Chapter 3: Processes-Concept

Chien Chin Chen

Department of Information Management
National Taiwan University

Outline

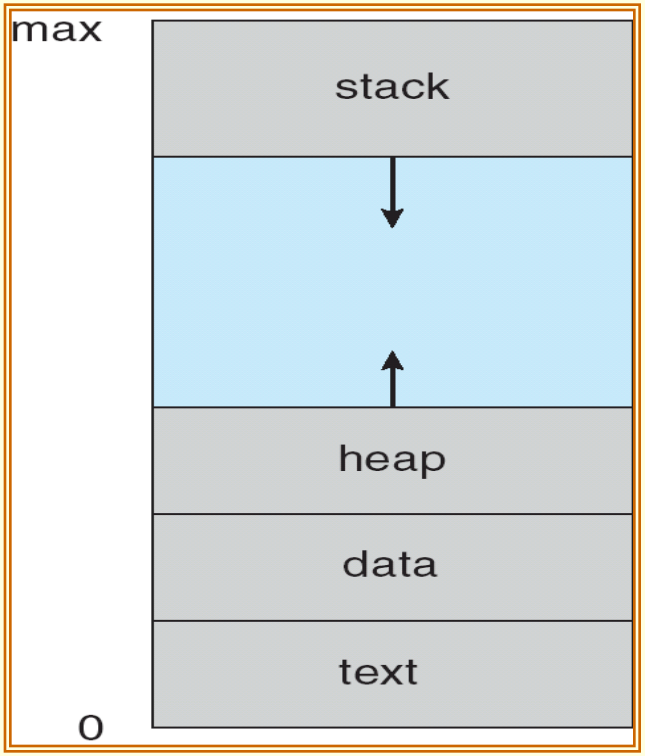
- Overview.
- Process Scheduling.
- Operations on Processes.
- Interprocess Communication.
- Examples of IPC Systems.
- Communication in Client-Server Systems.

Process Concept (1/3)

- The name of CPU activities varies in different systems:
  - Batch system – *jobs*.
  - Time-shared systems – *user programs* or *tasks*.
- Textbook uses the terms ***job*** and ***process*** almost interchangeably, but prefers *process*.
  - However, much of operating-system theory and terminology was developed during the batch era.
  - *Job* (such as job scheduling) would be used to avoid misunderstanding.

Process Concept (2/3)

- Process: 動態的
  - A program in execution.
  - Execution must progress in **sequential** fashion.
- A process in memory includes:
  - ***Text section***: the program code.
  - ***Stack***: contains temporary data (local variable, function parameters …).
  - ***Data section***: contains global variables.
  - ***Heap***: used for dynamical memory allocation.
  - ***Program counter***: the address of next instruction.
  - ***Register status***.



Process in memory

Process Concept (3/3)

- A program is not a process:
  - A program is a ***passive*** entity; a list of instruction stored on disk.
  - A process is an ***active*** entry.
    - A program becomes a process when an executable file is loaded into memory.
- Two (or more) processes may be associated with the same program.
  - For example, a user may invoke many copies of the web browser program.
  - Each of these is a separate process.
    - **While the text sections are equivalent, the data, heap, and stack sections vary.**

Process State (1/2)

- The ***state*** of a process is defined by the current activity of that process.
- Each process may be in one of the following states:
  - ***New***: the process is being created.
  - ***Running***: instructions are being executed.
  - ***Waiting***: the process is waiting for some event to occur.
    - Such as an I/O completion. 資料還沒來，等準備好才可以跑（所以輪不到他）
  - ***Ready***: the process is waiting to be assigned to a processor. 輪到你就可以跑
  - ***Terminated***: the process has finished execution.
- **Only one process can be running on any processor at any instance.**
  - Many processes may be *ready* and *waiting*.

Process State (2/2)

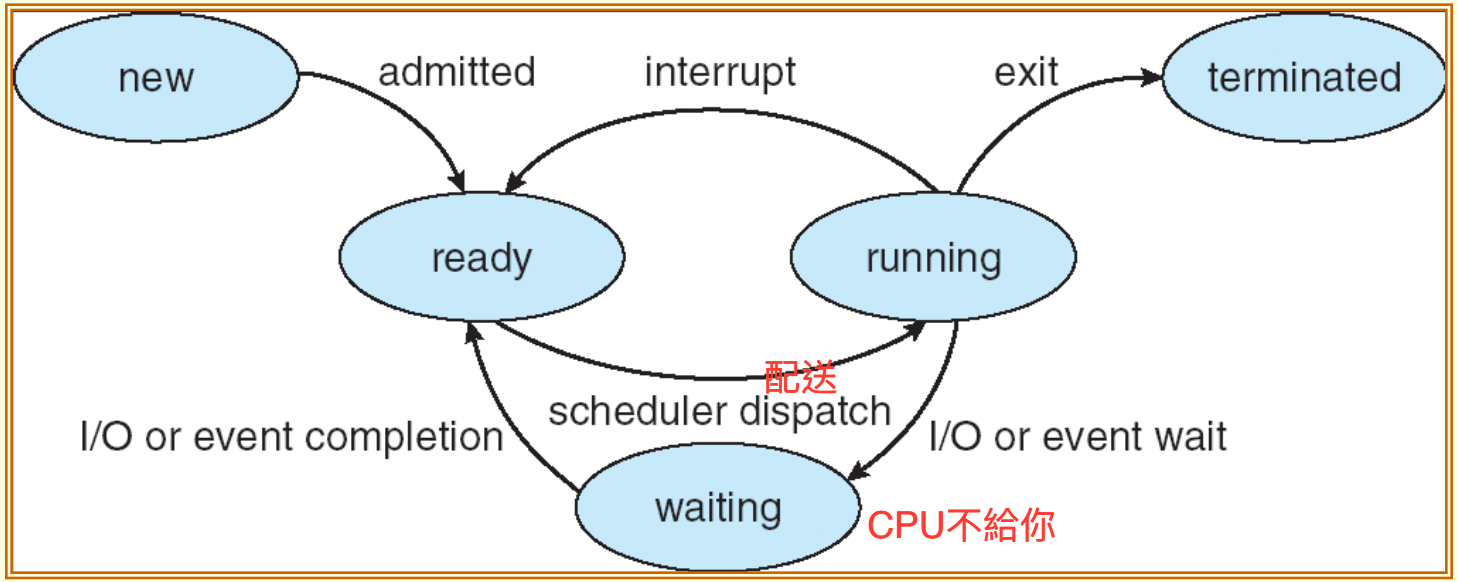
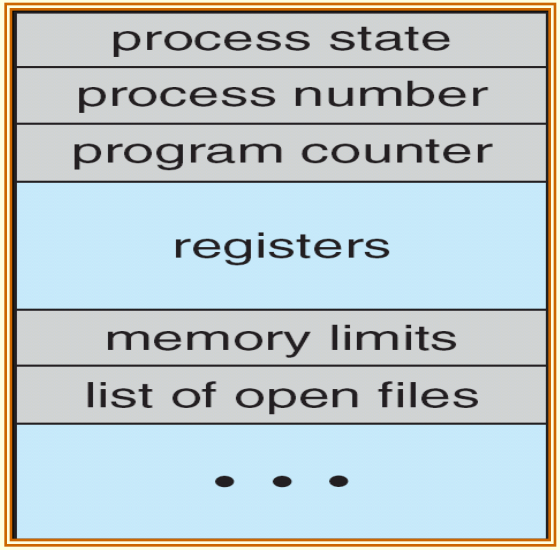


Diagram of Process State

Process Control Block

- ***Process control block*** (PCB): a representation of a process in the operating system.
- Information associated with each PCB:
  - **Process state**.
  - **Program counter**.
    - The address of next instruction.
  - **CPU registers**.
  - **CPU scheduling information**.
    - Process priority (chapter 5).
  - **Memory-management information** (chapter 8).
  - **Accounting information**.
    - E.g., amount of CPU time.
  - **I/O status information**.
    - Allocated devices, files, and so on.



Different processes have different PCB information.

Process Scheduling

- Multiprogramming:
  - To have some process running at all time.
  - To maximize CPU utilization.
- Time sharing:
  - To switch the CPU among processes so frequently.
  - Users can interact with each program while it is running.
- To do so … we need …
  - ***Process scheduler***:
    - Select an available process for execution on the CPU.
    - The rest have to wait until the CPU is free and can be rescheduled.

Scheduling Queues and Schedulers (1/8)

