Chapter 03
Processes

The programs in execution

Outline

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

Objectives

- 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.

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms ***job*** and ***process*** almost interchangeably

Process Concept (Cont.)

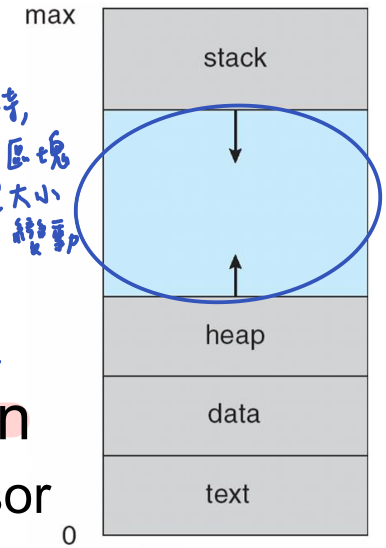
執行中的程式 → 程序

順序執行

執行時，知道一區塊記憶體大小會變動

- **Process** – a program in execution; process execution must progress in sequential fashion
  - ☐ Current activity including program counter, processor registers
  - ☐ Multiple parts

    執行程式碼(可執行碼)
    - The executable code, also called text section
    - Data section containing global variables (全域變數)
    - Heap containing memory dynamically allocated during run time (低 → 高)
    - Stack containing temporary data (高 → 低) 區域變數, return address
      - ➢ Activation record (Function parameters, return addresses, local variables) pushed onto stack
  - ☐ Fixed: Text and data section
  - ☐ Dynamic: Heap and stack
    - Do not overlapped under the control of OS

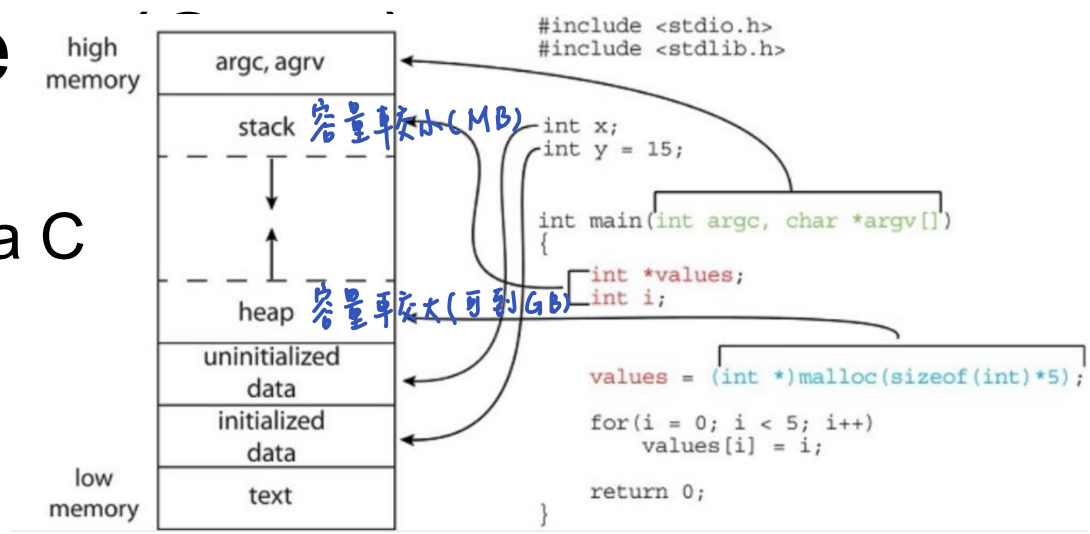


Process in Memory

Process Concept (Cont.)

- Program is ***passive*** entity stored on disk (executable file), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program
- Process as execution environment (Java as example)
  - Java virtual machine (JVM) as a process 程序也可以變成環境
  - Executable Java program is executed within JVM
  - E.g. run a java program Program.class by
    java Program

Process Conce



- Memory Layout of a C Program
  - Global data section
    - Initialized data
    - Uninitialized data
  - Separate section for argc and argv
  - GNU size command for determining the size. For example,
    size memory
    results in

data section

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

Exercises

- When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process?
  - [ ] Stack
  - [ ] Heap
  - [x] Shared memory segments

