PART 2. PROCESS MANAGEMENT

Chapter 3: Process Concept

WHAT'S AHEAD:

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

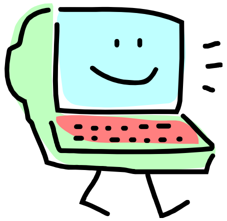
WE AIM:

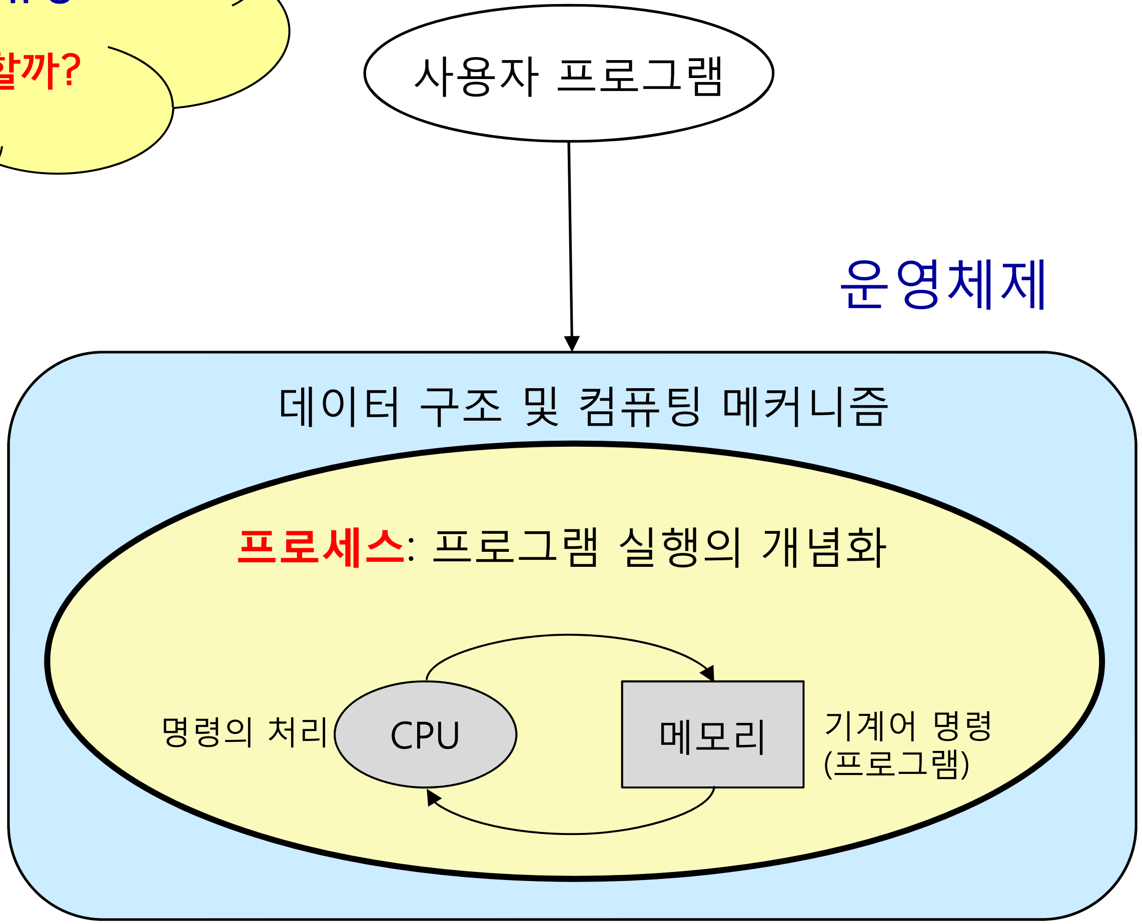
- To introduce the notion of a process
- To describe the various features of processes
- To explore interprocess comm.

Note: These lecture materials are based on the lecture notes prepared by the authors of the book titled *Operating System Concepts*, 9e (Wiley)

핵심 요점   프로세스란 무엇인가? 사용 동기는?

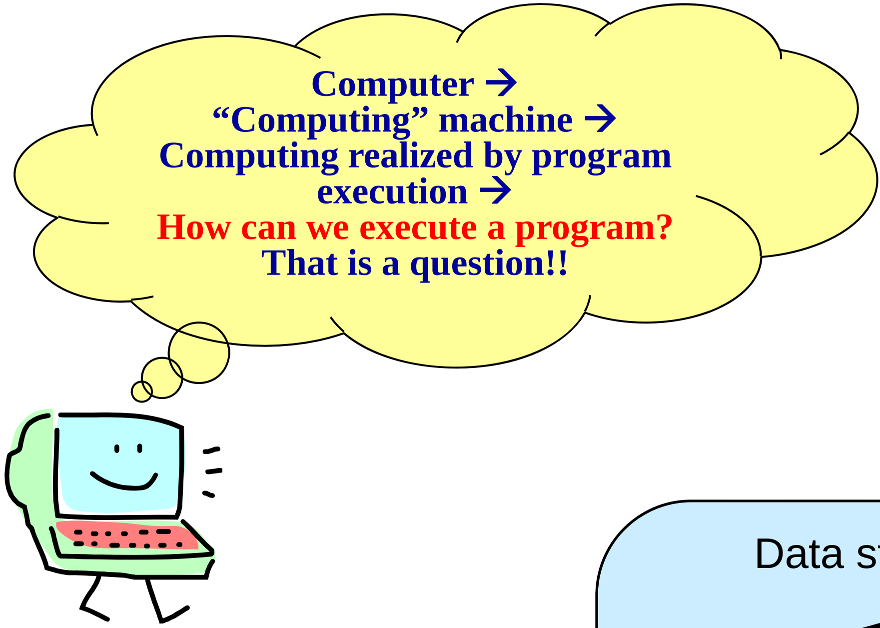
컴퓨터 → 컴퓨팅 기계 →
프로그램 실행에 의해 컴퓨팅
실현 →
어떻게 프로그램을 실행할까?
이것이 문제로다!!

