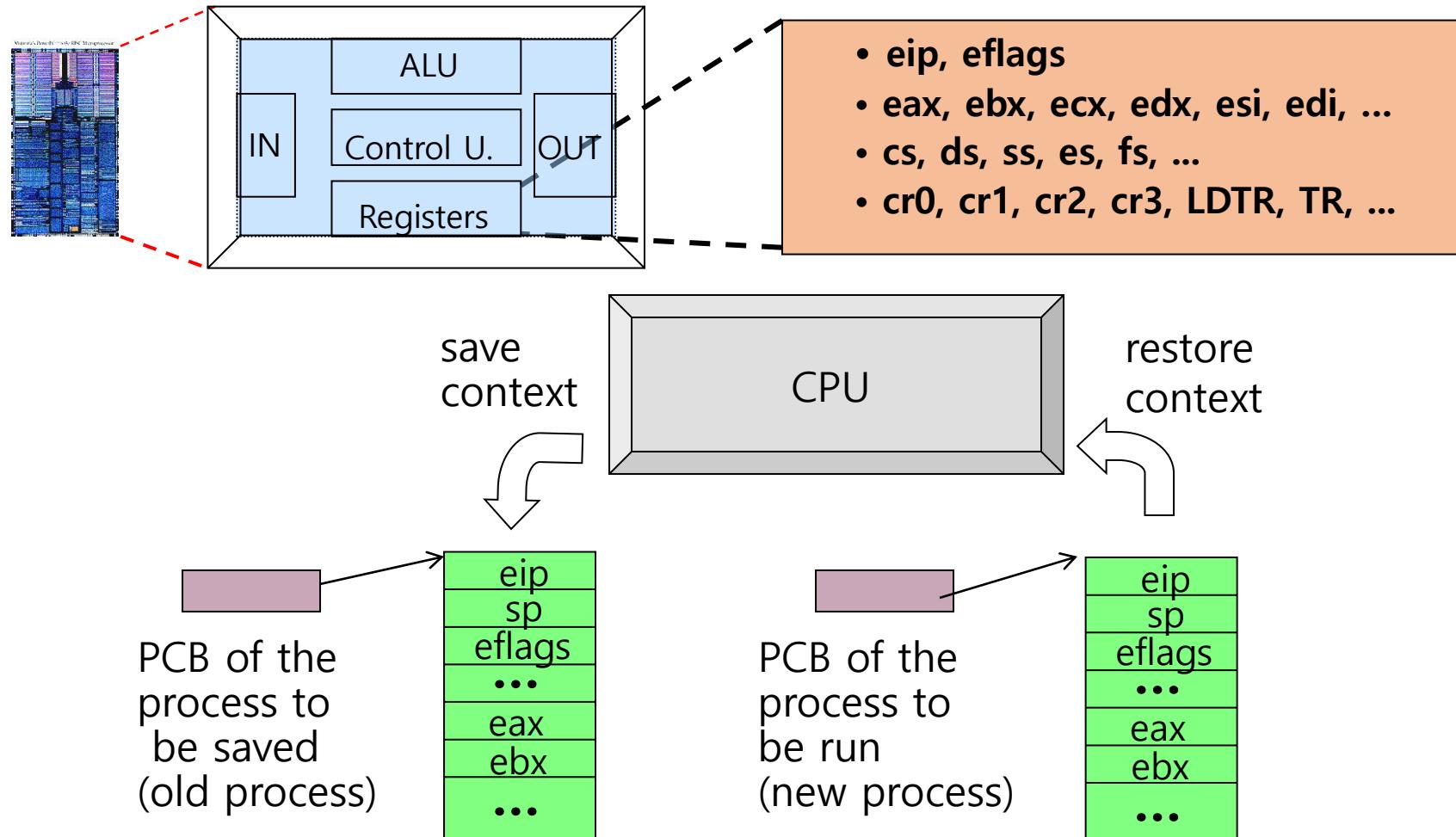
- ◎ Thread
 - » Dispatchable unit of work
 - » Executes sequentially and is interruptible
- ◎ Process is a collection of one or more threads



Recall
PCB

Registers

✓ The 80x86 architecture



Benefits of Threads

- ✓ Takes less time to create a new thread than a process
- ✓ Less time to terminate a thread than a process
- ✓ Less time to switch between two threads within the same process
- ✓ Less time to communicate between threads in the same process than to communicate between processes
- ✓ Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel
 - ◎ Inter-process communication : Invoking the kernel takes time (big overhead)

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제14강

Use of Multithreads

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



학습 목표

- 💬 멀티쓰레드(Multi-thread) 지원 운영체제의 종류를 파악한다.
- 💬 멀티쓰레드 이용 시 장점을 이해한다.
- 💬 커널 레벨 쓰레드, 유저 레벨 쓰레드 차이를 이해한다.

학습 내용

- ⌚ 멀티쓰레드(Multi-thread) 지원 운영체제의 종류
- ⌚ 멀티쓰레드 이용 시 장점
- ⌚ 커널 레벨 쓰레드, 유저 레벨 쓰레드 차이





Use of Multithreads



Multithreading

Operating system supports one or more threads of execution within a single process

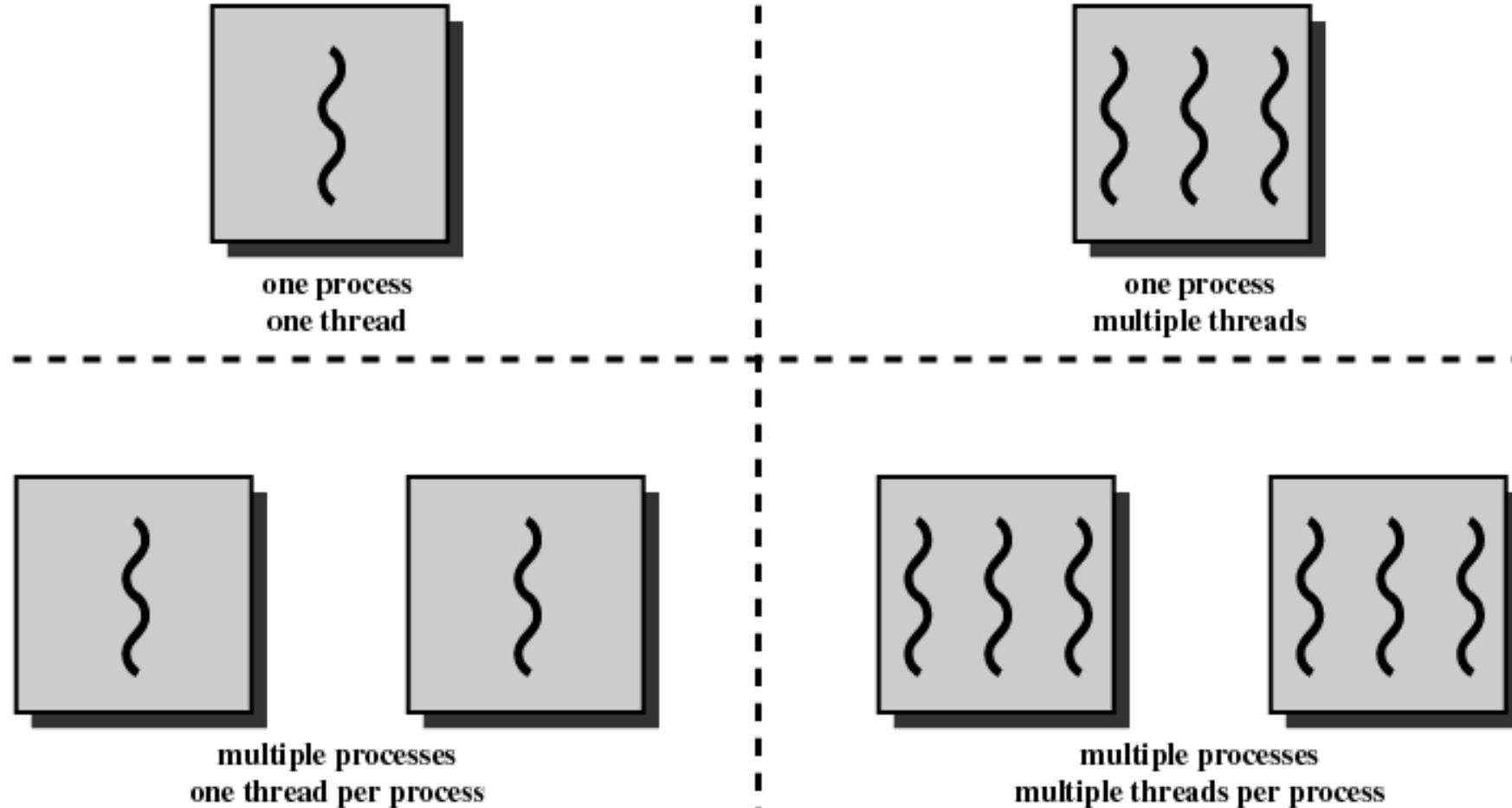
Single thread OS

- ◎ MS-DOS
- ◎ Original UNIX

Multiple threads OS

- ◎ Windows, Solaris (a modern UNIX), Linux, Mach, OS/2, ...

Multithreading



{ = instruction trace

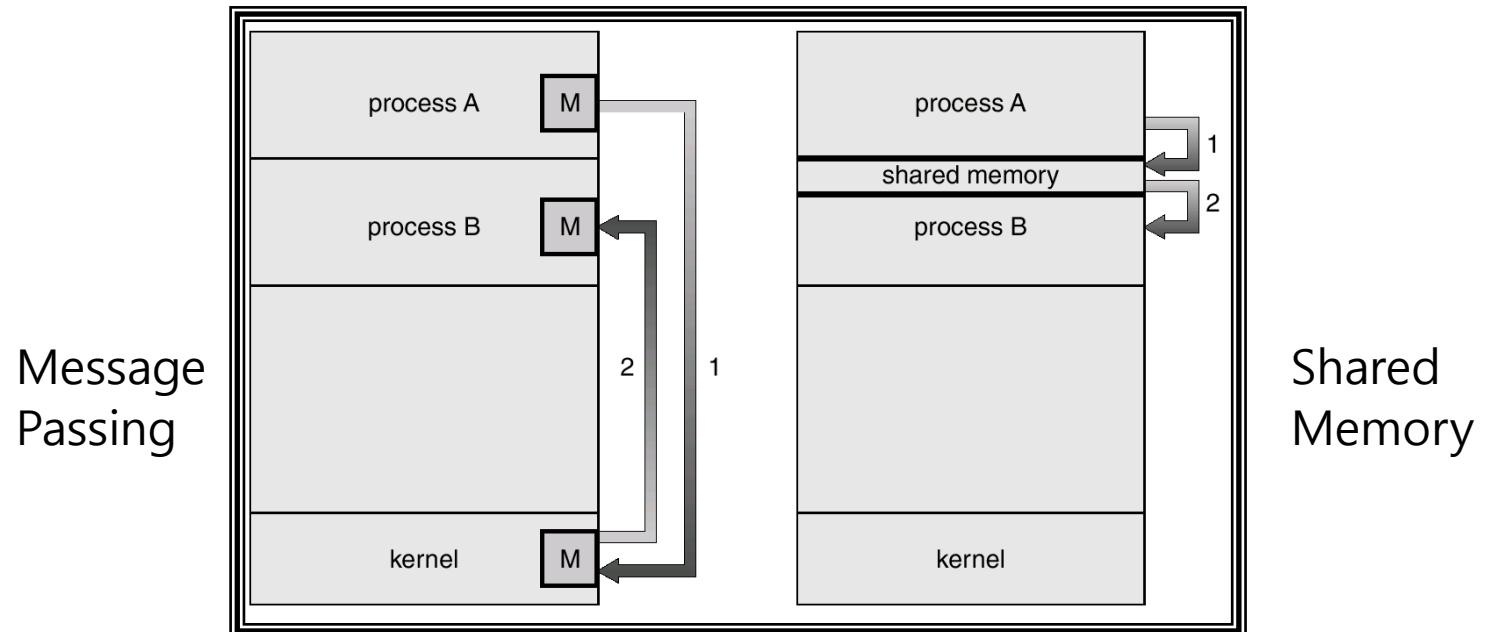
Threads and Processes [ANDE97]

Recall
Process Control :
Termination,
Communication

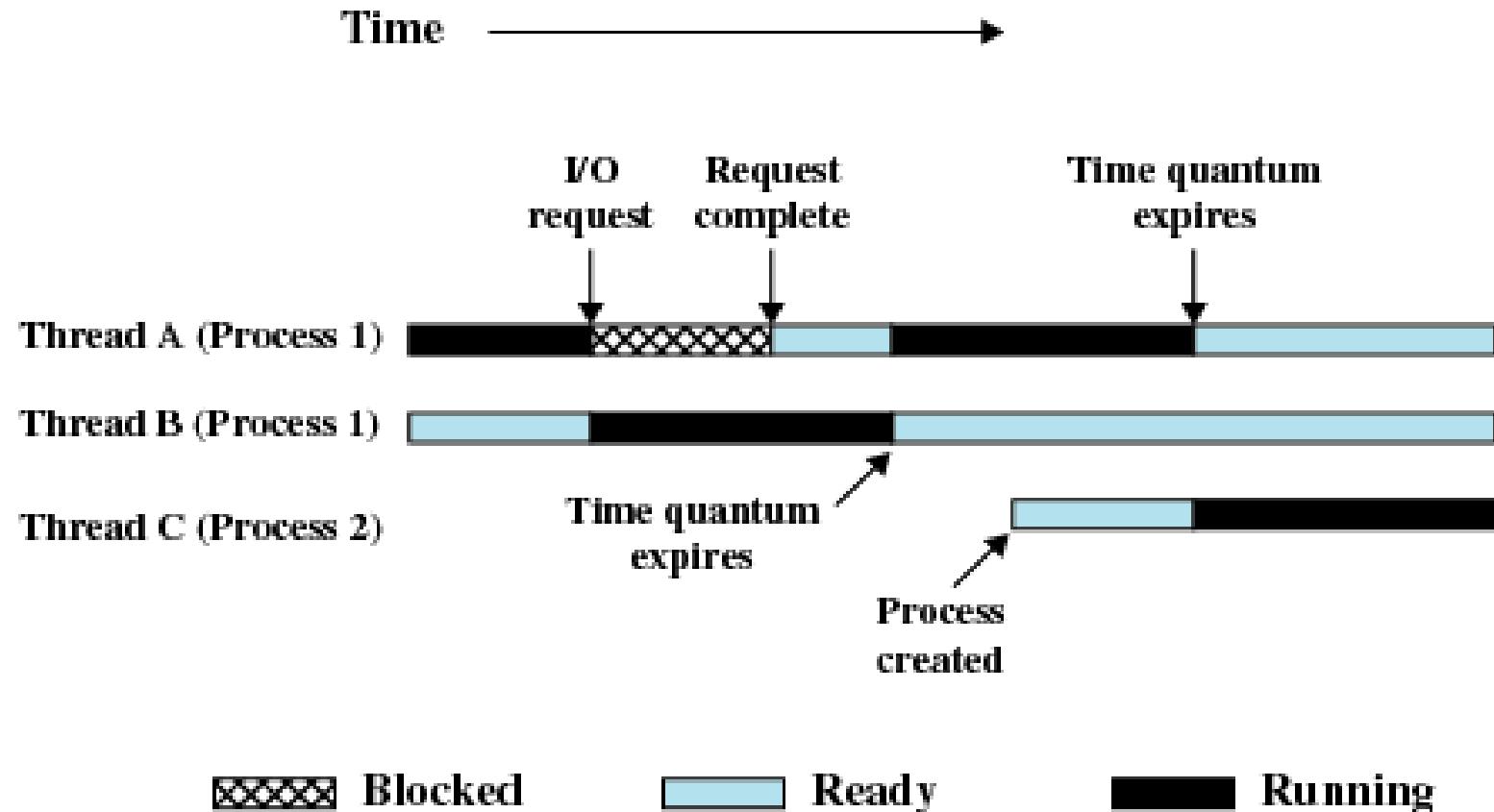
Inter-Process Communication (IPC)

- ✓ **Independent process** : not affect or be affected by another process
- ✓ **Cooperating process** : can affect or be affected by the execution of another process. → distributed system
- ✓ **Interprocess Communication(IPC) for the information sharing**

◎ Message passing, shared memory, Remote Procedure Call



Multithreading



Multithreading Example on a Uniprocessor

Recall
Timesharing

Multiprogramming vs. Multiprocessing

Time →



a. Interleaving (Multiprogramming, one processor) Also known as Concurrent programming



b. Interleaving and overlapping (Multiprogramming, one processor) Also known as Parallel programming

Processing Concurrent Works

Non-concurrent programming : one process, single thread

```
main( ) {  
    int data, transformed_data;  
    ...  
    input_handling(data);  
    transformed_data = transforming(data);  
    output_handling(transformed_data);  
    ...  
}  
  
input_handling(int a) { . . . }  
transforming(int a) { . . . }  
output_handling(int a) { . . . }
```

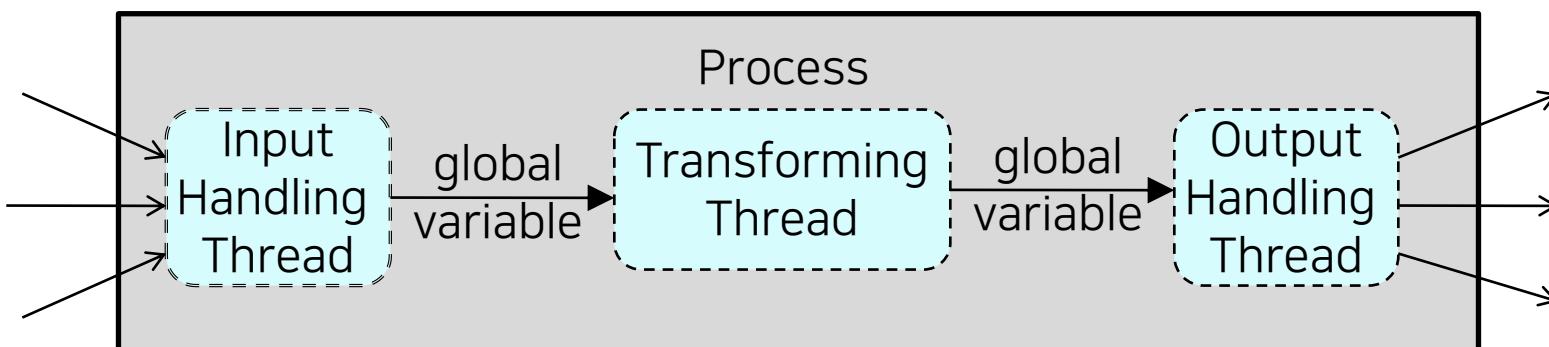
Processing Concurrent Works

✓ Processing of concurrent work may be done by

- ① Multiple processes

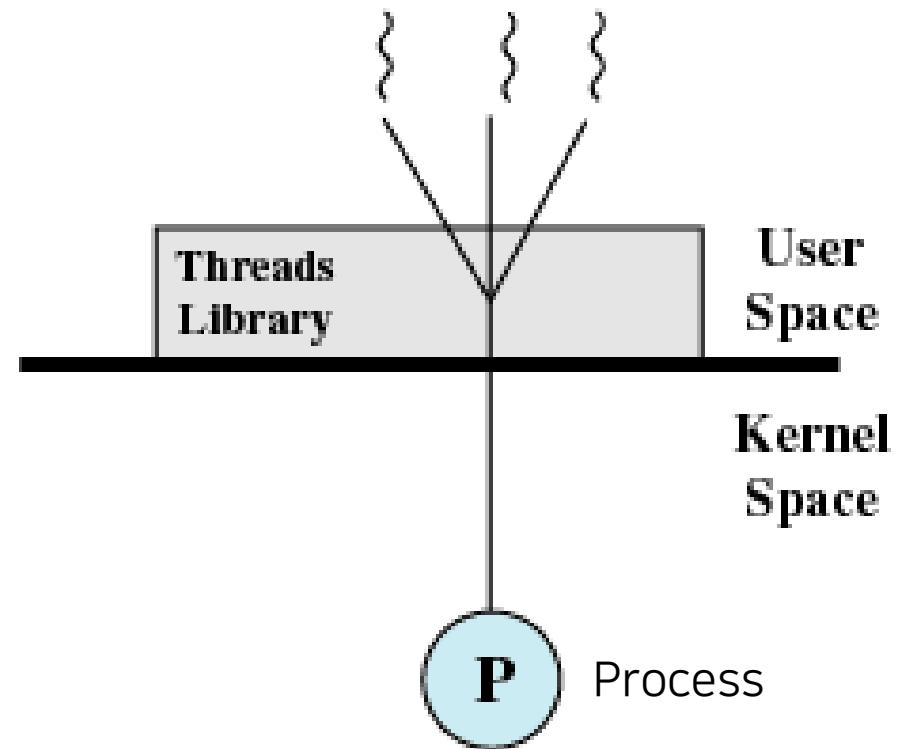


- ② Multiple threads



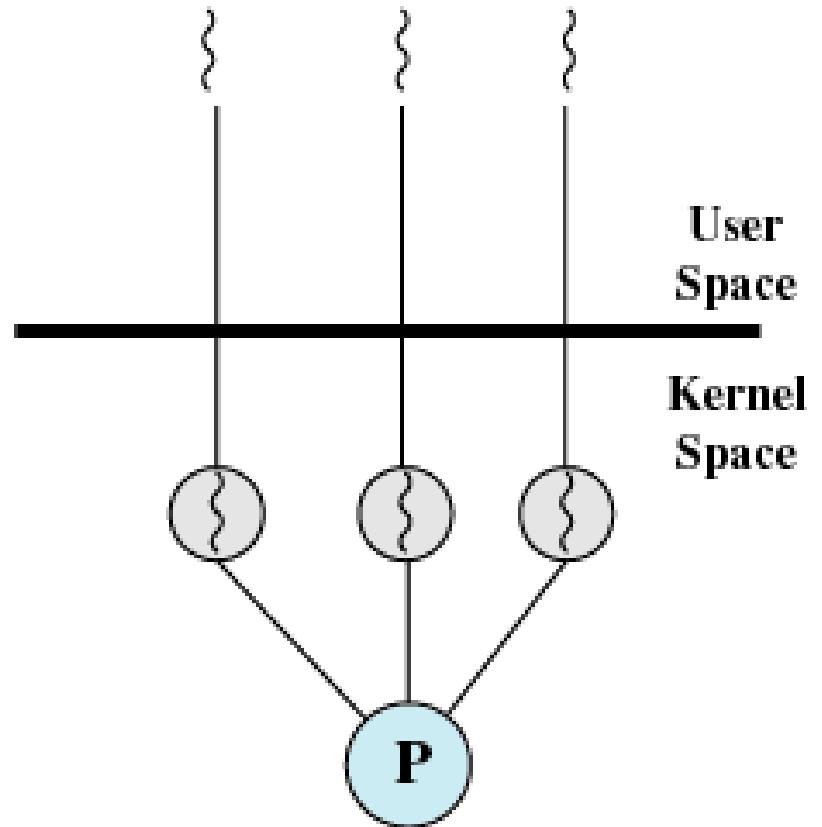
User-Level Threads

- ✓ All thread management is done by the application (thread library)
- ✓ The kernel is not aware of the existence of threads



Kernel-Level Threads

- ✓ Linux, Windows is an example of this approach
- ✓ Kernel maintains context information for the process and the threads
- ✓ Scheduling is done on a thread basis



Pure Kernel-level

Advantages vs. Disadvantages

Advantages of User-Level Thread

- ① Less overhead than Kernel-Level Thread
- ② Scheduling algorithm of User-Level Threads can be tailored to the application without disturbing the underlying OS scheduling
- ③ User-Level Threads can run on any operating systems.