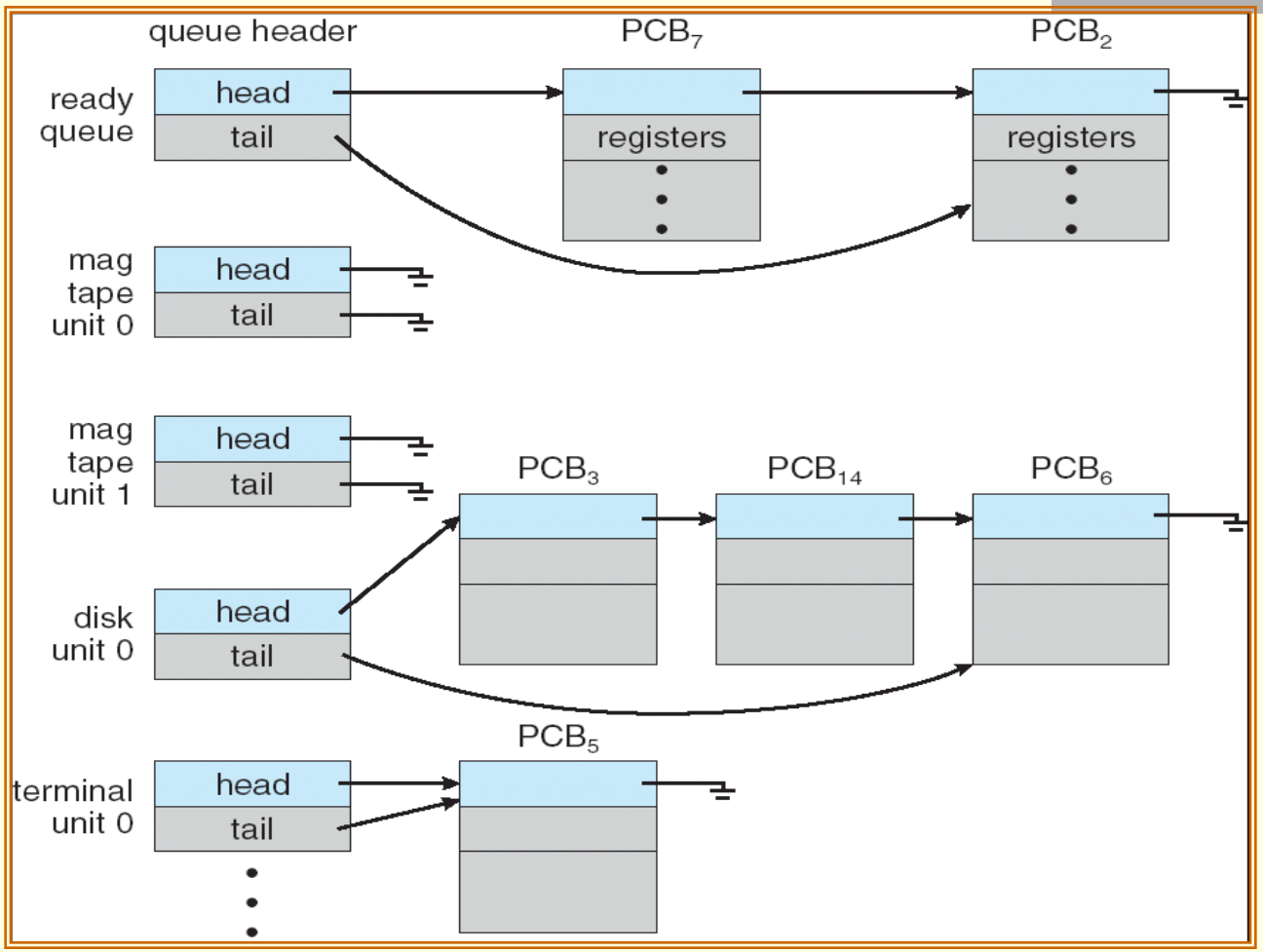
- The operating system usually contains a lot of **queues**.
  - Contain pointers to the first and final PCBs in the list (queue).
  - Each PCB includes a pointer field that points to the next PCB in the queue.
  - ***Job queue***:
    - Processes are initially spooled to a mass-storage device (e.g., a disk), where they are kept for later execution.
    - Job scheduler later selects processes from this pool and load them into memory for execution.
    - Often in batch system. **On some operating systems, job queue and job scheduler may be absent**.
      - Such as UNIX and MS Windows.
      - Just put every new process in memory for execution.

Scheduling Queues and Schedulers (2/8)

- ***Ready queue***:
  - Contain the processes residing in main memory.
  - Those processes are **ready** and waiting to execute.

- Operating systems also include other queues.
  - ***Device queue***:
    - A list of processes waiting for a particular I/O device.
      - For instance, there are many processes make requests to a disk.

Scheduling Queues and Schedulers (3/8)

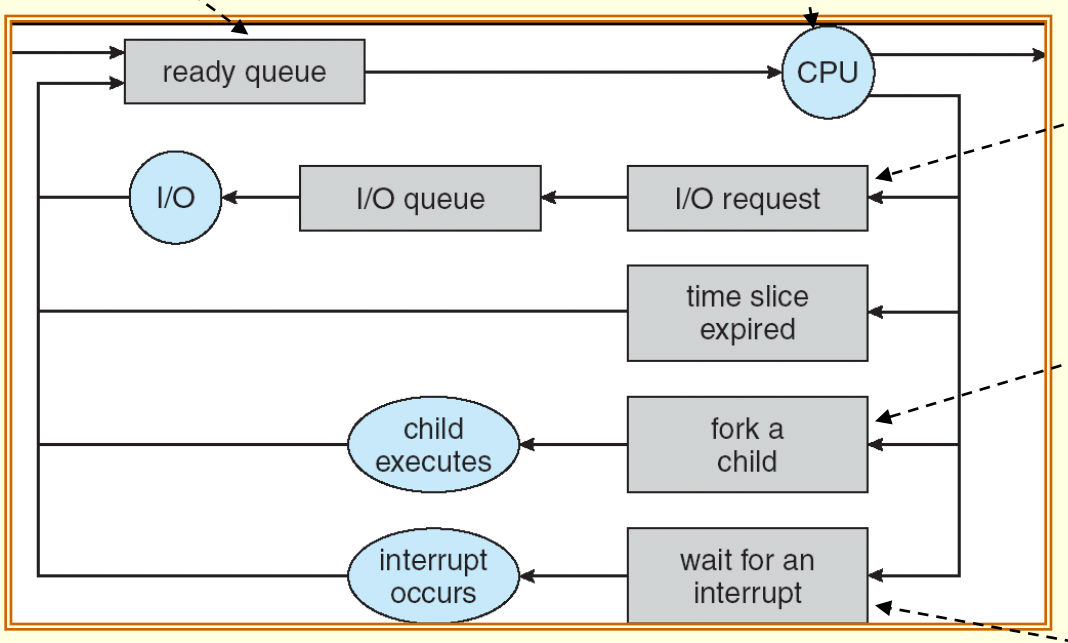


Ready Queue And Various I/O Device Queues

Scheduling Queues and Schedulers (4/8)

A new process is initially put in the ready queue (**ready** state).

Eventually, it will be selected (dispatched) for execution (**running** state).



The process may issue an I/O request, and switches to the **waiting** state.

The process may create a new sub-process and wait for the sub-process's termination (**waiting** state).

The process may be removed from the CPU by an interrupt, and be put back in the ready queue (**ready** state).

Processes migrate among the various queues.

When the process terminates, it is removed from all Queues and has its PCB and resources deallocated.

Scheduling Queues and Schedulers (5/8)

- A process migrates among the various scheduling queues throughout its lifetime.
  - The operating system must select process from these queues in some fashion.
- Two main schedulers:
  - ***Job scheduler*** (or **long-term** scheduler) – selects which processes should be brought into the ready queue.
  - ***CPU scheduler*** (or **short-term** scheduler) – selects which process should be executed next and allocates CPU. 要快
- The primary difference between the schedulers: **frequency of execution**.
  - Often, the short-term scheduler executes at leas once every 100 ms.
  - The long-term scheduler executes much less frequently.

Scheduling Queues and Schedulers (6/8)

- The short-term scheduler must be **fast**.
  - If it takes 10 ms to decide to execute a process for 100 ms, then 10 / (10 + 100) = 9% of the CPU is being wasted for scheduling the work.
- However, because of the longer interval between executions, the long-term scheduler can afford to take more time to decision.

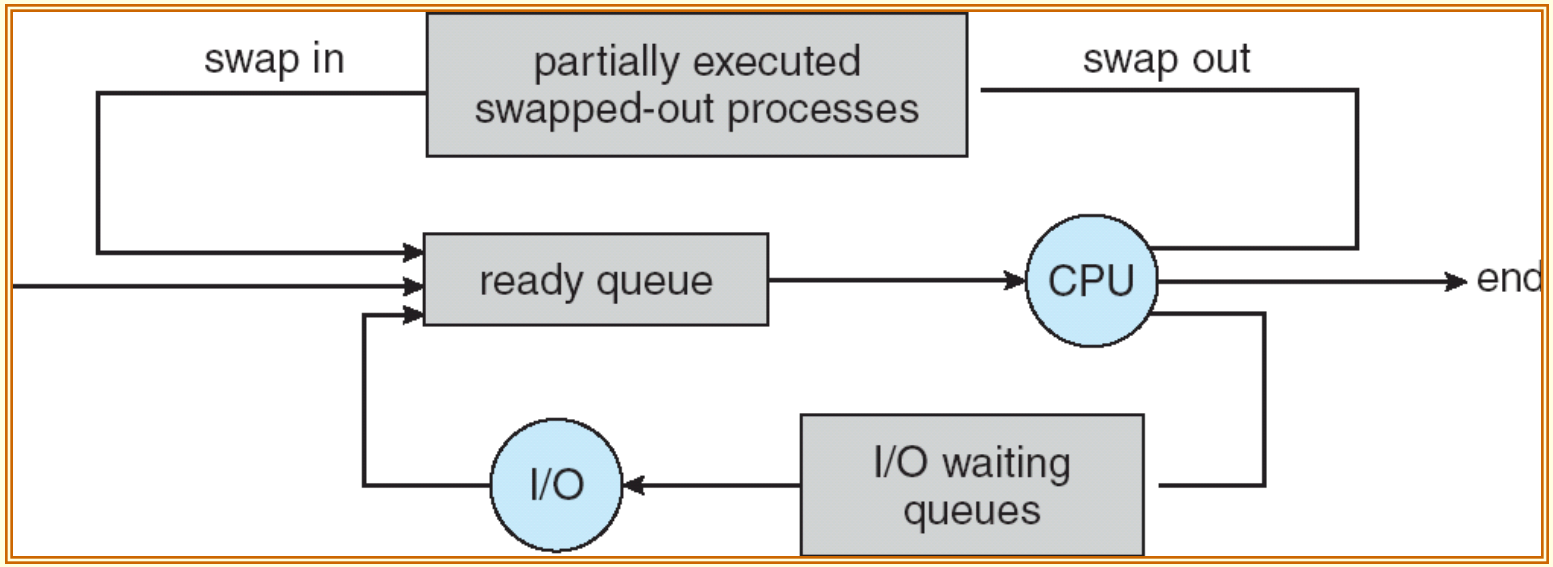
  久久叫一次
- The long-term scheduler controls the degree of multiprogramming.
- In general, processes can be described as either ***I/O bound*** or ***CPU bound***.
  - A **I/O-bound process** spends more of its time doing I/O.
  - A **CPU-bound process** uses more of its time doing computations.
  - A good long-term scheduler should select a good process **mix** of I/O-bound and CPU-bound processes that the system will be balanced.

Scheduling Queues and Schedulers (7/8)

- For systems with no long-term schedulers:
  - The stability of the systems may depend on the self-adjusting nature of human users.
    - If the performance declines to unacceptable levels on a multi-users system, some users will simply quit.

- Some operating systems may introduce a ***medium-term scheduler***. 老師沒講 O__O
  - It **removes** processes from memory and thus reduce the degree of multi-programming (**swap out**).
  - Later, the processes can be **reintroduced** into memory for execution (**swap in**).
  - Swapping may improve the process mix.
    - Or for better memory management.