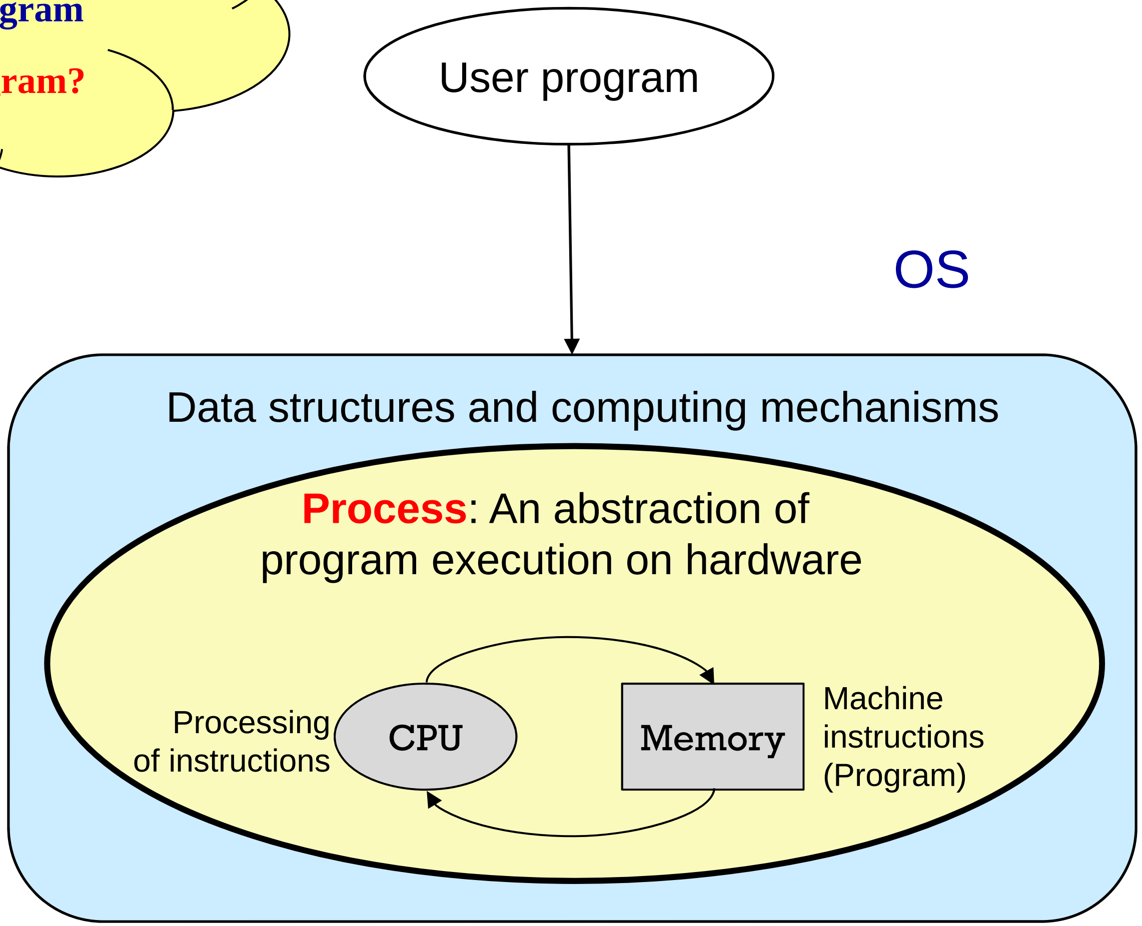




Core Ideas

What and Why a Process?





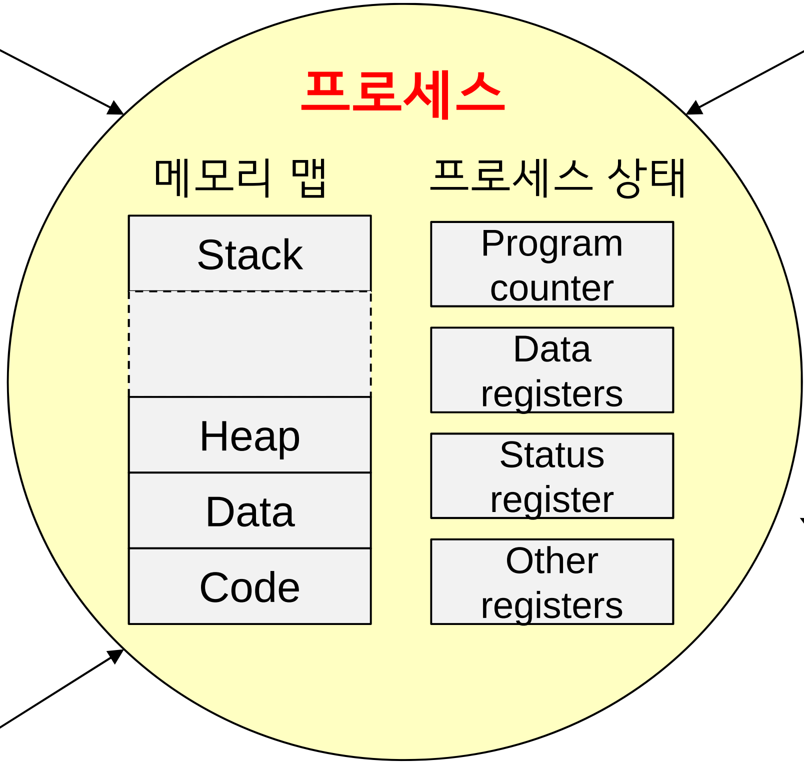
핵심요점 프로세스와 운영체제 환경

프로세스에 대한 연산

- 생성(creation)
- 종료(termination)
- 일시정지(suspension)
- 대기(wait)
- 적재 및 링킹 (loading & linking)
- 실행(execution)
- 등등

OS 제어 테이블

- 프로세스 테이블 (PCBs)
- 메모리 테이블
- 파일 테이블
- 입출력장치 테이블



프로세스 스케줄링

- 알고리즘
- 멀티프로그래밍
- 멀티태스킹
- 문맥 교환

프로세스 간 통신

- 공유 메모리
- 메시지 전달

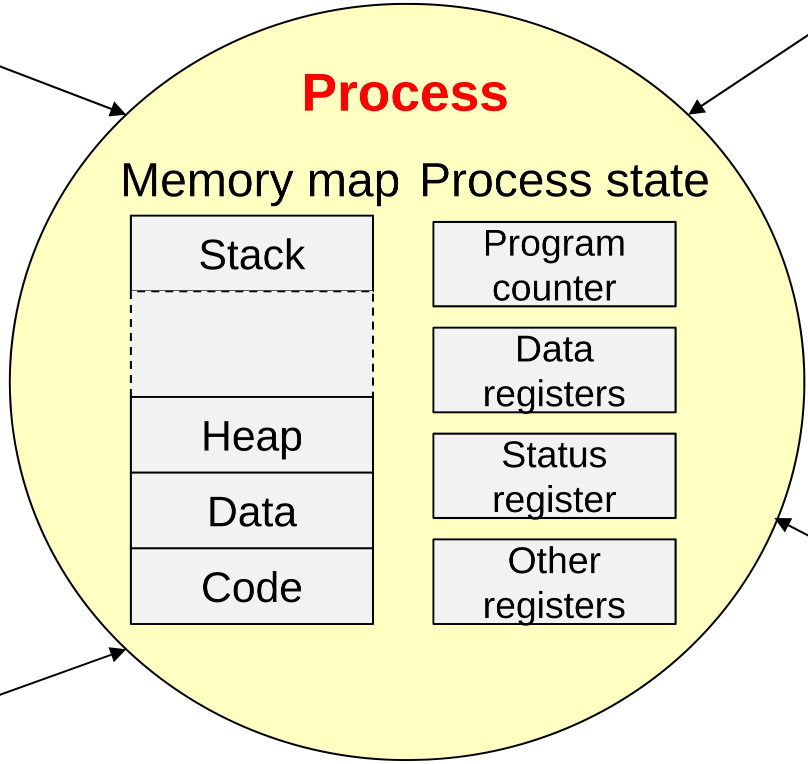
Core Ideas Process and OS Support

Operations on processes

- Creation
- Termination
- Suspension
- Wait
- Loading & linking
- Execution
- Etc.

OS control tables

- Process table (PCBs)
- Memory table
- File table
- I/O device table



Interprocess communication

- Shared memory
- Message passing

Process scheduling

- Algorithms
- Multiprogramming
- Multitasking
- Context switching

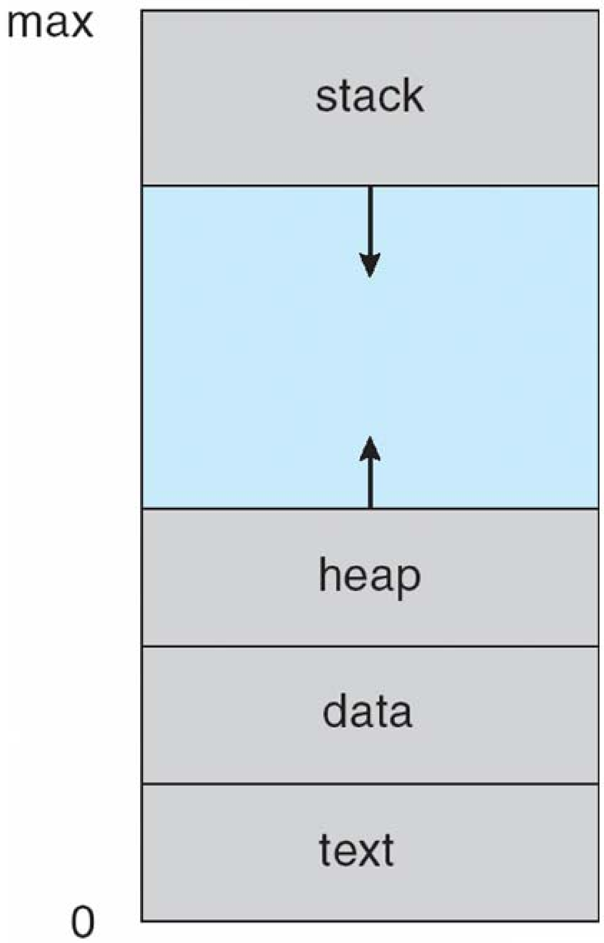
Process Concept

- An OS executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job, task*** and ***process*** almost interchangeably
- **Process** – Definition
  - A program in execution
  - A series of computational operations performed on the basis of instructions in a program
  - Process execution must progress in sequential fashion
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory

Process Concept (*Cont.*)

- Multiple parts comprising a process
  - 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
- 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

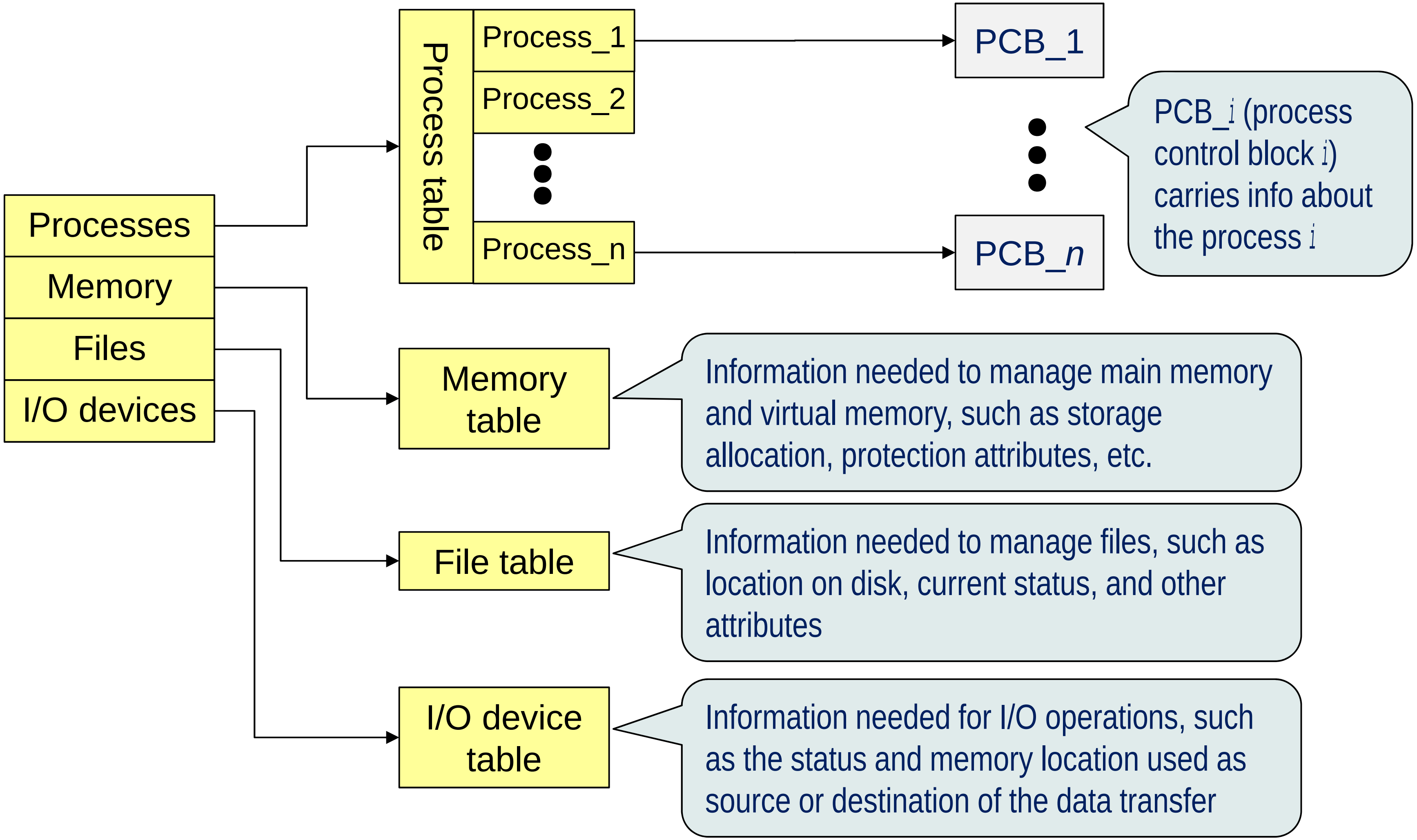


- Part of process image
- Defined in the virtual address space

Process Description

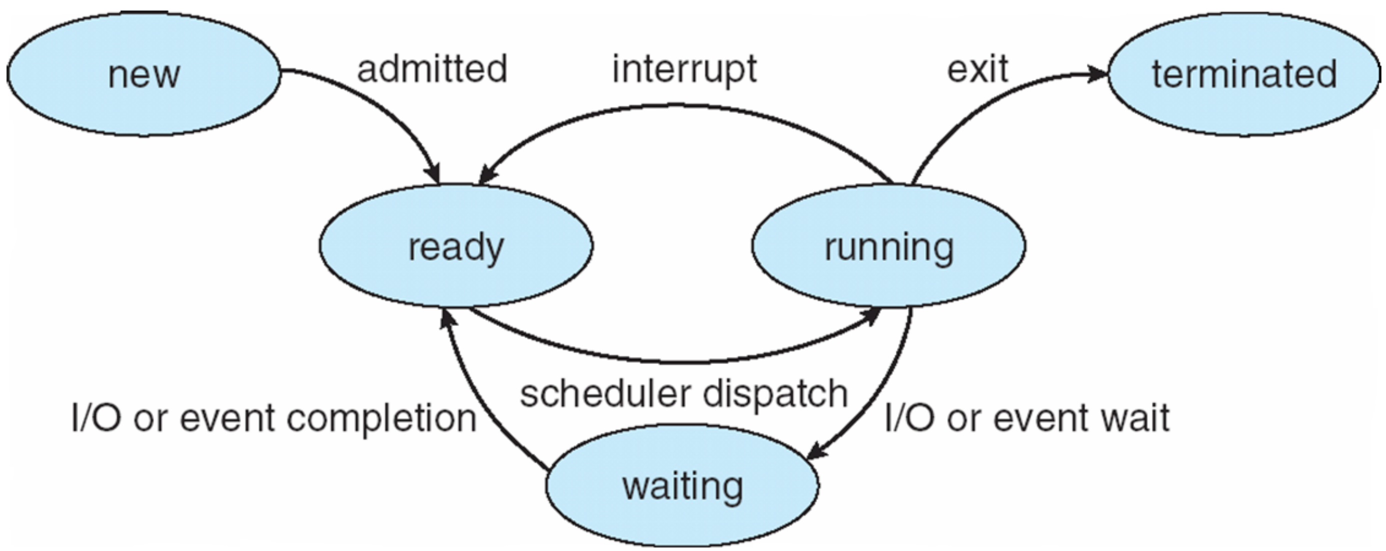
- OS controls physical/logical resources
  - Physical resources: CPU, main memory, I/O devices, etc.
  - Logical resources: processes, virtual memory, data structures, files, etc.
- OS controls resources using "control tables"
  - See next slide
- Process control structures
  - Information on executable program(s) and associated data structures
    - Process image: the collection of program, data, stack, and process control block (PCB)
  - Information on the usage of physical/logical resources
    - PCB contains process attributes that are used by the OS for process control
    - Processor state information: data registers, condition code register, stack pointer, program counter

