3. Process Concept

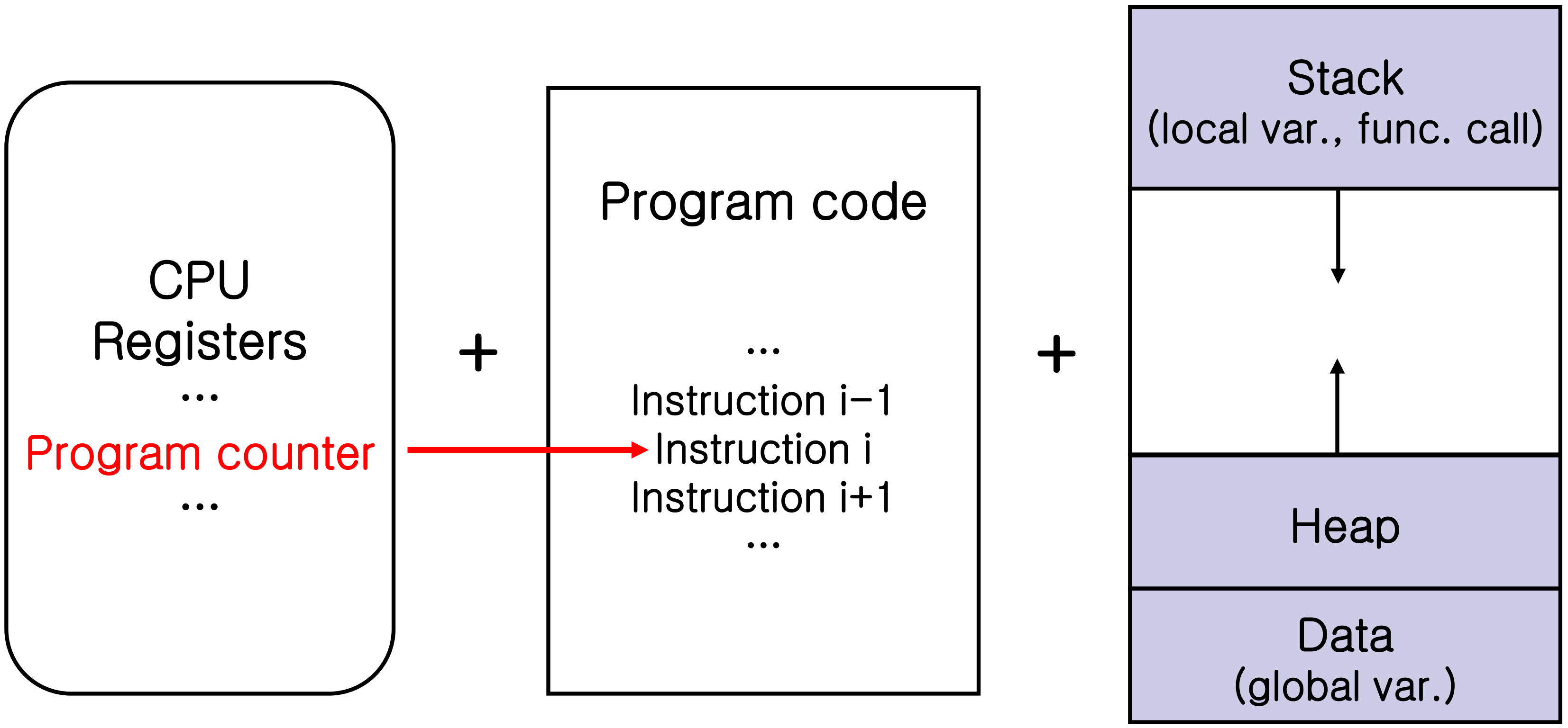
[ECE30021/ITP30002] Operating Systems

Agenda

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

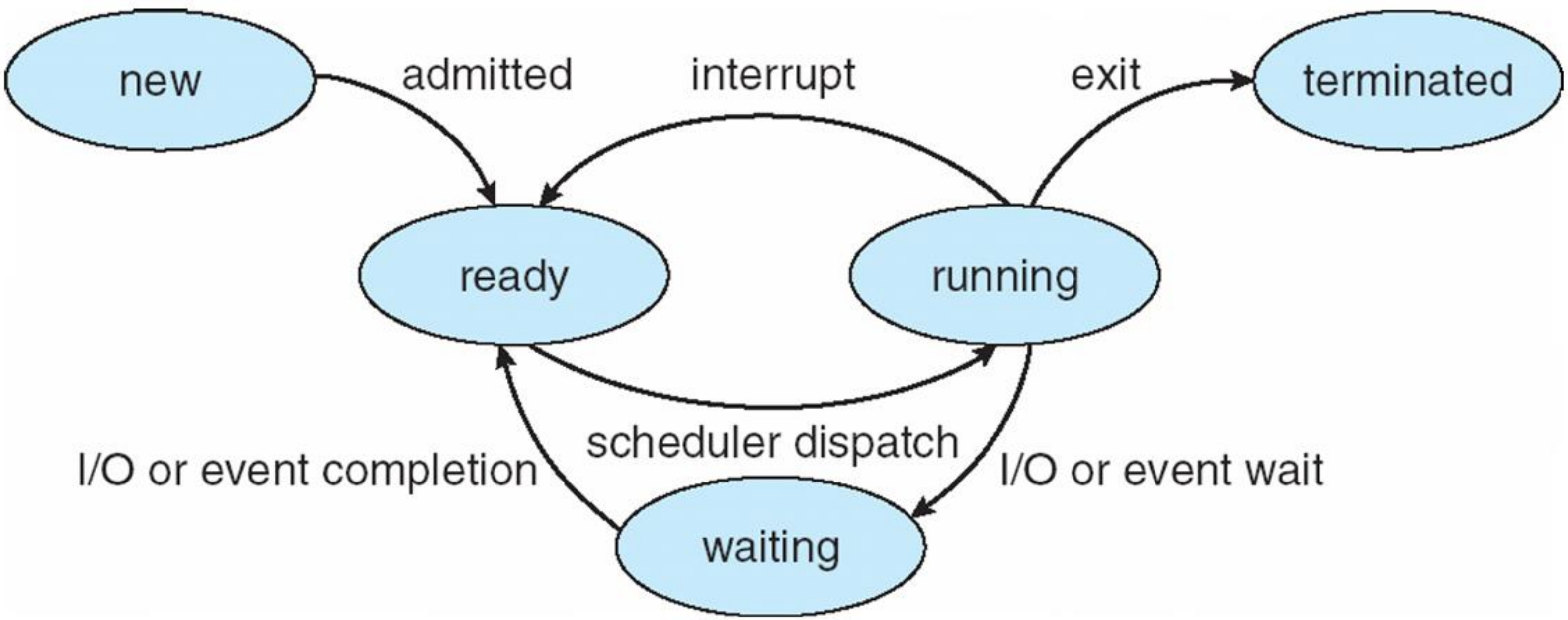
Process

- Process = program in execution + resource

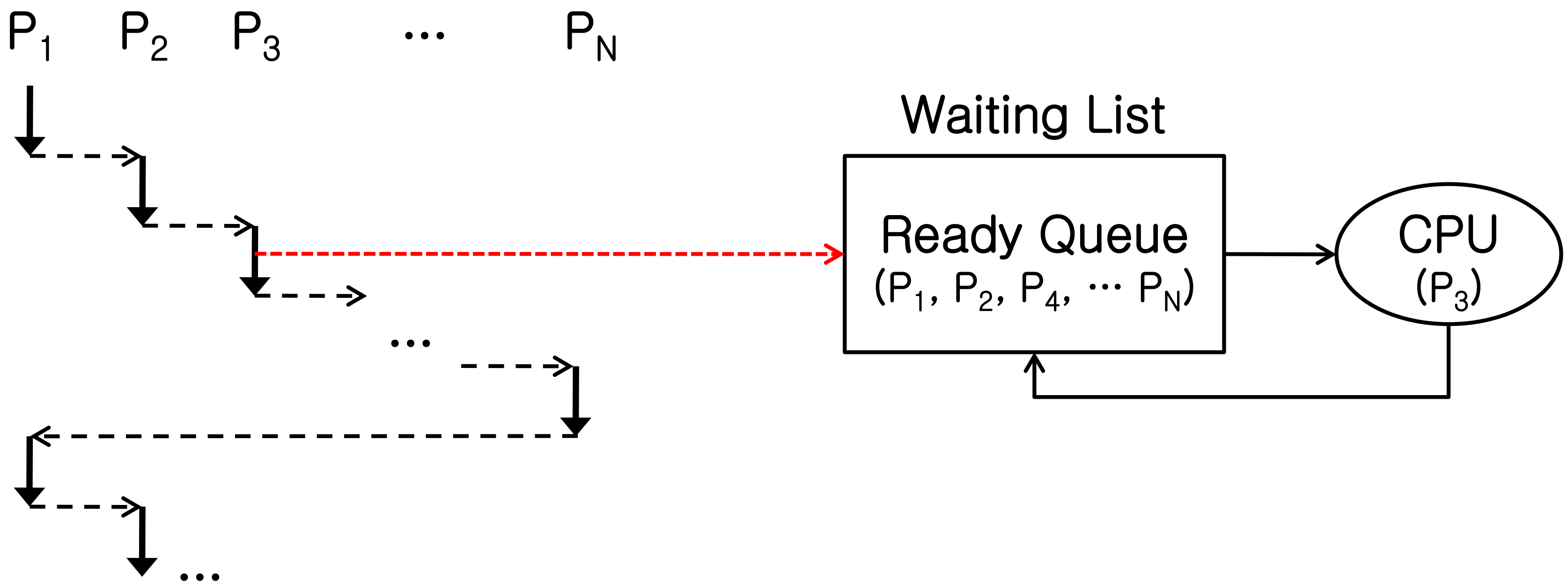


Process State

- **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



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)
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

