Chapter 3: Processes

Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Objectives

- To introduce the notion of a **process -- a program in execution**, which forms the basis of all computation
- To describe the various features of processes, including **scheduling, creation and termination, and communication**
- To explore **interprocess communication** using shared memory and message passing
- To describe **communication in client-server systems**

Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – **a program in execution**; process execution must progress in sequential fashion

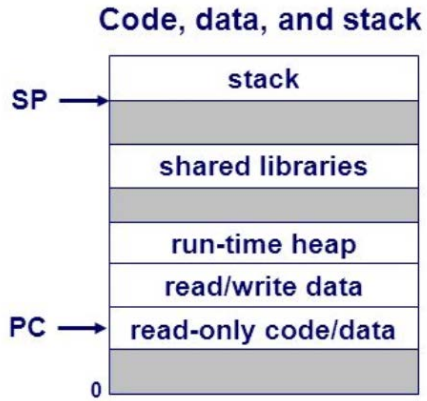
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing **global variables**
  - **Heap** containing memory **dynamically allocated during run time**

Process Concept (Cont.)

- **Process = process context + code, data and stack**

Program Context:
- Data registers
- Condition code
- Stack Pointer
- Program Counter

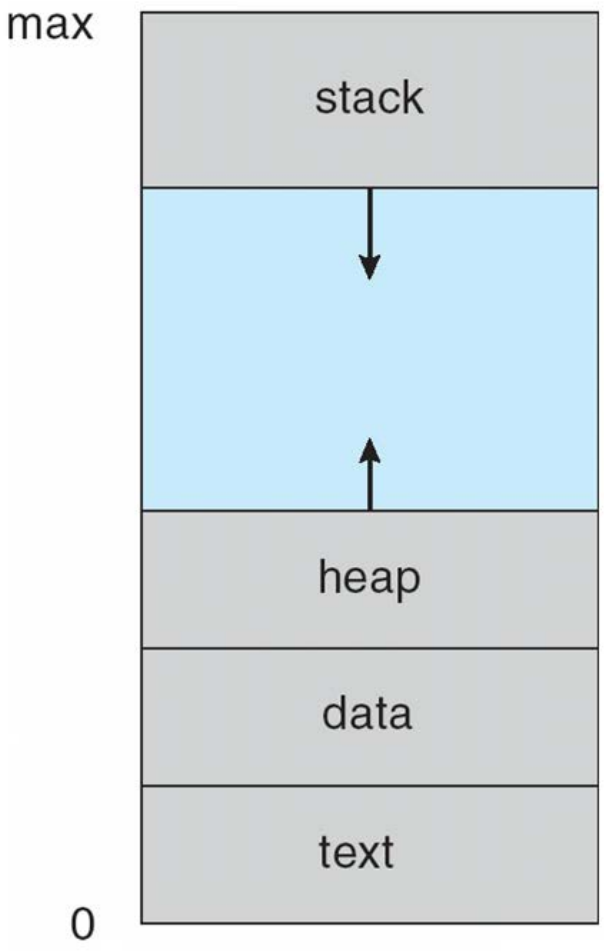
+



- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

Process in Memory

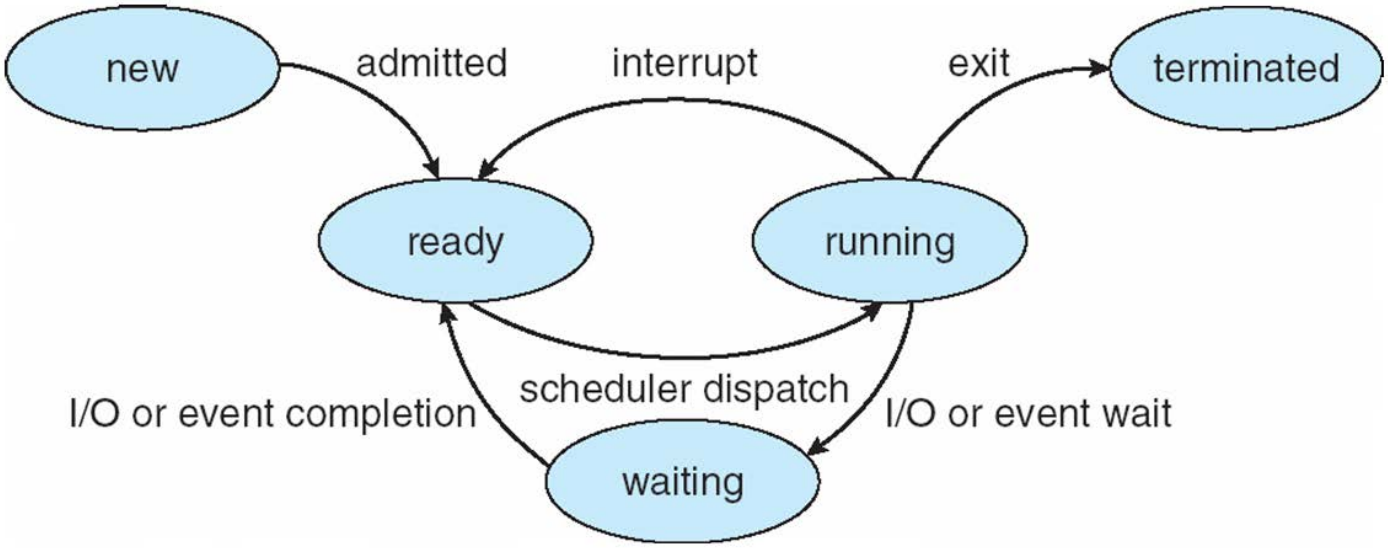
Current activity including **program counter**, processor registers



- **Stack** containing temporary data, function parameters, return addresses, local variables
- **Heap** containing memory **dynamically allocated during run time**
- **Data section** containing **global variables**
- The program code, also called **text section**

Process State

- As a process executes, it changes **state**
  1. **new**: The process is being created
  2. **ready**: The process is waiting to be assigned to a processor
  3. **running**: Instructions are being executed
  4. **waiting**: The process is waiting for some event to occur
  5. **terminated**: The process has finished execution



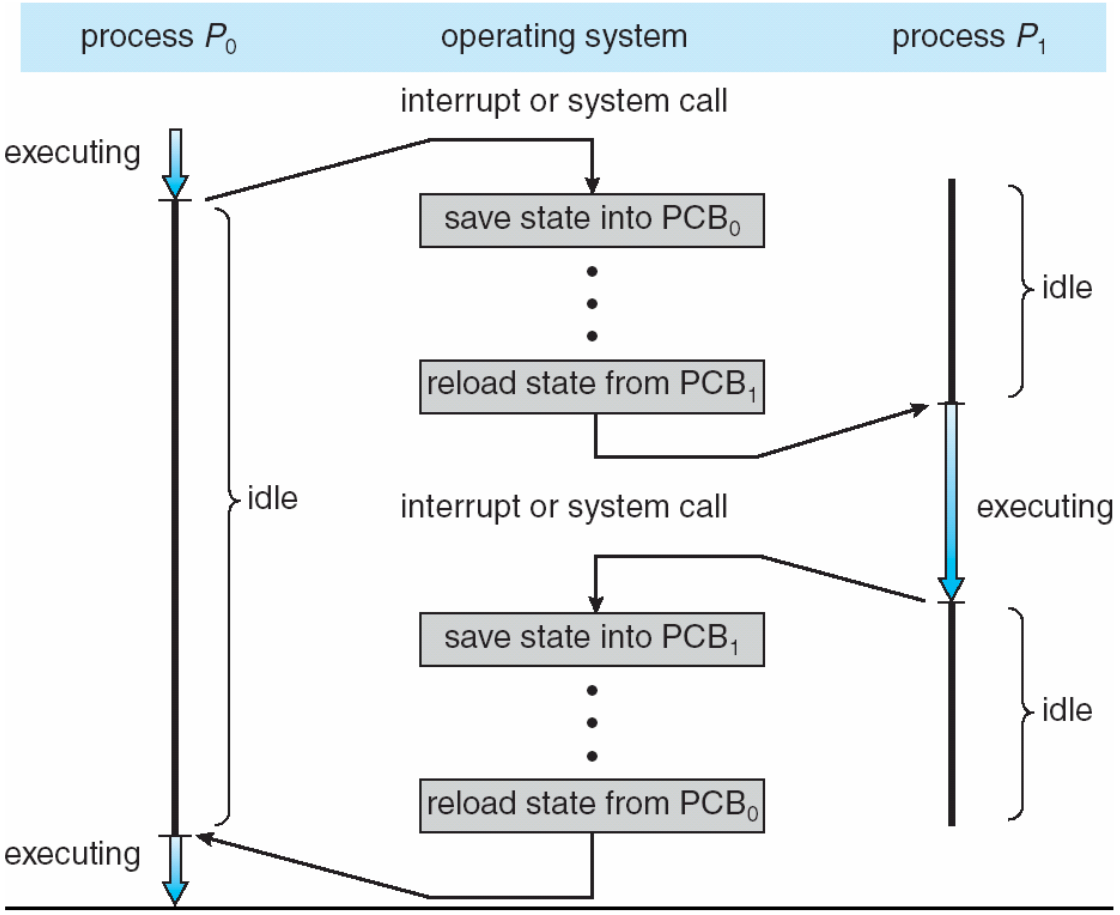
Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

CPU Switch From Process to Process



< Context switching >

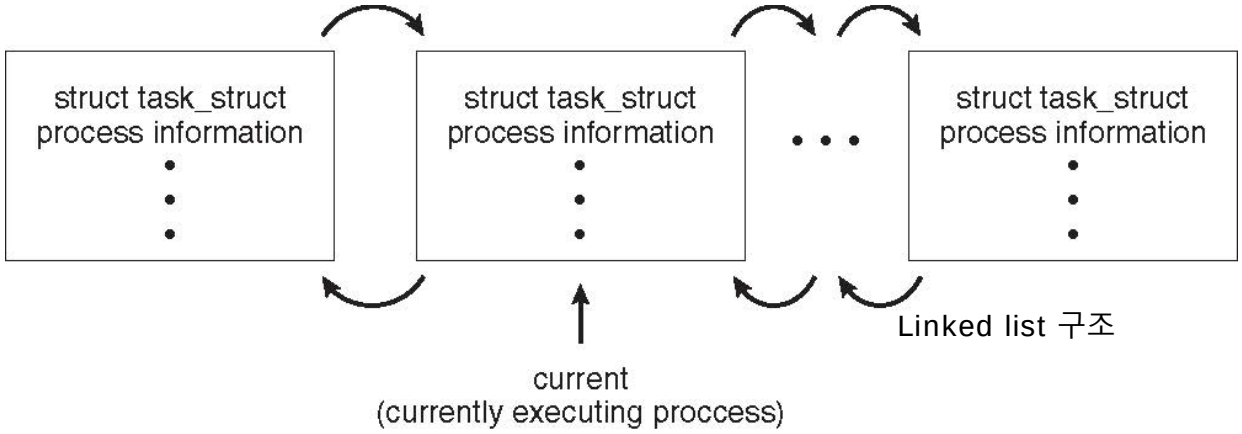
Threads

- So far, process has a single thread of execution
- **Consider having multiple program counters per process**
  - Multiple locations can execute at once
    - **Multiple threads of control** -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- See next chapter

Process Representation in Linux

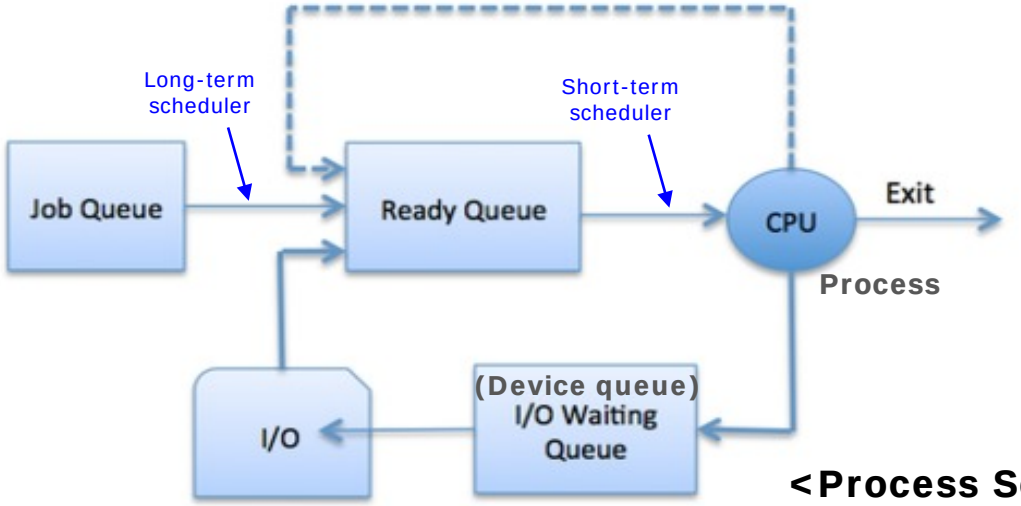
Represented by the C structure **task_struct**

```
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 */
```



Process Scheduling

- **Maximize CPU use, quickly switch processes onto CPU for time sharing**
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