OS Control Tables



Process State

- 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



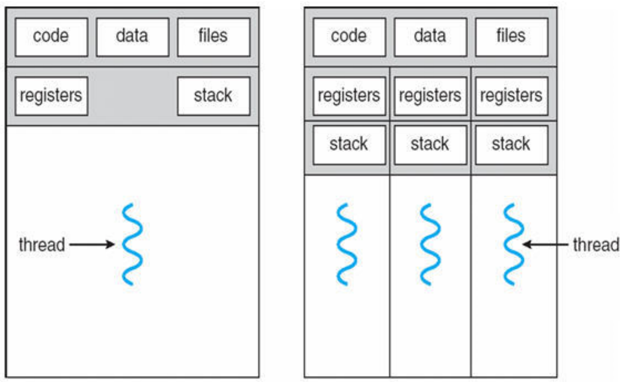
Process Control Block (PCB)

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

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



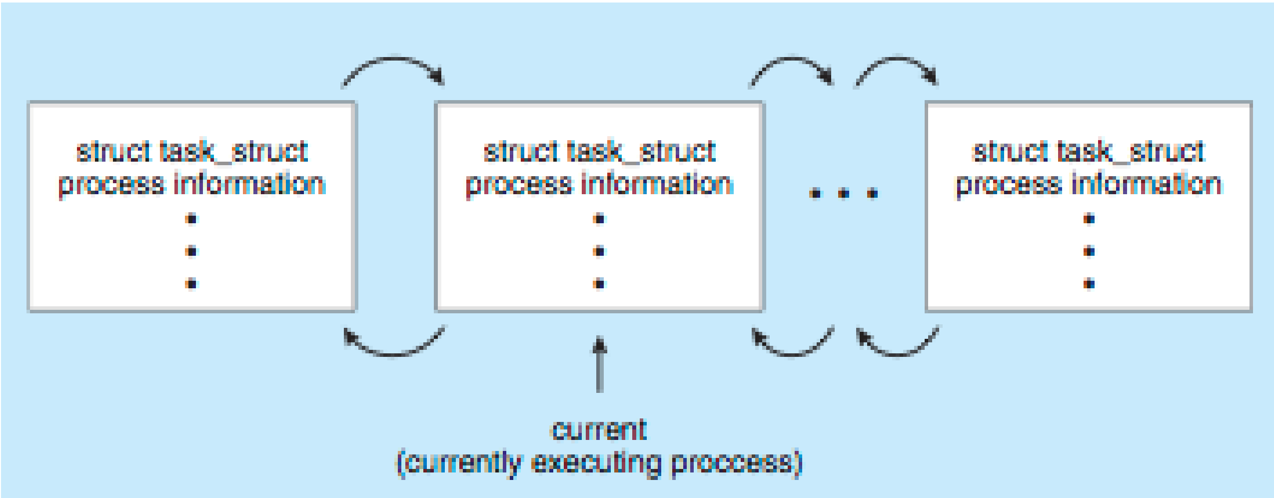
single-threaded process   multithreaded process

What is a thread (of execution)?
- a sequence of instructions in execution
- an execution trace of instructions
- schedulable as a separate entity
- sometimes, called “lightweight process”

Process Representation in Linux

- Represented by the C structure **task_struct**

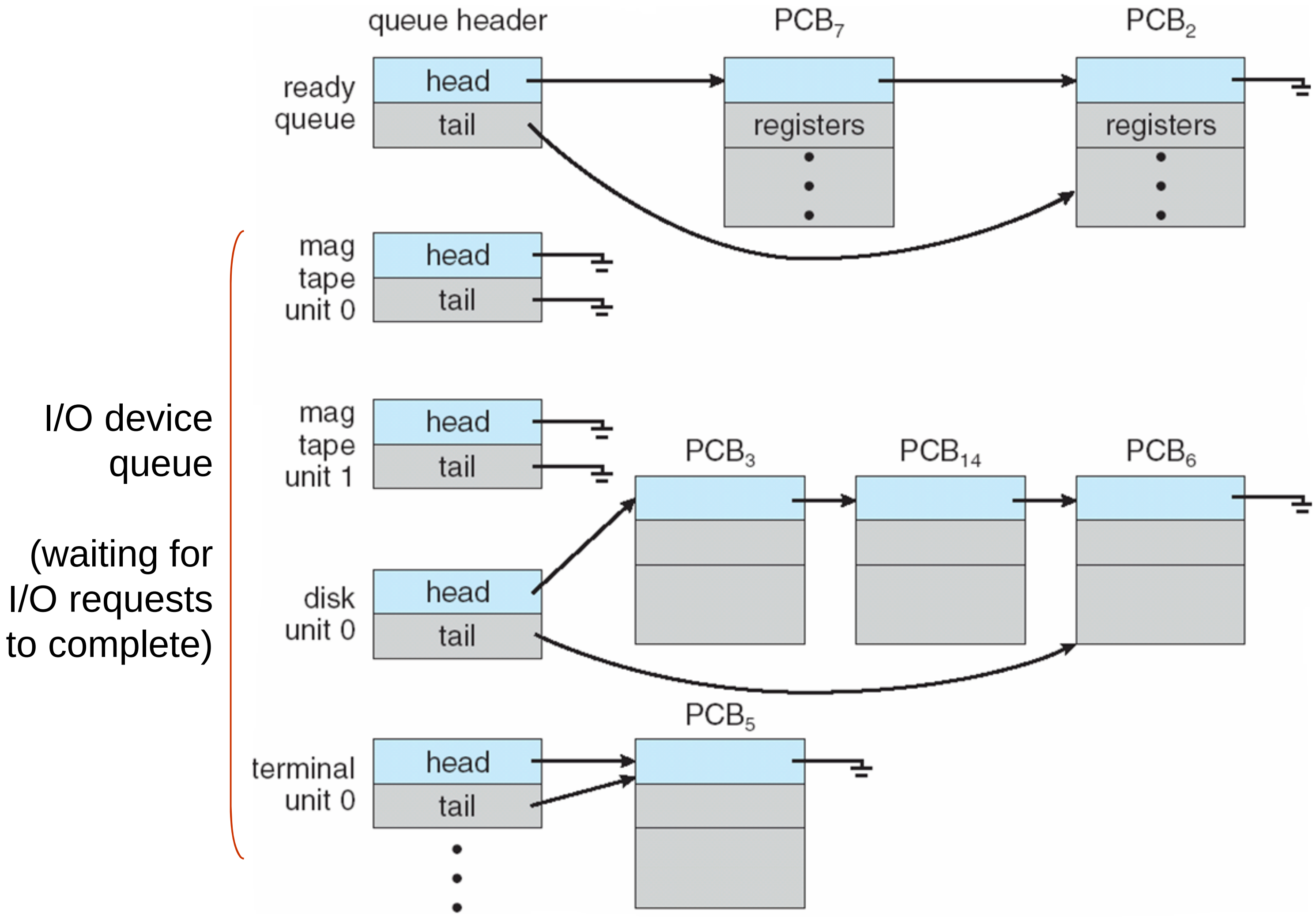
```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm struct *mm; /* address space of this process */
```



Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects one 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