Scheduling Queues and Schedulers (8/8)

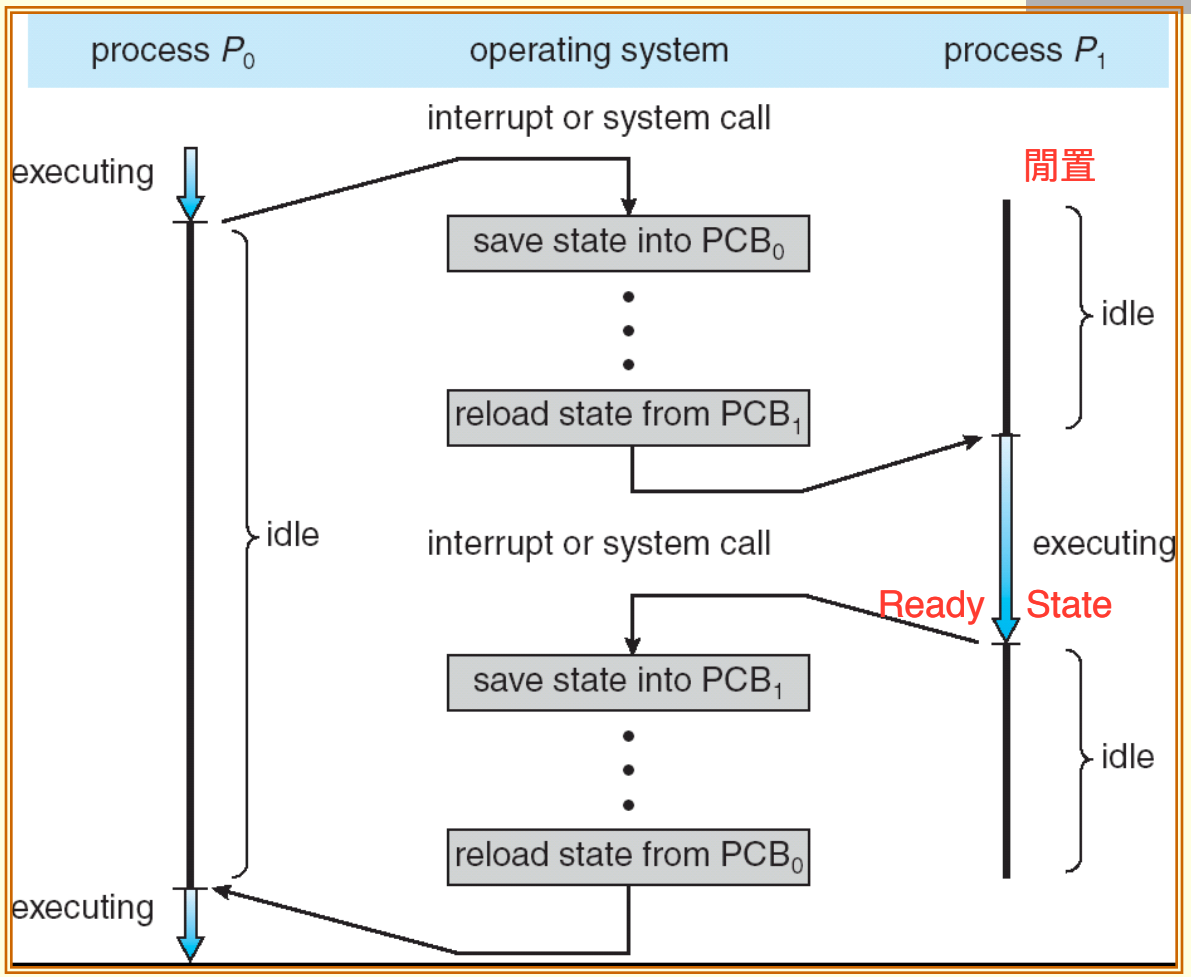


Addition of medium-term scheduling to the queueing diagram.

Context Switch (1/3)

- When CPU switches to another process, the system must **save** the **context** of the old process and **load** the saved **context** for the new process.
  - E.g., interrupts cause the operating system to change a CPU from its current task to run a kernel routine.
- Such a switching is known as a ***context switch***.
- The context is represented in the PCB of the process.
  - The value of the CPU registers. 因為運算都是做在register上
  - The process state.
  - Program counter.
  - …

Context Switch (2/3)



Context Switch (3/3)

- **Context-switch time is overhead**; the system does no useful work while switching.

- Dependent on hardware support.
  - Some processors provide multiple sets of registers.
    - A context switch simply requires changing the pointer to the current register set.

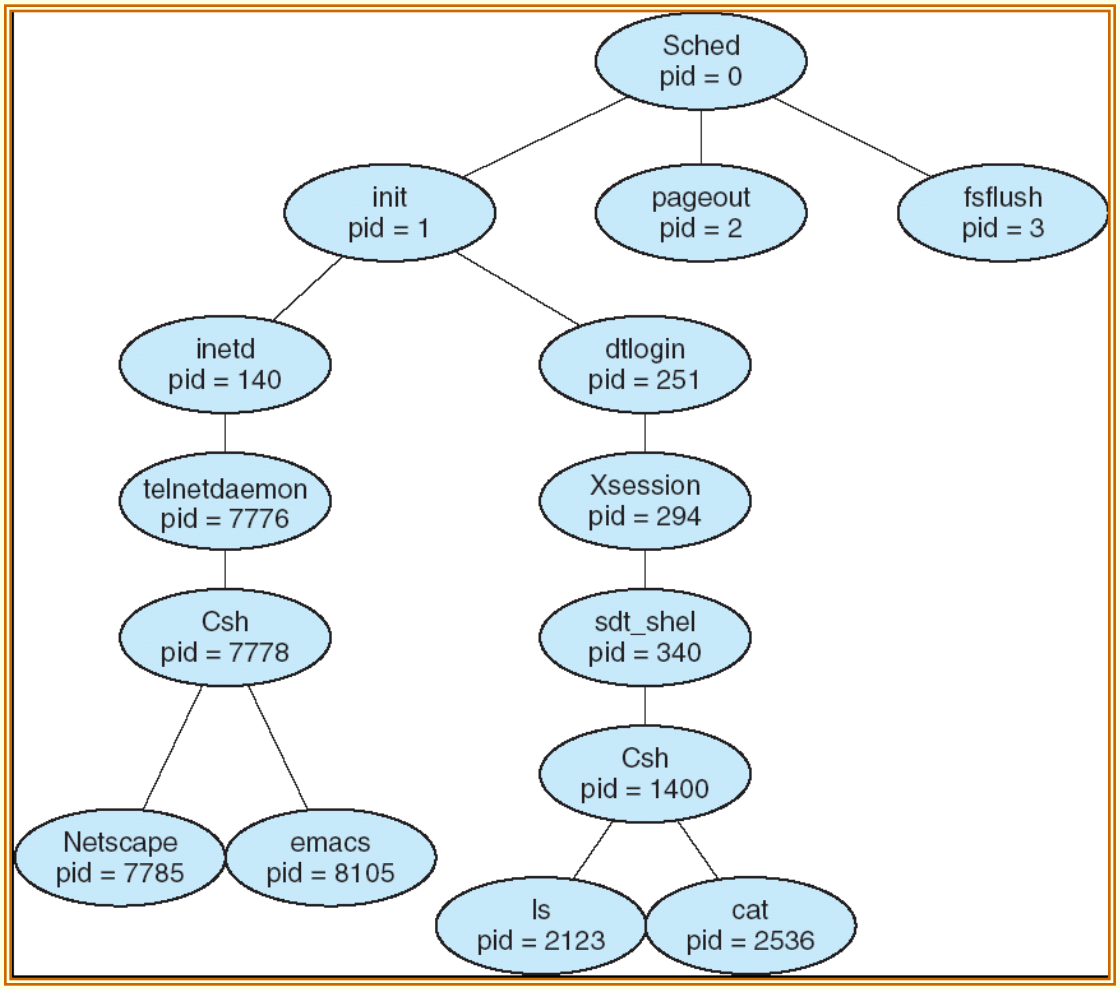
Operations on Processes

Process Creation (1/8)

- A process may create several new processes, via a create-process system call.
- The creating process is called a **parent** process, and the new processes are called the **children** of that process.
  - New processes may in turn create other processes, forming a **tree** of processes.
- Most operating system (UNIX and MS Windows) identify processes according to a unique process identifier (pid).
  - Usually an integer number.

Process ID

Process Creation (2/8)



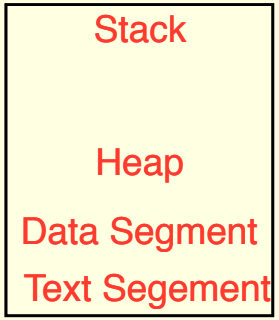
A tree of processes on a typical Solaris

Process Creation (3/8)

- On UNIX, the `ps` command can be used to obtain the information of all process.

Process Creation (4/8)

- When parent process holds certain resources (files, I/O devices).
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.
- When a process creates a new process, two possibilities exist in terms of execution:
  - Parent and children execute concurrently. 用搶的 自己的程序自己搶
  - Parent waits until children terminate. 通常是這個，因為叫parent先跑沒用厂厂
- There are also two possibilities in terms of the address space of the new process:
  - Child duplicate of parent.
  - Child has **a program loaded into it**.



Process Creation (5/8)

- UNIX examples: API function map 到一個system call
  - `fork()` system call creates new process (POSIX).
  - The new process consists of **a copy of the address space of the original process**.
  - Both process **continue (concurrently) execution** at the instruction after the `fork()`.
  - But … how to distinguish?
    - The return code for the `fork()` is zero for the new (child) process.
    - The process identifier (PID) of the child is returned to the parent.