每個程序獨自擁有

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

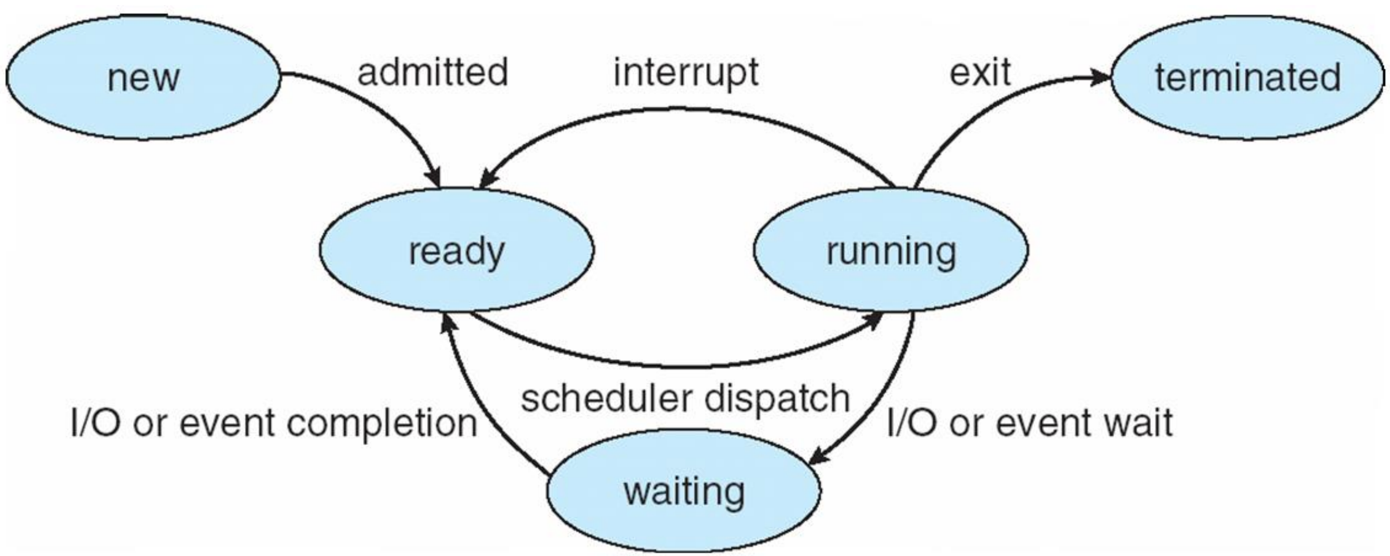


Diagram of Process State

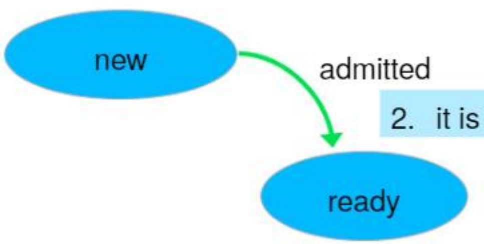
Process State

Process State Changes

1. a new process is created

Process State

Process State Changes

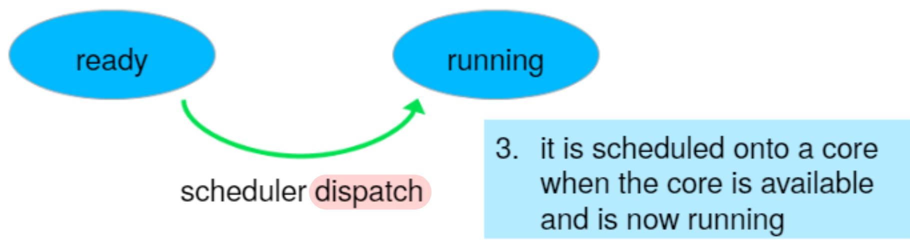


OS分配資源

2. it is moved to the ready state

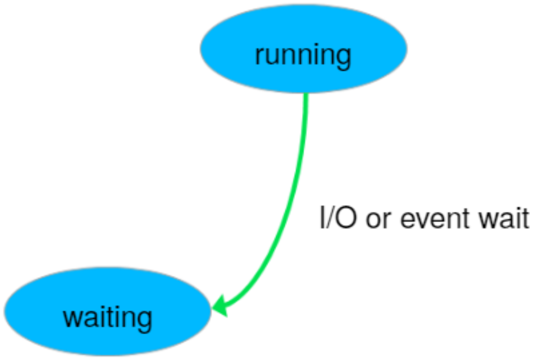
Process State

Process State Changes



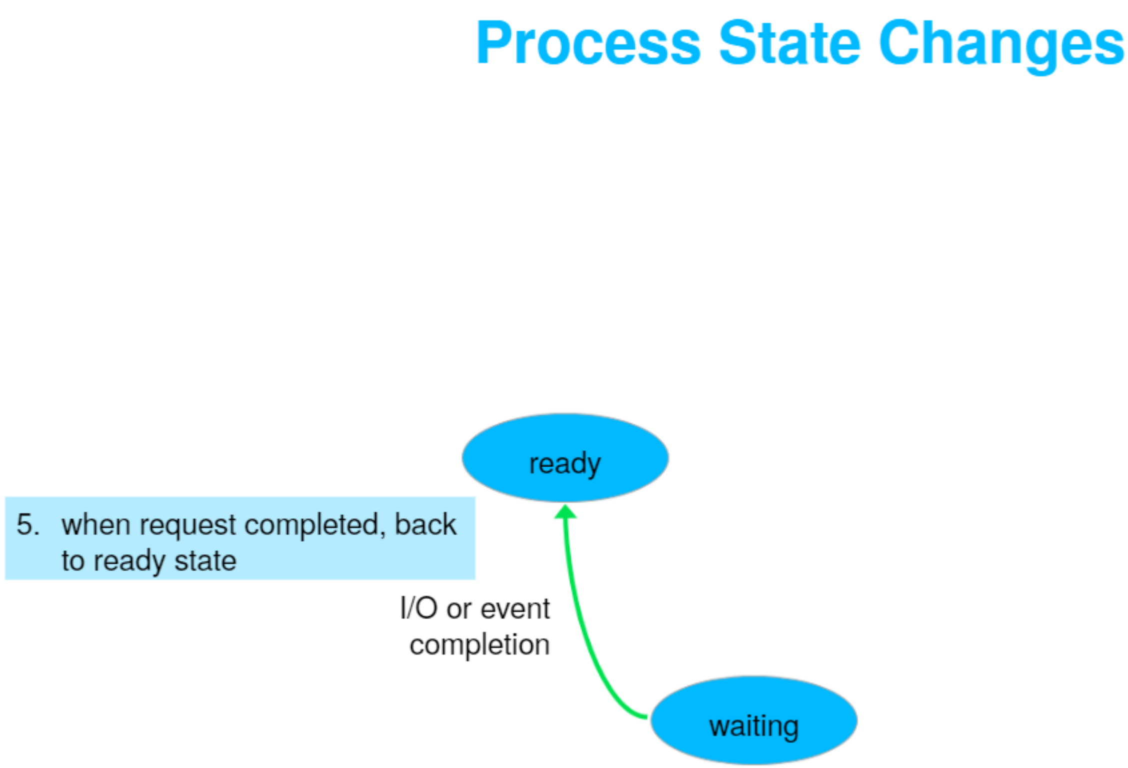
Process State

Process State Changes

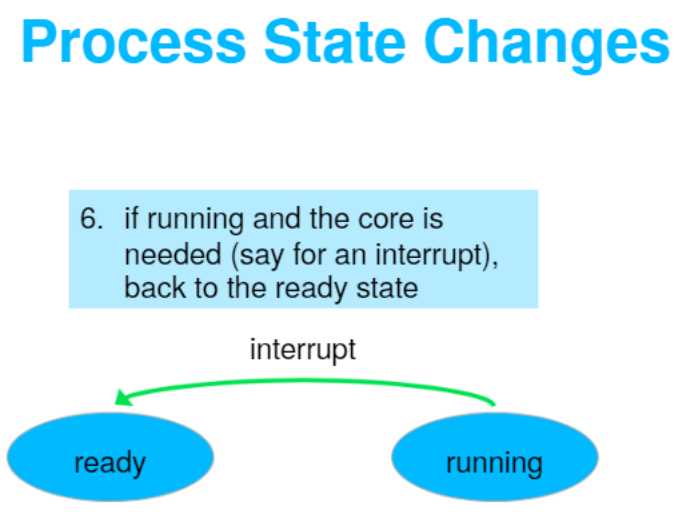


4. if an I/O request or event request occurs, moves to waiting state

Process State



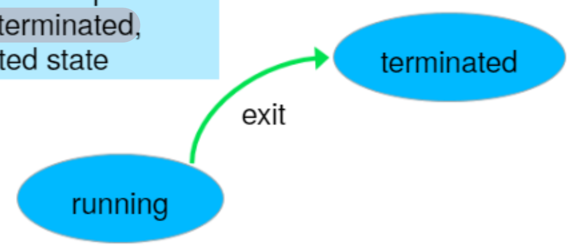
Process State



Process State

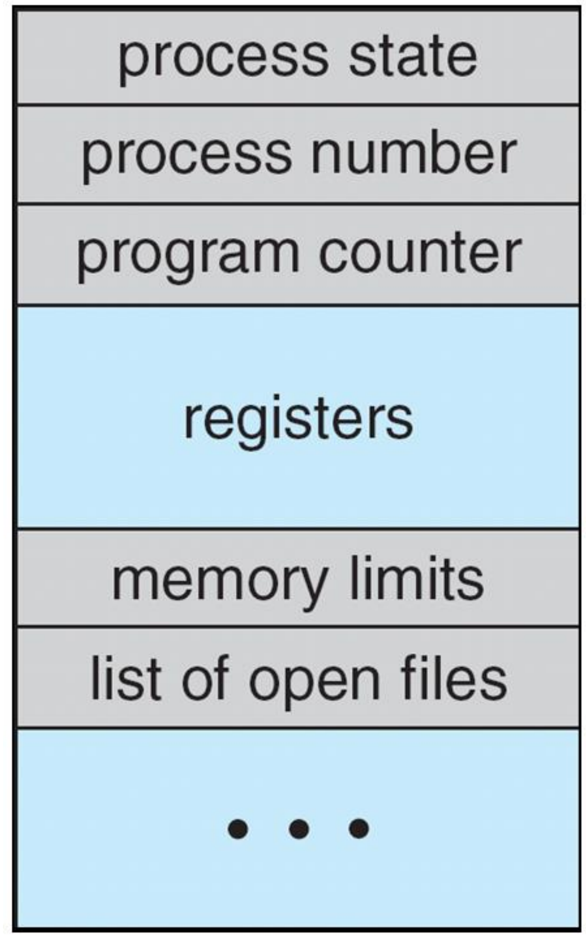
Process State Changes

7. cycle continues until the process exists, fails, or is terminated, moves to terminated state



