고려대학교
KOREA UNIVERSITY

KU-The Future

Operating Systems (10th Ed., by A. Silberschatz)

Chapter 3 Processes

Heonchang Yu

Distributed and Cloud Computing Lab.

Contents

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

Objectives

- To Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that use pipes and POSIX shared memory to perform interprocess communication.
- Describe client–server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

Process Concept

- An operating system executes a variety of programs:
  - ✓ Batch system – **jobs**
  - ✓ Time-shared systems – **user programs** or **tasks**
- Although we prefer the more contemporary term ***process***, the term ***job*** has historical significance.
- **Process** – a program in execution; The status of the current activity of a process is represented by the value of the **program counter** and the contents of the processor's registers.
- Multiple sections
  - ✓ **text section** : the executable code
  - ✓ **Data section** : global variables
  - ✓ **Heap section** : memory that is dynamically allocated during program run time
  - ✓ **Stack section** : temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

Process Concept

- Program is a ***passive*** entity stored on disk (**executable file**), process is an ***active*** entity.
  - ✓ Program becomes a process when an executable file is loaded into memory.
- Two common techniques for loading executable files
  - ✓ Double-clicking an icon representing the executable file
  - ✓ Entering the name of the executable file on the command line
- Several users may be running different copies of the mail program, or the same user may invoke many copies of the Web browser program. Each of these is a separate process.
  - ✓ Although the text sections are equivalent, the data, heap, and stack sections vary.
  - ✓ It is also common to have a process that spawns many processes as it runs.

Process Concept : Process in Memory

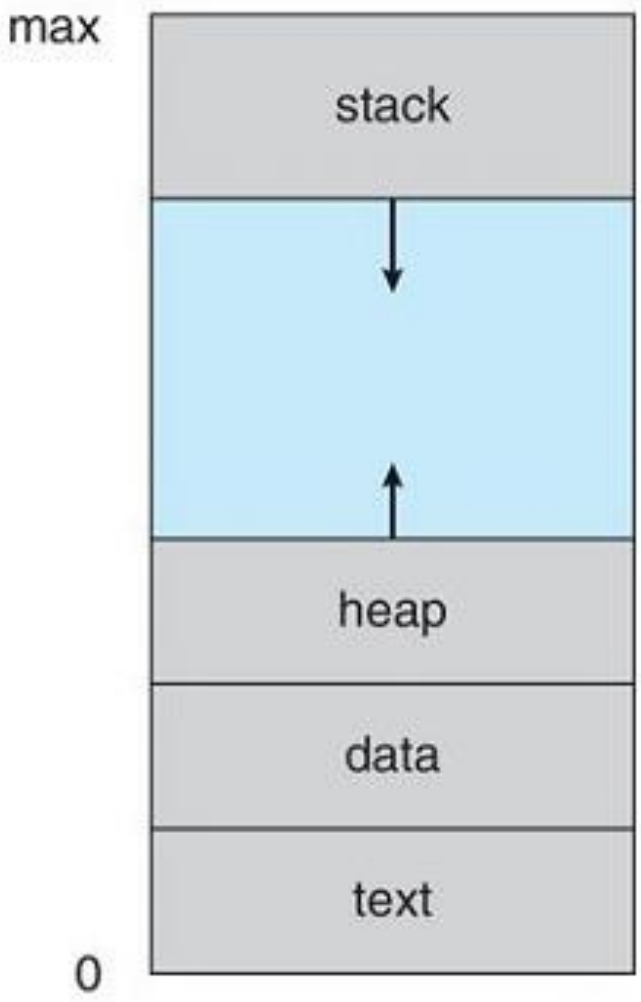
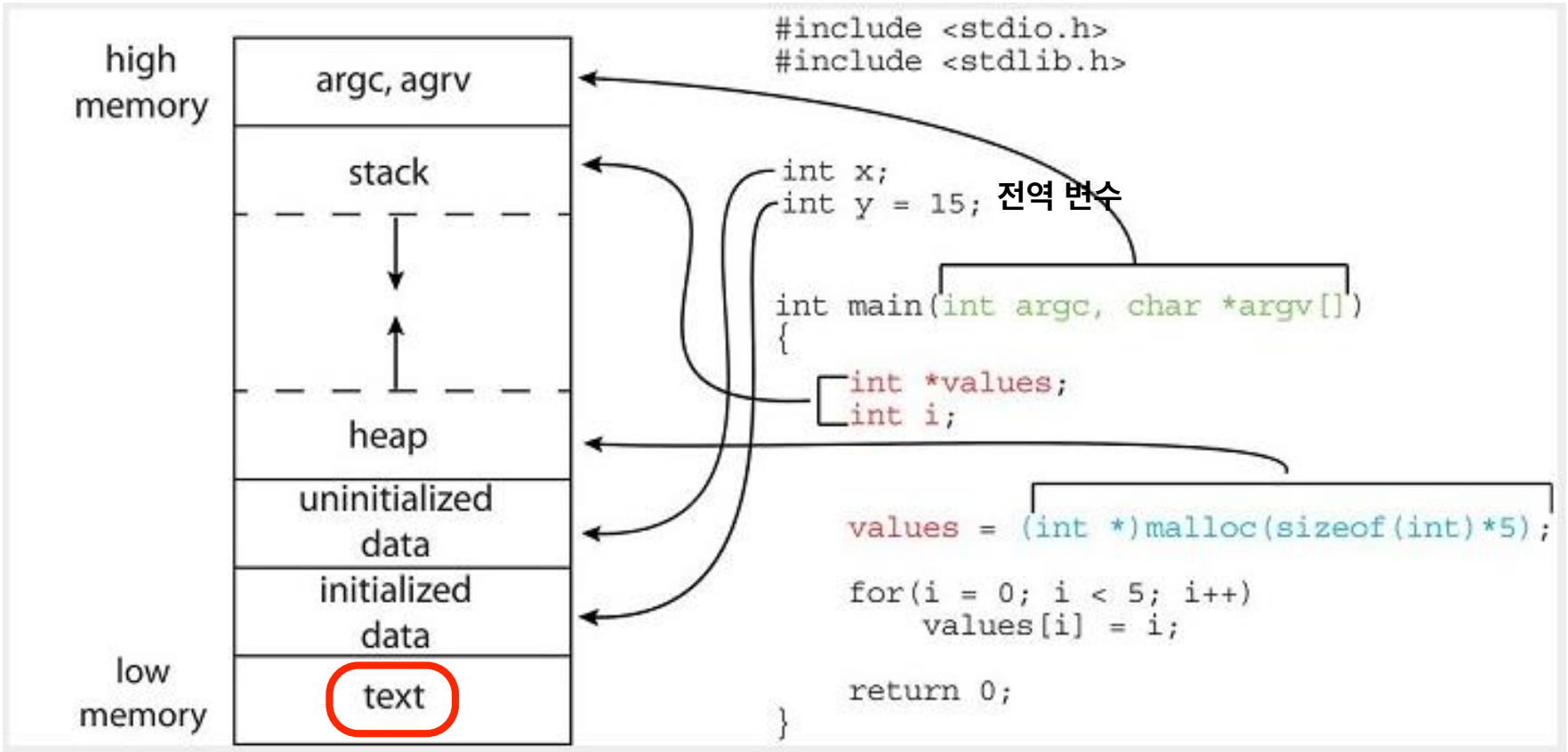


Figure 3.1 Layout of a process in memory.