Ready Queue And Various I/O Device Queues

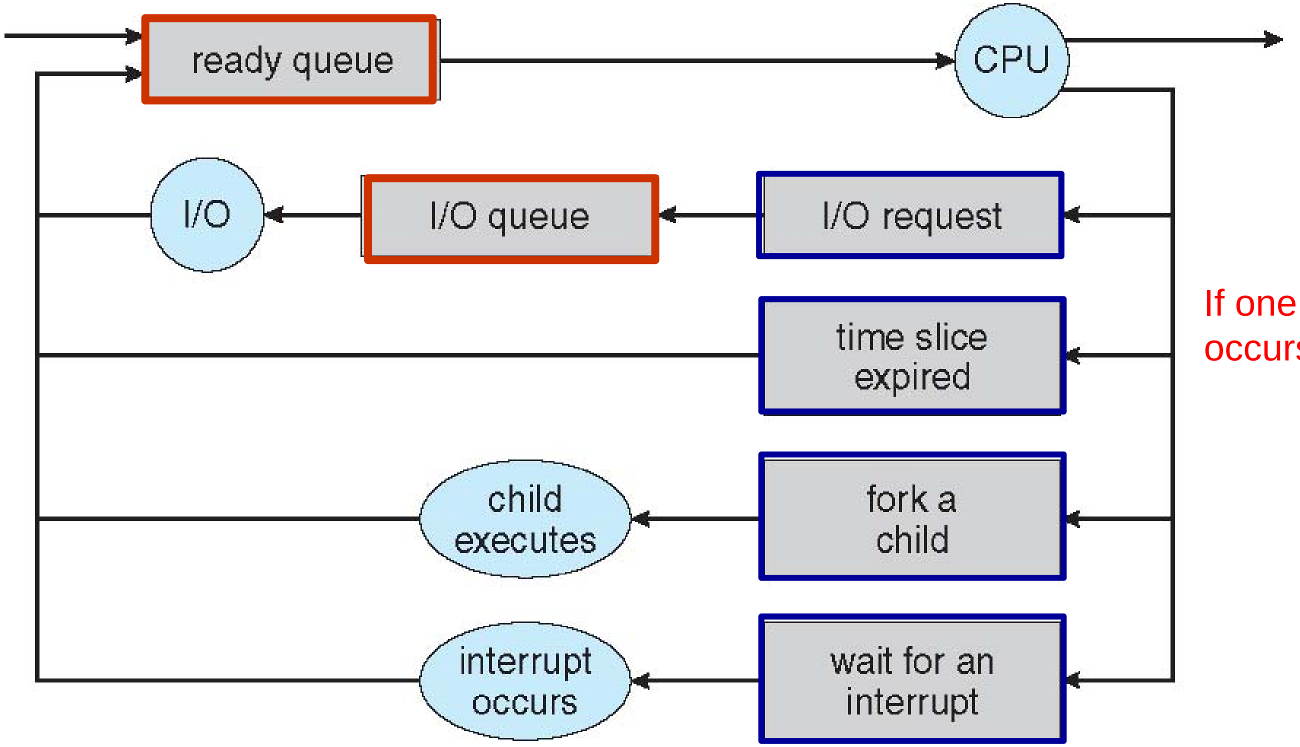


Representation of Process Scheduling

- Queuing diagram represents queues, resources, flows

Processes enter the execution system

Upon completion leave the system



If one of these events occurs, stop running

- Circles: servers, or the resources that serve the queues
- Arrows: the flow of processes in the system
- Red boxes: queues
- Blue boxes: description of events

Schedulers

- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
- **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 very frequently (milliseconds) $\Rightarrow$ (must be fast)

- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (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***

Multitasking in Mobile Systems

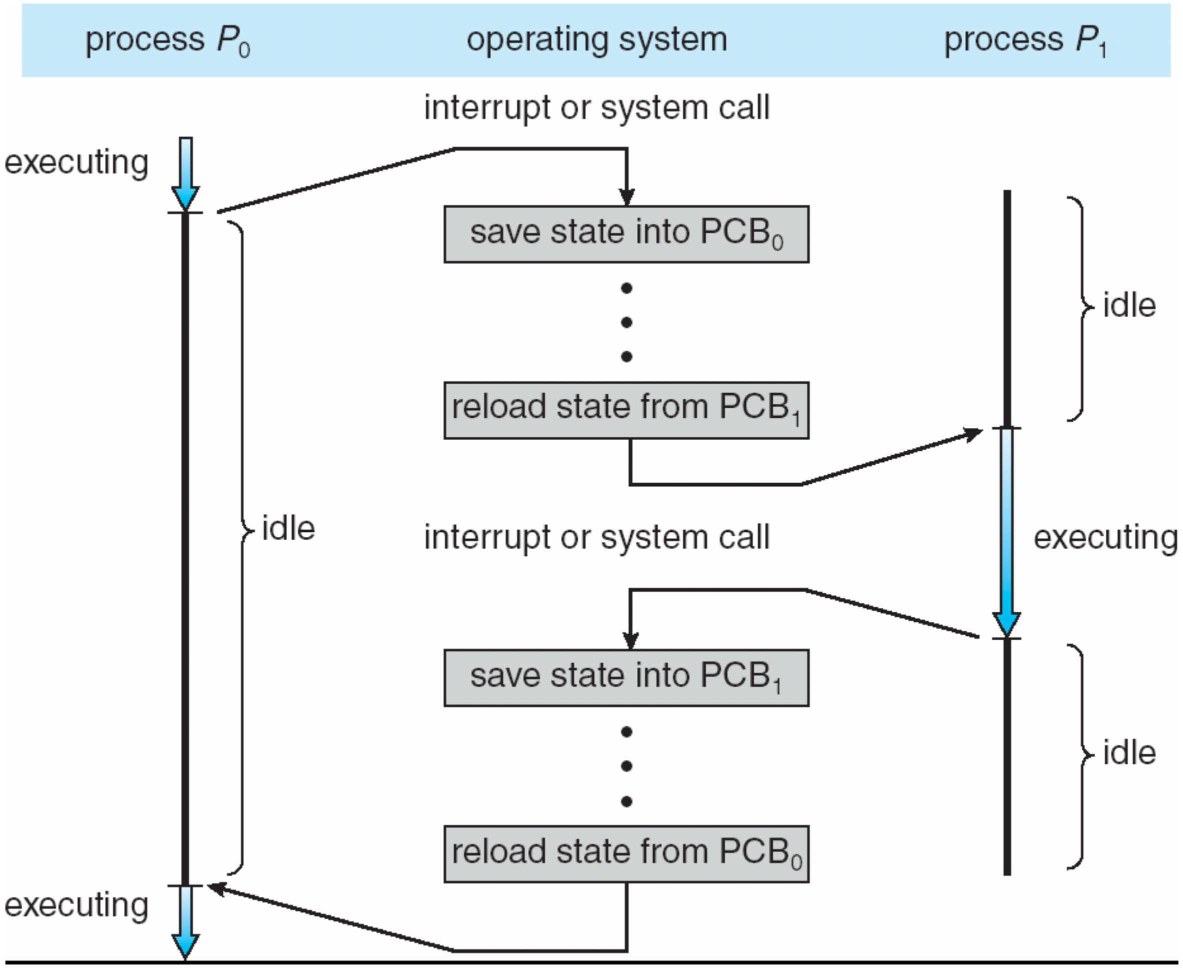
- Some systems / early systems allow only one process to run, others suspended
- Apple iOS probably limits multitasking due to battery life and memory use concerns
  - Single **foreground** process - controlled via user interface
  - Multiple **background** processes – in memory, running, but not on 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

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 is represented in the PCB
  - context: the state of a process
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB -> longer the context switch
- Context-switch time: dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

Context Switch (Cont.)

CPU switch from process to process



Operations on Processes

- System must provide mechanisms for process management
  - Process creation
  - Process termination
  - Process loading and execution
  - Process suspension
  - and so on

- Such mechanisms are implemented by system services
  - `fork()`, `exec()`, `wait()`, `exit()`, etc.

(*You are supposed to be familiar with these systems services*.)

