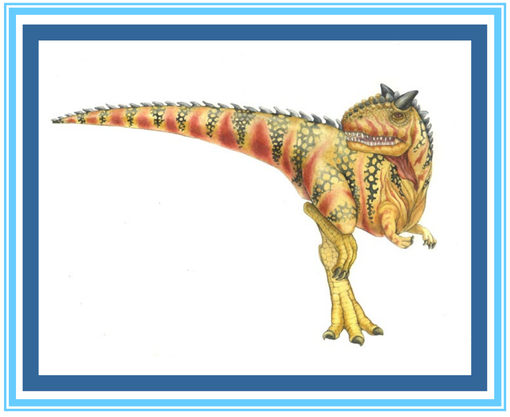
Chapter 3: Processes



Chapter 3: Processes

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

Objectives

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

Process Concept

- An OS executes a variety of programs:
  - **Batch system** – jobs (assigned work)
  - **Time-shared systems** – user programs or tasks
  - Even on a *single-user system*, a user may be able to run several programs at one time:
    - a word processor, a Web browser, and an e-mail package.

The Process (1)

- **Process** – a program in execution (its .exe file in main memory, and it is taken by the CPU for execution or waiting for execution); process execution must progress sequentially.
- A **process** has multiple parts:
  - The **program code**, also called the **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing *temporary data* such as
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
    - The structure of a process in memory is shown in **Figure 3.1.**

The Process (2)

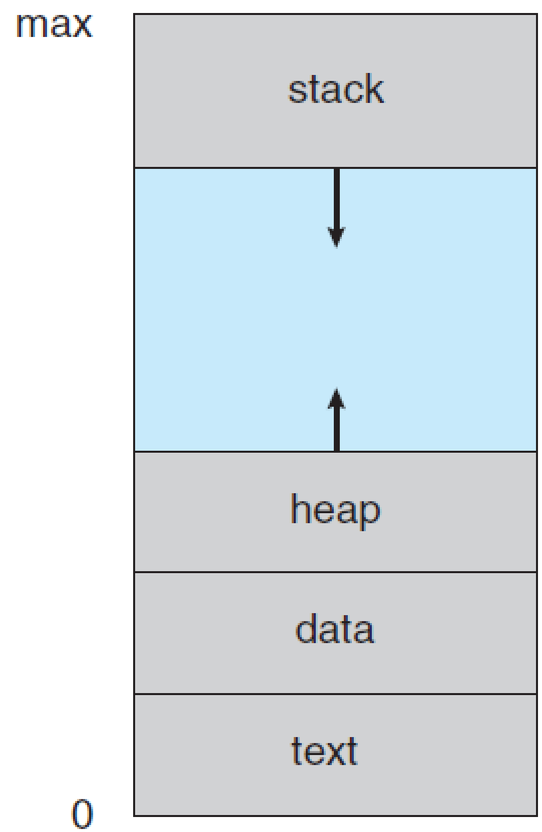


Figure 3.1 Process in memory.

The Process (3)

- It is noticed that a program (.exe file) by itself is not a process.
- A **program** is a ***passive entity,*** such as a file containing a list of instructions stored on a disk memory (often called an **executable file**).
- In contrast, a **process** is an ***active entity*** in **main memory** with a PC reg. Specifying the next instruction to execute with a set of associated resources.
  - That is, <u>**a program becomes a process only when its executable file is loaded into main memory**</u>.

The Process (4)

- Although **two processes** may be associated with the same program, they are considered two separate execution sequences.
  - For instance, several users may be running different copies of the mail program, or the same user invoke many copies of the web browser, are **separate processes**.
    - and although the text sections are equivalent, the data, heap, and stack sections vary.
- It is also common to have a **process** that ***spawns*** (generate a new) many processes as it runs.
  - An executable **Java** code is executed within the *Java virtual machine* (JVM). The JVM executes as **a process** that interprets the loaded *Java code* and takes actions.

Process State (1)

- As a **process** executes, it changes **state**;
  - **The state of a process is defined in part by the current activity of that process**.
- A **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 or reception of a signal).
  - **Ready:** The process is waiting to be assigned to a processor.
  - **Terminated:** The process has finished execution.

Process State (2)

- Only one process can be running on any processor at any instant. *Many* processes may be ***ready*** and ***waiting***.
  - *The state diagram corresponding* to these states is presented in **Figure 3.2.**