Process State : Memory layout of a C program



Process State : Memory layout of a C program

- This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:
  - ✓ The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
  - ✓ A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.
- GNU command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is `memory`, the following is the output generated by entering the command `size memory`:

```
text    data    bss    dec    hex    filename
1158    284     8      1450   5aa    memory
```

- The `data` field refers to uninitialized data, and `bss` refers to initialized data. (`bss` is a historical term referring to **block started by symbol**.) The `dec` and `hex` values are the sum of the three sections represented in decimal and hexadecimal

Process State 여기가 제일 중요한 파트다

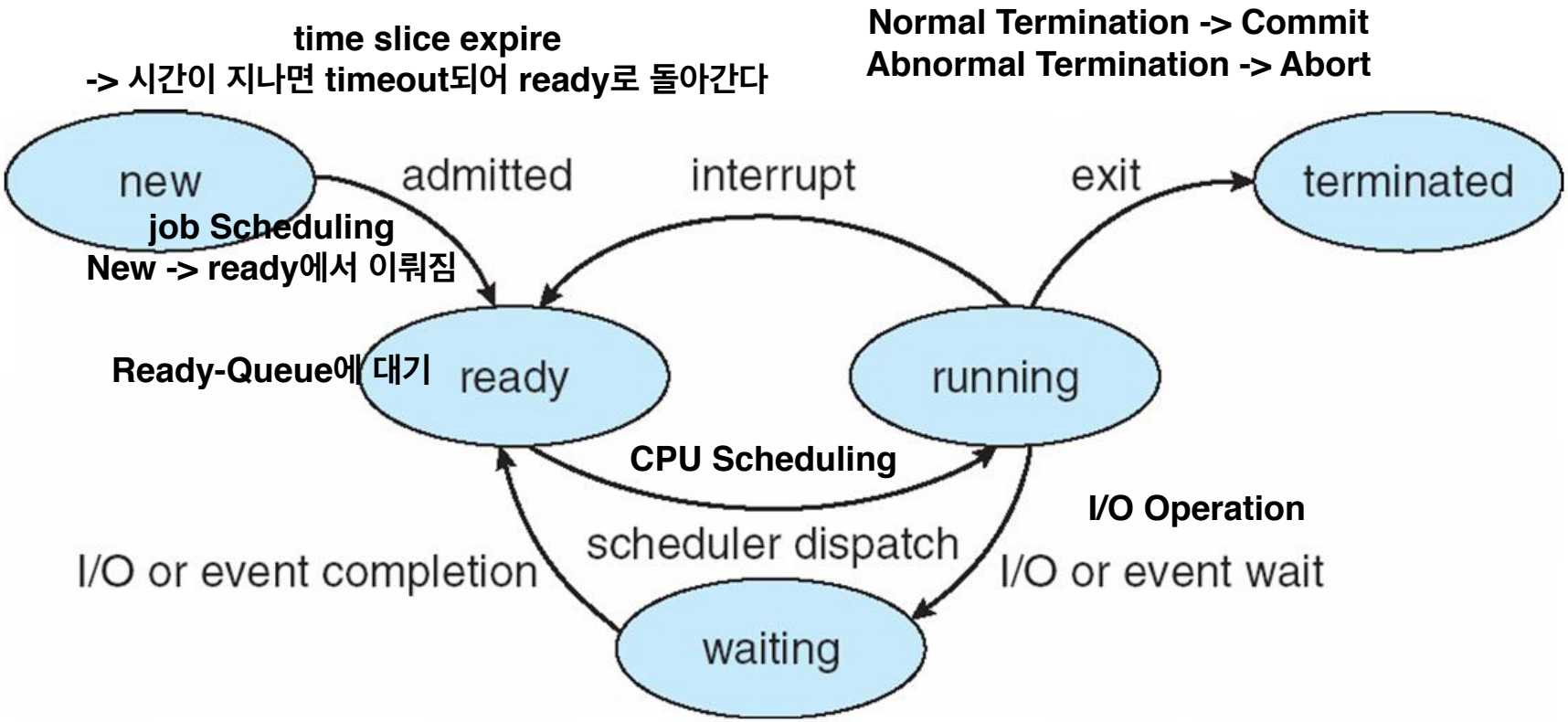
- As a process executes, it changes *state*
  - ✓ **New**: The process is being created
  - ✓ **Running**: Instructions are being executed
  - ✓ **Waiting**: The process is waiting for some event to occur
  - ✓ **Ready**: The process is waiting to be assigned to a processor
  - ✓ **Terminated**: The process has finished execution

생성에서 삭제까지 프로세스가 겪는 절차

하드웨어에서 프로그램이 실행되어 메모리로 넘어와 프로세스가 생성되었을 때 겪는 과정이다.

Diagram of Process State

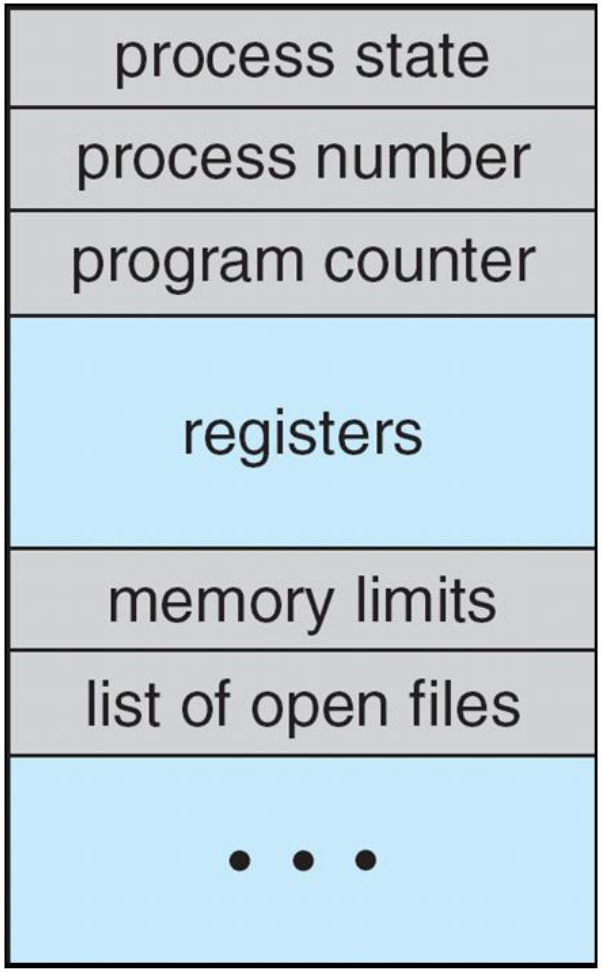
여기가 제일 중요한 파트다



Process Control Block (PCB)

- It contains many pieces of information associated with each process
  - also called **task control block**
  - ✓ Process state – new, ready, running, waiting, halted, etc
  - ✓ Program counter – the address of the next instruction to be executed
현재 수행하려고 하는 프로그램의 주소 값
  - ✓ CPU registers – include accumulators, index registers, stack pointers, and general-purpose registers
  - ✓ CPU scheduling information- includes a process priority, pointers to scheduling queues, and any other scheduling parameters
  - ✓ Memory-management information – includes the value of the base and limit registers and the page tables, or the segment tables
  - ✓ Accounting information – includes the amount of CPU and real time used, time limits, account numbers, job or process numbers
  - ✓ I/O status information – includes the list of I/O devices allocated to the process, a list of open files

