Module 2 – OS Principles

System calls, System/Application Call Interface – Protection: User/Kernel modes - Interrupts -Processes - Structures (Process Control Block, Ready List etc.), Process creation, management in Unix – Threads: User level, kernel level threads and thread models

System Calls

- System calls provide an interface to the services made available by an operating system.
- System call is a programmatic way in which a computer program requests a service from the kernel of the operating system.
- These calls are generally available as functions written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

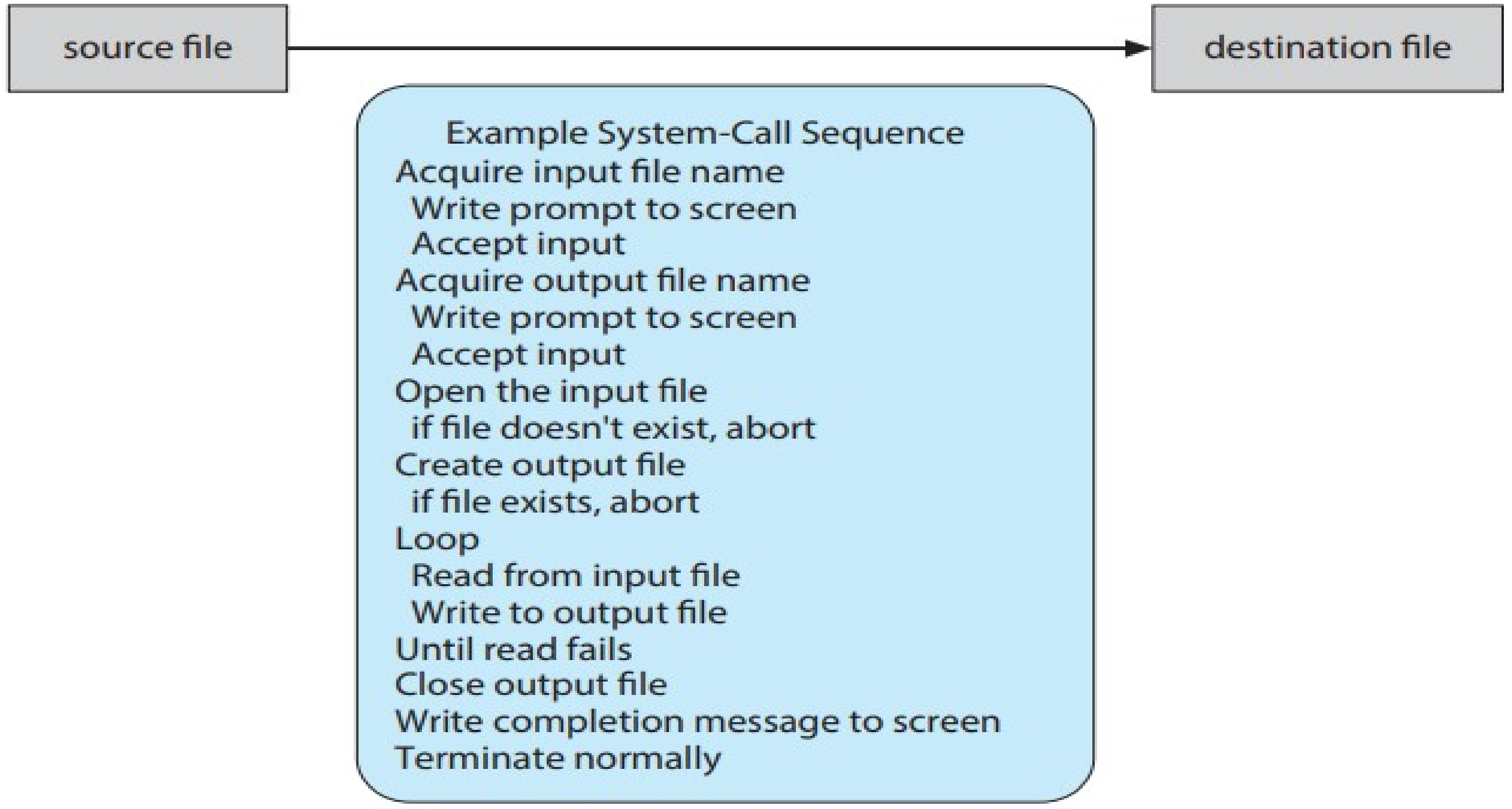
System Calls - Example

- Example to illustrate how system calls are used: Writing a simple program to read data from one file and copy them to another file.
- The first input that the program will need is the names of the two files: the input file and the output file.
- These names can be specified in many ways, depending on the operating-system design. One approach is to pass the names of the two files as part of the command— for example, the UNIX cp command:

```
cp in.txt out.txt
```

This command copies the input file in.txt to the output file out.txt.