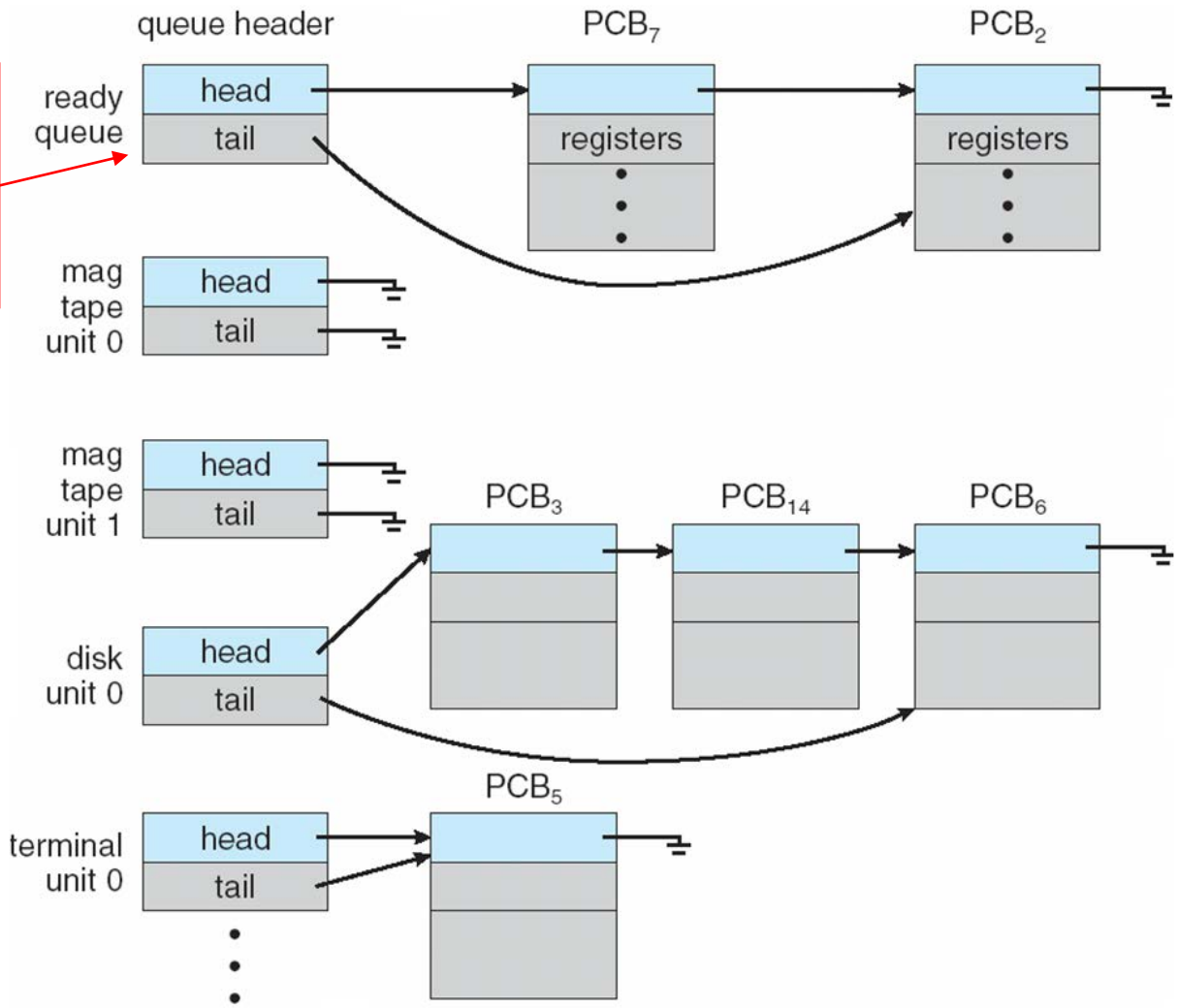


<Process Scheduling queue >

Ready Queue And Various I/O Device Queues

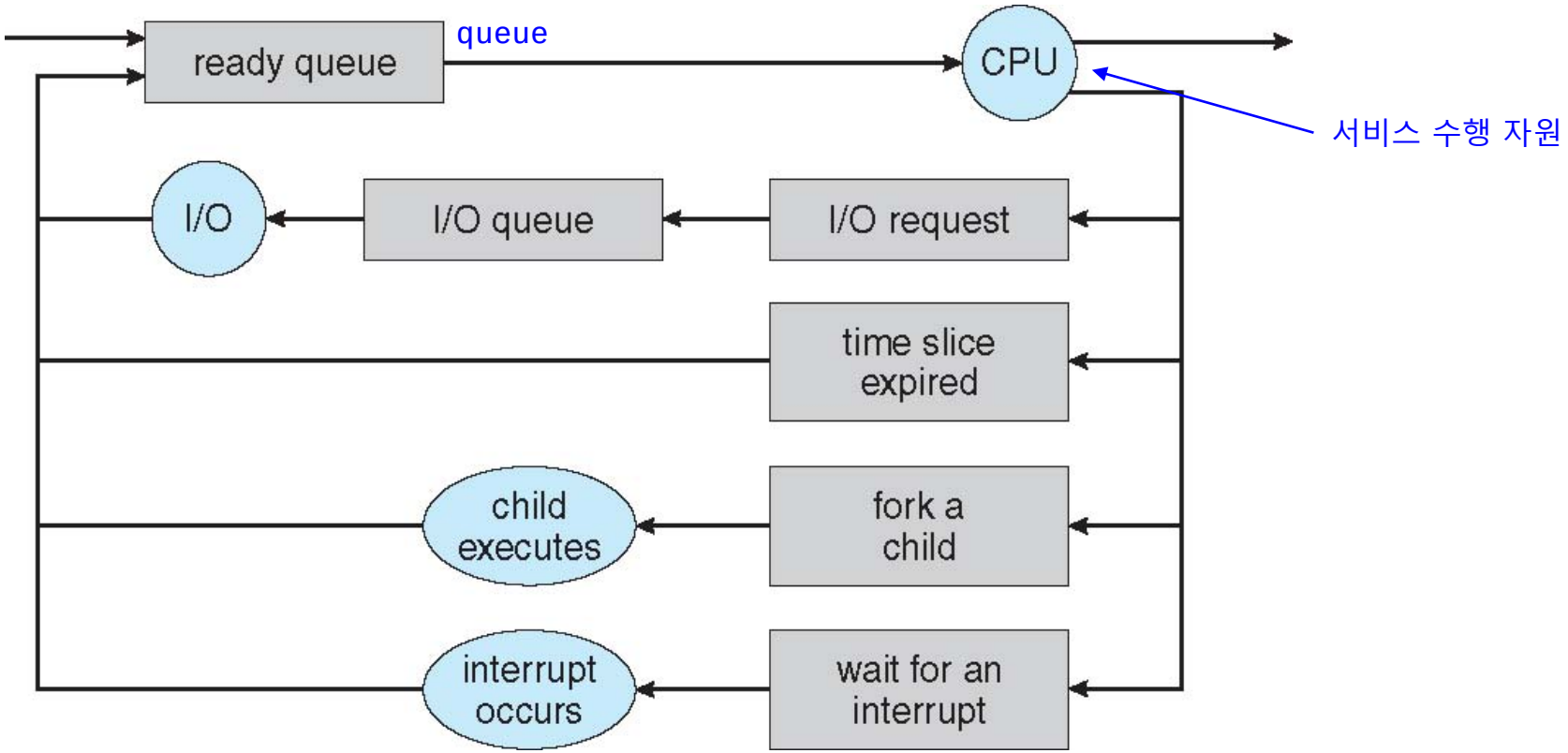
- 실행을 대기하는 process는 ready queue (list 구조임)에 유지됨
- Ready queue(list)의 header는 첫번째 PCB 가리킴
- Tail은 마지막 PCB를 가리킴

다른 device queue도 존재함



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows



- 새 process 는 ready queue에 저장됨(CPU에 dispatch되어 실행될 때까지 queue에서 대기함)
- CPU에 할당되어 실행되면,
  - 해당 Process에서 I/O 요청을 수행할 경우, I/O queue 에서 넣어짐
  - 해당 Process에 할당된 시간이 expire 될 경우, 다시 ready queue에서 기다리게 됨
  - 새로운 child process를 생성할 경우, child process 종료까지 기다리게 됨
  - 해당 process는 interrupt에 의해, 강제로 CPU에서 제거되고 ready queue에서 대기됨

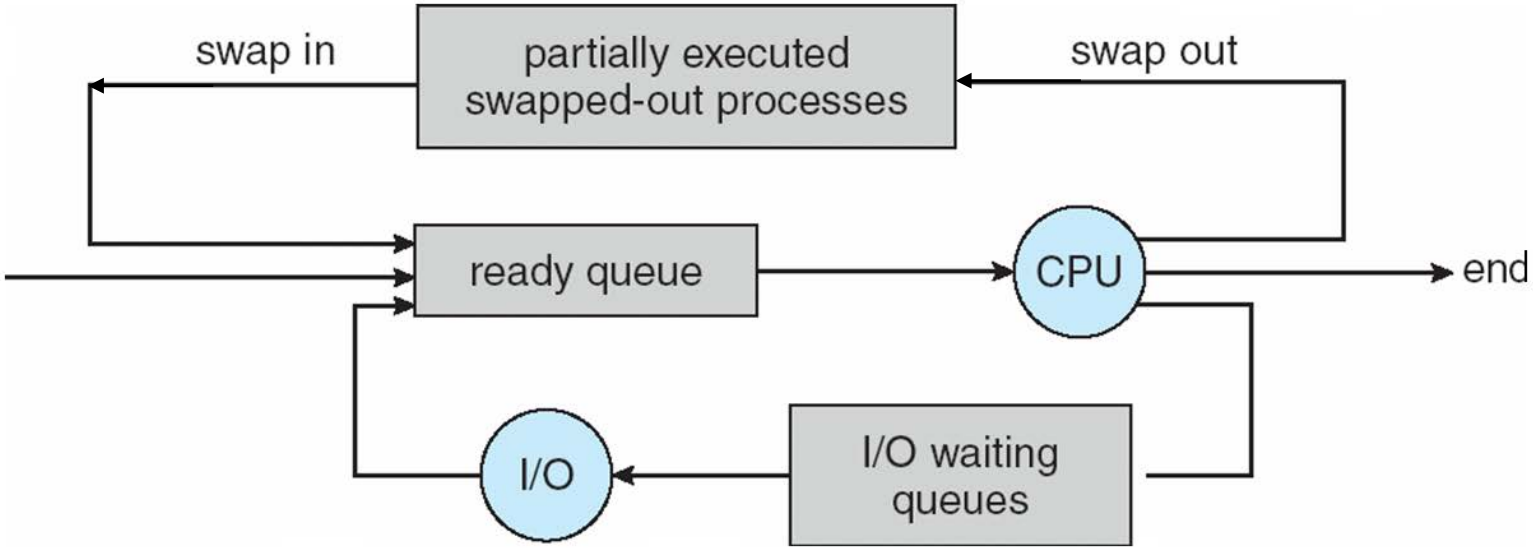
- Process 는 작업 종료까지 이 주기를 반복함. 종료시, 모든 queue에서 삭제되고 PCB와 자원을 deallocate함

Schedulers

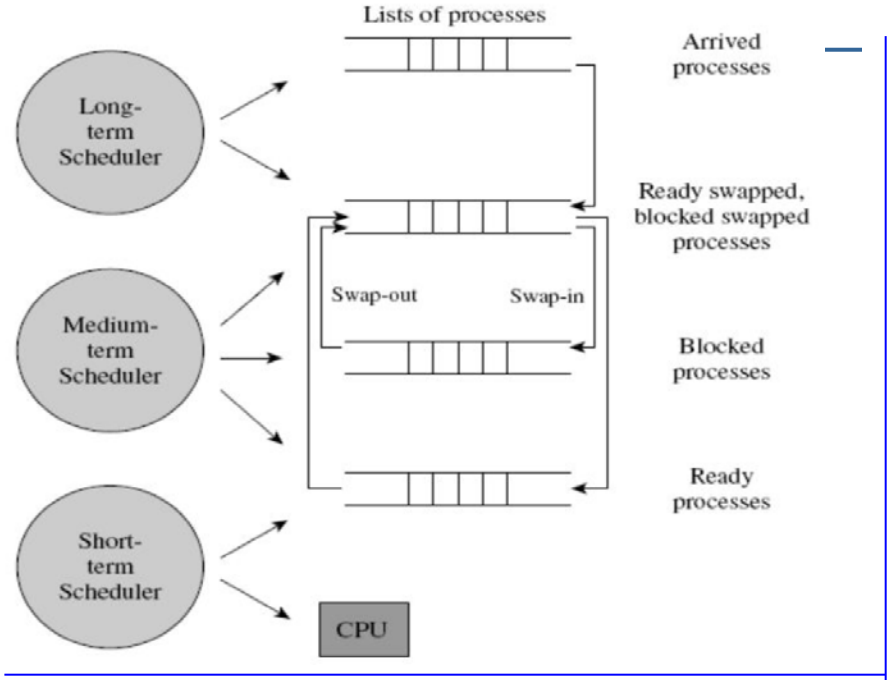
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