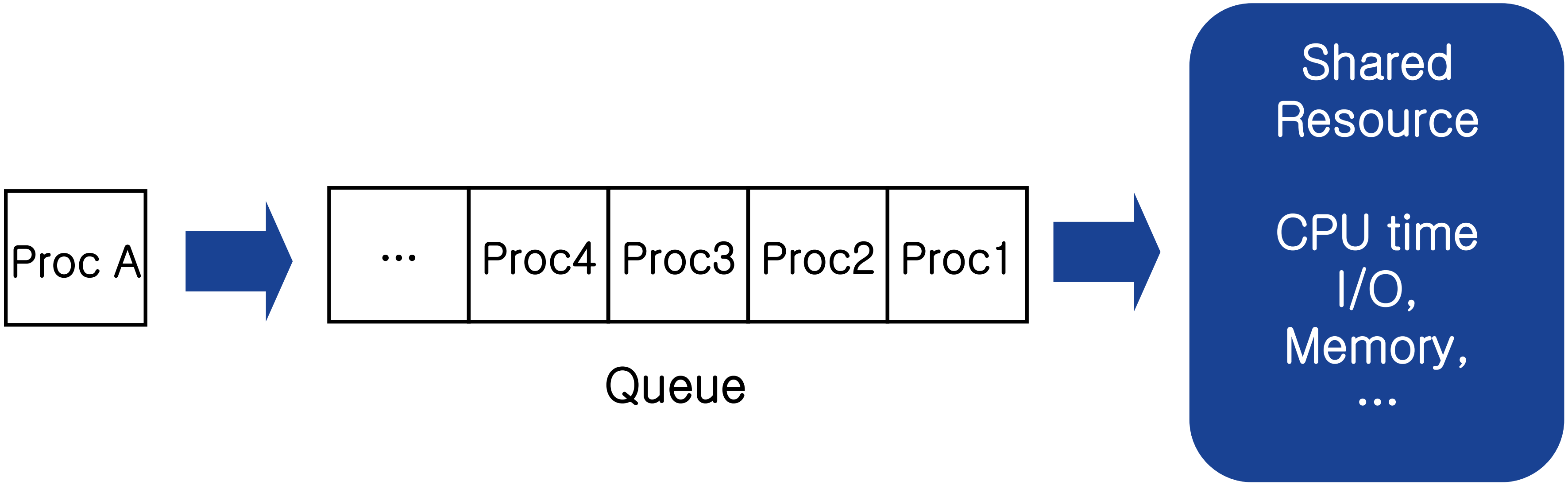
- Overview
- <u>Process scheduling</u>
- Operations on processes
- Inter-process communication
- Example of IPC system
- Communication in client-server systems

Process Scheduling

- Scheduling: assigning tasks to a set of resources
- 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
  - Users can interact with each program

Scheduling Queue

- Scheduling queue: waiting list of processes for CPU time or other resources

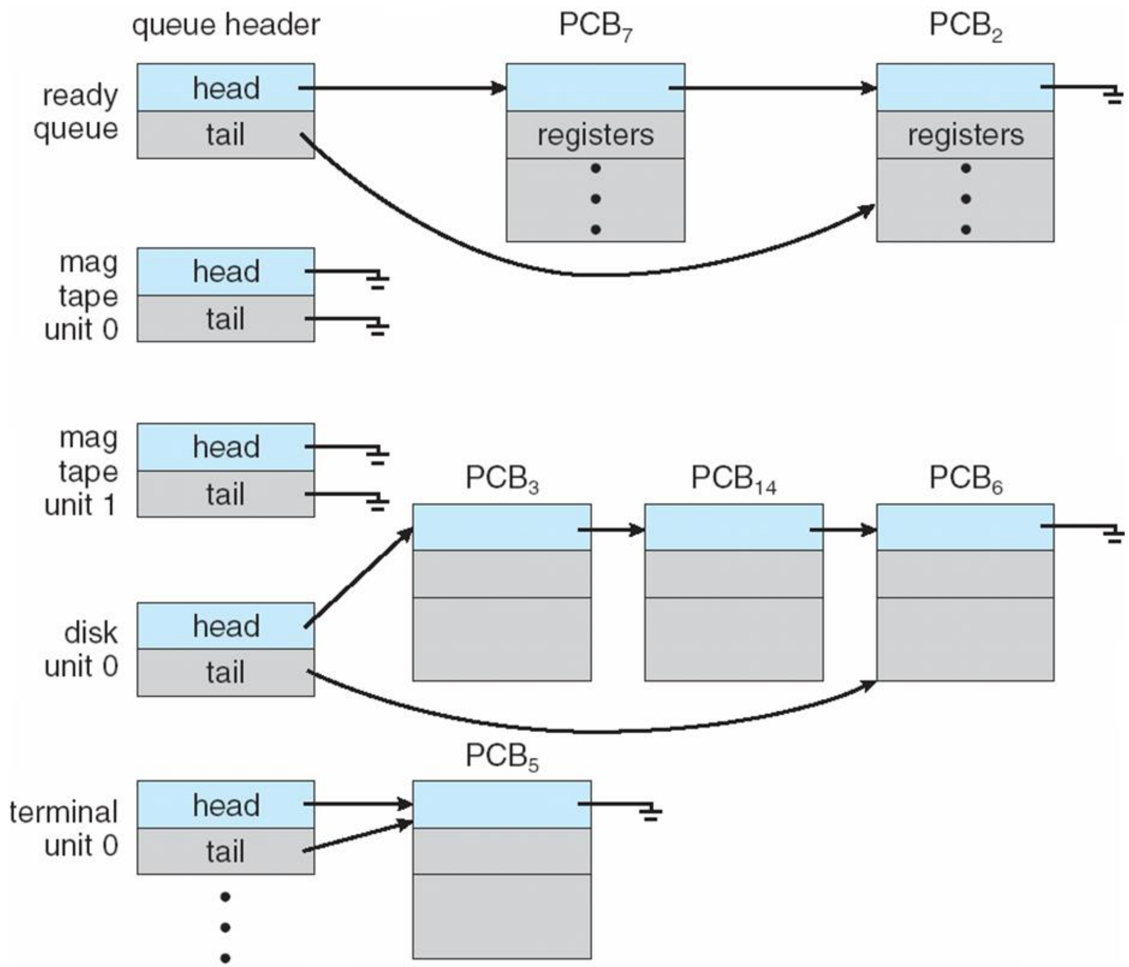


Types of Scheduling Queues

- Job queue
  - List of all processes in the system
- Ready queue
  - List of processes, residing in main memory, ready to execute
- Device queue
  - List of processes waiting for a particular I/O device.
  - Each device has its own device queue.