System Calls - Example



```
Example System-Call Sequence
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Figure 2.5 Example of how system calls are used.

System Calls - Example

- A second approach is for the program to ask the user for the names.
- In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files.
- On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified.
- This sequence requires many I/O system calls. Once the two file names have been obtained, the program must open the input file and create and open the output file. Each of these operations requires another system call.
- Possible error conditions for each system call must be handled. For example, when the program tries to open the input file, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should output an error message (another sequence of system calls) and then terminate abnormally (another system call).

System Calls - Example

- Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.
- When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call).
- Each read and write must return status information regarding various possible error conditions.
- On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read (such as a parity error).
- The write operation may encounter various errors, depending on the output device (for example, no more available disk space).
- Finally, after the entire file is copied, the program may close both files (two system calls), write a message to the console or window (more system calls),

Application Programming Interface

- Application developers design programs according to an application programming interface (API).
- The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.
- Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and macOS), and the Java API for programs that run on the Java virtual machine.
- A programmer accesses an API via a library of code provided by the operating system.
- In the case of UNIX and Linux for programs written in the C language, the library is called libc. Each operating system has its own name for each system call.
- The functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Windows function CreateProcess() (which is used to create a new process) actually invokes the NTCreateProcess() system call in the Windows kernel.

Application Programming Interface

Why would an application programmer prefer programming according to an API rather than invoking actual system calls?

The reasons are:

- Portability: An application programmer designing a program using an API can expect the program to compile and run on any system that supports the same API.
- Another important factor in handling system calls is the run-time environment (RTE)— the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders.
- The RTE provides a system-call interface that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system.
- Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.

Application Programming Interface

- The caller does not know nothing about how the system call is implemented or what it does during execution.
- Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call.
- Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the RTE.
- The relationship among an API, the system-call interface, and the operating system is shown in Figure 2.6, which illustrates how the operating system handles a user application invoking the open() system call

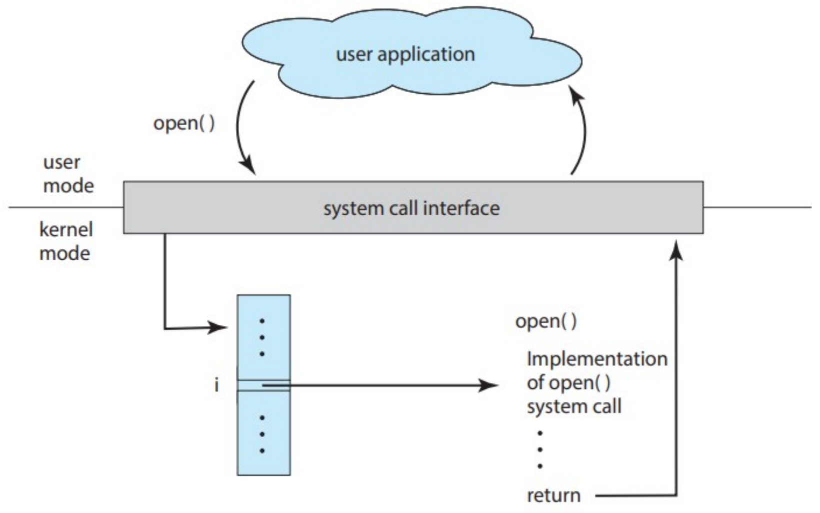


Figure 2.6 The handling of a user application invoking the `open()` system call.

Application Programming Interface

- Three general methods are used to pass parameters to the operating system.
- The simplest approach is to pass the parameters in registers.
- In some cases, however, there may be more parameters than registers.
- In these cases, the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register (Figure 2.7).
- Linux uses a combination of these approaches.
- If there are five or fewer parameters, registers are used.
- If there are more than five parameters, the block method is used.
- Parameters also can be placed, or pushed, onto a stack by the program and popped off the stack by the operating system.
- Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

Application Programming Interface

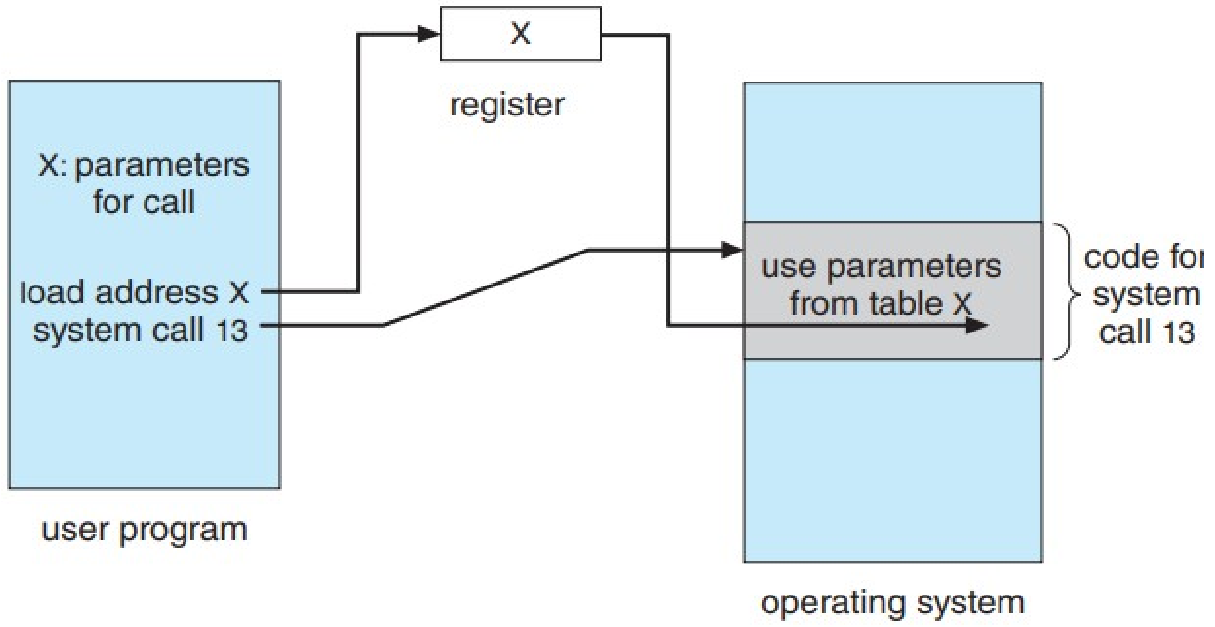


Figure 2.7 Passing of parameters as a table.

Types of System calls

System calls can be grouped into six major categories:

- Process control
- File management
- Device management
- Information maintenance
- Communications, and
- Protection

Process control

- create process, terminate process
- load, execute
- get process attributes, set process attributes
- wait event, signal event
- allocate and free memory

Process control

- ❖A running program needs to halt its execution either normally (end()) or abnormally (abort()).
- ❖If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated.
- ❖The dump is written to a special log file on disk and may be examined by a debugger—a system program designed to aid the programmer in finding and correcting errors, or bugs— to determine the cause of the problem.
- ❖Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter.
- ❖The command interpreter then reads the next command.
- ❖In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error.

Process control

- ❖A process executing one program may want to load() and execute() another program.
- ❖This feature allows the command interpreter to execute a program. For example, a user command or the click of a mouse.
- ❖An interesting question is where to return control when the loaded program terminates. This question is related to whether the existing program is lost, saved, or allowed to continue execution concurrently with the new program.
- ❖If control returns to the existing program when the new program terminates, we must save the memory image of the existing program; thus, we have effectively created a mechanism for one program to call another program.
- ❖If both programs continue concurrently, we have created a new process to be multiprogrammed. Often, there is a system call specifically for this purpose (create process()).

Process control

- ❖If we create a new process, or a set of processes, we should be able to control its execution.
- ❖This control requires the ability to determine and reset the attributes of a process, including the process's priority, its maximum allowable execution time, (get process attributes() and set process attributes()).
- ❖Having created new processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (wait time()).
- ❖We will want to wait for a specific event to occur (wait event()). The processes should then signal when that event has occurred (signal event()).
- ❖Two or more processes may share data. To ensure the integrity of the data being shared, operating systems often provide system calls allowing a process to lock shared data. Then, no other process can access the data until the lock is released. Such system calls include acquire lock() and release lock().

File management

- create file, delete file
- open, close
- read, write, reposition
- get file attributes, set file attributes

File management

- ❖The common system calls dealing with files are:

  create() and delete() files
- ❖Either system call requires the name of the file and the file’s attributes.
- ❖Once the file is created, we need to open() it and to use it.
- ❖We may also read(), write(), or reposition() (rewind or skip to the end of the file).
- ❖Finally, we need to close() the file, indicating that we are no longer using it.
- ❖We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system.
- ❖In addition, we need to determine the values of various attributes and set them if necessary.
- ❖File attributes include the file name, file type, protection codes, accounting information.
- ❖Two system calls, get file attributes() and set file attributes(), are required for this function.
- ❖Some operating systems provide many more calls, such as calls for file move() and copy().

Device management

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

Device management

- A process may need several resources to execute—main memory, disk drives, access to files, etc.
- If the resources are available, they can be granted, and control can be returned to the user process.
- Otherwise, the process will have to wait until sufficient resources are available.
- The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files).
- A system with multiple users may require us to first request() a device, to ensure exclusive use of it.
- After we are finished with the device, we release() it.
- Once the device has been requested (and allocated to us), we can read(), write(), and (possibly) reposition() the device.
- A set of system calls is used on both files and devices. Sometimes, I/O devices are identified by special file names, directory placement, or file attributes.

Information Maintenance

- get time or date, set time or date
- get system data, set system data
- get process, file, or device attributes
- set process, file, or device attributes

Information Maintenance

- ❖Many system calls exist for the purpose of transferring information between the user program and the operating system.
- ❖For example, most systems have a system call to return the current time() and date().
- ❖Other system calls may return information about the system, such as the version number of the operating system, the amount of free memory or disk space.
- ❖Another set of system calls is helpful in debugging a program. Many systems provide system calls to dump() memory. This provision is useful for debugging.
- ❖The program strace, which is available on Linux systems, lists each system call as it is executed.
- ❖Even microprocessors provide a CPU mode, known as single step, in which a trap is executed by the CPU after every instruction. The trap is usually caught by a debugger.
- ❖Many operating systems provide a time profile of a program to indicate the amount of time that the program executes at a particular location or set of locations.
- ❖A time profile requires either a tracing facility or regular timer interrupts. At every occurrence of the timer interrupt, the value of the program counter is recorded.
- ❖With sufficiently frequent timer interrupts, a statistical picture of the time spent on various parts of the program can be obtained.
- ❖ In addition, the operating system keeps information about all its processes, and system calls are used to access this information.

Communication

- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices

Communication

- ❖There are two common models of interprocess communication:

  the message passing model and the shared-memory model
- ❖In the message-passing model, the communicating processes exchange messages with one another to transfer information.
- ❖Messages can be exchanged between the processes either directly or indirectly through a common mailbox.
- ❖Before communication can take place, a connection must be opened.
- ❖The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network.
- ❖Each computer in a network has a host name. A host also has a network identifier, such as an IP address.
- ❖Similarly, each process has a process name, and this name is translated into an identifier by which the operating system can refer to the process.
- ❖The get hostid() and get processid() system calls do this translation.
- ❖The identifiers are then passed to the generalpurpose open() and close() calls provided by the file system or to specific open connection() and close connection() system calls, depending on the system's model of communication.
- ❖The recipient process usually must give its permission for communication to take place with an accept connection() call.

Communication

- ❖In the shared-memory model, processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes.
- ❖Normally, the operating system 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 is determined by the processes and is not under the operating system’s control.
- ❖The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Protection

- get file permissions
- set file permissions

Protection

- Protection provides a mechanism for controlling access to the resources provided by a computer system.
- Typically, system calls providing protection include set permission() and get permission(), which manipulate the permission settings of resources such as files and disks.
- The allow user() and deny user() system calls specify whether particular users can—or cannot—be allowed access to certain resources.

PROCESS CONCEPT

- A process is a program in execution.
- The status of the current activity of a process is represented by the value of the program counter and the contents of the processor's registers.

Difference between program and process

A program is a passive entity, such as a file containing the list of instructions stored on disk, whereas 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 icon representing the executable file and entering the name of the executable file on the command line.

PROCESS CONCEPT

The memory layout of a process is divided into multiple sections.

- Text section— the executable code
- Data section—global variables
- Heap section—memory that is dynamically allocated during program run time
- Stack section— temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

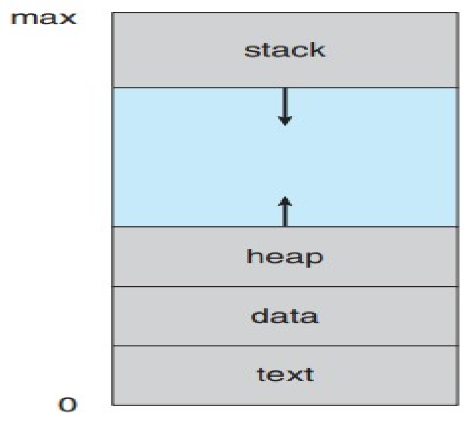


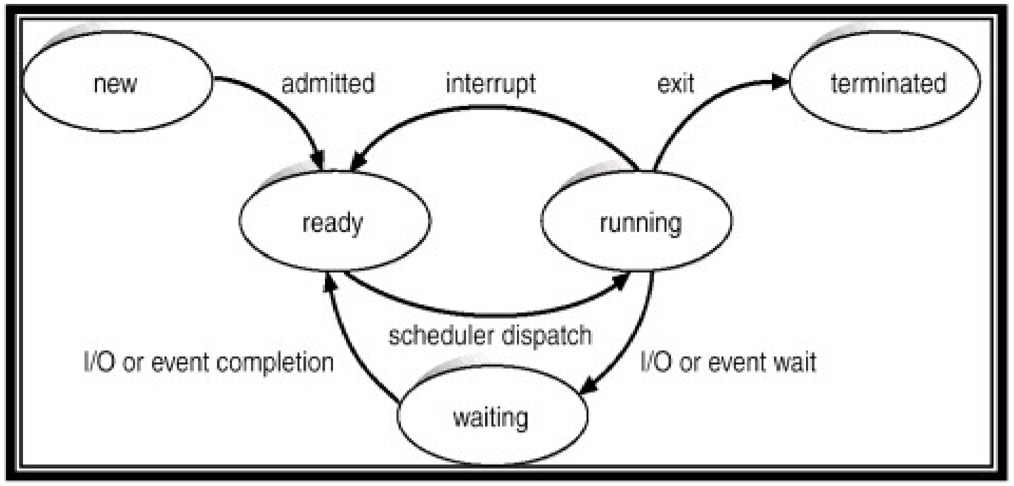
Figure 3.1 Layout of a process in memory.

PROCESS CONCEPT

- The sizes of the text and data sections are fixed, as their sizes do not change during program run time.
- However, the stack and heap sections can shrink and grow dynamically during program execution.
- 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.
- Similarly, the heap will grow as memory is dynamically allocated, and will shrink when memory is returned to the system.
- Although the stack and heap sections grow toward one another, the operating system must ensure they do not overlap one another.

Process States

- As a process executes, it changes state.
- The state of a process is defined by the current activity of that process.
- Each process may be in one of the following states:
  1. New: The process is being created.
  2. Running: Instructions are being executed.
  3. Waiting: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
  4. Ready: The process is waiting to be assigned to a processor.
  5. Terminated: The process has finished execution.



Process State Transition Diagram

Process Control Block

- Each process is represented in the operating system by a process control block (PCB) - also called a task control block
- A PCB defines a process to the operating system.
- It contains the entire information about a process.
- The information a PCB contains are:

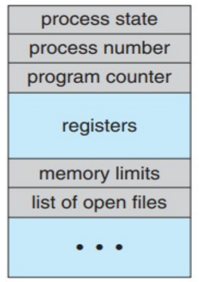


Figure 3.3 Process control block (PCB).

1. Process state: The state may be new, ready, running, waiting or terminated.
2. Program counter: The counter indicates the address of the next instruction to be executed for this process.
3. CPU registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.

Process Control Block

4. CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
5. Memory-management information: This information may include the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
6. Accounting information: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
7. Status information: The information includes the list of I/O devices allocated to this process, a list of open files, and so on.

PROCESS SCHEDULING

- The objective of multiprogramming is to have some process running at all times, so as to maximize CPU utilization.
- The objective of time sharing is to switch a CPU core among processes so frequently that users can interact with each program while it is running.
- Each CPU core can run one process at a time.
- For a system with a single CPU core, there will never be more than one process running at a time, whereas a multicore system can run multiple processes at one time.
- If there are more processes than cores, excess processes will have to wait until a core is free and can be rescheduled.
- The number of processes currently in memory is known as the degree of multiprogramming.

Schedulers

Processes can be described as either I/O bound or CPU bound.

- An I/O-bound process spends more of its time doing I/O than it spends doing computations.
- A CPU-bound process, generates I/O requests infrequently, using more of its time doing computation than an I/O-bound process uses.
- The system with the best performance will have a combination of CPU-bound and I/O bound processes.

PROCESS SCHEDULING

Scheduling Queues

There are 3 types of scheduling queues. They are:

1. Job Queue
2. Ready Queue
3. Device Queue

As processes enter the system, they are put into a job queue.

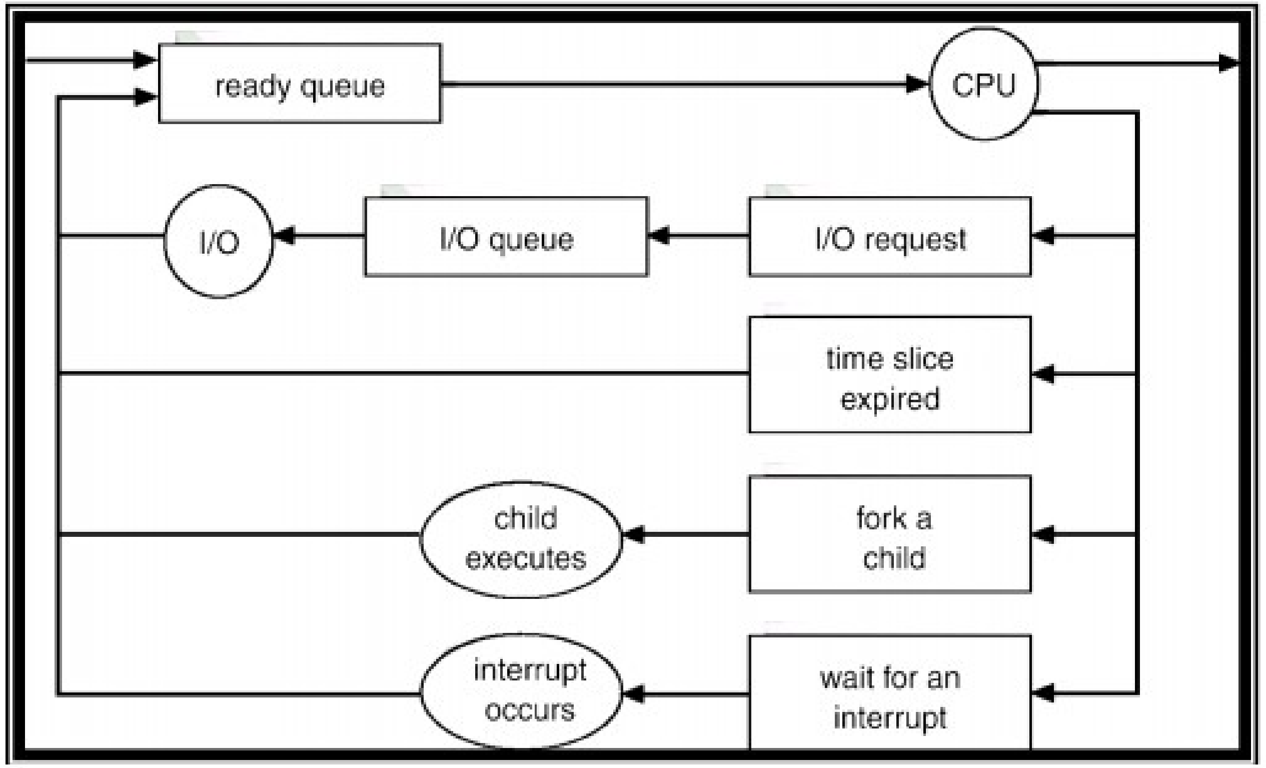
The processes that are residing in main memory and are ready and waiting to execute on a CPU's core are kept on a list called the ready queue.

The list of processes waiting for an I/O device is kept in a device queue for that particular device.

- A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution (or dispatched).
- Once the process is assigned 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 subprocess and wait for its termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready Queue.

PROCESS SCHEDULING

A common representation of process scheduling is a queuing diagram.



Queuing Diagram Representation of Process Scheduling

CONTEXT SWITCHING

- Interrupts cause the operating system to change a CPU core from its current task and to run a kernel routine.
- 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 is represented in the PCB of the process.
- It includes the value of the CPU registers, the process state and memory-management information.
- Perform 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.
- 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.

CONTEXT SWITCHING

- 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 a sing s).

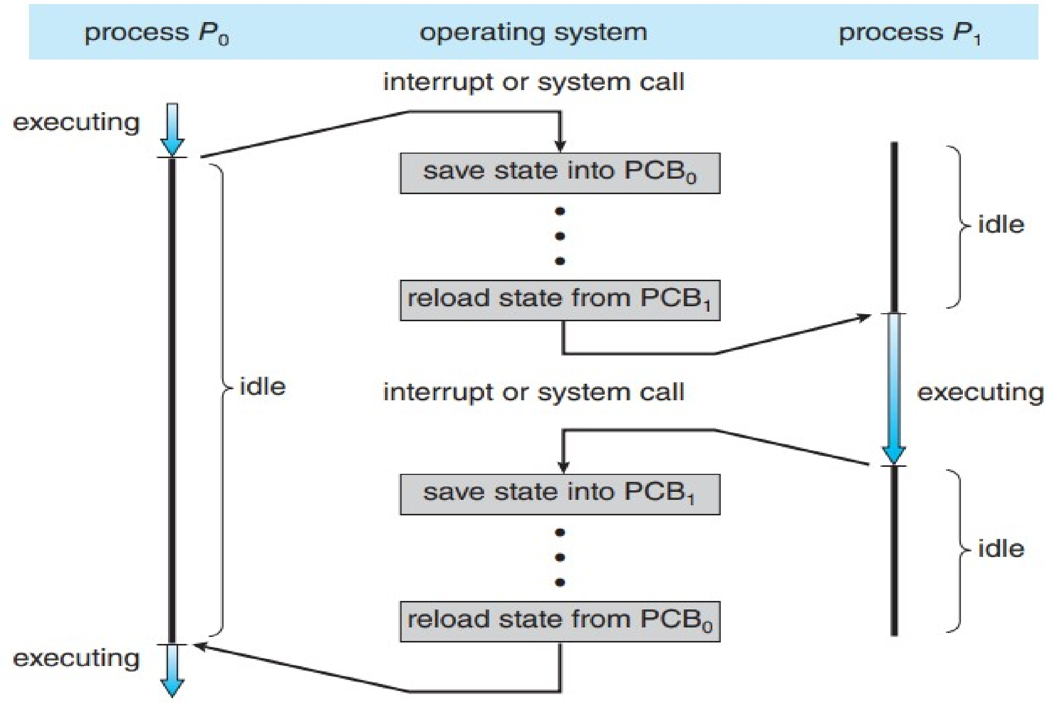


Figure 3.6 Diagram showing context switch from process to process.

Schedulers

- A process migrates between the various scheduling queues throughout its lifetime.
- The operating system must select, for scheduling purposes, processes from these queues in some fashion.
- The selection process is carried out by the appropriate scheduler.
- There are three different types of schedulers. They are:

1. Long-term Scheduler or Job Scheduler
2. Short-term Scheduler or CPU Scheduler
3. Medium term Scheduler

- The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution. It is invoked very infrequently. It controls the degree of multiprogramming.
- The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute, and allocates the CPU to one of them. It is invoked very frequently.
- Medium term Scheduler
  - Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling.
  - The medium-term scheduler, removes processes from memory and thus reduces the degree of multiprogramming.
  - At some later time, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is called **swapping**.

OPERATIONS ON PROCESSES

1. Process Creation

- A process may create several new processes, during the course of execution.
- The creating process is called a parent process, whereas the new processes are called the children of that process.
- Each of these new processes may in turn create other processes, forming a tree of processes.
- In UNIX, each process is identified by its process identifier(pid), which is a unique integer.
- The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.
- When a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.

OPERATIONS ON PROCESSES

Figure 3.7 illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid.

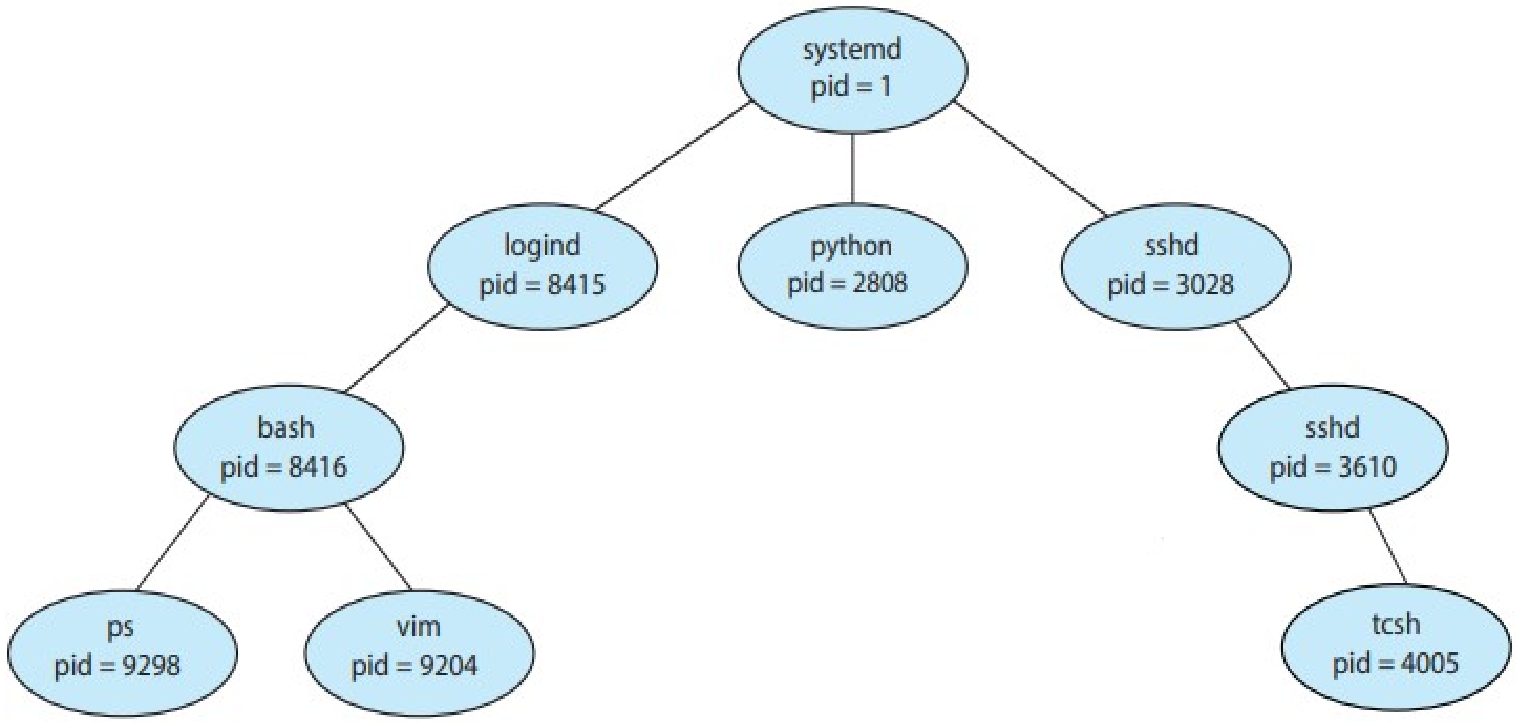


Figure 3.7 A tree of processes on a typical Linux system.

OPERATIONS ON PROCESSES

1. Process Creation

- When a process creates a new(child) process, two possibilities exist in terms of execution:
  1. The parent continues to execute concurrently with its children.
  2. The parent waits until some or all of its children have terminated.
- There are also two possibilities in terms of the address space of the new process:
  1. The child process is a duplicate of the parent process.
  2. The child process has a program loaded into it.
- A new process is created by the fork() system call.
- The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.
- Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

OPERATIONS ON PROCESSES

Process Creation

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <stdlib.h>
 void childprocess();
 void parentprocess();
 int main(void)
{
       int pid;
       pid=fork();
      if(pid <0)
           printf("\n Error in creating process ");
      else if(pid==0)
         childprocess();
      else
        parentprocess();
}
void childprocess()
{
      printf("\nExecuting in child process, pid=%d ",getpid());
      exit(0);
}
void parentprocess()
{
    int status;
    wait(&status);                                                              // waits until child completes execution and returns child's
status
    printf("\n Executing in parent process, pid=%d \n",getpid());
    printf("Child Returned:%d\n", status);    }
```

OPERATIONS ON PROCESSES

2. Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.
- At that point, the process may return a status value (output) to its waiting parent process (via the wait system call).
- All the resources of the process including physical and virtual memory, open files, and I/O buffers are deallocated and reclaimed by the operating system.
- A process can cause the termination of another process via an appropriate system call. (TerminateProcess() in Windows)
- Such a system call can be invoked only by the parent of the process that is to be terminated.
- Otherwise, a user or a misbehaving application could arbitrarily kill another user’s processes.

OPERATIONS ON PROCESSES

2. Process Termination

- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
    1. The child has exceeded its usage of some of the resources that it has been allocated.
    2. The task assigned to the child is no longer required.
    3. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system.
- A parent process may wait for the termination of a child process by using the wait() system call.
- The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child.
- This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated

OPERATIONS ON PROCESSES

3. Zombie Process & Orphan Process

- When a process terminates, its resources are deallocated by the operating system.
- However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status.
- A process that has terminated, but whose parent has not yet called wait(), is known as a zombie process.
- Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.
- If a parent did not invoke wait() and instead terminated, thereby leaving its child processes as orphans.
- The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Cooperating Processes

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is independent if it does not share data with any other processes executing in the system.
- A process is cooperating if it can affect or be affected by the other processes executing in the system.

Benefits of Cooperating Processes

1. Information sharing - Allow concurrent access to information
2. Computation speedup - If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing cores.
3. Modularity - Construct the system in a modular fashion, dividing the system functions into separate processes or threads
4. Convenience

Example

Producer – Consumer Problem

Cooperating Processes

- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data, (ie) send data to and receive data from each other.
- There are two fundamental models of interprocess communication: shared memory and message passing.
- In the shared-memory model, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

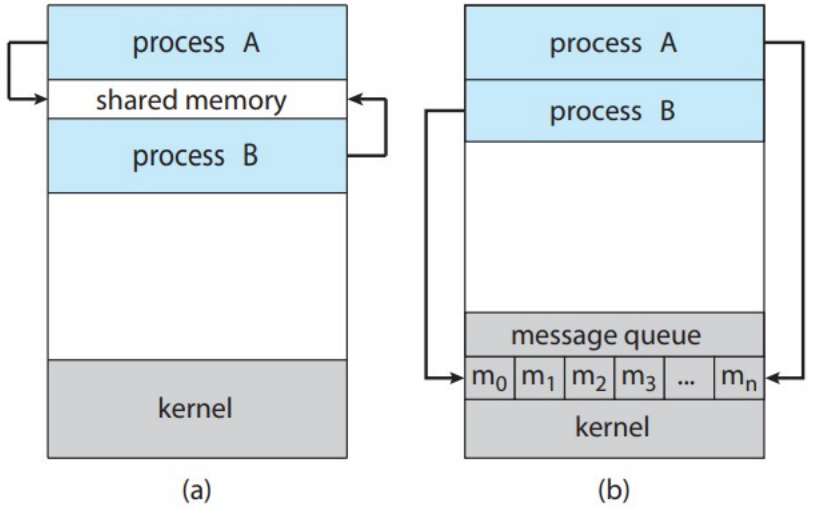


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

IPC in Shared Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- Normally, the operating system 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.
- To illustrate the concept of cooperating processes, consider the producer–consumer problem, which is a common paradigm for cooperating processes

Producer – Consumer Problem

- A producer process produces information that is consumed by a consumer process.
- For example, a web server produces (that is, provides) web content such as HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.
- 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.

The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.

The bounded buffer assumes 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.

Shared data

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
 int out = 0;
```

The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in == out ; the buffer is full when ((in + 1) % BUFFERSIZE) == out.

Producer – Consumer Problem

Producer Process

```
item nextproduced;
while (true)          /* produce an item in next produced */
  {
     while (((in + 1) % BUFFER_SIZE) == out);
  /* do nothing */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
```

Consumer process

```
item nextconsumed;
  while (true)
  {
     while (in == out);
  /* do nothing */
      nextconsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
  }
```

IPC in Message Passing systems

- Another way to achieve IPC is to provide the means for cooperating processes to communicate with each other via a message-passing facility.
- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.
- A message-passing facility provides at least two operations:

❖ send(message) and

❖ receive(message

INTER-PROCESS COMMUNICATION

- Messages sent by a process can be either fixed or variable in size.

Basic Structure:

- If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them.
- This link can be implemented in a variety of ways.
- Physical implementation of the link is done through a hardware bus , network etc,
- There are several methods for logically implementing a link and the operations:
  1. Direct or indirect communication
  2. Symmetric or asymmetric communication
  3. Automatic or explicit buffering

Direct Communication

Naming: Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

1. Direct Communication

Each process that wants to communicate must explicitly name the recipient or sender of the communication.

A communication link in this scheme has the following properties:

i. A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

ii. A link is associated with exactly two processes.

iii. Exactly one link exists between each pair of processes.

There are two ways of addressing namely

Symmetry in addressing - Both the sender process and the receiver process must name the other to communicate.

Asymmetry in addressing - Only the sender names the recipient; the recipient is not required to name the sender.

In symmetry in addressing, the send and receive primitives are defined as:

send(P, message) $\longrightarrow$ Send a message to process P

receive(Q, message) $\longrightarrow$ Receive a message from Q

In asymmetry in addressing , the send & receive primitives are defined as:

send (P, message) $\longrightarrow$ Send a message to process P

receive(id, message) $\longrightarrow$ receive message from any process, id is set to the name of the process with which communication has taken place

Indirect Communication

- With indirect communication, the messages are sent to and received from mailboxes, or ports.
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox.
- The send and receive primitives are defined as follows:

  send (A, message) $\rightarrow$ Send a message to mailbox A.

  receive (A, message) Receive a message from mailbox A.

  A communication link has the following properties:

i. A link is established between a pair of processes only if both members of the pair have a shared mailbox.

ii. A link may be associated with more than two processes.

iii. A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox .

Synchronization

- Communication between processes takes place through calls to send() and receive() primitives.
- Message passing may be either blocking or nonblocking also known as synchronous and asynchronous.
- Message passing may be either blocking or non-blocking.

1. Blocking Send - The sending process is blocked until the message is received by the receiving process or by the mailbox.
2. Non-blocking Send - The sending process sends the message and resumes operation.
3. Blocking Receive - The receiver blocks until a message is available.
4. Non-blocking Receive – The receiver retrieves either a valid message or a null.

Buffering

- A link has some capacity that determines the number of message that can reside in it temporarily. This property can be viewed as a queue of messages attached to the link.
- There are three ways that such a queue can be implemented.
- **Zero capacity:** The queue has a maximum length of zero; the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message (message system with no buffering).
- **Bounded capacity:** The queue has finite length n. Thus at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity:** The queue has potentially infinite length. Thus any number of messages can wait in it. The sender is never delayed.

Threads

A thread is a basic unit of CPU utilization;

It comprises of a thread ID, a program counter (PC), a register set, and a stack.

It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

A traditional process has a single thread of control.

If a process has multiple threads of control, it can perform more than one task at a time.

Figure 4.1 illustrates the difference between a traditional single-threaded process and a multithreaded process.

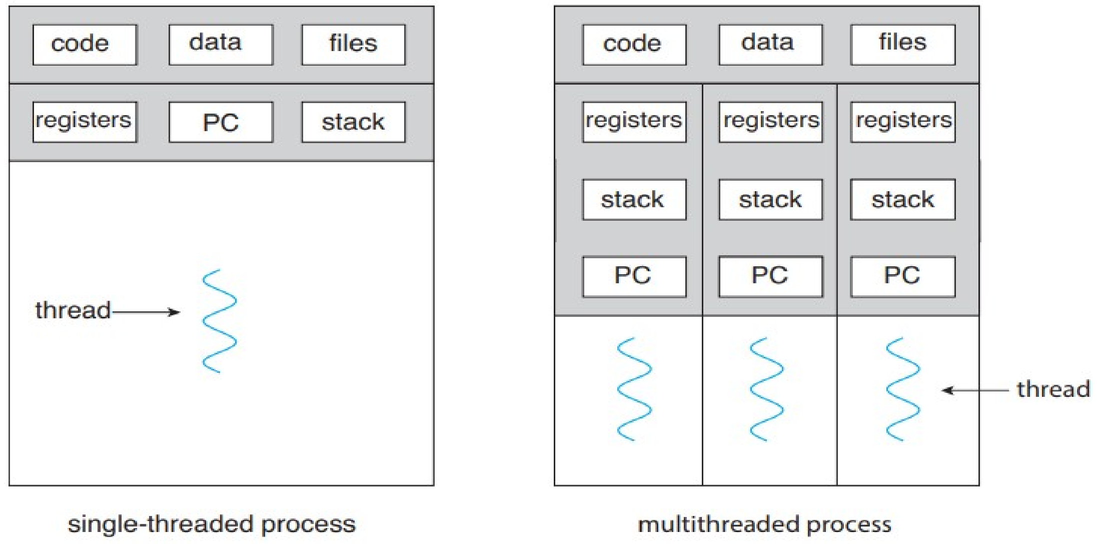


Figure 4.1 Single-threaded and multithreaded processes.

Threads

Most software applications that run on modern computers and mobile devices are multithreaded.

An application is implemented as a separate process with several threads of control.

Few examples of multithreaded applications:

- An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image.
- A web browser might have one thread display images or text while another thread retrieves data from the network.
- A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Benefits of Multithreading

1. Responsiveness:

- Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation.
- A single-threaded application would be unresponsive to the user until the operation had been completed. In contrast, if the time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user.

2. Resource sharing:

- Processes can share resources through techniques such as shared memory and message passing.
- Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default.
- The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

Benefits of Multithreading

3. Economy:

- Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
- In general thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.

4. Scalability (Utilization of Multiprocessor Architecture):

- The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.
- A single-threaded process can run on only one processor, regardless how many are available.

Multicore Programming

- Multiple computing cores can be placed on a single processing chip where each core appears as a separate CPU to the operating system. Such systems are referred as multicore programming.
- Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.
- Consider an application with four threads. On a system with a single computing core, concurrency means that the execution of the threads will be interleaved over time (Figure 4.3), because the processing core is capable of executing only one thread at a time.
- On a system with multiple cores, concurrency means that some threads can run in parallel, because the system can assign a separate thread to each core (Figure 4.4).

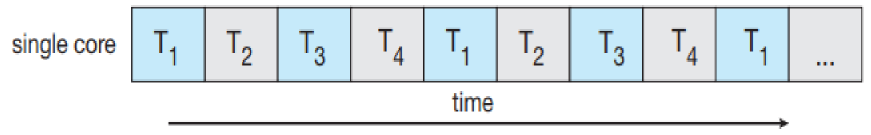


Figure 4.3 Concurrent execution on a single-core system.

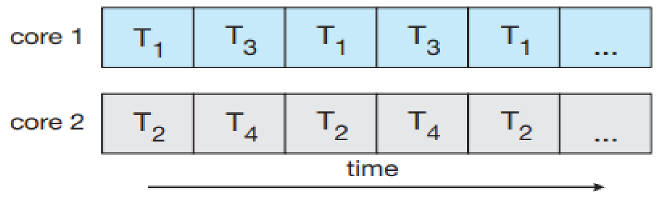

core 1
T_1 T_3 T_1 T_3 T_1 ...
core 2
T_2 T_4 T_2 T_4 T_2 ...
time

Figure 4.4 Parallel execution on a multicore system.

Multicore Programming

A concurrent system supports more than one task by allowing all the tasks to make progress.

In contrast, a parallel system can perform more than one task simultaneously.

Thus, it is possible to have concurrency without parallelism.

Before multiprocessor and multicore architectures, most computer systems had only a single processor, and CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes, thereby allowing each process to make progress.

Such processes were running concurrently, but not in parallel.

Multithreading models

User threads

- User threads are supported above the kernel and are managed without kernel support.
- Thread creation, management and scheduling are done in user space.
- Fast to create and manage.
- When a user thread performs a blocking system call, it will cause the entire process to block even if other threads are available to run within the application.
- Example: POSIX Pthreads, Mach C-threads and Solaris 2 UI-threads.

Kernel threads

- kernel threads are supported and managed directly by the operating system.
- Thread creation, management and scheduling are done in kernel space.
- Slow to create and manage
- When a kernel thread performs a blocking system call, the kernel schedules another thread in the application for execution.
- Example: Windows NT, Windows 2000 , Linux, Mac OS, Solaris 2,BeOS and Tru64 UNIX support kernel threads.

MULTITHREADING MODELS

1. Many-to-One
2. One-to-One
3. Many-to-Many

MULTITHREADING MODELS

Many-to-One:

- Maps many user-level threads to single kernel thread.
- Used on systems that does not support kernel threads.

Advantage: Thread management is done by the thread library in user space, so it is efficient.

Disadvantage:

1. The entire process will block if a thread makes a blocking system call.
2. Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessor systems.

- Green threads—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to one model.

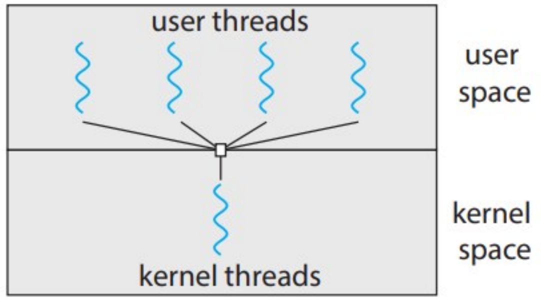


Figure 4.7 Many-to-one model.

MULTITHREADING MODELS

One-to-One:

- The one-to-one model maps each user thread to a kernel thread.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may burden the performance of a system.
- Linux, along with the family of Windows operating systems, implement the one-to-one model.

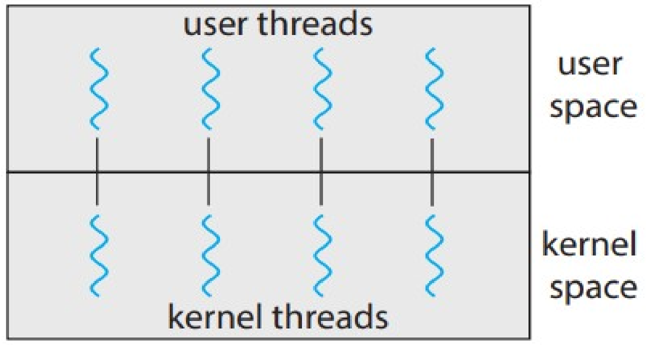


Figure 4.8 One-to-one model.

MULTITHREADING MODELS

Many-Many:

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a system with eight processing cores than a system with four cores).

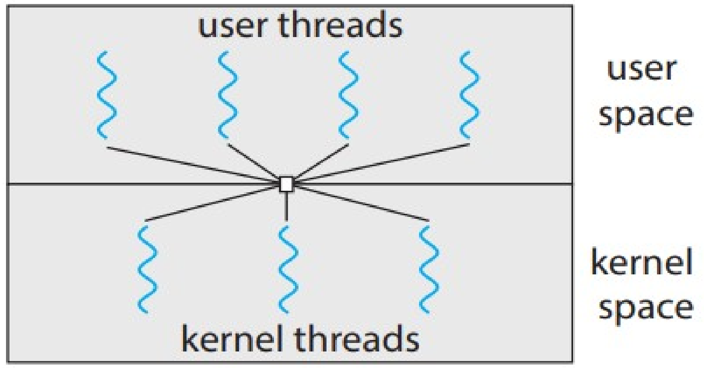


Figure 4.9 Many-to-many model.

MULTITHREADING MODELS

Two Level Model:

- One variation on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread.
- This variation is sometimes referred to as the two-level model.

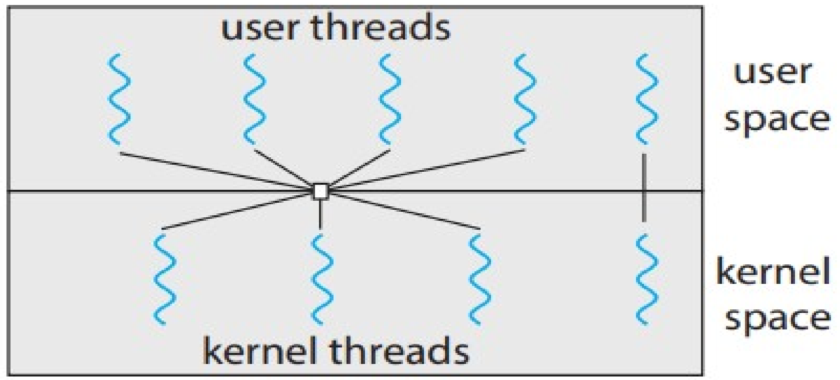


Figure 4.10 Two-level model.

Thread Libraries

A thread library provides the programmer with an API for creating and managing threads.

There are two primary ways of implementing a thread library.

The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java.

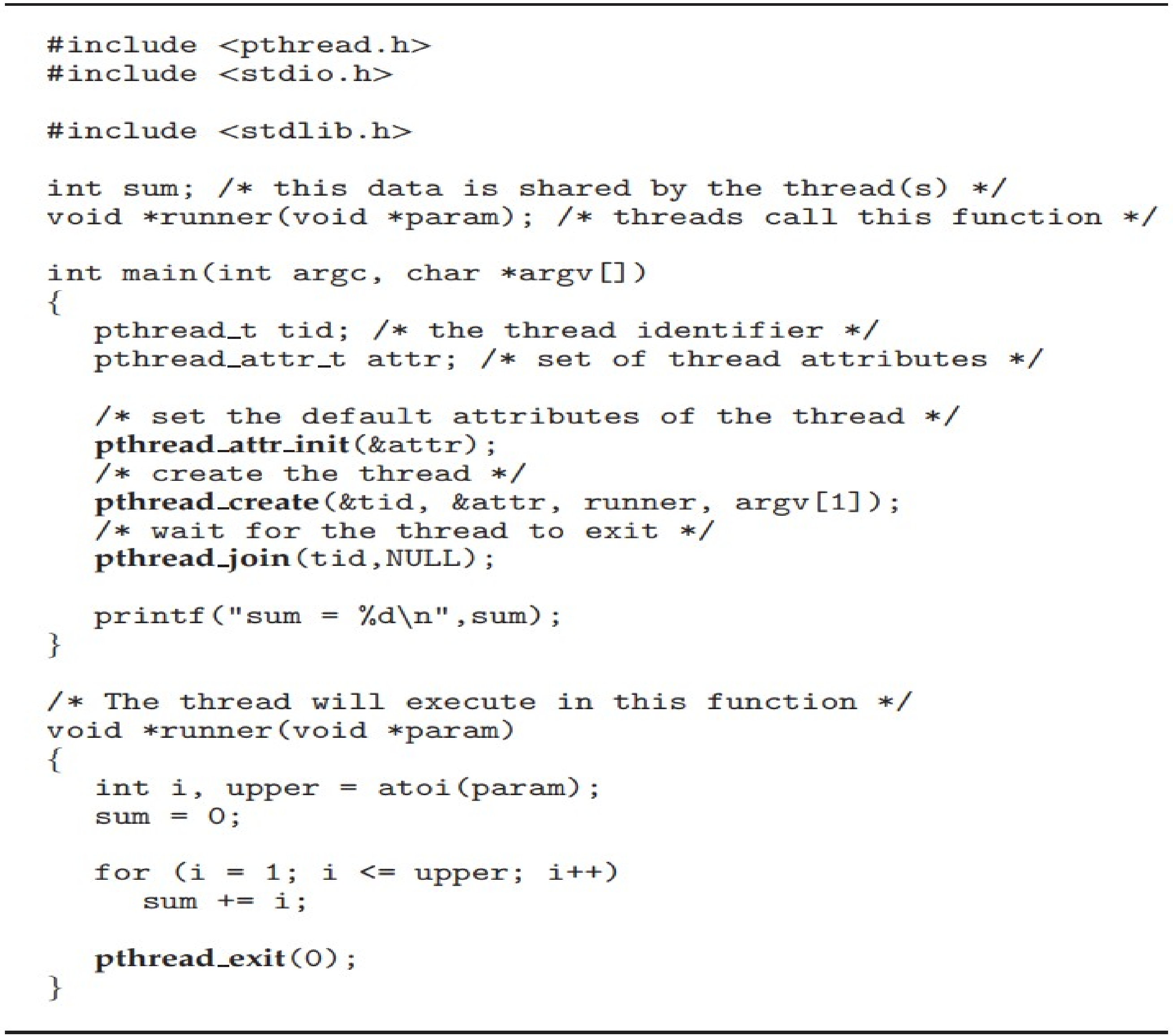
Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library.

The Windows thread library is a kernel-level library available on Windows systems.

The Java thread API allows threads to be created and managed directly in Java programs.

Pthreads

Pthreads refers to the POSIX standard defining an API for thread creation and synchronization.



```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Threading Issues

1. fork() and exec() system calls

- A fork() system call may duplicate all threads or duplicate only the thread that invoked fork().
- If a thread invoke exec() system call, the program specified in the parameter to exec will replace the entire process.

2. Thread cancellation

- It is the task of terminating a thread before it has completed.
- A thread that is to be cancelled is called a target thread. There are two types of cancellation namely

  1.Asynchronous Cancellation – One thread immediately terminates the target thread.

  2.Deferred Cancellation – The target thread can periodically check if it should terminate, and does so in an orderly fashion.

Threading Issues

3. Signal handling

1. A signal is used to notify a process that a particular event has occurred.
2. A generated signal is delivered to the process.
   a. Deliver the signal to the thread to which the signal applies.
   b. Deliver the signal to every thread in the process.
   c. Deliver the signal to certain threads in the process.
   d. Assign a specific thread to receive all signals for the process.
3. Once delivered the signal must be handled.
   a. Signal is handled by
      i. A default signal handler
      ii. A user defined signal handler

4. Thread pools

- Creation of unlimited threads exhausts system resources such as CPU time or memory. Hence we use a thread pool.
- In a thread pool, a number of threads are created at process startup and placed in the pool.
- When there is a need for a thread the process will pick a thread from the pool and assign it a task.
- After completion of the task, the thread is returned to the pool.

5. Thread specific data

Threads belonging to a process share the data of the process. However each thread might need its own copy of certain data known as thread-specific data.

Interrupts

Consider a typical computer operation: a program performing I/O. To start an I/O operation, the device driver loads the appropriate registers in the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as “read a character from the keyboard”). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver that it has finished its operation. The device driver then gives control to other parts of the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information such as “write completed successfully” or “device busy”. But how does the controller inform the device driver that it has finished its operation? This is accomplished via an interrupt.

Interrupts

Hardware may trigger an interrupt at any time by sending a signal to the CPU, by means of system bus.

Interrupts are a key part of how operating systems and hardware interact.

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location.

The fixed location usually contains the starting address where the service routine for the interrupt is located.

The interrupt service routine executes; on completion, the CPU resumes the interrupted computation. A timeline of this operation is shown in Figure 1.3.

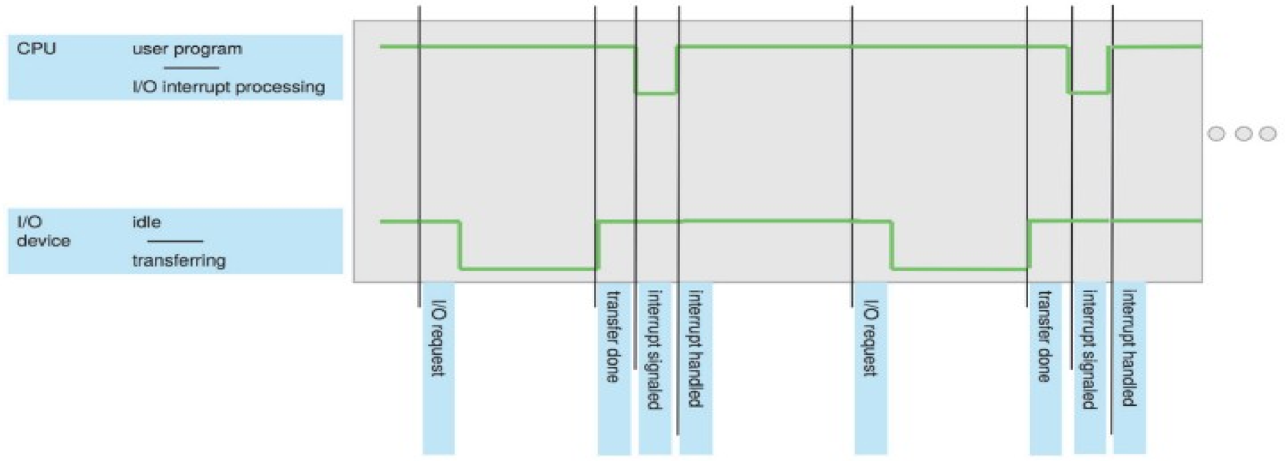


Figure 1.3 Interrupt timeline for a single program doing output.

Interrupt Processing

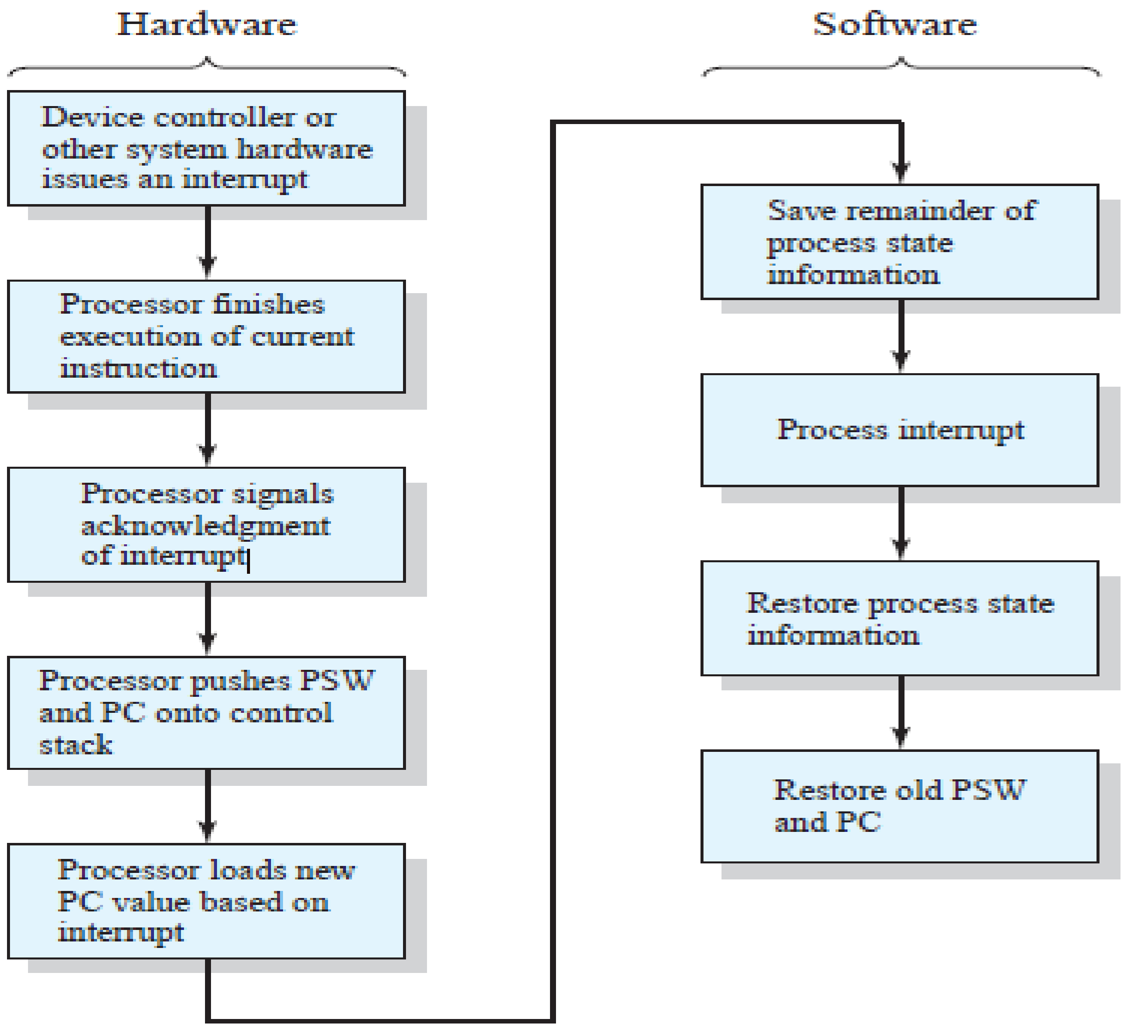

Hardware
Software
Device controller or other system hardware issues an interrupt
Processor finishes execution of current instruction
Processor signals acknowledgment of interrupt
Processor pushes PSW and PC onto control stack
Processor loads new PC value based on interrupt
Save remainder of process state information
Process interrupt
Restore process state information
Restore old PSW and PC

Figure 1.10 Simple Interrupt Processing

Interrupts - Implementation

Figure 1.4 summarizes the interrupt-driven I/O cycle.

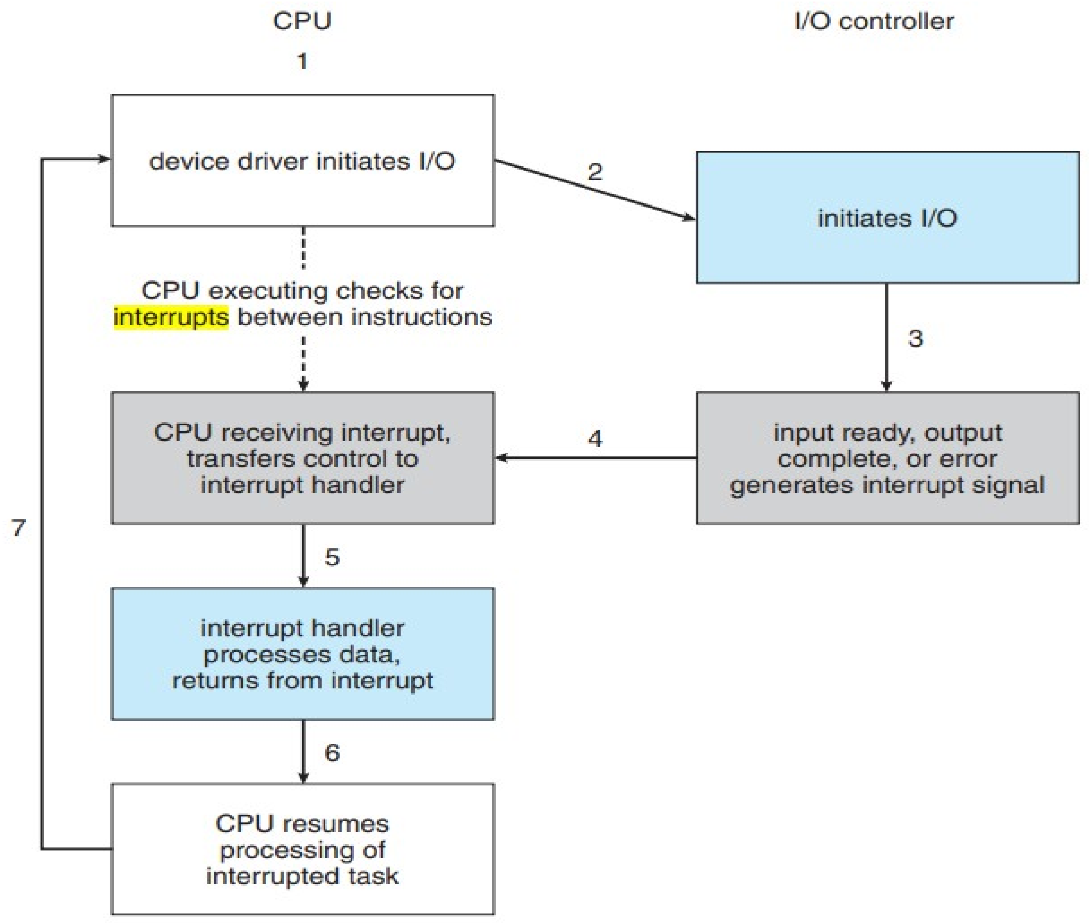

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Figure 1.4 Interrupt-driven I/O cycle.

Interrupts

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common.

The interrupt must transfer control to the appropriate interrupt service routine.

The method for managing this transfer would be to invoke a generic routine to examine the interrupt information.

The routine, in turn would call the interrupt-specific handler. However, interrupts must be handled quickly, as they occur very frequently.

A table of pointers to interrupt routines can be used to provide the necessary speed.

The interrupt routine is called indirectly through the table, with no intermediate routine needed.

Generally, the table of pointers is stored in low memory (the first hundred or so locations).

These locations hold the addresses of the interrupt service routines for the various devices.

This array, or interrupt vector, of addresses is then indexed by a unique number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device.

The interrupt architecture must also save the state information of whatever was interrupted, so that it can restore this information after servicing the interrupt.

If the interrupt routine needs to modify the processor state— for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning.

After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

Interrupts - Implementation

The basic interrupt mechanism works as follows.

The CPU hardware has a wire called the interrupt-request line that the CPU senses after executing every instruction.

When the CPU detects that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and jumps to the interrupt-handler routine by using that interrupt number as an index into the interrupt vector.

It then starts execution at the address associated with that index.

The interrupt handler saves any state it will be changing during its operation, determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt.

We say that the device controller raises an interrupt by asserting a signal on the interrupt request line, the CPU catches the interrupt and dispatches it to the interrupt handler, and the handler clears the interrupt by servicing the device.

Interrupts - Implementation

The basic interrupt mechanism enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service.

In a modern operating system, however, we need more sophisticated interrupt handling features.

1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device.
3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

In modern computer hardware, these three features are provided by the CPU and the interrupt-controller hardware.

Interrupts - Implementation

Most CPUs have two interrupt request lines.

One is the non-maskable interrupt, which is reserved for events such as unrecoverable memory errors.

The second interrupt line is maskable: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.

The maskable interrupt is used by device controllers to request service.

The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service.

In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector.

A common way to solve this problem is to use interrupt chaining, in which each element in the interrupt vector points to the head of a list of interrupt handlers.

Interrupts - Implementation

When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request.

This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

Figure 1.5 illustrates the design of the interrupt vector for Intel processors.

The events from 0 to 31, which are nonmaskable, are used to signal various error conditions.

The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

The interrupt mechanism also implements a system of interrupt priority levels.

These levels enable the CPU to defer the handling of low-priority interrupts without masking all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.