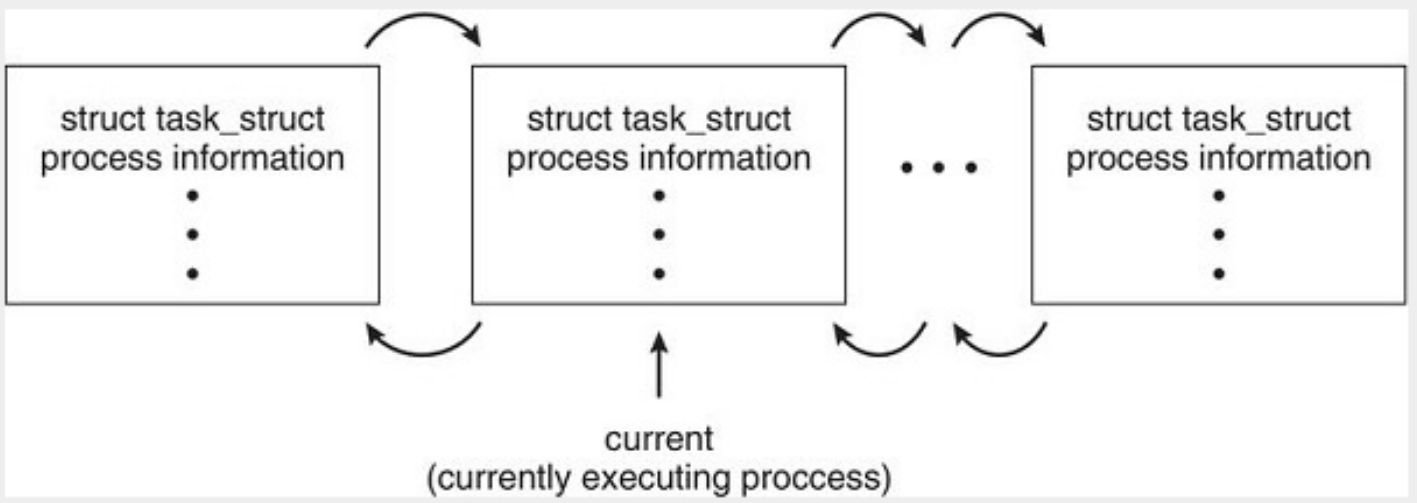
Process Control Block (PCB)