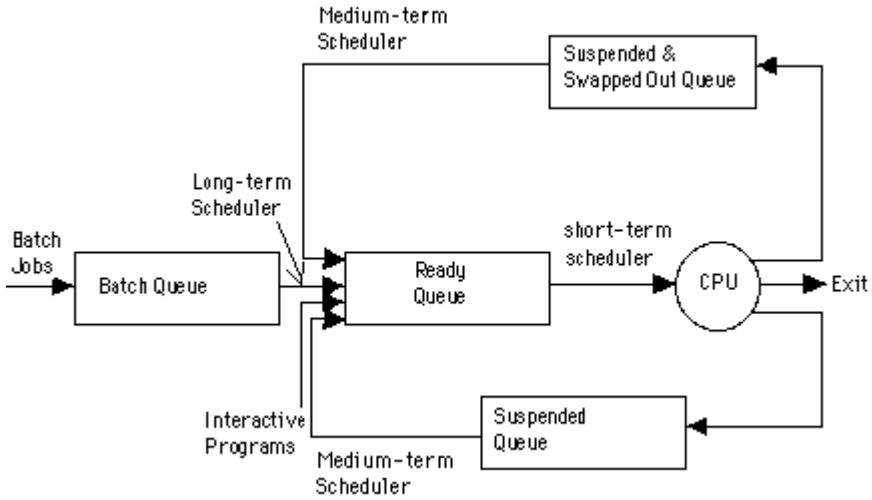
Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Addition of Medium Term Scheduling





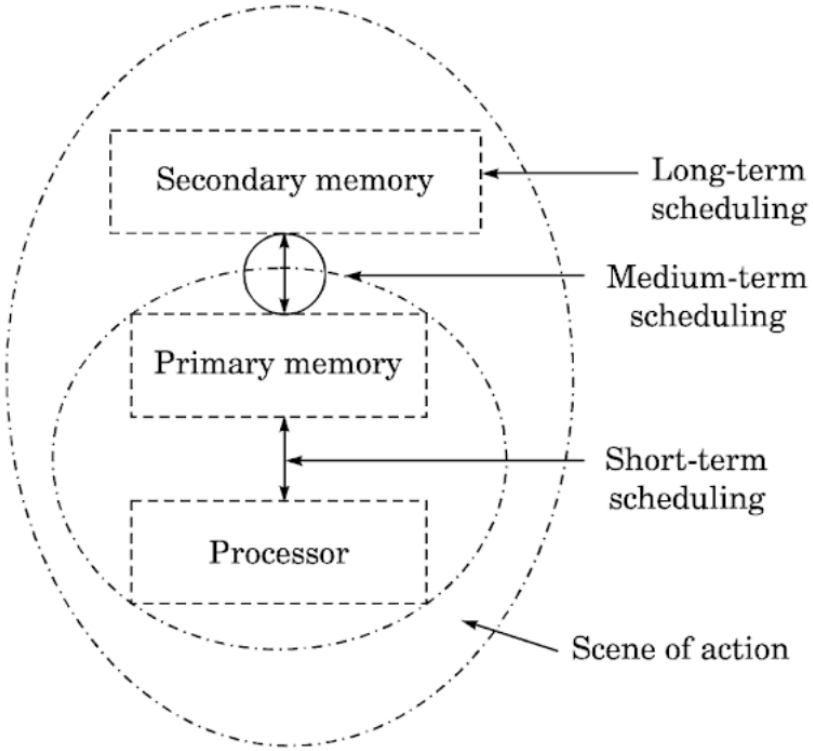


Figure 3.5 Scheduling considerations.

An introduction to Operating Systems Concepts and Practice (GNU/Linux)

http://staff.um.edu.mt/csta1/courses/lectures/csm202/os8.html

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - **Single foreground process-** controlled via user interface
  - **Multiple background processes**– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - **Service** can keep running even if background process is suspended
  - **Service** has no user interface, small memory use

- 참고) 4 main components in Android : Activities, Services, Content Providers, Broadcast receiver

Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and **load the saved state for the new process via a context switch**
- **Context** of a process represented in the PCB
- **Context-switch time is overhead;** the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- **Time dependent on hardware support**
  - **Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once**

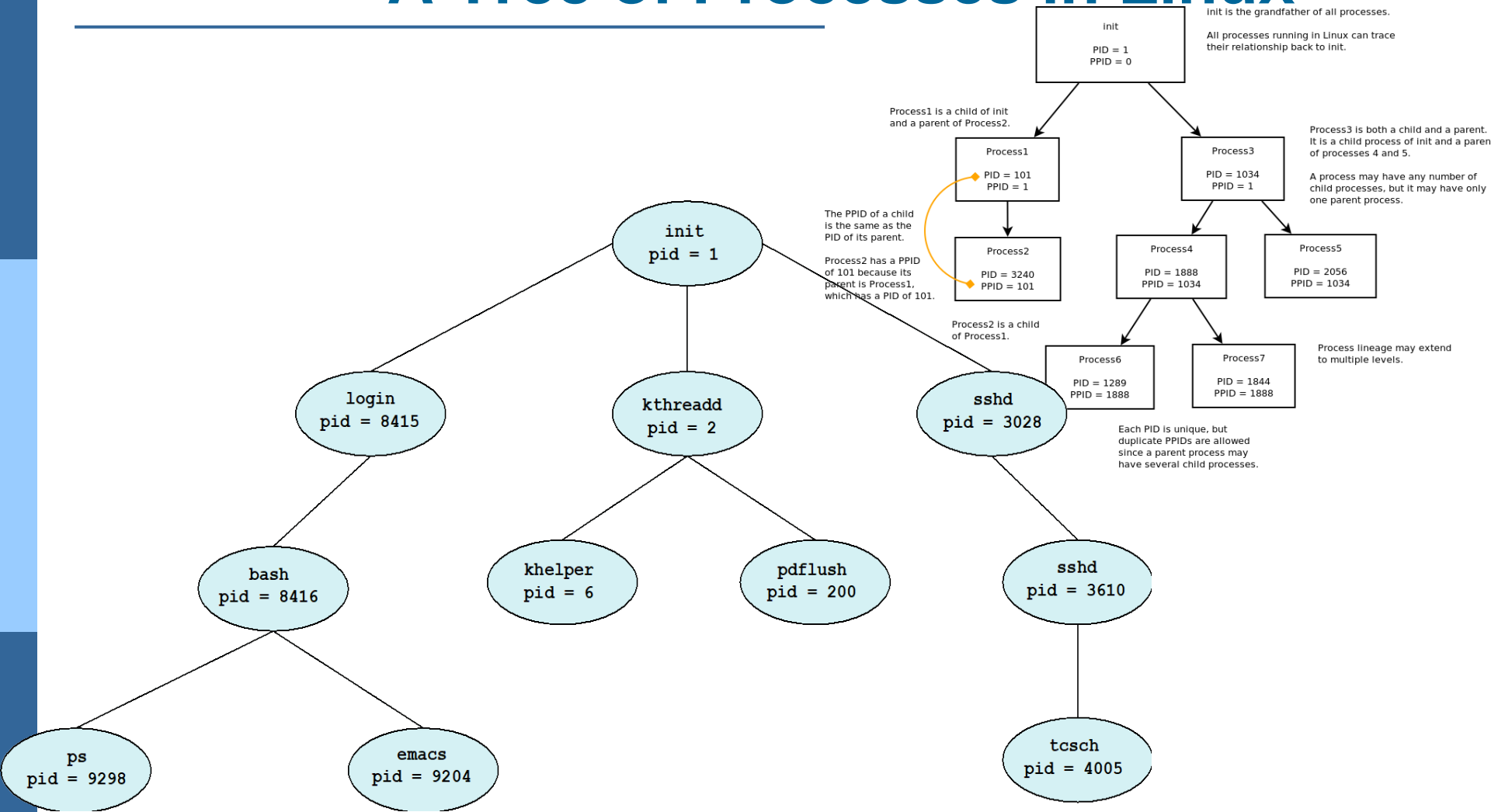
Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

Process Creation

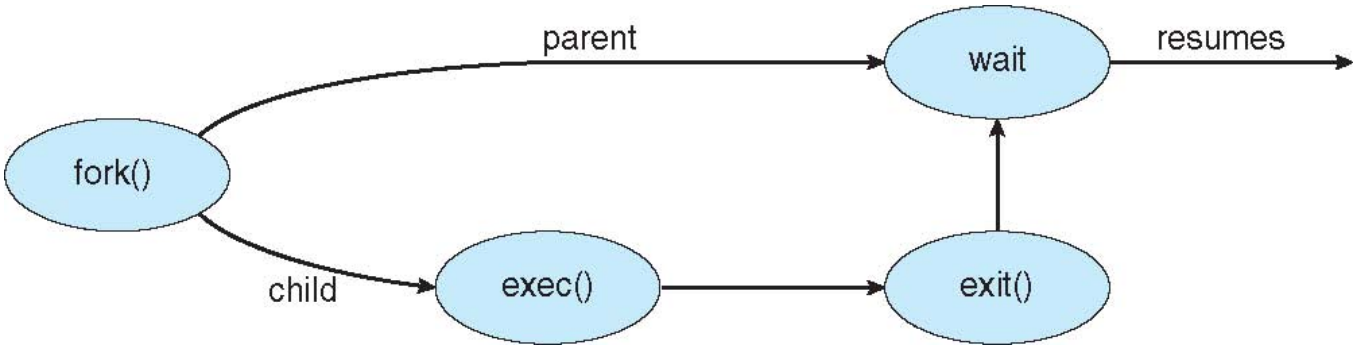
- **Parent process create children processes**, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- **Resource sharing options**
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- **Execution options**
  - Parent and children execute concurrently
  - Parent waits until children terminate

A Tree of Processes in Linux

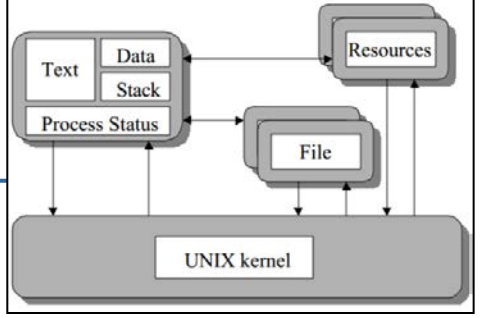


Process Creation (Cont.)

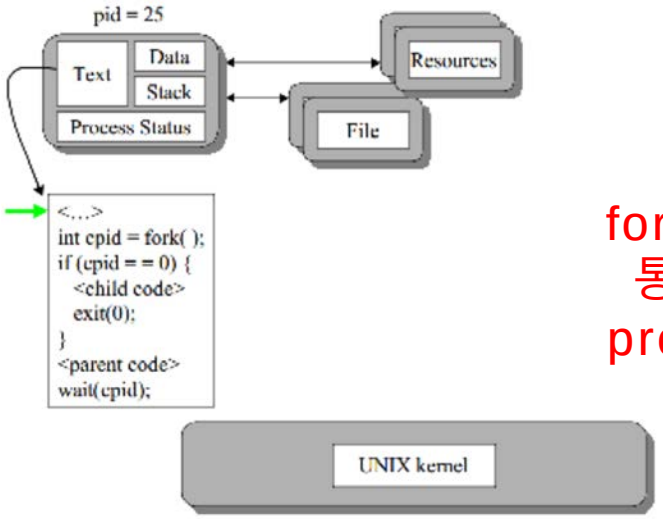
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- **UNIX examples**
  - **`fork()`** system call creates new process
    - parent process pid > 0, child process pid = 0
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