Disadvantages of User-Level Threads

- ④ When a User-Level Thread executes a blocking system call, all the threads in the process are blocked.
- ⑤ A multithreaded application cannot take advantage of multiprocessing (multiple processor system)

VAX Running UNIX-Like Operating System

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

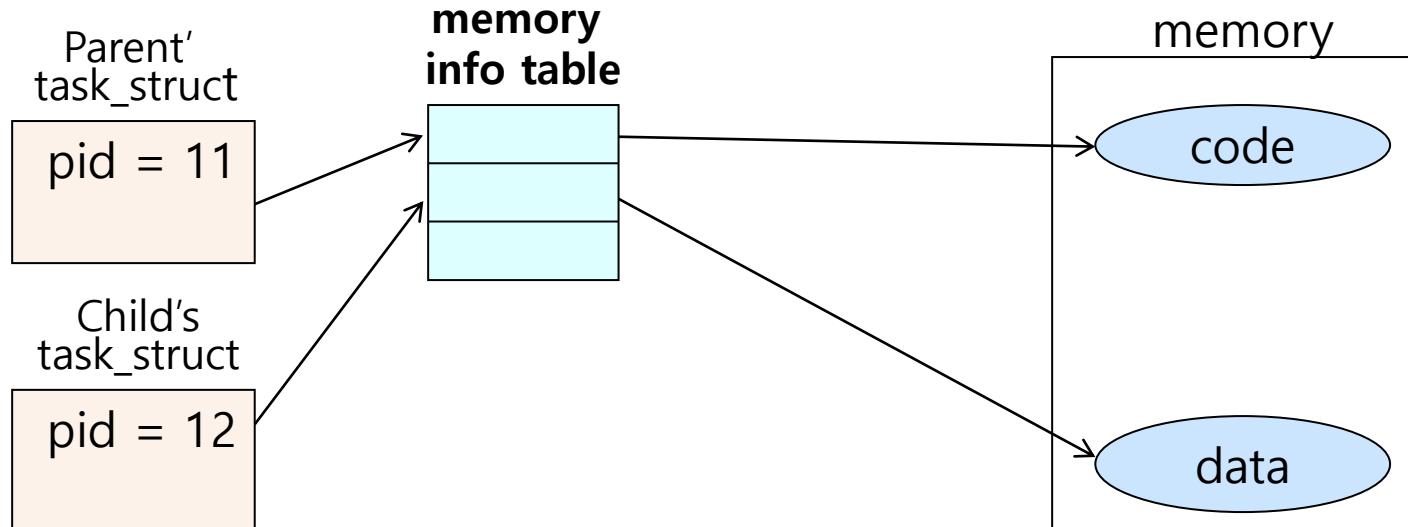
Thread and Process Operation Latencies(μs) [ANDE92]

심화

Linux Thread

✓ Linux's kernel-level thread is called the lightweight process

- ① Created by clone() system call
- ② OS allocates a task_struct for the created thread
- ③ The lightweight process shares its parent process's resource



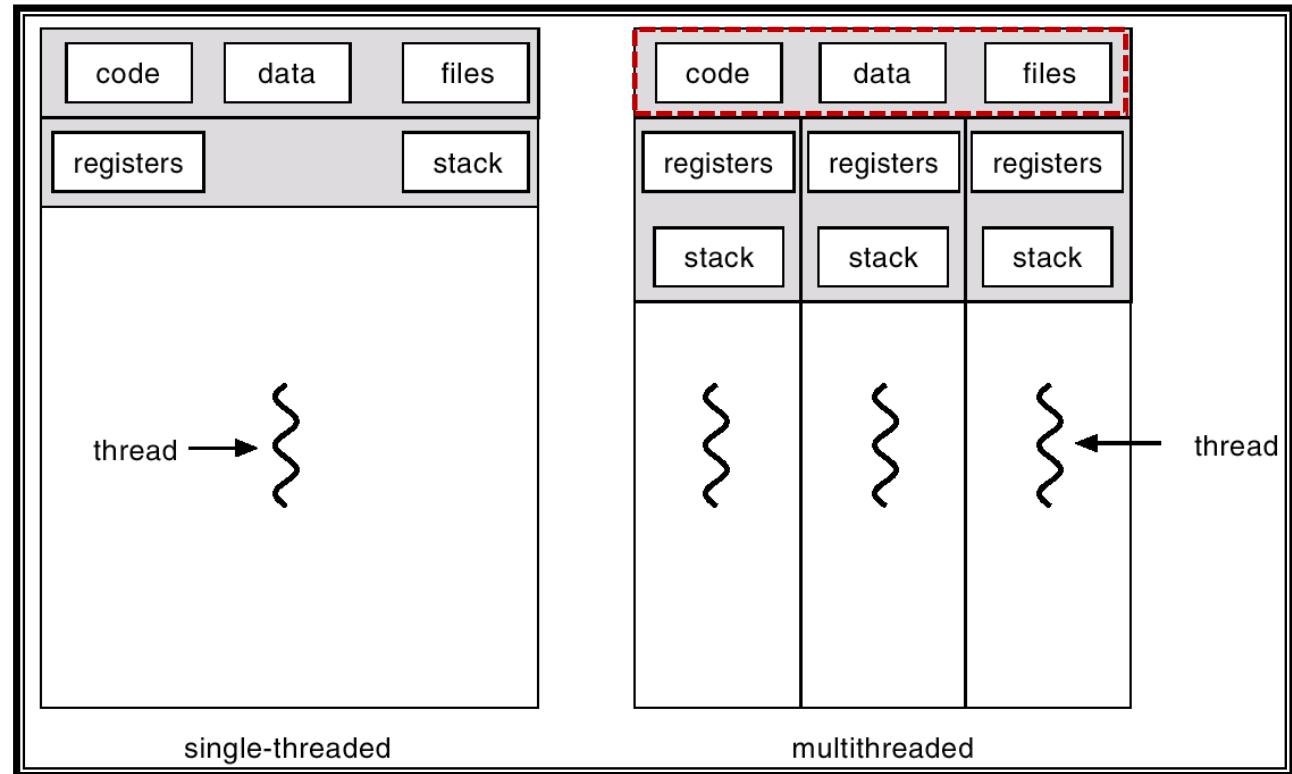
✓ The lightweight process is independently scheduled by the OS, like the parent process.

Recall
Difference of
Process & Thread

Resources of Process and Thread

✓ Process is divided into threads that can run concurrently

- ◎ Thread
 - » Dispatchable unit of work
 - » Executes sequentially and is interruptible
- ◎ Process is a collection of one or more threads



Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제15강

What is the Concurrency Control?

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



학습 목표

- 💬 Concurrency Control에 관련된 기본 용어들을 이해한다.
- 💬 Concurrency Control의 의미와 필요성을 이해한다.
- 💬 레이스 컨디션(Race Condition)이 발생하는 상황을 이해한다.

학습 내용

- ⌚ Concurrency Control 관련 기본 용어
- ⌚ Concurrency Control의 의미와 필요성
- ⌚ 레이스 컨디션이 발생하는 상황





What is the Concurrency Control?



Principles of Concurrency

✓ Multiple processes in

- ◎ Multiprogramming
- ◎ Multiprocessing
- ◎ Distributed processing

✓ Concurrency

- ◎ Communication among processes
- ◎ Sharing and competing for resources
- ◎ Synchronization of multiple activities in processes
- ◎ Allocation of processor time to processes

Operating System Concerns

Keep track of various processes

Allocate and deallocate resources

- ◎ Processor time
- ◎ Memory
- ◎ Files
- ◎ I/O devices

Protect data and resources

Output of process must be independent of the speed of execution of other concurrent processes

Some Key Terms in Concurrency

Race condition

◎ A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

Mutual exclusion

◎ The requirement that when one process accesses shared resources, no other process may access any of those shared resources.

Critical section

◎ A section of code within a process that requires access to shared resources and that may not be executed while another process is in a corresponding section of code.

◎ Only one program at a time is allowed in its critical section

If critical sections are not executed mutually exclusively, the race condition may occur.

Some Key Terms in Concurrency

✓ Starvation

- ④ A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

✓ Deadlock

- ④ A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

- ④ Starvation for an infinite duration

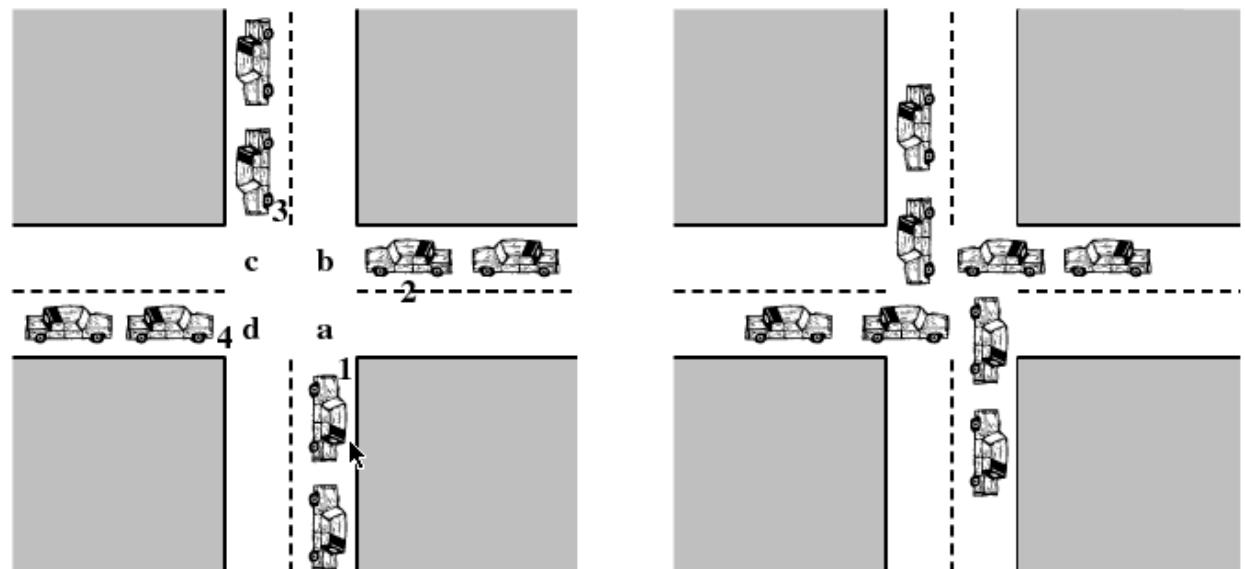


Illustration of Deadlock

심화

Difficulties of Concurrency

- ✓ Sharing of global resources
- ✓ Operating system managing the allocation of resources optimally
- ✓ Difficult to locate programming errors
- ✓ A simple example

```
void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

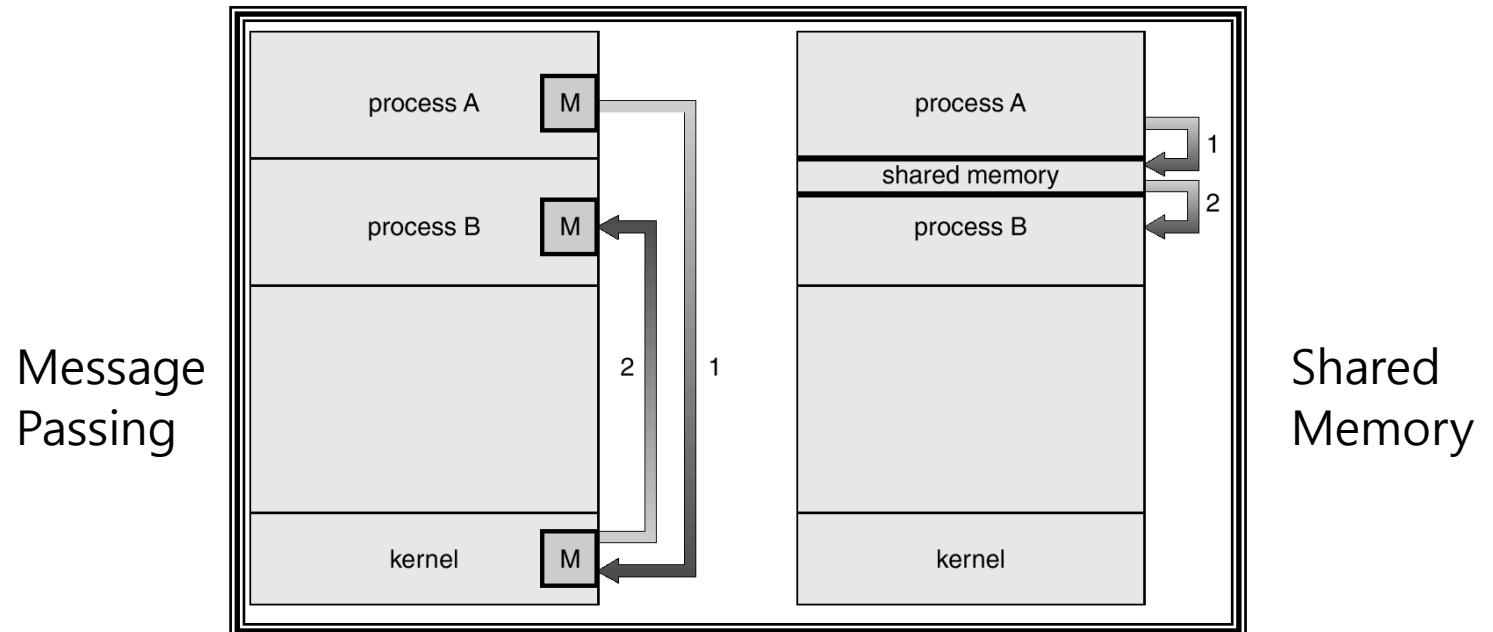
- ✓ Race condition : the process that updates the value of a global variable last determines the final value of the variable.

Recall
Process Control :
Termination,
Communication

Inter-Process Communication (IPC)

- ✓ **Independent process** : not affect or be affected by another process
- ✓ **Cooperating process** : can affect or be affected by the execution of another process. → distributed system
- ✓ **Interprocess Communication(IPC) for the information sharing**

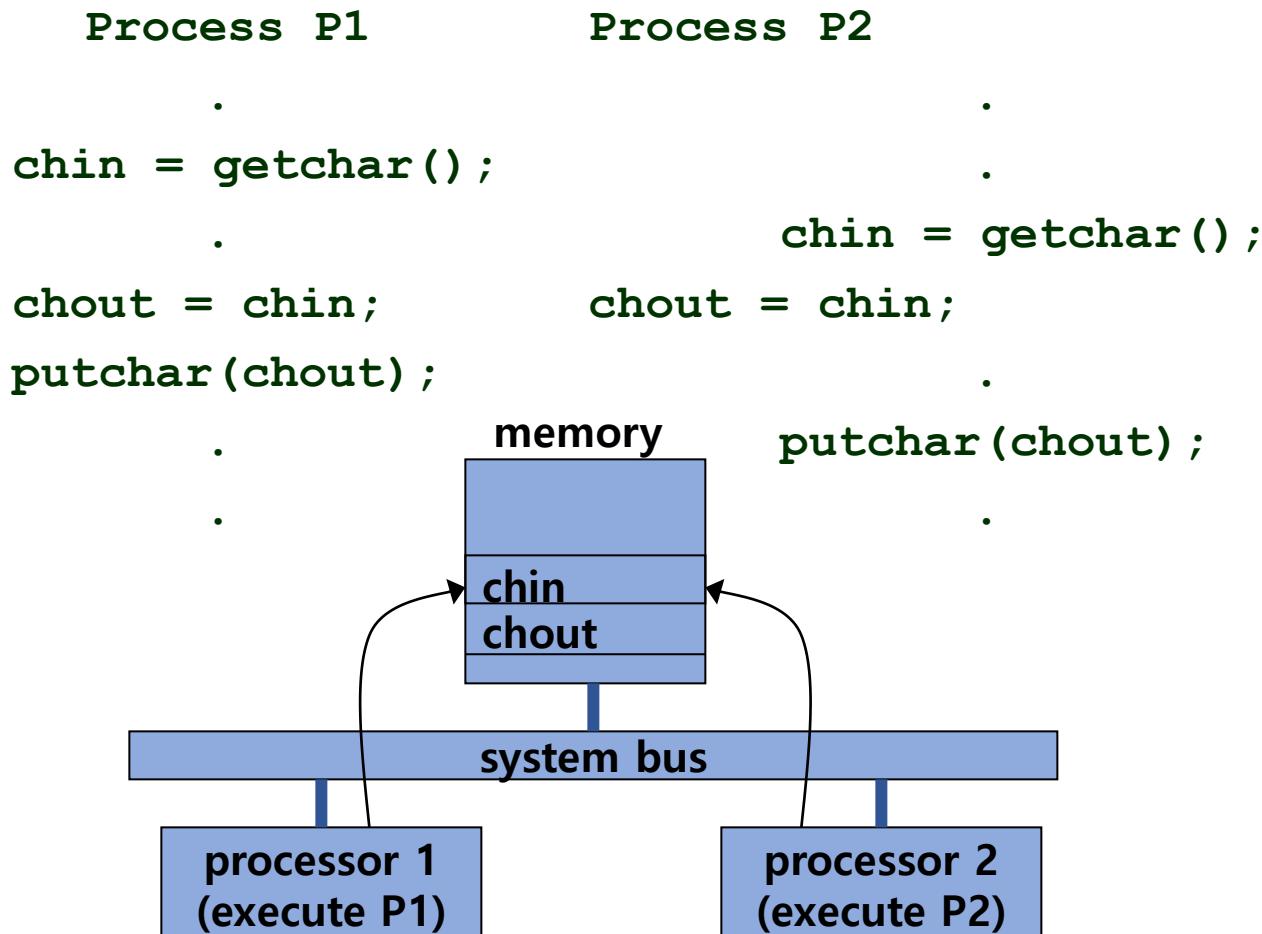
◎ Message passing, shared memory, Remote Procedure Call



심화

Race Condition in a Multiprocessor

- ✓ P1 , P2 are both executing each on a separate processor.



Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제16강

Race Condition

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



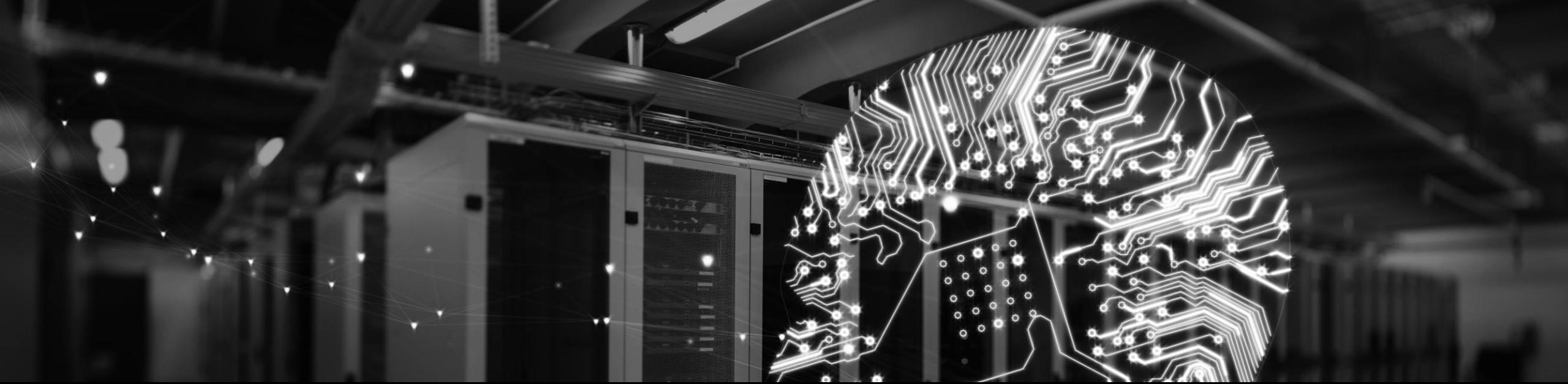
학습 목표

- 💬 Producer-consumer Problem의 의미를 이해한다.
- 💬 Producer-consumer Problem 코드를 학습한다.
- 💬 레이스 컨디션(Race Condition)의 발생 가능 이유를 이해한다.
- 💬 레이스 컨디션 방지 방법을 이해한다.
- 💬 Atomic Operation의 의미와 동작 방법을 이해한다.

학습 내용

- ⌚ Producer-consumer Problem의 의미
- ⌚ Producer-consumer Problem 코드
- ⌚ 레이스 컨디션의 발생 가능 이유
- ⌚ 레이스 컨디션 방지 방법
- ⌚ Atomic Operation의 의미와 동작 방법





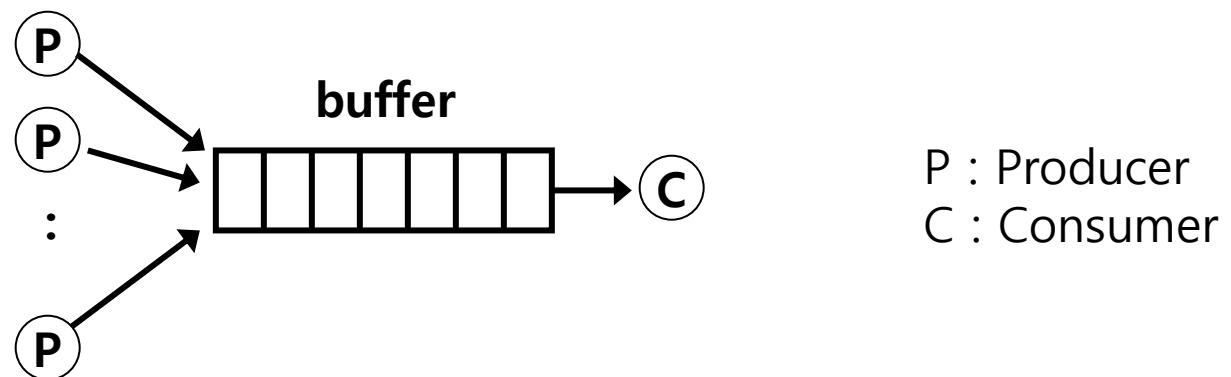
Race Condition



Example of Race Condition

✓ Producer/Consumer Problem

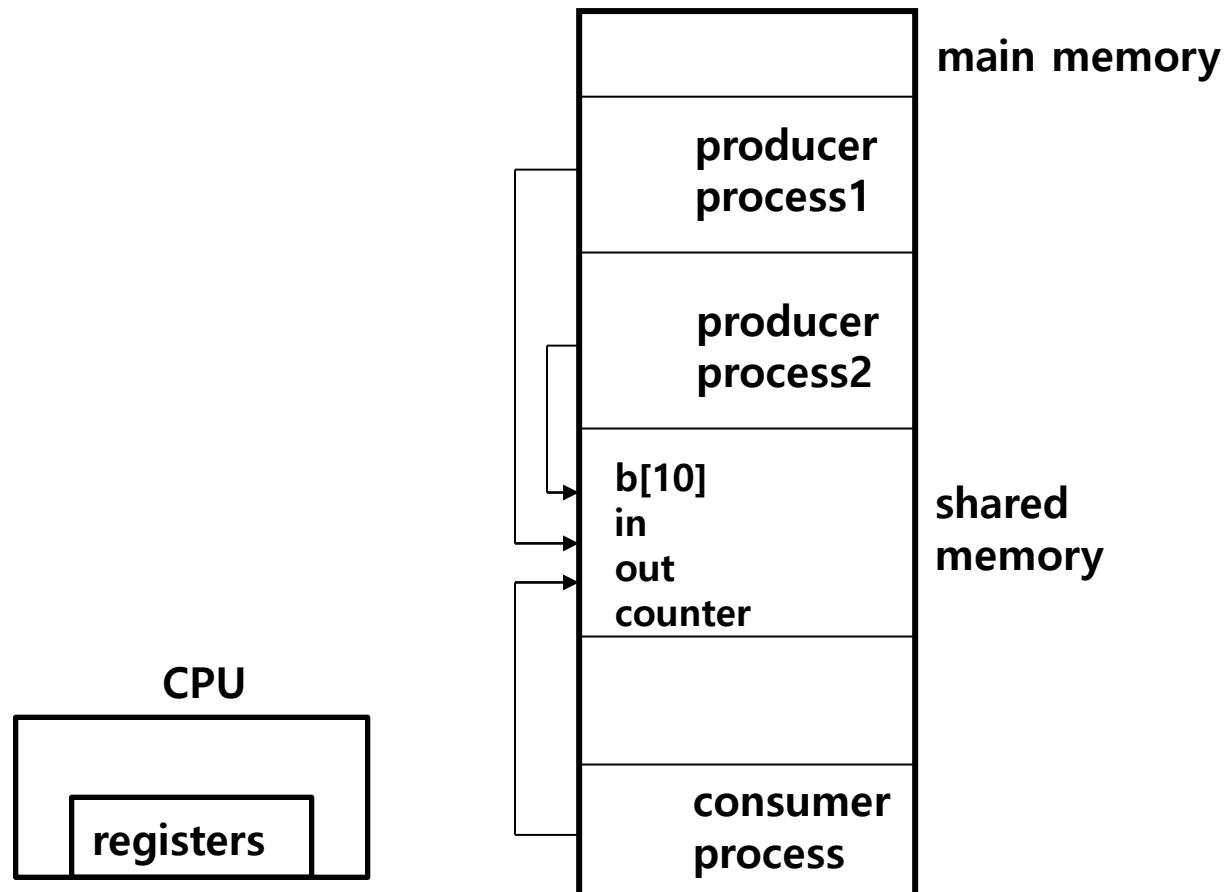
- ① Assume multiple producer processes and a consumer process run on one processor.
- ② One or more producers are generating data and placing these in a buffer
- ③ A single consumer is taking items out of the buffer one at time
- ④ An example of IPC(Interprocess Communication) of Chap 3.
- ⑤ Only one producer or consumer may access the buffer at any one time



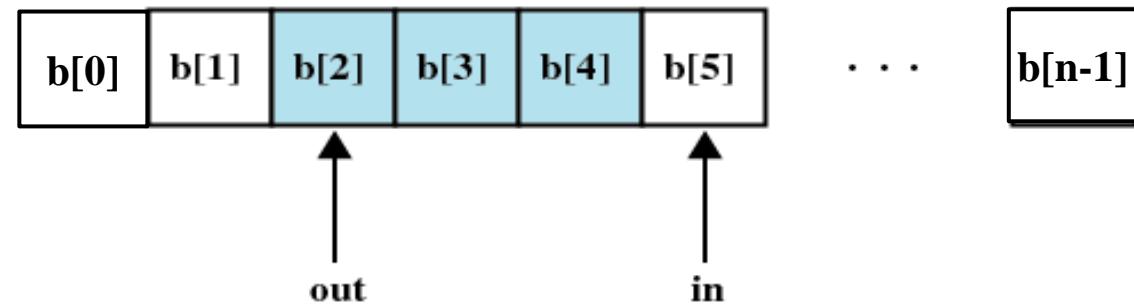
Producer/Consumer : Finite Buffer

✓ Shared data (variables stored in the shared memory)

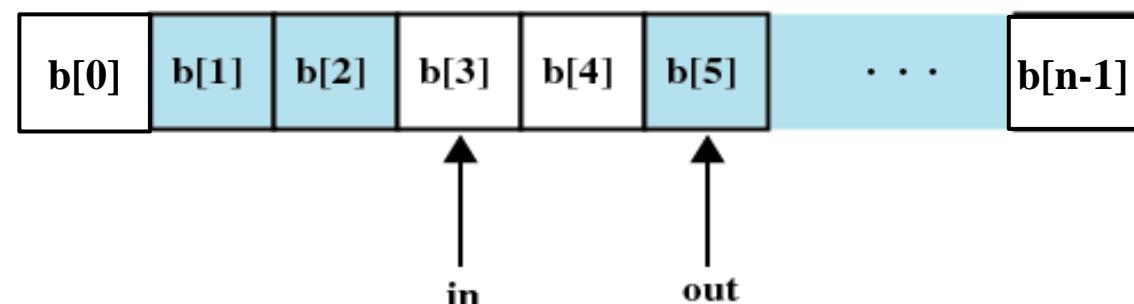
```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item b[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```