Process Representation in Linux

- Process control block in the Linux is Represented by the C structure `task_struct`, which is found in the `<include/linux/sched.h>`.

```
long state;                  /* state of the process */
struct schd_entity se;       /* 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

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization
- The objective of time sharing is to switch a CPU core among processes so frequently that users can interact with each program while it is running.
- Process scheduling queues
  - ✓ 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

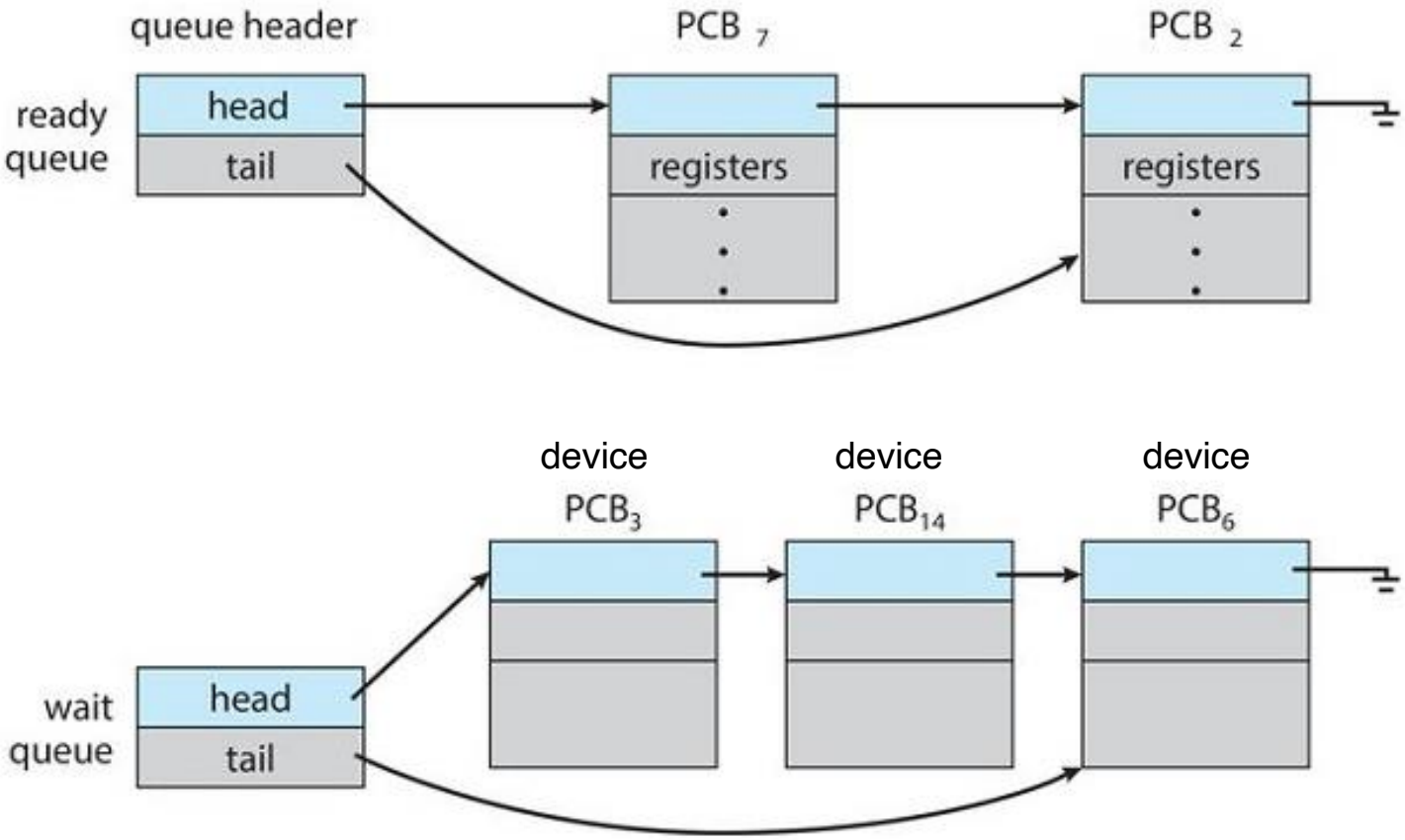


Figure 3.4 The ready queue and wait queues.

Queueing Diagram

- A common representation of process scheduling
  - ✓ Rectangular box – queues (ready queue and a set of wait queues)
  - ✓ Circles - resources

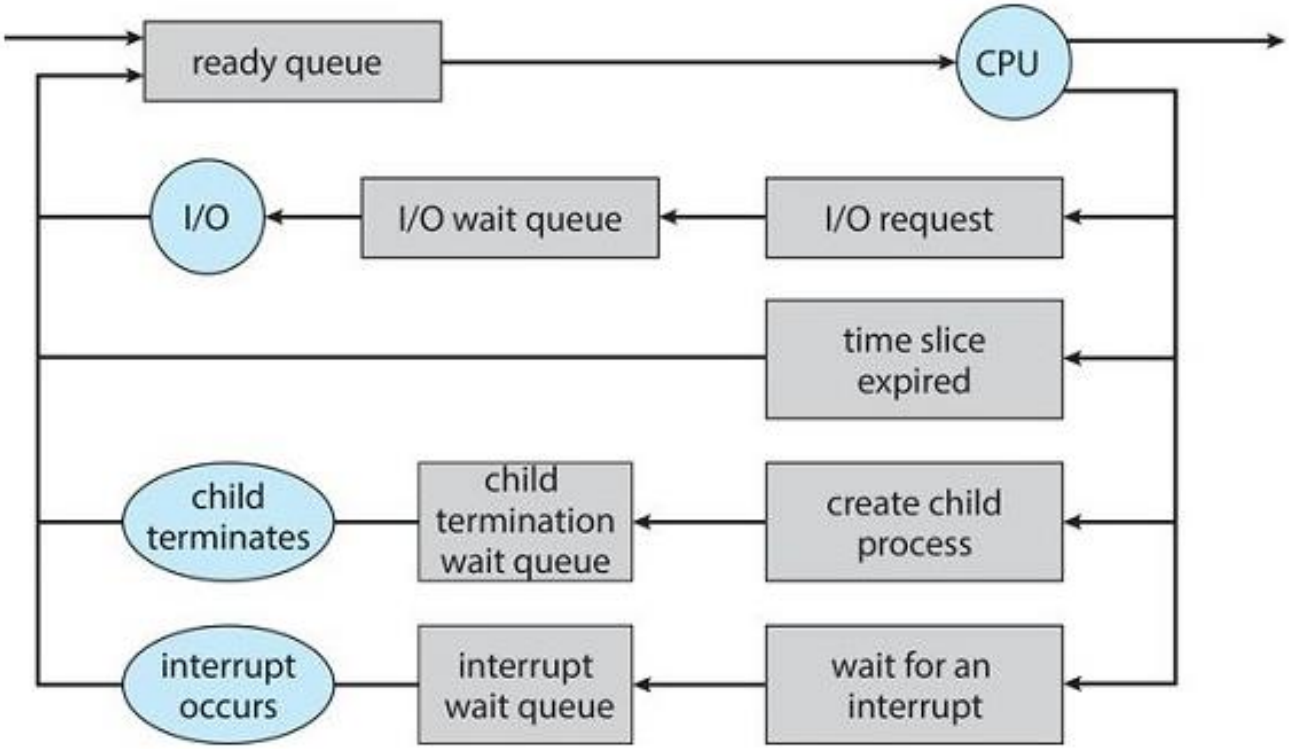
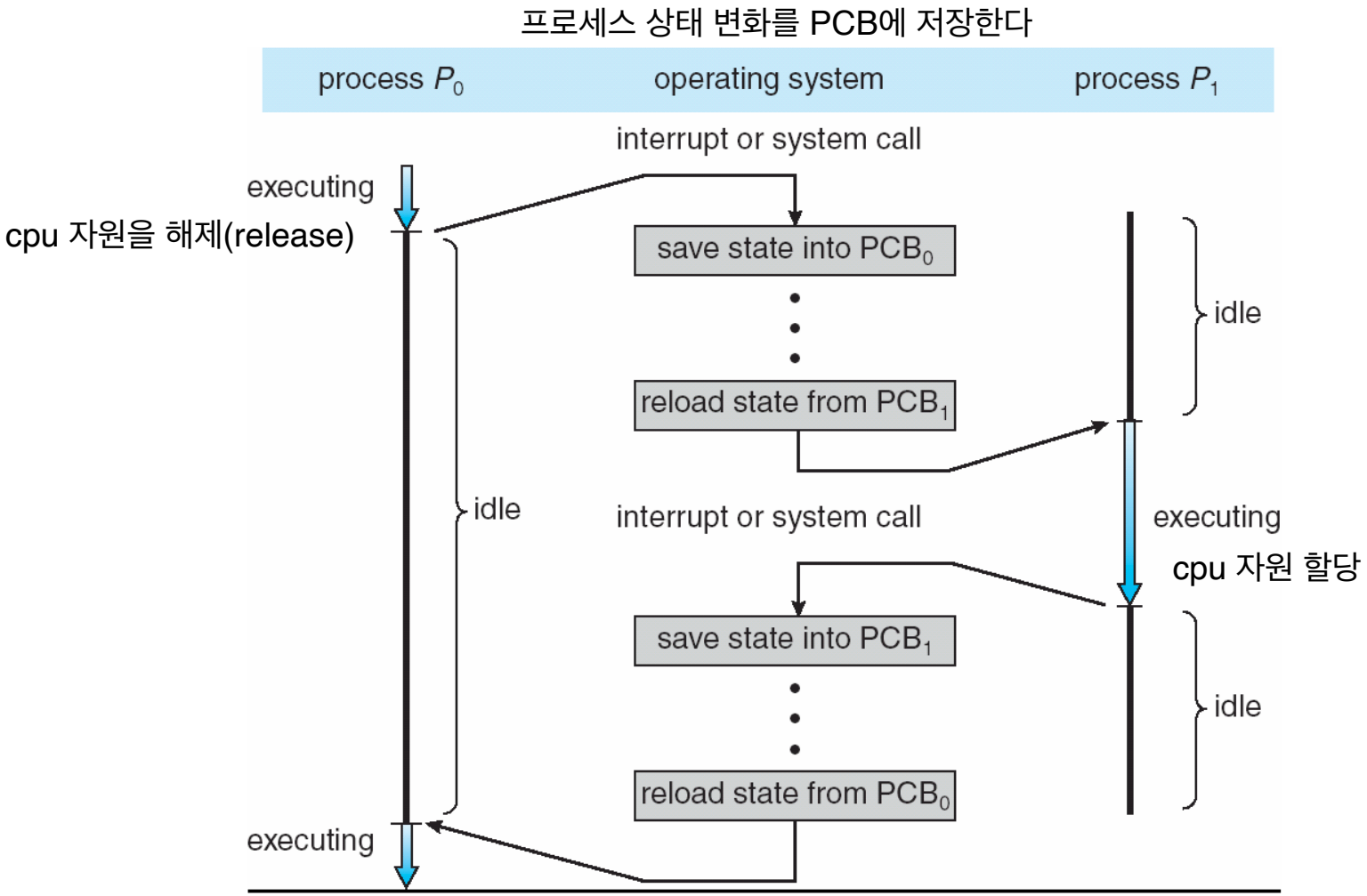


Figure 3.5 Queueing-diagram representation of process scheduling.

Context Switch From Process to Process



Schedulers

- **Long-term scheduler** (or job scheduler) – selects processes from this pool and loads them into memory for execution
- **Short-term scheduler** (or CPU scheduler) – selects from among the processes that are ready to execute and allocates the CPU to one fo them
- The primary distinction between these two schedulers lies in frequency of execution.
  - ✓ Short-term scheduler must select a new process for the CPU frequently (milliseconds) ⇒ (must be fast)
  - ✓ Long-term scheduler executes much less frequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory)

메모리 내에 상주하는 프로세스의 갯수

cpu scheduling은 엄청나게 많이 발생함

Schedulers (Cont.)

지금 얘기하고 있는 것은 코어가 하나일 때를 기준으로 얘기하는 것임

- Processes can be described as either:
  - ✓ **I/O-bound process** – spends more of its time doing I/O than it spends doing computations, many short CPU bursts
  - ✓ **CPU-bound process** – spends more time doing computations; few very long CPU bursts

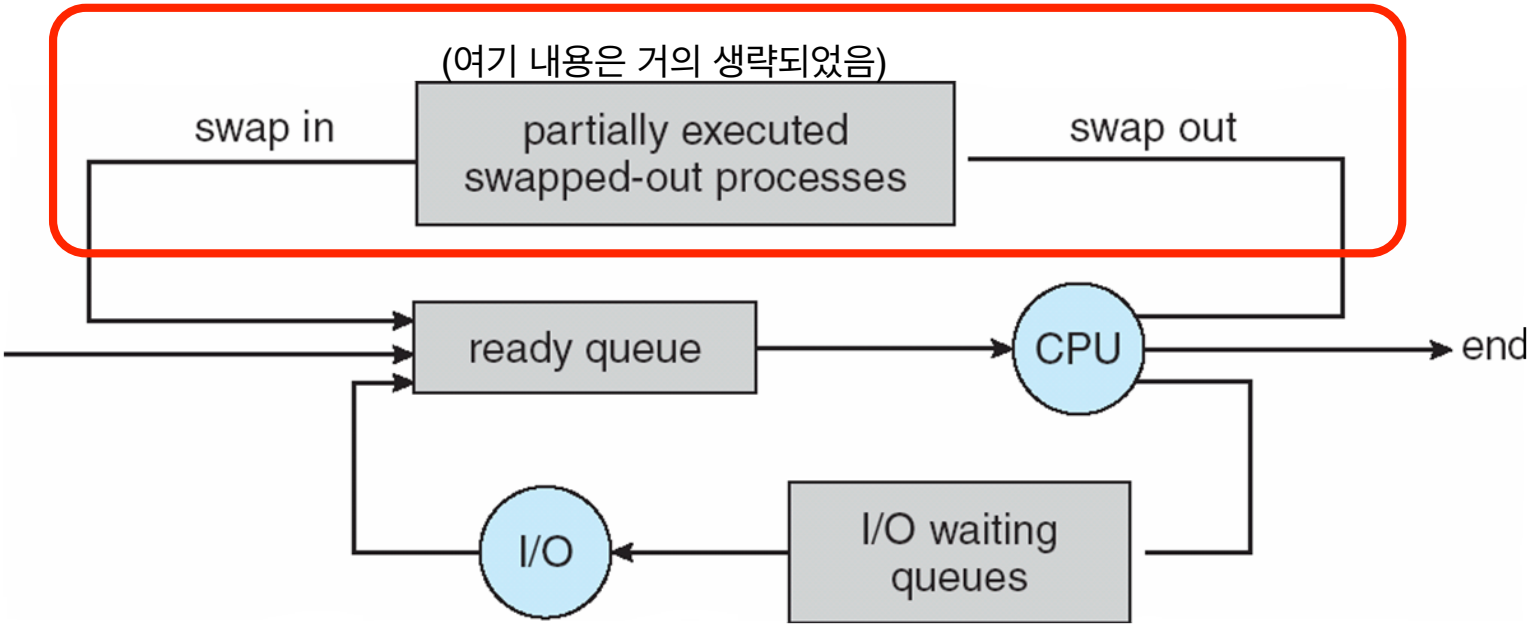
intensive

burst는 시간이라 보면 된다

- The long-term scheduler selects a good process mix of I/O-bound and CPU-bound processes
  - ✓ If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do.
  - ✓ If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced.

Medium-Term Scheduling

- Intermediate form of scheduling
  - ✓ It can be advantageous to remove processes from memory and thus reduce the degree of multiprogramming
  - ✓ Swapping – The process can be reintroduced into memory, and its execution can be continued where it left off. It is only necessary when memory has been overcommitted and must be freed up.



Context Switch

- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.
  - ✓ When a context-switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run
    old process ->지금 현재 수행중인 프로세스
    new process -> 이제 수행 준비가 된 프로세스
- Context-switch time is pure overhead; the system does no useful work while switching
- Context-switch time are highly dependent on hardware support
  - ✓ Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers).

Operations on Processes: Process Creation

- During the course of execution, a process may create several new processes.
  - ✓ The creating process is called a parent process, and the new processes are called the children of that process.
- Most operating systems identify processes according to a unique **process identifier** (**pid**).
- When a process creates a child process
  - ✓ Child process may be able to obtain its resources directly from OS
  - ✓ Child process may be constrained to a subset of the resources of the parent process
- When a process creates a new process
  - ✓ The parent continues to execute concurrently with its children.
  - ✓ The parent waits until some or all of its children have terminated.

A Tree of Processes on Linux

- `systemd` process (which always has a pid of 1) serves as the root parent process for all user processes, and is the first user process created when the system boots

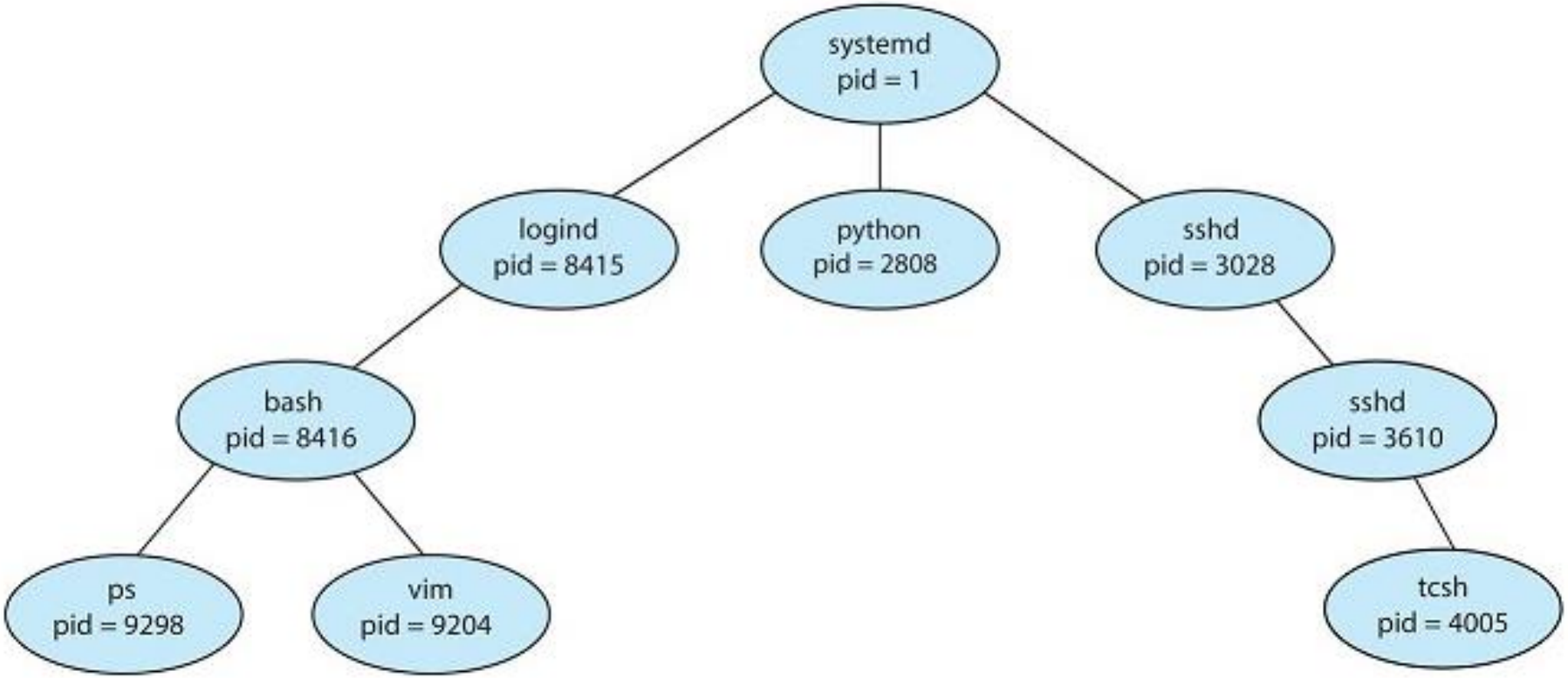


Figure 3.7 A tree of processes on a typical Linux system.

Operations on Processes: Process Creation

- Two address-space possibilities for the new process
  - ✓ The child process is a duplicate of the parent process.
  - ✓ The child process has a new program loaded into it.
- UNIX examples

부모 프로세스를 복사해서 자식 프로세스가 생성됨

  - ✓ **`fork()`** system call creates a new process
  - ✓ After a **`fork()`** system call, one of the two processes uses the **`exec()`** system call to replace the process' memory space with a new program.
  - ✓ **`exec()`** system call loads a binary file into memory (destroying the memory image of the program containing the **`exec()`** system call) and starts its execution.

Operations on Processes: Process Creation

- UNIX examples
  - ✓ The value of the variable `pid` for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual `pid` of the child process).
  - ✓ The parent waits for the child process to complete with the `wait()` system call.
  - ✓ When the child process completes (by either implicitly or explicitly invoking `exit()`), the parent process resumes from the call to `wait()`, where it completes using the `exit()` system call.

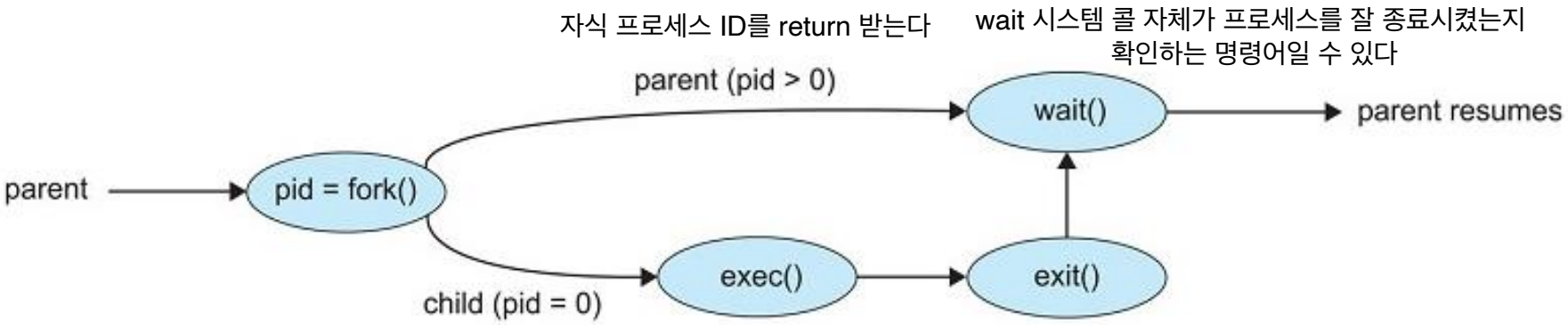


Figure 3.9 Process creation using the `fork()` system call.

Operations on Processes: Process Creation