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

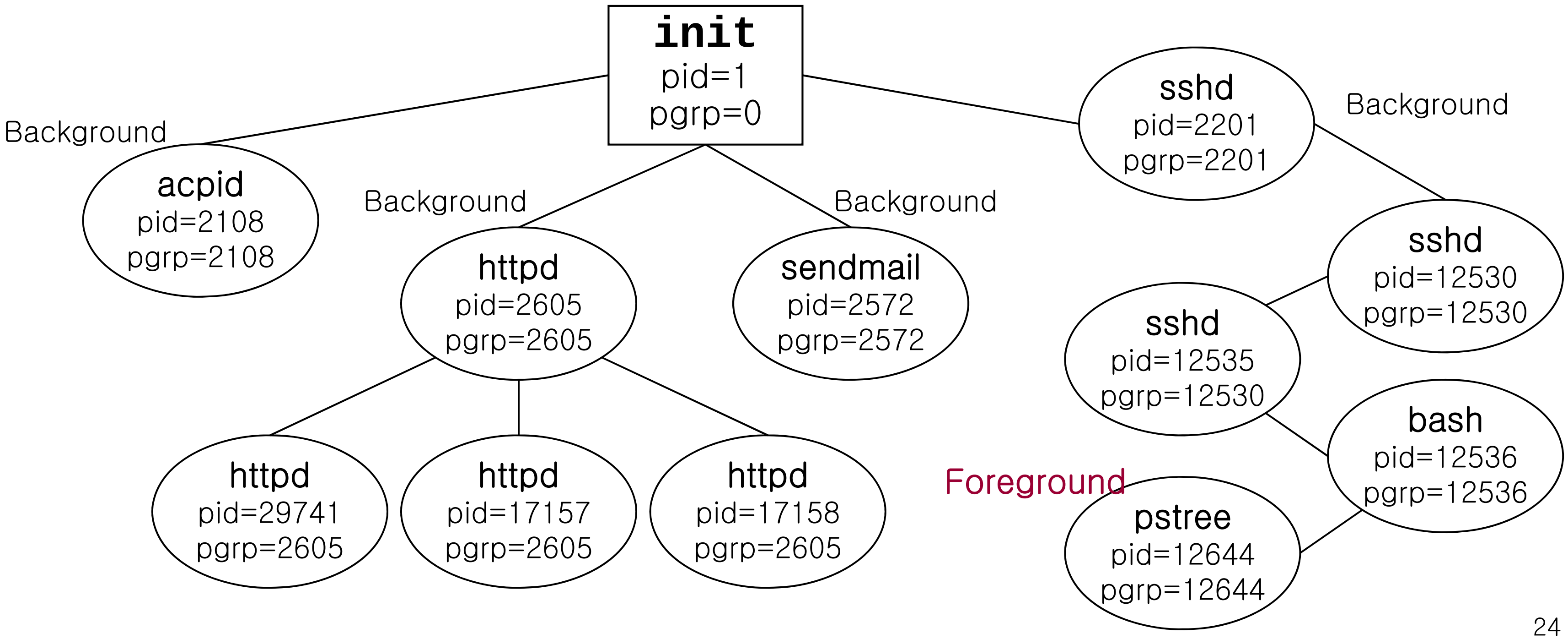
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Example: A tree of processes in Linux

  `linux> pstree -p                // -p: for pid`

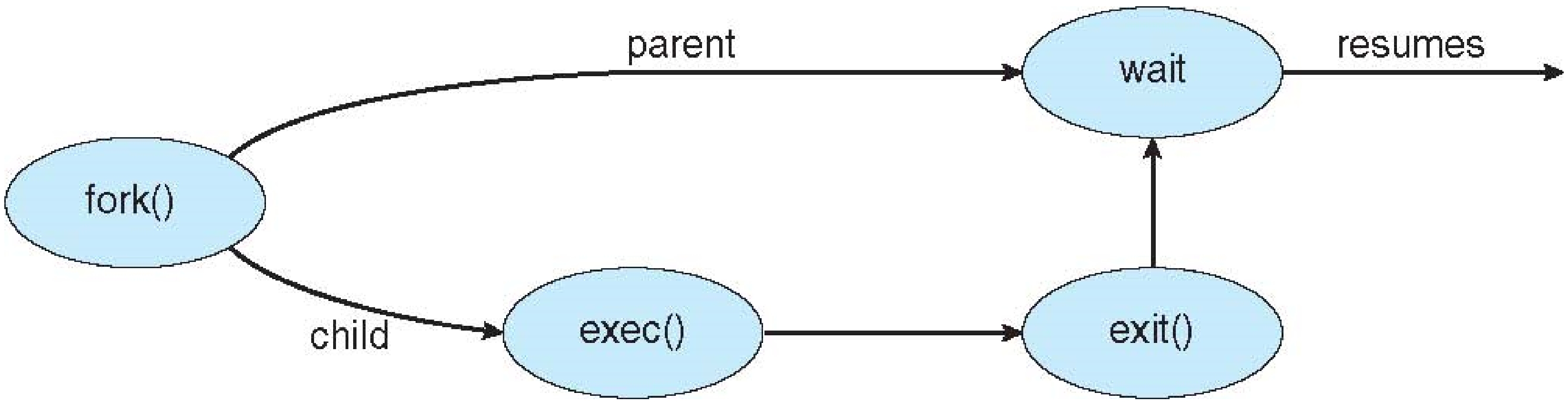
```
init(1)-+-acpid(2108)
        |-httpd(2605)-+-httpd(29741)
        |             |-httpd(17157)
        |             |-httpd(17158)
        |             `-httpd(23987)
        |-sendmail(2572)
        |-sshd(2201)---sshd(12530)---sshd(12535)---bash(12536)---pstree(12644)
```



Process Creation (Cont.)

- Address space
  - Child is given a duplicate of parent
  - Child has a program loaded into it

- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



Examples

Creating a separate process via Unix `fork()` system call (left) and Windows API (right)

```
#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;
}
```