How fork() works

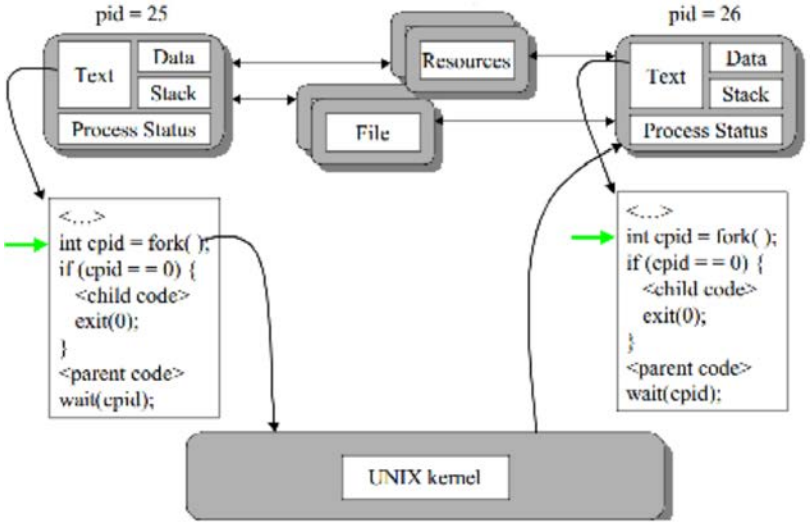


- How Fork works



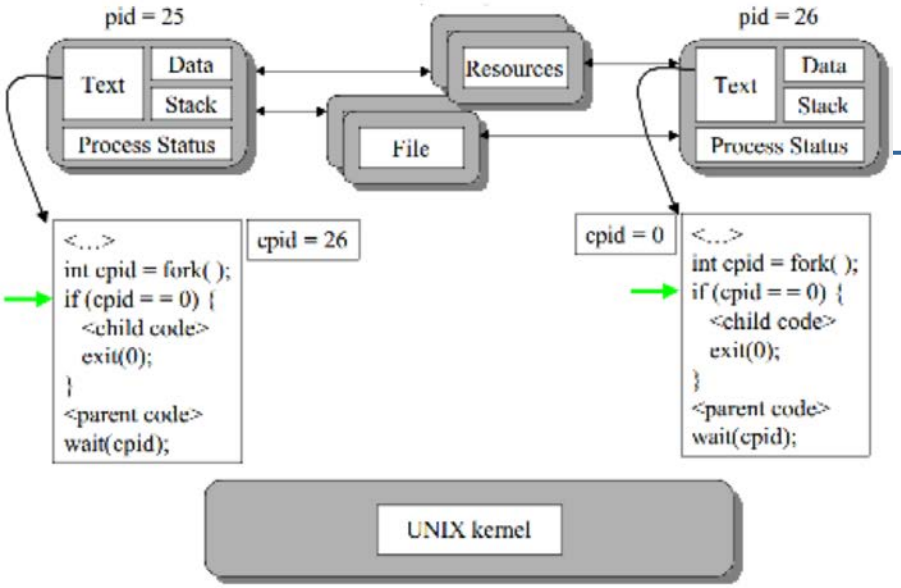
(1) pid 25번 process 실행중

fork() 실행을 통해, child process 생성

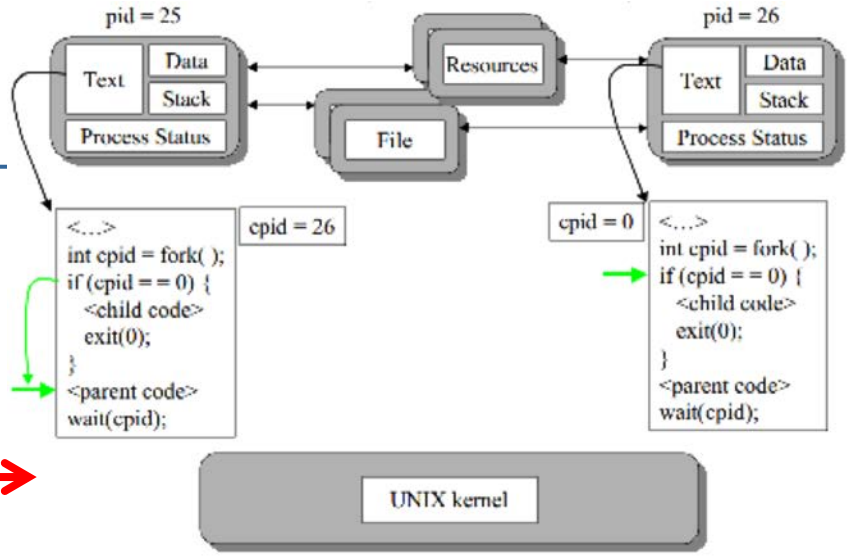


(2) pid 26번 child process 생성됨

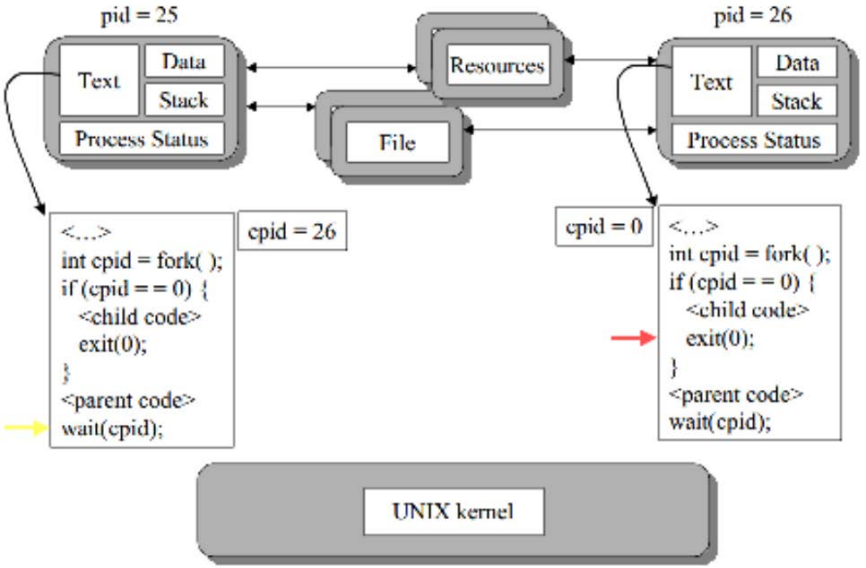
https://stackoverflow.com/questions/1653340/differences-between-fork-and-exec



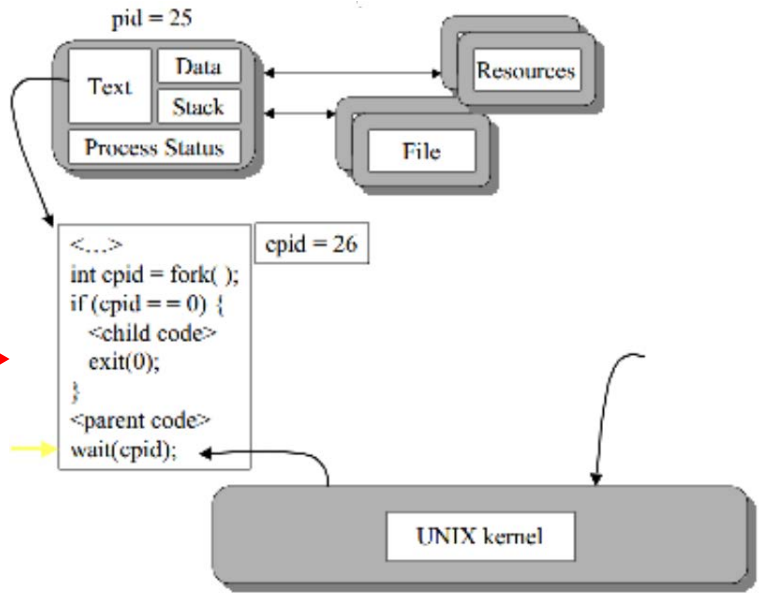
(3) parent process의 fork( ) return값은 26
child process의 fork( ) return 값은 0



(4) fork return값으로 실행 영역이 결정됨

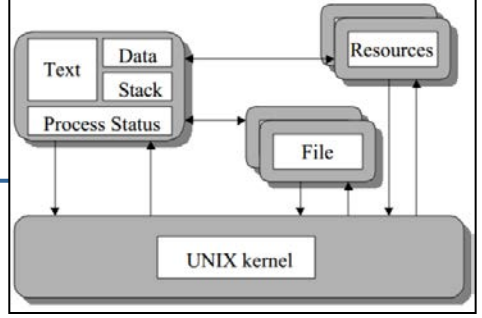


(5) parent process는 child process생성 후, 기다리고 있음. child process는 자신의 코드 실행

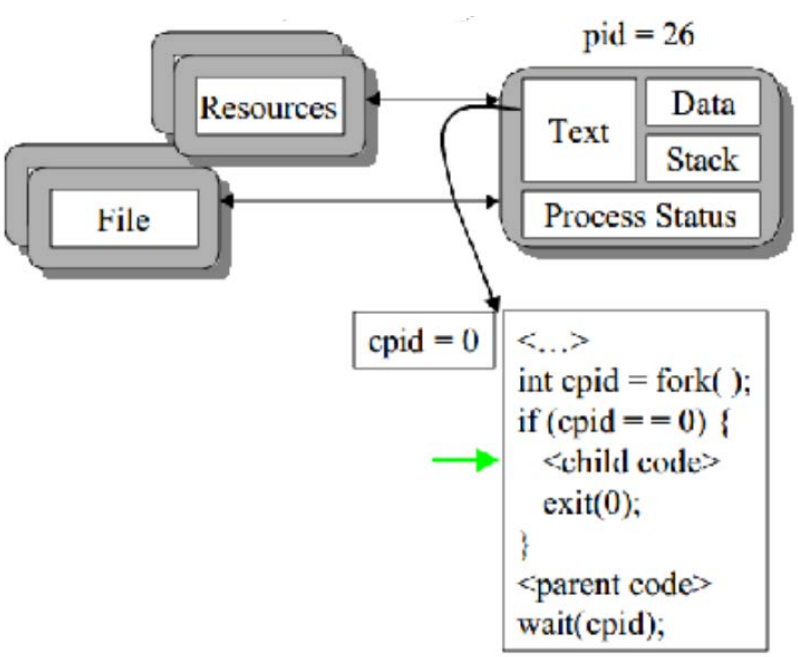


(6) child process 종료 후, parent process 다시 실행

How exec() works

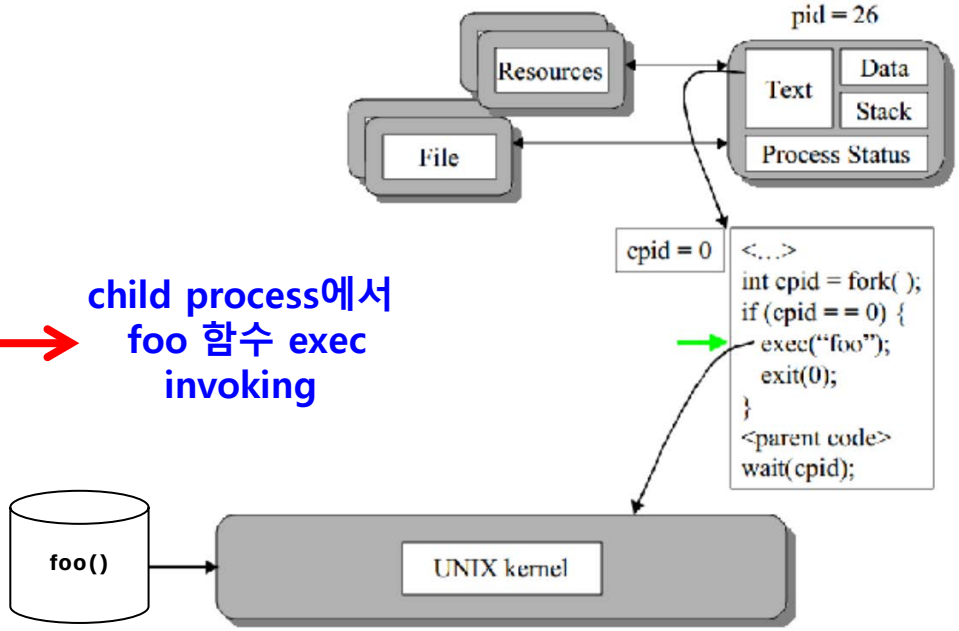


- exec( ) replaces the current process with new process's code, data, stack. Generally follows the structure



(1) child process 코드 실행 시작

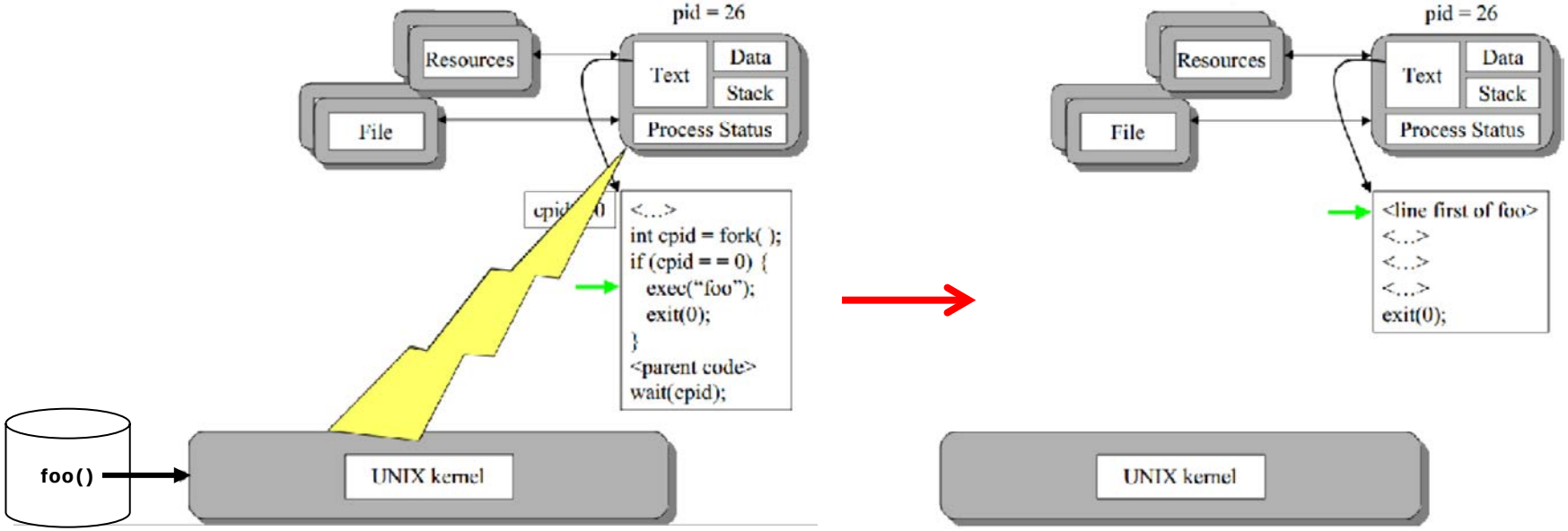
child process에서 foo 함수 exec invoking



(2) child process 에서 foo 함수 invoking

How exec() works

- exec( ) replaces the current process with new process's code, data, stack. Generally follows the structure