```
#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;
}
```

에러 처리

자식 프로세스

그 외에는 부모 프로세스다.

Operations on Processes: Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using `exit()`.
  - ✓ Return a status value to its waiting parent process (via `wait()` system call)
  - ✓ All the resources of process are deallocated by the OS
- Parent may terminate execution of one of its children (`abort()`)
  - ✓ The child has exceeded its usage of some of allocated resources. 정상적이지 않을 때, abort 정상일 때, Commit
  - ✓ The task assigned to the child is no longer required.
  - ✓ The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.
    - ❖ All children must be terminated - *cascading termination*
- **wait()** : Wait for termination of a child process, returning the pid:

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

  - ✓ A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie** process. 실행하면 종료해야하는데, 부모 프로세스에서 거둬들이지 않은 경우 (자식 프로세스를 좀비로 만듦) 깨끗하게 클리어가 안되서 pid를 모두 갖고 있으며, 메모리 낭비를 시키는 상태임
  - ✓ If parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as **orphans**. 부모 프로세스가 자식 프로세스를 거둬야하는데, 부모가 먼저 종료되는 상황

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
- Cooperating processes require an **interprocess communication** (**IPC**)
- Two models of IPC
  - ✓ Shared memory
  - ✓ Message passing 분산시스템은 메세지 패싱을 기본으로 채택함

Interprocess Communication

- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data.
  - ✓ Shared memory
    - ❖ A region of memory that is shared by cooperating processes is established. 공유 영역 설정
  - ✓ Message passing
    - ❖ Communication takes place by means of messages exchanged between the cooperating processes.
  - ✓ Message passing is easier to implement than shared memory
  - ✓ Shared memory can be faster than message passing 메세지 패싱이 구현이 더 용이함
    - ❖ Message passing systems are implemented using system calls and thus require the more time-consuming task of kernel intervention 메세지 패싱은 시스템 콜을 구현해서 사용한다. 시간 소모적인 작업을 더 추가적으로 요구할 수 있다.
  - ✓ Shared memory suffers from cache coherency issues, which arise because shared data migrate among the several caches.

공유하는 영역에는 반드시 Coherency 문제가 발생할 수 밖에 없다

Communications Models

Shared memory : 커널을 건드리지 않음
Message Passing : 커널을 직접 건드림

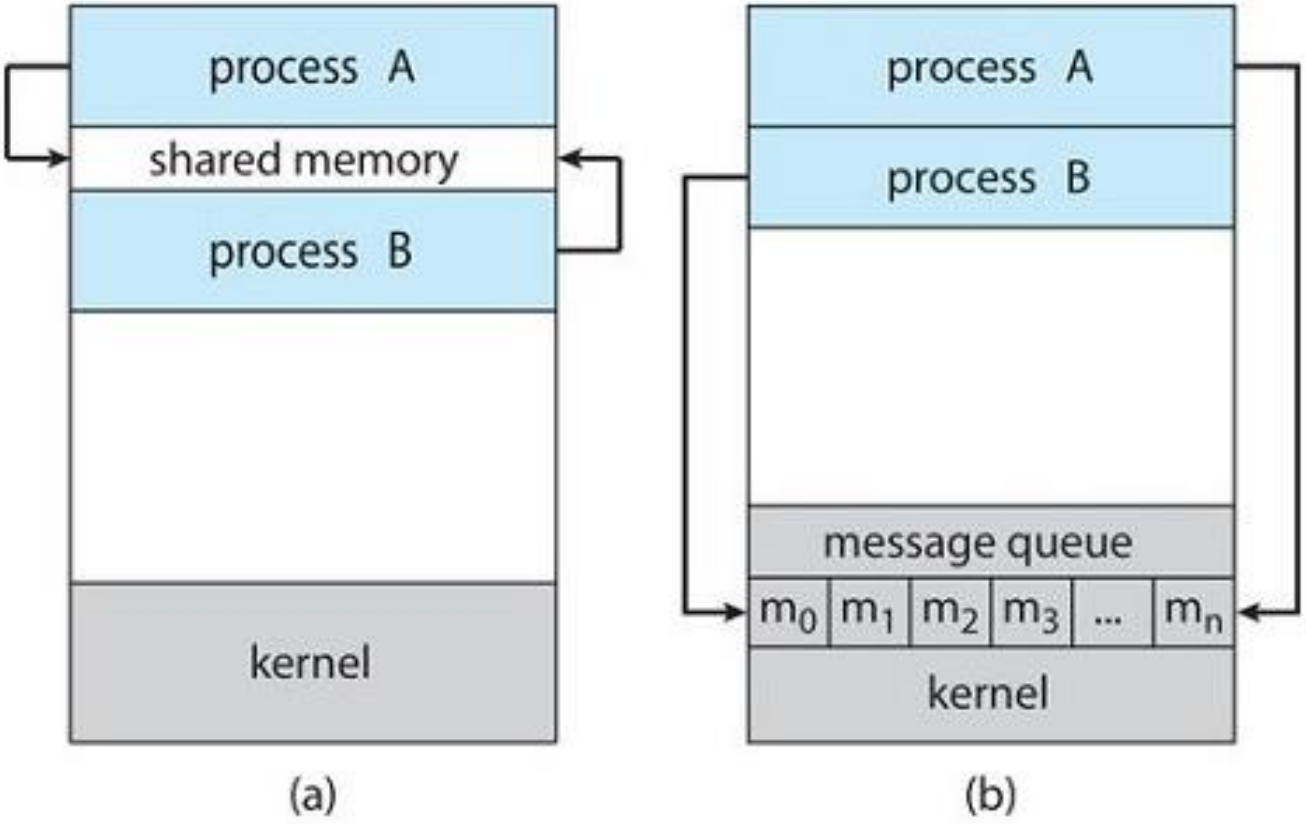


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

Producer-Consumer Problem

- IPC using shared memory requires communicating processes to establish a region of shared memory 큐를 기반으로한 커뮤니케이션 방법
- Producer-consumer problem - paradigm for cooperating processes
- A **producer** process produces information that is consumed by a **consumer** process 프로듀서 : 정보 생성
  소비자 : 정보를 소비하는 역할
  - ✓ One solution to the producer-consumer problem uses shared memory
  - ✓ A buffer of items that can be filled by the producer and emptied by the consumer
    - ❖ *unbounded-buffer* places no practical limit on the size of the buffer
    - ❖ *bounded-buffer* assumes a fixed buffer size

Bounded Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

큐를 기반으로 하는 것이다.

  - ✓ **in** – points to the next free position in the buffer
  - ✓ **out** – points to the first full position in the buffer

삭제될 위치는 Out, 삽입될 위치는 in으로 표기

- This scheme allows at most BUFFER_SIZE-1 items in the buffer at the same time.

알고리즘 구조로 인해 BUFFER_SIZE - 1만큼 저장

Bounded Buffer – Producer

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

   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Circular Queue를 사용해서 구현하는 형태

버퍼가 full인지 확인하는 것이다

꽉 차면 아무것도 할 일이 없으므로 wait()

아직 꽉 차지 않았다면, in pointer를 움직여 내용물을 채워나간다

Bounded Buffer – Consumer

내용물이 저장되어 있어야만 사용할 수 있는 형태