Process Creation (6/8)

fork_test.c

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

int main()
{
    pid_t  pid;
    pid = fork(); /* fork another process */
    if (pid < 0) { /* error occurred */
          printf("Fork Failed\n");
          exit(-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\n");
          exit(0);
    }
}
```

就是int

Step1.執行fork();
Step 2.回傳值給pid

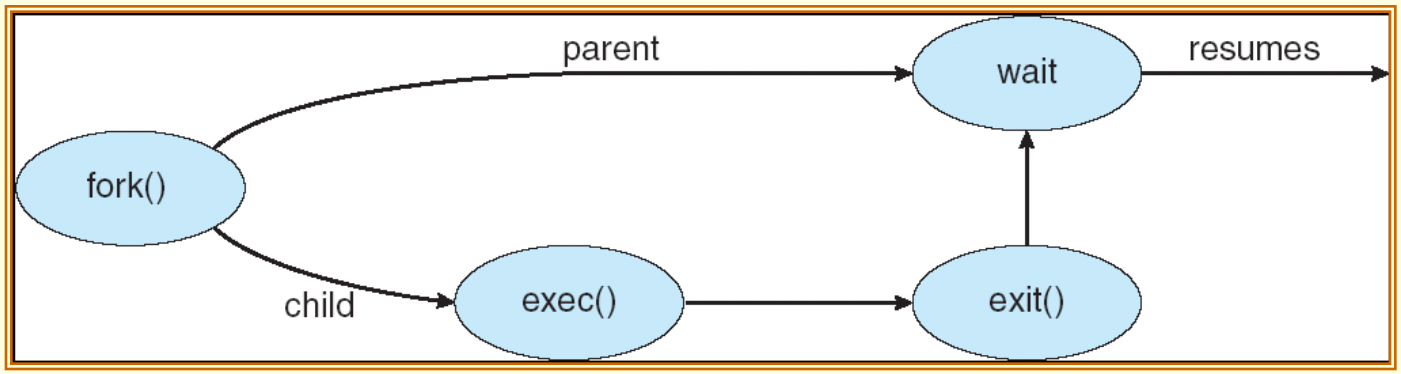
child要做的事情

回傳值>0 parent
回傳值=0 child

Process Creation (7/8)

- Typically, the **`exec()`** (or its family, such as **`execlp()`**) system call (POSIX) is used after a `fork()` by **one of** the two processes to **replace the process's memory space with a new program**.
  - For example, the child process overlays its address space with UNIX command `/bin/ls` using the `execlp()` system call.
- The parent can wait for the child process to complete with the **`wait()`** system call (POSIX).
  - When the child process completes, the parent process resumes from the call to `wait`.
- **`exit()`** system call: cause normal process termination (POSIX).

Process Creation (8/8)




```
paton@mercury:~/test/fork_test [80x24]
連線(C)  編輯(E)  檢視(V)  視窗(W)  選項(O)  說明(H)
[paton@mercury fork_test]$ gcc -o fork_test fork_test.c
[paton@mercury fork_test]$ ./fork_test
fork_test  fork_test.c
Child Complete
[paton@mercury fork_test]$
```

Process Termination

- Process executes last statement and asks the operating system to delete it by using the **`exit()`** system call.
  - Can return a status value (typically an integer) to its parent process (via the **`wait()`** system call).
  - Process' resources are deallocated by operating system.

- Parent may terminate execution of children processes (**`kill()`**, **POSIX**) when:
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - If parent is exiting.
    - Some operating system do not allow child to continue if its parent terminates.
    - All children terminated - ***cascading termination***.

Interprocess Communication

Cooperating Processes (1/4)

- Processes may be **independent** processes or **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.

- Reasons for process cooperation:
  - Information sharing.
  - Computation speed-up.
    - Can be achieved only if the system have multiple CPUs.
  - Modularity.
  - Convenience.
    - Users may work on many tasks (or sub-tasks) at the same time.

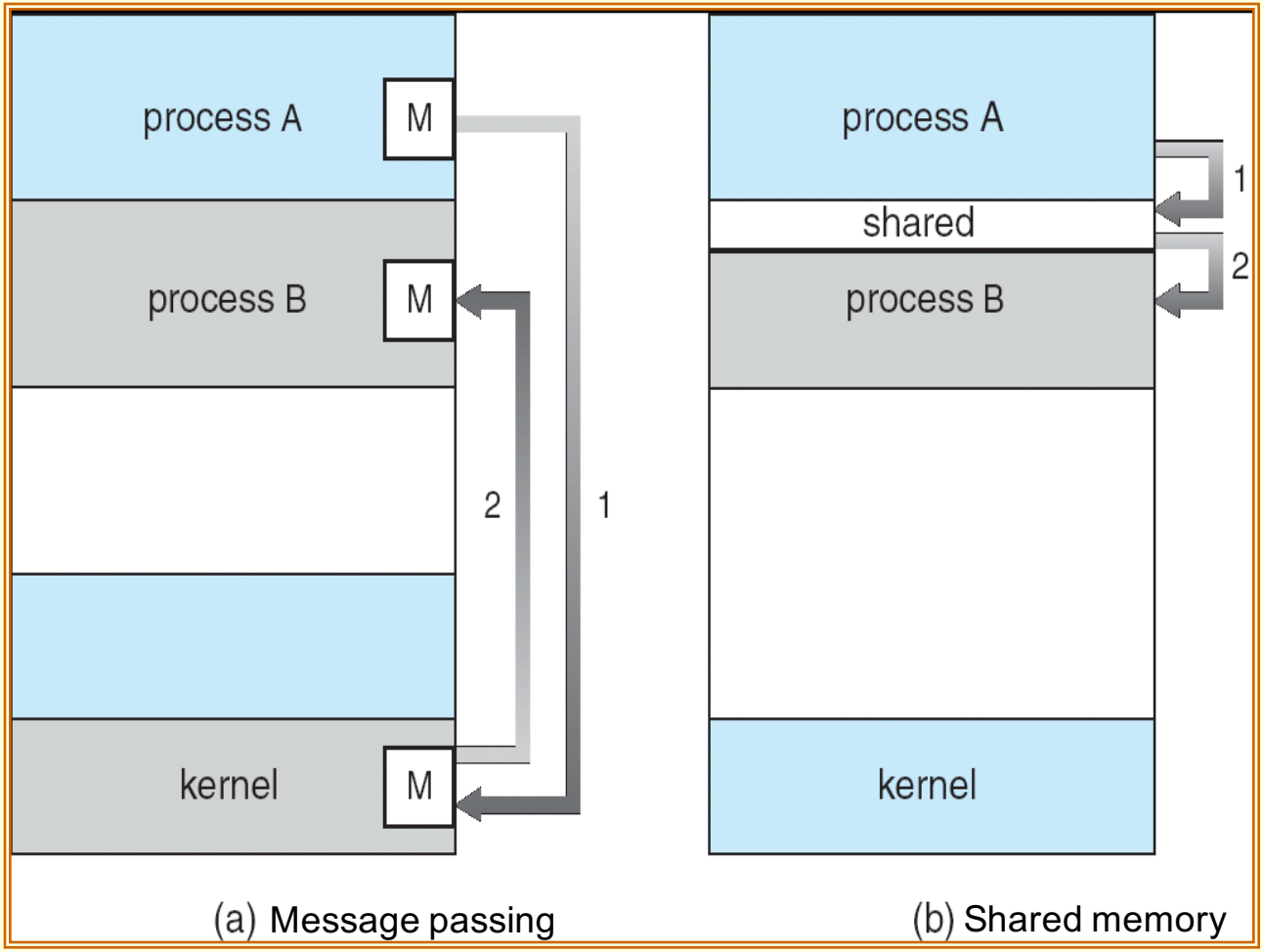
Cooperating Processes (2/4)

- ***Interprocess communication*** (IPC) is a mechanism that allows cooperating processes to exchange data and information.
- Two typical models of IPC:
  - ***Shared memory***.
    - A region of memory is shared by cooperating processes.
    - Processes can exchange information by reading and writing data to the region.
  - ***Message passing***.
    - Communication takes place by means of messages exchanged between the processes.
  - Both of the models are common and implemented in many operating systems.

Cooperating Processes (3/4)

- Message passing is useful for exchanging **smaller amounts of data**.
  - Because **no conflicts** need be avoided.
  - Usually **slow**, because message-passing systems are generally implemented as system calls.
    - Require the more time-consuming task of kernel intervention.
- Shared memory allows **maximum speed** and convenience of communication.
  - It can be done at memory speeds when within a computer.
  - Is **faster** than message passing.
    - Once shared memory is established, all accesses are treated as routine memory access.
    - No assistance from the kernel is required.

Cooperating Processes (4/4)



Shared-Memory Systems (1/4)

- A process creates a shared-memory region residing in its address space.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space. 誰都可以讀，誰都可以寫—>inconsistent
- Then, they can exchange information by reading and writing data in the shared area.
- The processes are responsible for ensuring that they are not writing to the same location simultaneously.
  - Chapter 6 will discuss synchronization to synchronize concurrent accesses.

Shared-Memory Systems (2/4)

- Shared memory as a ***producer-consumer problem***:
  - **producer** process produces information that is consumed by a **consumer** process.
  - A **buffer** (shared memory) can be filled by the producer and emptied by the consumer.
- Two types of buffers:
  - *unbounded-buffer* places no practical limit on the size of the buffer.
    - The consumer may have to wait for new items.
    - But the producer can always produce new items.
  - *bounded-buffer* assumes that there is a fixed buffer size.
    - The consumer may have to wait for new items.
    - The producer have to wait if the buffer is full.

producer自由自在地放，consumer就是等producer生

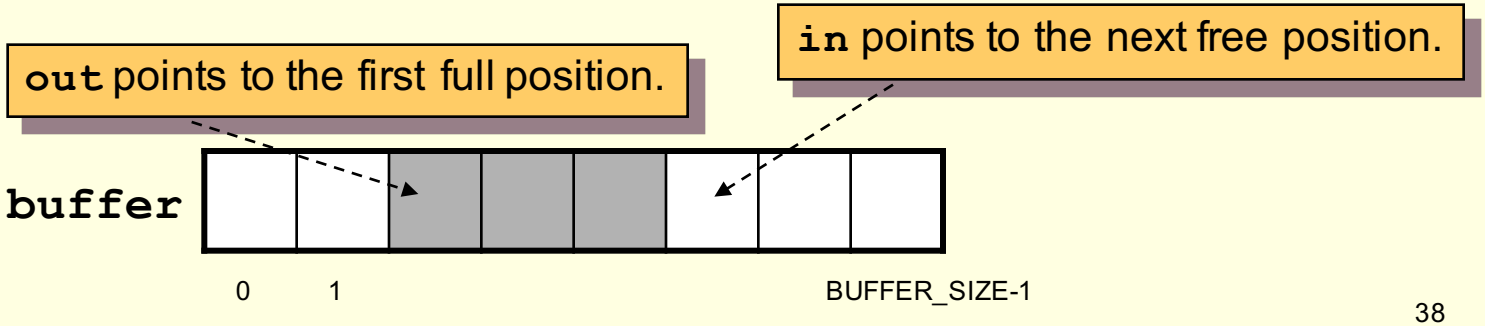
producer只有buffer滿要等，consumer就是等producer生（buffer是空的時候等）

Shared-Memory Systems (3/4)

- Implementation:

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

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

- The buffer is implemented as a **circular array**:
  - `in=(in++)%BUFFER_SIZE`
  - `out=(out++)%BUFFER_SIZE`
- At most `BUFFER_SIZE-1` items in the buffer at the same time.
- Empty: `in == out`
- Full: `((in+1)%BUFFER_SIZE) == out`



Shared-Memory Systems (4/4)

- The producer process:

```
item nextProduced;

while (true) {
     /* Produce an item */
     while (((in + 1) % BUFFER_SIZE)  == out)
               ;    // do nothing
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER SIZE;
}
```

一直等，鬼打牆 Ghost hit wall

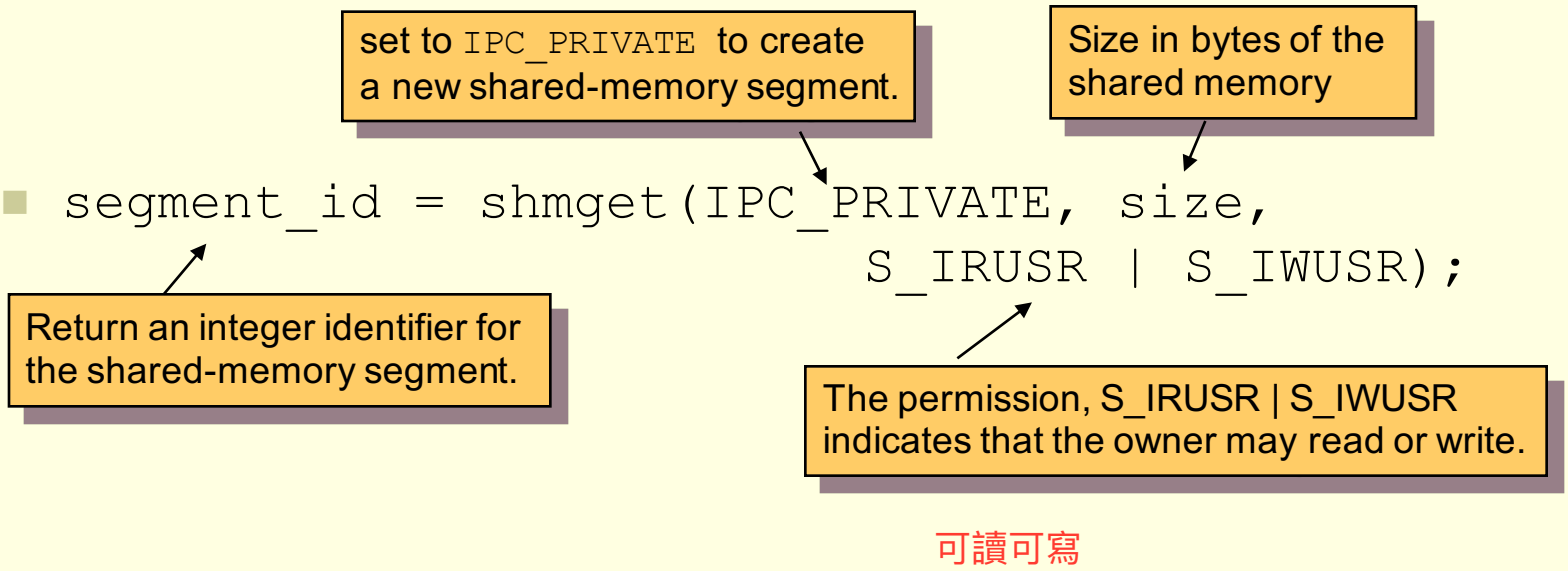
- The consumer process

```
item nextConsumed

while (true) {
     while (in == out)
               ;    /* do nothing
     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER SIZE;
}
```

空的，鬼打牆

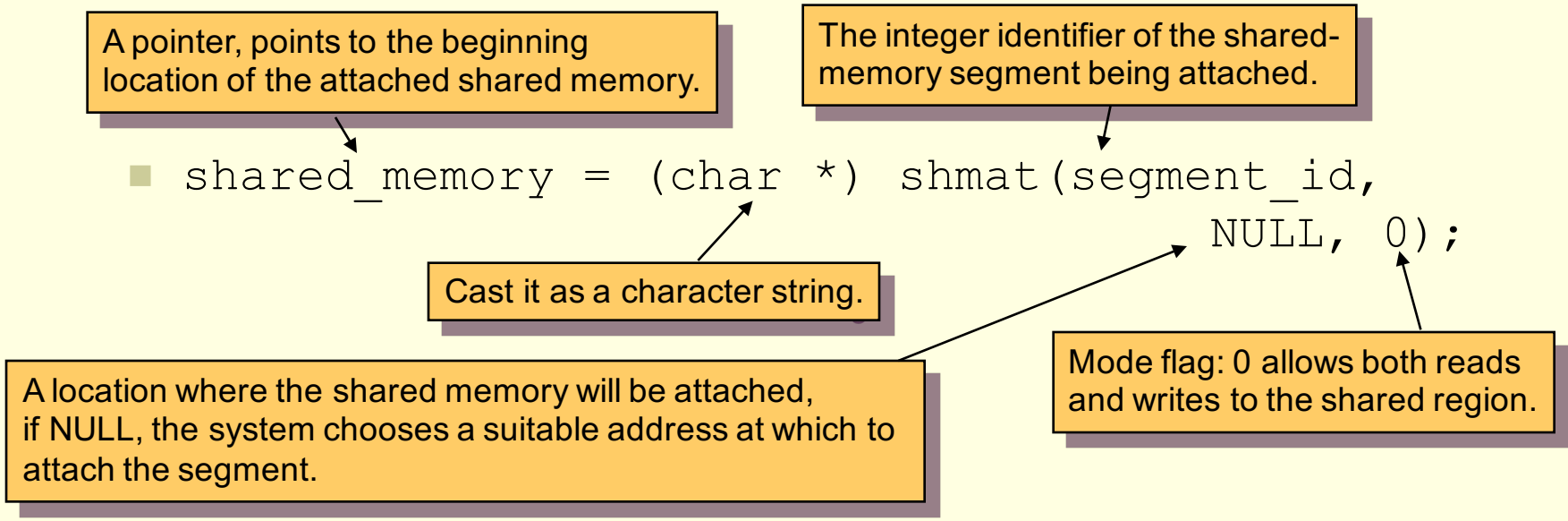
POSIX Shared Memory Example (1/5)

- A process must **first create** a shared memory segment using the `shmget()` system call (Shared Memory GET).



POSIX Shared Memory Example (2/5)

- Processes that wish to access a shared-memory segment must **attach** it to their address space using the `shmat()` (Shared Memory Attach) system call.



POSIX Shared Memory Example (3/5)

- Then, the process can access the shared memory as a routine memory access using the pointer returned from `shmat()`.
  - `sprintf(shared_memory, "hello");`
- Other processes sharing this segment would see the updates to the shared memory segment.

POSIX Shared Memory Example (4/5)

- When a process no longer requires access to the shared-memory segment, it **detaches** the segment from its address space.
  - `shmdt(shared_memory);`

Pass the pointer of the shared-memory region to detach.

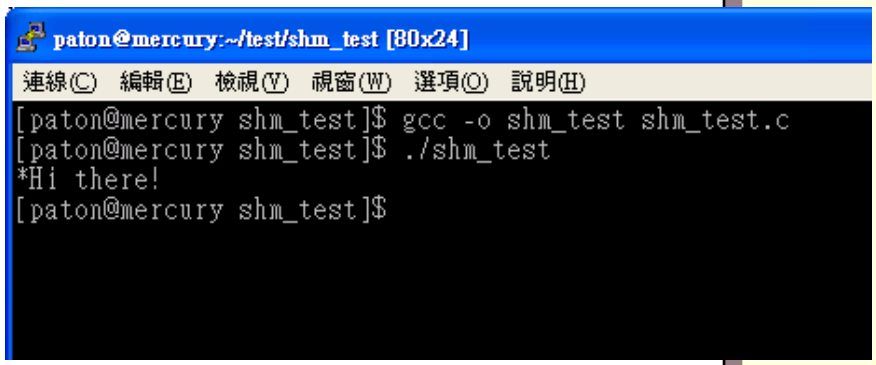
- Finally, a shared-memory segment can be removed from the system with the `shmctl()` system call.
  - `Shmctl(segment_id, IPC_RMID, NULL);`

The integer identifier of the shared-memory segment.

Destroy the shared segment

POSIX Shared Memory Example (5/5)

```
#include <stdio.h>              // shm_test.c
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
   /* the identifier for the shared memory segment */
   int segment_id;
   /* a pointer to the shared memory segment */
   char* shared_memory;
   /* the size (in bytes) of the shared memory segment */
   const int segment_size = 4096;

   /* allocate  a shared memory segment */
   segment_id = shmget(IPC_PRIVATE, segment_size, S_IRUSR | S_IWUSR);

   /* attach the shared memory segment */
   shared_memory = (char *) shmat(segment_id, NULL, 0);

   /* write a message to the shared memory segment   */
   sprintf(shared_memory, "Hi there!");

   /* now print out the string from shared memory */
   printf("*%s\n", shared_memory);

   /* now detach the shared memory segment */
   shmdt(shared_memory);

   /* now remove the shared memory segment */
   shmctl(segment_id, IPC_RMID, NULL);

   return 0;
}
```



Fork & Shared Memory (1/6)

- The parent process first creates a shared memory segment.
  - It then attaches the shared memory segment to its address space.
- Next, it creates two child processes.
  - Each process performs a certain computation and then writes its output to the shared memory.
- Finally, the parent process summarizes the outputs of the child processes.

Fork & Shared Memory (2/6)

```
#include <sys/types.h>
#include <stdio.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main()
{
  // create share memory
  int segment_id;
  const int size = 4096;
  char *shared_memory;

  segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
  // attach the shared memory segment
  shared_memory = (char *) shmat(segment_id,NULL,0);
  memset(shared_memory,0,size); initialization，都設成0
```

Fork & Shared Memory (3/6)

```
// create the first process
pid_t pid;
pid = fork();

if (pid < 0) {
   fprintf(stderr, "Fork Failed");
   exit(-1);
} else if (pid == 0) {