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
  - 記憶體範圍 **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 (執行緒)

- 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

Exercise

- Some computer systems provide multiple register sets. Describe what happens when a context switch occurs if the new context is already loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use? 程式切換

Process Representation in Linux

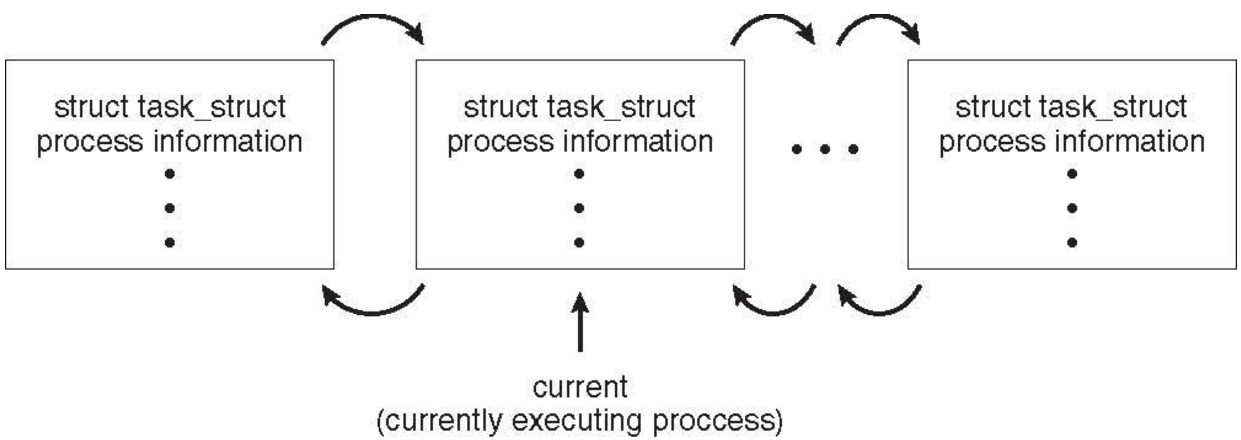
- Represented by the C structure `task_struct`:

指向父程序
(單一)
link list, 子程序
(會有多個)

```
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 */
```

- Change a process's state by assigning the member of state a new value

```
current->state = new_state;
```

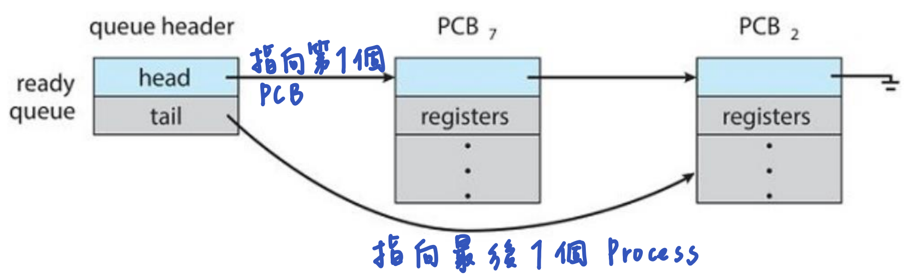


Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on a core
  - Each CPU core for one process
- Degree of multiprogramming - the number of processes in memory 影響 CPU utilization
- Types of processes
  - I/O-bound process – more time on I/O
  - CPU-bound process – more time on computation

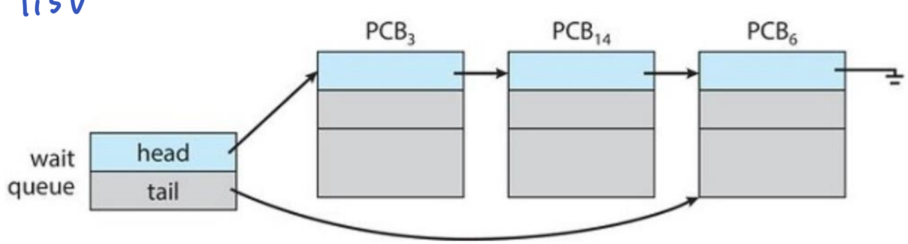
Process Scheduling

- Maintains scheduling queues of processes
  - Ready queue – set of all processes residing in main memory, ready and waiting to execute
    - Stored in a linked list, header to first PCB



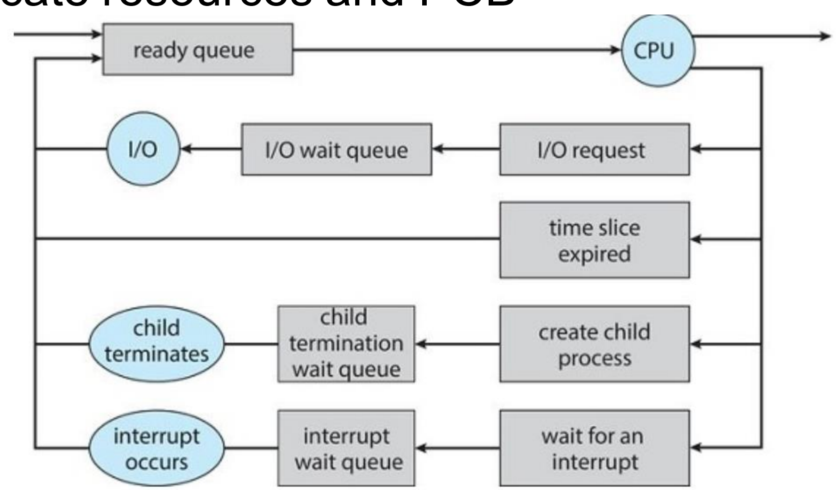
  - Wait queue – processes wait for event occur

link list



Process Scheduling

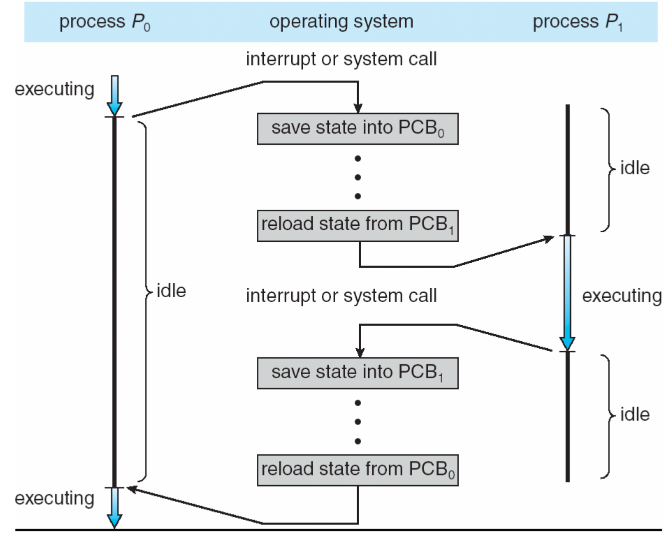
- Queueing diagram represents queues, resources, flows
  - New process is put into ready queue until it is dispatched
  - When executing, the process could
    - Issue an I/O request an go to I/O wait queue
    - Create a child process and go to wait queue
    - Be removed forcibly due to interrupt or expiring time slice
  - Remove from all queues when process terminated
    - Deallocate resources and PCB