```
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 */
}
```

큐가 비어있으므로 아무것도 안한다. wait()

큐 값이 있으면 out pointer를 움직이면서 내용물을 소비한다.

Interprocess Communication (IPC)

이 파트는 설명을 간략하게 하고 넘어갔음

- Message passing
  - ✓ Mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space
- Message passing facility provides two operations:
  - ✓ **send**(*message*) – be fixed or variable in size
  - ✓ **receive**(*message*)
- If *P* and *Q* want to communicate, they need to:
  - ✓ establish a *communication link* between them
  - ✓ exchange messages via send/receive
- Implementation of communication link
  - ✓ Physical implementation (e.g., shared memory, hardware bus)
  - ✓ Logical implementation
    - ❖ Direct or indirect communication
    - ❖ Synchronous or asynchronous communication
    - ❖ Automatic or explicit buffering

Implementation Questions

- 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?

Direct Communication

- Each process that wants to communicate must explicitly name the recipient or sender of the communication.
  - ✓ **send** (*P, message*) – send a message to process P
  - ✓ **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - ✓ A Link is established automatically
  - ✓ A link is associated with exactly two processes
  - ✓ Between each pair of processes, there exists exactly one link
- Addressing
  - ✓ Symmetry addressing – both the sender process and the receiver process must name the other to communicate
  - ✓ Asymmetry addressing – only the sender names the recipient; the recipient is not required to name the sender.
  - ✓ send() and receive() primitives :
    - ❖ send(P, message) – Send a message to process P
    - ❖ receive(id, message) – Receive a message from any process

Indirect Communication

- Messages are sent to and received from mailboxes, or ports
  - ✓ Each mailbox has a unique identification 메일 박스를 공유하는 프로세스들끼리만 전송한다는 뜻이다
  - ✓ Two processes can communicate only if the processes have a shared mailbox
- Primitives are defined as follows:
  - ✓ **send**(*A, message*) – send a message to mailbox A
  - ✓ **receive**(*A, message*) – receive a message from mailbox A
- Properties of communication link
  - ✓ A link is established between a pair of processes only if both members of the pair have a shared mailbox
  - ✓ A link may be associated with more than two processes
  - ✓ Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox

Indirect Communication

- Mailbox sharing
  - ✓ P_1, P_2, and P_3 share mailbox A
  - ✓ P_1 sends a message to A; Both P_2 and P_3 execute a `receive()` from A.
  - ✓ Which process will receive the message sent by P_1?
- Solutions
  - ✓ Allow a link to be associated with two processes at most.
  - ✓ Allow at most one process at a time to execute a `receive()` operation.
  - ✓ Allow the system to select arbitrarily which process will receive the message. The system may define an algorithm for selecting which process will receive the message.

Indirect Communication

- A mailbox may be owned either by a process or by the operating system
- If the mailbox is owned by a process, then we distinguish between the owner and the user.
- A mailbox that is owned by the operating system has an existence of its own.
- Operations
  - ✓ Create a new mailbox
  - ✓ Send and receive messages through the mailbox
  - ✓ Delete a mailbox

Synchronization

I/O에서의 Synchronization : 컨트롤을 바로 넘기지 않고 끝날 때까지 기다리는 것

- Message passing may be either **blocking** or **non-blocking**
- **Blocking**, known as **synchronous**
  - ✓ **Blocking send** – the sending process is blocked until the message is received
  - ✓ **Blocking receive** – the receiver blocks until a message is available
- **Non-blocking**, known as **asynchronous**
  - ✓ **Non-blocking send** – the sending process sends the message and resumes operation
  - ✓ **Non-blocking receive** – the receiver retrieves either a valid message or a null

Synchronization (Cont.)

구현할 때의 내용이라 빠르게 넘어감

- Different combinations of `send()` and `receive()` are possible.
  - ✓ When both send() and receive() are blocking, we have a **rendezvous**.
- The solution to the producer-consumer problem becomes trivial when we use blocking `send()` and `receive()` statements.

```
message next_produced;
while (true) {
    /* produce an item in next_produced */
    send(next_produced);
}

message next_consumed;
while (true) {
    receive(next_consumed);

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

Buffering

참고하라고 스킵

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
- Such queues can be implemented in three ways.
  - ✓ Zero capacity – a maximum length of zero
    - ❖ A message system with no buffering
    - ❖ The sender must block until the recipient receives the message.
  - ✓ Bounded capacity – finite length *n* 코딩하면 당연히 bounded
    - ❖ If the link is full, the sender must block until space is available in the queue.
  - ✓ Unbounded capacity – infinite length
    - ❖ The sender never blocks.

Examples of IPC Systems – Windows

참고하라고 스킵

- Message-passing facility in Windows is called the **advanced local procedure call (ALPC)** facility.
- Windows uses a port object to establish and maintain a connection between two processes. - connection ports and communication ports.
- When an **ALPC** channel is created, one of three message-passing techniques is chosen:
  1. For small messages (up to 256 bytes), the port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
  2. Larger messages must be passed through a section object, which is a region of shared memory associated with the channel.
  3. When the amount of data is too large to fit into a section object, an API is available that allows server processes to read and write directly into the address space of a client.

Examples of IPC Systems – Windows

참고하라고 스킵

- When a client wants services from a subsystem, it opens a handle to the server's connection-port object and sends a connection request to that port. The server then creates a channel and returns a handle to the client.
- The channel consists of a pair of private communication ports.