`pid`: a local variable, whose value is assigned by `fork()`

`pid` ← pid of child process

The **`getpid()`** function returns the process ID of the calling process.

```
#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 asks the operating system to delete it (**`exit()`**)
  - Output data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**`abort()`**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**
- Wait for termination, returning the pid:

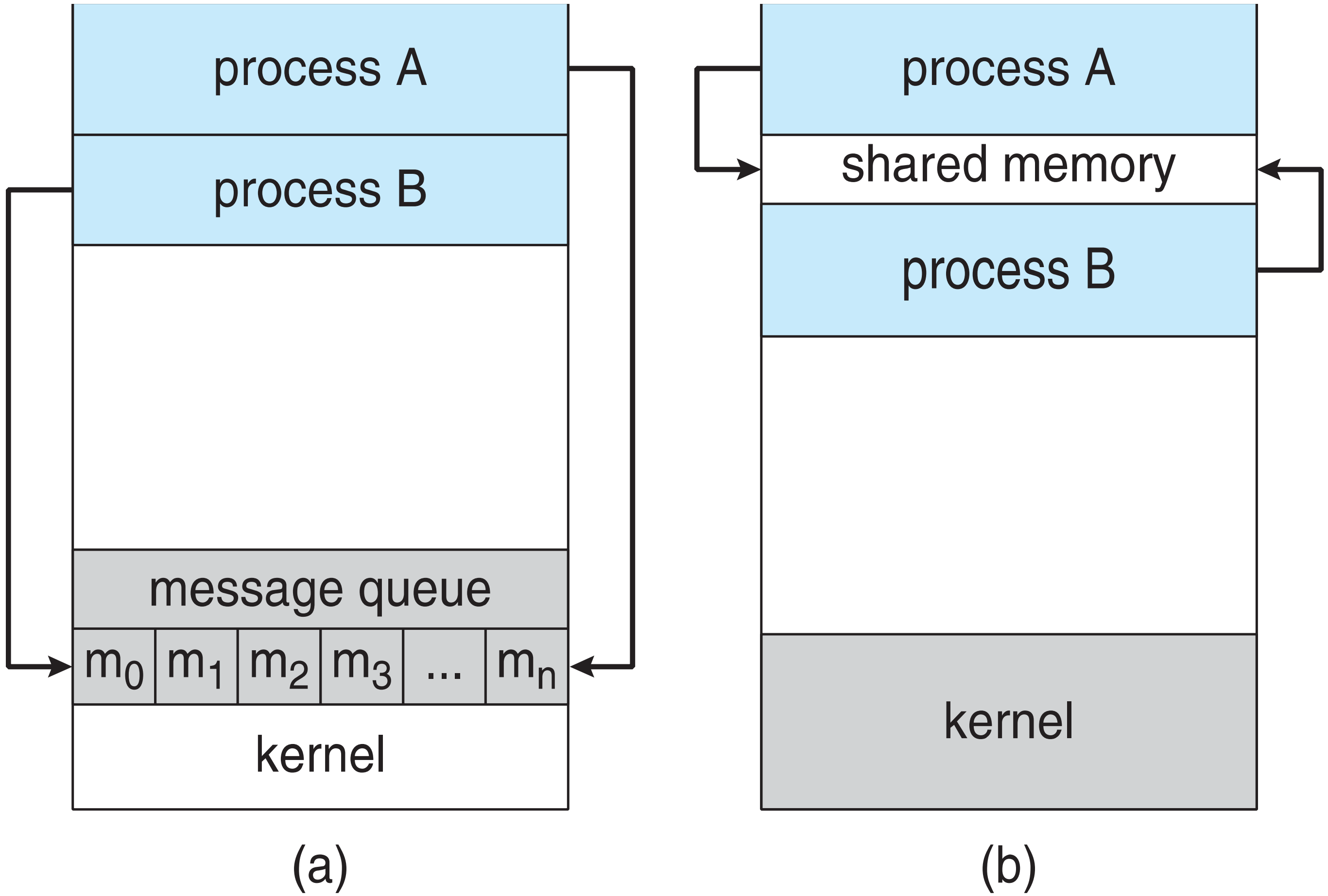
```
pid t pid; int status;
pid = wait(&status);
```

- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- ***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
  - 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

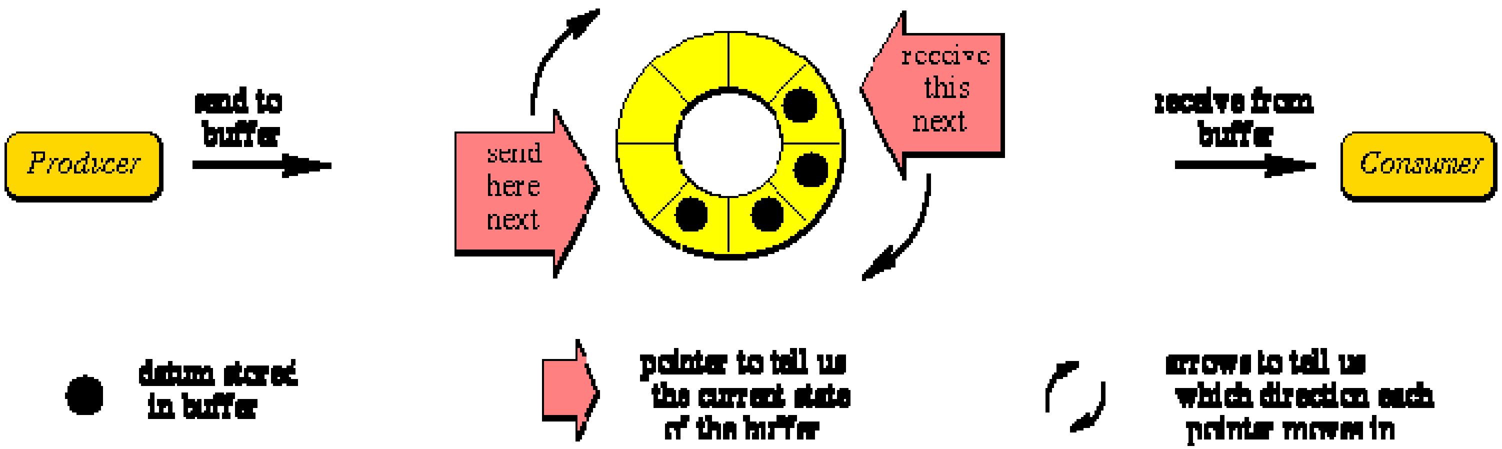


Message passing

Shared memory

Producer-Consumer Problem

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



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

Interprocess Communication – Shared Memory

- Bounded-Buffer – Shared-Memory Solution
  - Shared data

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

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

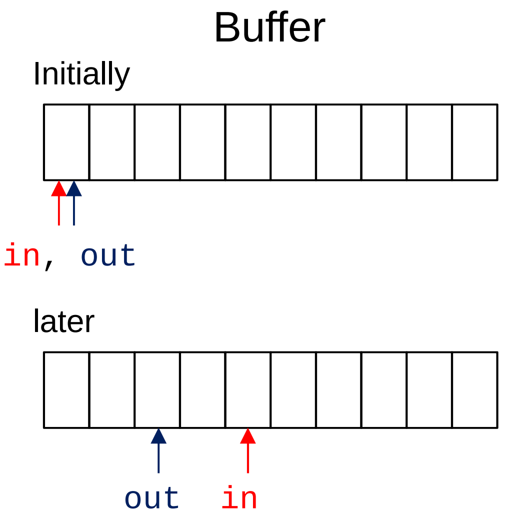
  - Solution is correct, but can only use BUFFER_SIZE-1 elements (next slide)

Bounded-Buffer – Producer & Consumer

Case of shared memory

- 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;
}
```



- Consumer

```
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 */
}
```

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*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a ***communication link*** between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., direct or indirect, synchronous or asynchronous, automatic or explicit buffering)
- Specifying a source/destination process
  - direct addressing, indirect addressing

Direct Communication

Case of msg passing

- 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

Case of msg passing

- Messages are directed to 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 (Cont.)

Case of msg passing

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:

  send(*A, message*) – send a message to mailbox *A*

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

Synchronization

Case of msg passing

- Message passing may be either **blocking** or **non-blocking**, also known as **synchronous** and **asynchronous**

- Types of **`send()`** and **`receive()`** primitives
  - **Blocking send** has the sender block until the message is received
  - **Non-blocking** send has the sender send the message and continue
  - **Blocking receive** has the receiver block until a message is available
  - **Non-blocking** receive has the receiver receive a valid message or null

Synchronization (Cont.)

Case of msg passing

- Different combinations possible
  - If both send and receive are blocking, we have a rendezvous (between the sender and the receiver)
- In the event of rendezvous, producer-consumer becomes trivial

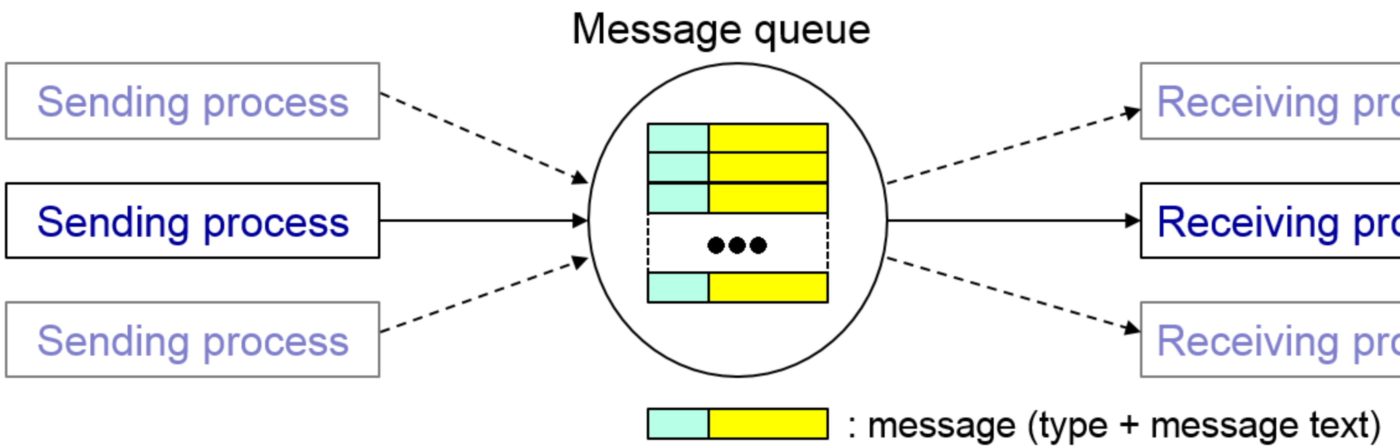
```
message next_produced;
while (true) {
     /* produce an item in next_produced */
   send(next_produced);
}

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

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

Buffering

Case of msg passing

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     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

There may be multiple senders and/or multiple receivers.



Examples of IPC Systems

- Example of IPC system – POSIX
- POSIX Shared Memory
  - Process first creates shared memory segment