Process Scheduling

- CPU Scheduling
  - CPU scheduler selects from among the processes that are in the ready queue and allocate a CPU core to one of them
  - Frequently executes
    - Every 100 milliseconds or more
  - Intermediate scheduling
    - A.k.a. swapping
    - Remove a process from memory to reduce the degree of multiprogramming, and re-enter into memory later
    - Necessary when memory is overcommitted

Process Scheduling



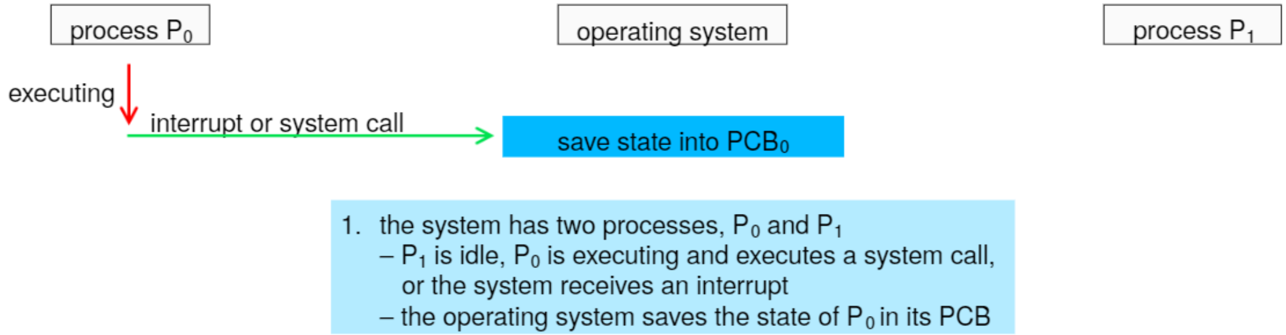
程序間進行切換

- **Context Switch**
  - 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 via a context switch
  - Context of a process represented in the PCB
    - Value of CPU registers, process state, memory-management information
  - State save of the current state, and state restore to resume operations
  - Time dependent on hardware support
    - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once
  - Context-switch time is overhead; the system does no useful work while switching
    - The more complex the OS and the PCB ➔ the longer the context switch

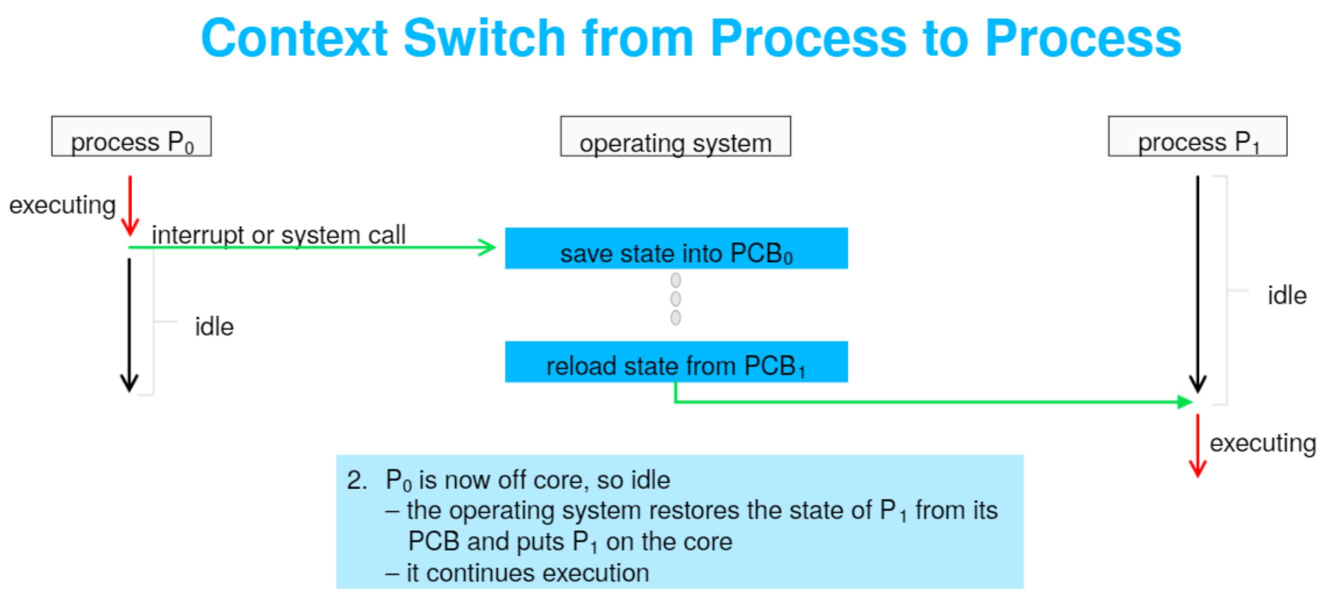
如果 Context-switch 太頻繁, ⇒ CPU utilization ↓