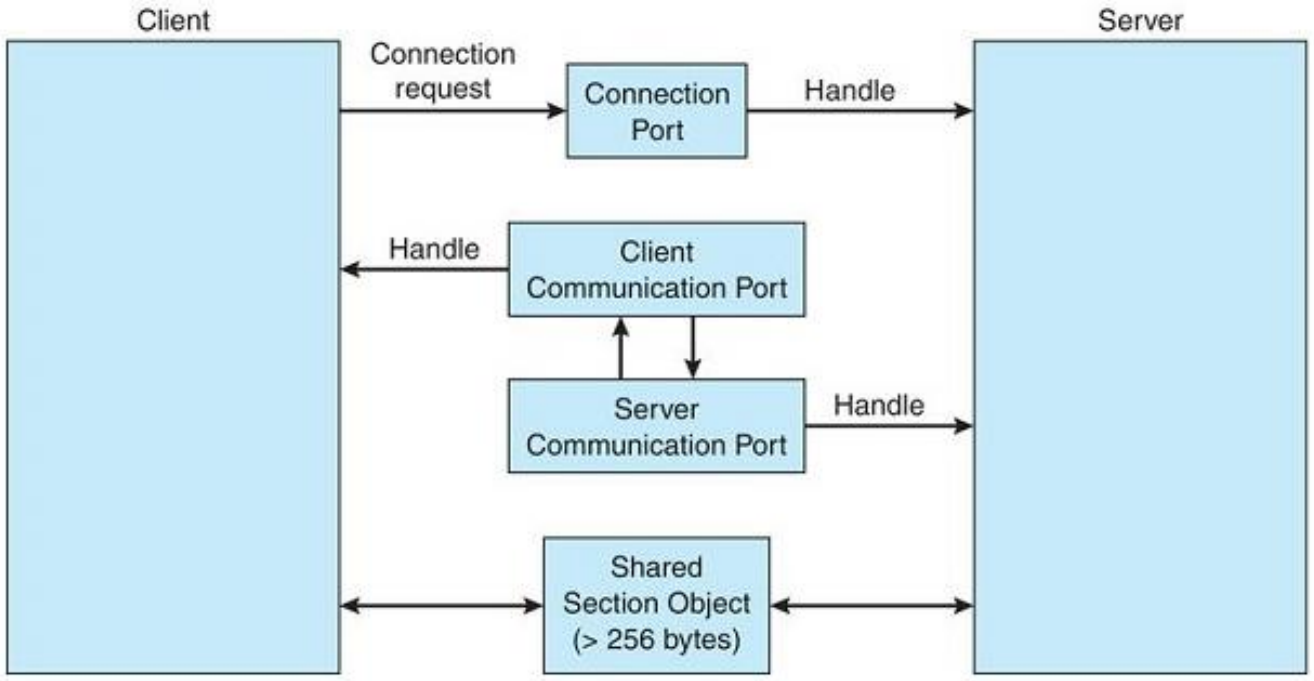


Figure 3.19 Advanced local procedure calls in Windows.

Communication in Client-Server Systems

참고하라고 스킵

- Sockets
  - A socket is defined as an *endpoint for communication*
  - Concatenation of an IP address and a port number
  - Sockets use a client-server architecture
    - ✓ The server waits for incoming client requests by listening to a specified port.
    - ✓ Once a request is received, the server accepts a connection from the client socket to complete the connection.
  - Figure 3.26
    - ✓ If a client on host X with IP address **146.86.5.20** wishes to establish a connection with a web server (which is listening on port **80**) at address **161.25.19.8**, host X may be assigned port **1625**.
    - ✓ The connection will consist of a pair of sockets: (**146.86.5.20:1625**) on host X and (**161.25.19.8:80**) on the web server

Socket Communication

참고하라고 스킵

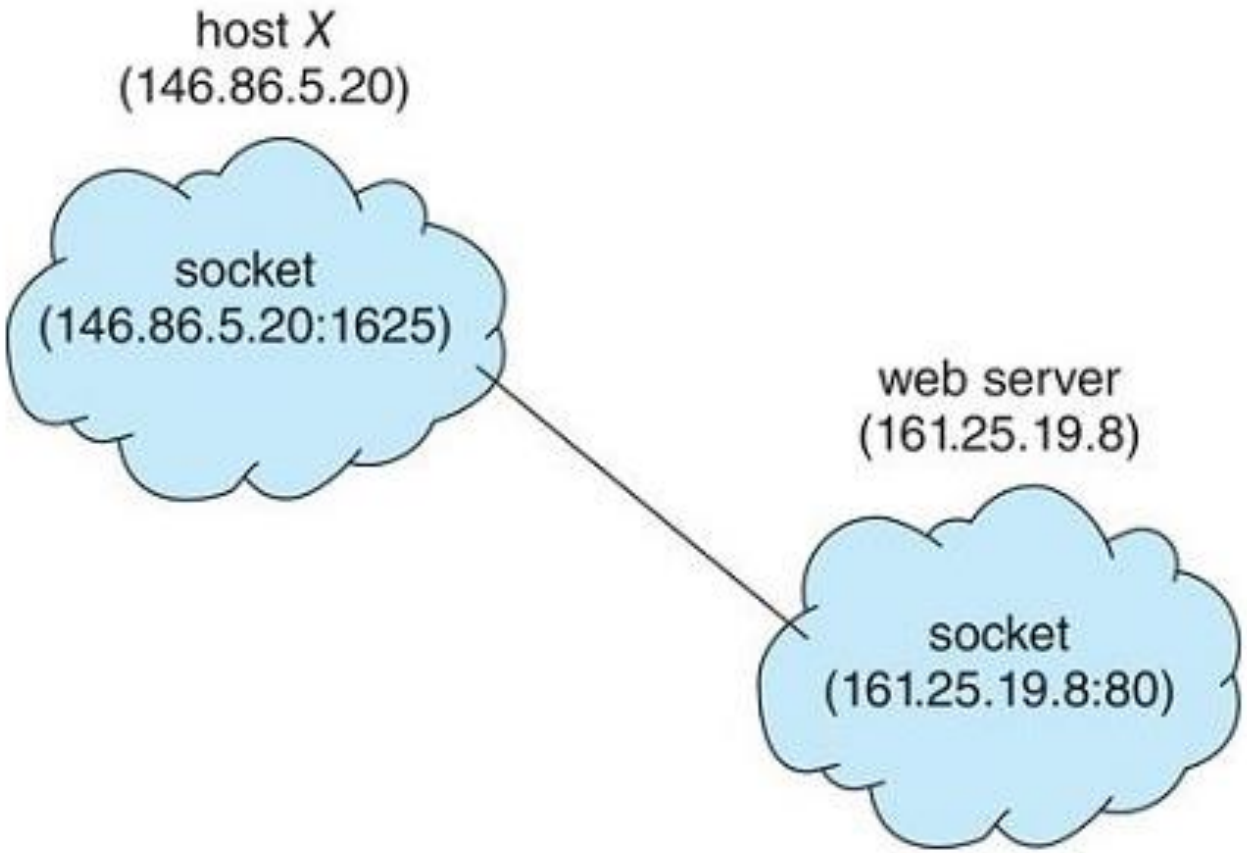


Figure 3.26 Communication using sockets.

Remote Procedure Calls

참고하라고 스킵

- Remote procedure call (RPC) abstracts the procedure-call mechanism for use between systems with network connections.
- **Stubs** – client-side proxy for the actual procedure on the server.
- When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure.
  - ✓ This stub locates the port on the server and *marshalls* the parameters.
  - ✓ Parameter marshalling involves packaging the parameters into a form that can be transmitted over a network.
- A similar stub on the server-side stub receives this message and invokes the procedure on the server.