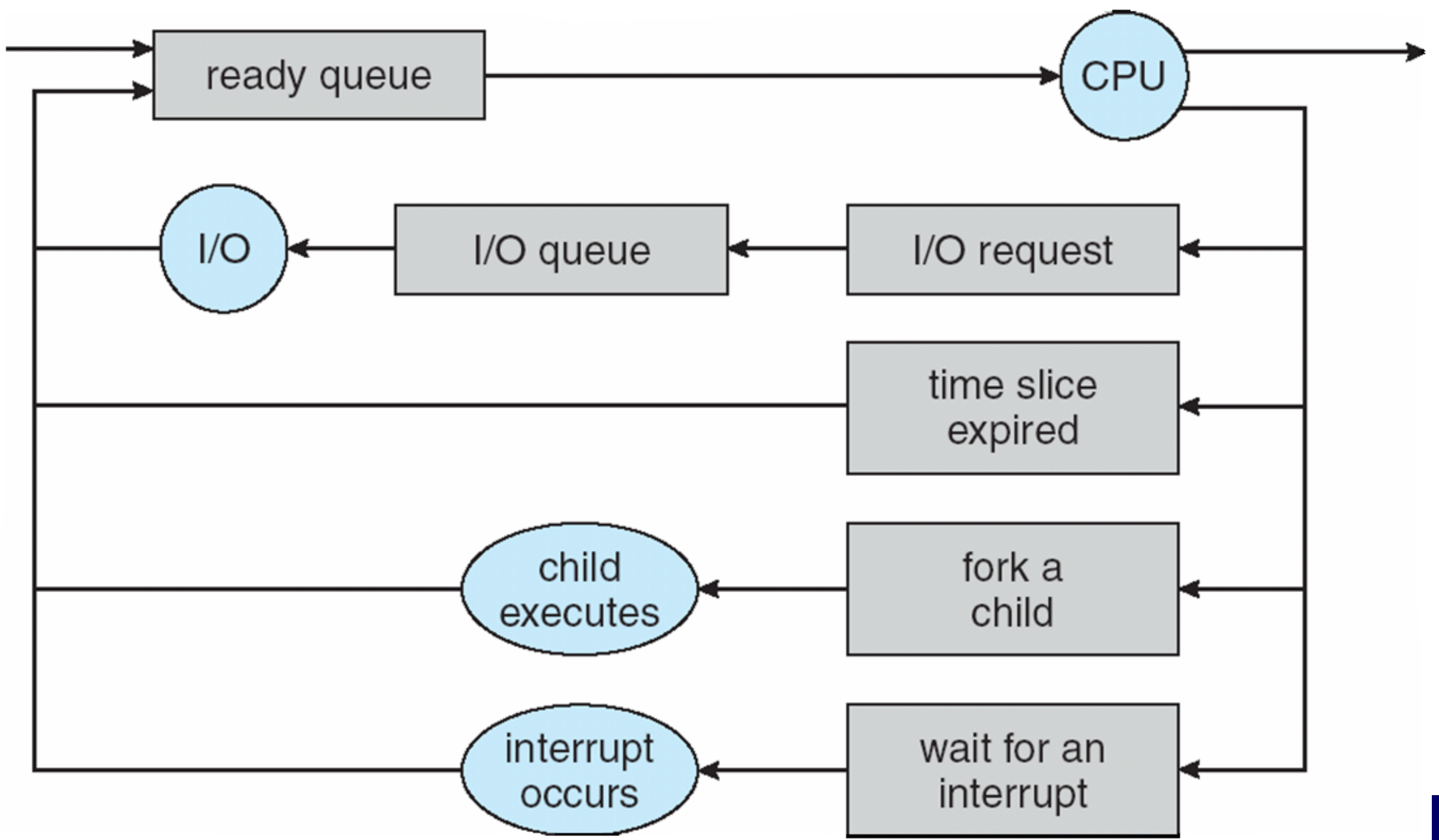
Scheduling Queue

- Each queue is usually represented by linked list



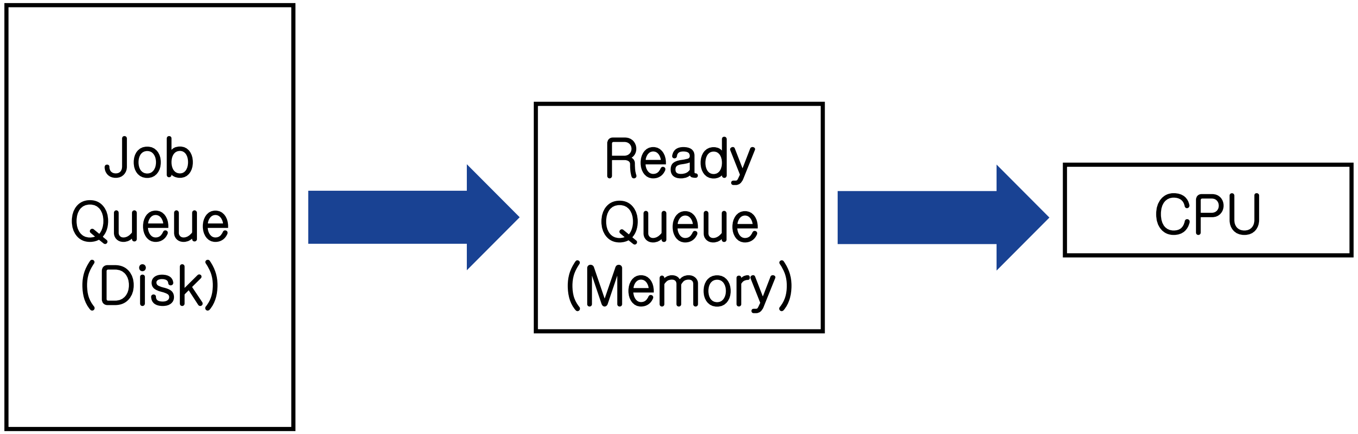
Queueing Diagram

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



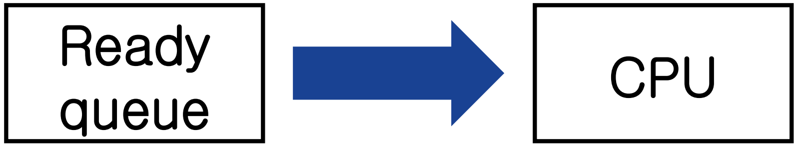
Schedulers

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



Schedulers

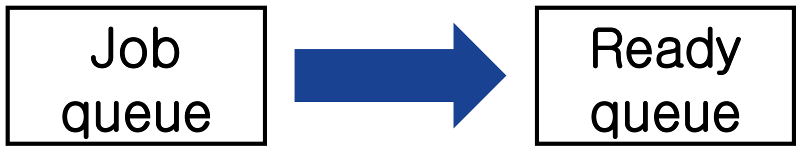
- Short-term scheduler (CPU scheduler)



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

Schedulers

- Long-term scheduler (job scheduler)

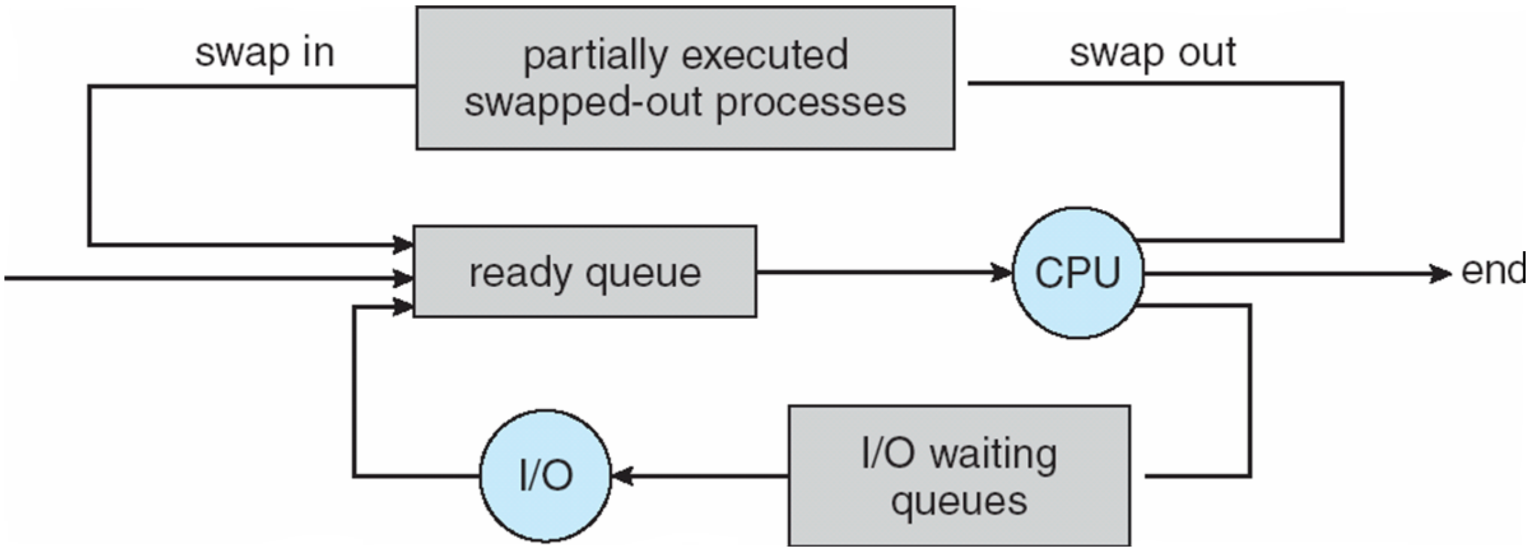


  - Controls degree of multiprogramming
    - In stable state, average process creation rate == average process departure rate
  - Executed less frequently
    - Executed only when a process leaves the system
  - Hopefully, long-term scheduler should select a good mix of I/O-bound and CPU-bound processes

Schedulers

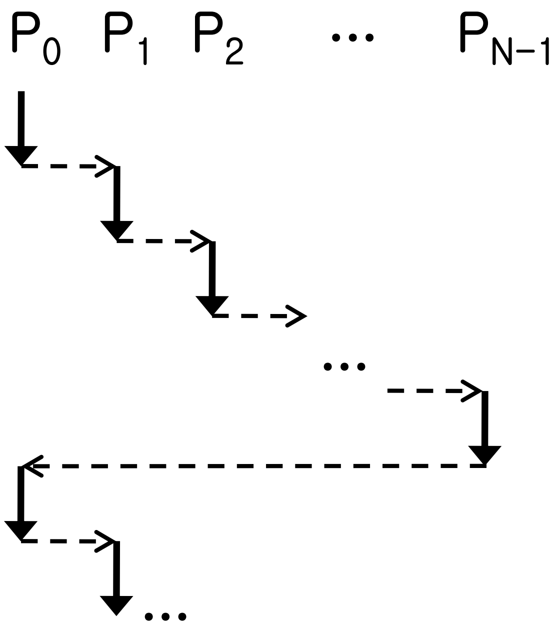
- In some systems, long-term scheduler may be absent or minimal
  Ex) UNIX, Windows
  - System stability depends on physical limitation or self-adjusting nature of human

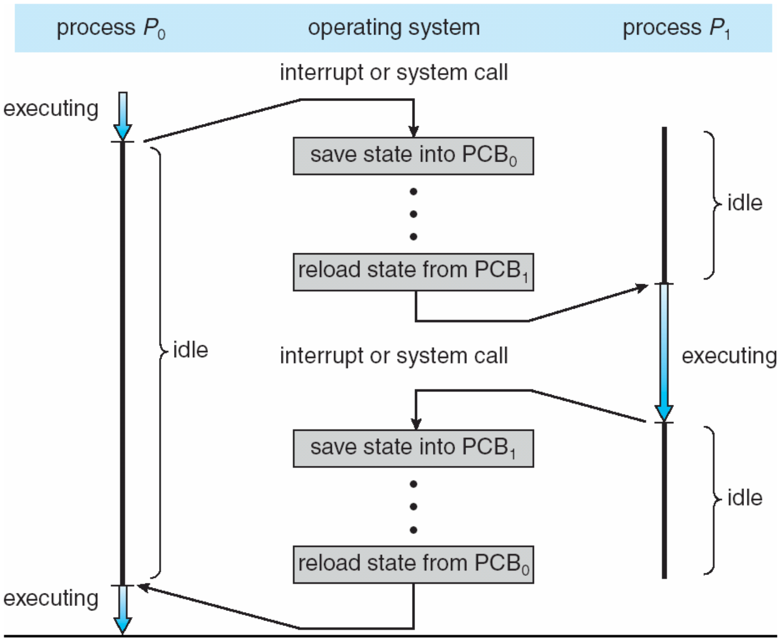
- Some time-sharing system has medium-term scheduler
  - Reduce degree of multiprogramming by removing processes from memory



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, …

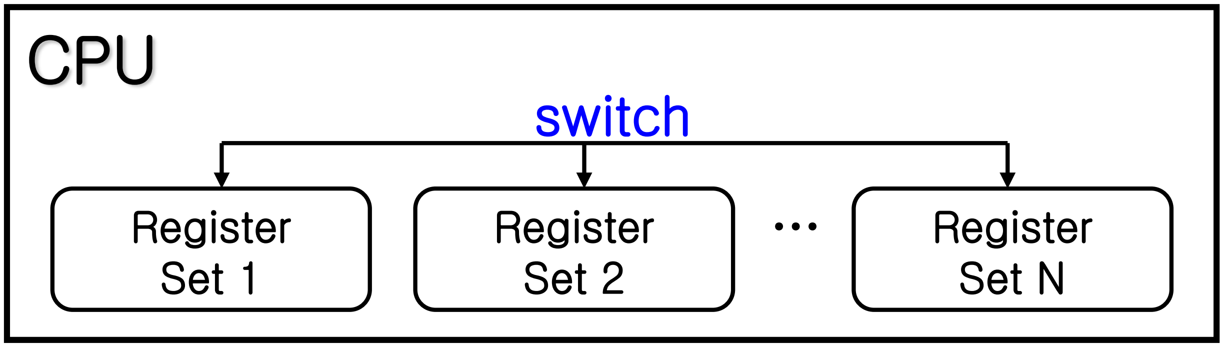
- When to switch?
  - Multitasking
  - Interrupt handling