Producer/Consumer : Finite Buffer



(a)

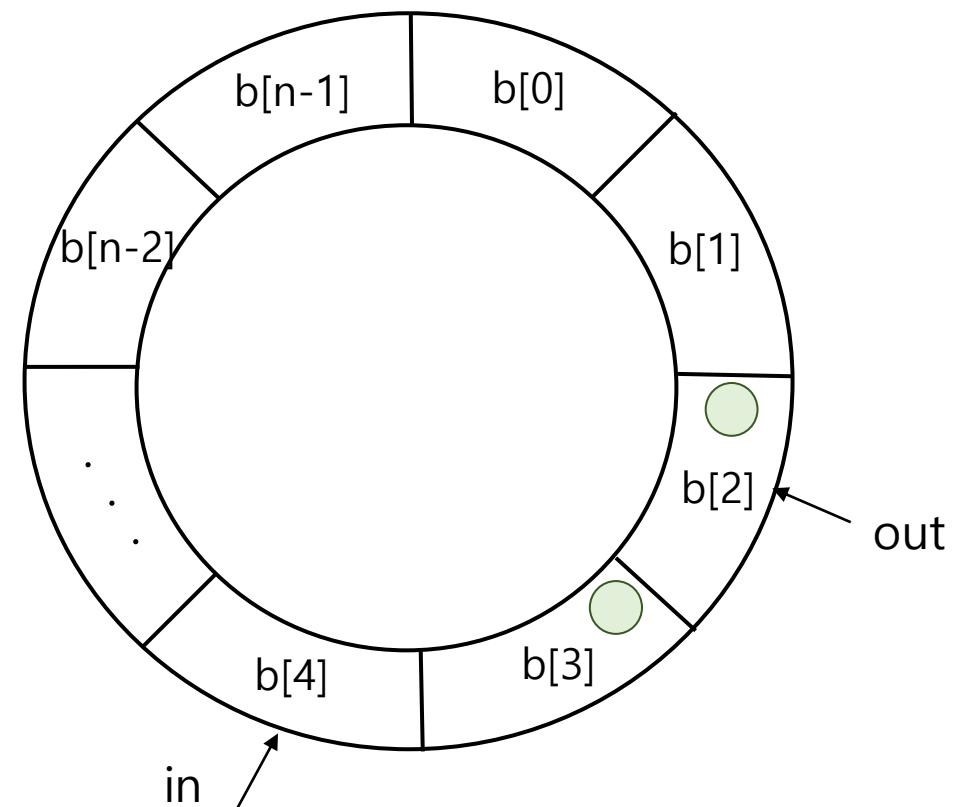


(b)

Finite Circular Buffer for the Producer/Consumer Problem

Producer/Consumer : Finite Buffer

✓ A finite-sized circular buffer



Producer/Consumer : Finite Buffer

✓ A solution for a finite-sized circular buffer

✓ Producer process :

```
item v;

while (1) {
    while(counter == BUFFER_SIZE); /* full, then do nothing */
    /* produce item v */
    b[in] = v;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Producer/Consumer : Finite Buffer

✓ Consumer process

```
item w;

while (1) {
    while (counter == 0) ; /* empty, then do nothing */
    w = b[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume item w */
}
```

Race Condition in a Uniprocessor(1)

✓ IPC using a shared memory

- ④ Synchronization between accesses by multiple processes to the shared memory is required

✓ The statements

```
counter++;  
counter--;
```

must be performed atomically for mutual exclusion.

- ④ Atomic operation means an operation that completes in its entirety without interruption.
- ④ Do not allow for an interrupt to occur (disable interrupts) during the execution of the operation.

✓ But in reality, they are not atomic operations.

- ④ Needs a synchronization mechanism to provide mutual exclusion.

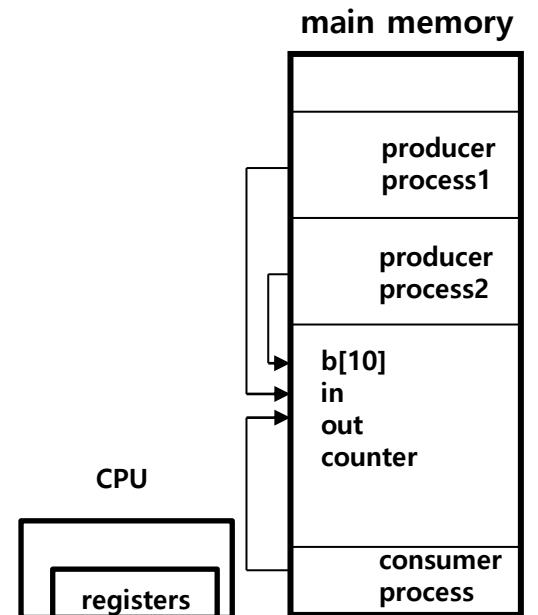
Race Condition in a Uniprocessor(2)

- ✓ The statement “counter++” may be implemented in lower level assembly language as:

```
load register, counter    (register ← counter)  
increment register        (register ← register + 1)  
store register, counter   (counter ← register)
```

- ✓ The statement “counter- -” may be implemented as:

```
load register, counter    (register ← counter)  
decrement register        (register ← register - 1)  
store register, counter   (counter ← register)
```



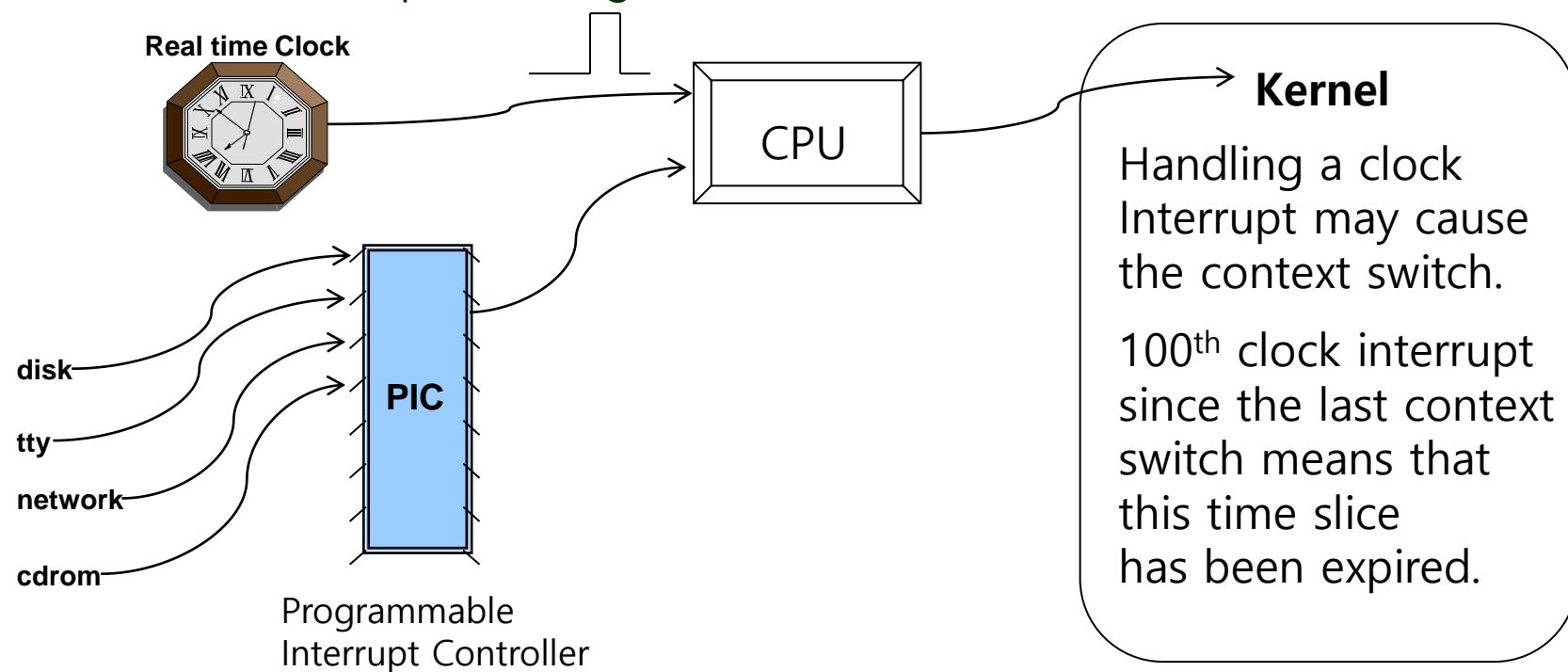
- ✓ If both the producer and consumer attempt to update the buffer concurrently by clock interrupt

◎ The assembly language statements may get interleaved (mixed in sequence).

Interrupt

- ✓ A mechanism that peripheral devices inform an asynchronous event to an operating system
- ✓ A clock interrupt occurs every millisecond (1 clock interrupt/ms)

◎ 100 clock interrupts during one time slice (100 milliseconds)



Race Condition in a Uniprocessor(3)

✓ Example of Race Condition:

Suppose a producer puts an item in the buffer and counter is 5,

load register, counter (register \leftarrow counter = 5)
increment register (register = 6)

At this time, clock interrupt occurs, CPU is dispatched to the consumer

load register, counter (register \leftarrow counter = 5)
decrement register (register = 4)
store register, counter (counter \leftarrow 4 ; register of consumer)

By clock interrupt, CPU is given back to the producer

store register, counter (counter \leftarrow 6 ; register of producer)

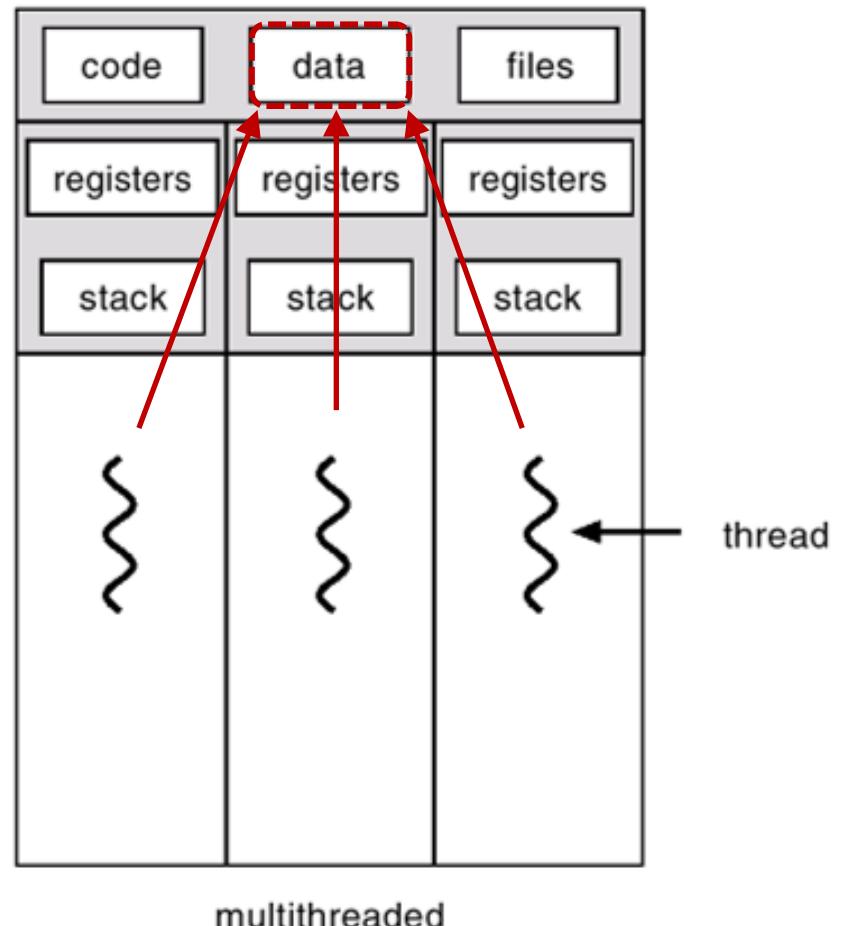
✓ Final value of “counter” is either 4 or 6.

- ④ Interleaving depends upon how the producer and consumer processes are scheduled.
- ④ In order for race condition not to happen, we need *mutual exclusion*.

Race Condition in One Process

Multithreading

- Threads of a process concurrently access **data** area of the process.
- Race condition may happen, we need to prevent the race condition.



How Do We Prevent Race Condition?

1. Make a critical section be executed as atomic operations.
It works, but it is **not a good way**
A critical section may be lengthy and concurrency is limited

2. Make a fence around a critical section
Inside the fence, mutual exclusion is guaranteed
Inside the fence, the **critical section is not atomic**

» 2 is better way

Atomic Operation

Execution of ordinary instruction (non-atomic operation)

CPU checks interrupt line
load register, counter
CPU checks interrupt line
increment register
CPU checks interrupt line
store register, counter
CPU checks interrupt line
next instruction

Execution of Atomic operation

interrupt disable
load register, counter
increment register
store register, counter
interrupt enable
CPU checks interrupt line
next instruction

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제17강

Critical-Section Problem :
Software Solution

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



학습 목표

- 💬 Critical-Section Problem을 이해한다.
- 💬 Critical-Section Problem 해결책의 요구 사항을 이해한다.
- 💬 소프트웨어적 해결 방법을 이해한다.

학습 내용

- ⌚ Critical-Section Problem
- ⌚ Critical-Section Problem 해결책의 요구 사항
- ⌚ 소프트웨어적 해결 방법





Critical-Section Problem



The Critical-Section Problem

- ✓ n processes all competing to use some shared data
- ✓ Each process has a code segment, called *critical section*, in which the shared data is accessed.

✓ Critical-Section Problem – Make sure that

- ◎ When one process is executing in its critical section
(e.g.: when A is executing in A's critical section),
- ◎ No other process is allowed to execute in B's critical section
(B is not allowed to execute B's critical section).

Requirements of Critical-Section Problem

1. Mutual Exclusion

- » If a process is executing in its critical section,
- » Then no other processes can enter (executes) their critical sections.

2. Progress

- » If there is a process that wishes to enter its critical section and
- » No other process is executing in its critical section,
- » Then this process should be able to enter by itself.

3. Bounded Waiting

- » If there is a process that wishes to enter its critical section and
 - A process is executing in its critical section,
- » Then the waiting process(processes) can enter its(their) critical section in
 - some time (does not wait indefinitely).

Recall
Race Condition

How Do We Prevent Race Condition?

1. Make a critical section be executed as atomic operations.
It works, but it is **not a good way**
A critical section may be lengthy and concurrency is limited

 2. Make a fence around a critical section
Inside the fence, mutual exclusion is guaranteed
Inside the fence, the **critical section is not atomic**
- » 2 is better way

Software Solution

- ✓ In case that there are two processes, P_i and P_j
- ✓ General structure of process P_i (other process P_j)

```
do  {
    entry section
    critical section
    exit section
    remainder section
} while (TRUE);
```

- ✓ Processes may share some common variables to synchronize their actions.

Software Solution 1

✓ Shared variables:

① **int turn;**

initially **turn = 0**

② **turn == i** $\Rightarrow P_i$ can enter its critical section

✓ Process P_i

```
do {  
    while (turn != i) ;  
    critical section  
    turn = j;  
    remainder section  
} while (TRUE);
```

✓ Satisfies mutual exclusion, but not progress. Why?

Software Solution 1

✓ Process P_0 ($i=0, j=1$)

```
do {  
    while (turn != 0) ;  
    critical section  
    turn = 1;  
    remainder section  
} while (TRUE);
```

✓ Process P_1 ($i=1, j=0$)

```
do {  
    while (turn != 1) ;  
    critical section  
    turn = 0 ;  
    remainder section  
} while (TRUE);
```

Software Solution 2

✓ Known as Peterson's Algorithm

- ④ Meets all three requirements; solves the critical-section problem for *two* processes.

✓ Process P_i

```
do {  
    flag [i] = true;  
    turn = j;  
    while (flag [j] and turn == j) ;  
        critical section  
    flag [i] = false;  
        remainder section  
} while (TRUE);
```

Software Solution 2

✓ Process P_0 ($i=0, j=1$)

```
do {  
    flag [0] = true;  
    turn = 1;  
    while (flag [1] and turn == 1) ;  
        critical section  
    flag [0] = false;  
        remainder section  
} while (TRUE);
```

✓ Process P_1 ($i=1, j=0$)

```
do {  
    flag [1] = true;  
    turn = 0;  
    while (flag [0] and turn == 0) ;  
        critical section  
    flag [1] = false;  
        remainder section  
} while (TRUE);
```

심화

Software Solution 3 : Bakery Algorithm

✓ Critical section for n processes ($n \geq 2$)

```
do {  
    choosing[i] = true;  
    number[i] = max(number[0], number[1], ..., number [n - 1])+1;  
    choosing[i] = false;  
    for (j = 0; j < n; j++) {  
        while (choosing[j]) ;  
        while ((number[j] != 0) && (number[j],j) < (number[i],i)) ;  
    }  
    critical section  
    number[i] = 0;  
    remainder section  
} while (TRUE);
```

✓ This software solution is not practical, do not use it!

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제19강

Semaphore

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



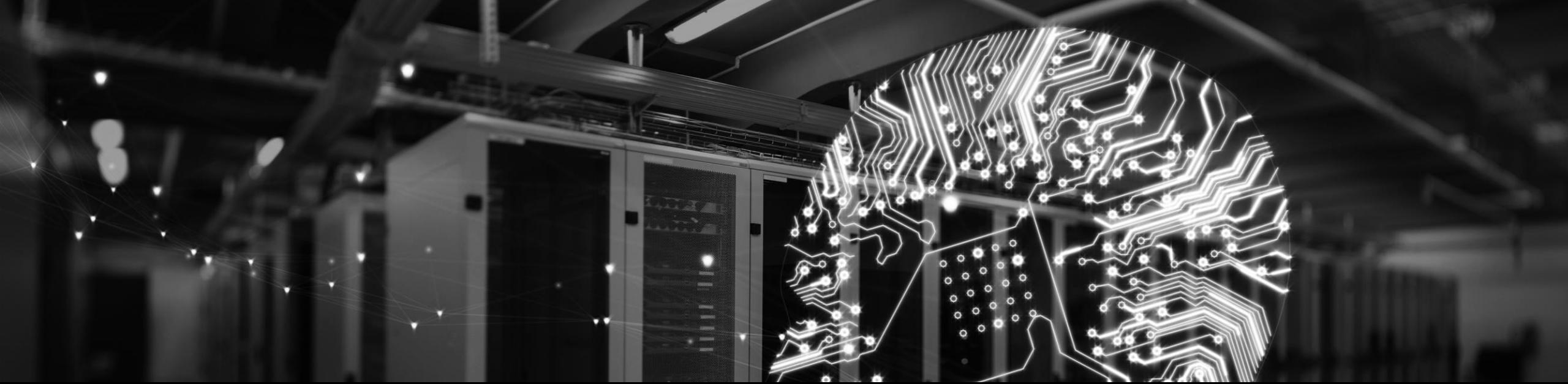
학습 목표

- ☞ 세마포어(Semaphore)의 정의를 이해한다.
- ☞ SemWait, SemSignal Operation을 이해한다.
- ☞ 세마포어를 이용한 Critical-Section Problem 해결책의 장단점을 이해한다.

학습 내용

- ☞ 세마포어(Semaphore)의 정의
- ☞ semWait, semSignal Operation
- ☞ 세마포어를 이용한 Critical-Section Problem 해결책의 장단점





Semaphore



Semaphores

A better hardware-supported synchronization mechanism

- ◎ Because the semaphore avoids busy waiting
- ◎ Meets mutual exclusion, progress, bounded waiting requirements

Special variable called a **semaphore** is used for signaling

- ◎ Semaphore is a variable that has an integer value
- ◎ If a process is waiting for a resource, it is blocked until that the resource is available.

semWait (s) operation decreases the value of the semaphore s

- ◎ Also called P(s) operation, or Wait(s) operation, or down(s)

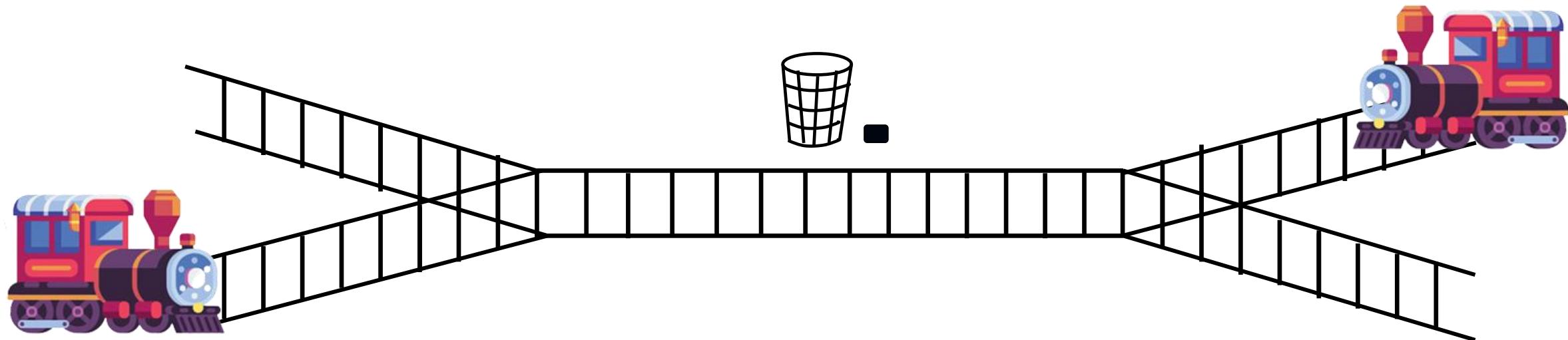
semSignal (s) operation increases value of the semaphore s

- ◎ Also called V(s), or Signal(s), or up(s), or sem_post(s) operation

semWait (s) / semSignal (s) function is carried out atomically (no interrupt is allowed).

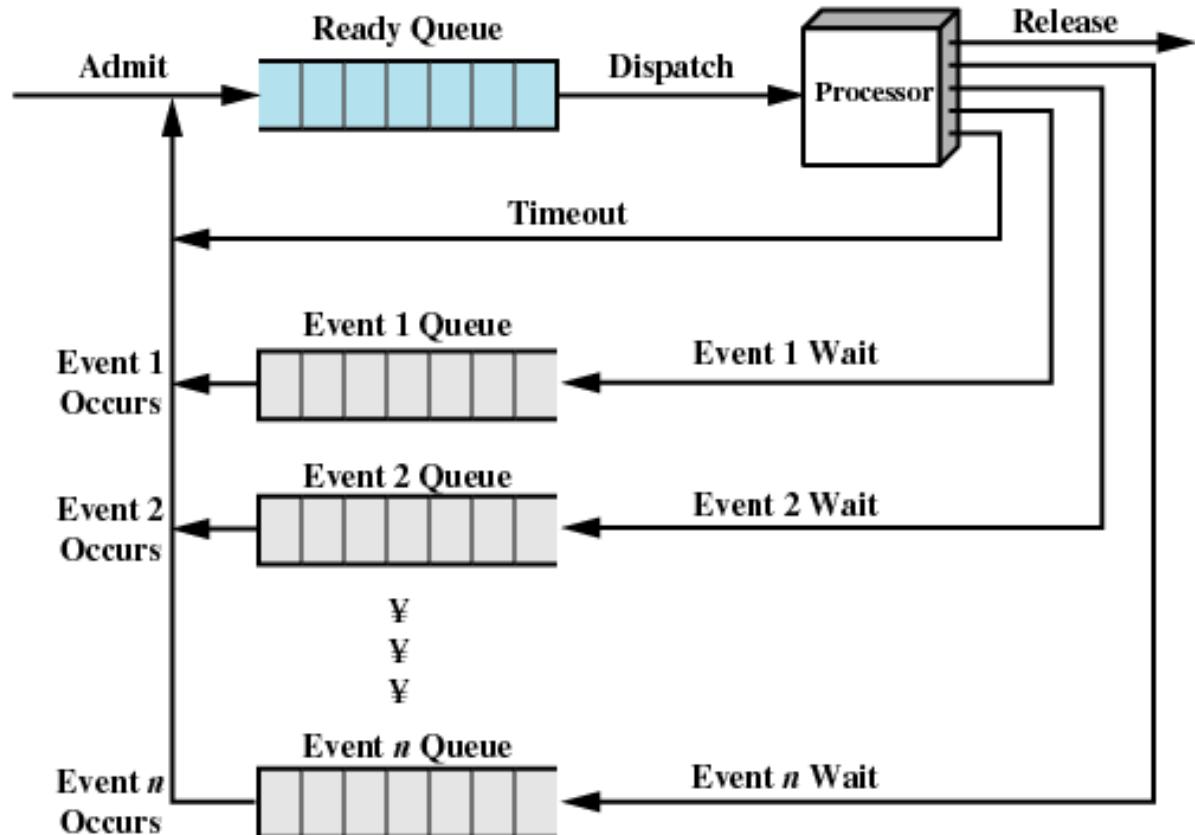
Semaphores

✓ How to prevent a collision?



Recall
Process State Model

Mutual Exclusion: Hardware Support



(b) Multiple blocked queues

Queuing Model

Semaphore Primitives

```
struct semaphore {
    int count;
    queueType queue;
}

void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0)
    {
        place this process in s.queue;
        block this process
    }
}
void semSignal(semaphore s)
{
    s.count++;
    if (s.count <= 0)
    {
        remove a process P from s.queue;
        place process P on ready list;
    }
}
```

A Definition of Semaphore primitives

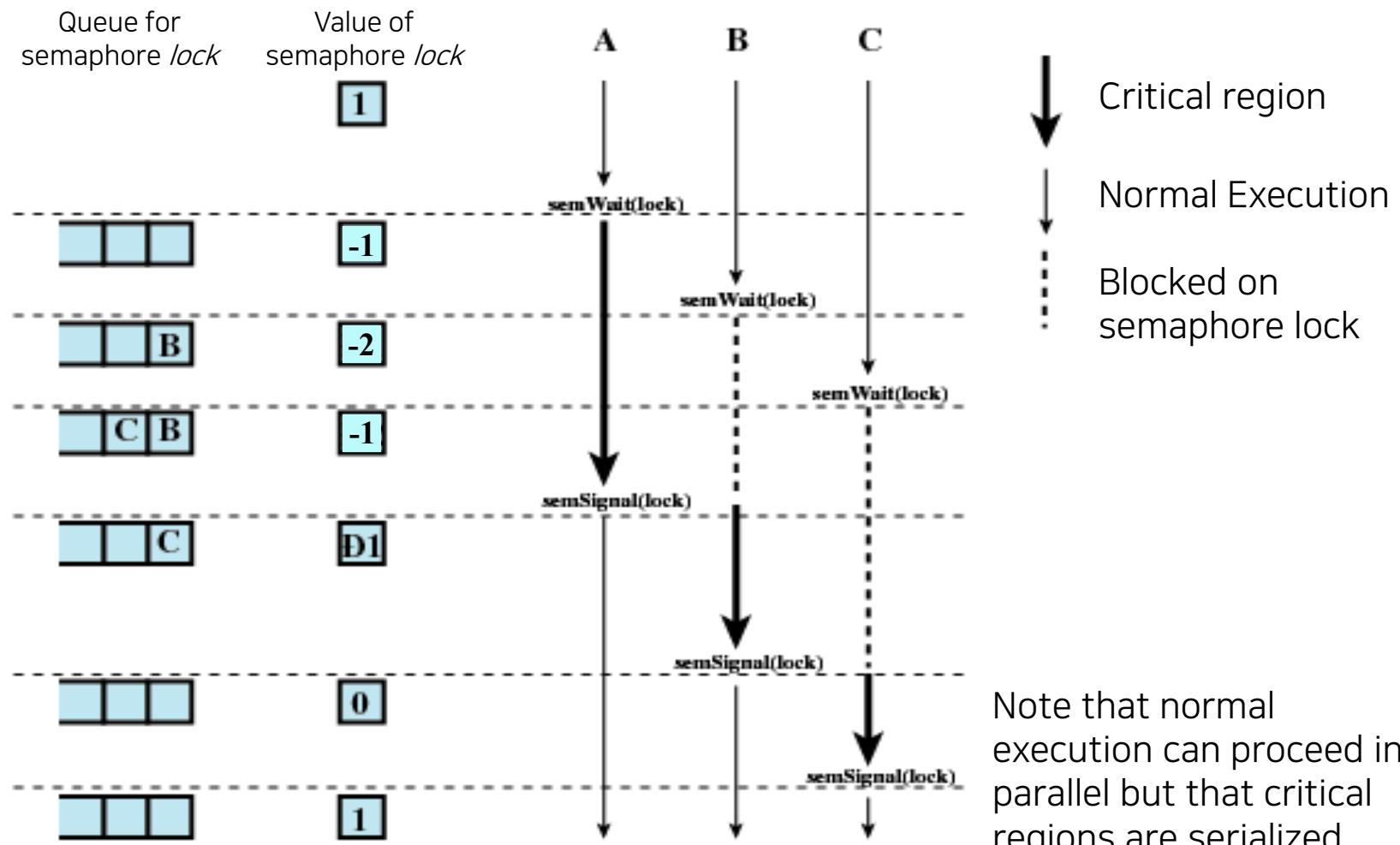
Mutual Exclusion Using Semaphores

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true)
    {
        semWait(s);
        /* critical section */;
        semSignal(s);
        /* remainder */;
    }
}
void main()
{
    parbegin (P(1), P(2), . . . , P(n));
}
```

Mutual Exclusion Using Semaphores

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true)
    {
        semWait(s);
        /* critical section */;
        semSignal(s);
        /* remainder */;
    }
}
void main()
{
    parbegin (P(1), P(2), . . . , P(n));
}
```

Mutual Exclusion Machine Instructions



심화

Recall

Software solution

Software Solution 3 : Bakery Algorithm

✓ Critical section for n processes ($n \geq 2$)

do {

```
choosing[i] = true;  
number[i] = max(number[0], number[1], ..., number [n - 1])+1;  
choosing[i] = false;  
for (j = 0; j < n; j++) {  
    while (choosing[j]) ;  
    while ((number[j] != 0) && (number[j],j) < (number[i],i)) ;  
}
```

critical section

number[i] = 0;

remainder section

} while (TRUE);

Recall
Hardware solution

Test and Set Instruction

```
/* program mutual_exclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true)
    {
        while (! testset(bolt)) /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */
    }
}

void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . , P(n));
}
```

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제21강

Deadlock and Starvation

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



학습 목표

- 💬 데드락(Dead lock) 발생 상황을 이해한다.
- 💬 Resource allocation graph를 이해한다.
- 💬 데드락 발생 조건을 이해한다.

학습 내용

- ⌚ 데드락(Dead lock) 발생 상황
- ⌚ Resource Allocation Graph
- ⌚ 데드락 발생 조건





What is a Deadlock?



Recall
What is Conc. Control?

Some Key Terms in Concurrency

✓ Starvation

- ④ A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

✓ Deadlock

- ④ A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
- ④ Starvation for an infinite duration

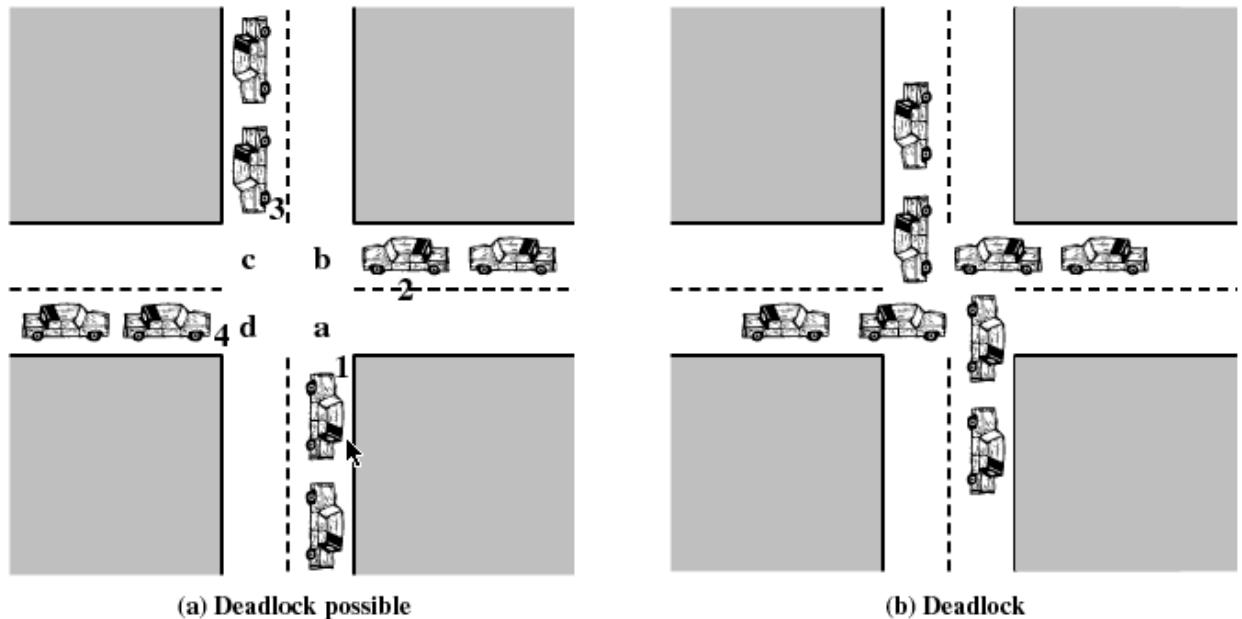


Illustration of Deadlock

Principles of Deadlock

- ✓ Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- ✓ No efficient solution
- ✓ Involve conflicting needs for resources by two or more processes

Process P

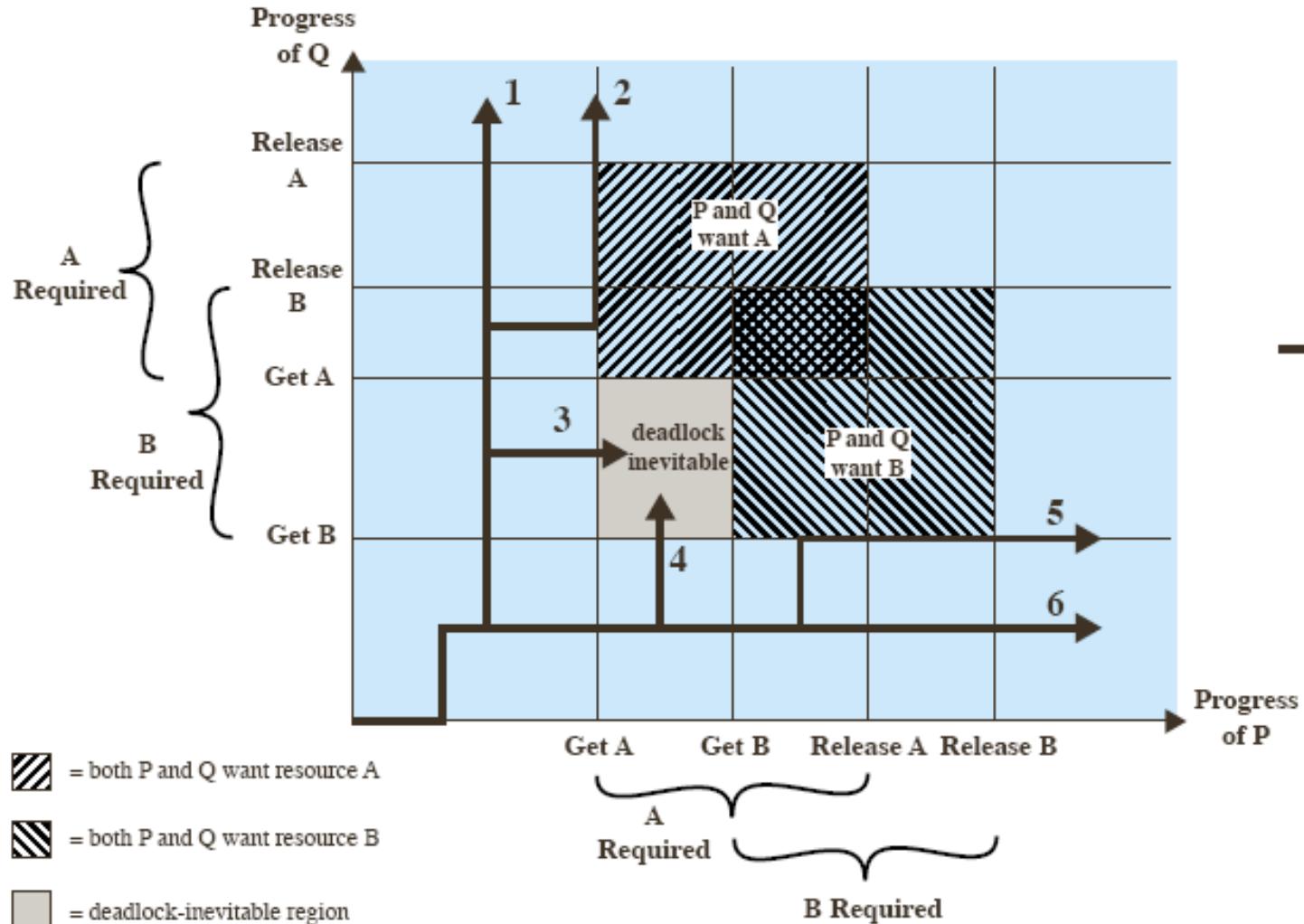
```
...
Get A ←
...
Get B
...
Release A
...
Release B
...
```

Process Q

```
...
Get B
...
Get A
...
Release B
...
Release A
...
```

심화

Deadlock



→ = Possible progress path of P and Q.
 Horizontal portion of path indicates P is executing and Q is waiting.
 Vertical portion of path indicates Q is executing and P is waiting.

Example of Deadlock

No Deadlock

Process P

...
Get A
...
Release A →
...
Get B
...
Release B
...

Process Q

...
Get B
...
Get A
...
Release B
...
Release A
...

Type of Resources

Reusable Resources

- ④ Processes obtain resources that they later release for reuse by other processes.
- ④ Used by only one process at a time and not depleted by that use.
- ④ Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores.
- ④ Deadlock occurs if each process holds one resource and requests the other.

Consumable Resources

- ④ Once a resource is used by a process, it is not used by other processes.
- ④ The resource disappears after it is consumed.

Deadlock Example : Reusable Resources

- ✓ Memory space is available for allocation of 200Kbytes, and the following sequence of events occur

P1

...

Request 80 Kbytes;

...

Request 60 Kbytes;

P2

...

Request 70 Kbytes;

...

Request 80 Kbytes;

- ✓ Deadlock occurs if both processes progress to their second request

Deadlock Example : Consumable Resources

- ✓ **Created (produced) and destroyed (consumed)**
- ✓ **Interrupts, signals, messages, and information in I/O buffers**
- ✓ **Deadlock may occur if a Receive message is blocking**
- ✓ **May take a rare combination of events to cause deadlock**
- ✓ **Compare with reusable resources**
- ✓ **Example of deadlock : deadlock occurs if blocking receive is used**

P1

...

Receive(P2); ①

...

Send(P2, M1); ③

P2

...

Receive(P1); ②

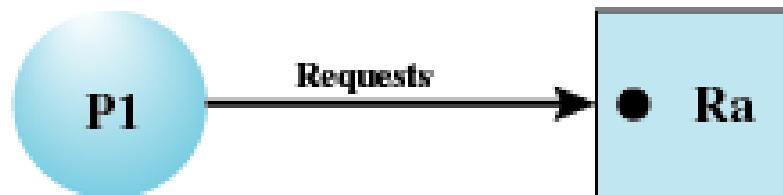
...

Send(P1, M2);

How to Represent Occurrence of Deadlock?

✓ Resource Allocation Graphs

- ④ Directed graph that depicts a state of the system of resources and processes

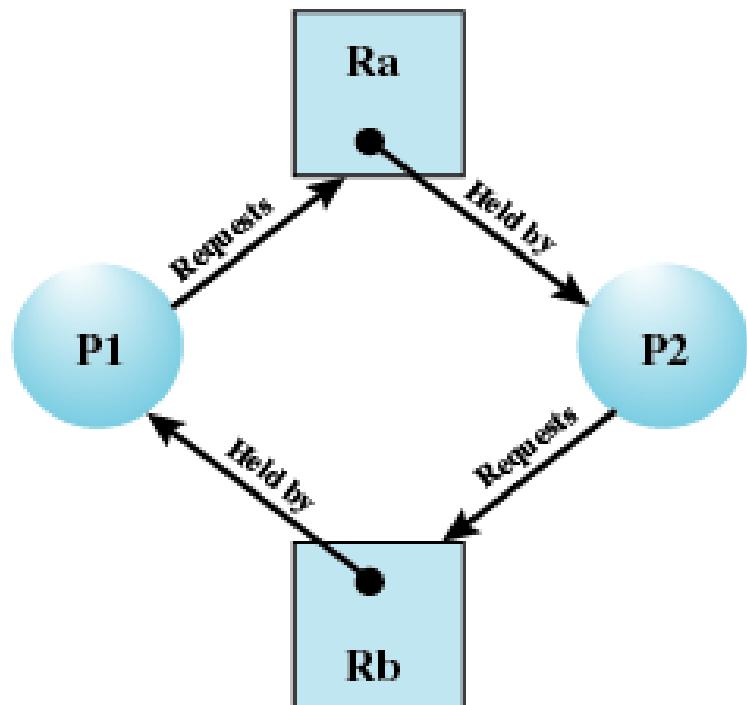


(a) Resource is requested

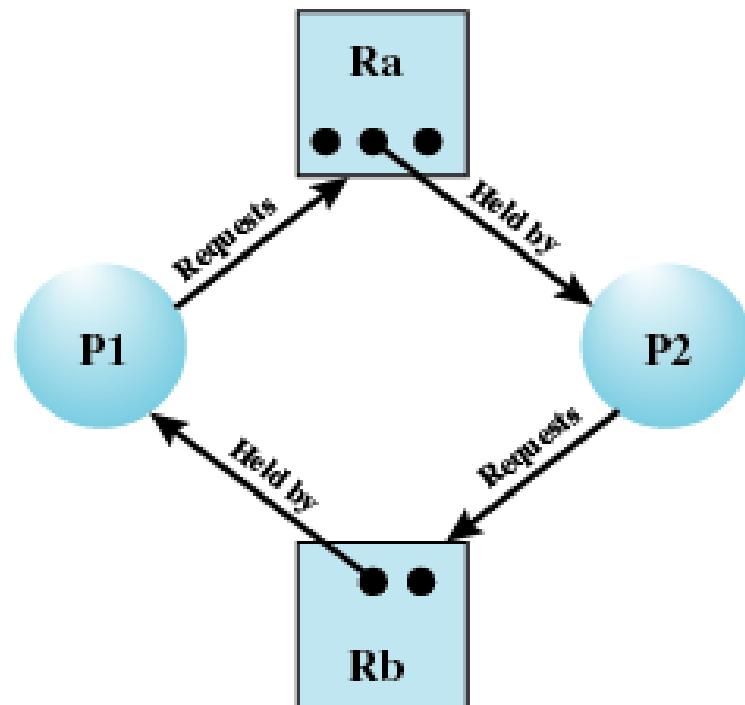


(b) Resource is held

Resource Allocation Graphs



(c) Circular wait

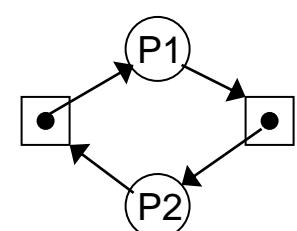
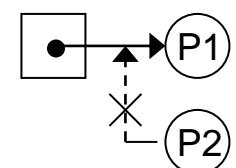
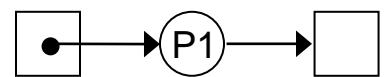
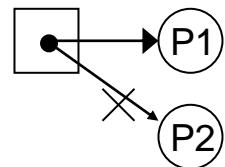


(d) No deadlock

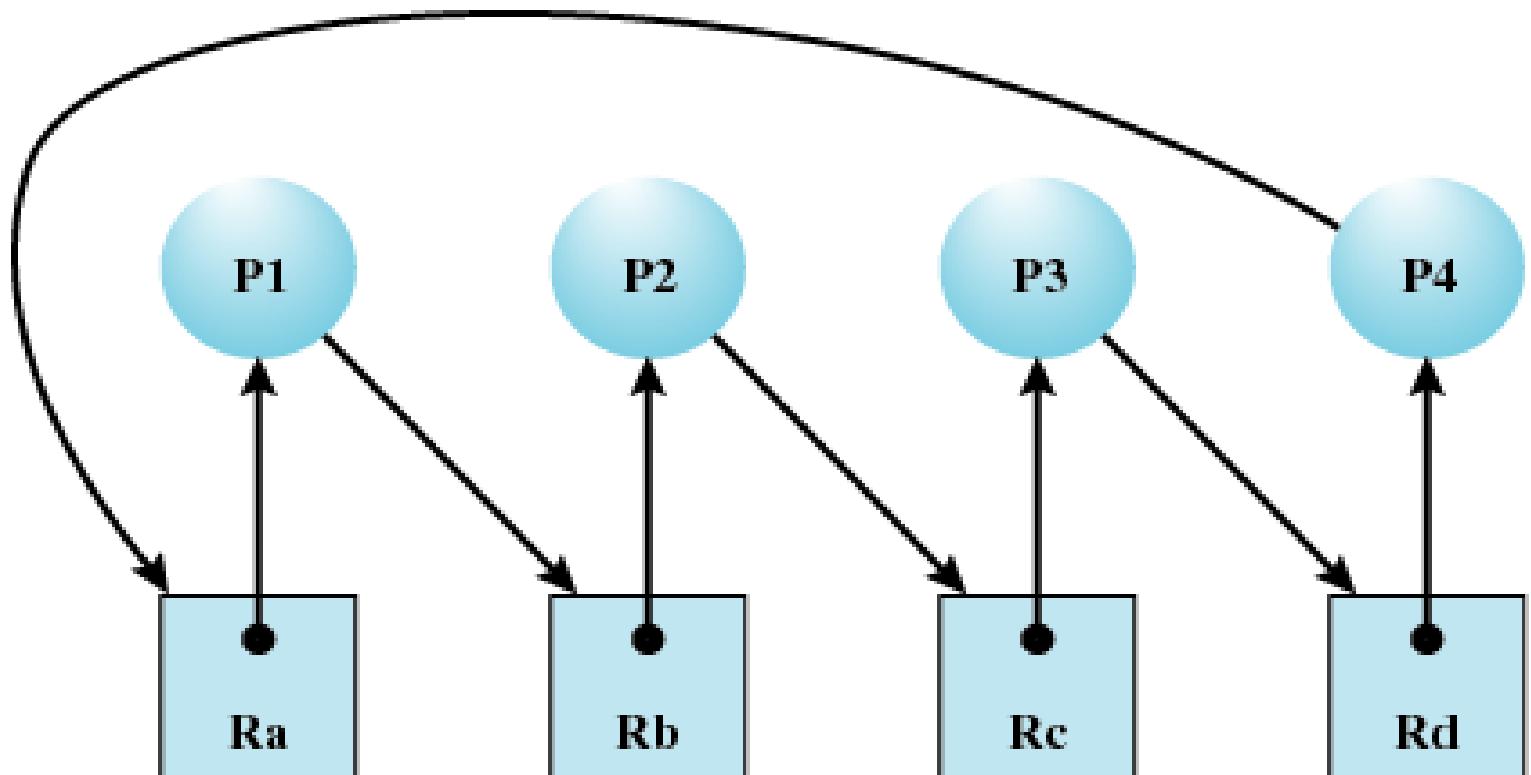
Example of Resource Allocation Graphs

Conditions for Deadlock

1. Mutual exclusion
 - Only one process may use a resource at a time
2. Hold-and-wait
 - process may hold allocated resources while awaiting assignment of others
3. No preemption
 - No resource can be forcibly removed from a process holding it
4. Circular wait
 - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain



Conditions for Deadlock



Resource Allocation Graph

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제22강

How to Handle Deadlocks?

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



학습 목표

- 💬 데드락(Deadlock) 대처 방법을 이해한다.
- 💬 데드락 예방 방법을 이해한다.
- 💬 데드락 발생 시 사후 대처 방법을 이해한다.
- 💬 Dinning philosopher's problem이 상징하는 현상을 이해한다.

학습 내용

- ⌚ 데드락(Deadlock) 대처 방법
- ⌚ 데드락 예방 방법
- ⌚ 데드락 발생 시 사후 대처 방법
- ⌚ Dinning philosopher's problem이 상징하는 현상





How to Handle Deadlocks?

Methods for Handling Deadlocks

1. Ensure that the system will never enter a deadlock state

- ④ Deadlock prevention : make processes not to satisfy 4 conditions
 - » Impractical : each of 4 conditions needs to be maintained
- ④ Deadlock avoidance : use deadlock avoidance algorithm which check safe state whenever resources are requested
 - » Impractical : claiming of resource need is difficult
 - » Inefficient : computational overhead

2. Allow the system to enter a deadlock state and then recover

- ④ Deadlock detection : run deadlock algorithm from time to time
 - » More practical than 1. Still has computational overhead

3. Operating system does nothing

- ④ If a problem occurs, let a human does something
 - » Used by most operating systems (UNIX, Windows, Linux, …)

Deadlock Prevention : Breaking 4 Conditions

✓ Break the Mutual Exclusion property

- ◎ Impossible, many resources require mutual exclusive access.
- ◎ So, Mutual Exclusion must be supported by the operating system

✓ Break the Hold and Wait property

- ◎ Requires a process request all of its required resources at one time
- ◎ Possible, but bad utilization of resources

✓ Break the No Preemption property

- ◎ Process must release resource and request again
- ◎ Operating system may preempt a process to require it releases its resources
- ◎ Possible, but bad utilization of resources

✓ Break the Circular Wait property

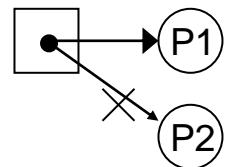
- ◎ Define a linear ordering of resource types
- ◎ Possible, but inconvenient

Recall
What is a Deadlock?

Conditions for Deadlock

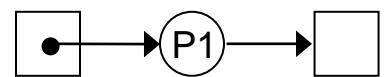
1. Mutual exclusion

- Only one process may use a resource at a time



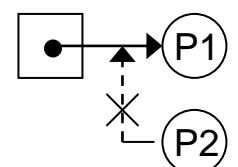
2. Hold-and-wait

- process may hold allocated resources while awaiting assignment of others



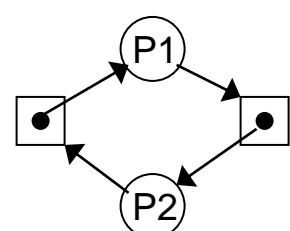
3. No preemption

- No resource can be forcibly removed from a process holding it



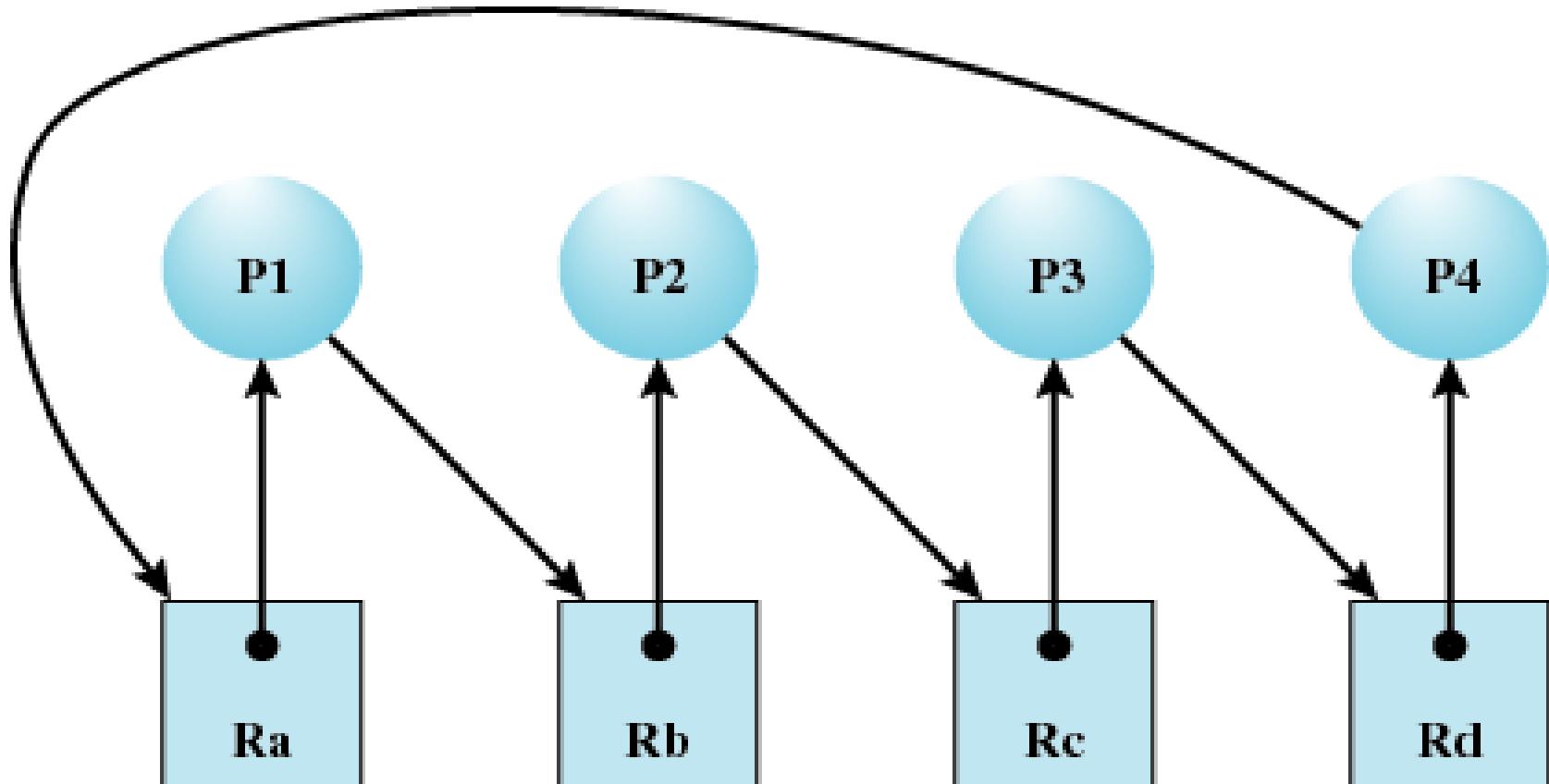
4. Circular wait

- A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain



Recall
What is a Deadlock?

Conditions for Deadlock



Resource Allocation Graph

Methods for Handling Deadlocks

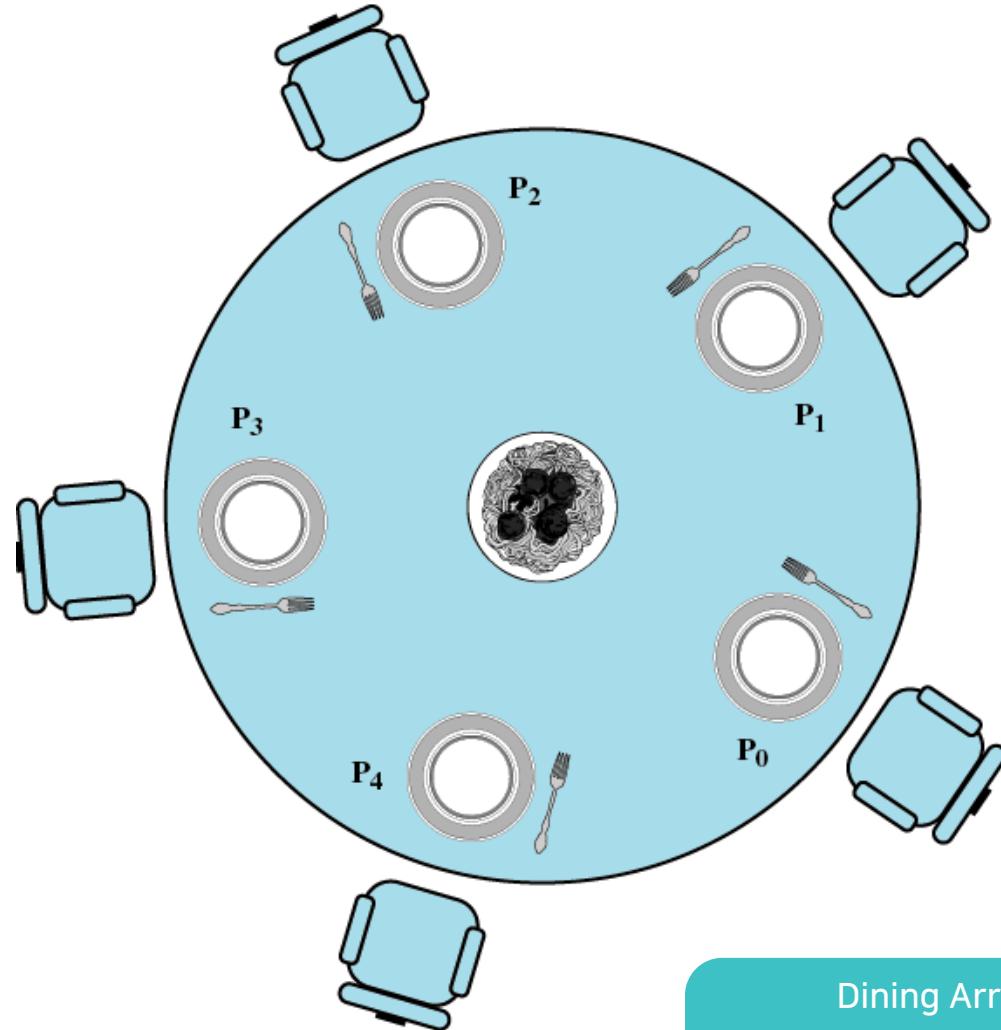
1. Abort all deadlocked processes, or
2. Abort one deadlocked process at a time until deadlock no longer exists

- ④ In which order should we choose to abort?
 - » Least amount of processor time consumed so far
 - » Least number of lines of output produced so far
 - » Most estimated time remaining
 - » Least total resources allocated so far
 - » Lowest priority

Dining Philosophers Problem

Deadlock

Starvation



Dining Arrangement for Philosophers

심화

An Incorrect Solution to the Problem

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true)
    {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
```

wait = semWait
signal = semSignal

심화

A Correct Solution to the Problem

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int I)
{
    while (true)
    {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
```

A Second Solution to the Dining Philosophers Problem

심화

Deadlock Prevention : Banker's Algorithm

✓ Use of the **banker's algorithm**

✓ **Banker's algorithm decides dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock**

- ◎ Requires knowledge of future process request
- ◎ Safe state is a state where there is at least one (execution) sequence that does not result in a deadlock
- ◎ Unsafe state is a state where safe sequence does not exist (Deadlock is inevitable)

✓ Approaches to avoid deadlock

- ◎ Do not start a process if its demands might lead to deadlock, or
- ◎ Do not grant an incremental resource request to a process if there is no safe sequence for this request. (Because this allocation might lead to deadlock)

심화

Banker's Algorithm : Safe State

- ✓ Current situation : P1 requested 80 KB, and it has been granted

Available vector = (200) // amount of main memory = 200 KB

Claim Matrix

	Memory
P1	140 (80+60) KB
P2	150 (70+80) KB

Allocation Matrix

	Memory
P1	80 KB
P2	0 KB

Need Matrix

	Memory
P1	60 KB
P2	150 KB

Available vector = (120)

- ✓ Then assume that P1 again requests 60 KB, and it is granted

Claim Matrix

	Memory
P1	140 (80+60) KB
P2	150 (70+80) KB

Allocation Matrix

	Memory
P1	140 KB
P2	0 KB

Need Matrix

	Memory
P1	0 KB
P2	150 KB

Available vector = (60)

- ✓ P1 can finish execution and then P2 can start and finish execution
- ✓ Safe sequence exist (P1 → P2). We can avoid a deadlock

심화

Banker's Algorithm : Unsafe State

- ✓ Current situation : P1 requested 80 KB, and it has been granted

Available vector = (200)

Claim Matrix

	Memory
P1	140 (80+60) KB
P2	150 (70+80) KB

Allocation Matrix

	Memory
P1	80 KB
P2	0 KB

Need Matrix

	Memory
P1	60 KB
P2	150 KB

Available vector = (120)

- ✓ Then P2 requests 70 KB, and assume it is granted

Claim Matrix

	Memory
P1	140 (80+60) KB
P2	150 (70+80) KB

Allocation Matrix

	Memory
P1	80 KB
P2	70 KB

Need Matrix

	Memory
P1	60 KB
P2	80 KB

Available vector = (50)

- ✓ P1 cannot finish and P2 cannot start and finish its execution

- ✓ Safe sequence does not exist! Deadlock will occur in the future

Recall
What is a Deadlock?

Deadlock Example : Reusable Resources

- ✓ Memory space is available for allocation of 200Kbytes, and the following sequence of events occur

P1

...

Request 80 Kbytes;

...

Request 60 Kbytes;

P2

...

Request 70 Kbytes;

...

Request 80 Kbytes;

- ✓ Deadlock occurs if both processes progress to their second request

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제23강

File

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



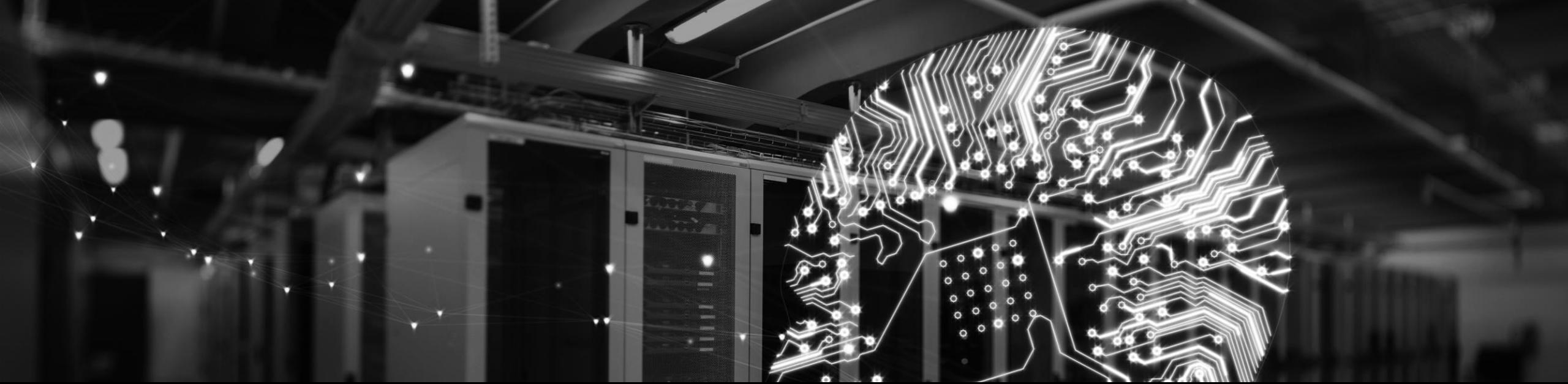
학습 목표

- 파일 속성을 이해한다.
- File Control Block 자료 구조를 이해한다.
- 장치 파일(Device file)의 의미를 이해한다.

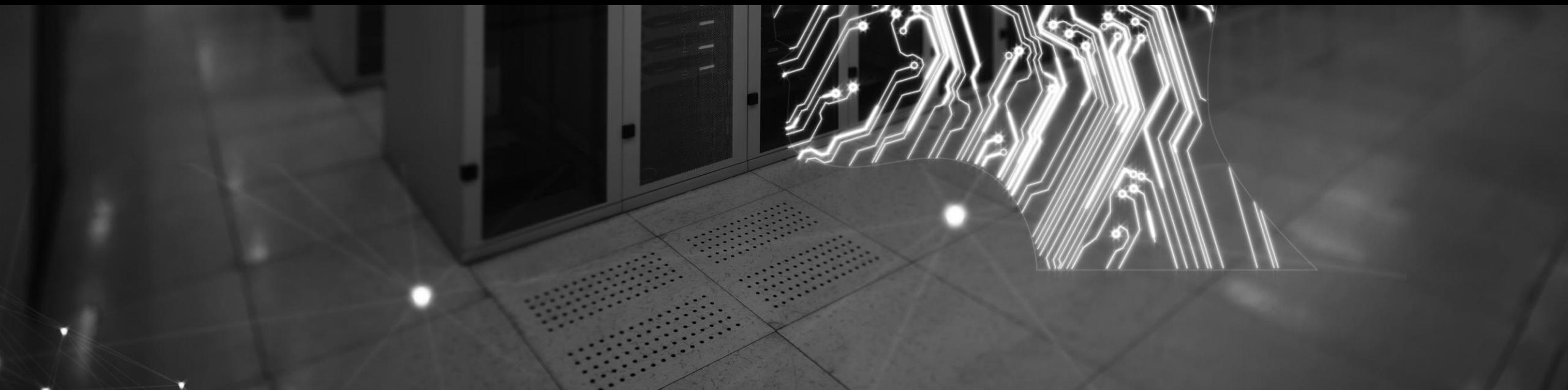
학습 내용

- 파일 속성
- File Control Block 자료 구조
- 장치 파일의 의미





File



Overview

✓ **File**

- ◎ Collection of similar records
- ◎ Treated as a single entity
- ◎ Have file names, long-term existence
- ◎ Sharable between processes, may restrict access
- ◎ File Operations : Create, Delete, Open, Close, Read, Write
- ◎ Information about the file (file attributes) is stored and managed in a data structure called File Control Block (FCB)

✓ **Input to applications is by means of a file**

✓ **Output is saved in a file for long-term storage (file system)**

✓ **Objects of file management**

- ◎ Files
- ◎ File system

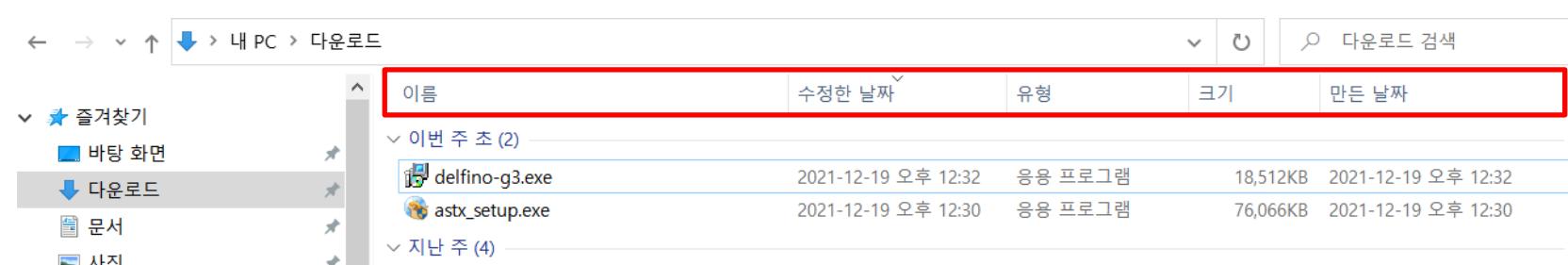
File Control Block

Kernel data structure to keep the information(attributes) about the file

◎ One FCB for each file

FCB

file name
file size (bytes)
user_id, group_id
file operation
creation time
last modified
last access time
address of file data
:
:
:



File Attributes of a FCB

Basic Information

- ◎ **File Name** Chosen by creator (user or program). Must be unique within a specific directory.
- ◎ **File Type** For example: text, binary, load module, etc.
- ◎ **File Organization** For systems that support different organizations

Address Information

- ◎ **Volume** Indicates device on which file is stored
- ◎ **Starting Address** Starting physical address on secondary storage
(e.g., cylinder, track, and block number on disk)
- ◎ **Size Used** Current size of the file in bytes, words, or blocks
- ◎ **Size Allocated** The maximum size of the file

File Attributes of a FCB

Access Control Information

- ④ **Owner** User who is assigned control of this file. The owner may be able to grant/deny access to other users and to change these privileges
- ④ **Access Information** A simple version of this element would include the user's name and password for each authorized user.
- ④ **Permitted Actions** Controls reading, writing, executing, transmitting over a network

File Attributes of a FCB

Usage Information

- ◎ **Date Created** When file was first placed in directory
- ◎ **Identity of Creator** Usually but not necessarily the current owner
- ◎ **Date Last Read Access** Date of the last time a record was read
- ◎ **Identity of Last Reader** User who did the reading
- ◎ **Date Last Modified** Date of the last update, insertion, or deletion
- ◎ **Identity of Last Modifier** User who did the modifying
- ◎ **Date of Last Backup** Date of the last time the file was backed up on another storage medium
- ◎ **Current Usage** Information about current activity on the file, such as process or processes that have the file open, whether it is locked by a process, and whether the file has been updated in main memory but not yet on disk

File Management Functions

- ✓ Identify and locate a selected file
- ✓ Use a directory(folder) to describe the location of all files plus their attributes
- ✓ Describe user access control on a shared system
- ✓ Blocking of access to files
- ✓ Allocate files to free blocks
- ✓ Manage free storage for available blocks
- ✓ Types of Files
 - ◎ Regular file, directory, device file, link, socket, pipe

File Operations

✓ System calls for files

- ◎open, close
- ◎read, write
- ◎lseek
- ◎dup
- ◎link
- ◎pipe, mkfifo
- ◎mkdir, readdir
- ◎mknod
- ◎stat
- ◎mount
- ◎sync, fsck

Device File

✓ Device file

- ◎ A special file, the way to access a device driver
- ◎ FCB of a device file has device type field and device number fields
 - » Major number to specify device types
 - » Minor number to specify device units

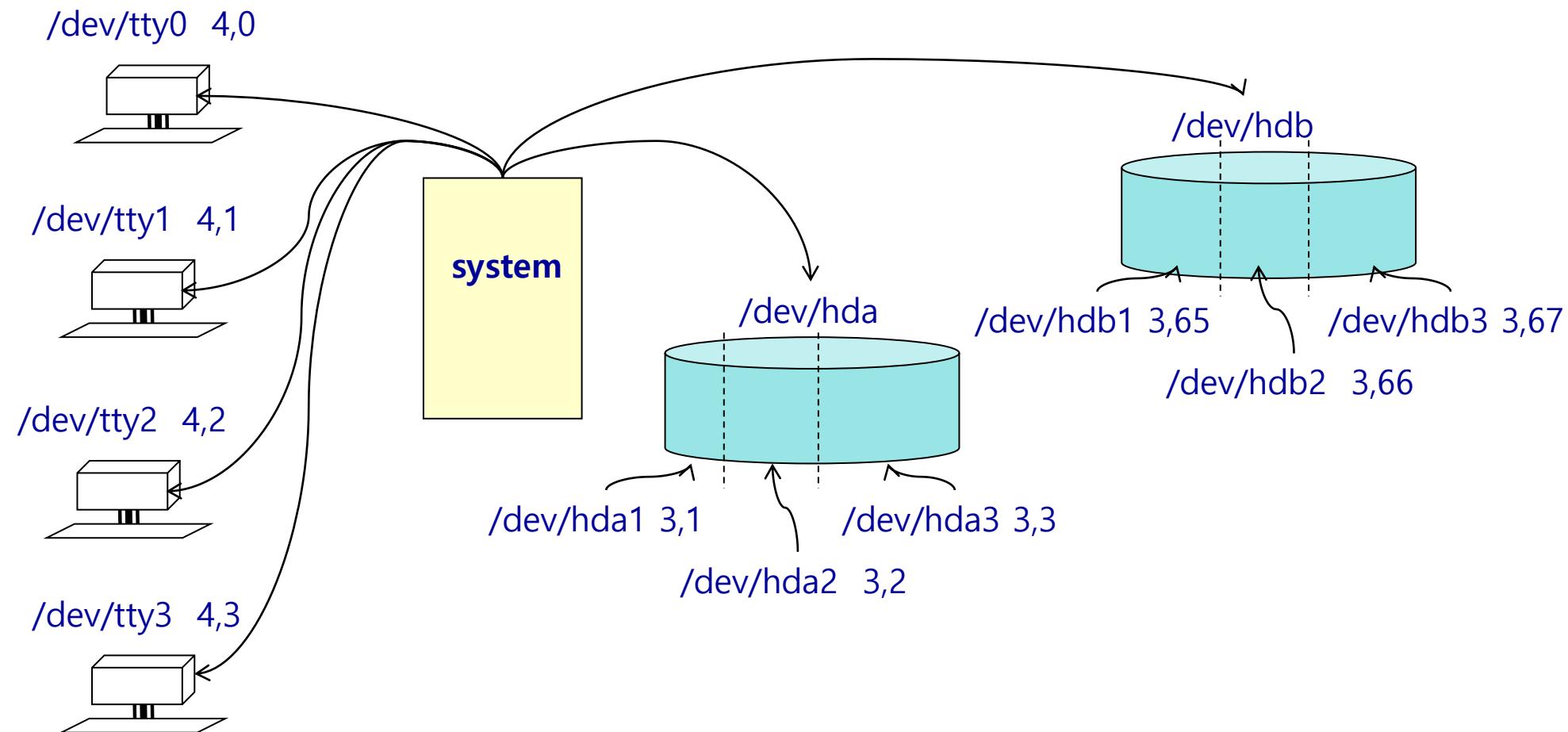
✓ Example of device files : ls -l /dev

```
...
brw-r---- 1 root disk 3 0 Oct 3 1993 hda
brw-r---- 1 root disk 3 1 Oct 3 1993 hda1
brw-r---- 1 root disk 3 2 Oct 3 1993 hda2
brw-r---- 1 root disk 3 3 Oct 3 1993 hda3
...
crw--w--w-- 1 card tty 4 0 May 16 1993 tty0
crw--w--w-- 1 card tty 4 1 May 16 1993 tty1
crw--w--w-- 1 card tty 4 2 May 16 1993 tty2
...
```

✓ Creation of a device file : use of 'mknod' command

- ◎ \$ mknod /dev/file_name [b|c] major_number minor_number

Device Numbers of Device File



Device Numbers of Device File

✓ Case study : Linux's major (device) number

◎ 0~255 /* include/linux/major.h */

Major	Character devices	Block devices
0		
1	mem	RAM disk
2		floppy (fd*)
3	terminal (PTY slave)	IDE hard disk (hd*)
4	terminal	
5	terminal & AUX	
6	Parallel Interface	
7	virtual console (vcs*)	
8		SCSI hard disk (sd*)
9	SCSI tapes (st*)	
10	Bus mice(bm, psaux)	
11		SCSI CD-ROM(scd*)
.....		
255		

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제24강

Directory

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



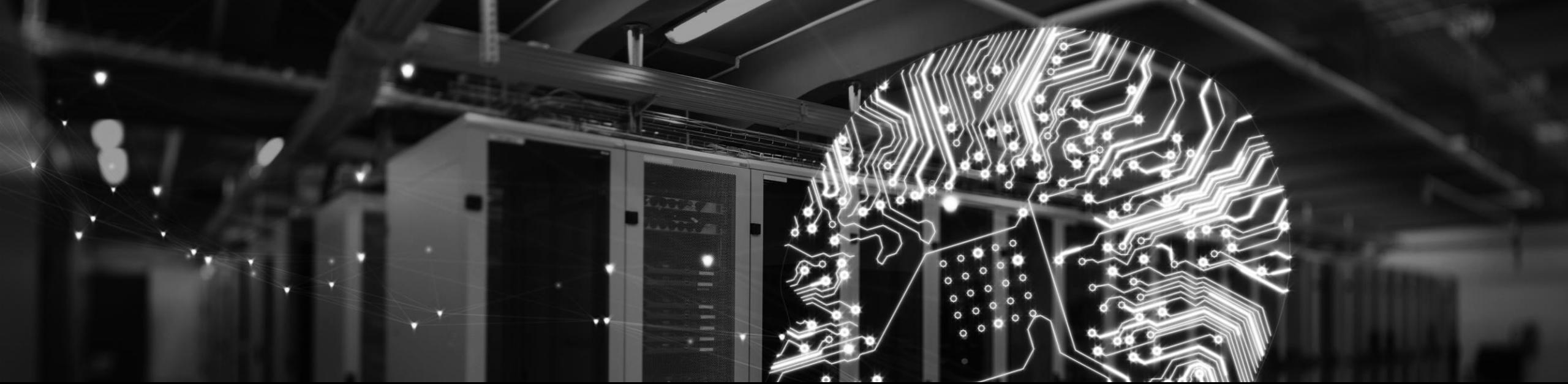
학습 목표

- 💬 디렉토리(Directory) 구조로서 계층구조 디렉토리의 정의와 탐색 방법을 이해한다.
- 💬 파일 접근 제어(Access control) 방법을 이해한다.

학습 내용

- ⌚ 계층 구조 디렉토리의 정의와 탐색 방법
- ⌚ 파일 접근 제어 방법





Directory



File Directory

Contains information about files (file attributes)

- ◎ Not the attribute information themselves, but the pointers to the file control blocks which have the attributes
- ◎ Provides mapping between file names and the files attributes

Directory itself is a file owned by the operating system

Hierarchical Directory

- Master directory with user directories underneath it**
- Each user directory may have subdirectories and files as entries**
- Files can be located by following a path from the root, or master, directory down various branches**
 - ◎ This is the pathname for the file
- Can have several files with the same file name as long as they have unique path names**
- Current directory is the working directory**
- Files are referenced relative to the working directory**

Hierarchical Directory : Example

List of directory entries, one for each file

◎ Each entry has attributes of the file

Sequential file with the name of the file serving as the key

◎ inode (index node) : FCB of Unix/Linux

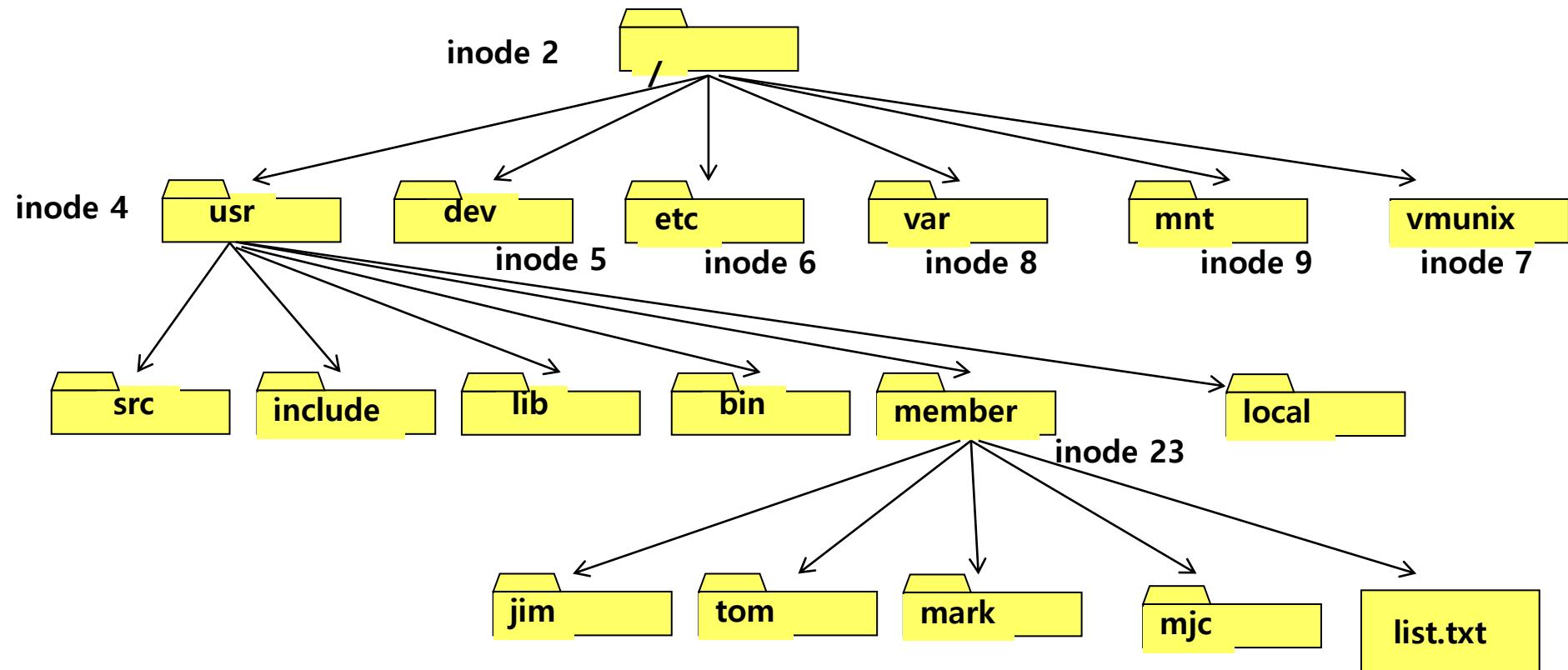
byte offset	inode number	file names
0	83	.
16	2	..
32	48	init
48	1798	fsck
64	118	temp
:	:	:

'.' is a symbol representing this directory
'..' is a symbol representing parent directory

Forces user to be careful not to use the same name for two different files

Hierarchical Directory : Example

✓ Hierarchical (tree) directory structure



Hierarchical Directory : Example

✓ UNIX/Linux directory structure

◎ inode (FCB) number of root directory : 2

» File names listed in the root directory are data of the root file

» In the following example, they are contained in the data block number 140

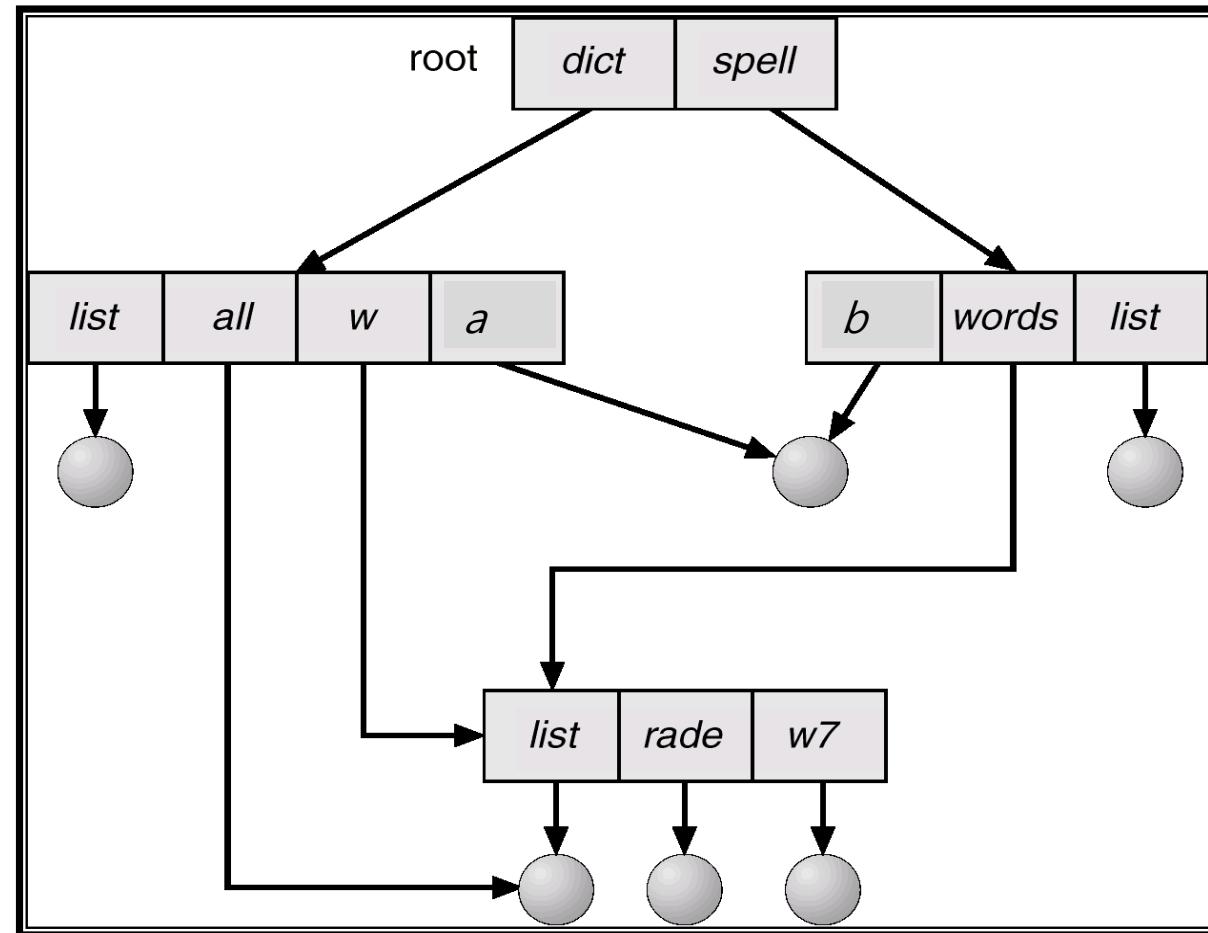
inode 2	data block 140
i_mode	2 ..
time	2 .
....	4 usr
140	5 dev
	6 etc
	7 vmunix
	8 var
	9 mnt

inode 4	data block 2700
i_mode	2 ..
time	4 .
....	12 src
2700	16 include
	17 lib
	20 bin
	23 member
	25 local

inode 23	data block 3900
i_mode	4 ..
time	23 .
....	32 jim
3900	33 tom
	37 mark
	41 mjc
	42 list.txt

Acyclic-Graph Directories

- Have shared subdirectories and files



심화

Acyclic-Graph Directories

- ✓ Two different names (aliasing)
- ✓ If “dict” deletes “a” ⇒ “spell/b” becomes dangling pointer.

✓ Solutions:

1. Leave the dangling pointers. It's up to users.
2. Entry-count solution

✓ There are other solutions, but they are not practical to use.

1. Search all the file system to remove the links to the deleted object.
2. Backpointers :
 - Keep a list of all references(links) to a file
 - When a link is established, a new entry is added to the reference list.
 - Delete all pointers when the original file is removed.
 - Variable size records a problem.

File Sharing

In a multiuser system, allow files to be shared among users

Two issues

- ⌚ Access rights
- ⌚ Management of simultaneous access

Simultaneous Access

- ⌚ User may lock entire file when it is to be updated
- ⌚ User may lock the individual records during the update
- ⌚ Mutual exclusion and deadlock are issues for shared access

Access Rights

✓ Owners

- ◎ Has all rights previously listed
- ◎ May grant rights to others using the following classes of users
 - » Specific user
 - » User groups
 - » All for public files

✓ Access Lists

- ◎ Mode of access: read, write, execute
- ◎ Three classes of users

		RWX
a) owner access	7	1 1 1
b) group access	6	1 1 0
c) public access	1	0 0 1

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제25강

File Systems

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



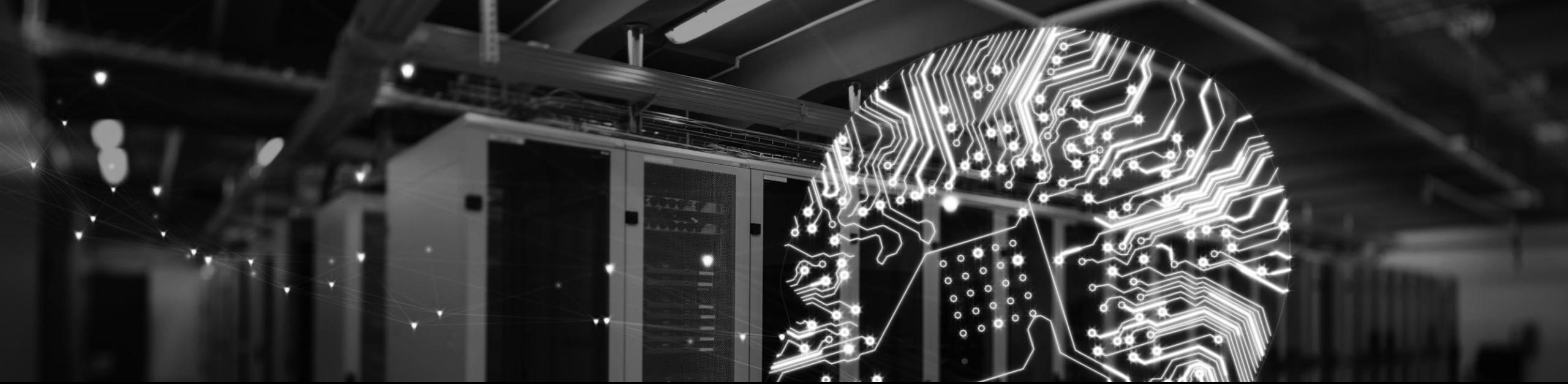
학습 목표

- 파일 시스템(File system)의 정의를 이해한다.
- 파일 시스템의 구성 정보를 이해한다.

학습 내용

- 파일 시스템의 정의
- 파일 시스템의 구성 정보





File Systems



File Systems

✓ File

- ◎ Logical storage unit
- ◎ Collection of related information

✓ File control block (FCB)

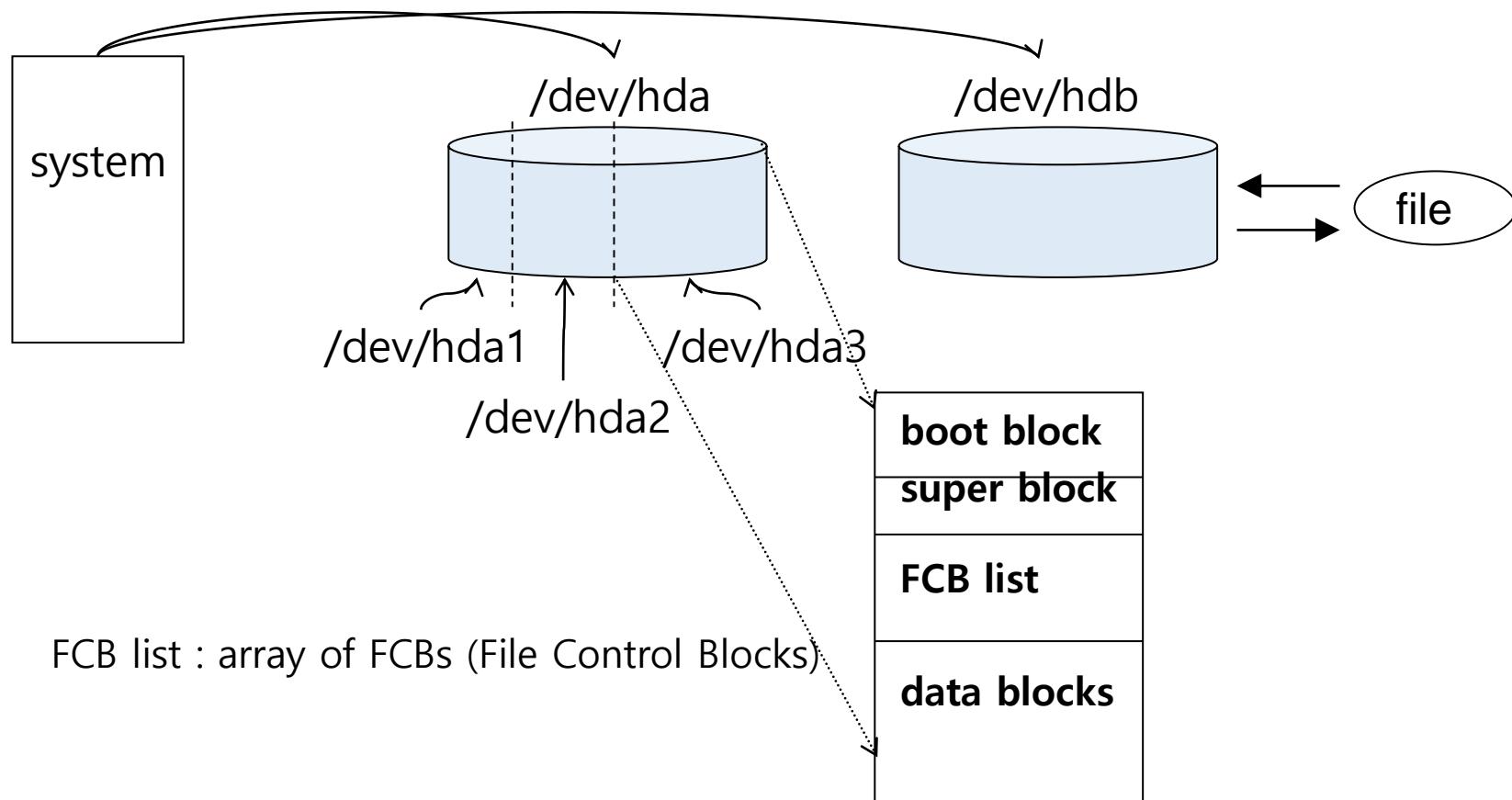
- ◎ Data structure consisting of information about a file (file attributes)
- ◎ Stored in the file system, cached in main memory
- ◎ Example) inode (index node) in UNIX, Master File Table in Windows NT

✓ File system resides on secondary storage (disks)

- ◎ Boot block
- ◎ Partition control block (superblock)
- ◎ Directory structure
- ◎ File control blocks
- ◎ Data blocks

File Systems

- ✓ The way a user of application may access files
- ✓ Programmer does not need to develop file management software

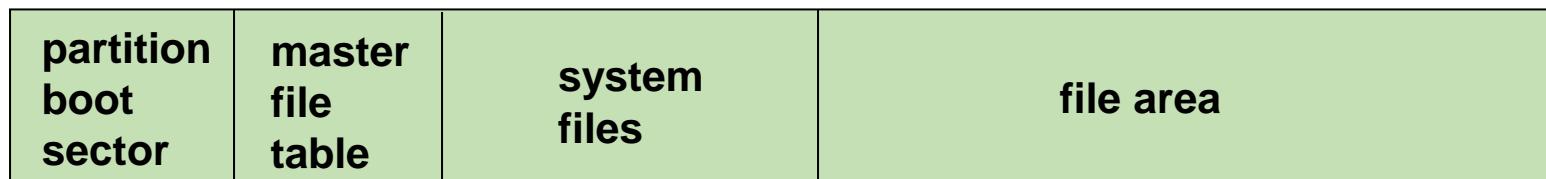


File System Structure

✓ UNIX File System

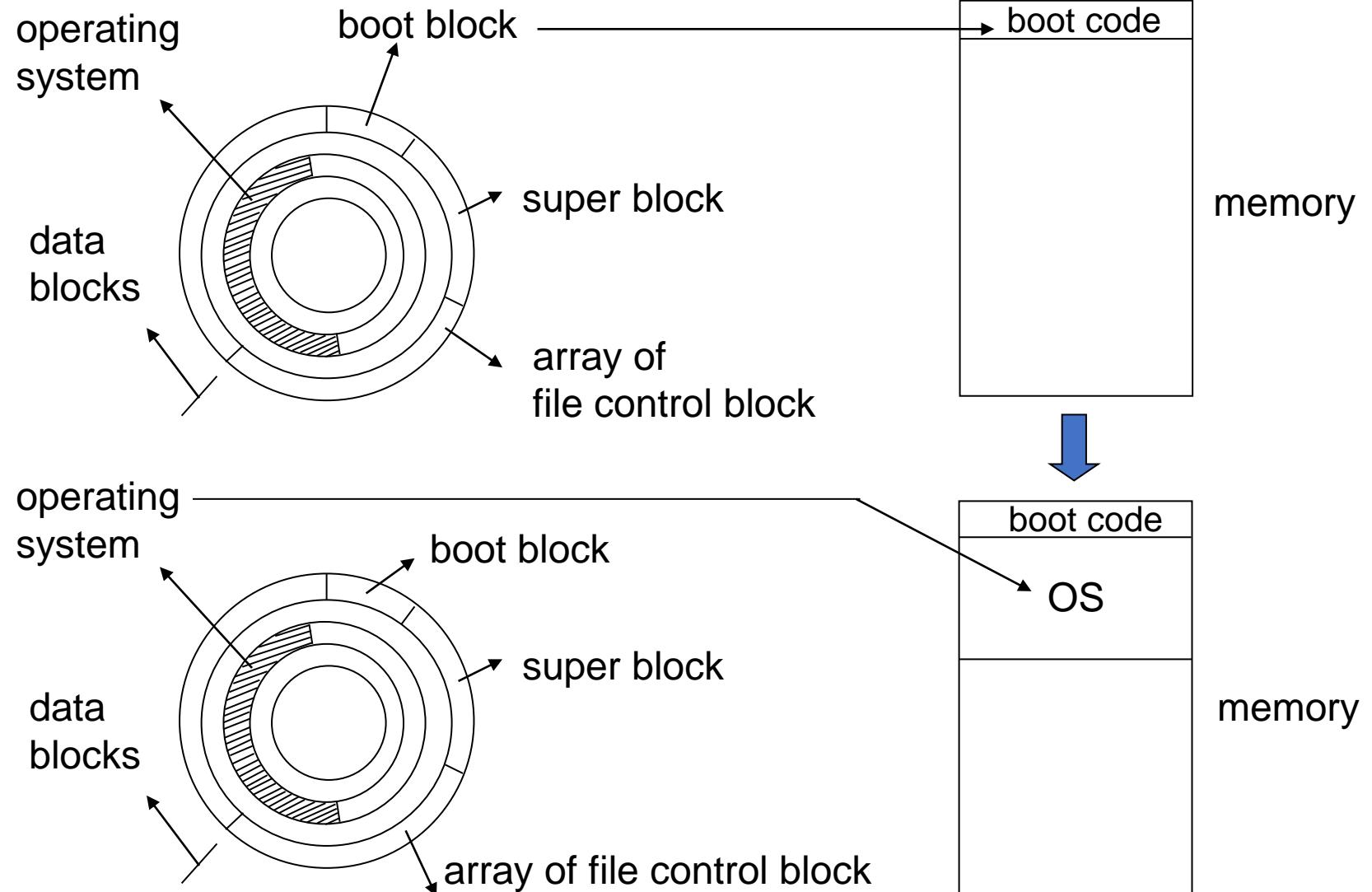


✓ Windows NTFS File System (Volume)



심화

Boot Block

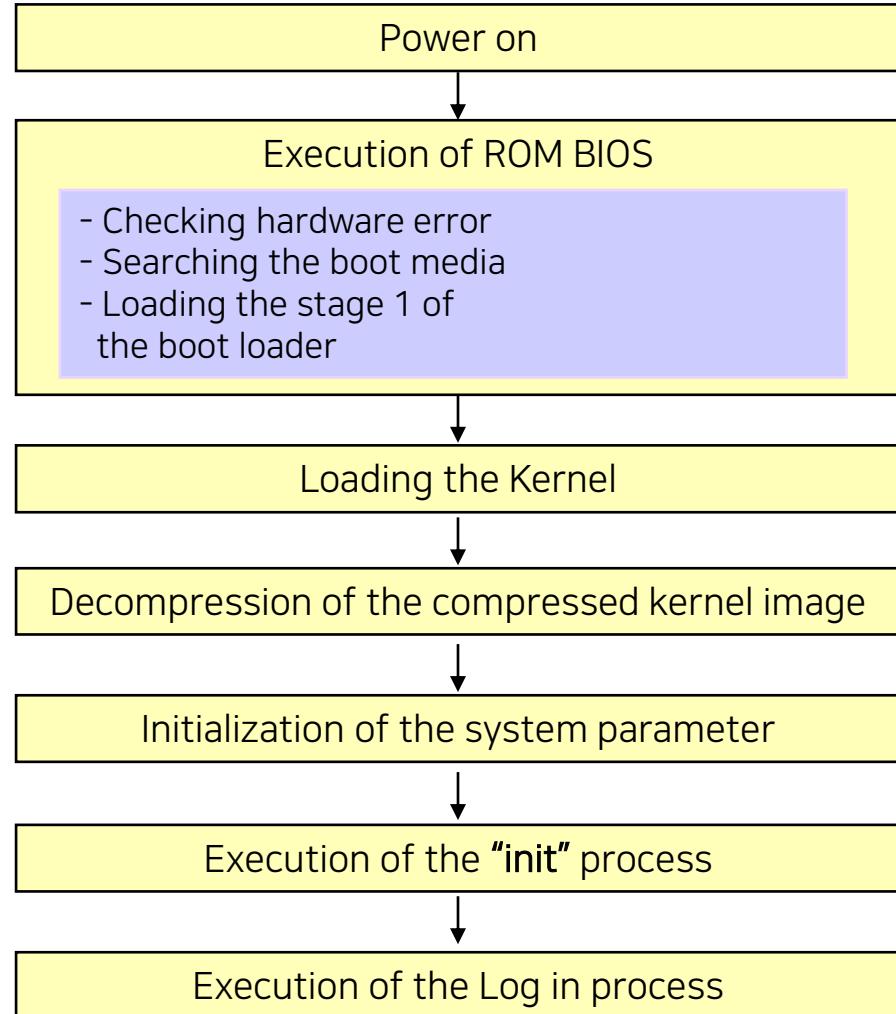


심화
Recall
Process Creation

Booting a Computer

✓ Example : Linux booting process

- ① Boot loader is located in the boot block of a hard disk
- ② Boot loader is the first program which is loaded into the main memory and executed by the processor
- ③ Boot loader loads the OS into the main memory
- ④ Then the OS executes the "init" process



Partition Control Block (Super Block)

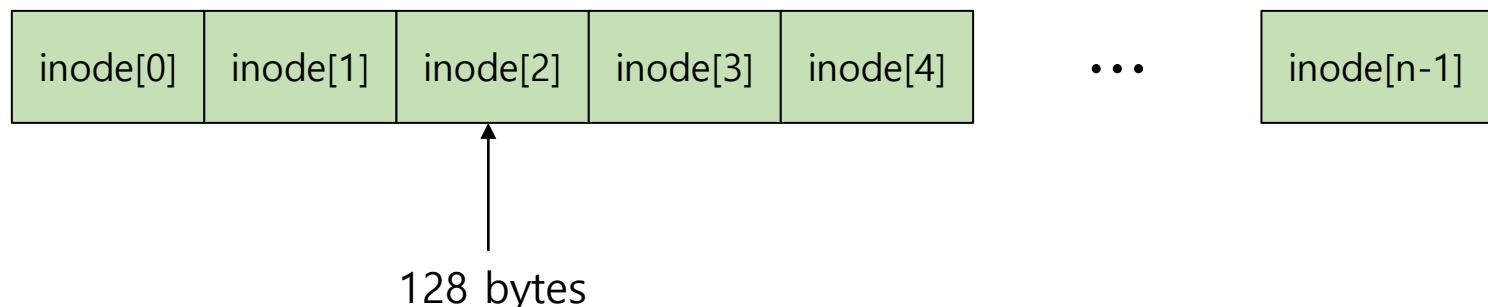
Contains information about the file system

- ④ Size(number of blocks) of the File system
- ④ Number of free data blocks in the File system
- ④ List of free data blocks of the File system
- ④ Number of inodes in the inode table
- ④ Number of free inodes in the File system
- ④ List of free inodes

File Control Blocks (Inode List)

✓ Array of inodes (FCB list)

- ◎ Size of one inode(File Control Block) is 128 bytes in Unix/Linux
- ◎ Inode list is the array of 128 byte structures
- ◎ Administrator specifies the size of the inode list when configuring a file system.
- ◎ The kernel identifies a specific inode by its index number of the inode array.
- ◎ Inode 2 is the inode of the root directory of the file system.



Recall
File

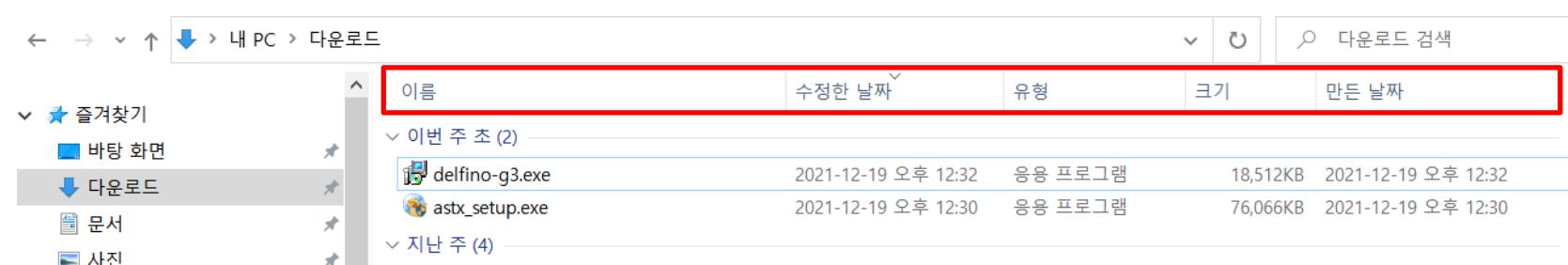
File Control Block

Kernel data structure to keep the information(attributes) about the file

◎ One FCB for each file

FCB

file name
file size (bytes)
user_id, group_id
file operation
creation time
last modified
last access time
address of file data
:
:
:



Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제18강

Critical-Section Problem :
Hardware Solution

충남대학교 인공지능학과/컴퓨터융합학부

최 훈

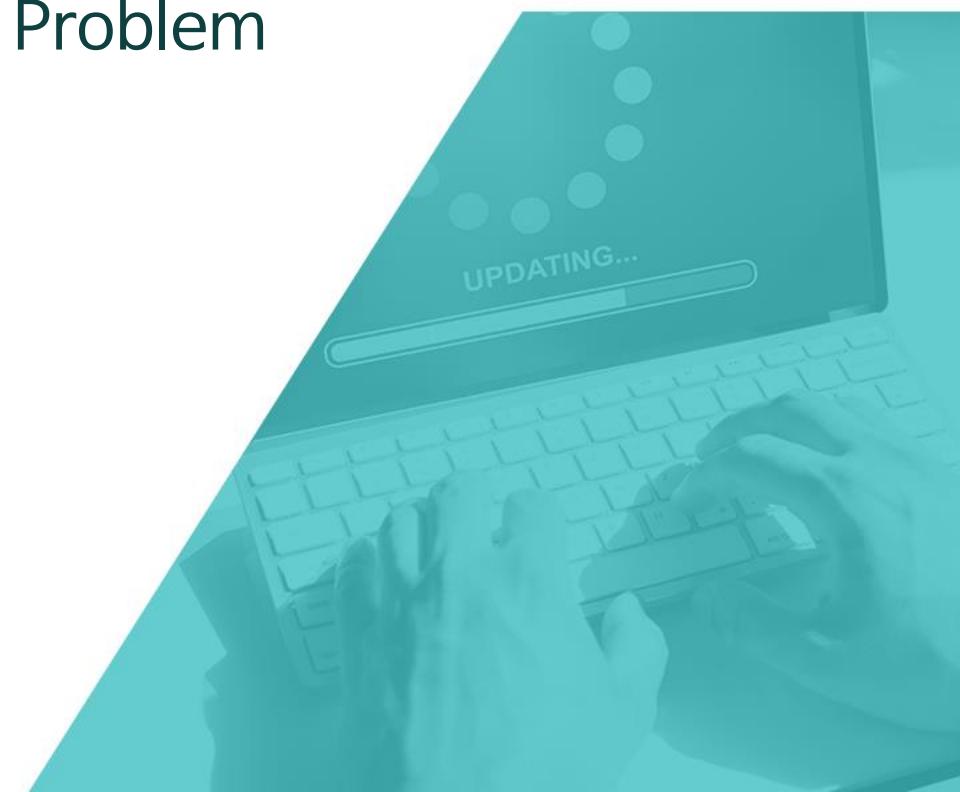


학습 목표

- ☞ 하드웨어 지원을 받는 Critical-Section Problem 해결책을 이해한다.
- ☞ 하드웨어 지원의 의미를 이해한다.
- ☞ Test and Set 인스트럭션을 이해한다.
- ☞ Test and Set 인스트럭션을 쓴 Critical-Section Problem 해결책의 장단점을 이해한다.

학습 내용

- ☞ Critical-Section Problem 해결책
- ☞ 하드웨어 지원의 의미
- ☞ Test and Set 인스트럭션
- ☞ Test and Set 인스트럭션을 쓴 Critical-Section Problem 해결책의 장단점





Critical-Section Problem : Hardware Solution



Recall
Race Condition

How Do We Prevent Race Condition?

1. Make a critical section be executed as atomic operations.
It works, but it is **not a good way**
A critical section may be lengthy and concurrency is limited

2. Make a fence around a critical section
Inside the fence, mutual exclusion is guaranteed
Inside the fence, the **critical section is not atomic**

» 2 is better way

Mutual Exclusion: Hardware Support

Interrupt Disabling

- ④ Process switch may occur when a running process invokes a system call or the process is interrupted
- ④ Disabling interrupts guarantees mutual exclusion because the current process in the critical section does not loose the CPU
- ④ But, if interrupt is disabled, concurrency is limited
 - » The efficiency of execution could be noticeably degraded

In a multiprocessor system, disabling interrupts on one processor will NOT guarantee mutual exclusion.

Recall
Race Condition

Atomic Operation

Execution of ordinary instruction (non-atomic operation)

CPU checks interrupt line
load register, counter
CPU checks interrupt line
increment register
CPU checks interrupt line
store register, counter
CPU checks interrupt line
next instruction

Execution of Atomic operation

interrupt disable
load register, counter
increment register
store register, counter
interrupt enable
CPU checks interrupt line
next instruction

Mutual Exclusion: Hardware Support

✓ Special machine instructions

✓ Test and Set instruction

```
boolean testset (i) {  
    if (i == 0) {          // i is false : no one has Key  
        i = 1;            // i is set to true (I have Key)  
        return true; // true means "I can enter."  
    }  
    else {                // if i is 1, Key is taken by other  
        return false; // false means "Do not enter!"  
    }  
}
```

✓ The whole testset function is carried out atomically (not divisible)

- ◎ Once the testset instruction(function) is begun,
- ◎ *no interruption is allowed in the middle until the function ends*
(not interrupted in the middle of execution of the function)

Test and Set Instruction

```
/* program mutual_exclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true)
    {
        while (! testset(bolt)) /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */
    }
}

void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . , P(n));
}
```

Mutual Exclusion Machine Instructions

✓ Advantages

- ① Applicable to any number of processes on either a single processor or multiple processors sharing main memory.
- ② It is simple and therefore easy to verify.
- ③ It can be used to support multiple critical sections.

Software Solution 3 : Bakery Algorithm

✓ Critical section for n processes ($n \geq 2$)

```
do {  
    choosing[i] = true;  
    number[i] = max(number[0], number[1], ..., number [n – 1])+1;  
    choosing[i] = false;  
    for (j = 0; j < n; j++) {  
        while (choosing[j]) ;  
        while ((number[j] != 0) && (number[j],j) < (number[i],i)) ;  
    }  
}
```

critical section

```
number[i] = 0;
```

remainder section

```
} while (TRUE);
```

Mutual Exclusion Machine Instructions

✓ Disadvantages

- ⌚ Busy-waiting consumes processor time
- ⌚ Bounded waiting is not supported
 - » Therefore starvation is possible when a process leaves a critical section and more than one process is waiting.

✓ Deadlock is possible

- ⌚ Suppose a low priority process PL has the critical section and
- ⌚ PL is interrupted by a higher priority process PH
 - ⌚ PH keep executing entry section to wait for the critical section
 - » busy waiting
 - » because PL is already in critical section
 - ⌚ PL will never be dispatched to finish the critical section
 - » because higher priority process PH has the processor

Mobility SW/AI Convergence 모빌리티SW/AI융합전공 **운영체제**

제20강 /

Use of Semaphores

충남대학교 인공지능학과/컴퓨터융합학부

최 훈 교수



학습 목표

- 💬 세마포어(Semaphore) 활용 사례로서 Producer-consumer problem을 이해한다.
- 💬 세마포어 활용 사례로서 프로세스간 실행 순서 제어 방법을 이해한다.
- 💬 세마포어 활용 사례로서 Readers-writers problem을 이해한다.
- 💬 Binary semaphore, Lock을 이해한다.
- 💬 모니터 개념을 이해한다.

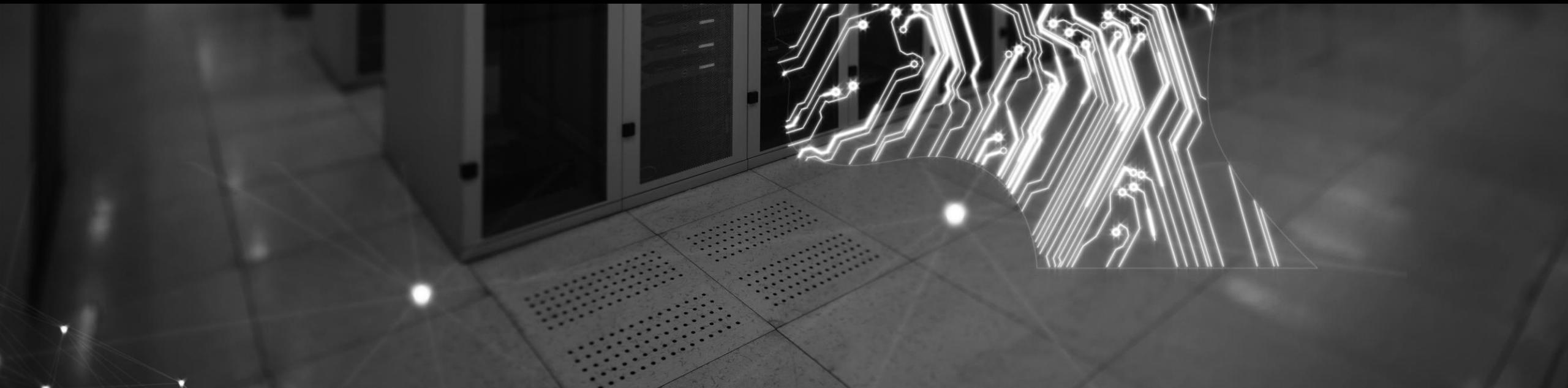
학습 내용

- ⌚ Producer-consumer problem
- ⌚ 프로세스간 실행 순서 제어 방법
- ⌚ Readers-writers problem
- ⌚ Binary semaphore, Lock
- ⌚ 모니터 개념





Use of Semaphores



Using Semaphores for Critical Section Problem

```
/* program producerconsumer */
semaphore counter = 0;
semaphore empty = BUFFER_SIZE;
semaphore mut_ex = 1;
void producer()
{
    while (true)
    {
        produce();
        semWait(empty);
        semWait(mut_ex);
        append(); /* put the produced item into the buffer */
        semSignal(mut_ex);
        semSignal(counter);
    }
}
```

Using Semaphores for Critical Section Problem

```
void consumer()
{
    while (true)
    {
        semWait(counter);
        semWait(mut_ex);
        take();           /* get an item from the buffer */
        semSignal(mut_ex);
        semSignal(empty);
        consume();
    }
}

void main()
{
parbegin (producer1, producer2,..., consumer);
}
```

Recall
Race Condition

Producer/Consumer : Finite Buffer

✓ Producer process :

```
item v;
while (1) {
    while(counter == BUFFER_SIZE); /* full, then do nothing */
    /* produce item v */
    b[in] = v;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

✓ Consumer process

```
item w;
while (1) {
    while (counter == 0); /* empty, then do nothing */
    w = b[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume item w */
}
```

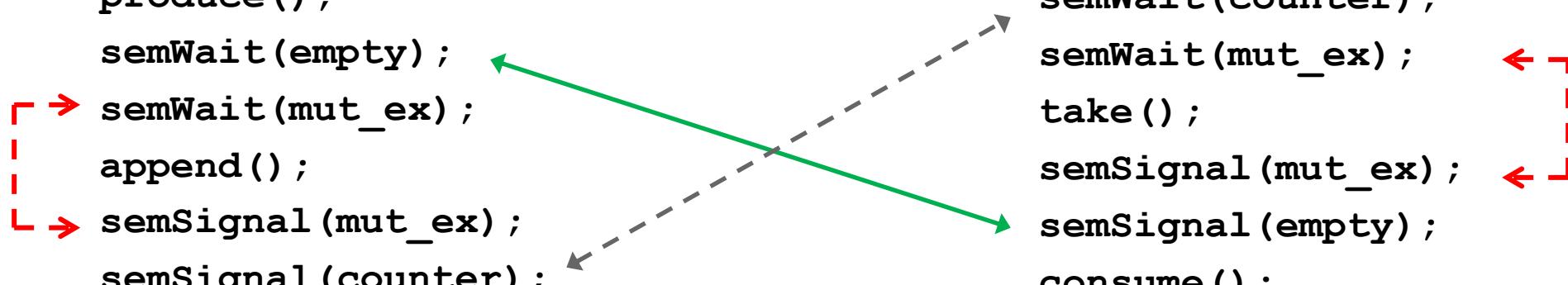
Pairing Semaphores

✓ Producer

```
while (true)
{
    produce();
    semWait(empty);
    [→ semWait(mutex);
    append();
    [→ semSignal(mutex);
    semSignal(counter);
}
```

✓ Consumer

```
while (true)
{
    semWait(counter);
    semWait(mutex);
    take();
    semSignal(mutex);
    semSignal(empty);
    consume();
}
```



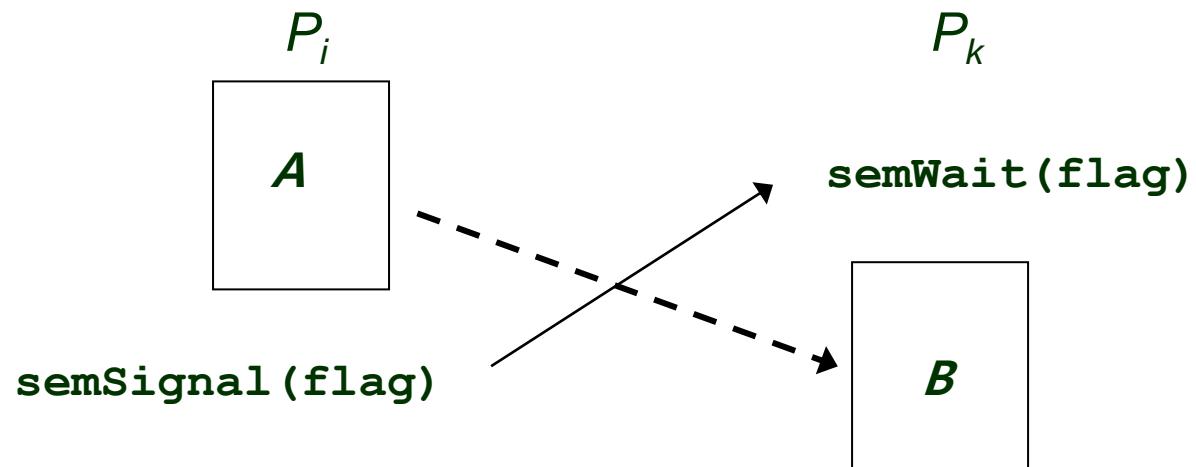
Using Semaphore for General Synchronization Tool

✓ Execute B in P_k only after A of P_i is executed

◎ Examples : writing into a shared memory (A), reading from a shared memory (B)

✓ Use semaphore *flag* initialized to 0

✓ Code :



Always execute A first, then B

Using Semaphore for Readers/Writers Problem

- ✓ Extension of a critical section problem
- ✓ Any number of readers may simultaneously read the file or data
- ✓ If a writer is writing to the file, no other (neither reader nor writer) can access the file

```
/* program readersandwriters */
int readcount;
semaphore rsem = 1, wsem = 1;

void main()
{
    readcount = 0;
    parbegin (reader1, reader2, ... writer1,writer2,...);
}
```

Writer Process

```
void writer()
{
    while (true)
    {
        semWait (wsem) ;      /* wsem : writer semaphore */
        WRITEUNIT () ;
        semSignal (wsem) ;
    }
}
```

Reader Process

```
void reader()
{
    while (true)
    {
        →semWait(rsem);      /* rsem : reader semaphore */
        readcount++;
        if (readcount == 1) semWait(wsem);
        →semSignal(rsem);

        READUNIT();

        →semWait(rsem);
        readcount--;
        if (readcount == 0) semSignal(wsem);
        →semSignal(rsem);
    }
}
```



Binary Semaphore, Lock

✓ Binary semaphores

- ◎ A type of semaphores which are restricted to the values 0 and 1
- ◎ 0 means not available and 1 means available
- ◎ All process which access the unfree critical section become blocked
- ◎ All processes wake up(changed to ready state) when the critical section becomes free → bounded waiting is not guaranteed
- ◎ Also called a mutual-exclusion semaphore (*mutex*)

✓ Lock

- ◎ A synchronization mechanism implemented by a binary semaphore

✓ Example of a lock

- ◎ C++ mutex, pthread_mutex of Linux Thread library
- ◎ pthread_mutex guarantees bounded waiting : if one or more threads are waiting to lock the mutex, pthread_mutex_unlock() causes one of those threads to return from pthread_mutex_lock()

심화

Monitors

✓ Monitor is a software module

- ◎ High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

✓ Main characteristics

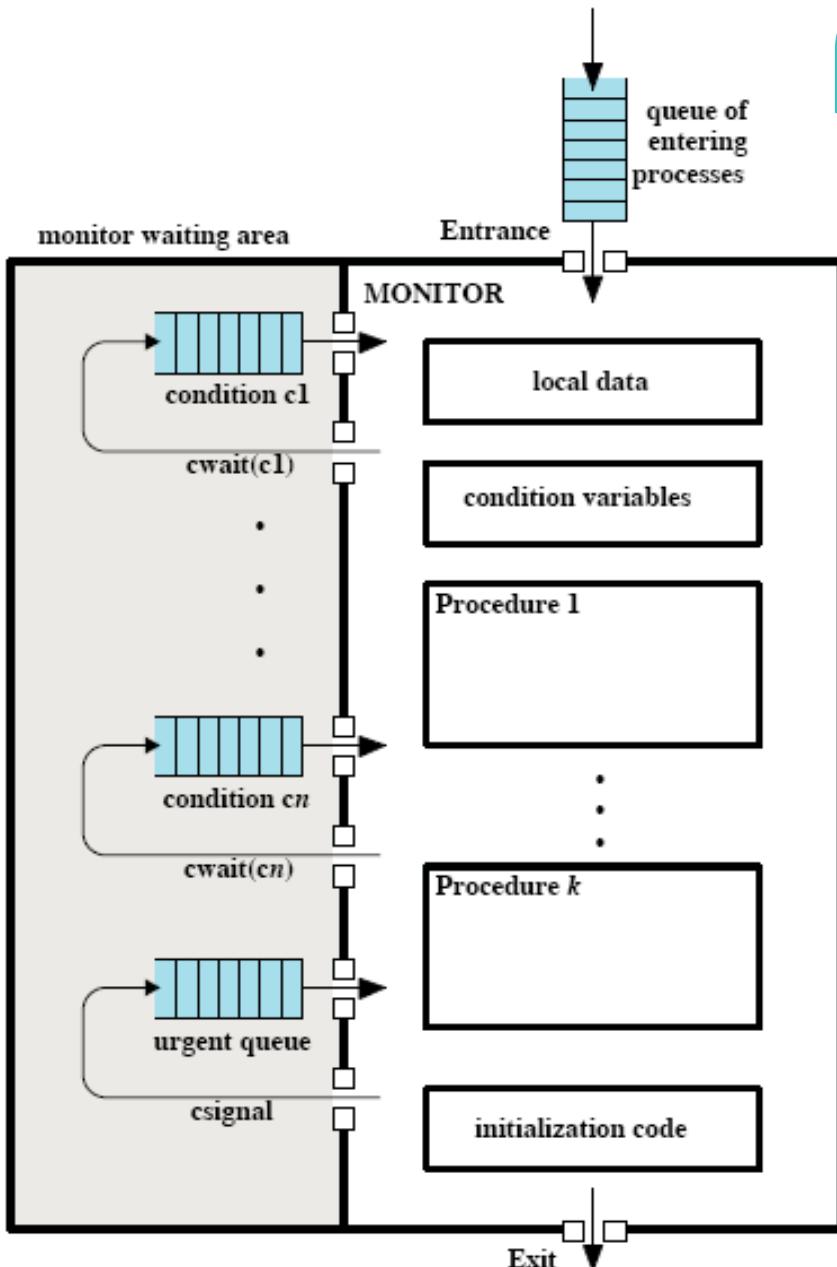
- ◎ Local data variables are accessible only by the monitor
- ◎ Process enters monitor by invoking one of monitor's procedures
- ◎ Only one process may be executing in the monitor at a time

심화

Monitors

```

monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        ...
    }
    procedure body P2 (...) {
        ...
    }
    procedure body Pn (...) {
        ...
    }
    {
        initialization code
    }
}
  
```



심화

Monitor Example

심화

Monitor Example

```
void producer()
{
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}
void consumer()
{
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor