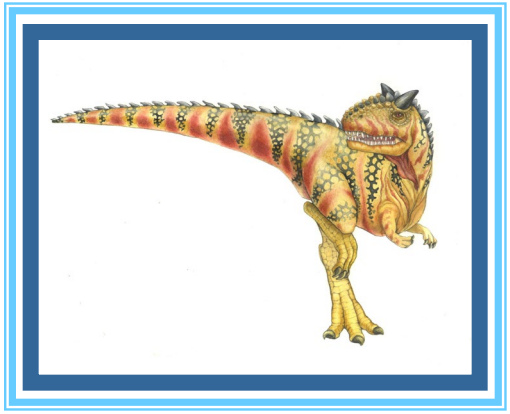
Chapter 3: Processes



Process Concept

- Early computers allowed only **one program to be executed at a time**.
  - This program had complete control of the system and access to all its ***resources***.
- Contemporary computer systems allow **multiple programs** to be **loaded into memory** and executed concurrently on a single CPU system.
- This evolution resulted in the notion of a **process**
  - which is a program in execution (means *executable file of the program in the main memory, the CPU scheduling algorithm controls that*).
  - *A **process** is the unit of work in a modern computing system*.

Process Concept

- Therefore, a system consists of a collection of **processes**, some *executing user code*, *others executing operating system code*.
- Potentially, all these **processes** can execute concurrently, with a single CPU (or multiple CPUs).
- 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

- **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 **text section** is the **.exe file**
  - **Stack** stores *temporary data when invoking functions*
    - 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

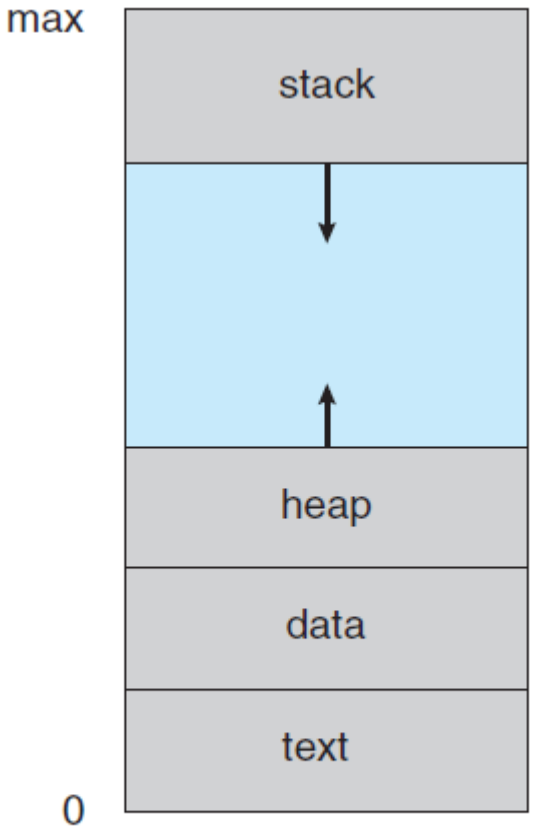


Figure 3.1 Process in memory.

The Process

- Notice that the sizes of the **text (.exe file** section) and **data** sections of the memory are **fixed**, as their sizes do not change during **program run time**.
- The **stack** and **heap** sections can shrink and grow dynamically during **program execution (run time)**.
  - The **heap** will grow as memory is *dynamically allocated*, and will shrink when memory is *returned to the system*.
- Each time a **function is called**, an **activation record** containing *function parameters*, *local variables*, and the *return address* is pushed onto the **stack**;
  - *When control is returned from the function, the **activation record** is popped from the stack*.

The Process

- **A processor register** that supports program execution is the **program counter** register.
- It is emphasized that *a program by itself is not a process*.
- A **program** is a ***passive*** entity, such as a file containing a list of instructions stored on **disk** (often called an .**exe file**).
- A **process** is an ***active*** entity, with a **program counter** specifying the next instruction to execute and a set of associated resources.
  - A **program becomes a process** when an **executable file** is loaded into memory.
  - Two common techniques for **loading executable files** are double-clicking an .**exe icon** of the program on a disk drive or entering the name of the **executable file** on the command line.

The Process

- 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 invokes many copies of the web browser, which are **separate processes**.
- A process can itself be an execution environment for other code.
  - The **Java programming environment** provides a good example.
  - In most cases, a Java program runs within the **Java Virtual Machine (JVM)**.
  - The **JVM** executes as a process that interprets the **loaded Java code** and takes actions (via native machine instructions) on its behalf.
  - For example, to run the compiled Java program `Program.class`, we would enter: `java Program`

Process State

- 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

- It is important to realize that only **one process can be *running* on any processor core at any instant**. Many processes may be ***ready*** and ***waiting***, however.
- The state diagram corresponding to these states is presented in **Figure 3.2.**

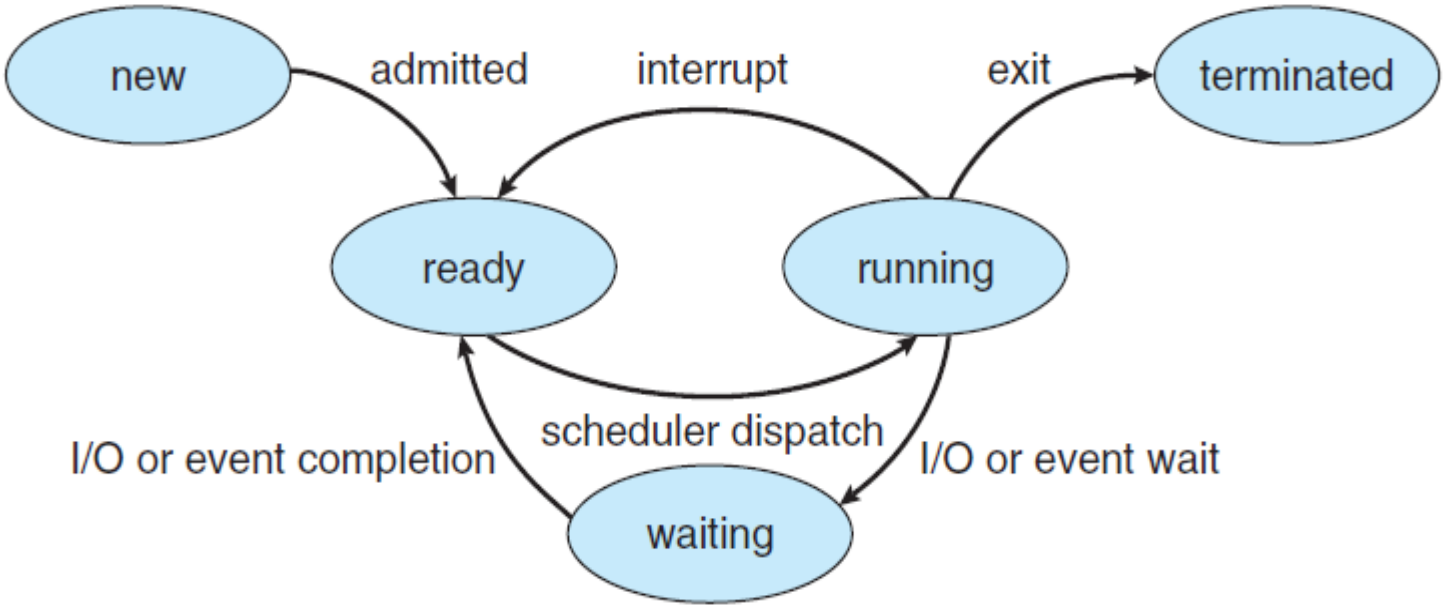


Figure 3.2 Diagram of process state.

Process Control Block

- Each process is represented in the **OS** by a **process control block** (**PCB**).
- A **PCB** is shown in **Figure 3.3.**
- In brief, the **PCB** serves as the repository for all the data needed to start, or restart, a process, along with some accounting data.

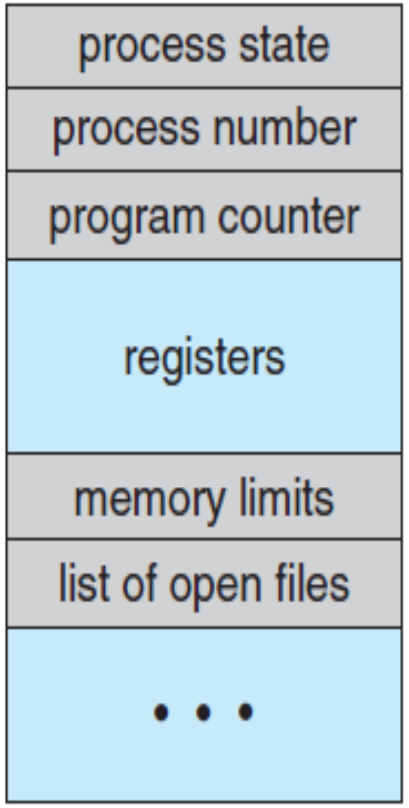


Figure 3.3 Process control block (PCB).

Process Control Block

- A **PCB** contains many pieces of information associated with a specific **process**:
  - **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 depends 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.

Process Scheduling

- The **objective of multiprogramming** is to have some process always running to **maximize CPU utilization**.
- The **objective of time sharing** (**Round Robin** method) is to ***switch*** a CPU core among processes so frequently that users can interact with each program while it is running (see **Figure 3.4**).
  - To meet these objectives, the **CPU scheduler** selects a set of several available processes from the **Ready Queue** for program execution on **a core**.
  - For a system with a **single CPU core**, there will be **one process** running at a time.
  - *The number of processes currently in memory is known as the **degree of multiprogramming***.

A CPU Switches from Process to Process

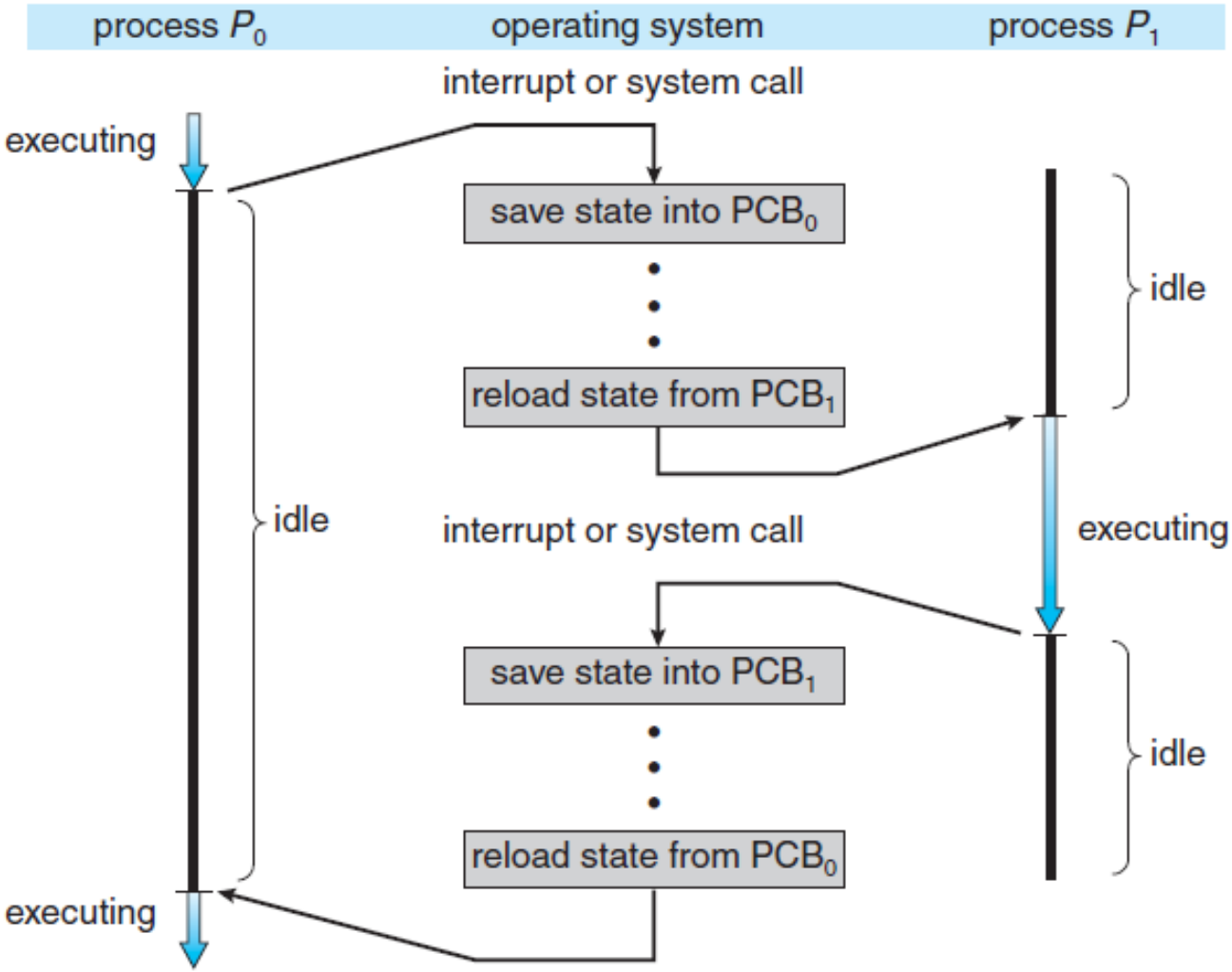


Figure 3.4 Diagram showing CPU switch from process to process.

Threads

- The process model discussed so far has implied that a process is a program that performs a single **thread** of execution.
- This **single-threaded control** allows the process to perform only one task at a time.
- Most modern operating systems have extended the **process concept to multiple threads of execution**, enabling it to perform more than one task at a time.
- This feature is beneficial only on **multicore systems**, where multiple threads can run in parallel.
- A **multithreaded word processor** could assign one thread to manage user input while another thread runs the spell checker.
- On systems that support **thread execution**, the **PCB** is expanded to include information for each thread called a **TCB** (thread control block).
  - Other changes throughout the system are also needed to support threads. **Chapter 4 explores threads in detail**.

Scheduling Queues

- As processes are created, they are put into a **ready queue**, where they are **ready and waiting to execute on a CPU's core**.
  - This queue is generally stored as a **linked list**; a **ready-queue header contains pointers to the first PCB in the list**, and each **PCB includes a pointer field that points to the next PCB in the ready queue** (**Figure 3.4**).
  - When a process is allocated a **CPU core**, it executes for a while. Eventually, it terminates, is interrupted, or waits for a particular event, such as the completion of an **I/O request**.
  - Suppose the process makes an **I/O request** to a device such as a disk.
  - Since **I/O devices** run significantly slower than processors, the process will have to **wait for the I/O** to become available. Processes that are waiting for a specific event — such as **I/O completion** — are placed in a **wait queue** (**Figure 3.4**).

Scheduling Queues

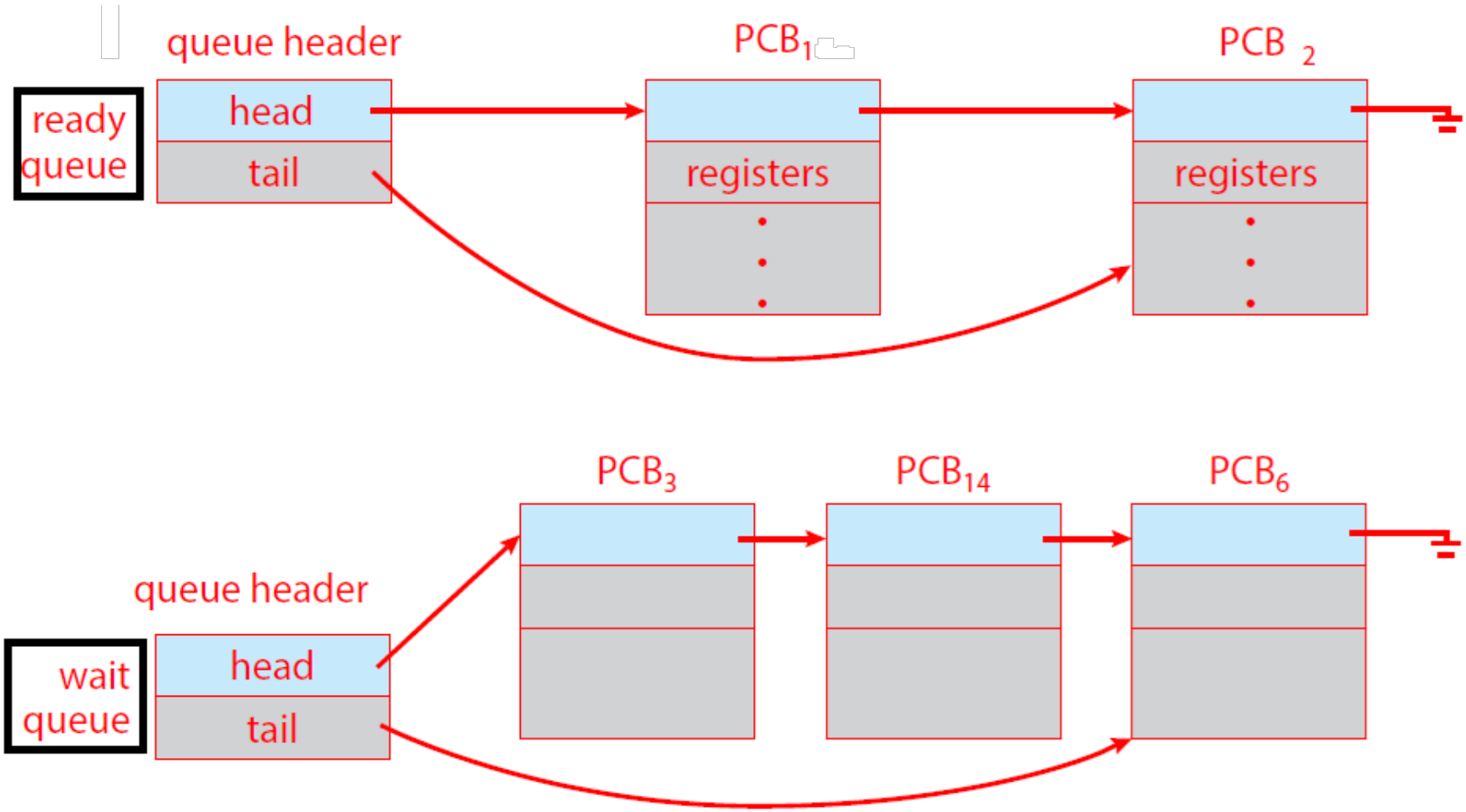


Figure 3.4 The ready queue and wait queues.

Scheduling Queues

- A common representation of process scheduling is a **queueing diagram** (in **Figure 3.5**).
- **Two types of queues** are present: the **ready queue** and a set of **wait 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

- Once the **process is allocated a CPU core**, one of several events could occur:
  - The process could issue an **I/O request** and then be placed in an **I/O wait queue**. The process could create a **new child process** and then be placed in a **wait queue**.
  - The process could be removed from the core, because of an **interrupt** or **its time slice expires,** and be put back in the **ready queue**.
- In the first two cases (in **Figure 3.5**), 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.**
  - The **PCB and resources** of the terminated process are **deallocated**.

Scheduling Queues

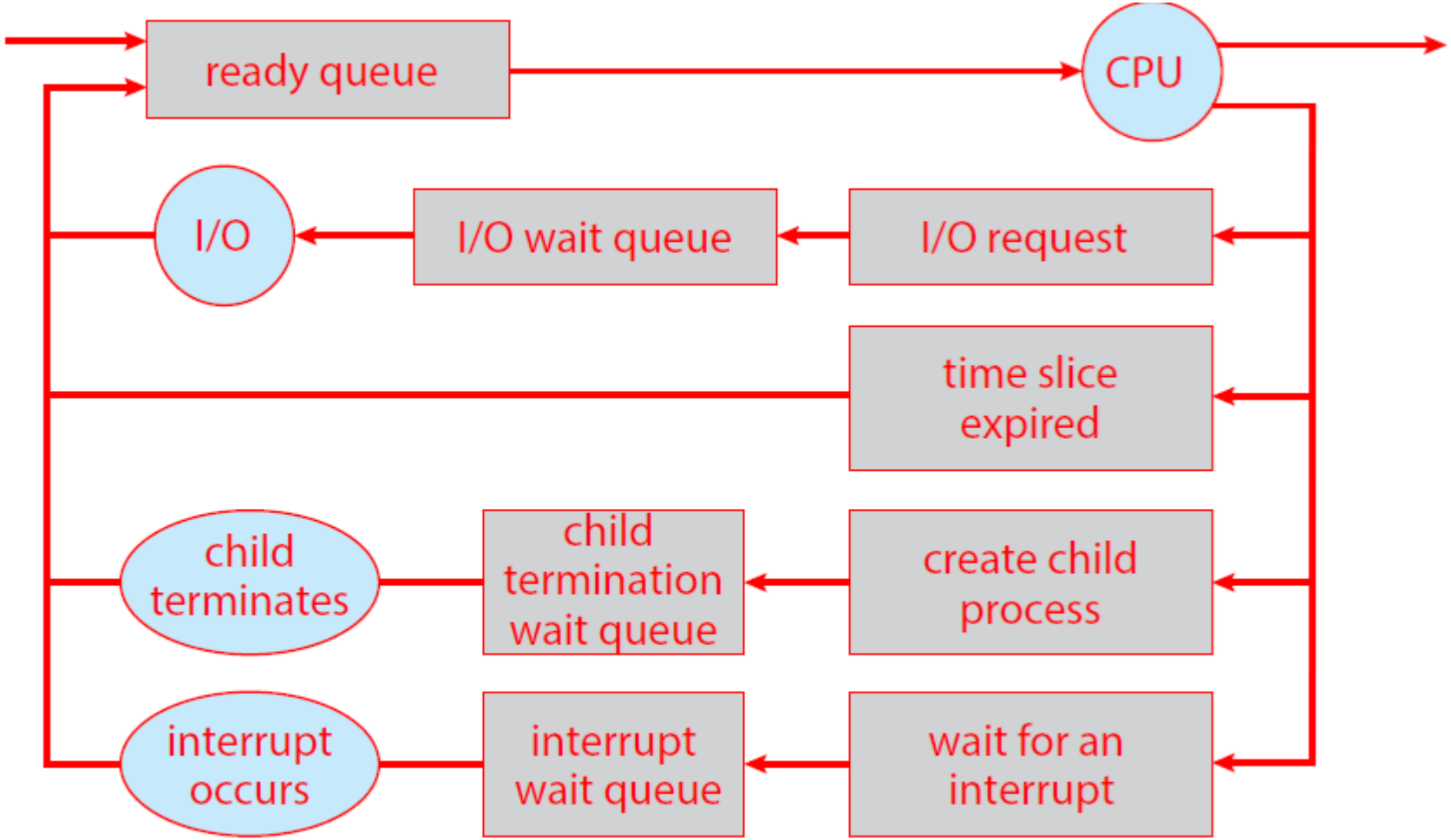


Figure 3.5 Queueing-diagram representation of process scheduling.

CPU Scheduling

- A **process** migrates among the **ready queue** and **various wait queues** throughout its lifetime.
- **The role of the CPU scheduler** is to select a process from the ready queue and allocate a CPU core to it.
- The **CPU scheduler** must frequently select **a new process** for the CPU.
- An **I/O-bound process** may execute for only a **few milliseconds** before waiting for **an I/O request**.
- Although a **CPU-bound process** will require a **CPU core,** it forcibly removes it from the **CPU** and places it in the **ready queue**, scheduling another process to run (due to the **preemptive nature** of the scheduler).

CPU Scheduling

- Therefore, the **CPU scheduler executes** at least once every **100 milliseconds**, although typically much more frequently.
  - A process may execute for only a **few milliseconds** before waiting for **an I/O request**.
  - Often, the **preemptive scheduler** executes processes at least once every 100 milliseconds.
  - If it takes **10 milliseconds** to decide to execute a process for **100 milliseconds** (total is 110ms), then 10/(110) = **9%** of the CPU is being used (wasted) simply for scheduling the work.

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

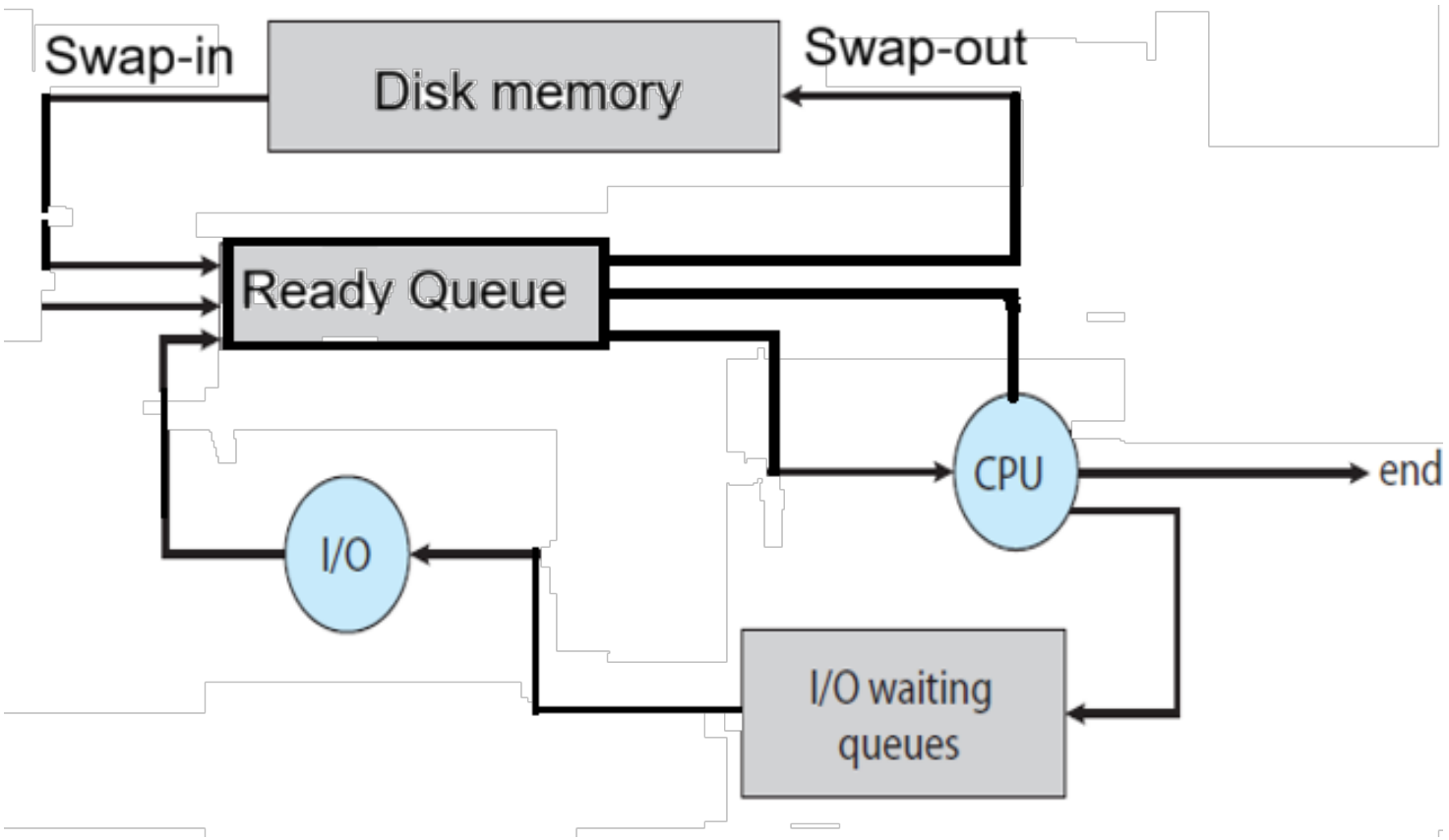
```
printf("\n a+b= %d", c);
printf("\n (a*b)-c = %d", d);
printf("\n a-b = %d", e);
printf("\n d/e = %d", f);
}
```

} I/O cycle

Process Swapping

- Some operating systems have an **intermediate form of scheduling**, known as **swapping**
  - The key idea of **swapping** is to remove a ***process*** from memory (**ready queue**) to **disk memory** (**virtual memory**) and thus *reduce the degree of multiprogramming*.
  - Later, the process can be reintroduced into memory, and its execution can be continued where it left off.
  - This scheme is known as ***swapping*** because a process can be "**swapped out**" from **memory to disk**, where its status is saved, and later "**swapped in**" from **disk back to memory**, where its status is restored.
  - *Swapping is typically only necessary when memory has been overcommitted and must be freed up*.

Process Swapping



Context Switch

- **Interrupts** cause the operating system to change a **CPU core** from its current task and to run a **kernel routine**.
  - ***Such operations happen frequently on general-purpose systems***.
  - When **an interrupt occurs**, *the system needs to save the current* ***context*** *of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it*.
- The **context of a process** is represented in the form of a **PCB** of the OS.
  - PCB includes the value of the **CPU registers**, **the process state** (see Figure 3.2), and **memory-management information**.
  - A **state save** of the **current state of the CPU core**, be it in **kernel or user mode**, and then a **state restore** to resume operations.

Context Switch

- **Switching the CPU core 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** and is illustrated in **Figure 3.6**.
- **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.
- **Switching speed** varies from machine to machine, depending on the ***memory speed***, ***the number of registers*** that must be copied, and the existence of special instructions (such as ***load*** or ***store*** instructions).
- Context-switch times are highly dependent on **hardware support**.

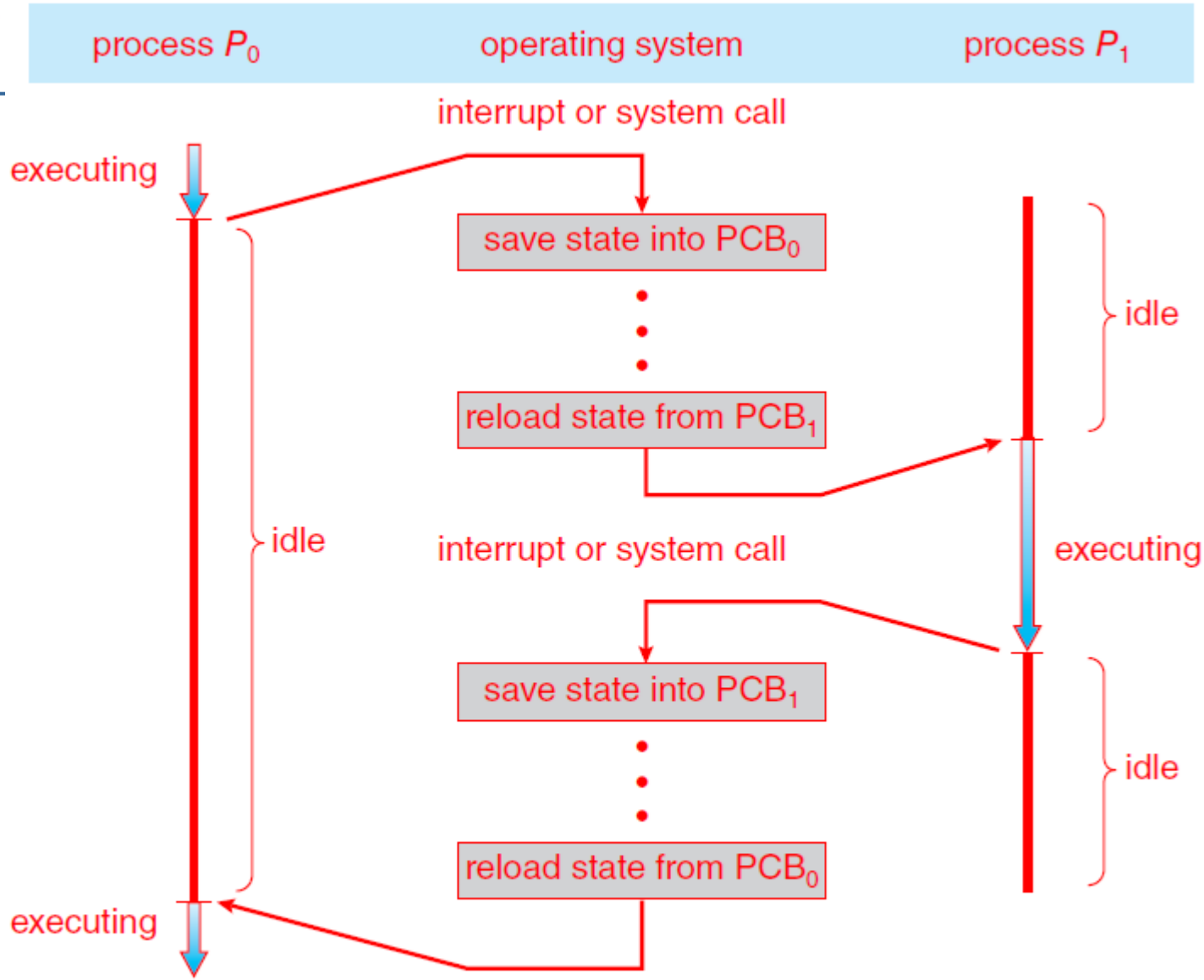


Figure 3.6 Diagram showing context switch from process to process.

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.

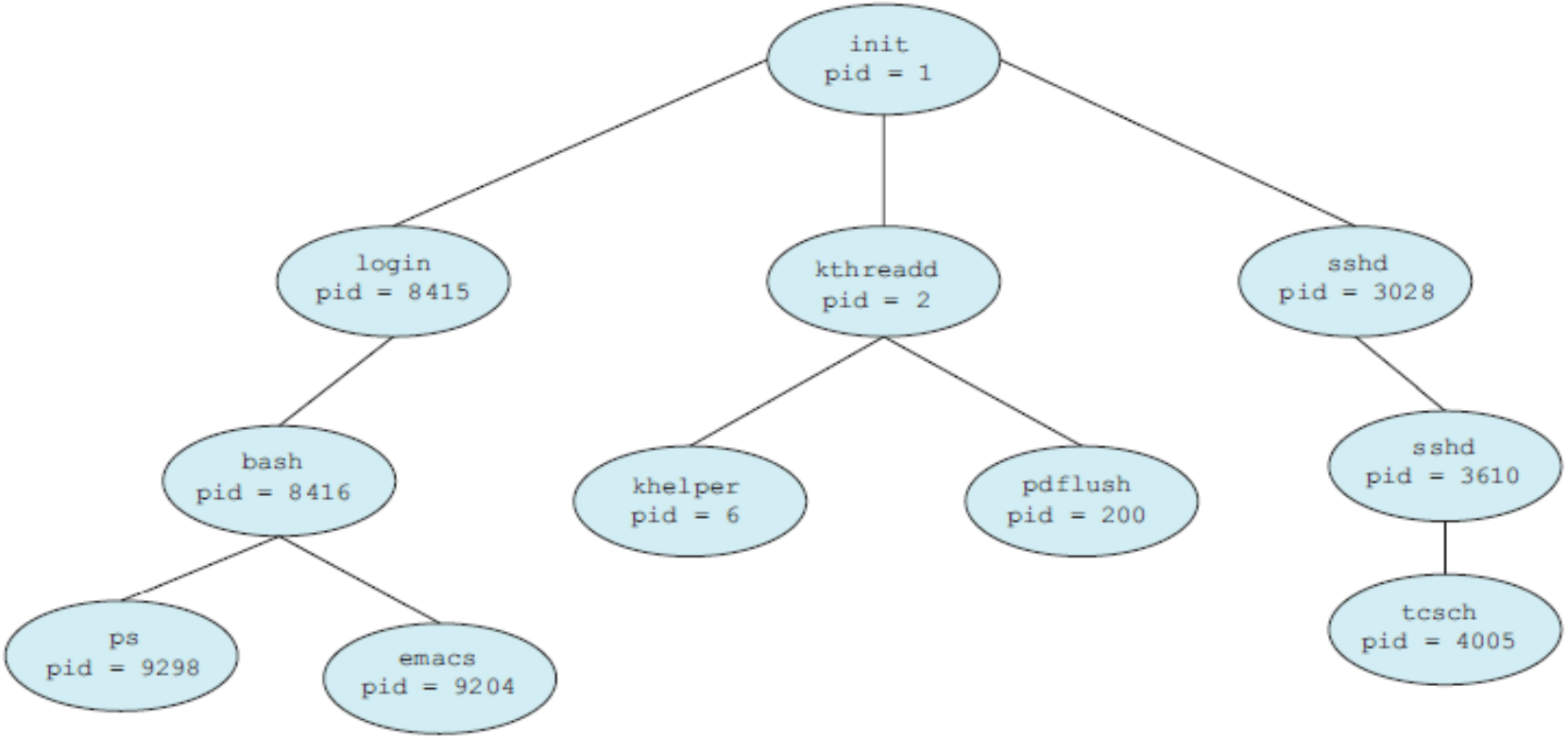


Figure 3.8 A tree of processes on a typical Linux system.

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

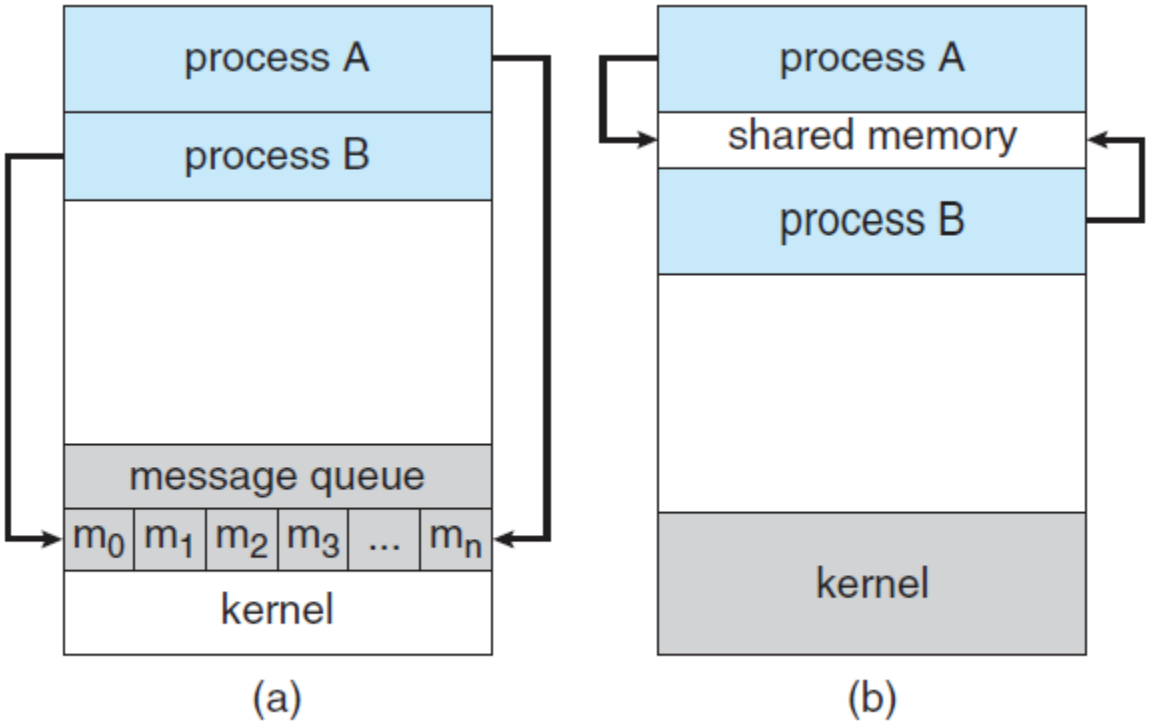


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

Interprocess Communication

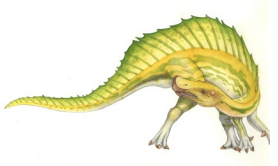
- **Message passing** is useful for exchanging smaller amounts of data, because no conflicts need be avoided.
  - It is easier to implement in a **distributed system** than shared memory.
  - It needs **system calls** to implement .
  - Hence it requires more time-consuming task of kernel intervention.
- **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.

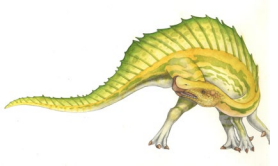
Synchronization

- 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**

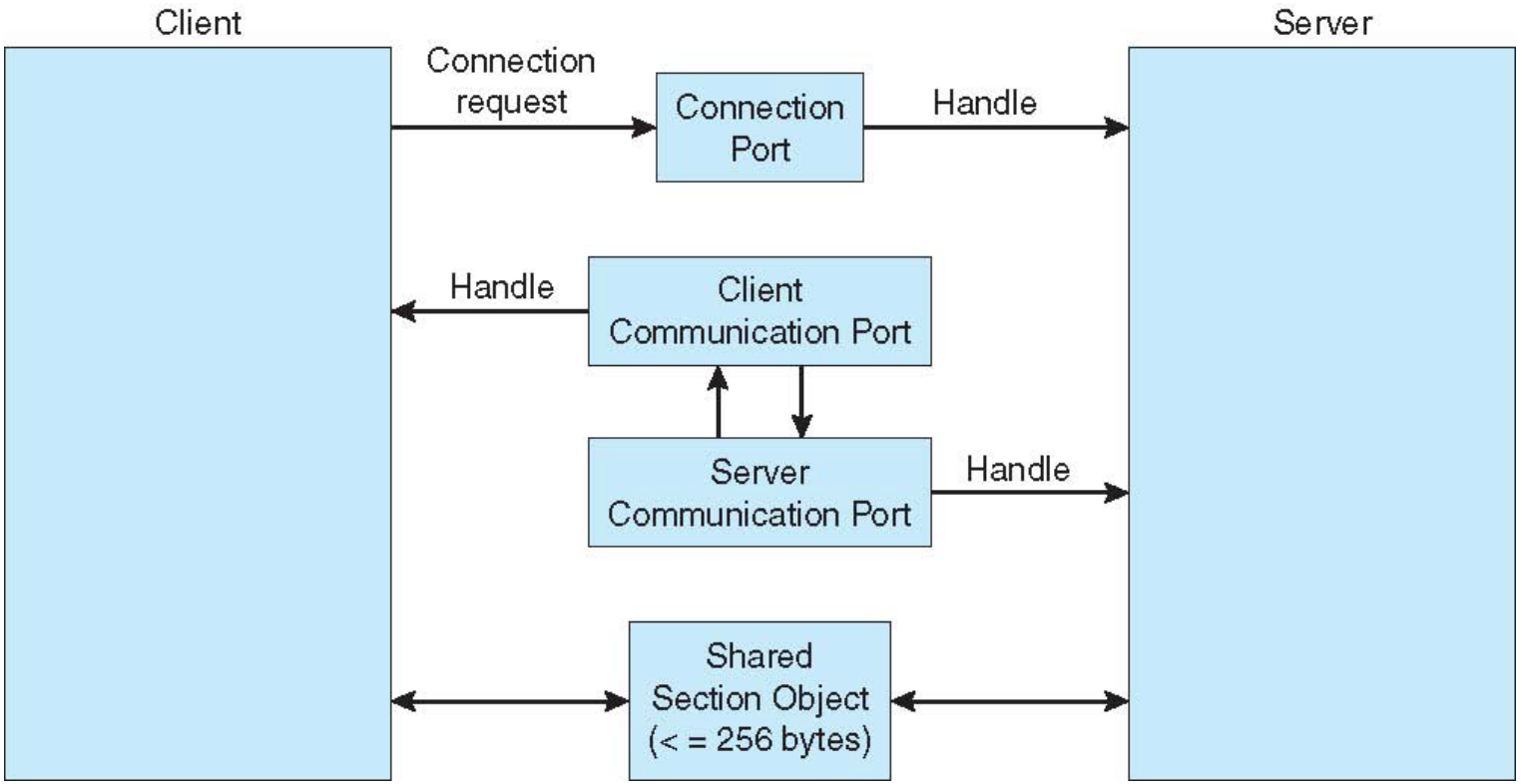
```
        message next_produced;
while (true) {
    /* produce an item in next produced */
send(next_produced);
  message next_consumed;
} while (true) {
     receive(next_consumed);

     /* 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



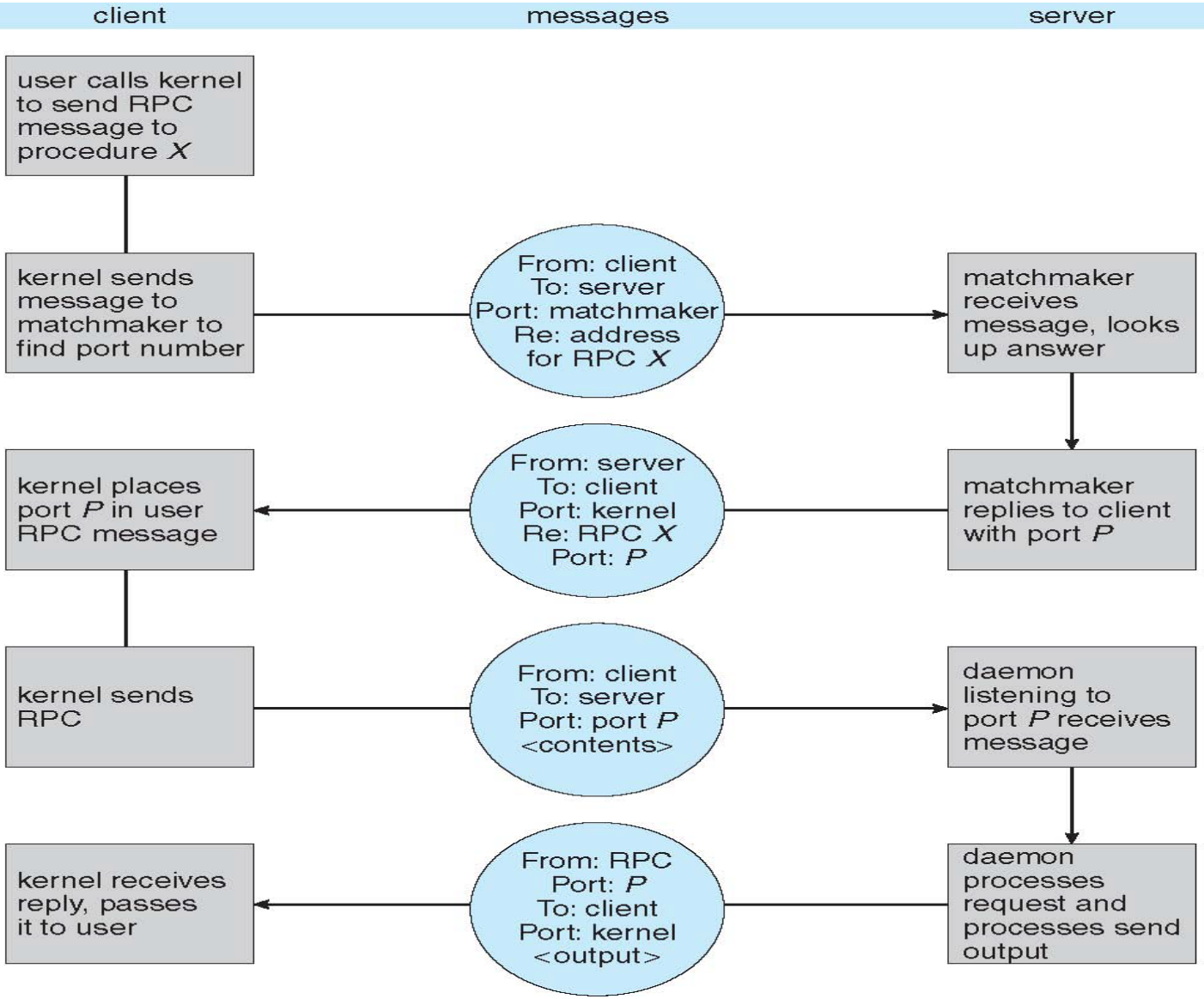
Remote Procedure Calls

- **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.