Process Scheduling

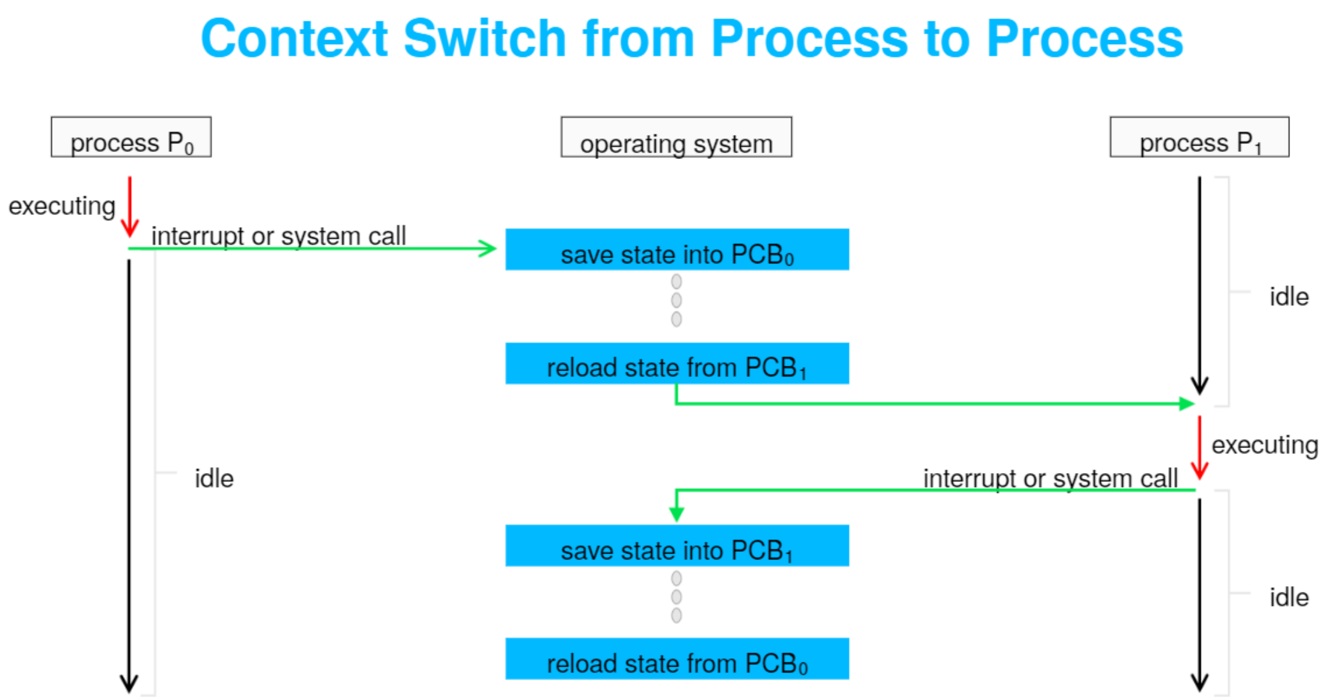
Context Switch from Process to Process



Process Scheduling



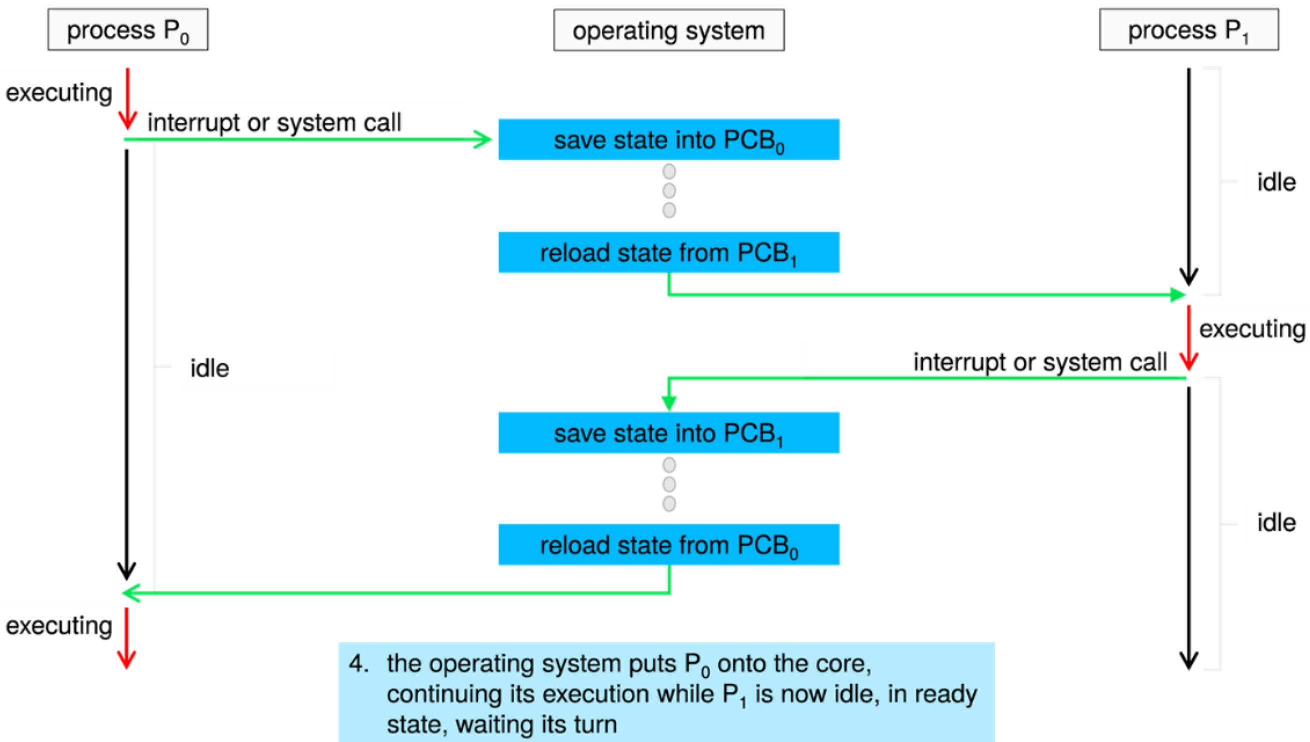
Process Scheduling



3. P1 execution is interrupted, the operating system saves its state to its PCB and restores the next process's state (P0 in this case) to prepare it to continue execution

Process Scheduling

Context Switch from Process to Process

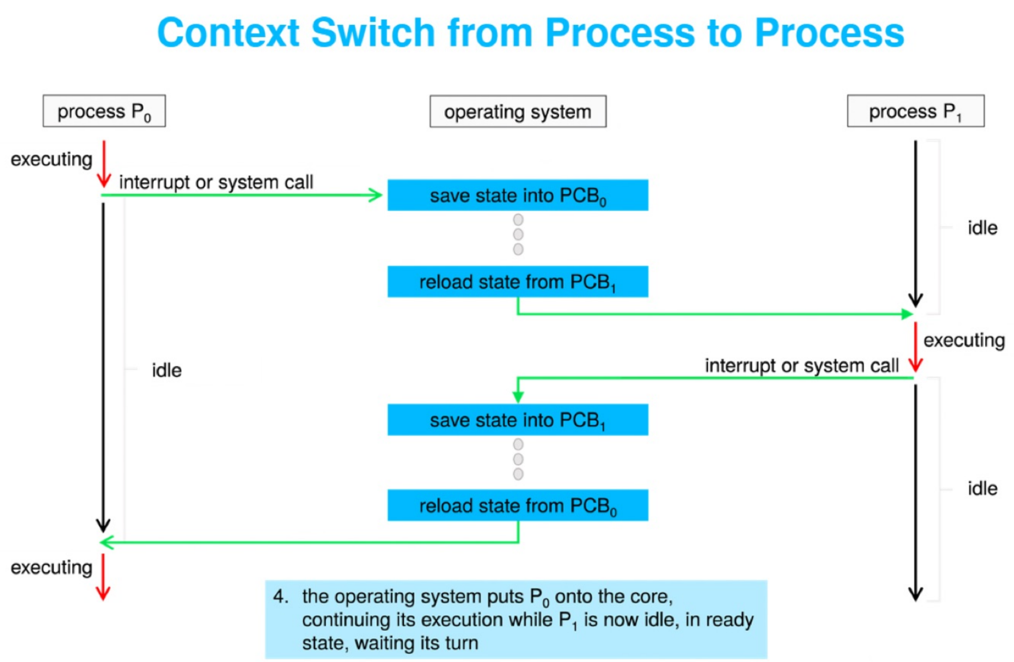


Process Scheduling

- Multitasking in Mobile Systems
  - Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
  - Due to screen real estate, user interface limits iOS provides for
    - 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
    - Fewer restrictions as the hardware progress. For example, the iPad tablets allow split-screen – running two foreground apps at a time
  - 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

Exercise

- Describe the actions taken by a kernel to context-switch between processes.