   // write a message to the segment
   sprintf(shared_memory,"message from P1");
   // detach the segment then exit
   shmdt(shared_memory);

   exit(0);
```

because the child process is a copy of the parent, the shared memory region will be attached to the child's space as well.

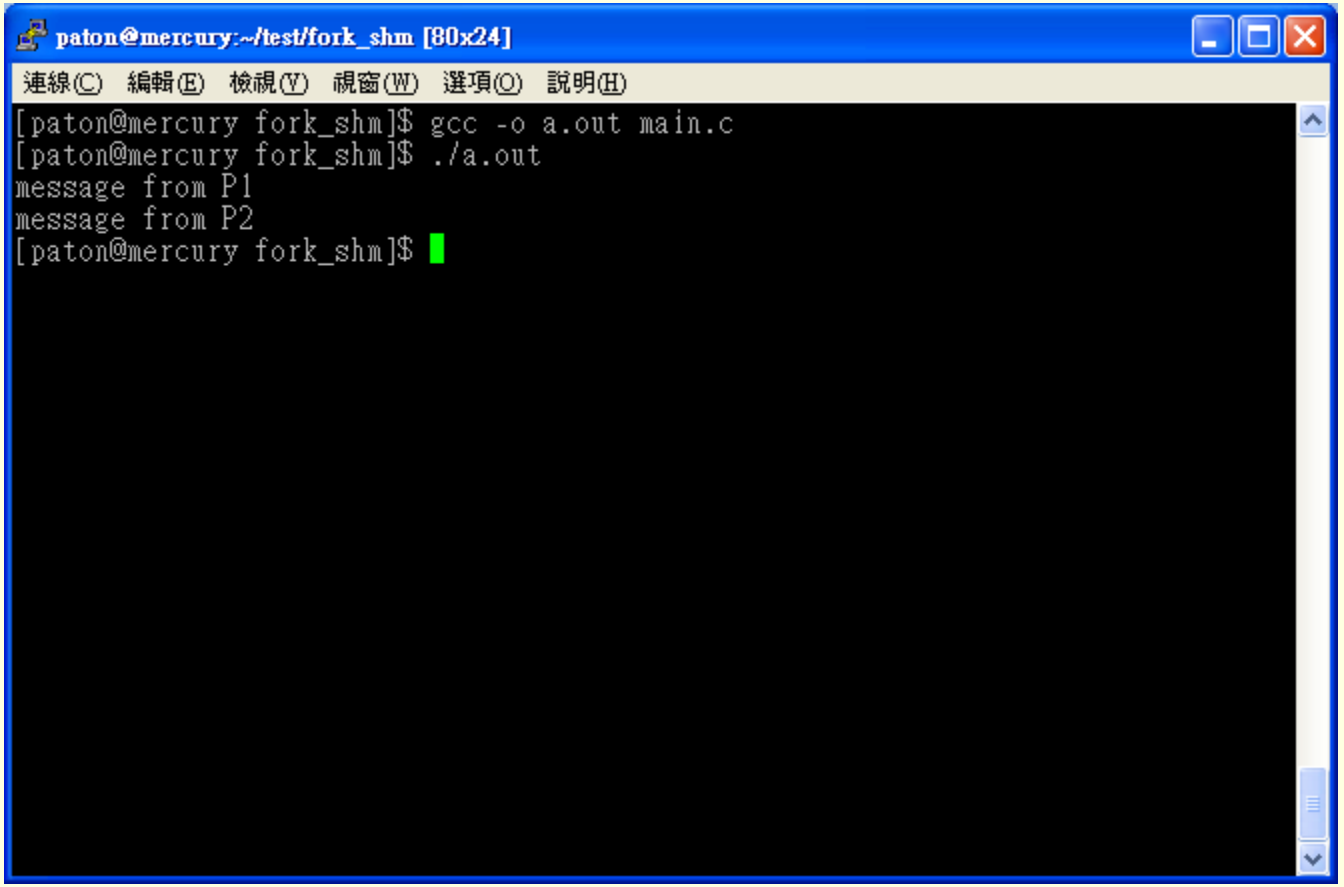
Fork & Shared Memory (4/6)

```
} else {

   // create the second process
   pid = fork();
   if(pid < 0) {
     fprintf(stderr, "Fork another process failed");
     exit(-1);
   } else if (pid == 0) {

     // write a message to the segment
     sprintf(shared_memory+1024,"message from P2");
     // detach the segment then exit
     shmdt(shared_memory);
     exit(0);
```

Fork & Shared Memory (5/6)

```
  } else {

      int count = 2;
      while (count > 0) {
        wait(NULL);
        count--;
      }

      // read messages posted by child processes
      printf("%s\n",shared_memory);
      printf("%s\n",shared_memory+1024);

      shmdt(shared_memory);
      // parent removes the shared memory segment
      shmctl(segment_id,IPC_RMID,NULL);

      exit(0);
    }
  }
}
```

Fork & Shared Memory (6/6)



Message-Passing Systems (1/7)

- Processes communicate with each other **without** resorting to **shared variables**.
- The communication actions are synchronized.
- Useful in a distributed (network) environment.

- A message-passing facility provides at least two operations:
  - ***send(message)***
  - ***receive(message)***
  - message size can be **fixed** or **variable**.
    - **fixed-sized messages** are easy to implement, but makes the programming task difficult.
    - **Variable-sized messages** require a more complex system-level implementation, but the programming task becomes simpler.

Message-Passing Systems (2/7)

- Processes that want to communicate must have a way to refer to each other.
- ***Direct-symmetry-communication***: 對稱
  - processes that want 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`.
- ***Direct-asymmetry-communication***: 非對稱
  - Only the sender names the recipient.
  - `send(P, message)` – send a message to process `P`.
  - `receive(id, message)` – receive a message from **any** process; the variable `id` is set to the name of the sender.

Message-Passing Systems (3/7)

- The disadvantage in both of these ***hard-coding*** schemes (symmetric and asymmetric).
  - Changing the identifier of a process may necessitate examining all other processes.
  - All references to the old identifier must be modified to the new identifier.

- ***Indirect communication***:
  - The messages are sent to and received from ***mailboxes***, or ***ports***.
  - Each mailbox has a unique identification.
  - A process can communicate with some other process via **a number of** different mailboxes.
  - A mailbox may be owned either by a process of by the operating system.

  - `send(A, message)` – send a message to mailbox `A`.
  - `receive(A, message)` – receive a message from mailbox `A`.

Message-Passing Systems (4/7)

- In indirect communication scheme, a communication link has the following properties:
  - A link is established between a pair of processes only if both members of the pair have a shared mailbox.
  - A link may associated with more than two processes.
  - Between each pair of communicating processes, there may be a number of different links.

很多process

Message-Passing Systems (5/7)

- A practical problem of 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. Sender is notified who the receiver was.

Message-Passing Systems (6/7)

- There are different design options for implementing the `send()` and `receive()` primitives.
  - **Blocking** (***synchronous***) **send**: the sending process is blocked until the message is received by the receiving process or by the mail box.
  - **Nonblocking** (***asynchronous***) **send**: the sending process sends the message and resumes operation.
  - **Blocking receive**: the receiver blocks until a message is available.
  - **Nonblocking receive**: the receiver retrieves either a valid message or a null.
  - Different combinations of `send()` and `receive()` are possible.
    - E.g., **rendezvous**: `send()` and `receive()` are blocking, a trivial solution to the producer-consumer problem.

Message-Passing Systems (7/7)

- ***Buffering***:
  - Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary **queue**.
  - Queue can be:
    - **Zero capacity**: can not have any messages waiting in it.
      - The sender must block until the recipient receives the message.

no buffer

    - **Bounded capacity**: the queue has finite length *n*.
      - If the queue is full, the sender must block until space is available in the queue.

limit size

    - **Unbounded capacity**: the sender never blocks.

Message-Passing Example: Mach (1/4)

- Most communication in Mach is carried out by messages.
- Messages are sent to and received from mailboxes called ***ports*** in Mach.
- The `port_allocate()` system call creates a new mailbox and allocates space for its queue of messages.
- The task (process) that creates the mailbox is the owner of the mailbox.
  - Only one task (the owner) at a time can receive from a mailbox, but the right can be sent to other tasks (processes) if desired.

Message-Passing Example: Mach (2/4)

- The messages consist of a **fixed-length header** followed by a **variable-length data portion**.
- **Header** includes:
  - The length of the message.
  - **Two** mailbox names.
    - The mailbox to which the message is being sent.
    - Commonly, we expect a reply, so the mailbox of the sender is passed on to the receiving task, as a “return address”.
- **The variable part** includes:
  - a list of typed data items.
    - Each entry in the list has a type, size, and value.

Message-Passing Example: Mach (3/4)

- When sending a message to a **non-full** mailbox:
  - The message is **copied** to the mailbox, and the sending task continues.
  - **Double-copy** is the major problem of the message systems because the message is first copied from the sender to the mailbox and then from the mailbox to the receiver.
- When sending a message to a **full** mailbox, the sending task has four options:
  - Wait indefinitely until there is room in the mailbox.
  - Wait at most *n* milliseconds.
  - Do not wait at all but rather return immediately.
  - Temporarily cache a message.

Message-Passing Example: Mach (4/4)

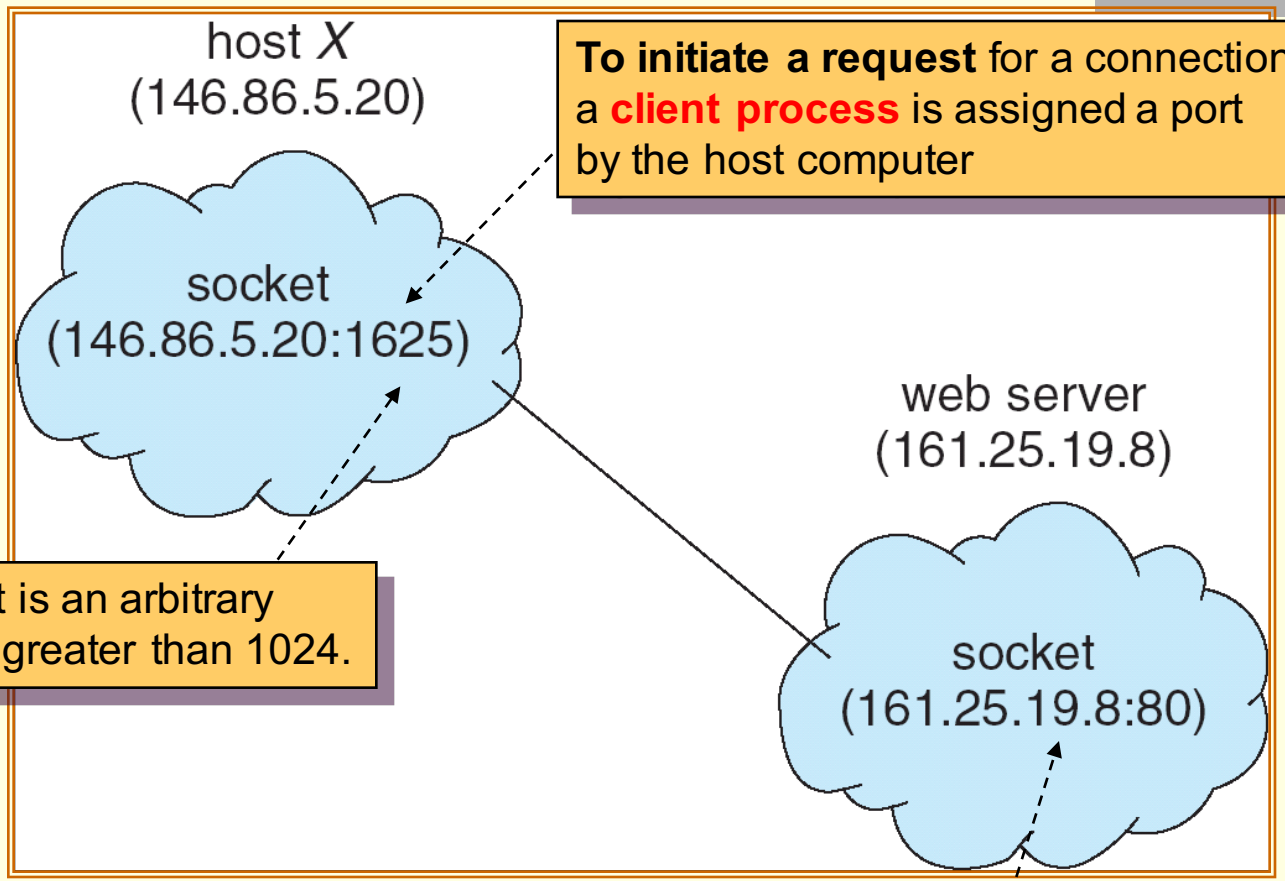
- The receive operation must specify the mailbox or mailbox set from which a message is to be received.
  - A mailbox set is a collection of mailboxes grouped together for the purposes of the task.
  - If no message is waiting to be received, the receiving task can either wait at most *n* milliseconds or not wait at all.
- A `port_status()` system call returns the number of messages in a given mailbox.

Communication in Client-Server Systems

Sockets (1/2)

- A ***socket*** is defined as an ***endpoint*** *for communication.*
- A socket is identified by an ***IP address*** and a ***port number***.
  - The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**.
- A pair of processes communicating over a network employ a pair of sockets – one for each process.
  - The server waits for incoming client requests by listening to a specified port.
  - Once a request is received, the server accepts (constructs) a connection from the client socket to complete the connection.
- Why port?
  - A way to communicate is to know the process ID of the hosts.
  - However, IDs change frequently.
  - Each port represents a network services.
  - All ports below 1024 are considered ***well known***.
    - http – 80, telnet – 23, ftp – 21 …

Sockets (2/2)



To initiate a request for a connection, a **client process** is assigned a port by the host computer

This port is an arbitrary Number greater than 1024.

Packets are delivered to the **appropriate process** based on the destination IP and port number

Java Sockets Example (1/4)

- Java provides three different types of sockets:
  - ***Connection-oriented*** (TCP) sockets – implemented with the `Socket` class.
  - ***Connectionless*** (UDP) sockets – implemented with the `DatagramSocket` class.
  - ***Multicast sockets*** – implemented with `MulticastSocket` class.

- Example Server:
  - Use connection-oriented TCP sockets.
  - Listen to port 6013.
  - When a client request is received, the server returns the data and time to the client.

Java Sockets Example (2/4)

DateServer.java