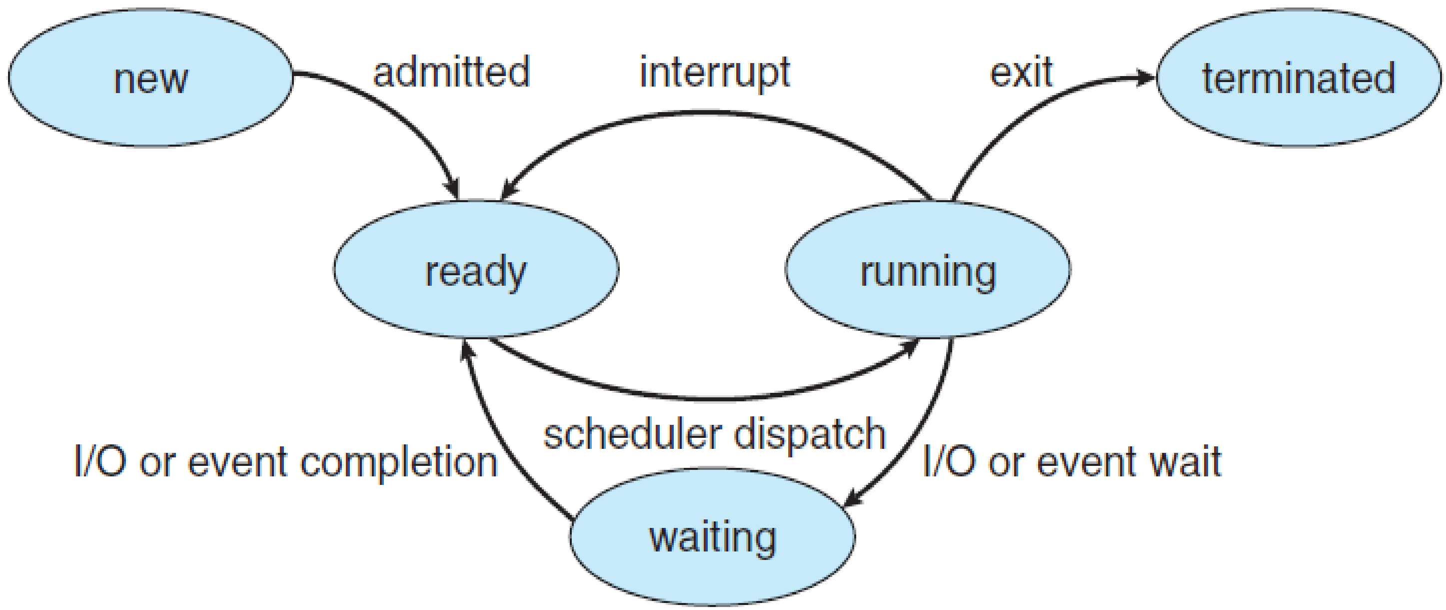


Figure 3.2 Diagram of process state.

Process Control Block (1)

- Each process is represented in the OS by a **process control block** (**PCB**)—also called a **task control block.**
- A **PCB** is shown in **Figure 3.3**.

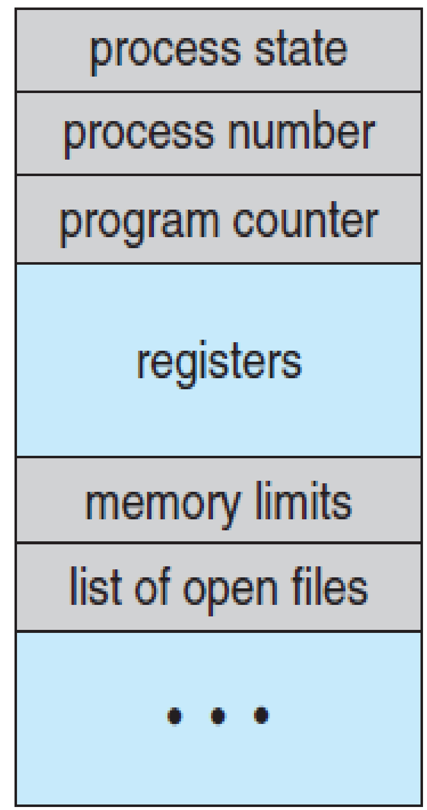


Figure 3.3 Process control block (PCB).

Process Control Block (2)

- A **PCB** contains many pieces of information associated with a specific **process**, including these:
  - **Process state:** The state may be new, ready, running, waiting, halted, and so on.
  - **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
  - **CPU registers:** The size of registers depending on the computer architecture.
  - **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, etc.
  - **Memory-management information:** This information includes the value of the *base* and *limit* registers and the *page tables*, etc
  - **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, etc.
  - **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

A CPU Switches from Process to Process

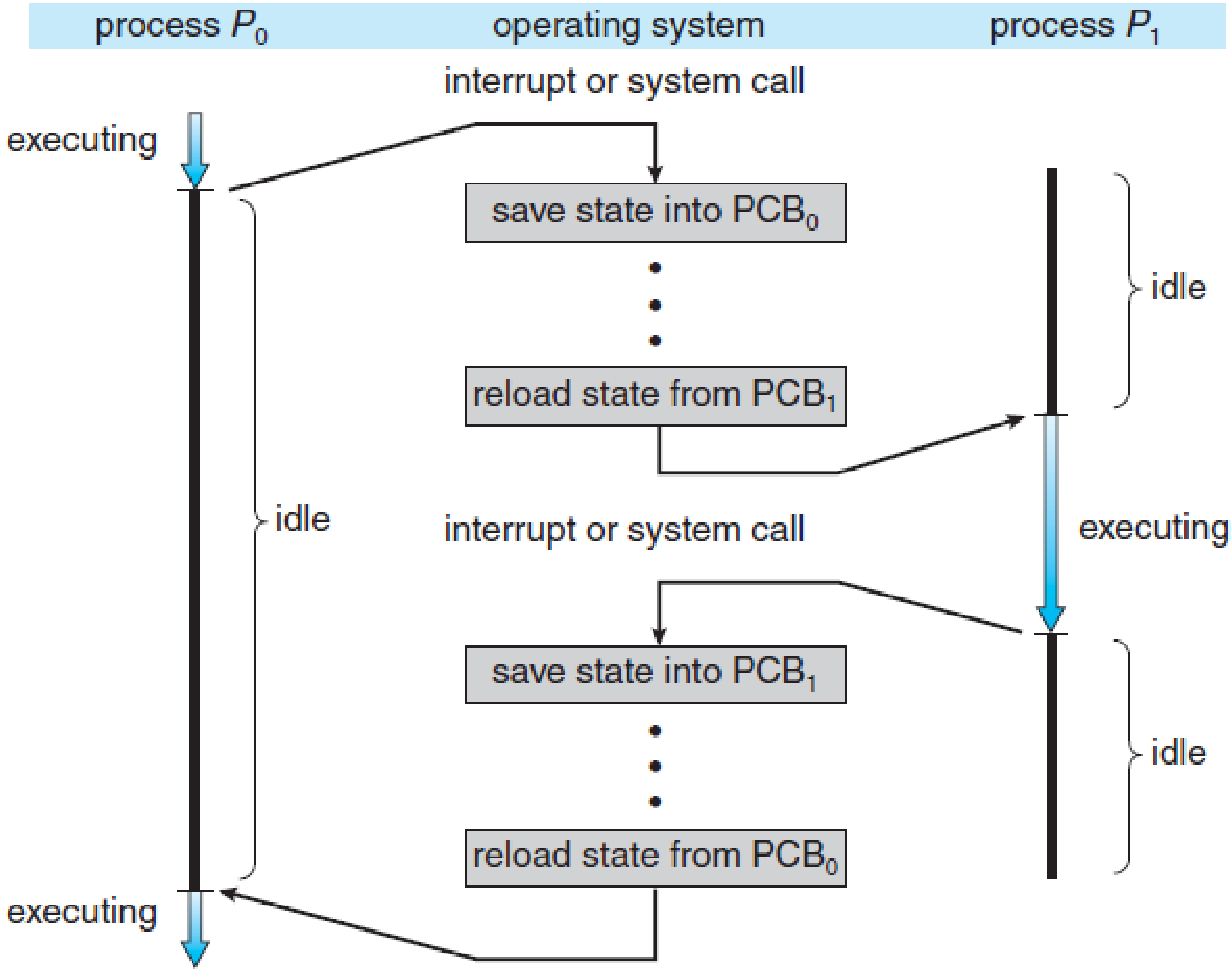


Figure 3.4 Diagram showing CPU switch from process to process.

Threads

- The process model discussed **is a program** that performs a **single thread** of execution.
  - For example, when a process is running a word-processor program, a **single thread of instructions** is being executed at each time.
  - This **single thread of control** allows the process to perform only one task at a time.
- Most modern OSs have extended the process concept to allow a process to have **multiple threads** of execution and thus to perform more than one task at a time.
  - This feature is especially beneficial on **multicore** systems, where **multiple threads can run in parallel**.

Process Scheduling

  - The objective of **multiprogramming** is to have some process running at all times, to **maximize CPU utilization**.
  - The objective of **time sharing** is to switch the CPU among processes so users can interact with each program while it is running.
- To meet these objectives, the **process scheduler** selects an available process for program execution on the CPU.
- For a **single-processor system**, there will never be more than one running process.
  - If there are more processes, the rest will have to wait until the CPU is free.

Scheduling Queues

- As **processes** enter the system, they are put into a **job queue,**
  - **which consists of all processes in the system.**
- The processes that are residing in **main memory** and are **ready** and **waiting** to execute are kept on a list called the **ready queue.**
  - This **queue** is generally stored as a linked list.
  - A **ready-queue** header contains pointers to the first and final PCBs in the list.
  - Each PCB includes a pointer field that points to the next PCB in the **ready queue**.
- The list of processes waiting for a particular **I/O device** is called a **device queue.**

Scheduling Queues

- A common representation of **process scheduling** is a **queuing diagram,** such as that in **Figure 3.6**.
  - Each rectangular box represents a **queue**.
- Two types of queues are present:
  - the **ready queue** and
  - a set of **device queues**
  - The **circles** represent the ***resources*** that serve the queues, and the **arrows** indicate the ***flow of processes*** in the system.
- A **new process** is initially put in the **ready queue**.
  - It waits there until it is selected for execution, or **dispatched.**

Scheduling Queues

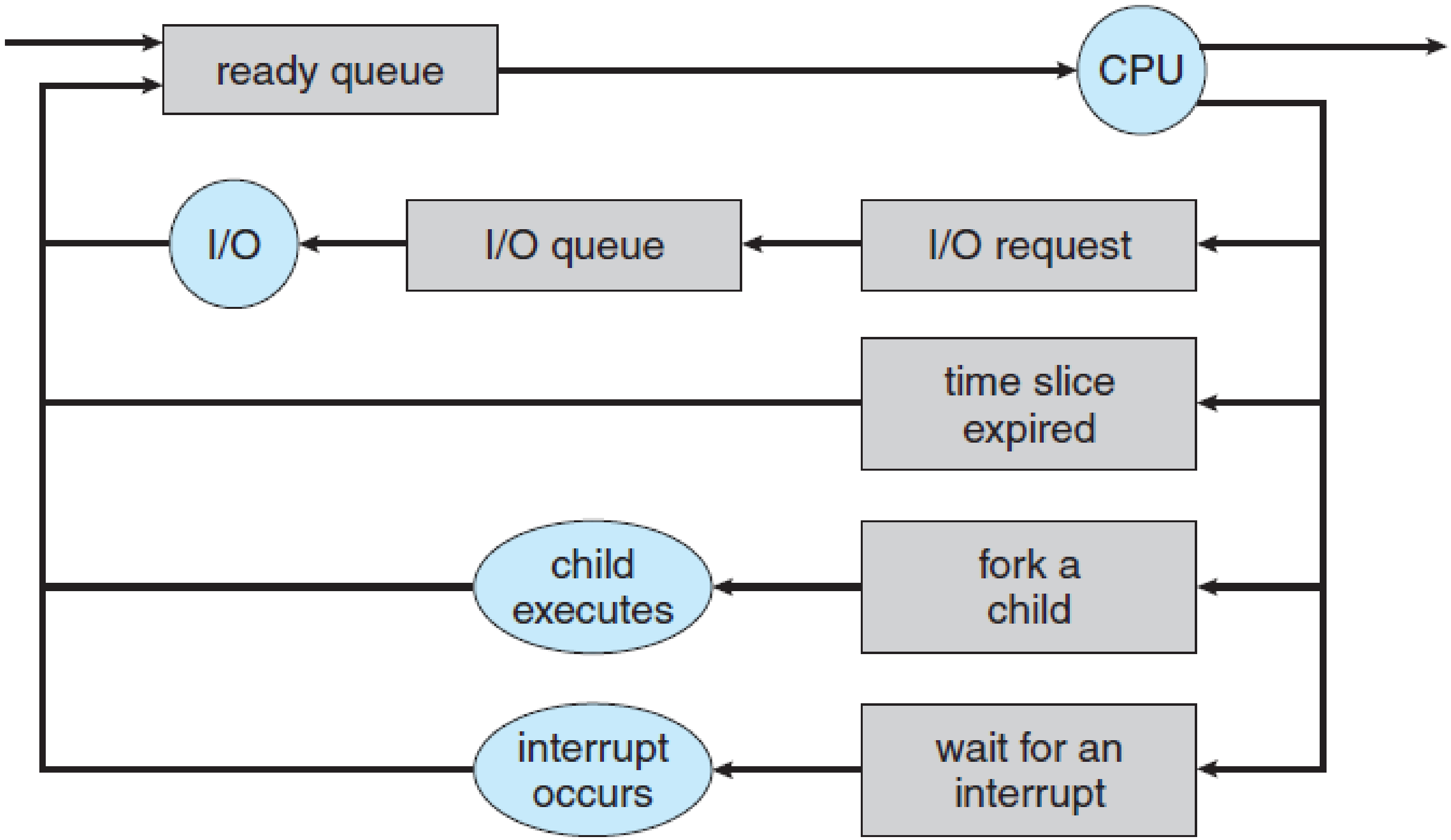


Figure 3.6 Queueing-diagram representation of process scheduling.

Scheduling Queues

- Once the process is allocated to the CPU and is executing, one of several events could occur:
  - The process could issue an **I/O request** and then be placed in an **I/O queue**.
  - The process could create a **new child process** and wait for the child's termination.
  - The process could be **removed forcibly** from the CPU, as a result of an **interrupt**, and be put back in the **ready queue**.
- In the first two cases, the process eventually switches from the **waiting state** to the **ready state** and is then put back in the **ready queue**.
- A process continues this cycle until it terminates.

Schedulers

- *A* ***process*** *migrates among the various scheduling queues throughout its lifetime*.
  - The OS must select, for **scheduling purposes**, processes from these queues in some fashion.
  - The selection process for execution is carried out by the appropriate **scheduler**.

Long-term and Short-term schedulers

- In a **batch system**, more processes are submitted than can be executed immediately (by a single processor).
  - These processes are **pooled to a mass-storage device** , where they are kept for later execution.
- A **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution.
- A **short-term scheduler**, or **CPU scheduler**, selects the processes from the **ready queue** that are ready to execute and allocates CPU to one of them.
  - **The primary distinction between these two schedulers lies in frequency of execution**.

Short-term schedulers

- The **short-term scheduler** selects a new process from the **ready queue** for the CPU frequently.
  - A process may execute for only a few milliseconds before waiting for **an I/O request**.
  - Often, the short-term scheduler executes at least once every 100 milliseconds.
  - If it takes 10 milliseconds to decide to execute a process for 100 milliseconds (total is110ms), then 10/(110) = 9% of the CPU is being used (wasted) simply for scheduling the work.

Long-term schedulers

- The **long-term scheduler** executes much **less frequently.**
- The long-term scheduler controls the **degree of multiprogramming** (*the number of processes in memory*).
- Because of the longer interval between executions, the **long-term scheduler** can afford to take more time to decide which process should be selected for execution.

I/O-bound process and CPU-bound process

- In general, most processes can be described as either **I/O bound** or **CPU bound**.
  - An **I/O-bound process** is one that spends most of its time doing **I/O** than it spends doing computations.
  - A **CPU-bound process,** generates less I/O requests and using more of its time doing computations.

```
{
printf("\nEnter the first integer: ");
scanf("%d", &a);
printf("\nEnter the second integer: ");
scanf("%d", &b);
```

I/O cycle

```
c = a+b
d = (a*b)-c
e = a-b
f = d/e
```

CPU cycle

```
printf("\n a+b= %d", c);
printf("\n (a*b)-c = %d", d);
printf("\n a-b = %d", e);
printf("\n d/e = %d", f);
}
```

I/O cycle

I/O-bound process and CPU-bound process

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

Medium-term scheduler (1)

- Some operating systems, such as *time-sharing systems*, may have intermediate level of scheduling, **medium-term scheduler** is shown in **Figure 3.7**.
  - The key idea behind a **medium-term scheduler** is that **it can remove a running process into a special memory** and thus *reduce the degree of multiprogramming.*
  - Later, the process can be reintroduced into ready queue (main memory), and its execution can be continued where it left off.
  - This scheme is called **swapping.** The process is **swapped out**, and is later **swapped in**, by the **medium-term scheduler**.
- **Swapping** is necessary to be free the memory for the **priority** or **pre-empted** process.

Medium-term scheduler (2)

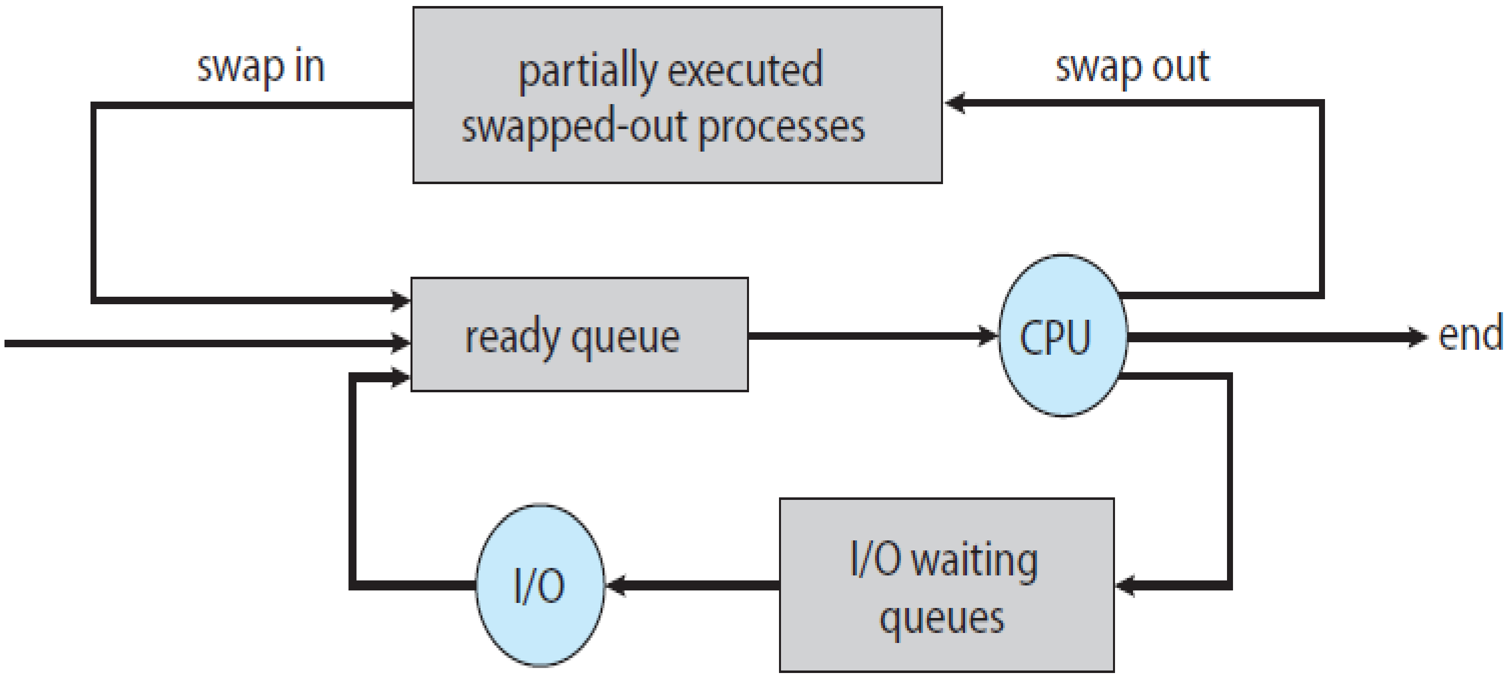


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

Context Switch (1)

- When an **interrupt** occurs, the OS needs to save the current **context** of the process running on the CPU so that it can restore that context when its processing is done.
  - The context is represented in the **PCB** of the process.
  - It includes the value of the **CPU registers**, the **process state** (see **Figure 3.2**), and **memory-management information**.
- **Switching** the CPU to another process requires performing a **state save** of the current process and a state restore of a different process. This task is known as a **context switch.**
  - When a **context switch** occurs, the **kernel** saves the *context of the old process* in its **PCB** and loads the saved context of the new process scheduled to run.
  - **Context-switch time** is pure overhead, because the system does no useful work while switching.

Context Switch (3)

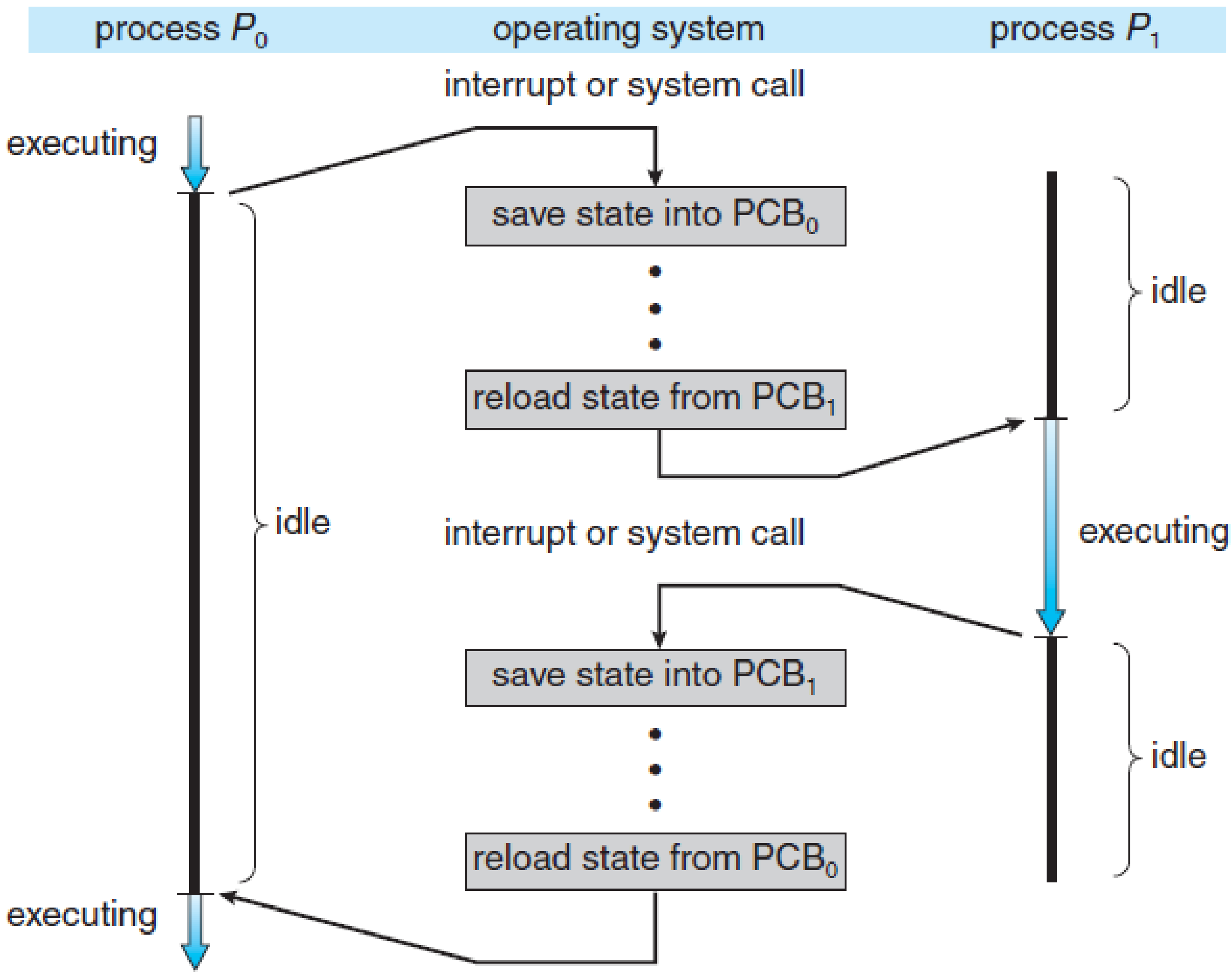


Figure 3.4 Diagram showing CPU switch from process to process.

Process Creation (1)

- During the course of execution, a **process** may create several new processes (called **spawning**).
- The creating process is called a **parent process**, and the new processes are called the **children processes**.
  - Forming a **tree** of processes.
  - Most operating systems including **UNIX**, **Linux**, and **Windows** identify processes according to a unique **process identifier** (or **pid**),
- The **pid** provides a unique value for each process in the system,
  - used as an index to access various attributes of a process within the kernel.

Process Creation (2)

- **Figure 3.8** illustrates a typical process tree for the **Linux** operating system, showing the name of each process and its pid.

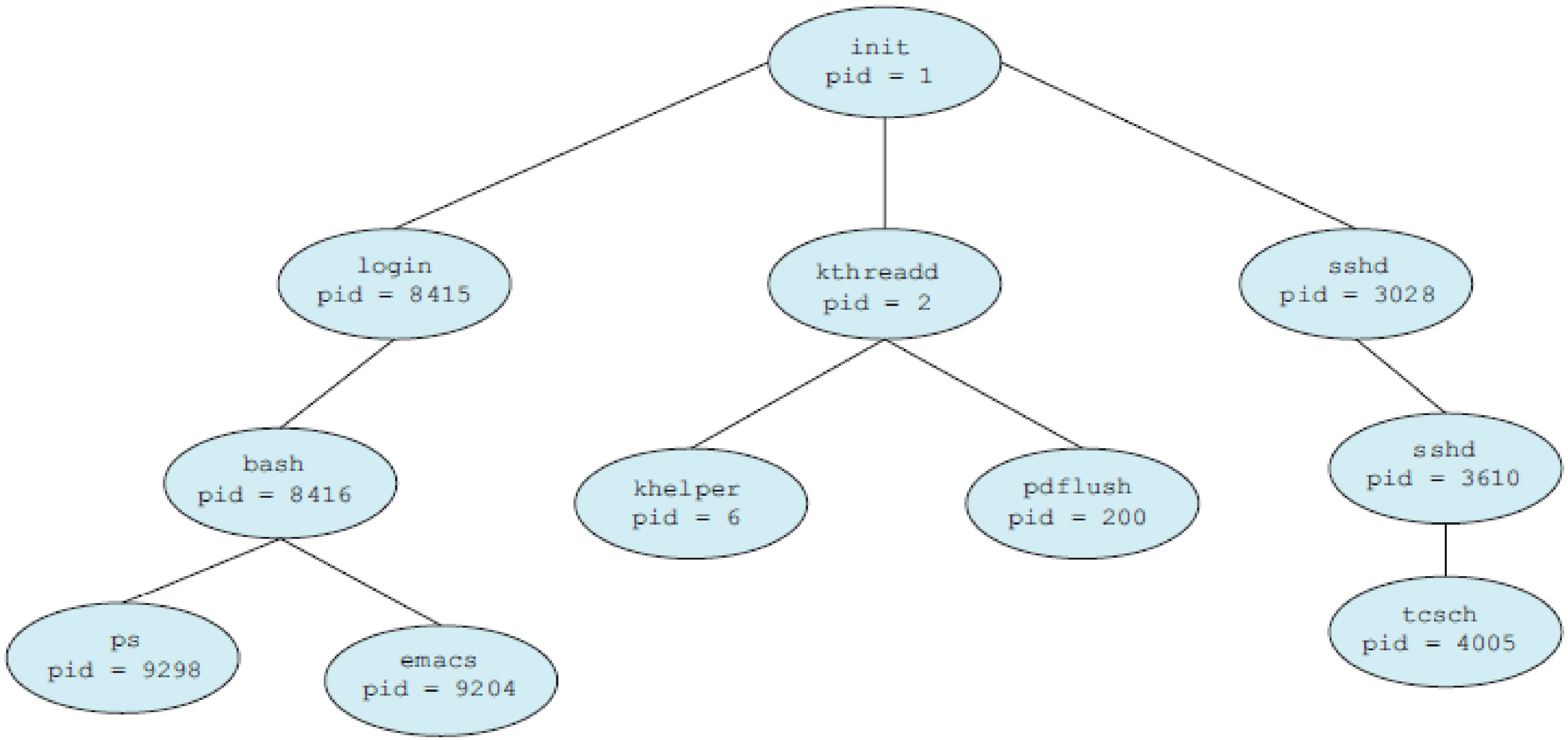


Figure 3.8 A tree of processes on a typical Linux system.

Process Creation (3)

- The **init** process (which always has a pid of 1) serves as the root parent process for all user processes.
- Once the system has booted, the **init** process can also create various user processes, such as a web or print server, an **ssh** server, and the like.
- The **kthreadd** process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, **khelper** and **pdflush**).
- The **sshd** process is responsible for managing clients that connect to the system by using **ssh** (which is short for ***secure shell***).
- The **login** process is responsible for managing clients that directly log onto the system.
- In this example, a client has logged on and is using the **bash** shell, which has been assigned pid 8416.
- Using the **bash** command-line interface, this user has created the process **ps** as well as the **emacs** editor.
- On UNIX and Linux systems, we can obtain a listing of processes by using the **ps** command. For example, the command **ps -el**

Process Termination

- Process executes last statement and then asks the OS to delete it using the **exit()** system call.
  - Returns status data from child to parent (via `wait()`)
  - Process' resources are de-allocated by operating system
- Parent may terminate the execution of children processes using the **abort()** system call. 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.
  - A process can cause the termination of another process via an appropriate system call (for example **TerminateProcess()** in **Windows**).

Independent and Cooperating processes

- Processes executing concurrently in the OS may be either ***independent*** *processes* or ***cooperating processes***.
  - A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system.
    - Any process that does not share data with any other process is independent.
  - A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system.
    - Clearly, any process that shares data with other processes is a **cooperating process.**

Process Cooperation

- There are several reasons for providing an environment that allows **process cooperation**:
  - **Information sharing**: several users may be interested in the same piece of information (for instance, a shared file), system must allow concurrent access to such information.
  - **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks and allow them to execute in parallel.
  - **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
  - **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

Inter-process Communication (IPC)

- **Cooperating processes** require an **inter-process communication (IPC) mechanism** that will allow them to exchange data and information.
- There are two fundamental models of IPC:
  - **shared memory:** In the shared-memory model, a region of memory that is shared by cooperating processes is established.
    - Processes can then exchange information by reading and writing data to the shared region.
  - **Message passing:** In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.
- The two communications models are contrasted in **Figure 3.12**.

Interprocess Communication

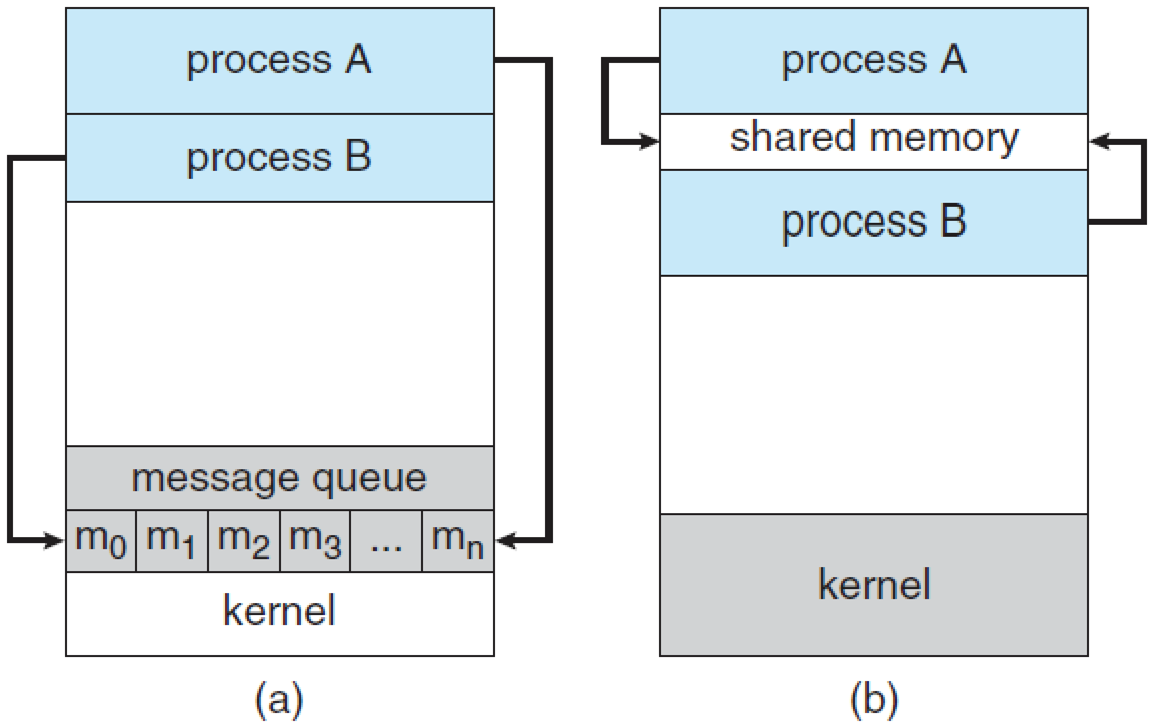


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

Interprocess Communication

- **Message passing** is useful for exchanging smaller amounts of data, because no conflicts need be avoided.
  - It is easier to implement in a **distributed system** than shared memory.
  - It needs **system calls** to implement .
  - Hence it requires more time-consuming task of kernel intervention.
- **Shared memory** can be faster than message passing, since its does not need system calls to implement.
  - Here **system calls** are required only to establish **shared memory regions**.
    - Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Shared-Memory Systems

- The OS tries to prevent one process from accessing another process's memory.
  - **Shared memory** requires that two or more processes agree to remove this restriction.
  - They can then exchange information by reading and writing data in the shared areas.
  - The form of the data and the location are determined by these processes and are not under the operating system's control.
  - The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Shared-Memory Systems

- To illustrate the concept of **cooperating processes**, let's consider the **producer–consumer** problem, which is a common paradigm for cooperating processes.
  - A **producer process** produces information that is consumed by a consumer process.
    - For example, a **compiler** may produce **assembly code** that is consumed by an **assembler** section of the compiler.
    - The assembler, in turn, may produce **object modules** that are consumed by the **loader** section of the assembler.
- The **producer–consumer problem** also provides a useful metaphor for the **client–server** paradigm.

producer–consumer problem

- One solution to the **producer–consumer** problem uses **shared memory**.
  - To allow producer and consumer processes to run concurrently, we must have available **a buffer** of items that can be filled by the producer and emptied by the consumer.
  - This **buffer** will reside in a region of memory that is **shared** by the **producer** and **consumer** processes.
  - A **producer** can produce one item while the **consumer** is consuming another item.
  - The **producer** and **consumer** must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

producer–consumer problem

- Two types of buffers can be used:
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size
- In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.
- The code for the producer process is shown in **Figure 3.13**, and the code for the consumer process is shown **in Figure 3.14.**

producer–consumer problem

```
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;
}
```

Figure 3.13 The producer process using shared memory.

producer–consumer problem

```
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 */
}
```

Figure 3.14 The consumer process using shared memory.

Message Passing

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

Message Passing (Cont.)

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

Message Passing (Cont.)

- Implementation of communication link
  - **Physical:**
    - Shared memory
    - Hardware bus
    - Network
  - **Logical:**
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Direct Communication

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

Indirect Communication - Mailbox

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

- Operations
  - **create** a new mailbox (port)
  - **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

Indirect Communication

- **Mailbox sharing**
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- **Solutions**
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver.
    - Sender is notified who the receiver was.

Synchronization

- Communication between processes takes place through calls to **send()** and **receive()** primitives.
- There are different design options for implementing each primitive. 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
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Synchronization (Cont.)