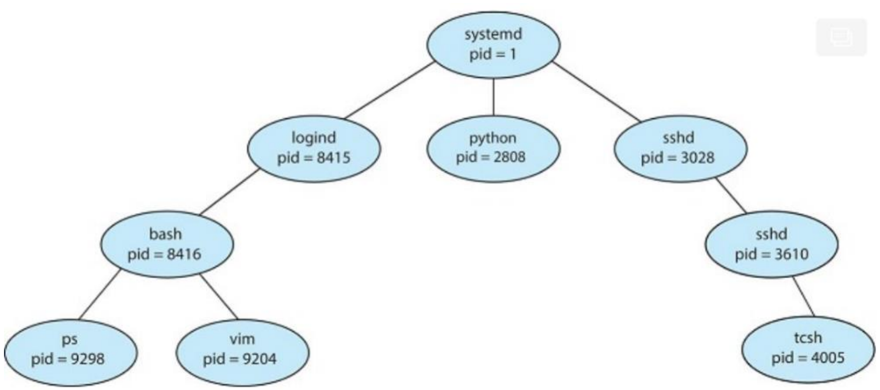


Operations on Processes

- System must provide mechanisms for:
  - Process creation
  - Process termination
  - …

Operations on Processes

- 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)
    - Using command `ps` for listing processes in UNIX and Linux: `ps -el`
    - Linux provide `pstree` command



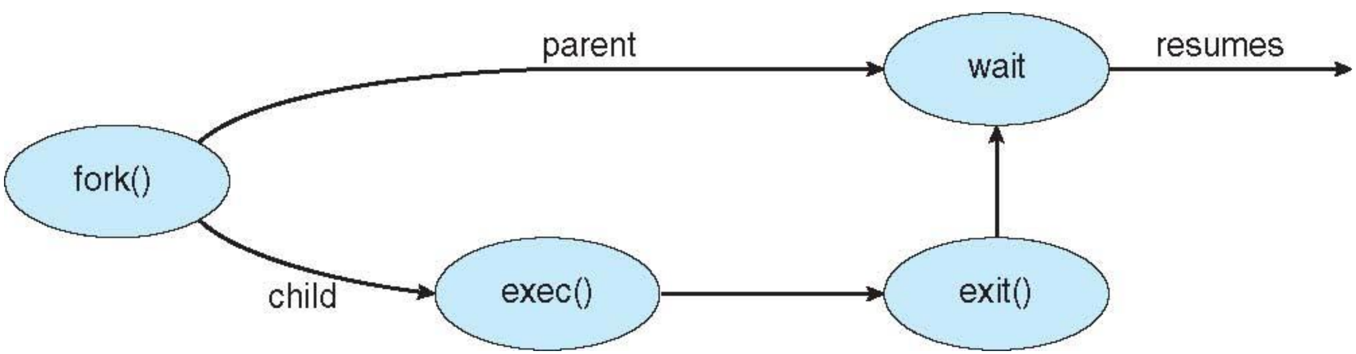
Operations on Processes

- Resource sharing options
  - Parent and children share all resources 全部
  - Children share subset of parent's resources 部分
  - Parent and child share no resources 沒有
- Pass input from parent to child process
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space options
  - Child duplicate of parent
  - Child has a program loaded into it

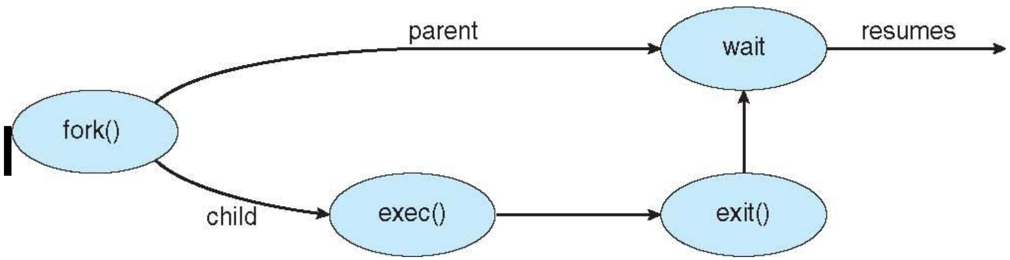
Operations on Processes

- UNIX Example
  - Process identifier pid for each process
  - **`fork()`** system call creates new process 判別是否是 child process
    - Both parent and child processes run concurrently
    - Return child's pid for parent process
    - Return 0 for the child process

    $\begin{cases} 0 \text{ , child process} \\ \text{child's pid, parent process} \end{cases}$
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
    - Loads executable file into memory and starts execution
    - Parent process can do something or just wait by **`wait()`**
    - Does not return until an error occurs



Operations on Pr



- UNIX Example
  - **`fork()`** creates child process
    - pid < 0: error
    - pid = 0: child
    - pid > 0: parent
  - Child inherits from parent
    - Privileges and scheduling attributes
    - Resources
  - **`execlp()`** runs a UNIX command
    - A version of **`exec()`**
  - **`wait()`** lets parent wait child process to terminate

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

int main()
{
pid_t pid;

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

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

    return 0;
}
```

C Program Forking Separate Process

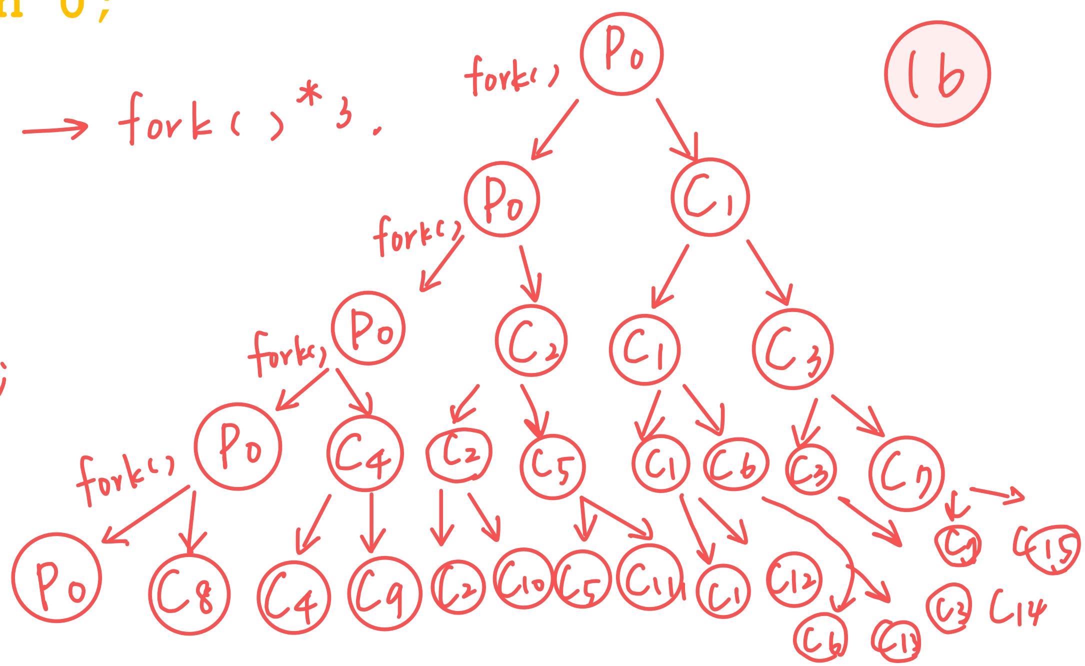
Exercise

- Including the initial parent process, how many processes are created by the program shown below?

```
#include <stdio.h>
#include <unistd.h>
int main() {
   for (int i = 0; i < 4; i++) fork();
   return 0;
}
```

fork( ); → fork( )*3.
fork( );
fork( );
fork( );

16



Operations on Processes

- Process Termination
  - Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
    - Returns status data from child to parent (via **`wait()`**)
    - Process' resources are deallocated by operating system
  - Parent may terminate the execution of children processes using proper system call.
    - **`TerminateProcess()`** in Windows
    - **`abort()`** in UNIX
    - Need to know the process identifiers to be terminated

More Exercises

- Explain the circumstances under which the line of code marked **`printf("LINE J");`** will be reached.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
   pid_t pid;
   pid = fork(); //fork a child process
   if (pid < 0) { //error occurred
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { //child process
      execlp("/bin/ls","ls",NULL); → TRUE, 永遠不會 print "LINE J"
      printf("LINE J"); → Error message
   }
   else { //parent process
      wait(NULL); // parent waits for child
complete
      printf("Child Complete");
   }
}
```

Operations on Processes

- Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
- Some operating systems do not allow child to exists if its parent has terminated. OS不允許沒有父程序的子程序.
  - If a process terminates, then all its children must also be terminated. This is known as **cascading termination.**
  - The termination is initiated by the operating system.
- Terminate via **`exit()`** system call
  - **`exit(1)`**: exit with status 1
  - Called directly or indirectly

Operations on Processes

- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

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

- If no parent waiting (did not invoke **`wait()`**) process is a zombie 僵屍
  - Only exists briefly
- If parent terminated without invoking **`wait`**, process is an orphan 孤兒

作為 parent

  - **`init`** (root of UNIX system) as the new parent in UNIX
  - **`systemd`** or other processes in Linux systems

Exercise

- Explain the role of the **`init`** (or **`systemd`**) process on UNIX and Linux systems in regard to process termination.

`init` (root of UNIX system) as the new parent in UNIX
`systemd` or other processes in Linux systems

Operations on Processes

- Android Process Hierarchy
  - Terminate process to reclaim memory due to resource constraints
  - According to importance hierarchy in increasing order
  - From most to least important processes are

越高
越容易
中止

  - Foreground process—The process the user is currently interacting with and visible on the screen
  - Visible process—The process not directly visible but performing an activity referred to the foreground process
  - Service process—A process running on the background with apparent activity to the user
    - E.g. streaming music
  - Background process—A process performing an activity not apparent to the user.
  - Empty process—A process holding no active components

高

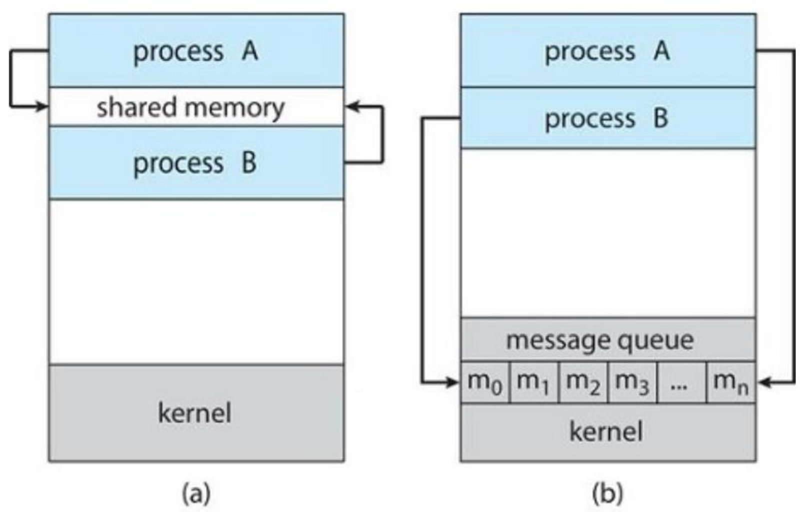
  - Assign as high rank as possible

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** ex, copy and paste.
  - **Computation speedup** 工作分割
  - **Modularity** 模組化

Interprocess Communication

- Cooperating processes need interprocess communication (IPC) with two models
  - **Shared memory** – read / write to shared memory
    - Faster due to shared-memory regions
    - No kernel intervention for memory accesses 不用透過kernel
  - **Message passing**
    - Exchange small data due to no conflict 不用注意同步的問題
    - Easy to implement 容易實作



IPC in Shared-Memory Systems

- Shared memory requires that two or more processes agree to remove the constraint of accessing another process's memory.
- **Producer-Consumer Problem** (生產者 v.s. 消費者)
  - Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - Must have a buffer of items filled by producer and emptied by consumer
    - Need to synchronize for using item before filling it
    - Unbounded-buffer places no practical limit on the size of the buffer: Continue producing without waiting consuming 不限制緩充大小
    - Bounded-buffer assumes that there is a fixed buffer size: Producer should wait for consuming 限制緩充大小

IPC in Shared-Memory Systems

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

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

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

Consumer 消費 item

生產

消費

  - Solution is correct, but can only use BUFFER_SIZE-1 elements
    - **Empty** if `in == out`
    - **Full** if `((in + 1) % BUFFER_SIZE) == out`

IPC in Shared-Memory Systems

- Bounded-Buffer – Producer 產生 item

```
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;
}
```

IPC in Shared-Memory Systems

- Bounded-Buffer – Consumer 使用 item

```
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 */
}
```

IPC in Shared-Memory Systems

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

IPC in Message-Passing Systems

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
  - Useful in distributed system
  - E.g. internet chat room
- Message-passing facility provides two operations:
  - `send(message)`
  - `receive(message)`
- The message size is either
  - Fixed: easy for system implementation but difficult to programming, or
  - Variable: easy for programming but require more complex system implementation

IPC in Message-Passing Systems

- If processes P and Q wish to communicate, they need to:
  - 使用 send & receive
  - Establish a communication link between them
  - Exchange messages via send/receive
- Implementation issues
  - Physical
    - Shared memory
    - Hardware bus
    - Network
  - Logical
    - Direct or indirect communication
    - Synchronous or asynchronous communication
    - Automatic or explicit buffering

IPC in Message-Passing Systems

- Naming
  - Communication can be direct or indirect
  - Direct communication
    - Processes must name each other explicitly:
      - **`send`** (*P, message*) – send a message to process P
      - **`receive`**(*Q, message*) – receive a message from process Q
    - Properties of communication link
      - Links are established automatically
      - A link is associated with exactly one pair of communicating processes
      - Between each pair there exists exactly one link
    - Symmetry in addressing- sender process and receiver process must name the other.

IPC in Message-Passing Systems

- Asymmetry in addressing- only the sender names the recipient
- Processes for communication are
  - **`send(P, message)`**—Send a message to process P
  - **`receive(id, message)`**—Receive a message from any process
- Disadvantages of direct communication
  - Limited modularity due to the use of hard-coding techniques
  - Need to check all process definitions when changing the identifier of one process

不對稱只有這種傳輸方式!!

傳送資訊給特定的人

不限制資訊來源(來者不拒)

IPC in Message-Passing Systems

- Indirect Communication
  - Messages are directed and received from mailboxes (also referred to as ports) 埠號
    - Each mailbox has a unique id
    - Processes can communicate only if they share a mailbox
  - Primitives are defined as:
    - **`send`**(*A, message*) – send a message to mailbox A
    - **`receive`**(*A, message*) – receive a message from mailbox A
  - 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

IPC in Message-Passing Systems

- 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.