(3) foo( ) 함수 관련 정보가 loading 됨

(4) foo( ) 함수 실행됨

C Program Forking Separate Process

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

int main()
{
pid_t pid;

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

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Creating a Separate Process via Windows API

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - exit ( ) : 자발적 프로세스 종료
  - 어떤 일이 일어나는가?
    - Returns status data from child to parent (via **`wait()`**)
    - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Abort ( ) : Parent가 child process 종료 시킴: 비자발적 프로세스 종료
  - 언제 일어나는가 ?
    - Child has exceeded allocated resources
    - Task assigned to child is no longer required
    - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call**.** The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- Child process has terminated, however, if no parent waiting (did not invoke **`wait()`**) process is a **zombie**
  - Child가 exit하면, parent proces는 이를 wait하다가 exit code는 받아야 함(reaping해야 함). 그런데, 이 reaping하는 parent가 없음. 그러면 exit code는 계속 process table에 남아있음. Process table 점유 & 소비…☹)
  - parent의 wait 호출되어야 process table 정리됨
- If parent terminated without invoking **`wait`** , process is an **orphan**
  - Child가 running하는데도 parent 는 죽었음 ☹
  - 이 경우, init process가 parent를 대신하며(child process를 init process가 입양함), …, 나중에 wait를 호출함

Multiprocess Architecture – Chrome Browser

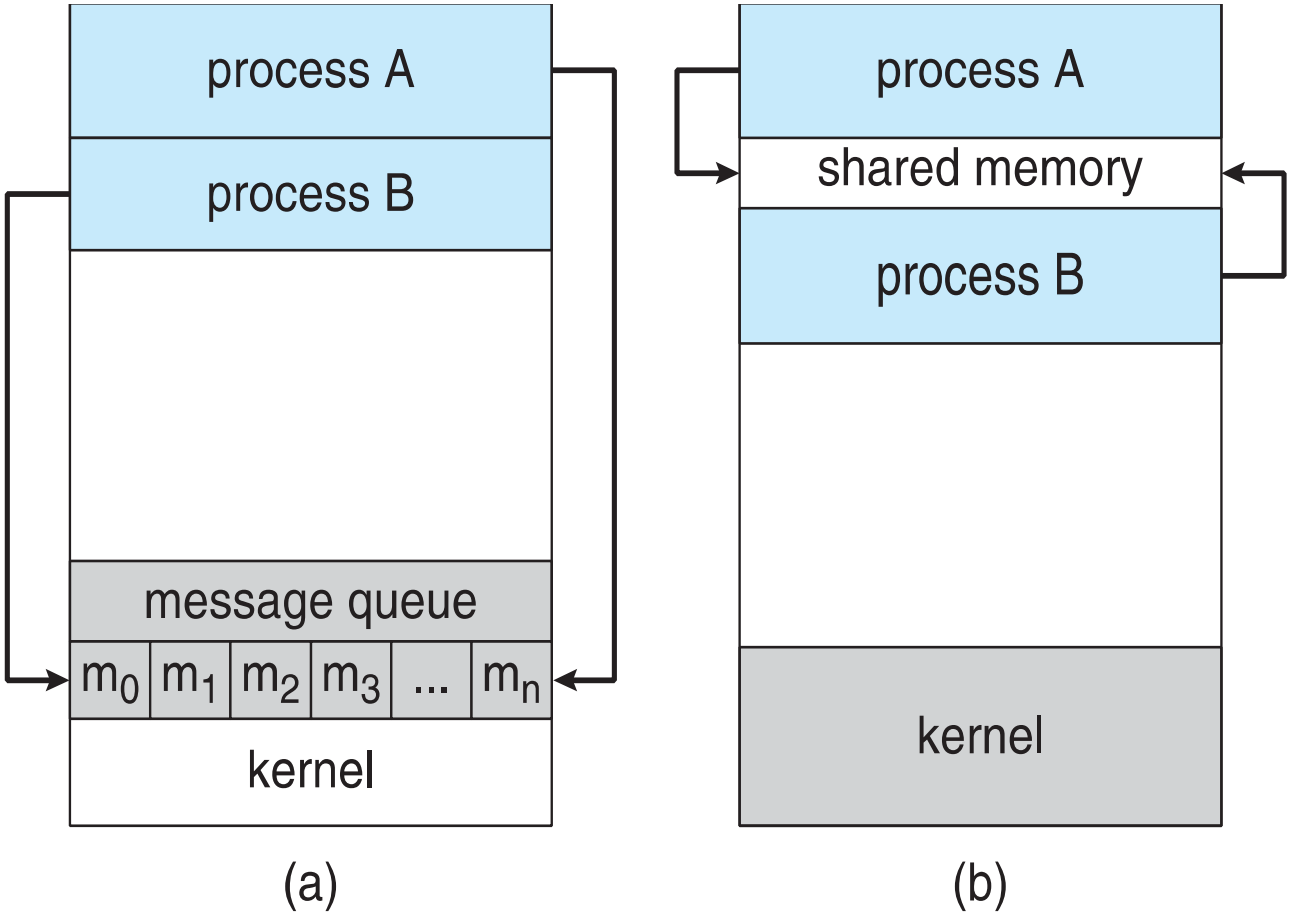
- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- **Google Chrome Browser is multiprocess with 3 different types of processes:**
  - **Browser process** manages user interface, disk and network I/O
  - **Renderer process** 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

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- **Reasons for cooperating processes:**
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- **Two models of IPC**
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing. (b) shared memory.



Cooperating Processes

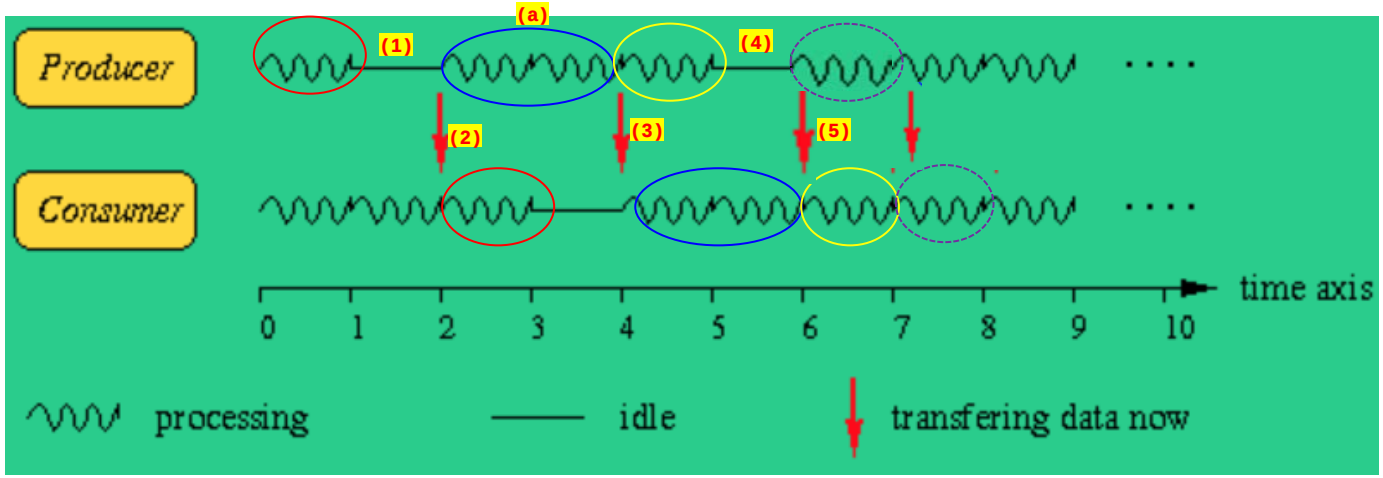
- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Producer-Consumer Problem

- Paradigm for cooperating processes, ***producer* process** produces information that is consumed by a ***consumer* process**
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

참고) Producer-Consumer Problem

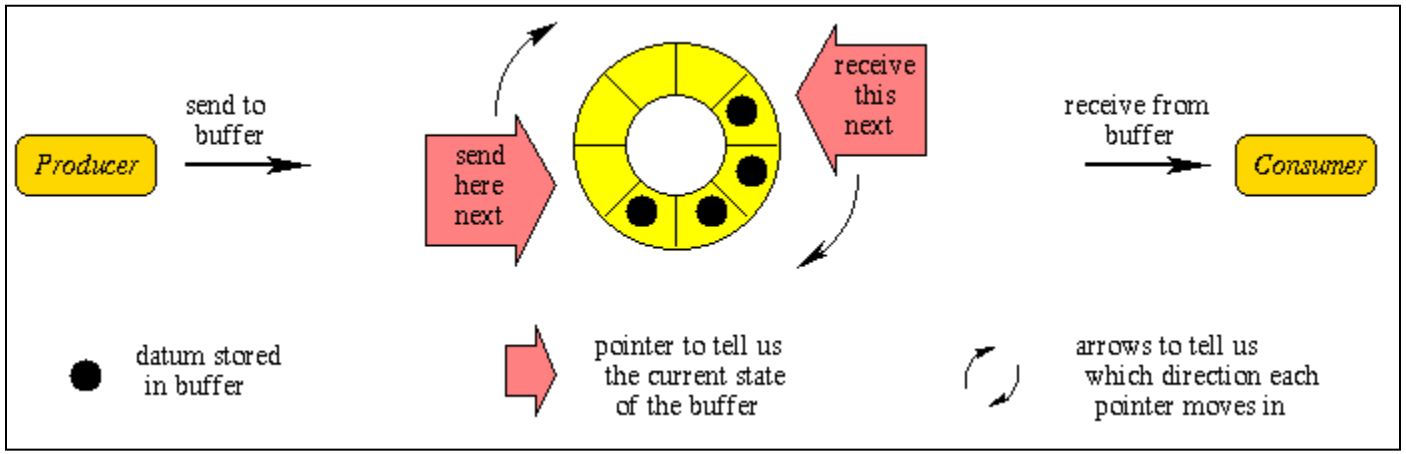
- The Producer-Consumer Problem
  - Producer는 자신의 작업을 한 후, 그 결과를 Consumer에게 보내는 Consumer가 준비가 되어있지 않으면 못보냄
  - Consumer는 Producer로부터 수신된 데이터를 처리하는데 Producer에서 보내지 않으면 처리하지 못함
  - 아래 그림처럼 동작한다고 가정하자
  - (1) Consumer가 이전 수신작업때문에 받지 못하는 상황이므로, Producer는 처리한 data를 보내지 못하고 wait함(idle 상태) → (2)에서 P는 C에게 작업을 보냄. P는 새로운 작업 (a) (2단위 작업이라고 가정)을 수행함. (3) (a)작업 결과를 C에게 보냄. 이전에 C는 할일이 없어서 idle 상태에 있었음. (4). → P가 C에게 작업을 보내고 P는 다른 작업 수행
  - 최대한 idle 상태없이 P와 C가 작업할 수 있도록 하는 문제 : Producer-Consumer Problem



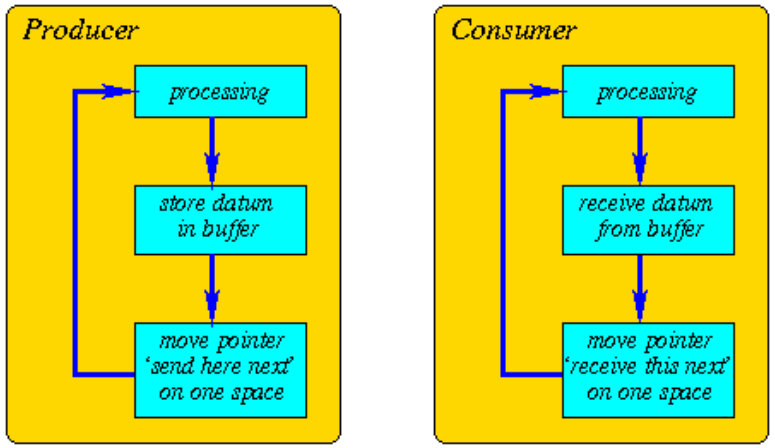
https://www.dcs.warwick.ac.uk/~sgm/open/pc.html

참고) Producer-Consumer Problem

- 1st idea for Producer-Consumer Problem
  - 아래 그림처럼, buffer를 도입하여, P와 C가 idle 상태에 있는 것을 줄이고자 함