```
shm_fd = shm_open(name, O CREAT |
                        O RDRW, 0666);
```

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

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

  - Now the process could write to the shared memory

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

IPC POSIX Producer

```
#include <stdio.h>
#include <stlib.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_RDRW, 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;
}
```

Shared memory

IPC POSIX Consumer

```
#include <stdio.h>
#include <stlib.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;
}
```

Shared memory

Examples of IPC Systems - Mach

- Mach communication is message-based
  - Even system calls are messages
  - Each task gets two mailboxes at creation - Kernel for kernel ↔task comm. and Notify for event notification
  - Only three system calls needed for message transfer

    `msg_send(), msg_receive(), msg_rpc()`
  - Mailboxes (called "port" in Mach) needed for communication, created via

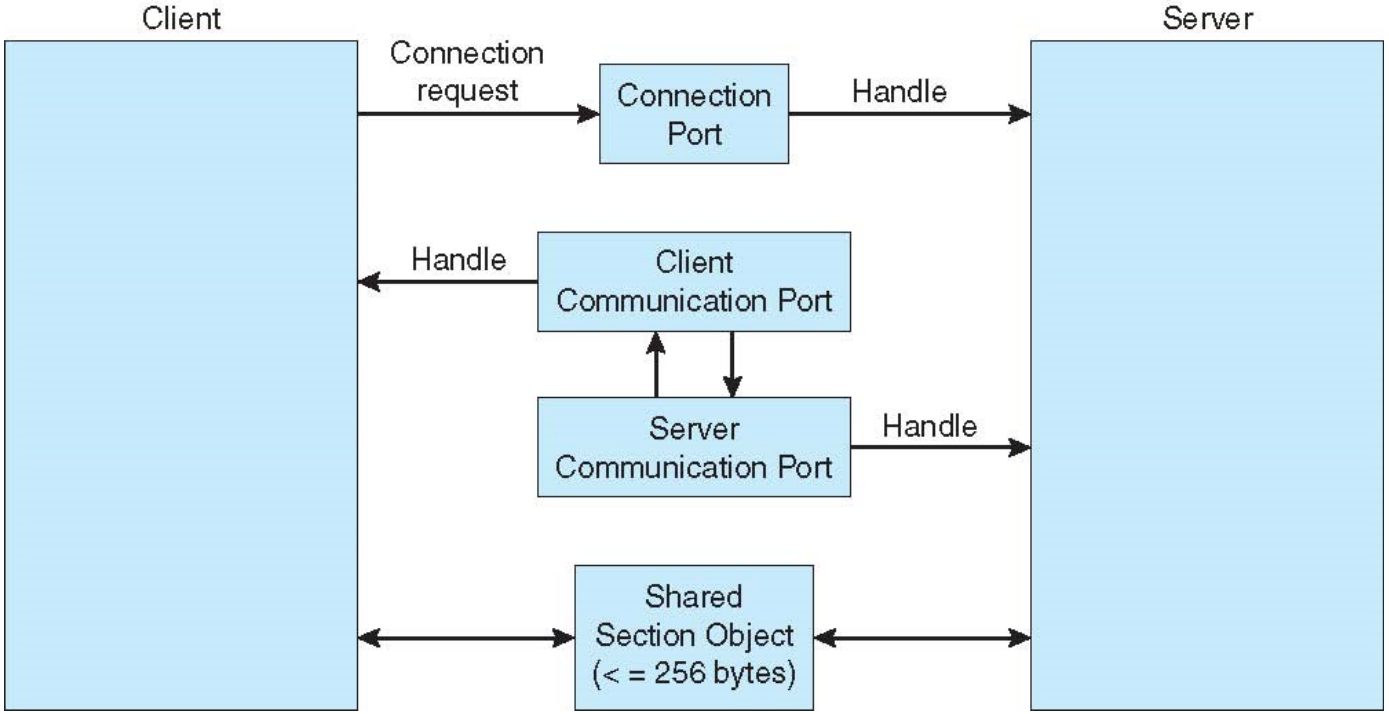
    `port_allocate()`
  - Send and receive are flexible, for example four options if mailbox full:
    - Wait indefinitely
    - Wait at most n milliseconds
    - Return immediately
    - Temporarily cache a message

Mach OS
- A microkernel developed at CMU as a replacement for the kernel in the BSD version of UNIX
- Provides the basis of the OS kernel in Mac OS X (*not* microkernel)

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
  - Communication works as follows:
    - 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 XP



Summary

- 프로세스에 대하여
  - 프로세스란 프로그램이 실행되는 동안의 모습을 말한다
  - 프로세스는 컴퓨터가 계산하는데 있어 중심역할을 수행하며, 온갖 상태 정보를 다 PCB에 유지한다
- 프로세스의 상태
  - CPU에서 실행될 때와 CPU 밖에서 실행을 기다리고 있을 때로 나뉜다
  - running, ready, wait states
- 실행의 궤적(쓰레드)?
  - 프로그램에서 명령이 실행되는 일련의 흐름 혹은 궤적을 쓰레드라 부름
  - 혹은 이렇게 실행을 주도하는 주체를 쓰레드라고도 함
- 스케줄러
  - 프로세스가 CPU 상에서 수행될 순서를 정하는 주체는 스케줄러이다.
  - 스케줄러는 각종 큐를 유지한다

- On the process
  - "Process" refers to the appearance of a program while in execution.
  - A process plays the central role in computing activity of a computer. It maintains state information in PCB.
- Process states
  - A process is in two different states: while executing on CPU and while waiting off CPU
  - running, ready, wait states
- Threads of execution?
  - Thread: a flow or locus of instructions executed in a program
  - Or, the entity that generates such a flow
- Scheduler
  - Ordering the processes to execute on CPU is performed by a scheduler.
  - Schedulers maintains various queues.

Summary (Cont.)

- 단기 스케줄러 혹은 디스팻쳐는 프로세스를 CPU에서 실제 실행되게 함
- 문맥교환(context switching)은 멀티프로그래밍 시스템에서 생기는 오버헤드
- 프로세스에 대한 연산
  - 프로세스의 생성(fork), 종료(exit), 실행(혹은 적재)(exec), 대기(wait), 등
- 프로세스간 통신(IPC)
  - 왜 필요한가? – 데이터/정보 공유 및 교환, 서비스의 요청/제공, 명령의 전달, 이벤트의 통지, 등
  - 공유기억(shared memory) 이용
  - 메시지 이용: 직접(link), 간접(mailbox)
    - 동기화: blocking(동기적), nonblocking(비동기적)
    - 버퍼링
- IPC 예
  - POSIX, Mach, Windows

- A short-term scheduler (dispatcher) lets a process run on CPU.
- Context switching is the overhead incurred in multiprogramming systems.
- Operation on processes
  - process creation (fork), termination (exit), execution or loading (exec), waiting (wait), etc.
- Interprocess communication (IPC)
  - why necessary? – sharing and exchange of data/information, service request/provision, delivery of commands, event notification, etc.
  - shared-memory approach
  - message-passing approach: direct(link), indirect(mailbox)
    - synchronization: blocking(synchronous), nonblocking(asynch)
    - buffering
- IPC examples
  - POSIX, Mach, Windows