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

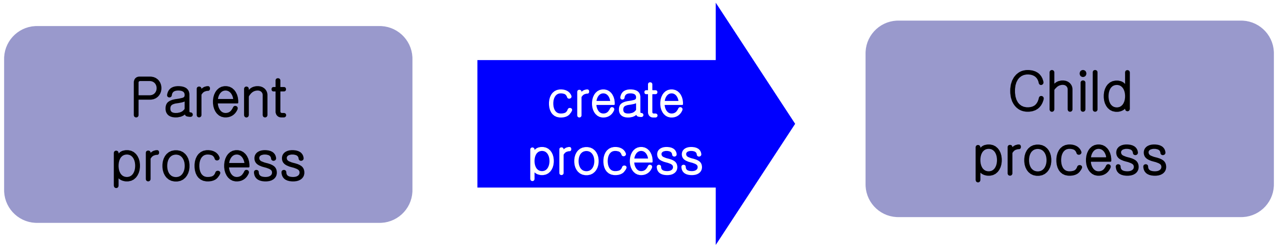


Operations on Processes

- Process create
- Process termination
- Process communication

Process Creation

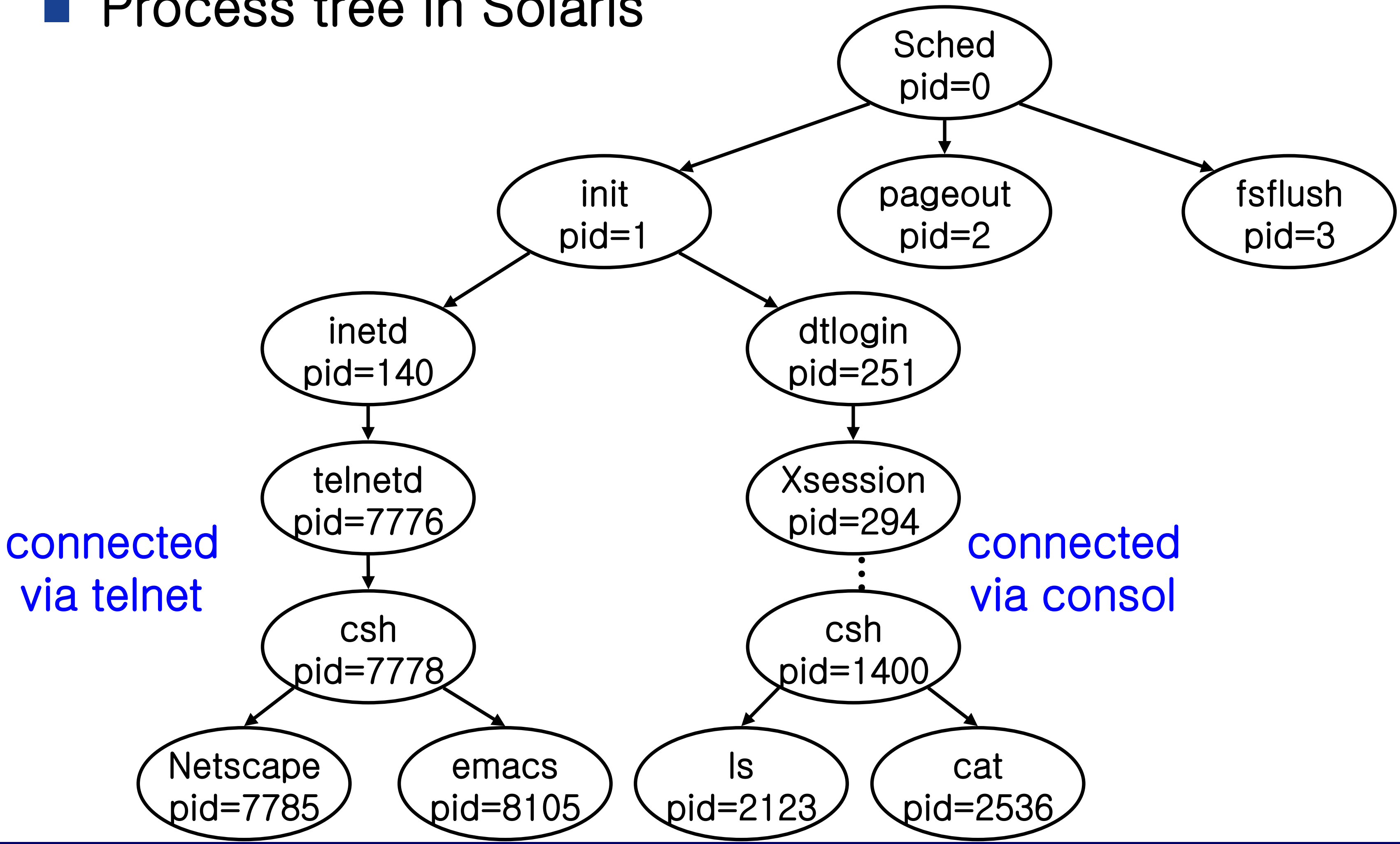
- Create-process system call
  - Creates a process and assigns a pid (process ID).



- 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

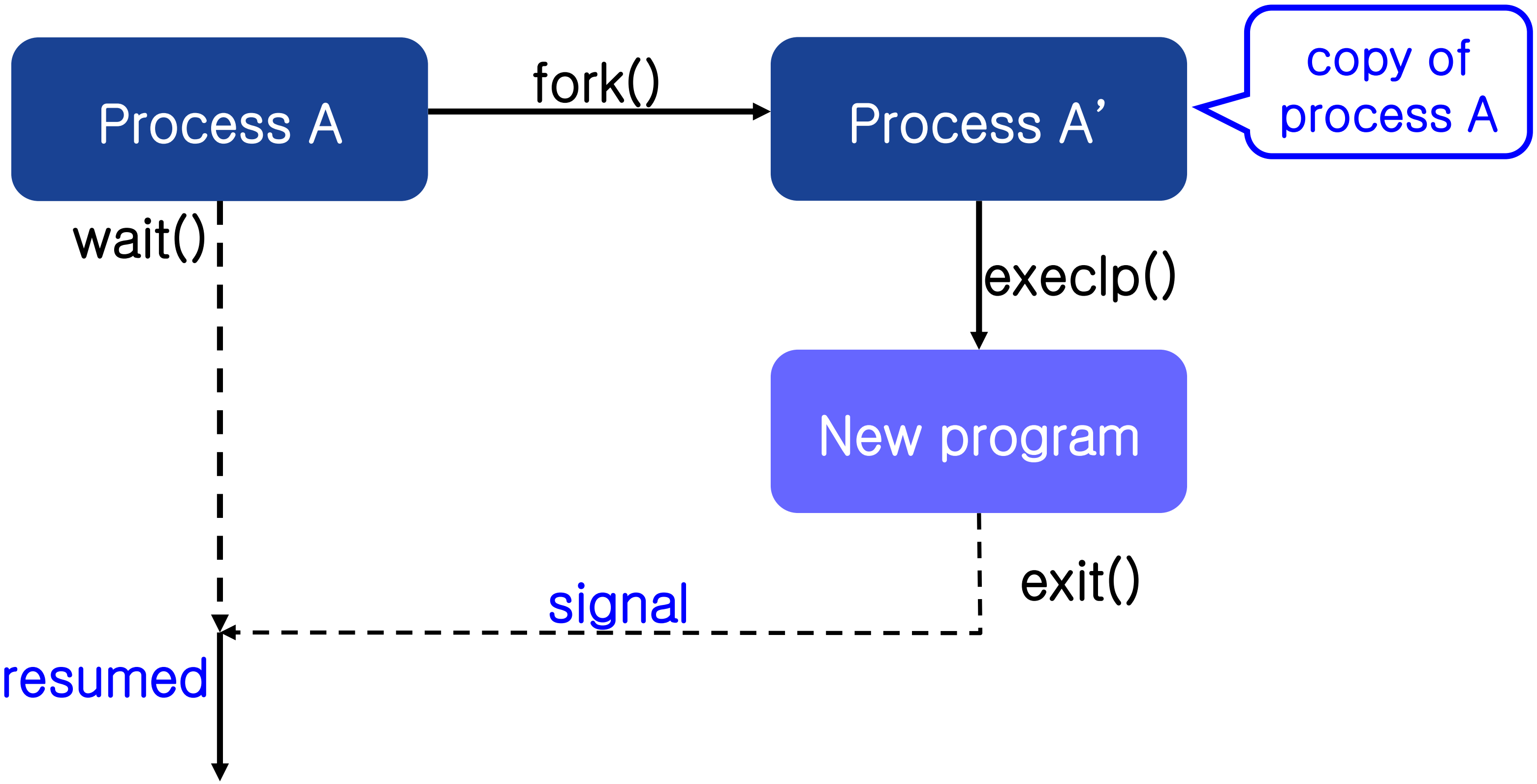
Resource	Child requests its own resource directly from OS or A subset of parent's resource is shared
Execution	Concurrent execution or Parent waits until child is terminated
Address space	Program code and data are shared or Child process has a new program loaded into it

Process Creation in UNIX

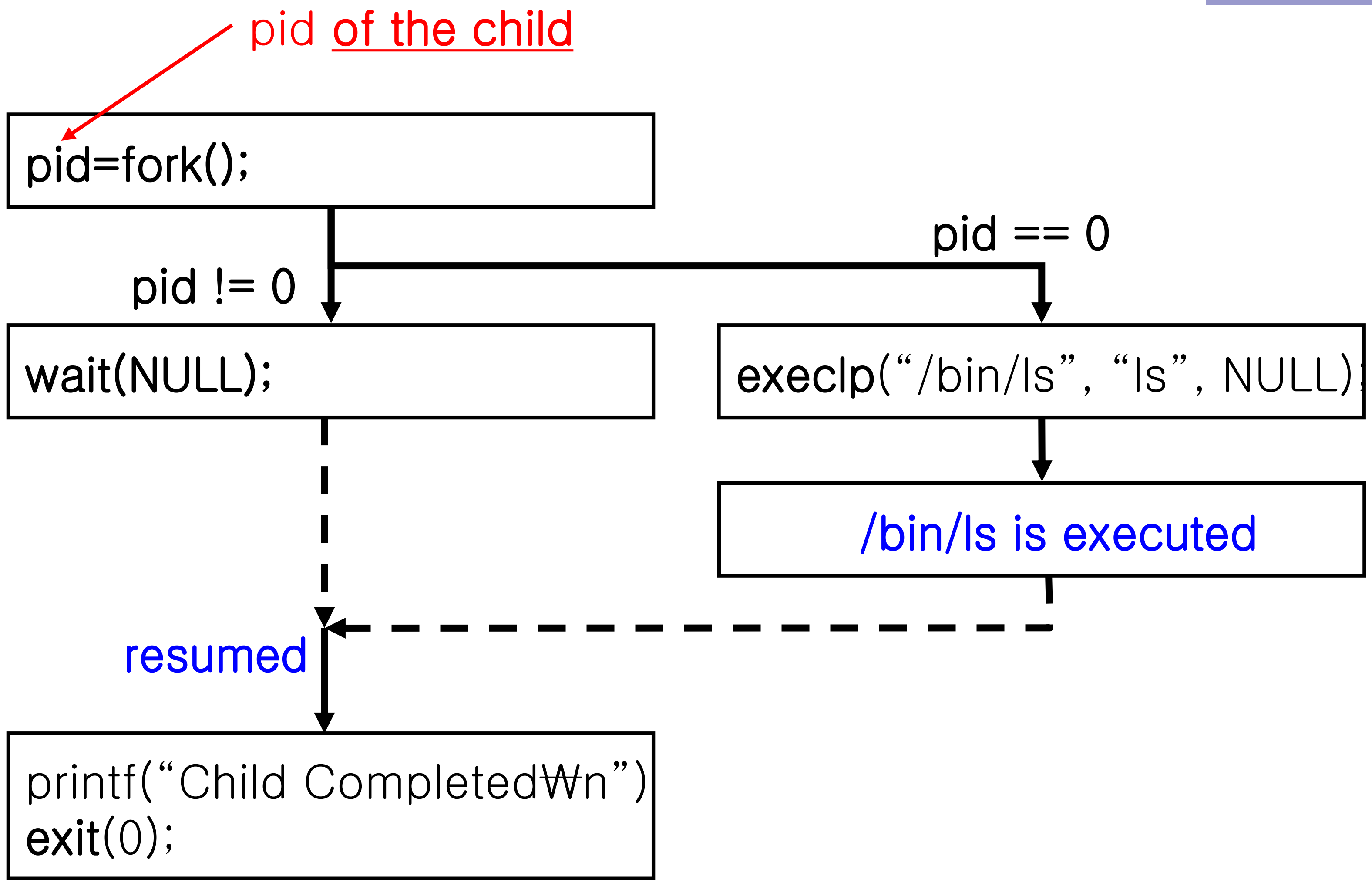
- UNIX system calls related to process creation
  - fork(): create process and returns its pid
    - In parent process, return value is pid of child
    - In child process, return value is zero
  - exec() family: execute a program. The new program substitutes the original one.
    - execl(), execv(),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

Example of Process Creation

- Executing other program



Example of Process Creation



Example of Process Creation

```
int main()
{
   pid_t pid = fork();            // create a process
   if(pid < 0){                   // error occurred
      fprintf(stderr, “fork failed₩n”);
      exit(-1);
   } else if(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);
   }
}
```

Example of Process Creation

- Parent process

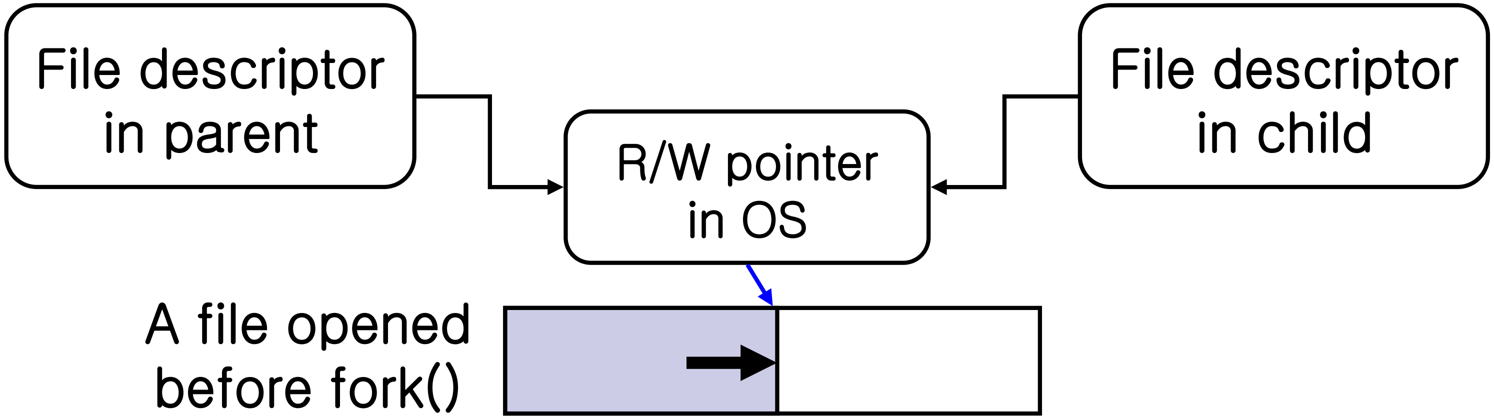
```
int main()
{
   pid_t pid = fork();
   if(pid < 0){
      fprintf(stderr, "fork failed₩n");
      exit(-1);
   } else if(pid == 0){
      execlp("/bin/ls", "ls", NULL);
   } else {
      wait(NULL);
      printf("Child Completed₩n")
      exit(0);
   }
}
```

- Child process

```
int main()
{
   pid_t pid = fork();
   if(pid < 0){
      fprintf(stderr, "fork failed₩n");
      exit(-1);
   } else if(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 **pid** returned from **fork()**
  - Files
    - Opened before fork(): shared with parent
    - Opened after fork(): not shared



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 child process that is alive
  - -1 means it has no child process

Process Creation in win32

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

Agenda

- Overview
- Process scheduling
- Operations on processes
- **<u>Inter-process communication</u>**
- 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

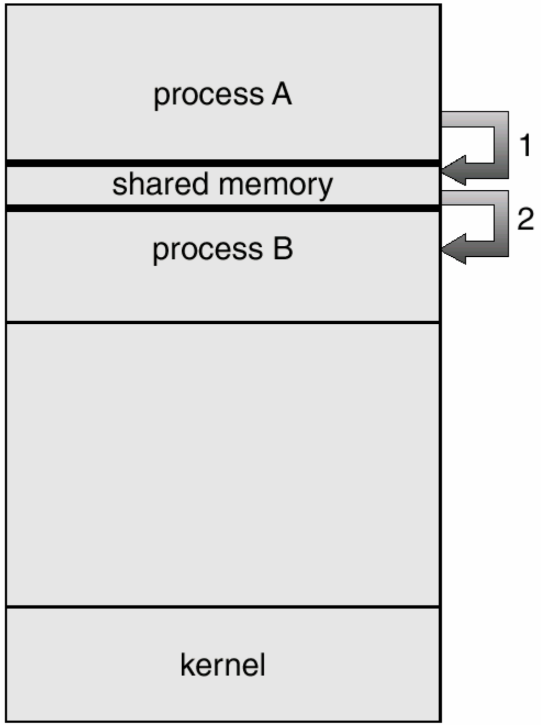
- IPC Models
  - Shared-memory model
  - Message-passing model

Shared-Memory Systems

- Shared-memory segment
  - 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

Example) producer-consumer problem



Producer-Consumer Problem

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

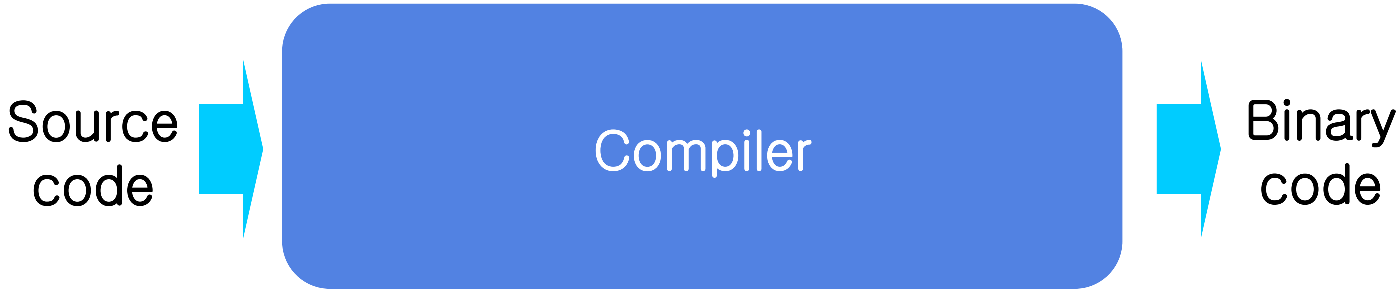
  - Producer: produce information for consumer
  - Consumer: consume information written by producer

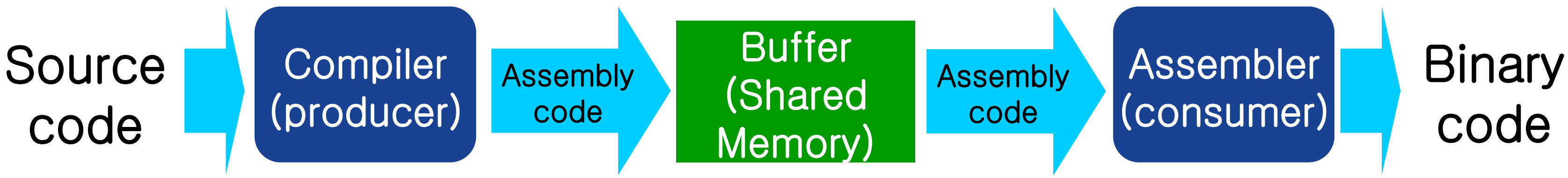
  Ex) compiler – assembler, server – client

  Note! Producer and consumer should be synchronized.

  → Discussed in chapter 6

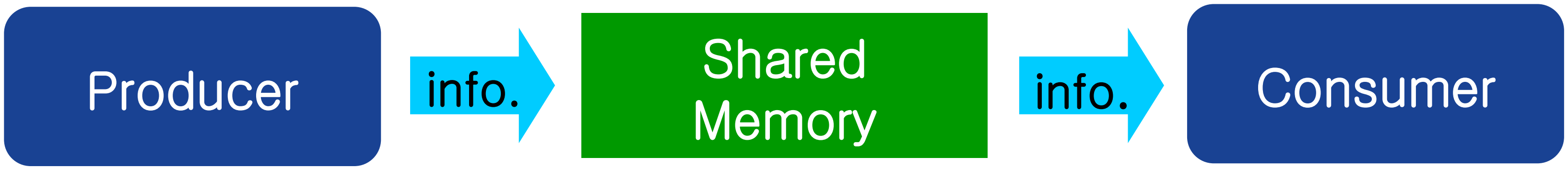
Producer–Consumer Problem





Producer–Consumer Problem

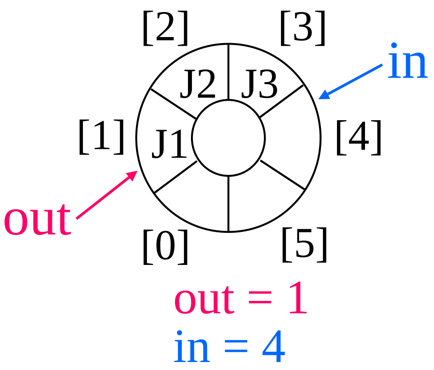
- Two types of buffer
  - Unbounded buffer
    - No practical limit on buffer size
    - Producer can always produce
  - Bounded buffer
    - Producer must wait if buffer is full.



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

Producer-Consumer Problem using Bounded Buffer

- Producer

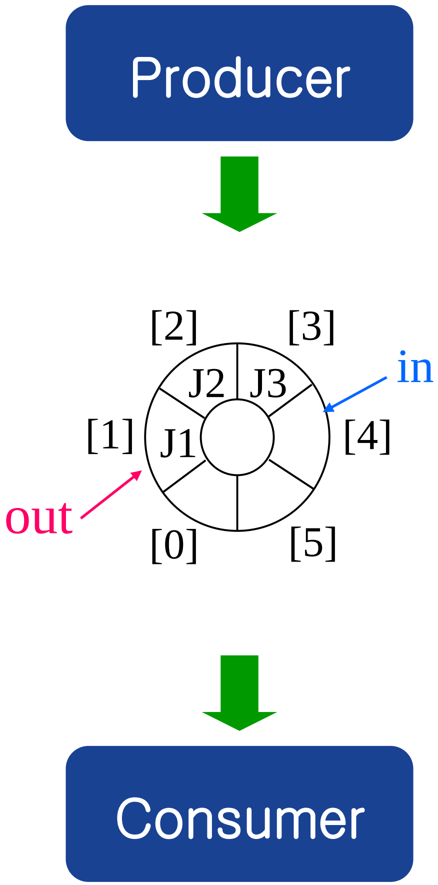
```
item nextProduced;

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
}
```

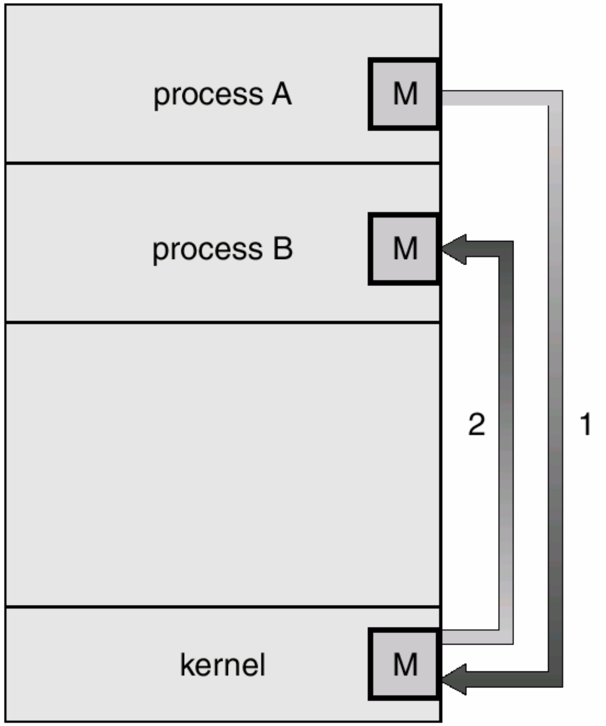


Message-Passing Systems

- 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

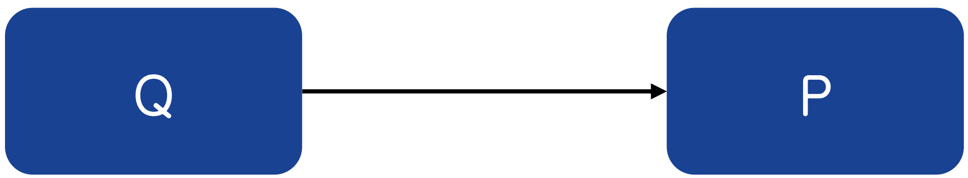
- Essential operations
  - send(message)
  - receive(message)

- (Logical) Implementation methods
  - Direct/indirect
  - Synchronous/asynchronous
  - Buffering
    - Zero/bounded/unbounded capacity

  ➔ Reading assignment: read the textbook for detail.

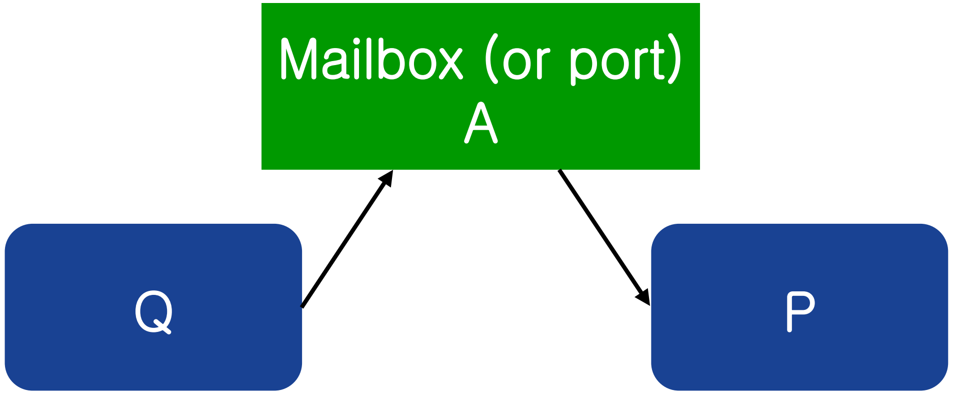
Direct/Indirect Communication

- Direct communication: connection link directly connects processes



send(P, message) receive(Q, message)

- Indirect communication: processes are connected via mailbox



send(A, message) receive(A, message)

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
- <u>Example of IPC system</u>
- Communication in client-server systems

Examples of IPC Systems

- Shared-memory (POSIX)

Reading Assignment: read the following documents to understand how to allocate (shmget), attach (shmat), detach (shmdt), and deallocate (shmctl) shared memory block.

  - www.xevious7.com/linux/lpg_6_4_4.html (Korean)
  - www.cs.cf.ac.uk/Dave/C/node27.html (English)

- Message-passing (MACH)

- Local Procedure Call (Windows XP)
  - Undocumented internal API

[POSIX] Shared-Memory

- Create shared memory

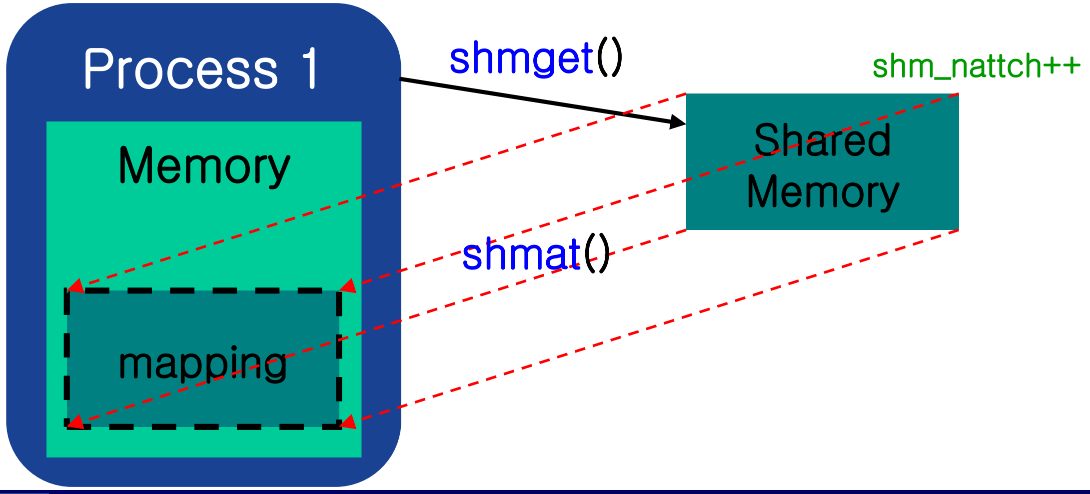
  int shmget (key_t key, int size, int shmflg);

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

- 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)



[POSIX] Shared-Memory

- Use shared memory through attached address as ordinary memory

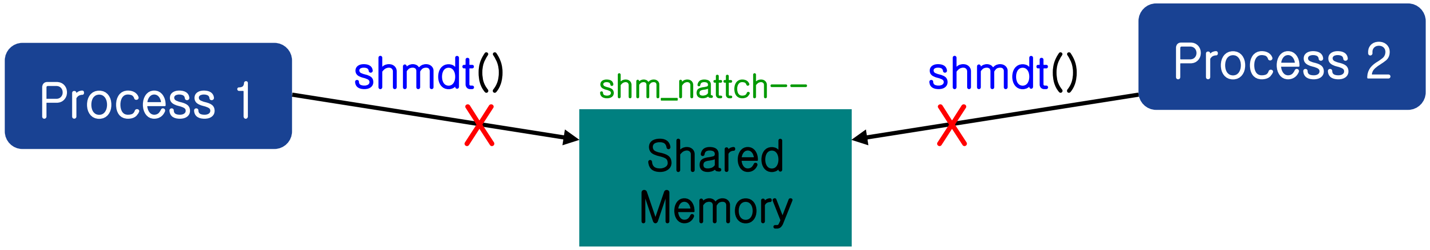
  Ex) sprintf(shared_mem, "Writing to shared memory");

- 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.



[POSIX] Shared-Memory

- Deallocating a shared memory block

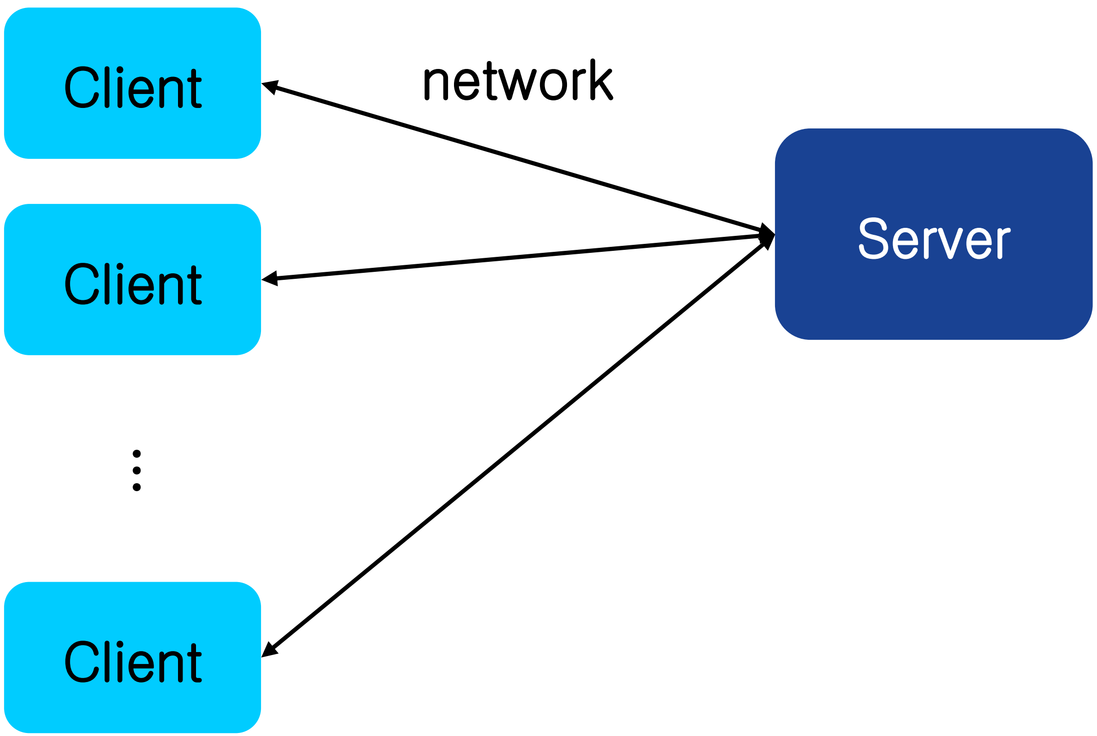
  shmctl(shmid, IPC_RMID, NULL);

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

Agenda

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

Client–Server



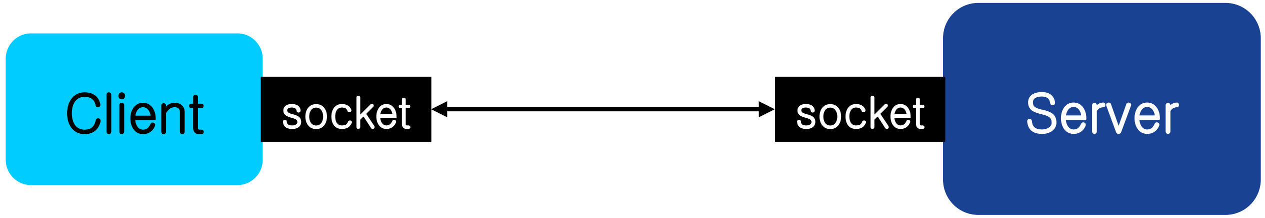
Communications in Client-Server Systems

- Socket
  - Data communication
- RPC (Remote Procedure Call)
  - Procedure call between systems
    - Procedural programming
- RMI (Remote Method Invocation) of JAVA
  - Invocating method of <u>object</u> in other system
    - Object oriented programming

Socket

- Socket: logical endpoint for communication

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



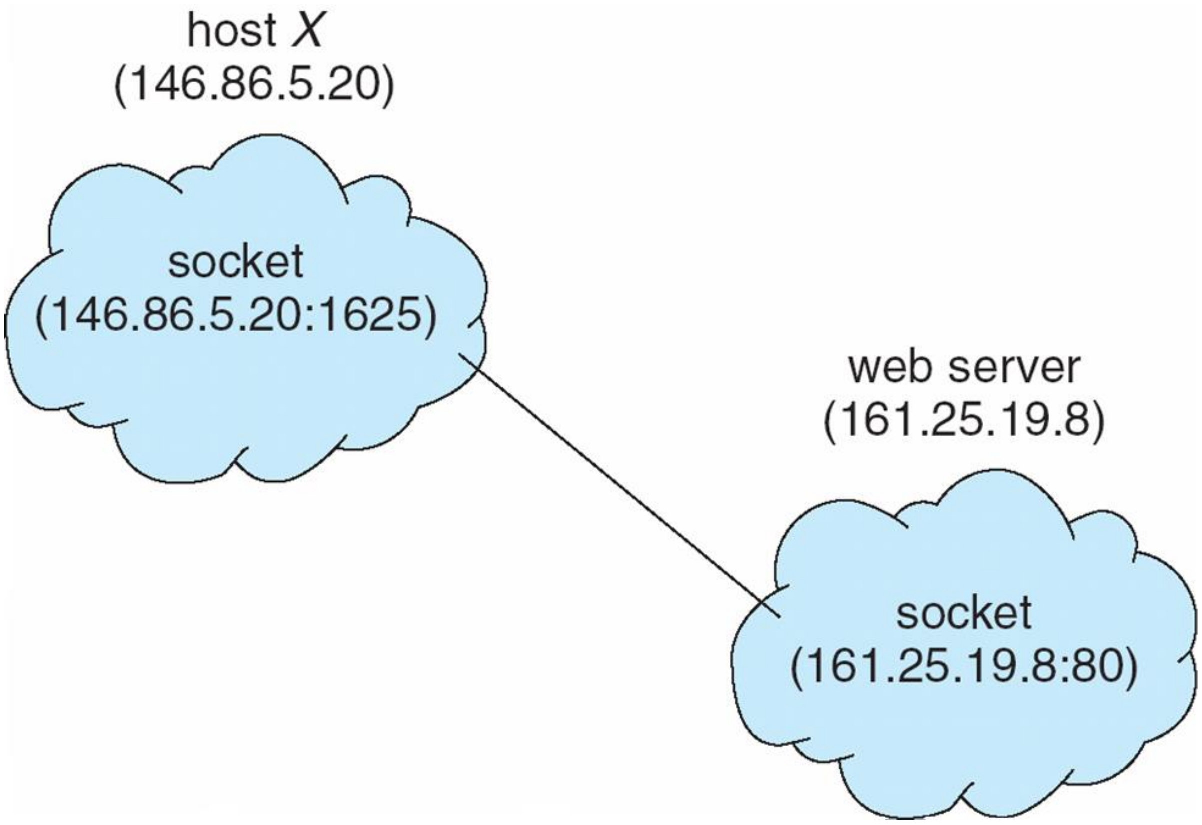
  - Each connection is identified by a pair of sockets.

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
    - 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.

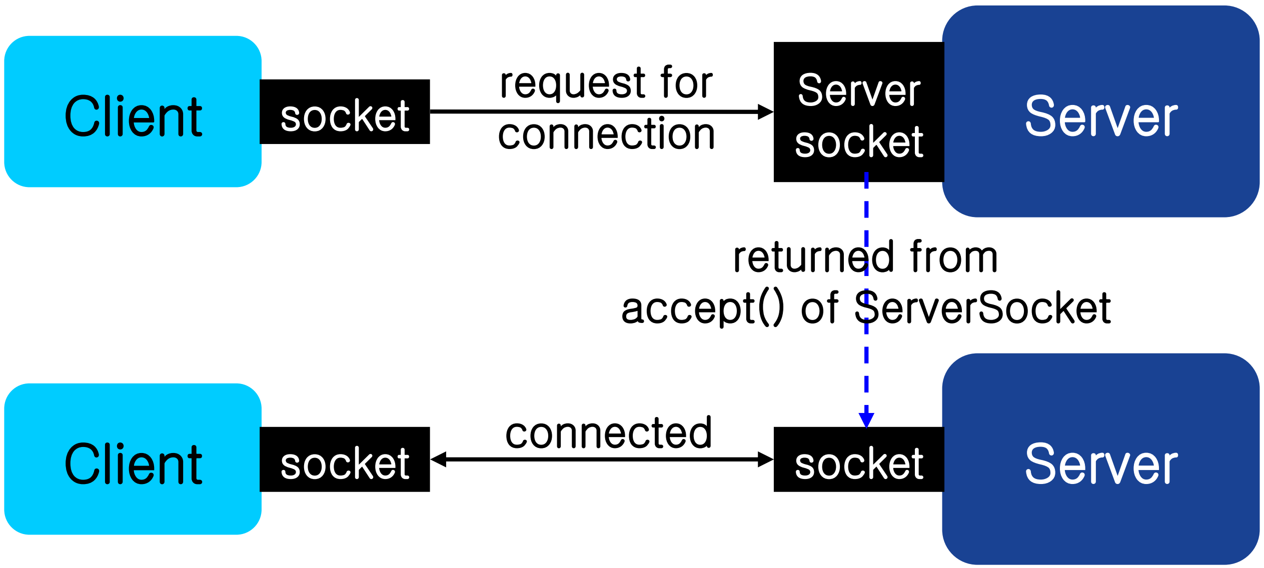
    Ex) a client 146.86.5.20 assigned a port 1625
  - Client request a connection to server.

    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>

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.getOutputStream(), true);

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

client.close();
```

- Client (given *sock*)

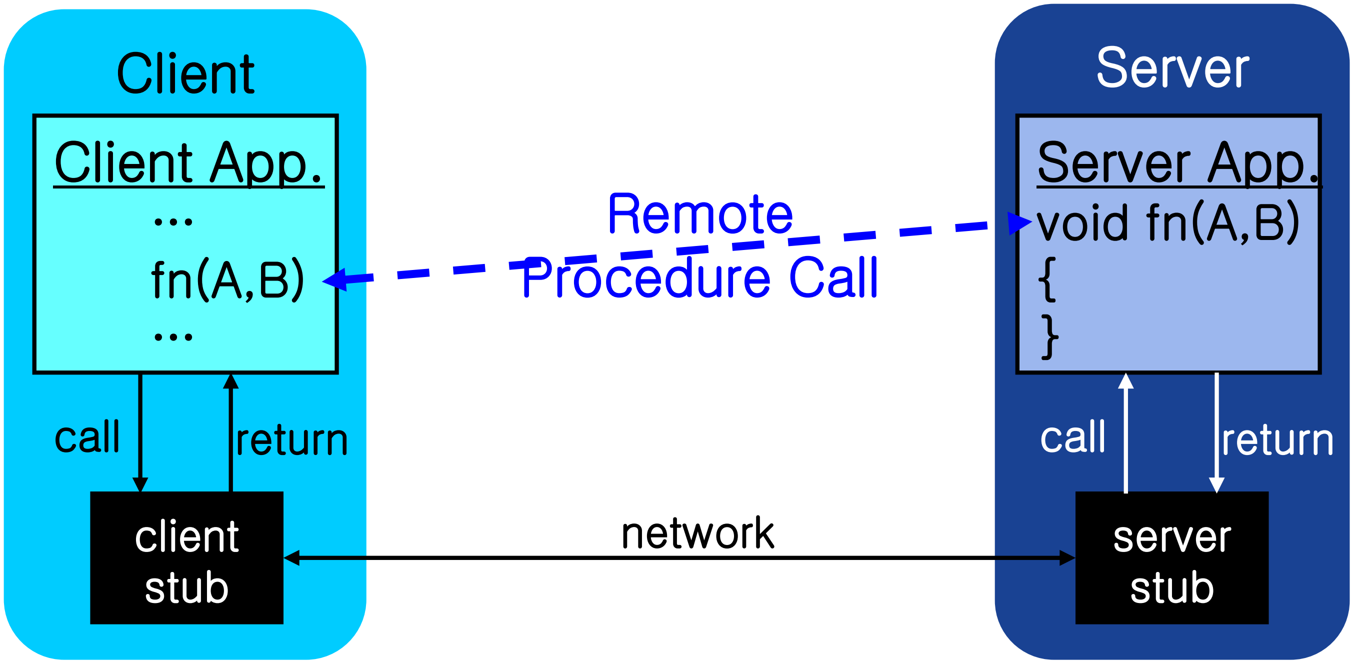
```
InputStream in =
    sock.getInputStream();
BufferedReader bin = new
    BufferedReader(new
    InputStreamReader(in))

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

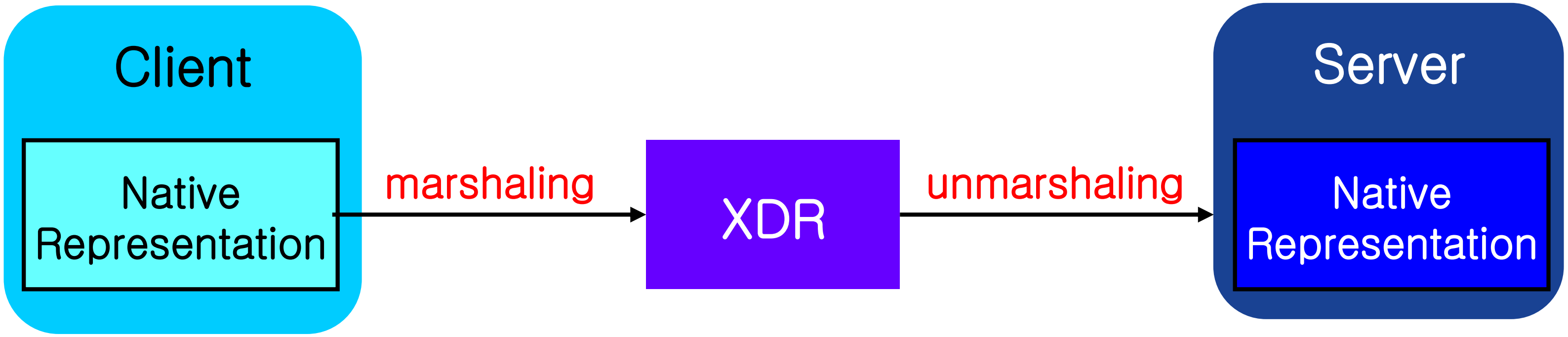


Remote Procedure Calls

- 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 <u>**standard format**</u>
    - <u>XDR: eXternal Data Representation</u>
  - Marshalling: native representation -> XDR
  - Unmarshalling: XDR -> native representation



RPC Reference Sites

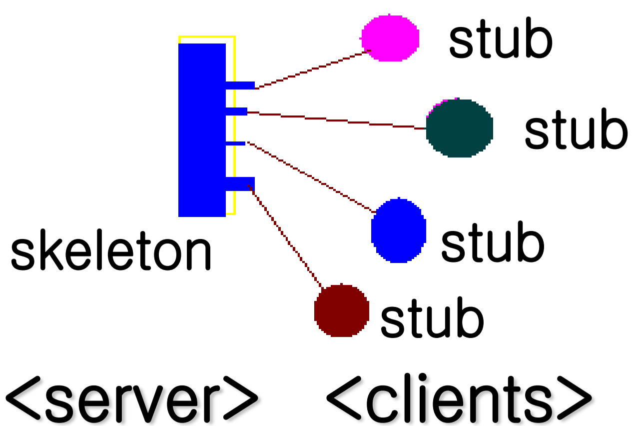
- Windows
  - MSDN RPC page: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/rpcank.asp

- Unix
  - Document about *rpcgen*.

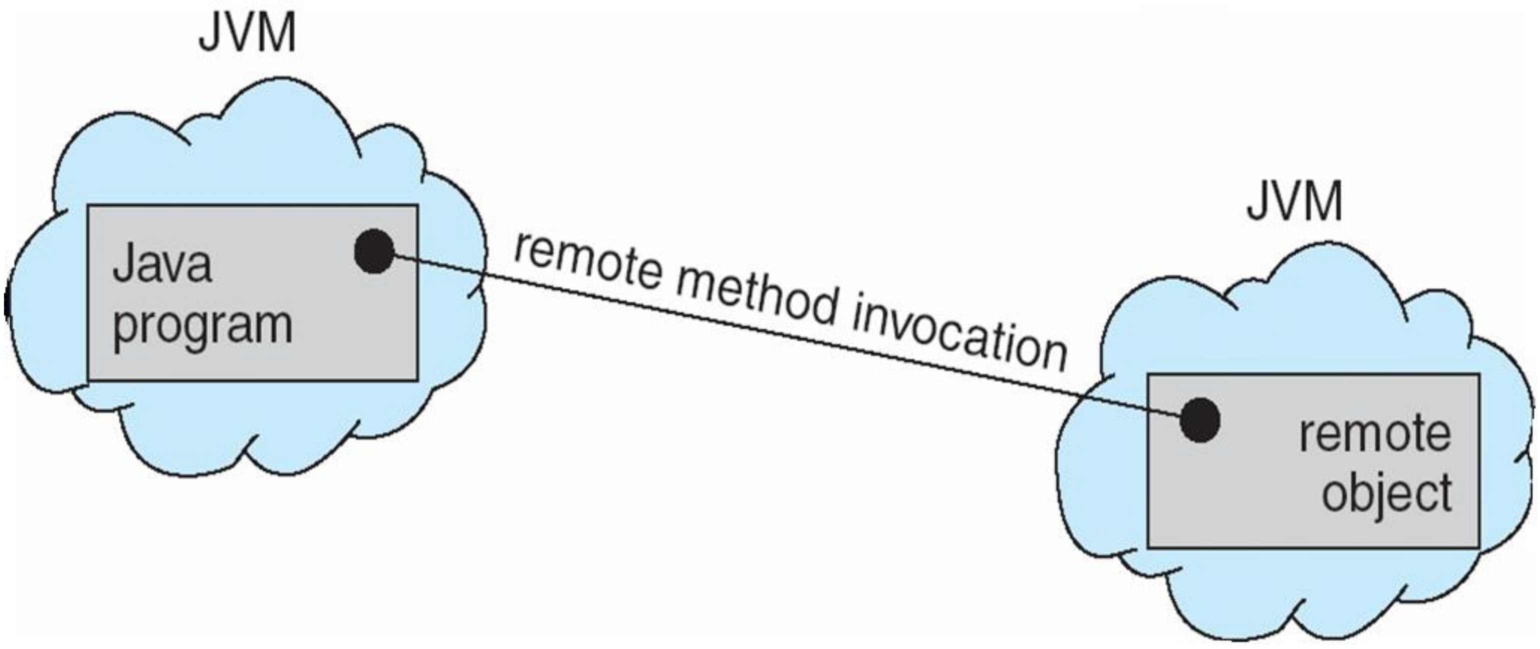
Remote Method Invocation (RMI)

- 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)