```
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();
         // we have a connection

         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 more connections
         client.close();
     }
   } catch (IOException ioe) {
      System.err.println(ioe);
   }
  }
}
```

Creates a server socket object, bound to the specified port.

Listens for (and **block till**) a connection to be made to this socket and accepts it.

Return a socket that the server can use to communicate with the client.

The server closes the socket to the client and resumes listening for more requests.

http://java.sun.com/j2se/1.4.2/docs/api/java/net/ServerSocket.html

Java Sockets Example (3/4)

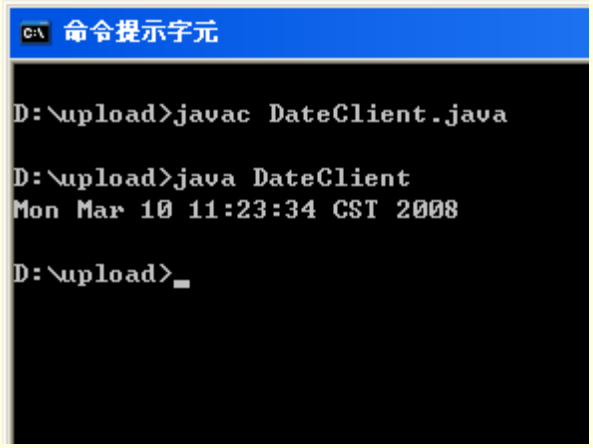
DateClient.java

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

public class DateClient
{
   public static void main(String[] args)   {
     try {
         // this could be changed to an IP name
         // or address other than the localhost
         Socket sock = new Socket("127.0.0.1",6013);
         InputStream in = sock.getInputStream();
         BufferedReader bin = new BufferedReader(new
                                              InputStreamReader(in));

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

         sock.close();
     } catch (IOException ioe) {
         System.err.println(ioe);
     }
   }
}
```

Create a socket and request a connection with the server on port 6013

server的

Java Sockets Example (4/4)



- The IP address 127.0.0.1 is a special IP address known as the ***loopback***.
  - **it is referring to itself!!**
  - The example runs the client and server on the same host to communicate using the TCP/IP protocol.

Remote Procedure Calls (1/6)

- The ***remote procedure call*** (RPC) was designed as a way to abstract the **procedure-call** mechanism for use between systems with network connections.
  - Allow a client to invoke a procedure on a remote host as it would invoke a procedure locally.
- Is a **message-based communication** because the communicating processes are executing on separate systems.
- The message are **well structured** and are thus no longer just packets of data.

Remote Procedure Calls (2/6)

- The RPC takes place by providing a ***stub*** on the client side.
  - Stub: **client-side proxy** for the actual procedure on the server.
  - A separate stub exists for each separate remote procedure.

- RPC operations:
  1. When the client invokes a remote procedure, the PRC system calls the appropriate stub, passing it the required parameters.
  2. The stub **locates the port** on the server and **marshals the parameters**, and then **transmits a message** to the server.
  3. Server has a similar stub that receives this message and invokes the procedure.
  4. If necessary, return values are passed back to the client using the same technique.

Remote Procedure Calls (3/6)

- ***Parameter marshalling***:
  - Different systems use different data representation.
    - Big/little-endian: use the high/low memory address to store the most signification byte.
  - Many RPC systems define a machine-independent representation of data.
    - For example, external data representation (XDR).
  - Parameter marshalling involves converting the machine-dependent data into XDR before they are sent to the server.
  - On the server side, the XDR data are **unmarshalled** and converted to the machine-dependent representation for the server.

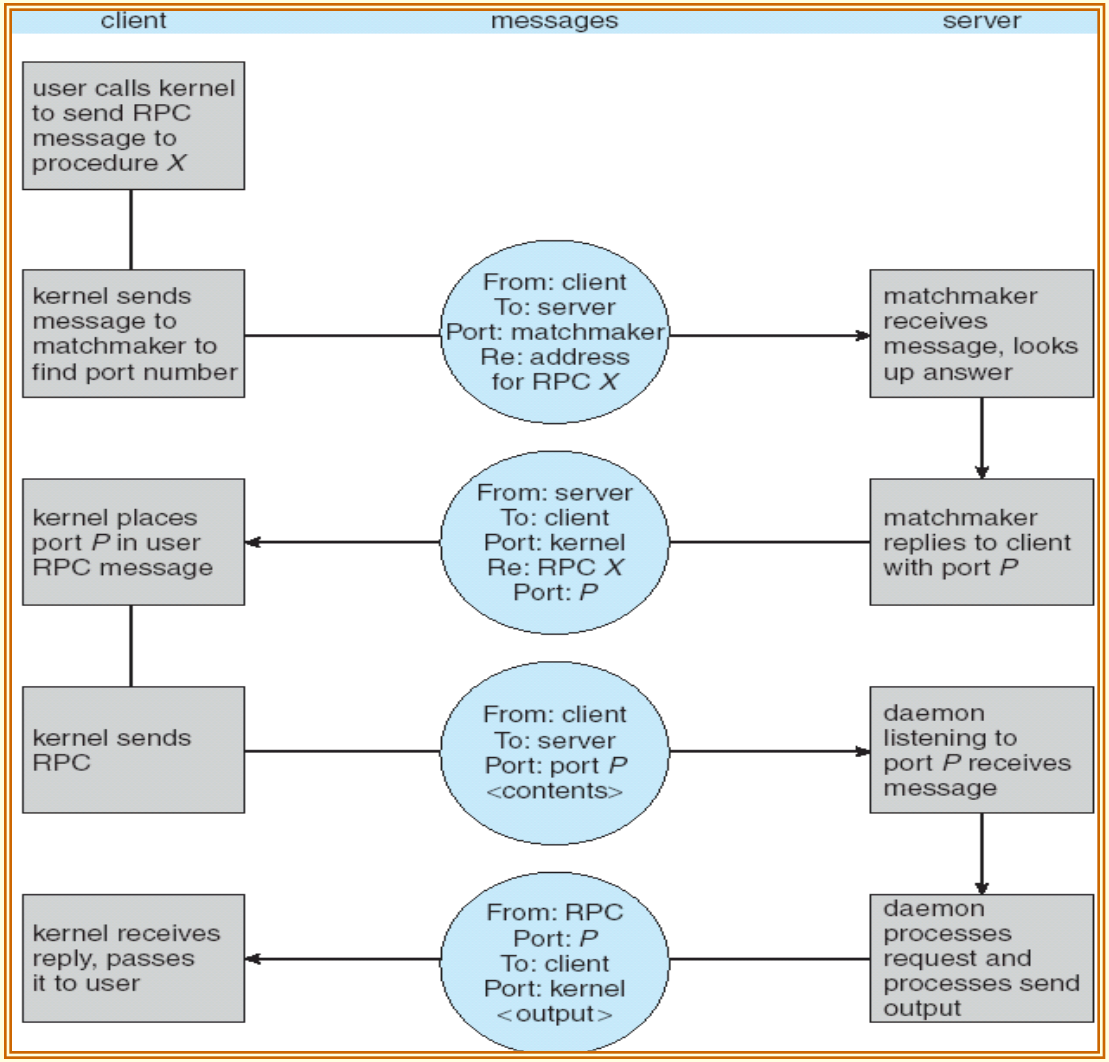
Remote Procedure Calls (4/6)

- Problem 1 – RPCs can **fail**, or be **executed more than once**, as a result of network errors.
  - The operating system have to ensure that messages are acted on ***exactly once***, ***rather than at most one***.
- **For at most once**:
  - Each message is attached a timestamp (in the sender side).
  - The server keep a large history of all the timestamps of messages it has already processed to detect repeated messages.
- **For exactly once**:
  - the server must acknowledge to the client that the RPC call was received and executed.
  - The client must resend each RPC call periodically until it receives the ACK for that call.
  - Of course, the server must implement the “at most once” protocol.