IPC in Message-Passing Systems

- A mailbox may be owned either by a ***process*** or by the ***operating system***.
- Owned by ***process***
  - Owner: receive messages
  - User: send messages
  - Disappear when owner process terminates
- Owned by ***operating system***
  - Independent to process
  - Allow process to do 主動建立與移除 mailbox
    - ✓ Create a new mailbox (port)
    - ✓ Send and receive messages through mailbox
    - ✓ Destroy a mailbox
  - Ownership
    - ✓ Creator
    - ✓ Pass to other processes through proper system calls

IPC in Message-Passing Systems

- Synchronization
  - Message passing may be either blocking or non-blocking
  - Blocking is considered synchronous
    - Blocking send -- the sender is blocked until the message is received
    - Blocking receive -- the receiver is blocked until a message is available
  - Non-blocking is considered asynchronous
    - Non-blocking send -- the sender sends the message and continue
    - Non-blocking receive -- the receiver receives:
      - A valid message, or
      - Null message

IPC in Message-Passing Systems

- Different combinations possible 會合
  - If both send and receive are blocking, we have a rendezvous
  - Producer

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

  - Consumer

```
message next_consumed;
while (true) {
    receive(next_consumed);
    /* consume the item in next_consumed */
}
```

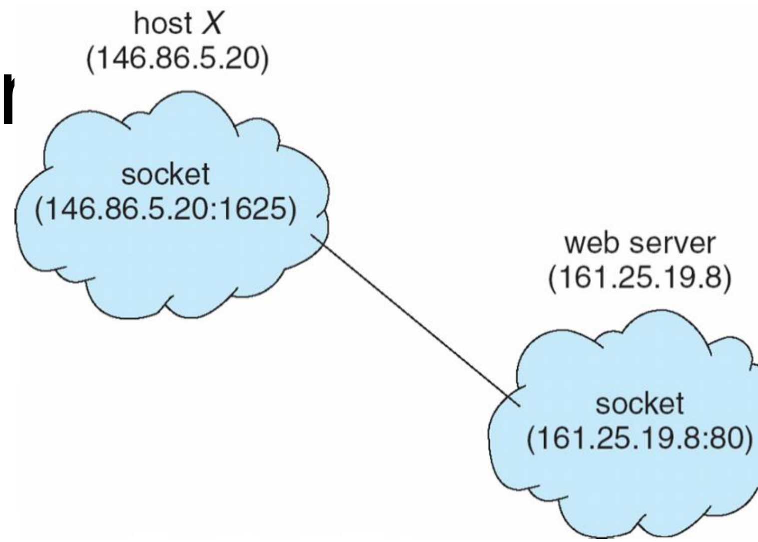
IPC in Message-Passing Systems

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

Communications in Client-Server Systems

- Shared Memory
- Message Passing
- **Sockets**
- **Remote Procedure Calls**

Communications in Client-Server Systems



- Sockets
  - An endpoint for communication
    - A pair of sockets for communicating processes
  - Concatenation of IP address and port – a number included at start of message packet to differentiate network services on a host
    - Socket 161.25.19.8:1625 == port 1625 on host 161.25.19.8
  - Client–server architecture: server waits clients by listening to a specified port.
  - All ports below 1024 are well known, used for standard services for server
    - E.g. port 22 for SSH, port 21 for FTP, port 80 for HTTP
  - Clients use arbitrary port number greater than 1024.

Communications in Client-Server Systems

- Special IP address 127.0.0.1 (loopback) to refer to system on which process is running
  - Allow communication between client and server on the same machine
- Common and efficient
- Low-level communication
  - Only allow unstructured stream of bytes
  - Structure on data should be imposed by client or server

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- Stubs
  - Client-side proxy for the actual procedure on the server
  - The client-side stub locates the server and marshals the parameters
  - The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

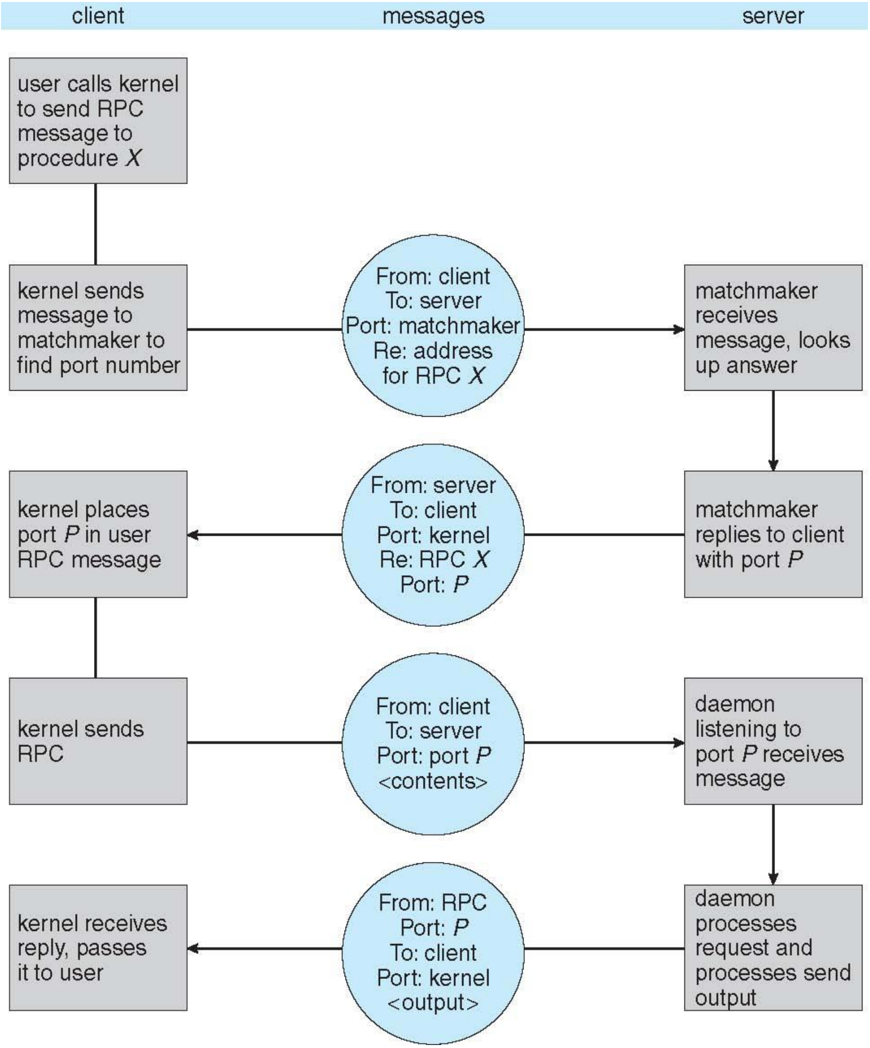
Remote Procedure Calls

- Data representation handled via External Data Representation (XDR) format to account for different architectures
  - Big-endian and little-endian
  - Client-side parameter marshaling involves converting machine-dependent data into XDR.
- Remote communication has more failure scenarios than local
  - Due to duplication or more than once execution
  - Ensure messages are delivered ***exactly once*** rather than ***at most once***
    - ***At most once:*** attaching a timestamp to each message
    - ***Exactly once:*** use at most once with acknowledge mechanism (ACK messages)

Remote Procedure Calls

- The RPC scheme requires a binding of the client and the server port. But how?
- Two approaches:
  - Statically bind by fixed port address
    - An RPC call has a fixed port number
  - Dynamically bind by rendezvous mechanism
    - OS typically provides a rendezvous (or matchmaker) service to connect client and server
    - Rendezvous service on a fixed RPC port
    - Client sends request to rendezvous daemon for port address of RPC
    - Overhead of the initial request but more flexible

Remote Procedure Calls



Execution of RPC