- **With message passing, Producer-consumer becomes trivial**

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

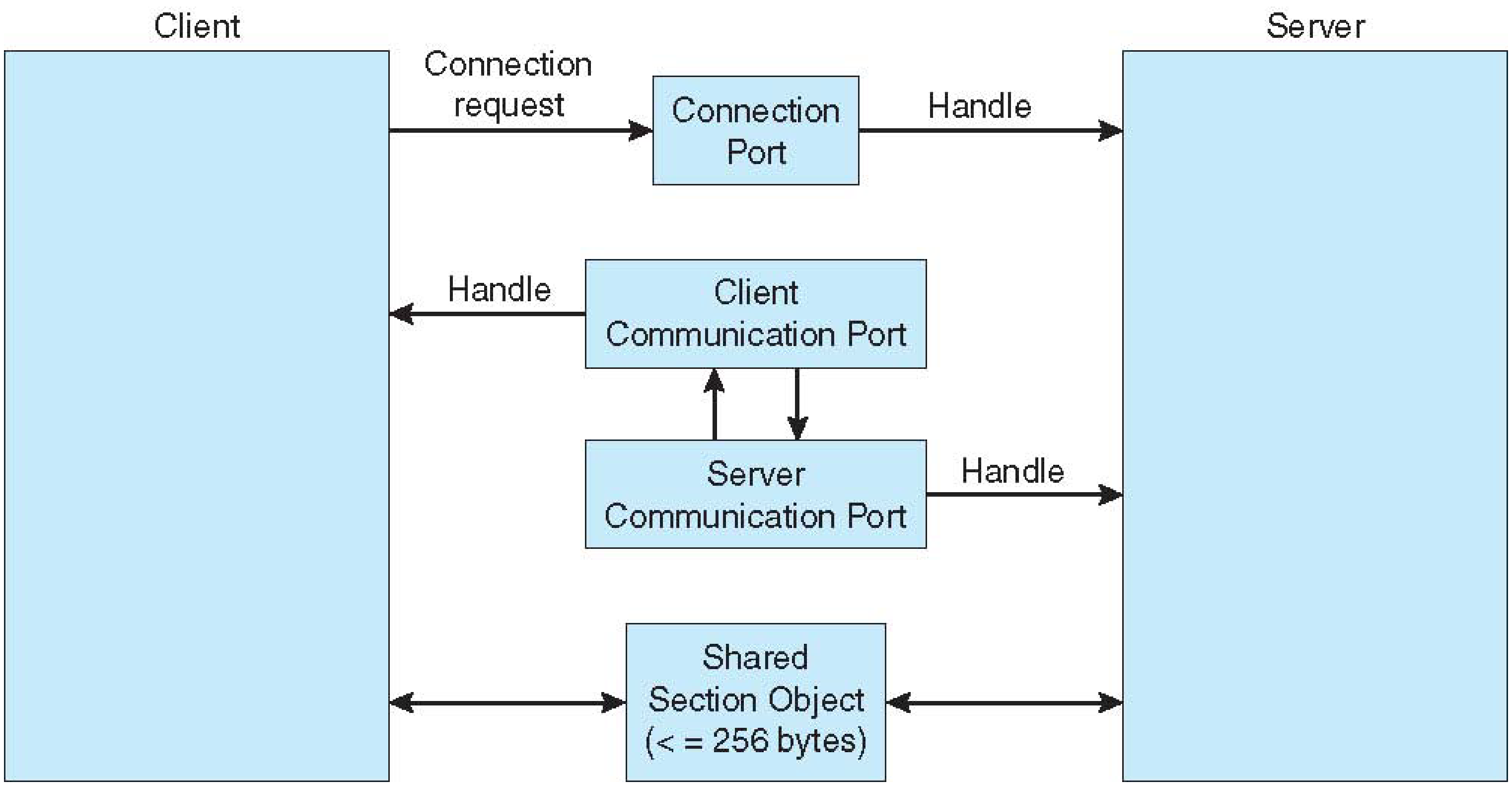
```
message next_consumed;
while (true) {
   receive(next_consumed);

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

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



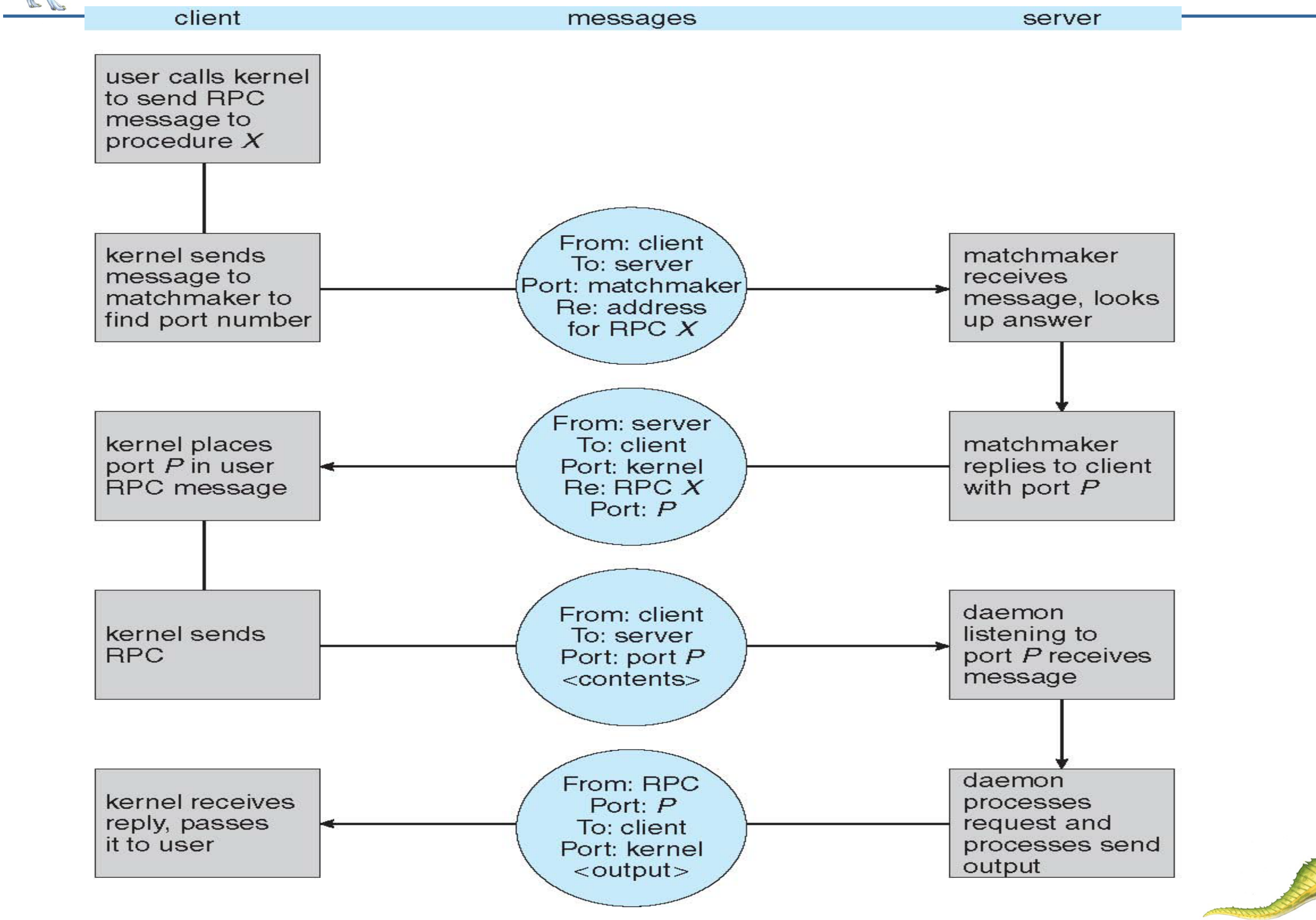
Remote Procedure Calls

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

Remote Procedure Calls (Cont.)

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

Execution of RPC



Programming Exercises

Show **thread creation** in Java using *Executor* and *Runnable* objects.

End of Chapter 3