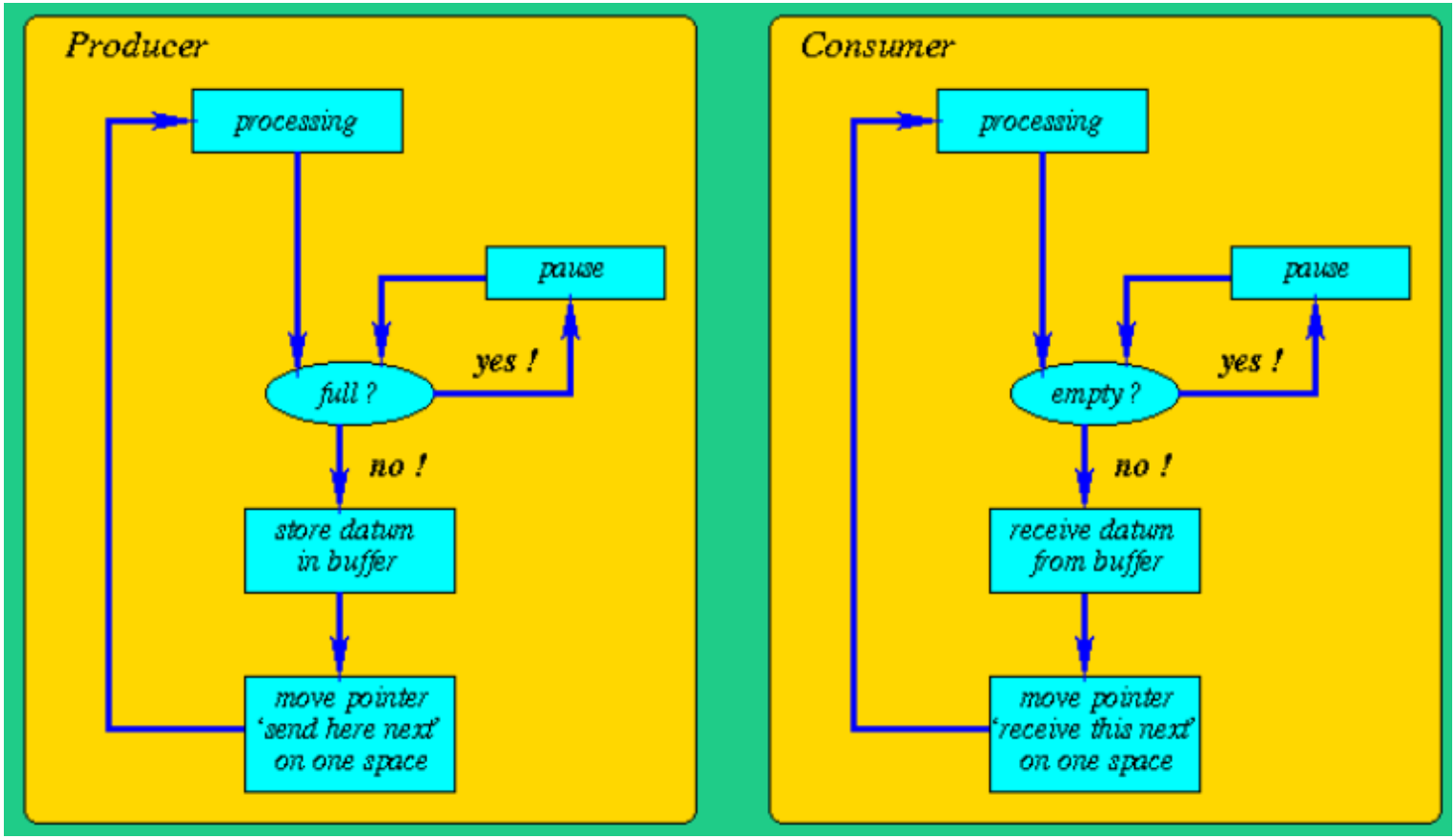


  - 동작 algorithm은 다음과 같음. 그런데 P와 C는 기본적으로 서로 독립적으로 동작하기때문에, 정상 동작이 안될 수 있음



참고) Producer-Consumer Problem

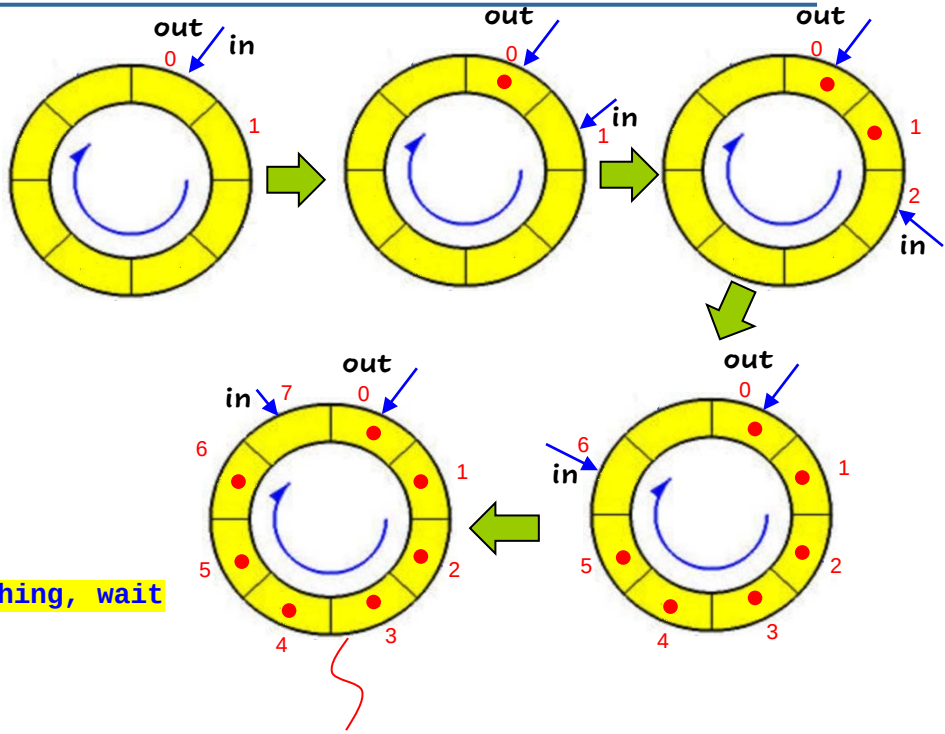
- 2nd idea for Producer-Consumer Problem
  - P와 C가 상호간 직접 통신이 없는 상황이므로, 아래처럼, P와 C가 각각 Buffer의 상태가 full 인지empty 인지 확인할 수 있도록 하면, 정상 동작 가능함



https://www.dcs.warwick.ac.uk/~sgm/open/pc.html

Bounded-Buffer – Shared-Memory Solution

```
#define BUFFER_SIZE 8
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0; // buffer내 다음 비어있는 위치 가르킴
int out = 0;// 버퍼내, 첫번째 채워져 있는 위치 가르킴
// in==out : buffer는 비어져 있음
// ((in+1)%BUFFER_SIZE) == out)이면 buffer는 full

item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out); //do nothing, wait

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
item next_consumed;
while (true) {
    while (in == out); /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_consumed */
}
```



Solution is correct, but can only use BUFFER_SIZE-1 elements

(위 그림처럼, buffer7이 비어져 있는 경우,
in=7임. (out=0 일 경우),

그런데, (in+1)%8도 0이 되는데,
즉, out와 같은 값이 되므로, buffer7에 값을 못쓰게 됨)

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - We are concerned here not with link's physical implementation, but rather with its logical implementation
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Direct Communication

- Processes must **name each other explicitly:**
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- **Messages are directed and received from mailboxes (also referred to as ports)**
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox

- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

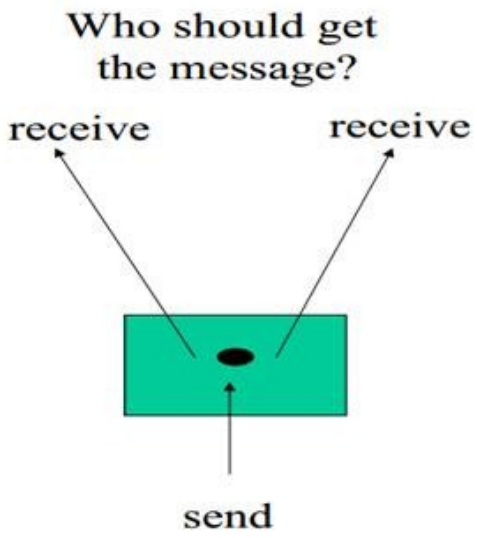
- **Operations**
  1. create a new mailbox (port)
  2. send and receive messages through mailbox
  3. destroy a mailbox

- **Primitives are defined as:**
  - **send(destination, message)**
  - **send**(***A***, *message*) – send a message **to mailbox A**

  - **receive(source, message)**
  - **receive**(***A***, *message*) – receive a message **from mailbox A**

Indirect Communication

- **Mailbox sharing**
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?



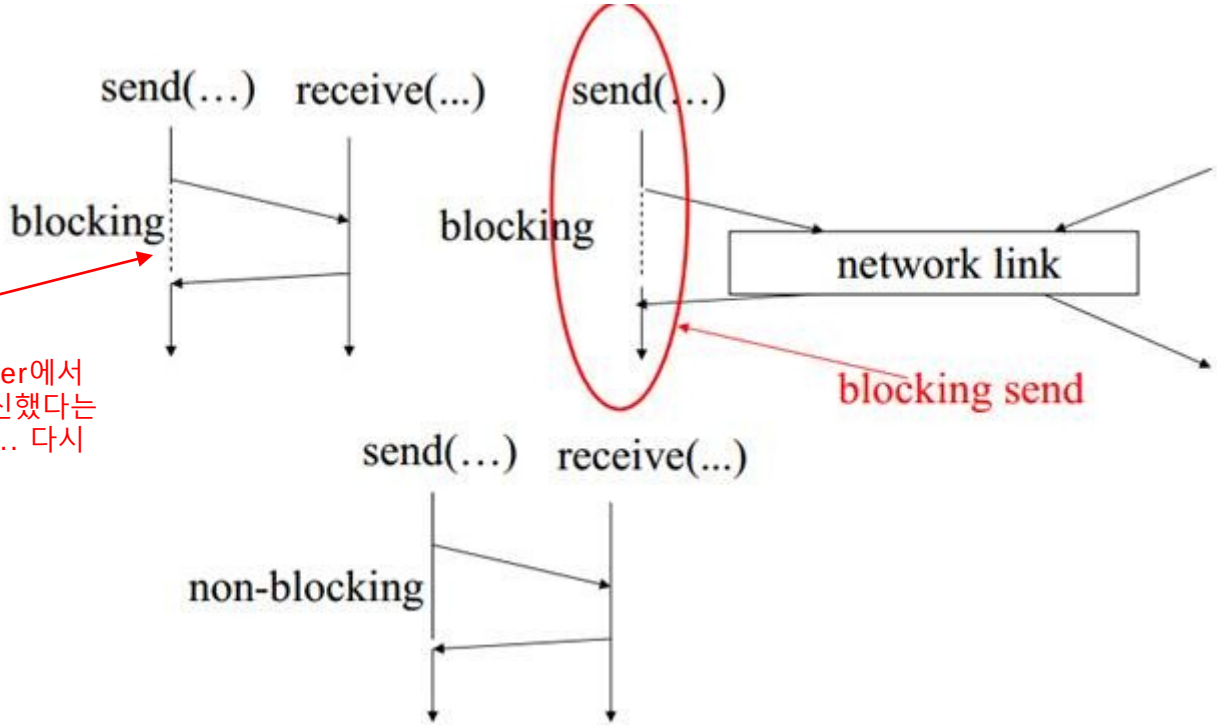
- **Solutions**
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver **(For example, round robin, where processes take turns receiving messages).** Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received(by a receive() operation) (receive() 함수에 의해 수신이 확인되기 전까지, 즉, receiver가 msg 수신하여 return 하기 전까지, sender는 새로운 메시지 send하지 못함)
  - **Blocking receive** – the receiver is blocked until a message is available (send쪽에서 메시지 올때까지 기다림)
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue. Messages are queued in an unbounded msg buffer, and the sender is never blocked.
  - **Non-blocking receive** -- the receiver is never blocked. A receive command returns an indication of whether or not a message was received.
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Synchronization

- **blocking send vs. non-blocking send**



- **Blocking send**
- sender가 "receiver에서 해당 메시지를 수신했다는 것을 확인한 이후".. 다시 보낼 수 있음

- **Blocking send** – sender blocked until message is received by receiver (or by mailbox)
- **Non-blocking send** – sending process resumes operation right after sending
- **Blocking receive** – receiver blocks until message is available
- **Non-blocking receive** – receiver retrieves a valid message or returns an error code
- Any combination of the above send/receive is possible

Synchronization (Cont.)

- Blocking send/receive를 사용할 경우, Producer-consumer becomes trivial
- 생산자는 blocking send( )를 호출 → 수신자가 msg를 수신할 때까지 기다리기만 하면 됨
- 소비자도 blocking receive()를 호출 → 송신자가 송신 메시지 준비할 때까지 기다림

```
message next_produced;
        while (true) {
            /* produce an item in next produced */
        send(next_produced);
//  Blocking send인 경우 : 이전 send msg 수신때까지 새로운 send는 blocking됨
}

message next_consumed;
while (true) {
   receive(next_consumed);
//  Blocking receive인 경우, 즉 receive할 msg 있을때까지 기다림.

   /* consume the item in next consumed */
}
```

- 이 경우, send와 receive에 대한 동기화는 쉬움

Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
  1. **Zero capacity** – no messages are queued on a link.
     Sender must wait for receiver (rendezvous)
  2. **Bounded capacity** – finite length of *n* messages
     Sender must wait if link full
  3. **Unbounded capacity** – infinite length
     Sender never waits

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - **Process first creates shared memory segment**

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

(공유객체이름, 객체생성|읽기/쓰기 가능한 상태로 열림, 접근권한)

  - Also used to open an existing segment to share it
  - **Set the size of the object**

```
ftruncate(shm_fd, 4096);
```

// file descriptor를 사용해서 파일크기를 특정 크기로 조정함

  - **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;
}
```

Examples of IPC Systems - Mach

- **Mach communication is message based**
  - Even system calls are made by messages
  - When a task is created, **two special mailboxes are created - Kernel and Notify**
  - Only three system calls needed for message transfer

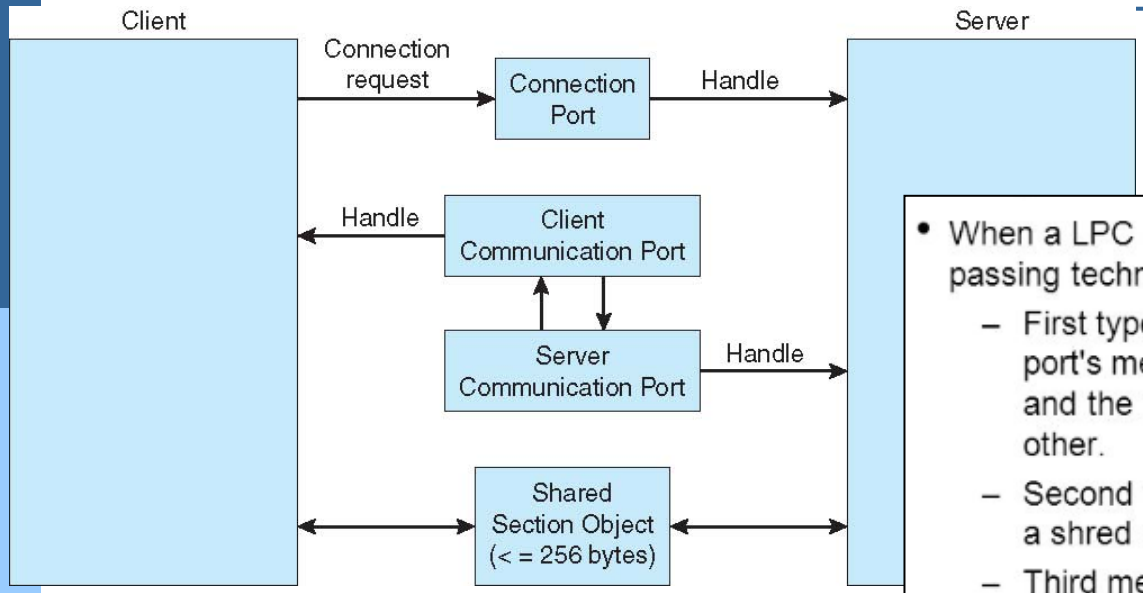
    `msg_send(), msg_receive(), msg_rpc()`

  - **port_allocate()** system call creates a new mailbox and allocates space for its queue of messages.
  - Send and receive are flexible**, for example four options if mailbox full:**
    - Wait indefinitely
    - Wait at most *n* milliseconds
    - Return immediately
    - Temporarily cache a message

Examples of IPC Systems – Windows

- **Message-passing centric via advanced local procedure call (LPC) facility**
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Connection port와 Communication port 제공
  - Communication works as follows:
    - Server가 Connection port 만들어서 공표함 → The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Local Procedure Calls in Windows

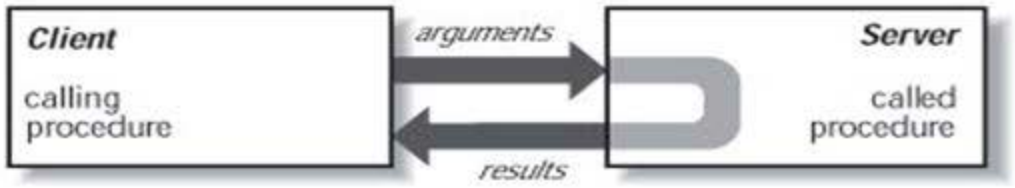


- When a LPC channel is created, one of three types of message passing techniques must be specified.
  - First type is suitable for small messages, up to 256 bytes; port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
  - Second type avoids copying large messages by pointing to a shred memory section object created for the channel.
  - Third method, call *quick* LPC is used by graphical display portions of the Win32 subsystem.

Windows XP message passing is called Local Procedure-Call (LPC)

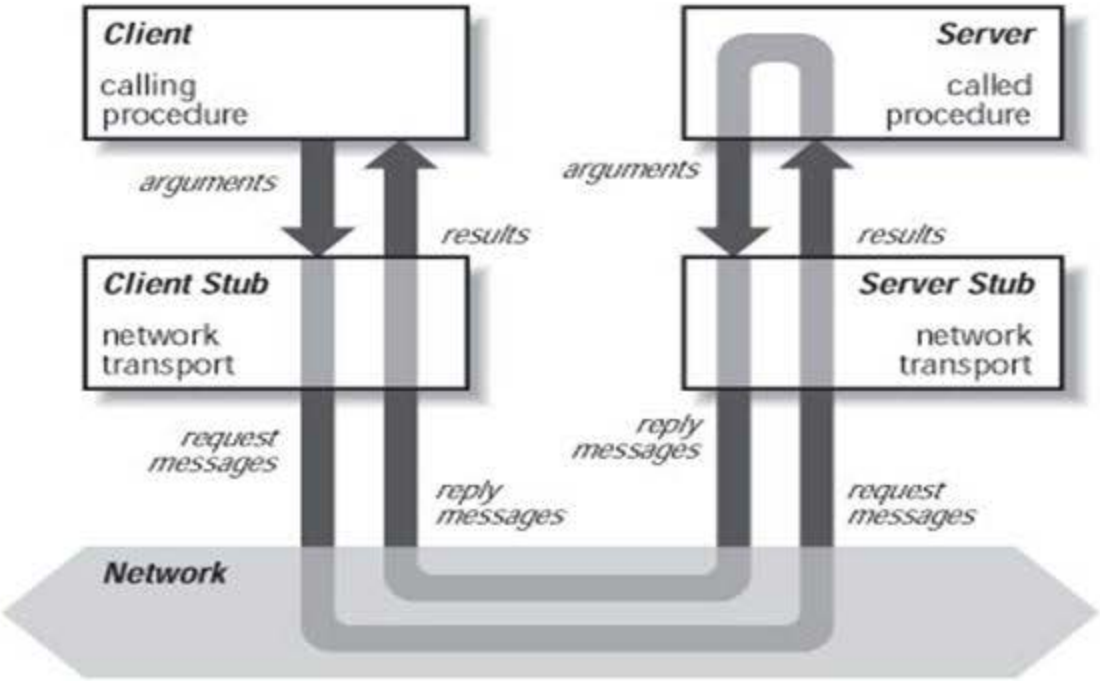
- for small messages,
  1. client sends connection request to server
  2. server sets up two private communication ports and notifies client
  3. messages are sent back and forth via the private ports
- for large messages,
  1. a shared section object is set up (shared memory)
  2. client & server can communicate by writing/reading shared memory

Local vs. Remote Procedure Calls



Local Procedure Call

In a local procedure call, a calling process executes a procedure in its own address space.



Remote Procedure Call

In a remote procedure call, the client and server run as two separate processes. It is not necessary for them to run on the same machine.

The two processes communicate through stubs, one each for the client and server. These stubs are pieces of code that contain functions to map local procedure calls into a series of network RPC function calls.

Communications in Client-Server Systems

- **Sockets**
- **Remote Procedure Calls**
- **Pipes**
- Remote Method Invocation (Java)

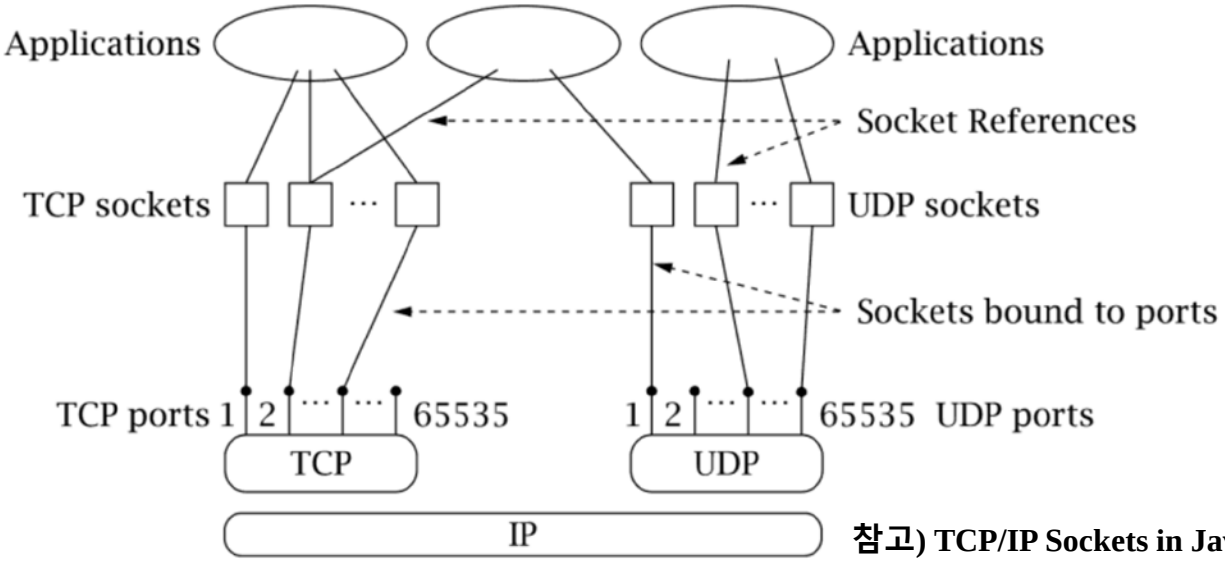
Sockets

- A **socket** is defined as an endpoint of two-way communication link b/w two programs running on the network
- Concatenation of **IP address and port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address **127.0.0.1** (**loopback**) to refer to system on which process is running

More on Sockets

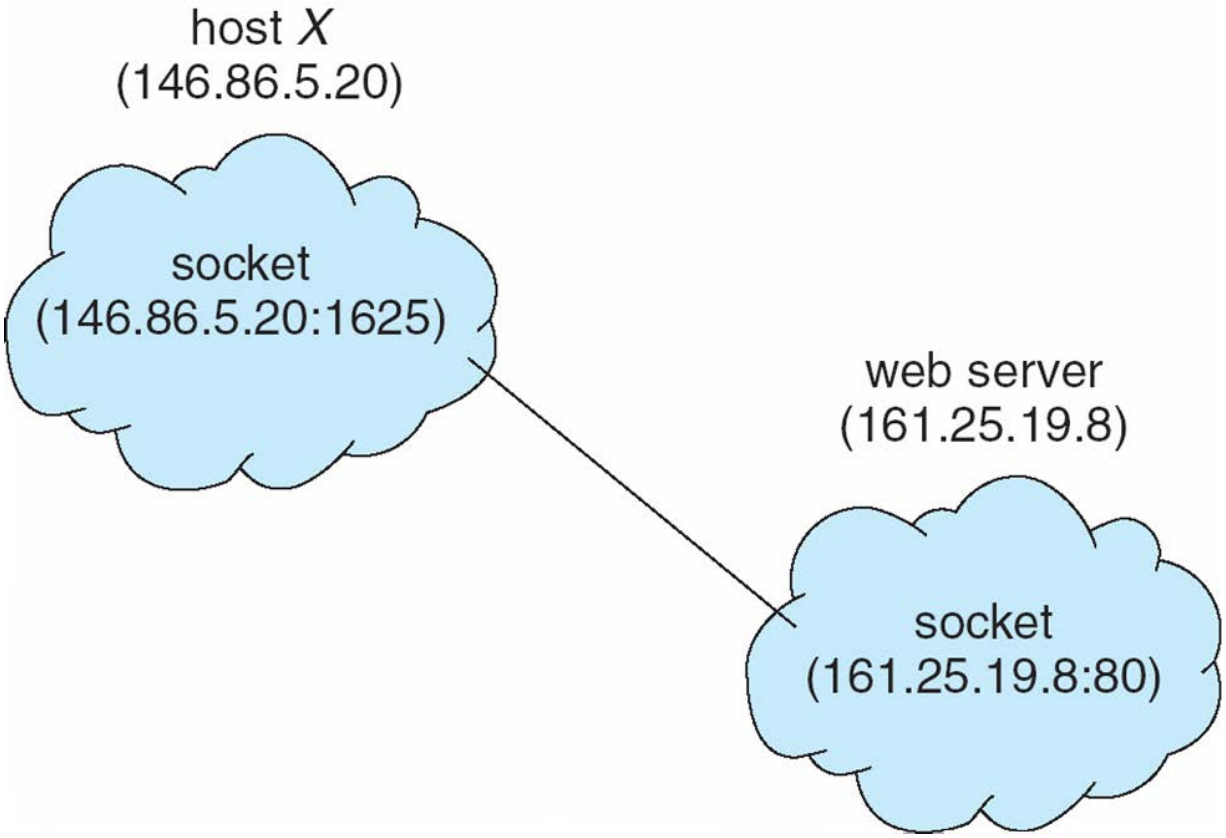
A *socket* is an abstraction through which an application may send and receive data, in much the same way as an open file handle allows an application to read and write data to stable storage. A socket allows an application to plug in to the network and communicate with other applications that are plugged in to the same network. Information written to the socket by an application on one machine can be read by an application on a different machine and vice versa.

Different types of sockets correspond to different underlying protocol suites and different stacks of protocols within a suite. This book deals only with the TCP/IP protocol suite. The main types of sockets in TCP/IP today are *stream sockets* and *datagram sockets*. Stream sockets use TCP as the end-to-end protocol (with IP underneath) and thus provide a reliable byte-stream service. A TCP/IP stream socket represents one end of a TCP connection. Datagram sockets use UDP (again, with IP underneath) and thus provide a best-effort datagram service that applications can use to send individual messages up to about 65,500 bytes in length.



참고) TCP/IP Sockets in Java: Practical Guide for Programmers

Socket Communication



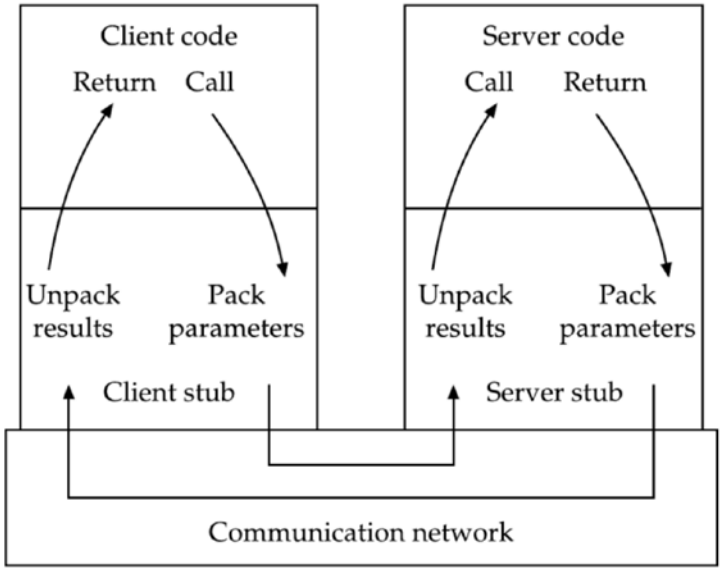
Sockets in Java

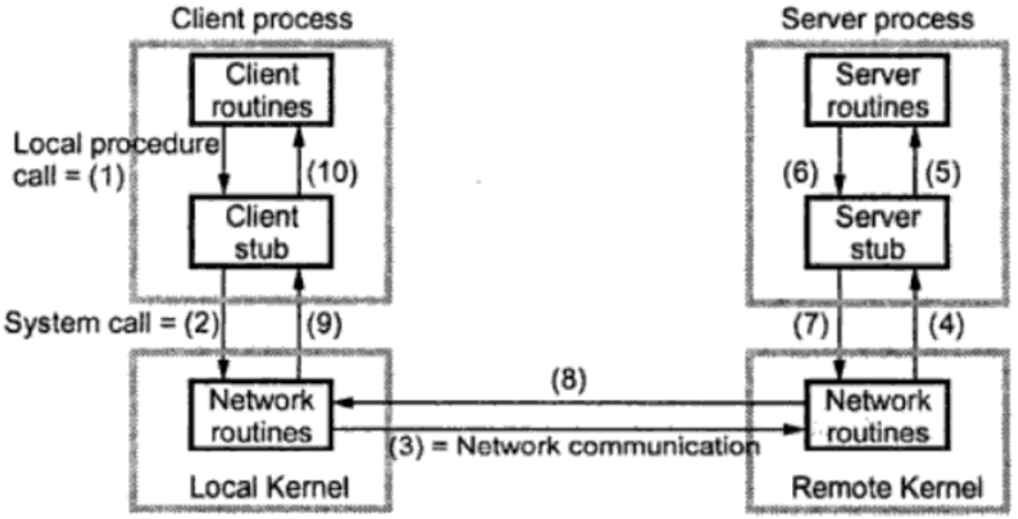
- **Three types of sockets**
  - **Connection-oriented** (**TCP**)
  - **Connectionless** (**UDP**)
  - **`MulticastSocket`** class– data can be sent to multiple recipients
- Consider this “Date” server:

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Remote Procedure Calls

- **Remote procedure call (RPC)** abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – **client-side proxy** for the actual procedure on the server
- **The client-side stub locates the server** and **marshals** the parameters
- **The server-side stub** receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)



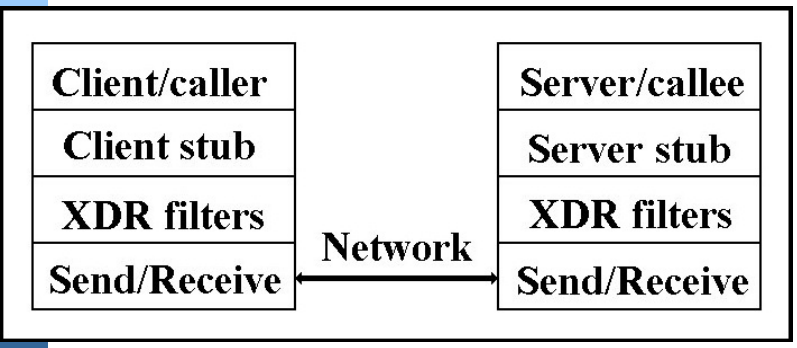


1. The client calls a local procedure, called the client stub. It appears to the client that the client stub is the actual server procedure that it wants to call. The purpose of the stub is to package the arguments for the remote procedure, possibly put them into some standard format and then build one or more network messages. The packaging of the client's arguments into a network message is termed marshaling.
2. These network messages are sent to the remote system by the client stub. This requires a system call to the local Kernel.
3. The network messages are transferred to the remote system. Either a connection-oriented or a connection-less protocol is used.
4. A server stub procedure is waiting on the remote system for the client's request. It unmarshals the arguments from the network message and possibly converts them.
5. The server stub executes a local procedure call to invoke the actual server function, passing it the arguments that it received in the network messages from the client stub.
6. When the server procedure is finished, it returns to the server stub with return values.
7. The server stub converts the return values, if necessary, and marshals them into one or more network messages to send back to the client stub.
8. The messages get transferred back across the network to the client stub
9. The client stub reads the network messages from the local Kernel.
10. After possibly converting the return values, the client stub finally returns to the client function. This appears to be a normal procedure return to the client.

Networking Technology for Digital Devices by V.S.Bagad, I.A.Dhotre

Remote Procedure Calls (Cont.)

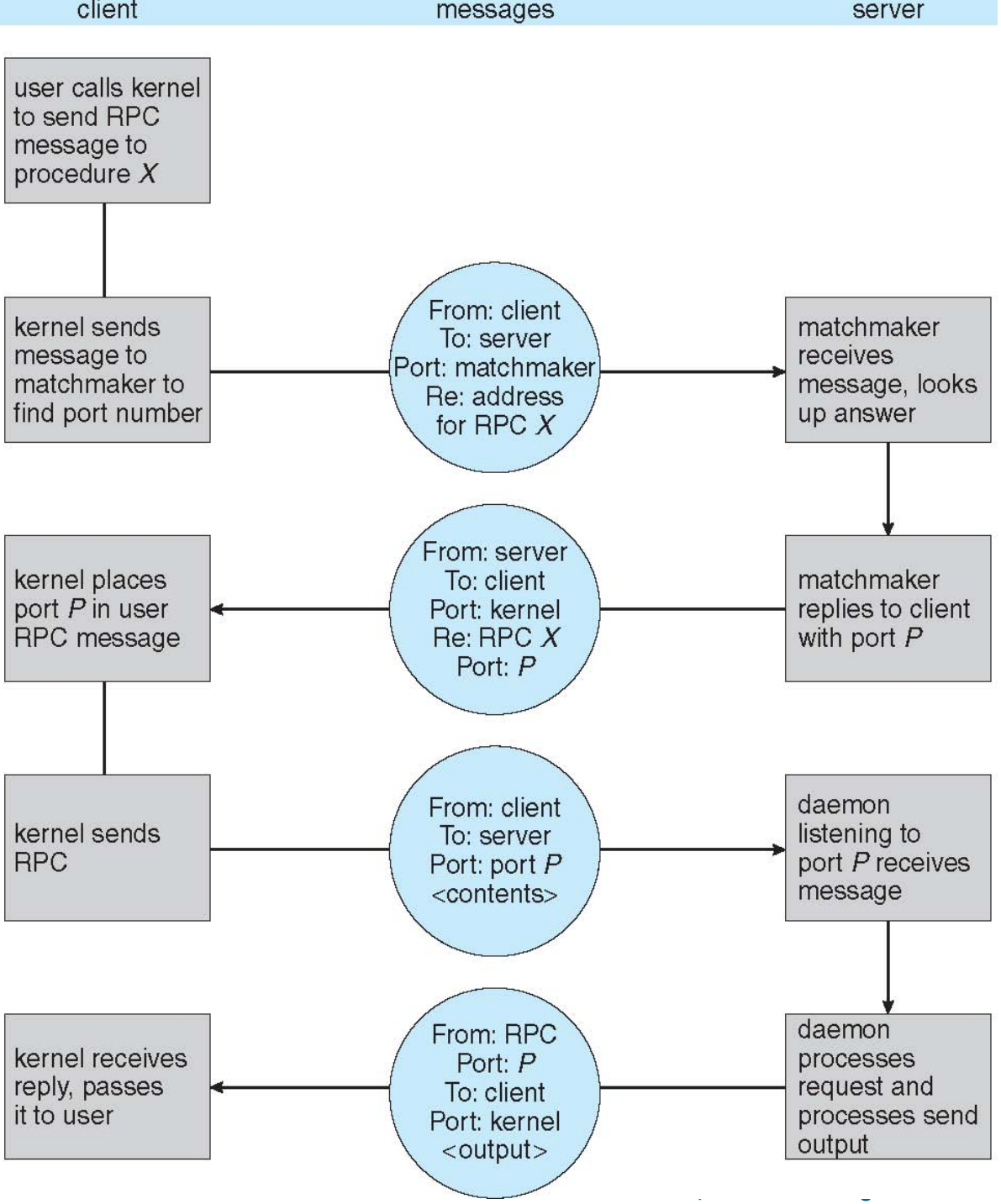
- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides **a rendezvous (or matchmaker) service** to connect client and server



- XDR(External Data Representation)은 RPC 호출과 응답 메시지에서 값을 코드화 하기 위해 사용된 표준
- RFC 1014에 정의
- 여러 가지 데이터 타입 정의, RPC 메시지에 전송되는 방법 정의
- 송신자: XDR형식에 RPC 메시지를 설립
- 수신기: 원래 표현으로 XDR 형식 변환
- XDR에 정의된 다른 데이터 유형
  - 부호없는 정수, 부울값, 부동 소수점, 고정길이 배열, 가변길이 배열과 구조형 포함

Execution of RPC

- RPC(Remote procedure call)
  - » 네트워크 상의 프로세스들 사이에서 프로시저 호출(procedure calls)을 추상화
- 클라이언트 측 스터브(stub)
  - » 원격 서버와 포트를 찾고
  - » 파라미터를 정돈(marshal)하여
  - » 메시지 전달
- 서버 측 스터브(stub)
  - » 원격 서버의 프로시저에 대한 대행자(proxy)로서
  - » 메시지 받아
  - » 정돈된 파라미터를 풀어(unmarshal)
  - » 프로시저를 수행함

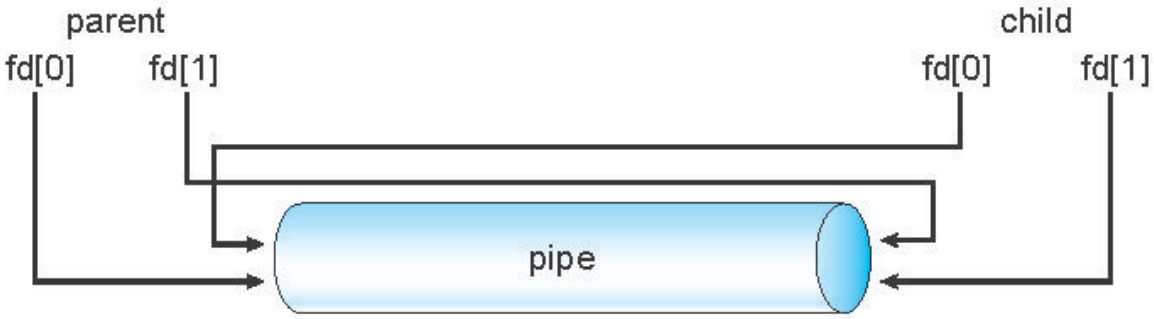


Pipes

- Acts as a conduit allowing two processes to communicate
- **In implementing a pipe, four issues must be considered:**
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes** – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created**. (즉, 부모-자식 프로세스 관계에서만 pipe 통신 가능)**
- **Named pipes** – can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary Pipes allow 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)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**
- See Unix and Windows code samples in textbook

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

End of Chapter 3