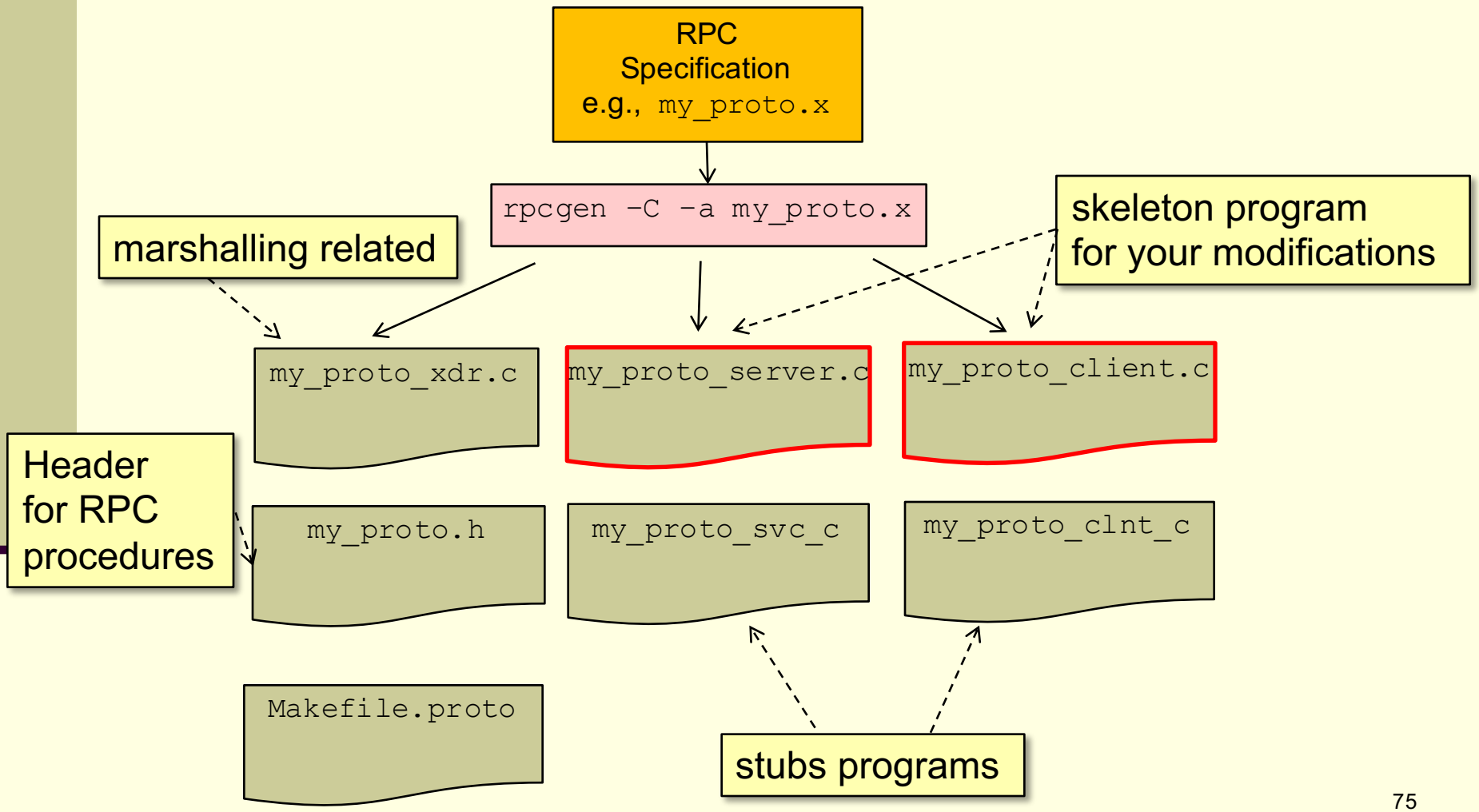
Remote Procedure Calls (5/6)

- Problem 2 – How does a client know the port numbers on the server?
  - Approach 1: the information may be predetermined, in the form of **fixed port addresses**.
    - The server can not change the port number of the requested service.
  - Approach 2: the operating system provides a ***matchmaker*** daemon on a fixed RPC port.
    - A client then sends a message containing the name of the RPC to the matchmaker requesting the port address of the RPC it needs to execute.
    - More flexible, but requires the extra overhead.

Remote Procedure Calls (6/6)



A Simple RPC Example (1/6)



A Simple RPC Example (2/6)

- RPC specification file – e.g., `my_proto.x`
  - Based on SUN RPC Language, a C-like specification language.
  - The template is sufficient for simple services.

```
program TEST_PROG
{
        version TEST_VERS
        {
                long TEST(void) = 1;
        } = 1;
} = 0x31234567;
```

Your RPC functions

RPC number, 2000000 – 3fffffff defined by the user

A Simple RPC Example (3/6)

- `rpcgen -a -C my_proto.x`
  - `-a`: generate skeleton programs for users.
  - `-C`: using ANSI C standard.

- Then you have to modify `my_proto_server.c` and `my_proto_client.c` to define and use RPC procedures.

A Simple RPC Example (4/6)

- my_proto_server.c

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "my_proto.h"

long *
test_1_svc(void *argp, struct svc_req *rqstp)
{
        static long  result;

        /*
         * insert server code here
         */
         result = 100;

        return &result;
}
```

A Simple RPC Example (5/6)

- my_proto_client.c

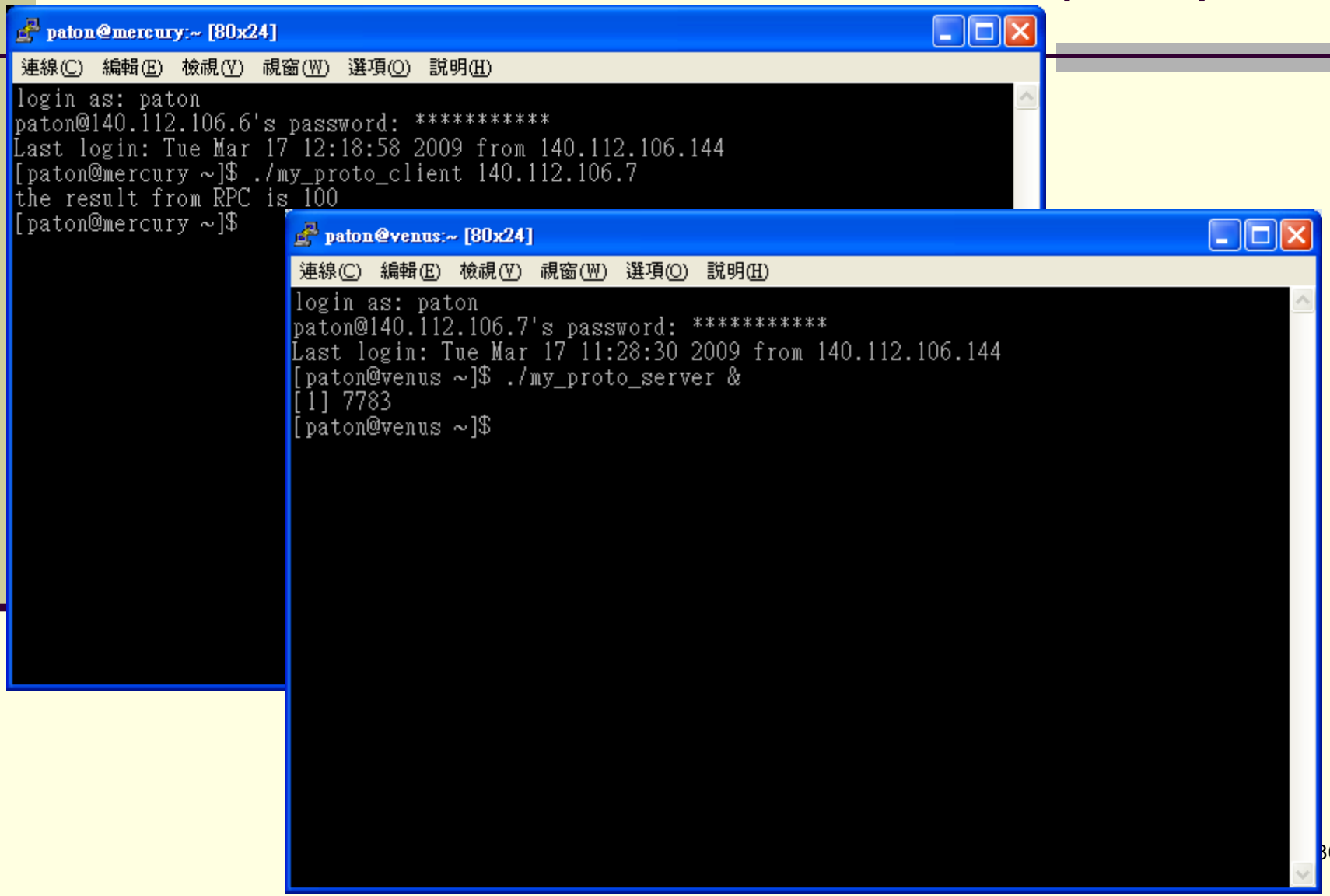
```
...
long *result_1;
#ifndef DEBUG
        clnt = clnt_create (host, TEST_PROG, TEST_VERS, "udp");
        if (clnt == NULL) {
                clnt_pcreateerror (host);
                exit (1);
        }
#endif  /* DEBUG */

        result_1 = test_1((void*)&test_1_arg, clnt);
        if (result_1 == (long *) NULL) {
                clnt_perror (clnt, "call failed");
        }
         printf("The result from RPC is %d\n",*result_1);
#ifndef DEBUG
        clnt_destroy (clnt);
#endif   /* DEBUG */
...
```

to connect remote service

make remote procedure call

disconnect the service

A Simple RPC Example (6/6)



End of Chapter 3

Homework 1: