3. Processes

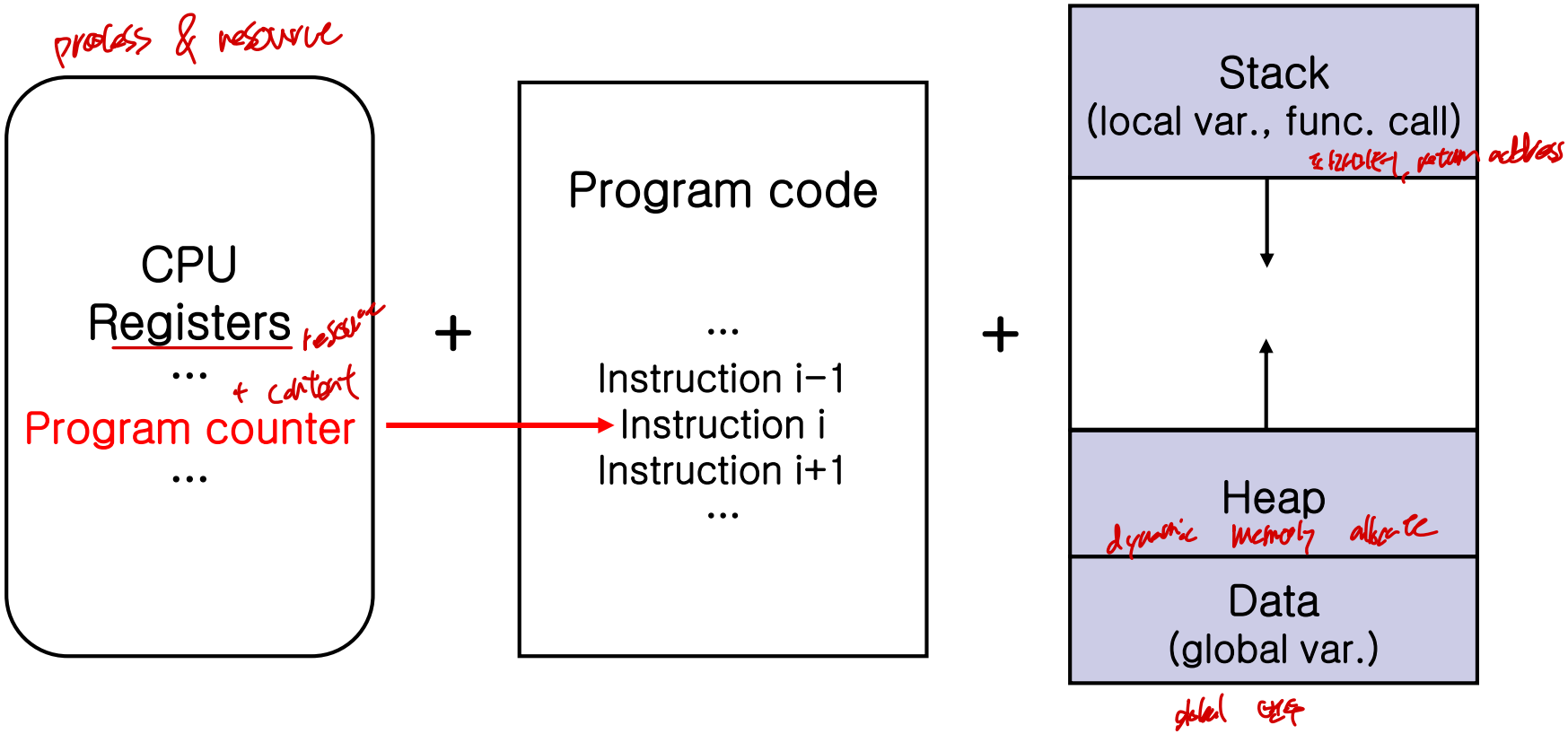
[ECE30021/ITP30002] Operating Systems

Agenda

- Overview
- Process scheduling
- Operations on processes
- Inter-process communication
- Example of IPC system
- Communication in client-server systems

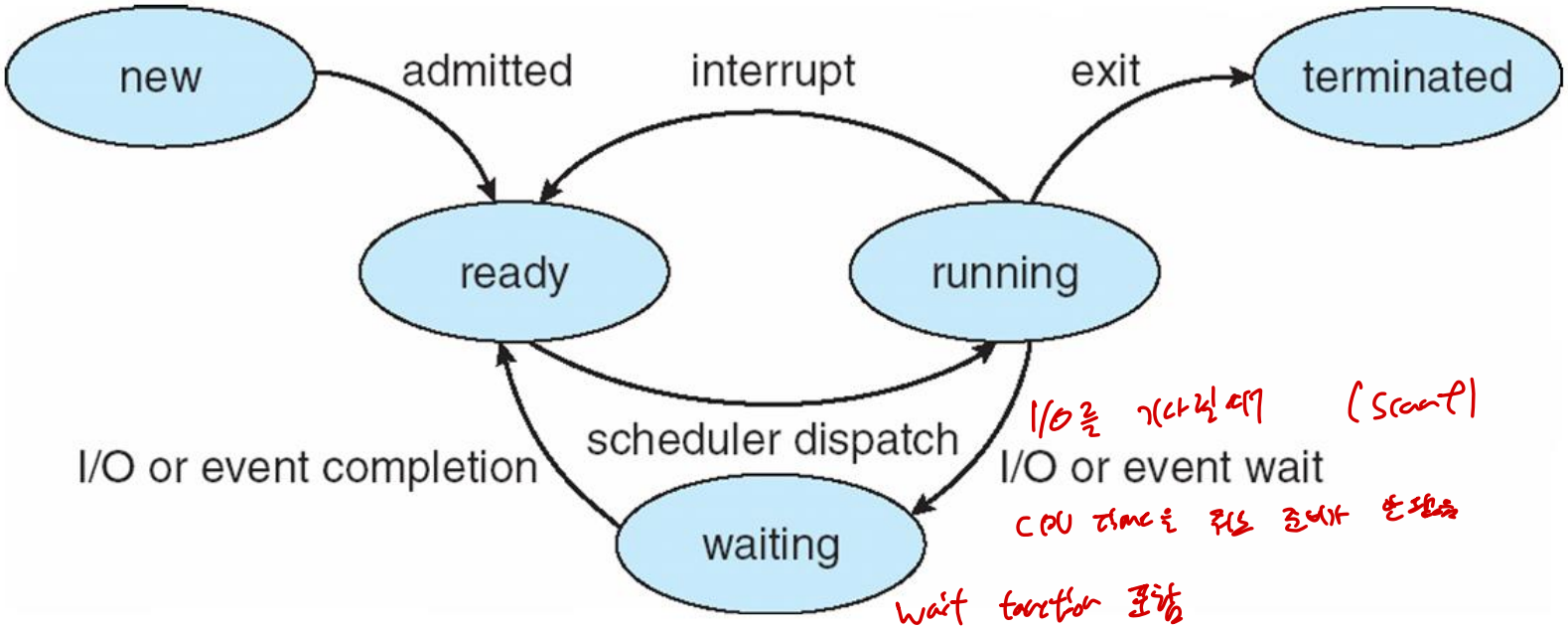
Process

- Process = program in execution + resource

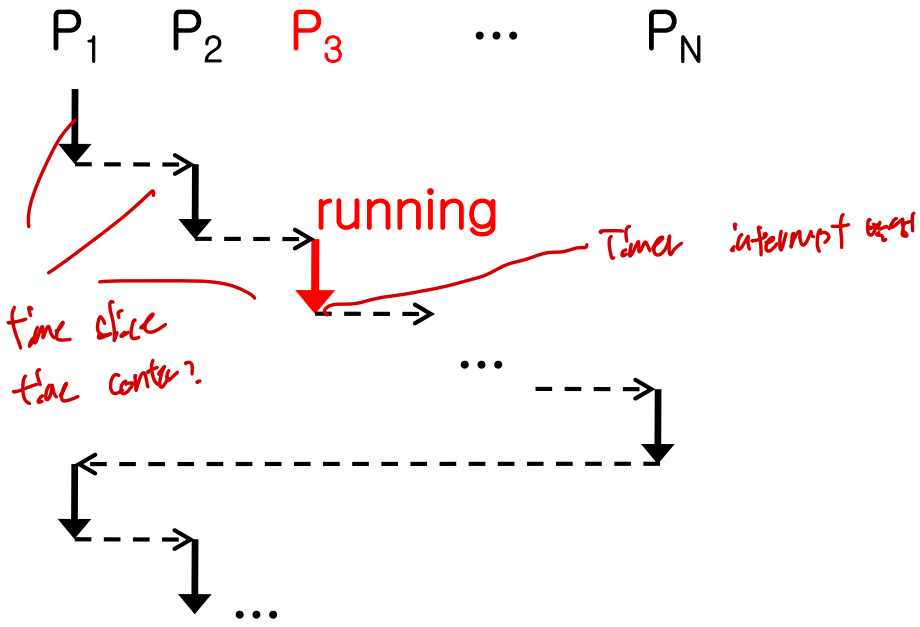


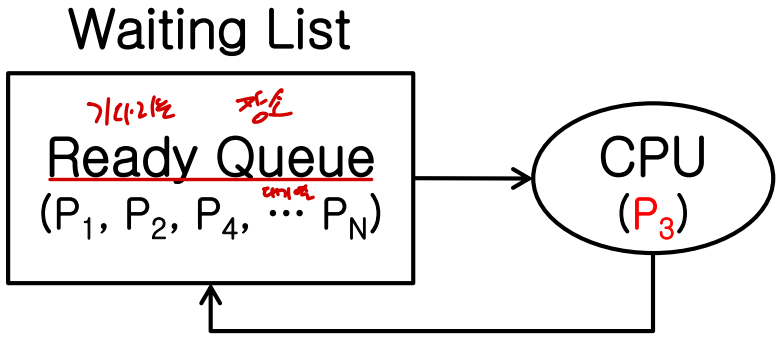
Process State

- **New**: being created
- **Running**: in execution
  - Only one process can be running on a processor at any time
- **Ready**: waiting to be assigned to a processor
- **Waiting**: waiting for some event to occur
- **Terminated**:



Ready/Running State





프로세스의 모든 정보 저장: 주민등록 등본, 학생기록부 같은거

Process Control Block (PCB)

- OS manages processes using PCB
  - Process Control Block (PCB): repository for any information about process

Contents	Examples
Process state	new, ready, running, waiting, terminated, …
Process number	pid (Process ID) unique number, 중복 X
CPU Registers	program counter (address of next instruction to execute) accumulator, general registers, stack pointer, …
CPU Scheduling info.	priority, pointer to queue, …
Memory-management info.	base and limit registers, page/segment table, …
Accounting info.	CPU-time used, time limits, account #, …
I/O status info.	List of open files, I/O devices allocated

Agenda

- Overview
- Process scheduling
- Operations on processes
- Inter-process communication
- Example of IPC system
- Communication in client-server systems

Process Scheduling

기다리는 여러개의 task들 중에 resource를 할당할 task를 선택

- **Scheduling**: assigning tasks to a set of resources

  제한되어있음

  - 운영체제 : 먼저온애

- **Process scheduling**: selecting a process to execute on CPU
  - Only one process can run on each processor at a time.
  - Other processes should wait

- **Objectives** of scheduling
  - Maximize CPU utilization CPU 최대한 활용 => CPU 오래쓰는걸 원함.
  - Users can interact with each program

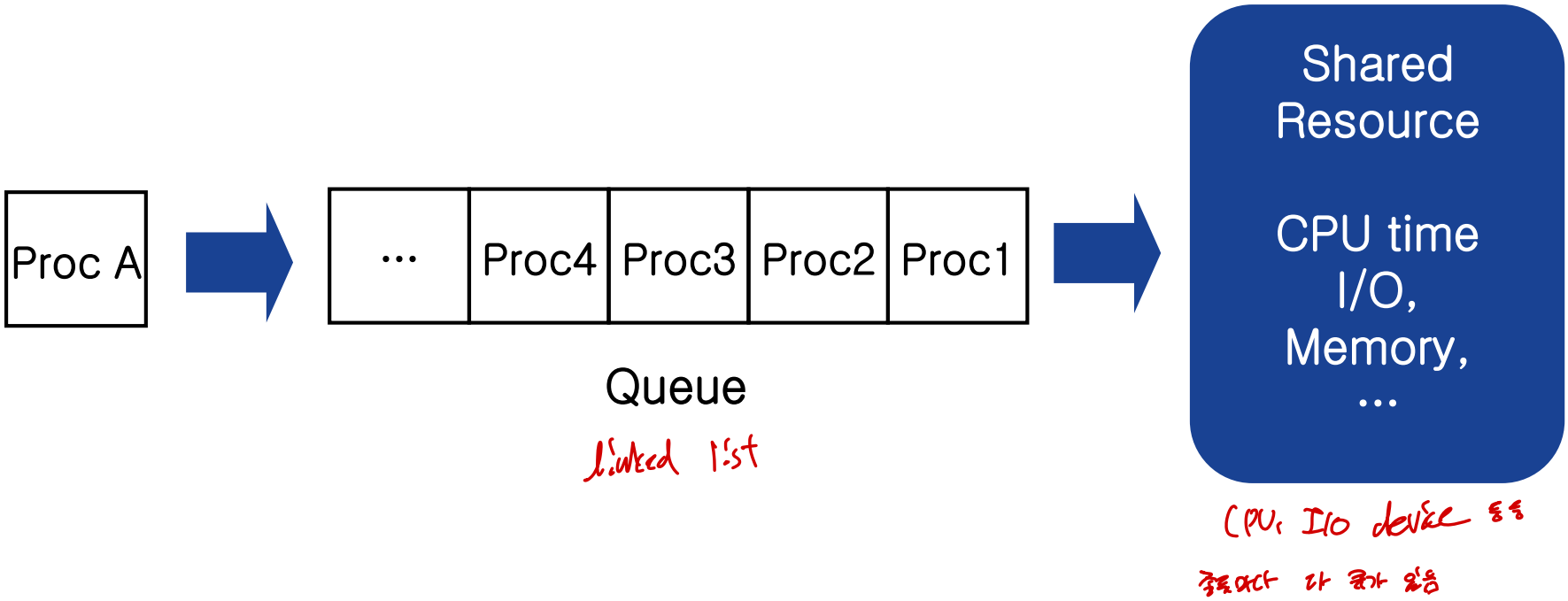
스위칭 과정에서 : performance ↓, 반응성 ↑

스위칭 자주 X : performance ↑, 반응성 ↓

리얼타임 시스템은 반응성이 중요하다.

Scheduling Queue

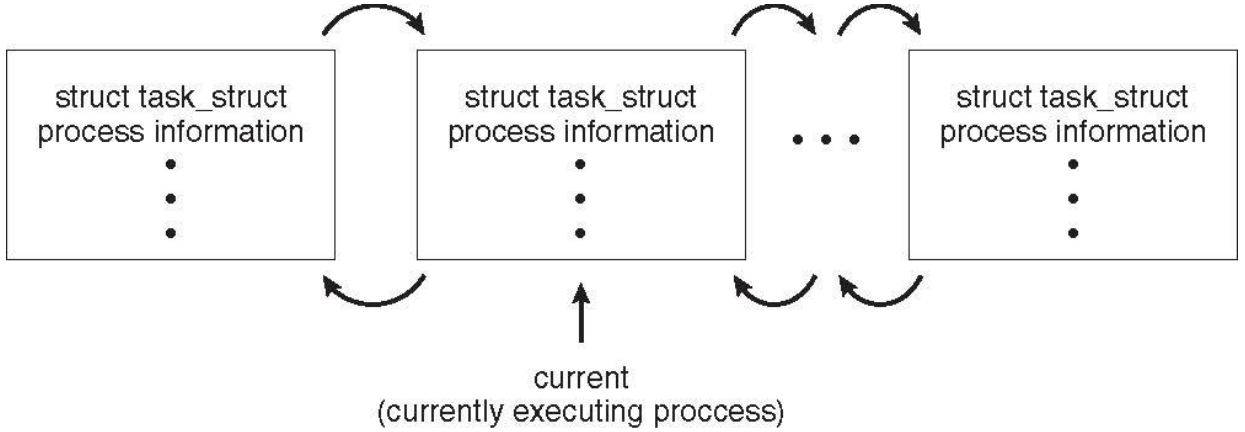
- Scheduling queue: waiting list of processes for CPU time or other resources.
  - Ready queue, job queue, device queue



PCB in Linux

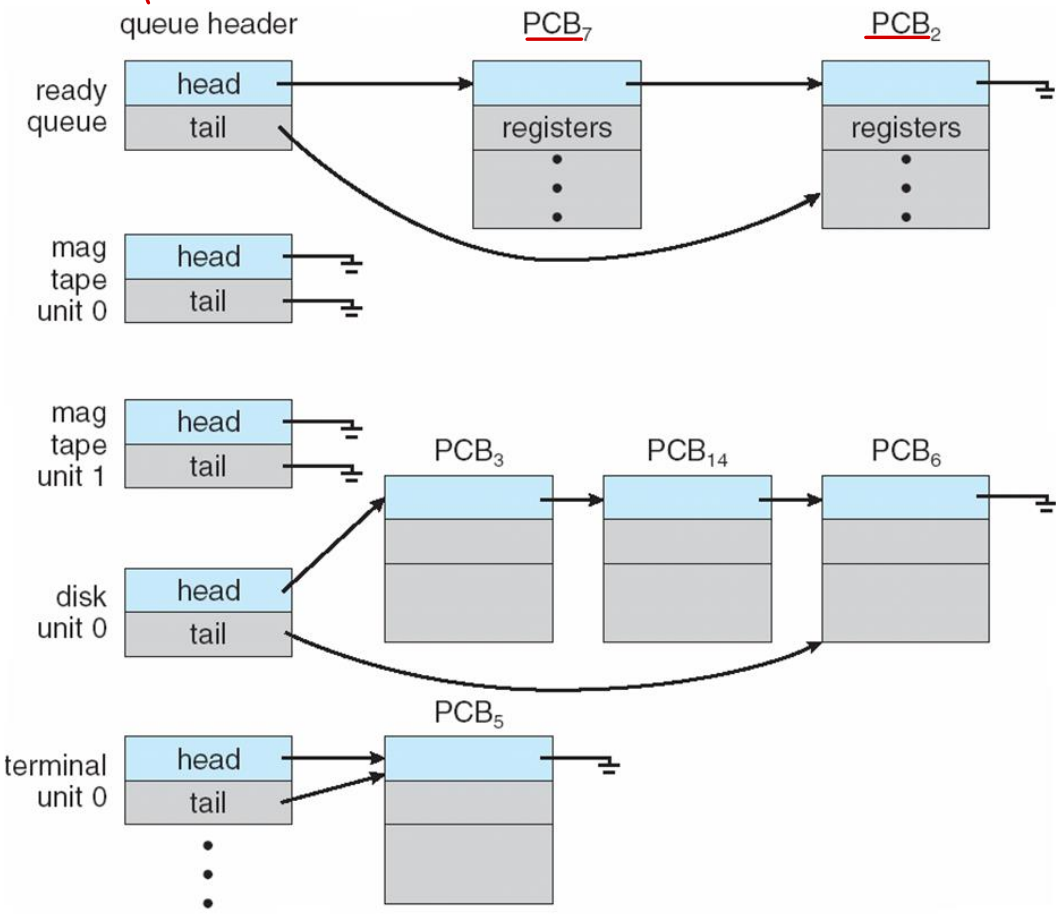
- C Structure task_struct

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



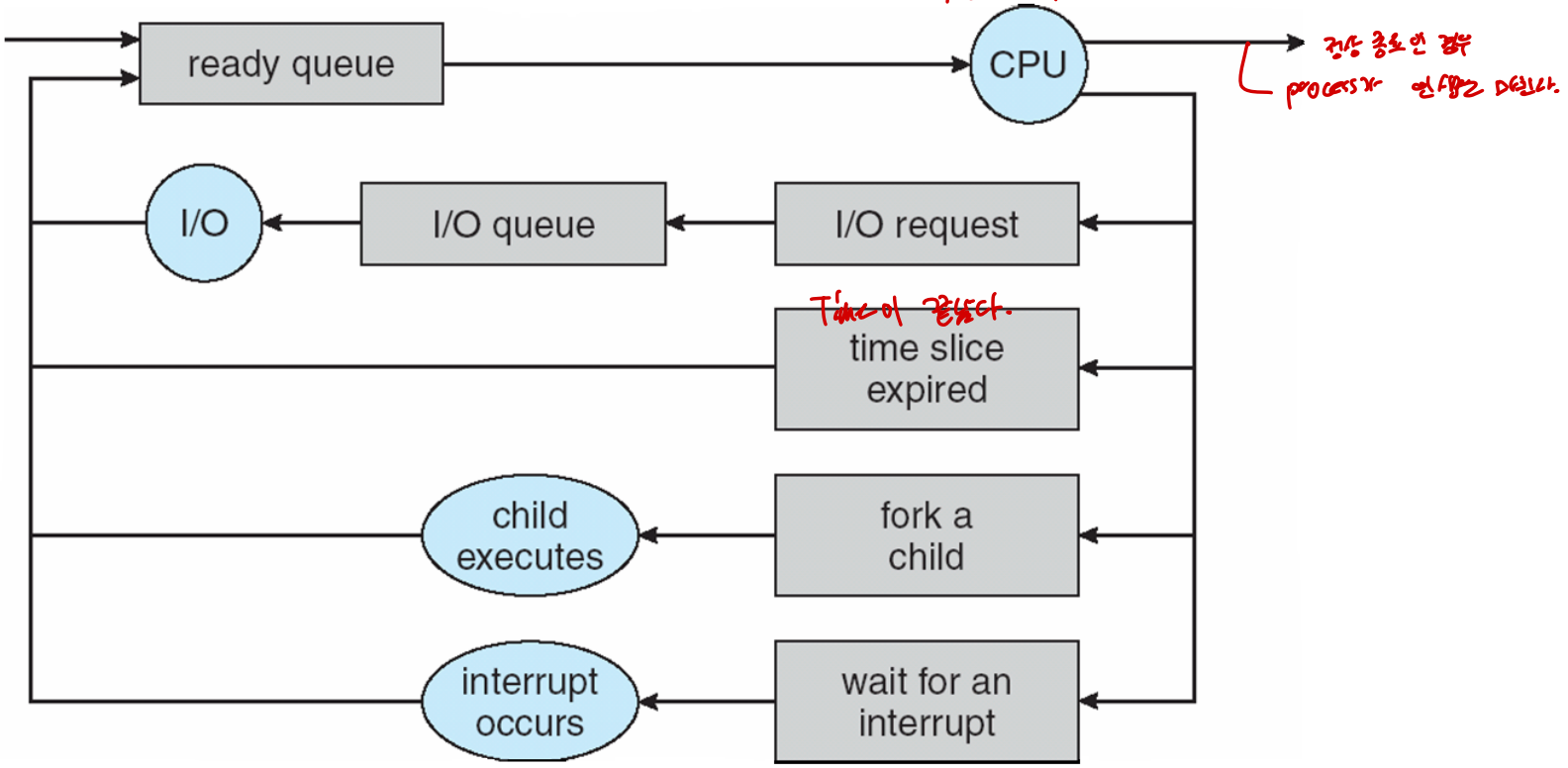
Scheduling Queue

- Each queue is usually represented by a linked list of PCBs



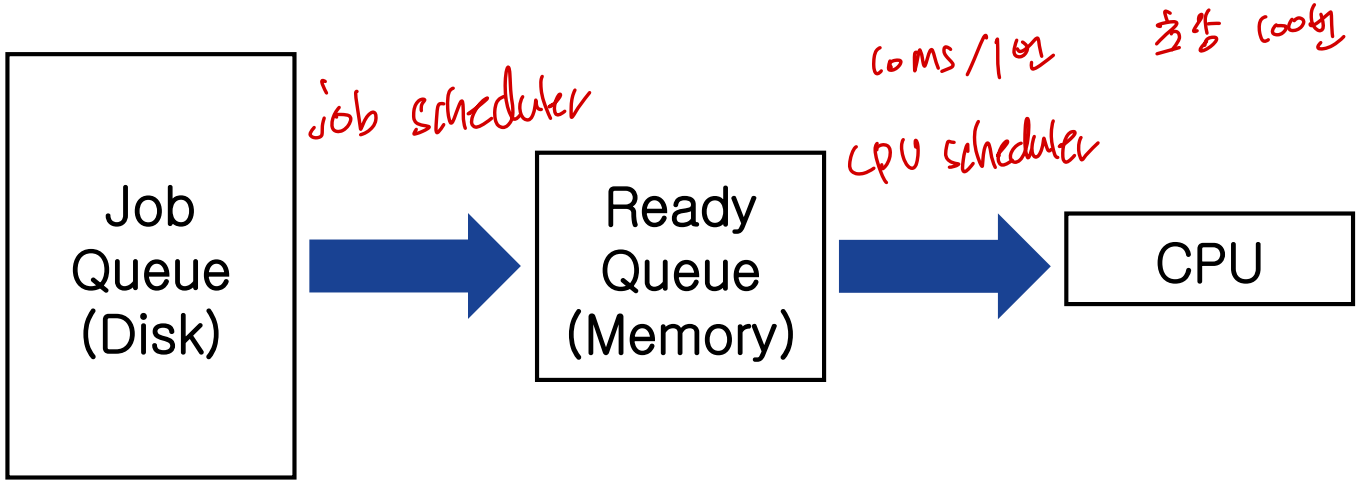
Queueing Diagram

- Representation of process scheduling
  - A process migrates among various scheduling queues throughout its lifetime



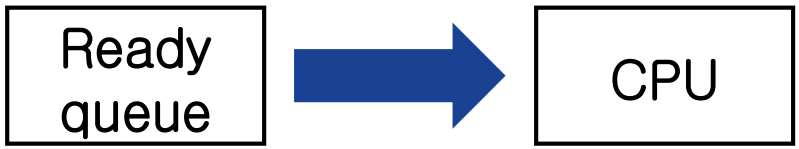
Schedulers

- Scheduler selects processes from queues in some fashion.
  - Long-term scheduler (job scheduler)
  - Short-term scheduler (CPU scheduler)



Schedulers

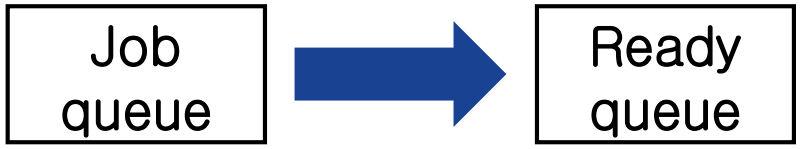
- Short-term scheduler (CPU scheduler)



  - Executed frequently (at least once every 100 msec.).
  - Scheduling time should be very short.

Schedulers

- Long-term scheduler (job scheduler)

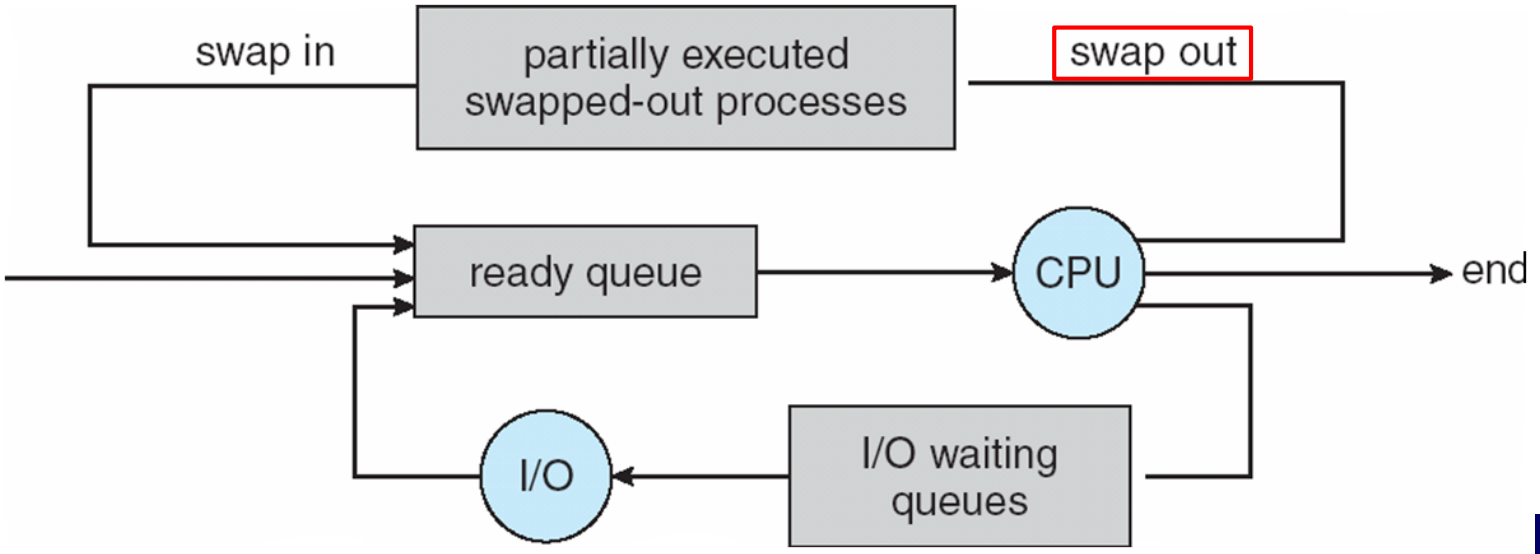


  - Controls degree of multiprogramming
    - In stable state, average process creation rate == average process departure rate
  - Executed less frequently
    - Executed only when a process leaves the system
  - Hopefully, long-term scheduler should select a good mix of I/O-bound and CPU-bound processes

Handong Global University

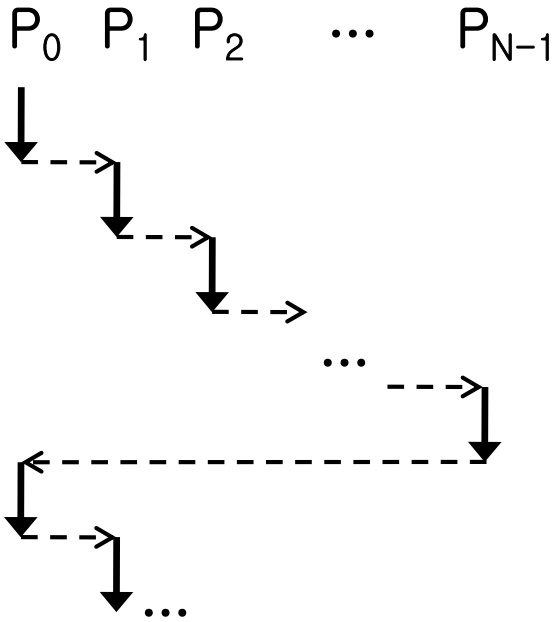
Schedulers

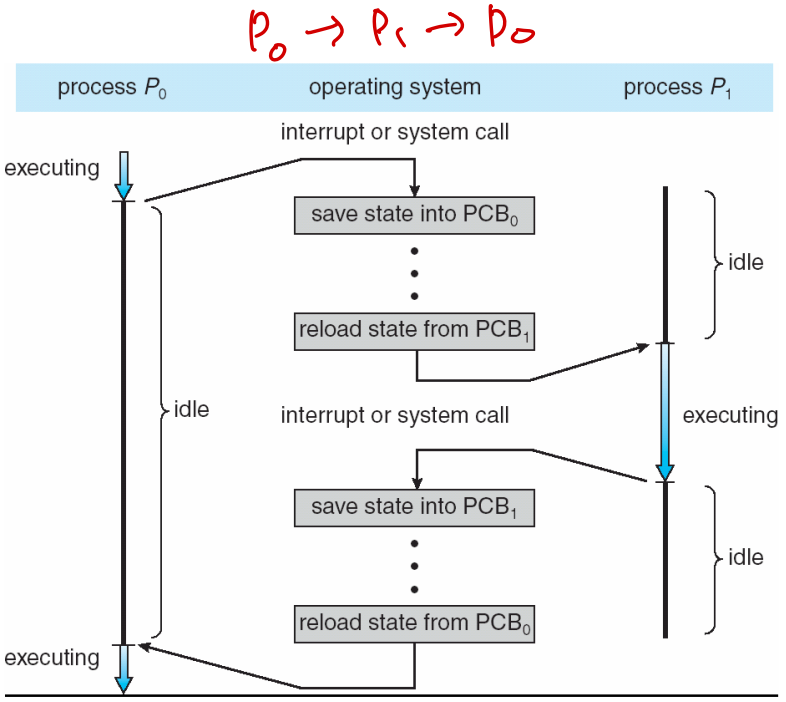
- In some systems, long-term scheduler may be absent or minimal
  Ex) UNIX, Windows
  - System stability depends on physical limitation or self-adjusting nature of human
- Some time-sharing system has medium-term scheduler
  - Reduce degree of multiprogramming by removing processes from memory



Context Switch

- Switching running process requires context switch
  - Save state (context) of current process (PCB)
  - Restore state (context) of the next process





Context Switch

- Context switch: the computing process of storing and restoring the state (context) of a CPU such that multiple processes can share a single CPU resource.

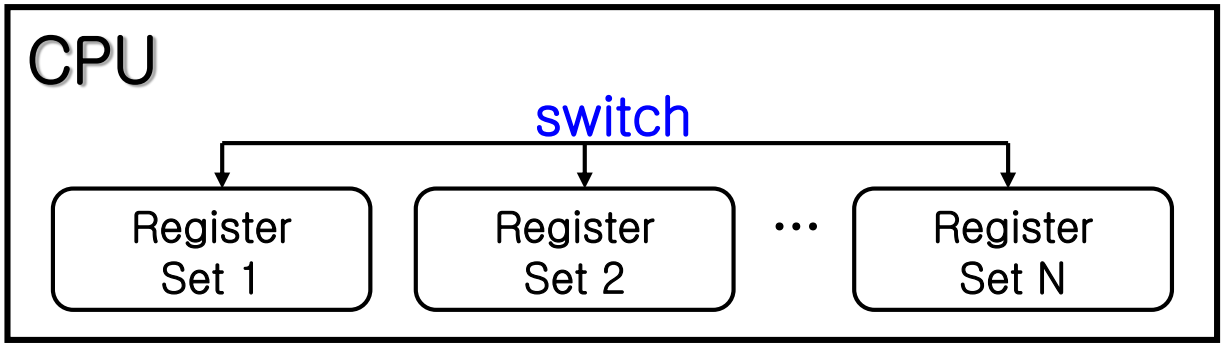
- "Context" includes
  - Register contents
  - OS specific data
    - Extra data required by advanced memory-management technique

    Ex) page table, segment table, …

- When to switch?
  - Multitasking
  - Interrupt handling

Context Switch

- Context switching requires considerable overhead.
- H/W supports for context-switching
  - H/W switching (eg. single instruction to load/save all registers)
    cf. However, S/W switching can be more selective and save only that portion that actually needs to be saved and reloaded.
  - Multiple set of register for fast switching
    Ex) UltraSPARC

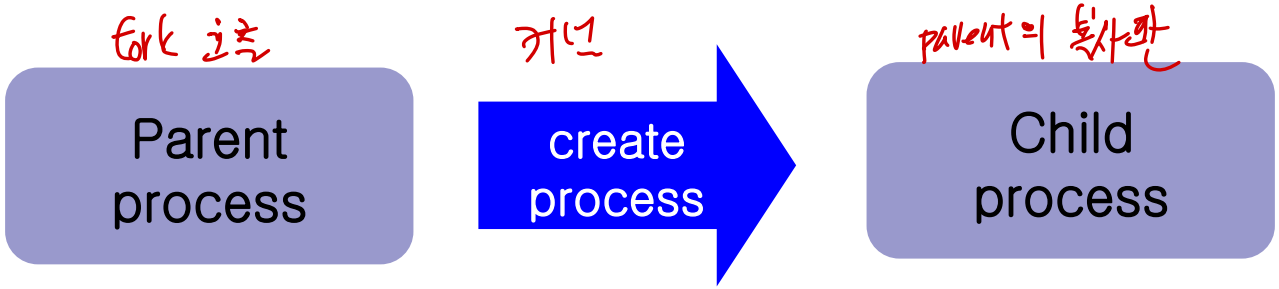


Operations on Processes

- Process create
- Process termination
- Process communication

Process Creation

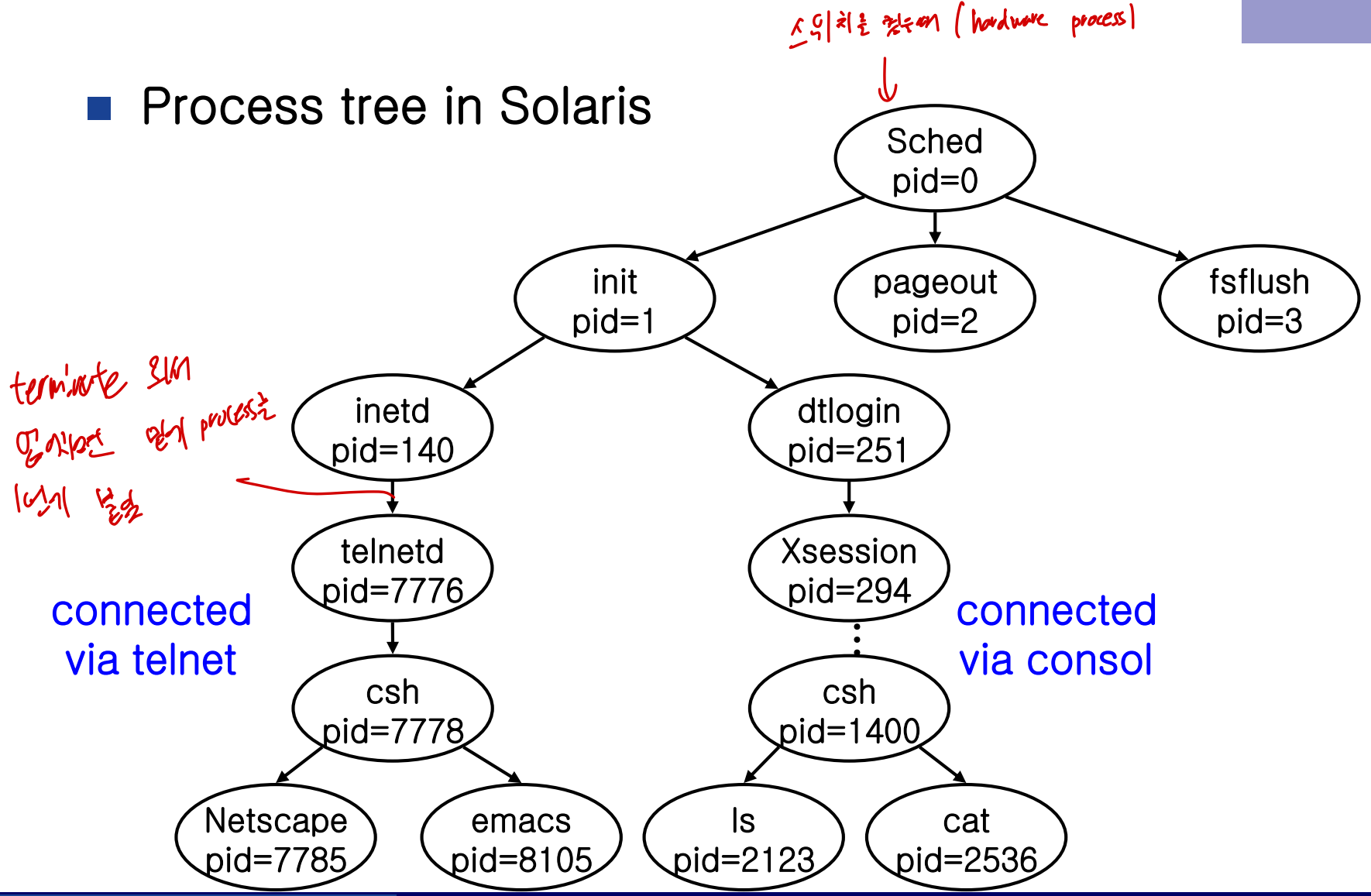
- Create-process system call
  - Creates a process and assigns a pid (process ID).



- Process tree
  - Parent-child relation between processes

Example of Process Tree

- Process tree in Solaris



Displaying Process Information

- UNIX
  - ps [-el]
- Windows
  - Task manager (windows system program)
  - Process explorer (freeware)

Process Creation

- Some options to create a process

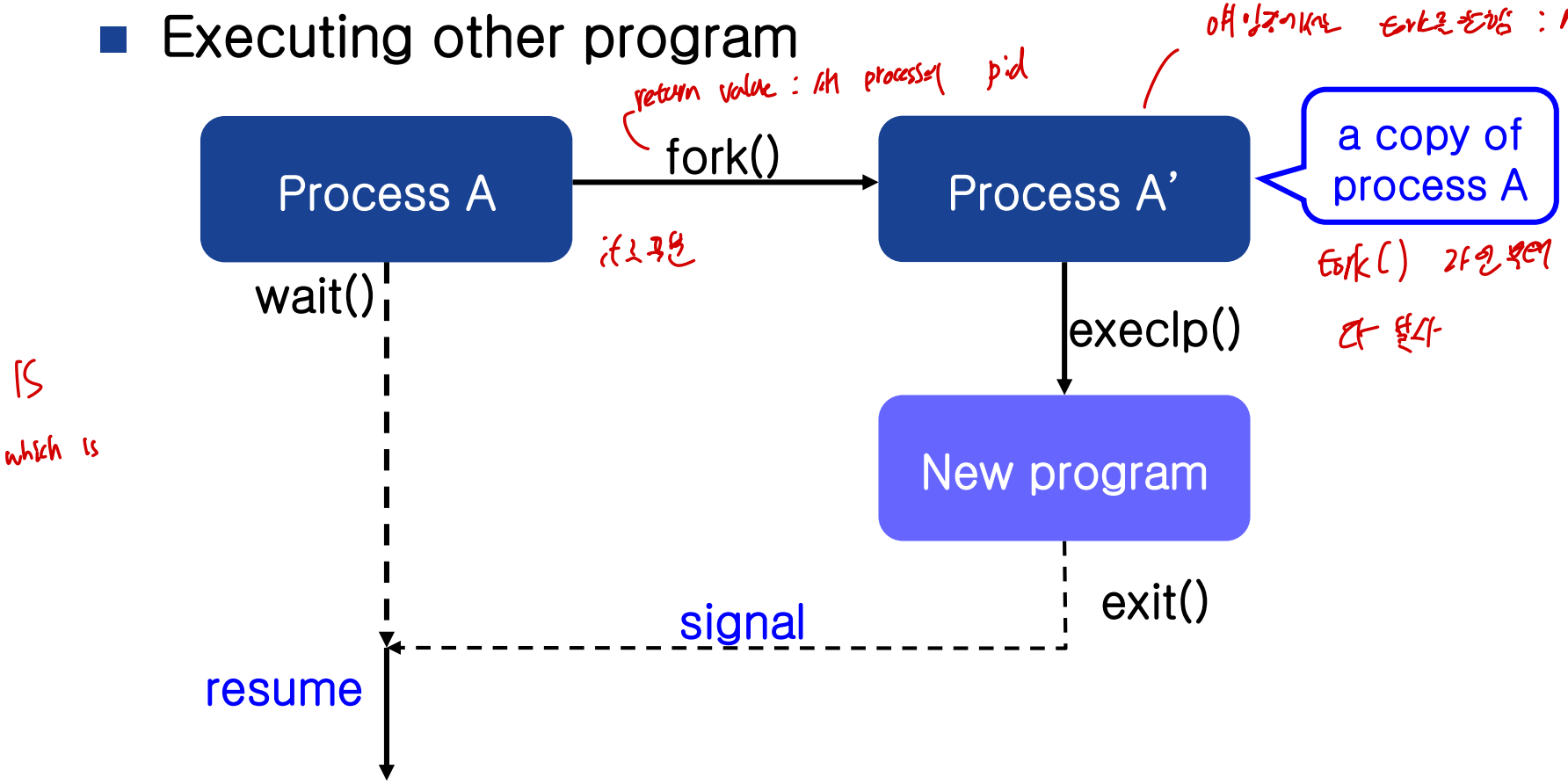
Resource	Child requests its own resource directly from OS or A subset of parent's resource is shared
Execution	Concurrent execution or Parent waits until child is terminated
Address space	Program code and data are shared or Child process has a new program loaded into it

Process Creation in UNIX

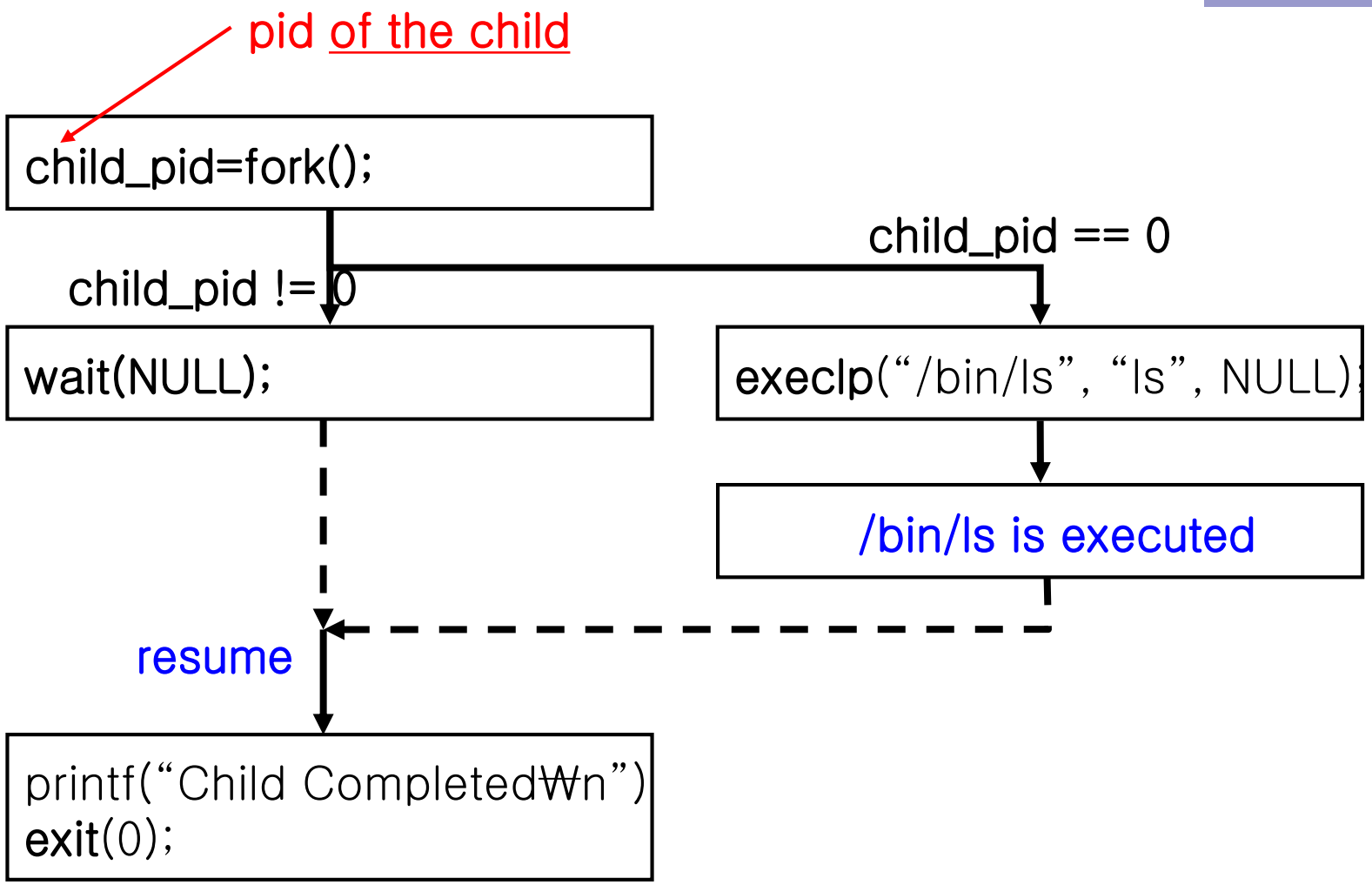
- UNIX system calls related to process creation
  - fork(): create process and returns its pid
    - In parent process, return value is pid of child
    - In child process, return value is zero
  - exec() family: execute a program. The new program substitutes the original one.
    - execl(), execv(), **execlp()**, **execvp()**, execle(), execve()
  - wait(): waits until child process terminates

Example of Process Creation

- Executing other program



Example of Process Creation



Example of Process Creation

```
int main()
{
   pid_t child_pid = fork();     // create a process
                                 // in general, pid_t is defined as int
   if(child_pid < 0){            // error occurred
      fprintf(stderr, "fork failed₩n");
      exit(-1);
   } else if(child_pid == 0){    // child process
      execlp("/bin/ls", "ls", NULL);
   } else {                      // if pid != 0, parent process
      wait(NULL);       //  waits for child process to complete
      printf("Child Completed₩n");
      exit(0);
   }
   return 0;
}
```

Example of Process Creation

- Parent process

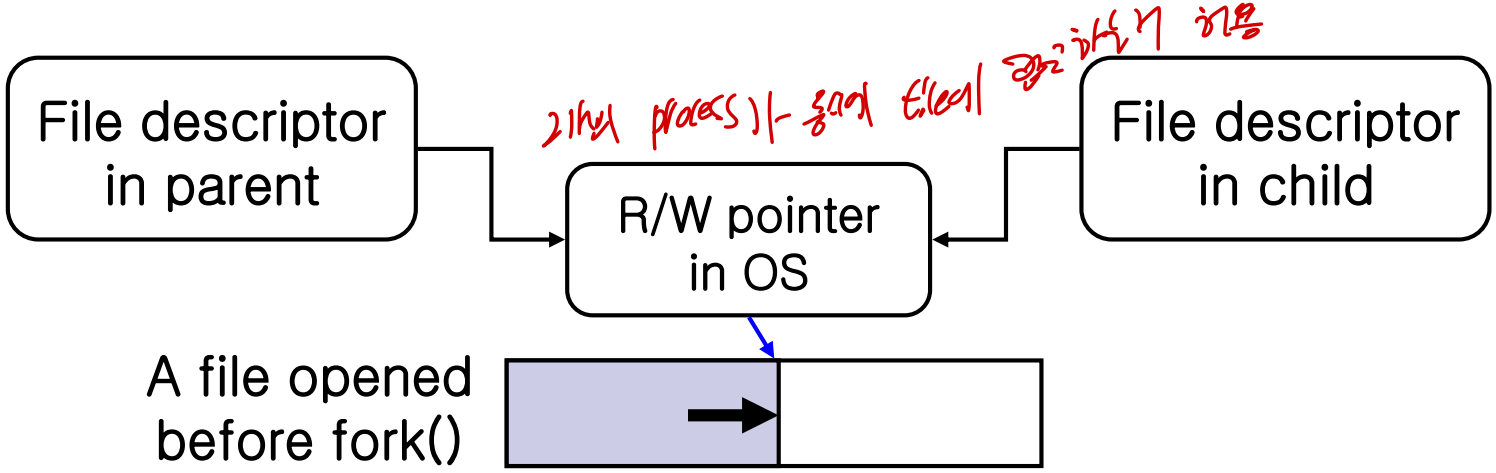
```
int main()
{
   pid_t child_pid = fork();
   if(child_pid < 0){
      fprintf(stderr, "fork failed₩n");
      exit(-1);
   } else if(child_pid == 0){
      execlp("/bin/ls", "ls", NULL);
   } else {
      wait(NULL);
      printf("Child Completed₩n")
      exit(0);
   }
}
```

- Child process

```
int main()
{
   pid_t child_pid = fork();
   if(child_pid < 0){
      fprintf(stderr, "fork failed₩n");
      exit(-1);
   } else if(child_pid == 0){
      execlp("/bin/ls", "ls", NULL);
   } else {
      wait(NULL);
      printf("Child Completed₩n")
      exit(0);
   }
}
```

More About fork()

- Resource of child process
  - Data (variables): copies of variables of parent process
    - Child process has its own address space.
    - The only difference is the pid of child returned from fork().
  - Files
    - Opened before fork(): shared with parent
    - Opened after fork(): not shared



More About exec Family

- Functions in exec family (declared in <unistd.h>)
  - int **execl**(const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/);
  - int **execv**(const char *path, char *const argv[]);
  - int **execlp**(const char *file, const char *arg0, ..., const char *argn, char * /*NULL*/);
  - int **execvp**(const char *file, char *const argv[]);
  - **execle**(), **execve()**

cf. Argument names

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

More About wait()

pid_t wait(int *stat_loc);

- stat_loc : an integer pointer
  - If state_loc == NULL, it is ignored
  - Otherwise: receives status information from child process
    - wait(&stat); // in parent process
    - exit(code); // in child process
      - code == (stat >> 8) & 0xff

- Return value of wait
  - pid of the terminated child
  - -1 means it has no child process

Process Creation in win32

fork()

- CreateProcess() fork + exe
  - Similar to fork() of UNIX, but much more parameters to specify properties of child process

- WaitForSingleObject() thread가 exit할때까지
  - Similar to wait() of UNIX

- void ZeroMemory(PVOID *Destination*, SIZE_T *Length*);
  - Fills a block of memory with zeroes.

For more detail, please refer MSDN homepage
(http://msdn.microsoft.com)

Process Termination

- Normal termination
  - exit(int return_code): invoked by child process
    - Clean-up actions
      - Deallocate memory
      - Close files
      - ETC.
    - return_code is passed to parent process
      - Usually, 0 means success
      - Parent can read the return code

```
int status = 0;
wait(&status);                  // wait until the child is terminated.
ret = WEXITSTATUS(status);      // return_code from the child
```

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - Browser process manages user interface, disk and network I/O
  - Renderer process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in sandbox restricting disk and network I/O, minimizing effect of security exploits
  - Plug-in process for each type of plug-in

Agenda

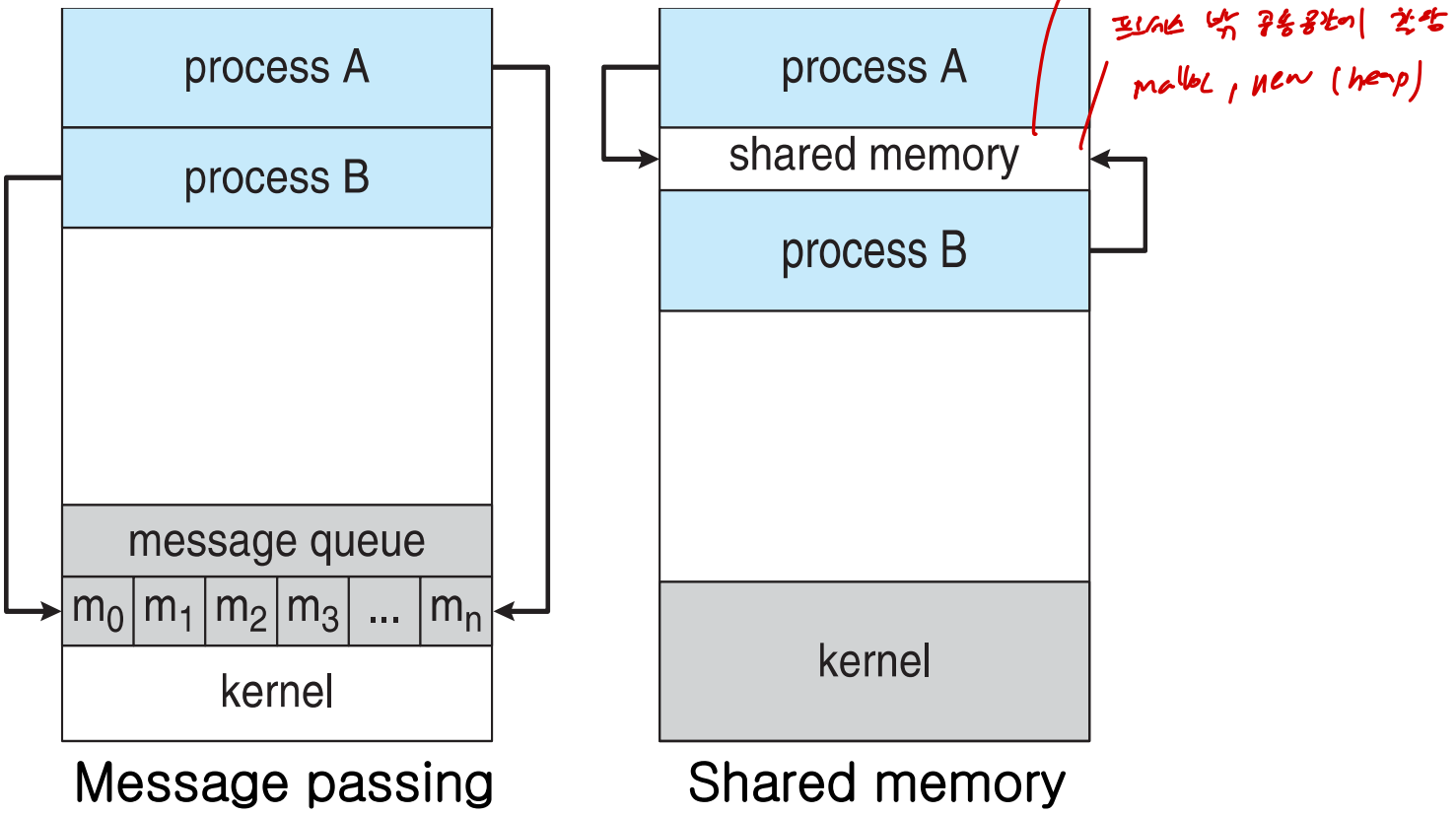
- Overview
- Process scheduling
- Operations on processes
- Inter-process communication
- Example of IPC system
- Communication in client-server systems

Inter-process Communication (IPC)

- Goal of IPC: cooperation
  - Information sharing
    - Shared file, …
  - Computation speedup
    - Multiple CPU or I/O
  - Modularity
    - Dividing system functions
  - Convenience
    - Editing, printing, compiling in parallel

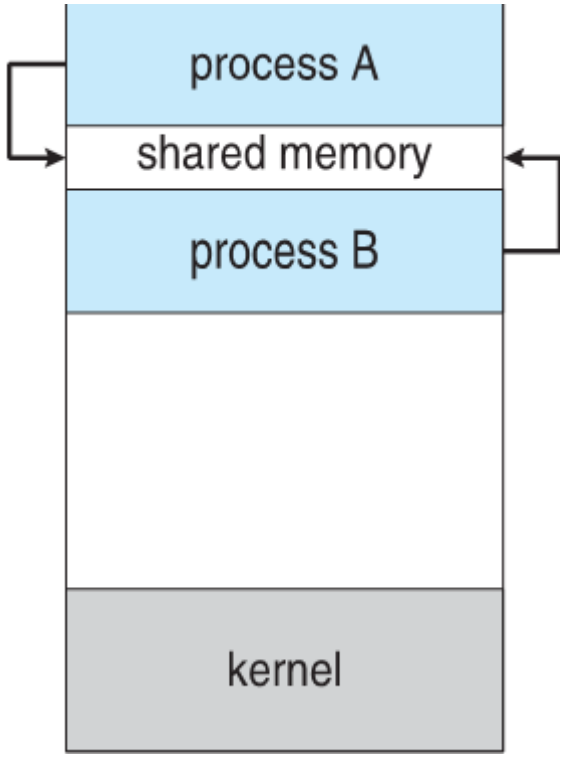
Inter-process Communication (IPC)

- IPC models
  - Message passing model
  - Shared-memory model



Shared-Memory Systems

- Shared-memory segment 공유메모리 → process 끼리 메모리 공간을 만들어 그곳 해쳐놓음
  - Special memory space that can be shared by two or more processes.
  - Form of data and location is not determined by OS, but those processes.
    - Processes should avoid simultaneous writing by themselves



- Advantage
  - Fast

-> Suitable for large amount of data

Ex) producer-consumer problem

Producer–Consumer Problem

- Producer and consumer communicate information (item) through shared memory

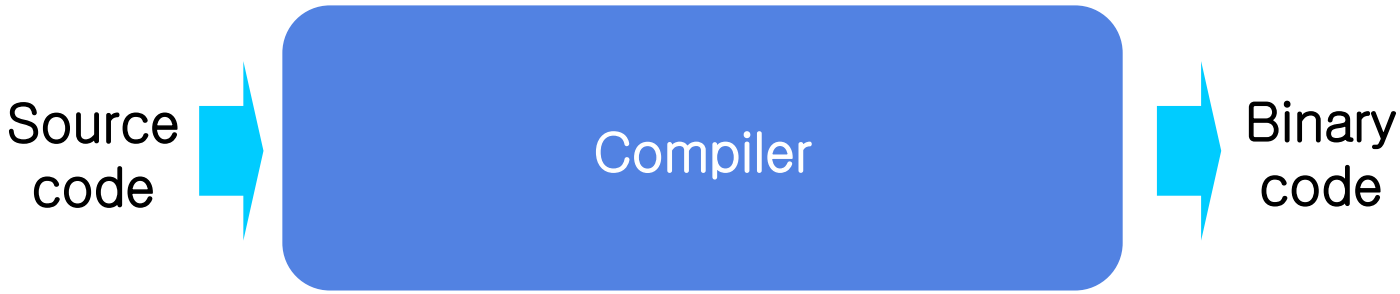
  - Producer: produce information for consumer
  - Consumer: consume information written by producer

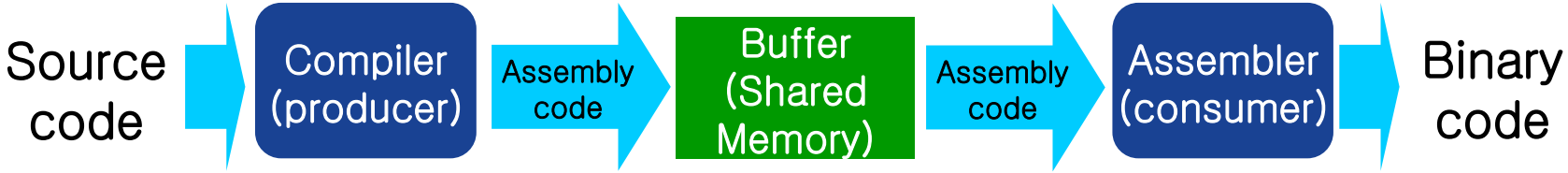
Ex) compiler – assembler, server – client

Note! Producer and consumer should be synchronized.

→ Discussed in chapter 6

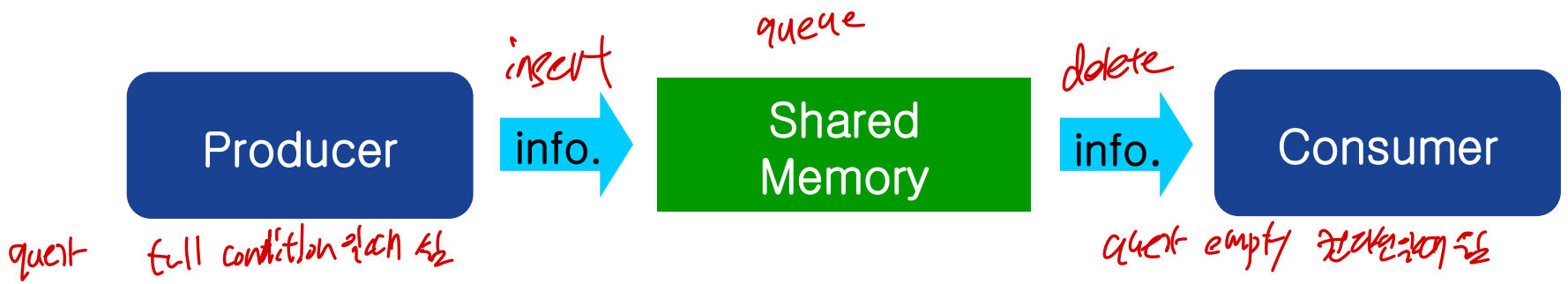
Producer-Consumer Problem





Producer–Consumer Problem

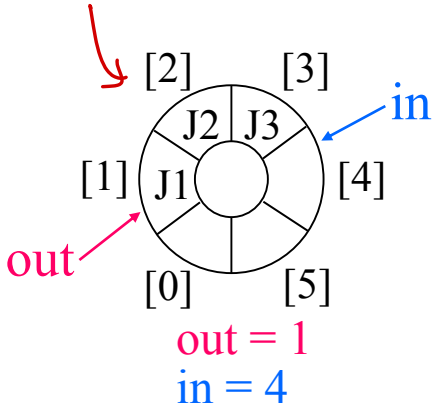
- Two types of buffer
  - Unbounded buffer
    - No practical limit on buffer size
    - Producer can always produce
  - Bounded buffer
    - Producer must wait if buffer is full.



Producer-Consumer Problem using Bounded Buffer

- Representation of buffer
  - Buffer is represented by circular queue

```
#define BUFFER_SIZE 6
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;          // tail or rear
int out = 0;         // head or front
```



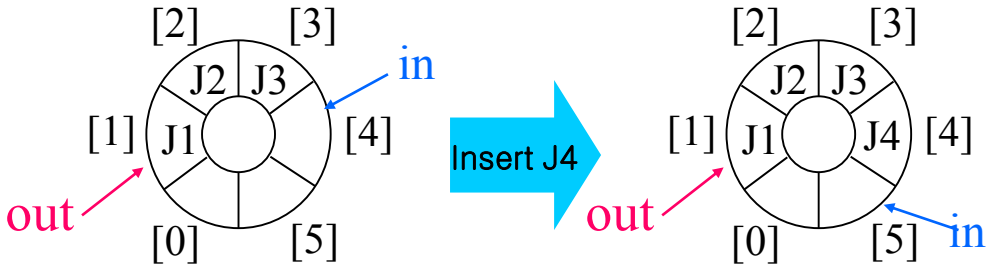
  - Empty/full condition
    - in == out: buffer is empty
    - (in+1)%BUFFER_SIZE == out: buffer is full

    Cf. Buffer can store at most BUFFER_SIZE − 1 items

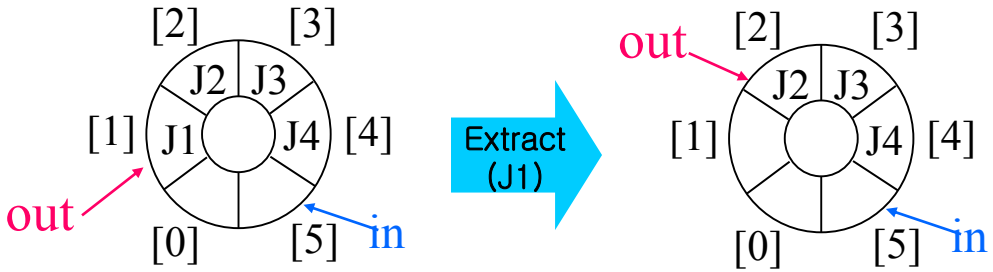
Circular Queue

- Circular queue: fixed-size buffer whose logical structure is circular
  - Last element is followed by first element

- Inserting an item
  - buffer[in] = newItem;
  - in = (in + 1) % n;



- Extracting an item
  - item = buffer[out];
  - out = (out + 1) % n;



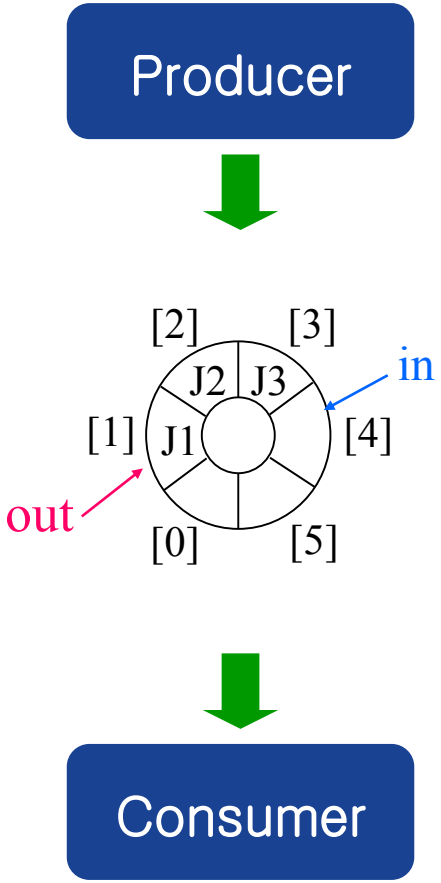
Producer-Consumer Problem using Bounded Buffer

- Producer

```
item nextProduced;

while (1) {
   // produce an item in nextProduced
   while (((in + 1) % BUFFER_SIZE) == out); // waiting
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
}
```

- Consumer

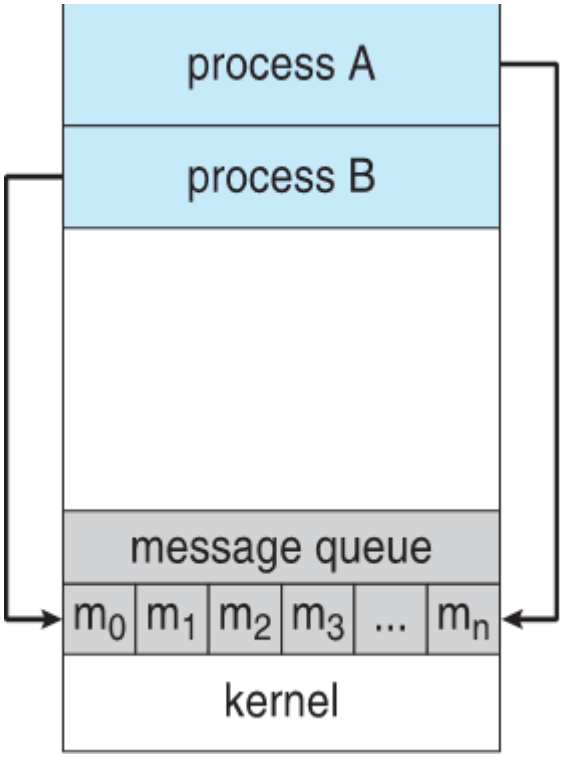
```
item nextConsumed;

while (1) {
   while (in == out);                    // waiting
   nextConsumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   // consume the item in nextConsumed
}
```



Message-Passing Systems

- Process communication via passage-passing facility provided by OS

- Advantage
  - No conflict
    -> Suitable for smaller amounts of data
  - Communication between processes on different computer



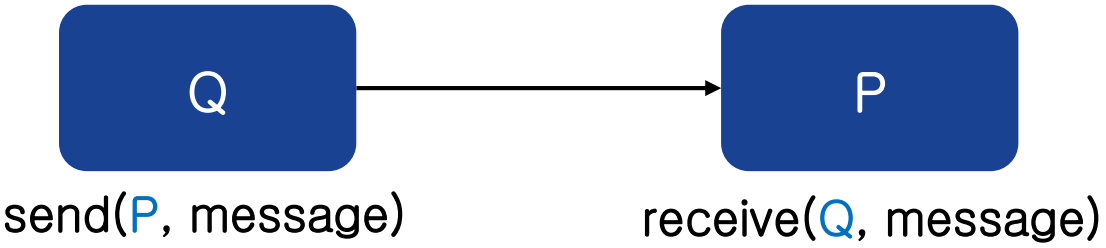
Message-Passing Systems

- For message passing, communication link should be exist between the processes
- Essential operations
  - send(message)
  - receive(message)
- (Logical) Implementation methods
  - Direct/indirect
  - Synchronous/asynchronous
  - Buffering
    - Zero/bounded/unbounded capacity

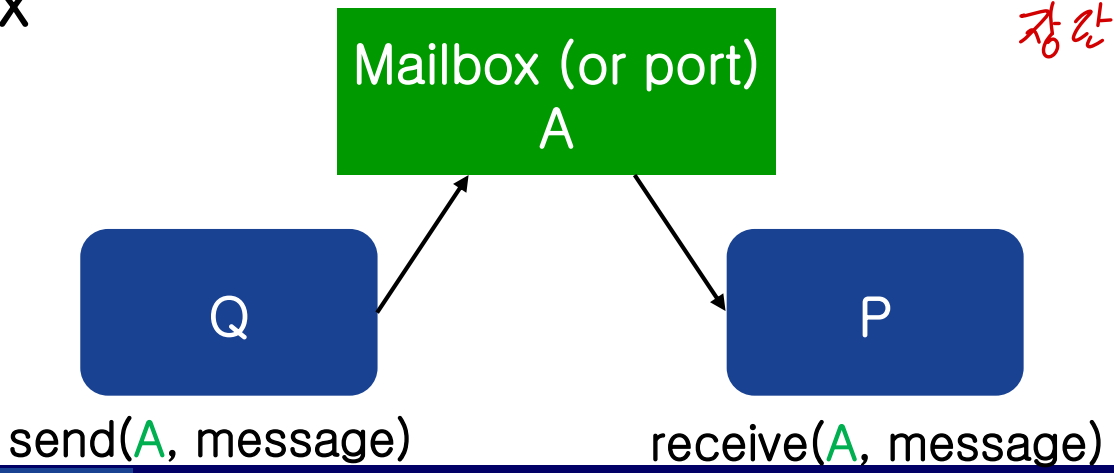
➔ Reading assignment: read the textbook for detail.

Direct/Indirect Communication

- Direct communication: connection link directly connects processes



- Indirect communication: processes are connected via mailbox



정한샘 : 교과서보기

Buffering

- During communication, messages are stored in temporary queue (buffer)

- Three kinds of buffer capacity
  - Zero capacity: only blocking send is possible
  - Bounded capacity: buffer has finite length n
    - If buffer is full, sender must be blocked
    - Otherwise, sender can resume
  - Unbounded capacity: buffer has infinite capacity
    - Sender never blocks

Agenda

- Overview
- Process scheduling
- Operations on processes
- Inter-process communication
- Example of IPC system
- Communication in client-server systems

Examples of IPC Systems

- Shared-memory (System-V, POSIX)
- Message-passing (MACH)
- Local Procedure Call (Windows XP)
  - Undocumented internal API

[System V] Shared-Memory

- Create shared memory

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

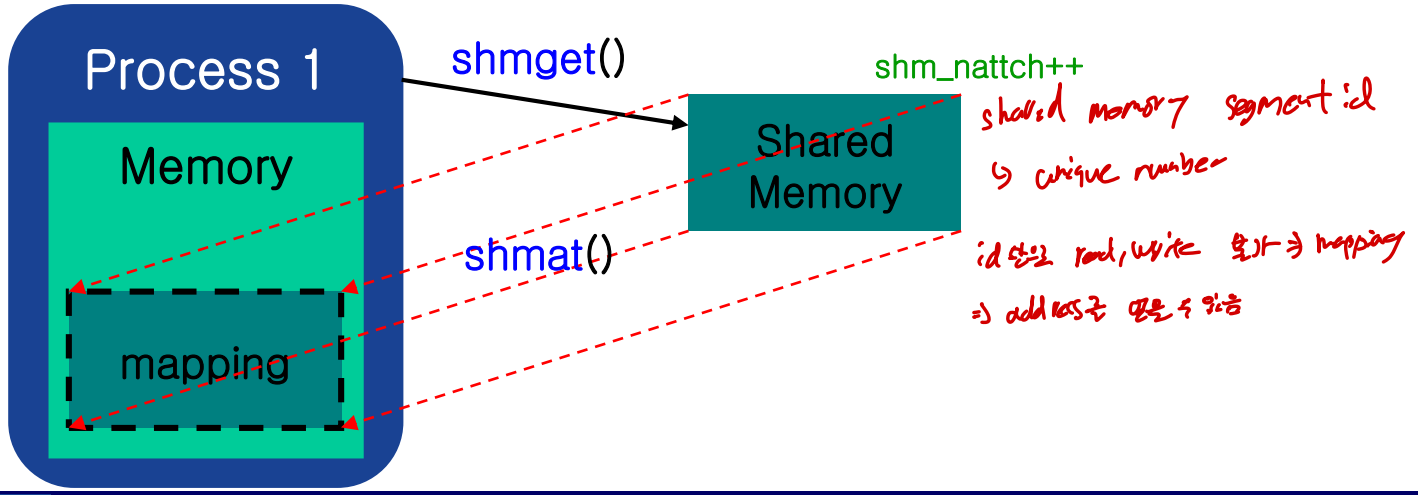
  Ex) seg_id = shmget(IPC_PRIVATE, size, S_IRUSR|S_IWUSR);

  Can create 32k shared memory segments at maximum.

- Attach shared memory to address space of a process

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

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



[System V] Shared-Memory

- Use shared memory through attached address as ordinary memory

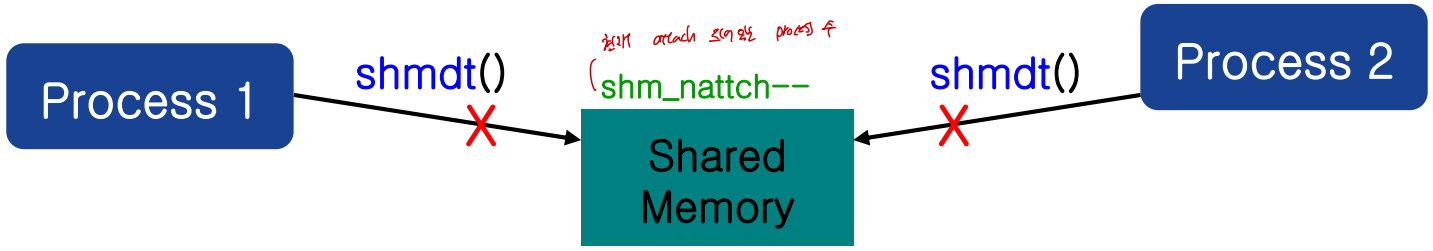
  Ex) sprintf(shared_mem, “Writing to shared memory”);

- Detach shared memory from address space of process

  *int shmdt (char *shmaddr);*

  Ex) shmdt(shared_mem);

  - If all processes detaches the shared memory segment, OS discards it.



[System V] Shared-Memory

- Deallocating a shared memory block

shmctl(shmid, IPC_RMID, NULL);

- Deallocates the shared memory block when the shm_nattch becomes zero.

Reading Assignment

Search the following topics from Internet and study them

- System V shared memory
  - shmget() – allocate a share memory block
  - shmat() – attach a shared memory block to a process
  - shmdt() – detach a shared memory block from a process
  - shmctl() – controls and deallocate a shared memory block
  - www.xevious7.com/linux/lpg_6_4_4.html (Korean)
  - www.cs.cf.ac.uk/Dave/C/node27.html (English)

POSIX Shared Memory

- POSIX shared memory
  - Communication through memory-mapped file.
  - Process creates or open shared memory segment

    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    - O CREAT: create if it does not yet exist
    - O RDWR: open for reading and writing

  - Set the size of the object

    ftruncate(shm_fd, 4096);

  - Map shared memory segment to process address space

    shared_memory = mmap(
    0, 4096, PROT_WRITE, MAP_SHARED, shm_fd, 0);

  - Now the process could write to the shared memory

    sprintf(shared_memory, "Writing to shared memory");

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Reading Assignment

Search the following topics from Internet and study them

- POSIX shared memory (requires compilation option '-lrt')
  - shm_open(): http://man7.org/linux/man-pages/man3/shm_open.3.html
  - shm_unlink(): http://man7.org/linux/man-pages/man3/shm_unlink.3p.html
  - ftruncate(): http://man7.org/linux/man-pages/man3/ftruncate.3p.html
  - mmap(): http://man7.org/linux/man-pages/man2/mmap.2.html
  - munmap(): http://man7.org/linux/man-pages/man3/munmap.3p.html

Reading Assignment

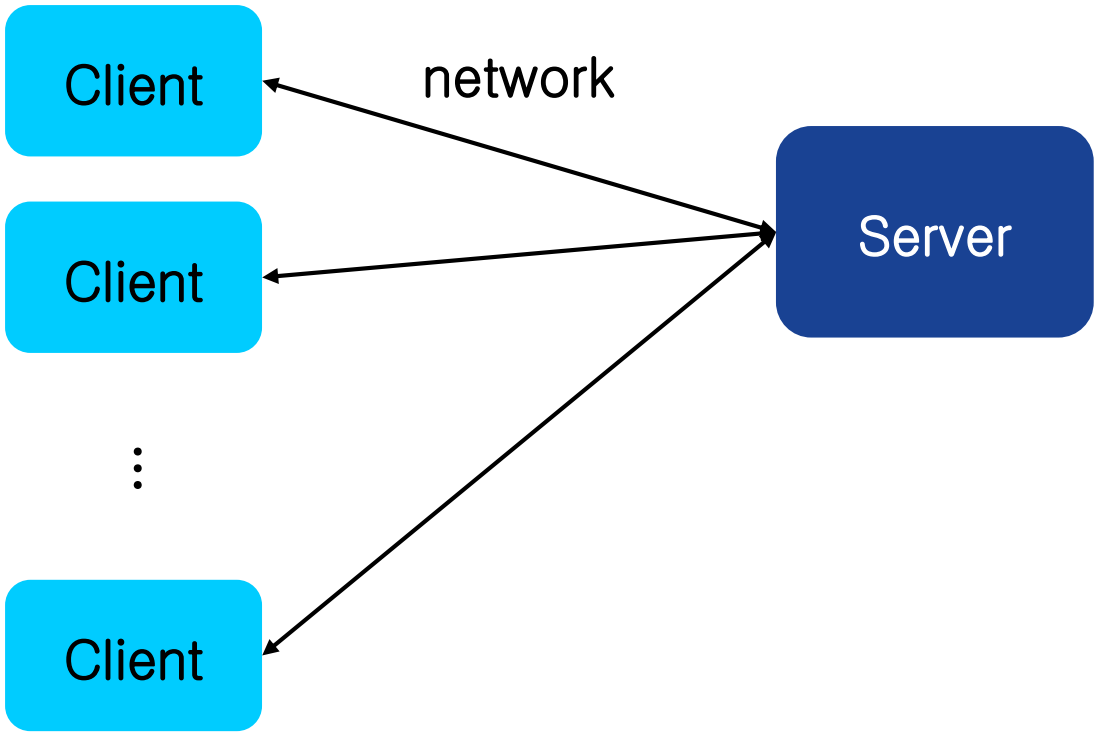
Search the following topics from Internet and study them

- POSIX message passing
  - msgget() – create a message queue 할당시 allocation
    - http://forum.falinux.com/zbxe/index.php?document_srl=420147&mid=C_LIB (Korean)
    - http://www.tutorialspoint.com/unix_system_calls/msgget.htm (English)
  - msgsnd() – send a message to a message queue
    - http://forum.falinux.com/zbxe/index.php?document_srl=420634&mid=C_LIB (Korean)
    - http://www.tutorialspoint.com/unix_system_calls/msgsnd.htm (English)
  - msgrcv() – receive a message from a message queue
    - http://forum.falinux.com/zbxe/index.php?document_srl=420636&mid=C_LIB (Korean)
    - http://www.tutorialspoint.com/unix_system_calls/msgrcv.htm (English)
  - msgctl() – control/deallocate message queue (eg: msgctl(msgq, IPC_RMID, NULL);)
    - http://forum.falinux.com/zbxe/index.php?document_srl=421044&mid=C_LIB (Korean)
    - http://www.tutorialspoint.com/unix_system_calls/msgctl.htm (English)

Agenda

- Overview
- Process scheduling
- Operations on processes
- Inter-process communication
- Example of IPC system
- Communication in client-server systems

Client–Server



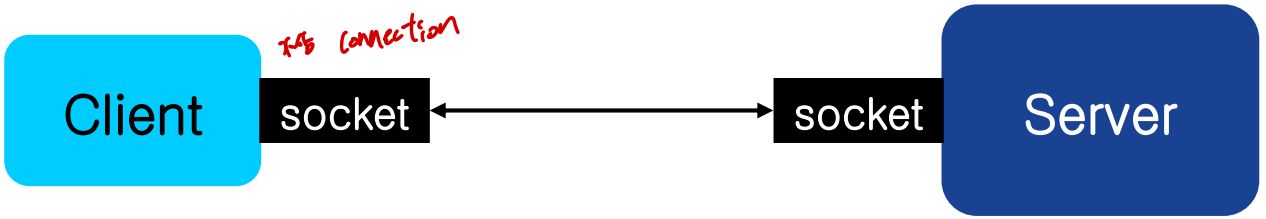
Communications in Client-Server Systems

- Socket
  - Data communication
- RPC (Remote Procedure Call)
  - Procedure call between systems
    - Procedural programming
- Pipes
  - Often used for Input/output redirection
- RMI (Remote Method Invocation) of JAVA
  - Invocating method of object in other system
    - Object oriented programming

Socket

- Socket: logical endpoint for communication

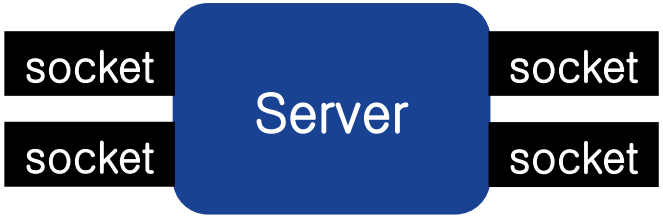
- Identified by <ip address>:<port #>



- Each connection is identified by a pair of sockets.

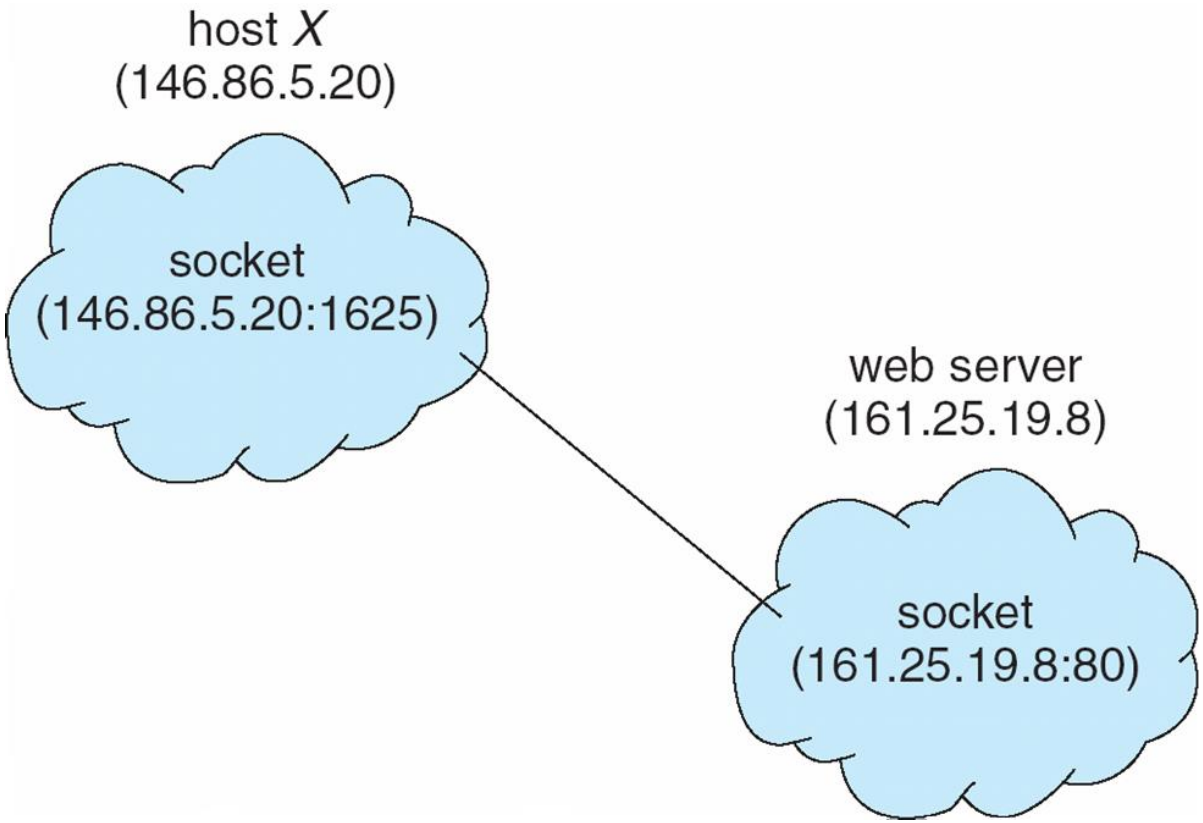
Socket

- Port: logical contact point to a computer recognized by TCP and UDP protocols
  - A computer may have multiple ports (0 ~ 65535)



  - Well-known services have their own ports below 1024
    Ex) telnet: 23, ftp: 21, http: 80 , ssh: 22
    - Server always listens corresponding port.
  - Ports above 1024 can be arbitrary assigned for network communication

Socket


host X
(146.86.5.20)
socket
(146.86.5.20:1625)
web server
(161.25.19.8)
socket
(161.25.19.8:80)

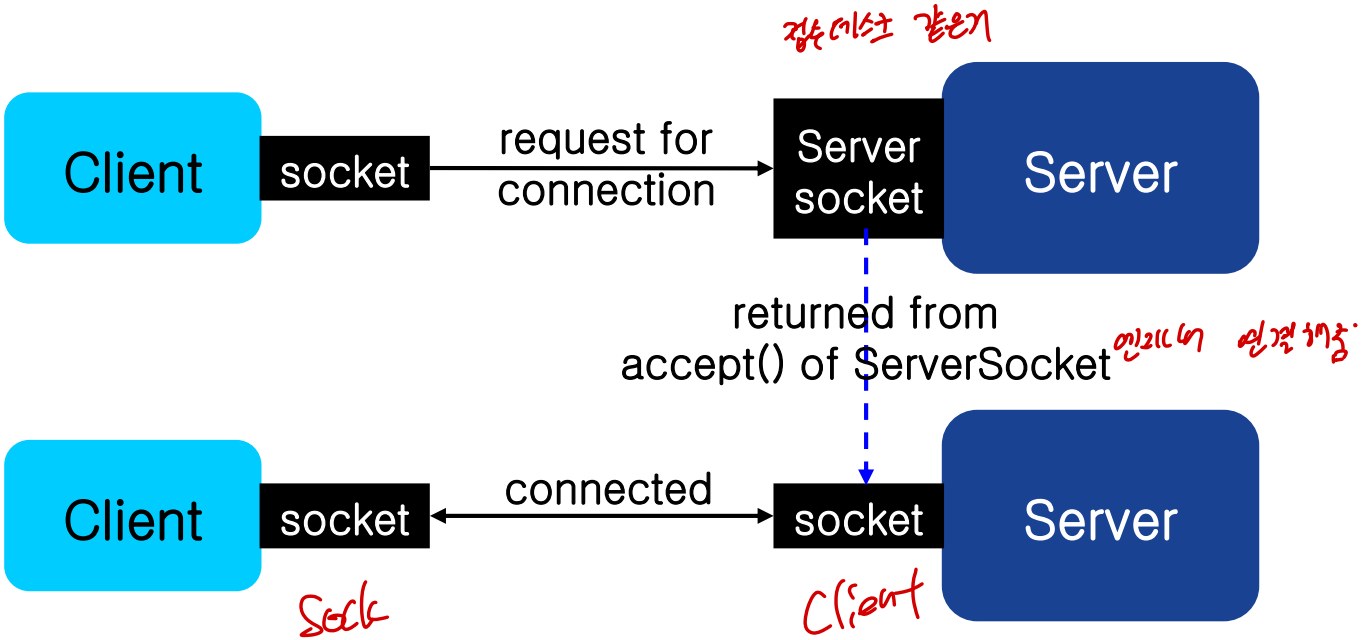
Socket

- Server opens a port to accept connection request.
- Initiating connection
  - Client arbitrary assigns a port above 1024.
    Ex) a client 146.86.5.20 assigned a port 1625
  - Client request a connection to server.
    Ex) a web server 161.25.19.8 (port # of web service: 80)
  - If server accepts request, connection is established.
    Ex) <146.86.5.20:1625> – <161.25.19.8:80>

Client socket Server socket

Java Socket

- Socket classes
  - ServerSocket: accepts request for connection
  - Socket: in charge of actual communication



Java Socket

- Server
  1. Create a ServerSocket

     ServerSocket ***socket*** = new ServerSocket(6013);

  2. Wait for a client

     Socket ***client*** = *socket*.accept();

  4a. If a client is accepted, communicate with client via *client*

- Client

  3. Create a socket to server

     Socket ***sock*** = new Socket("127.0.0.1", 6013);

  4b. If connection was established, communicate with server via *sock*

Java Socket

- Server (given *client*)

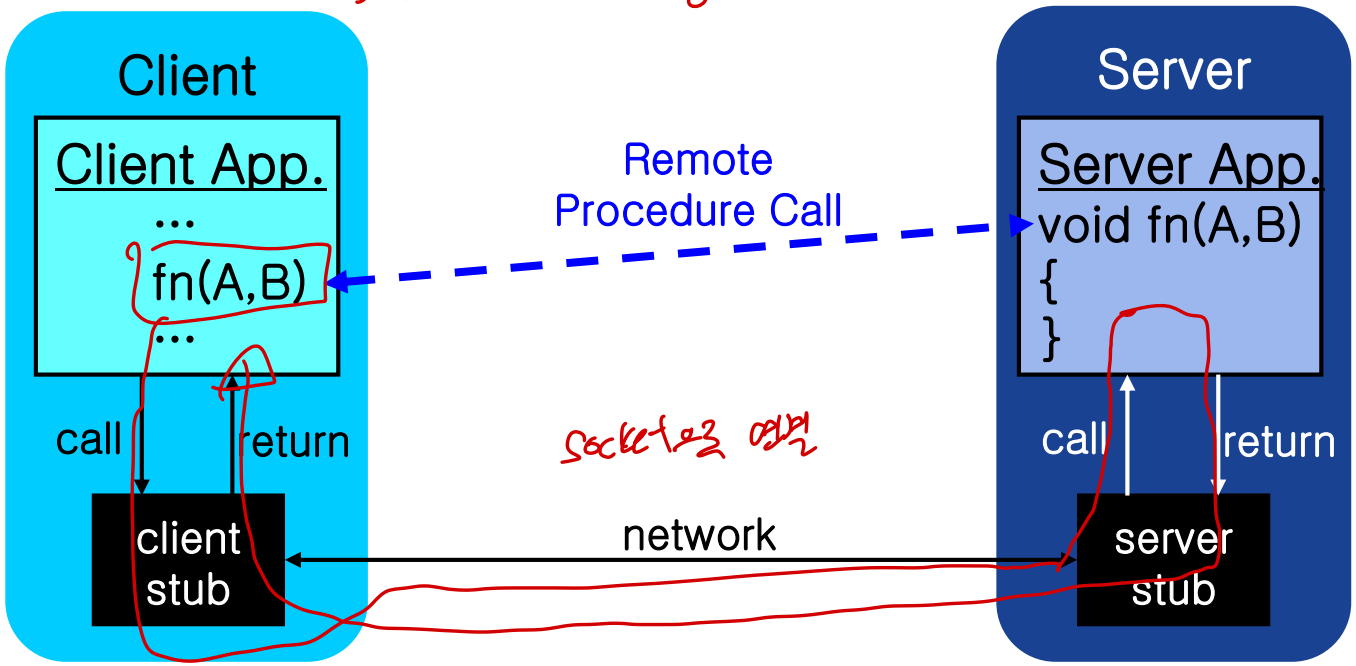
```
PrintWriter pout = new
    PrintWriter(client.getOutputS
    tream(), true);

pout.println(new
    java.util.Date().toString());

client.close();
```

- Client (given *sock*)

```
InputStream in =
    sock.getInputStream();
BufferedReader bin = new
    BufferedReader(new
    InputStreamReader(in))

String line;
while((line = bin.readLine()) !=
    null)
    System.out.println(line);

sock.close();
```

Remote Procedure Calls (RPC)

- RPC: procedure call mechanism between systems
- On server, RPC daemon listens a port
- Client sends a message containing identifier of function and parameters



Remote Procedure Calls

- RPC is served through stubs
  - Client invoke remote procedure as it would invoke a local procedure call

- Stub: a small program providing interface to a larger program or service on remote side
  - Client stub / server stub
  - Locate port on server
  - Marshal / unmarshal parameters

Remote Procedure Calls

- Parameter marshaling

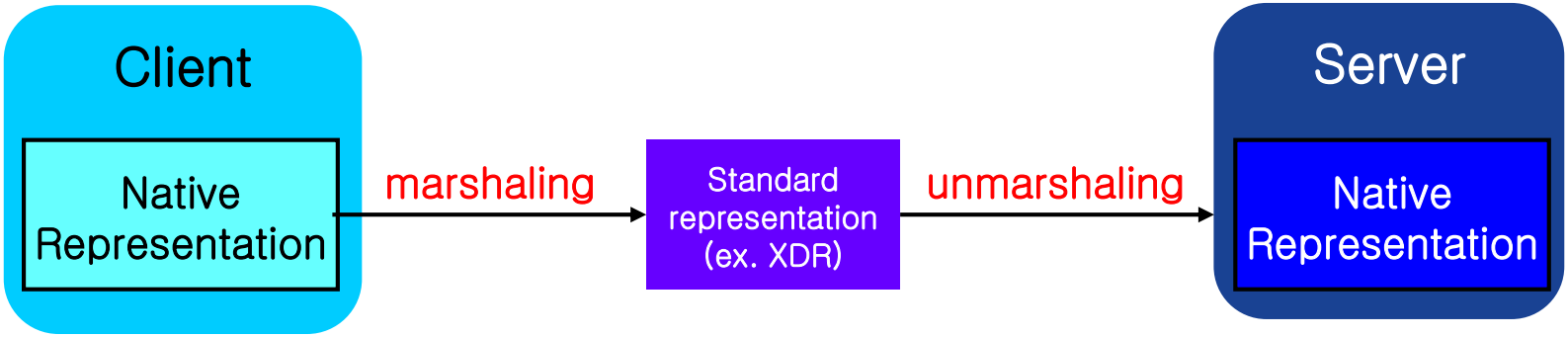
  Motivation: each system has its own data format

  Ex) Representation of integer on a system may different from that on other system

  ➔ Parameter should be transferred in machine-independent standard representation

  Ex) XDR (eXternal Data Representation)

  - Marshalling: Packaging (native format ➔ standard format)
  - Unmarshalling: Unpackaging (standard format ➔ native format)

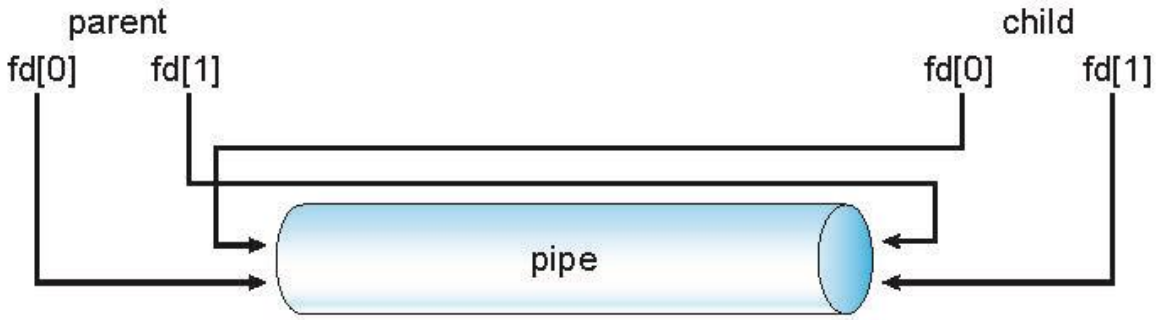


RPC Reference Sites

- Windows
  - MSDN RPC page: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/rpcank.asp
- Unix
  - Document about *rpcgen*.

Pipes

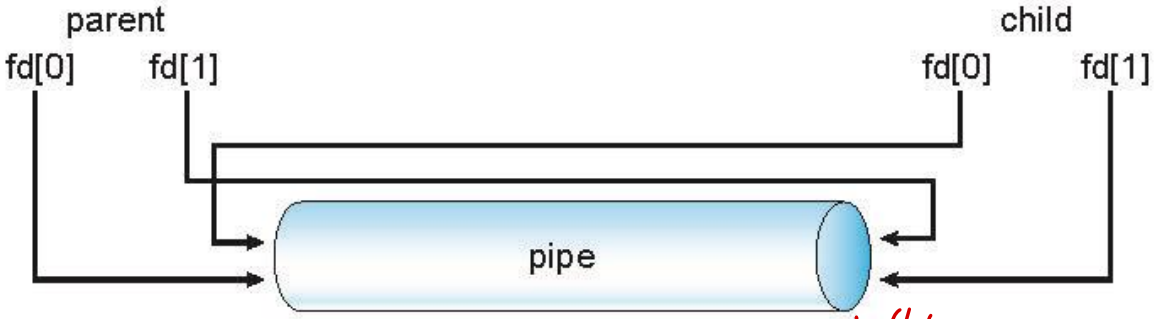
- Pipes acts as a conduit allowing two processes to communicate



- Ordinary pipes
  - Unidirectional communication between parent and child
  - Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes
  - Can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary pipes allow unidirectional communication in standard producer–consumer style
  - Producer writes to one end (the write-end of the pipe)
  - Consumer reads from the other end (the read-end of the pipe)
  - Require parent–child relationship between communicating processes



- Windows calls these anonymous pipes

Example: Ordinary Pipes

```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
   char write_msg[BUFFER_SIZE] = "Greetings";
   char read_msg[BUFFER_SIZE];
   int fd[2];
   pid_t pid;

   /* create the pipe */
   if (pipe(fd) == -1) {
      fprintf(stderr,"Pipe failed");
      return 1;
   }

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }

   if (pid > 0) { /* parent process */
      /* close the unused end of the pipe */
      close(fd[READ_END]);

      /* write to the pipe */
      write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

      /* close the write end of the pipe */
      close(fd[WRITE_END]);
   }
   else { /* child process */
      /* close the unused end of the pipe */
      close(fd[WRITE_END]);

      /* read from the pipe */
      read(fd[READ_END], read_msg, BUFFER_SIZE);
      printf("read %s",read_msg);

      /* close the read end of the pipe */
      close(fd[READ_END]);
   }

   return 0;
}
```

Named Pipes

- Named pipes are more powerful than ordinary pipes
  - Communication is bidirectional
  - No parent-child relationship is necessary between the communicating processes
  - Several processes can use the named pipe for communication
  - Provided on both UNIX and Windows systems

Example: Named Pipes

```
/* set up security attributes allowing pipes to be inherited */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* allocate memory */
ZeroMemory(&pi, sizeof(pi));

/* create the pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
   fprintf(stderr, "Create Pipe Failed");
   return 1;
}

/* establish the START_INFO structure for the child process */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);
```

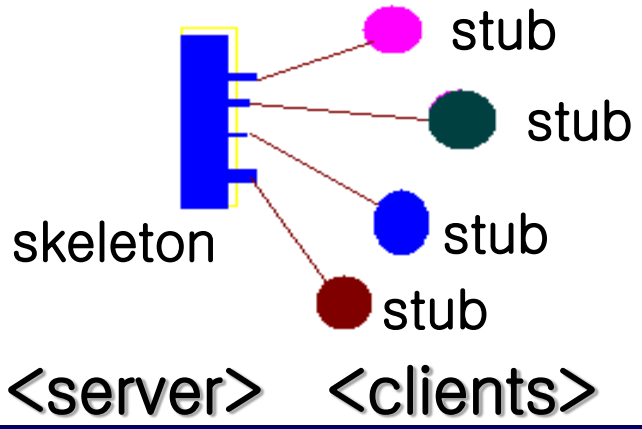
Example: Named Pipes

```
/* create the child process */
CreateProcess(NULL, "child.exe", NULL, NULL,
 TRUE, /* inherit handles */
 0, NULL, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
   fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

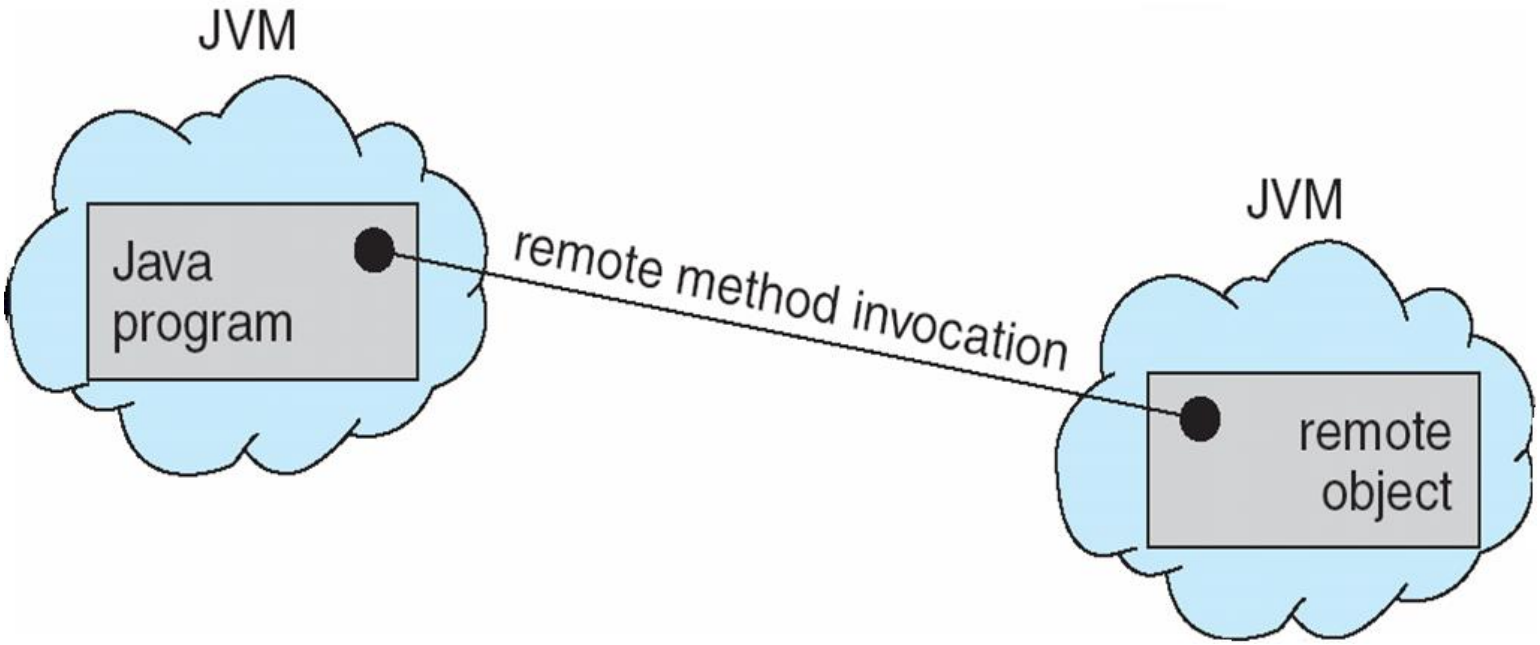
Remote Method Invocation (RMI)

- RMI: Java feature to invoke method on remote object

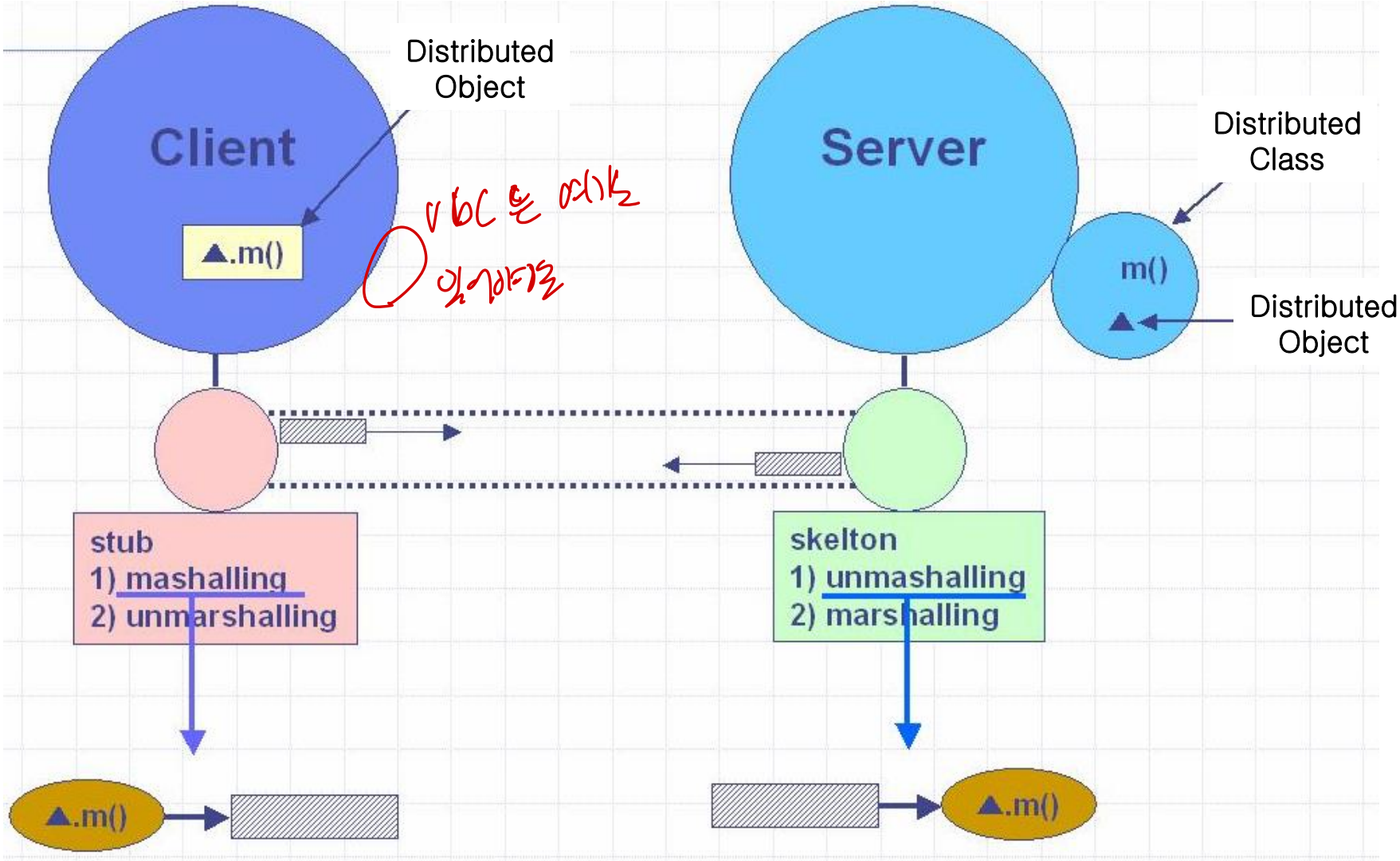
	RPC	RMI
Technical Background	Procedural Programming	Object-oriented Programming
Parameter	Ordinary data structures	Object parameter is possible
Interface	client stub / server stub	stub / skeleton



Remote Method Invocation (RMI)



Remote Method Invocation (RMI)



Remote Method Invocation (RMI)