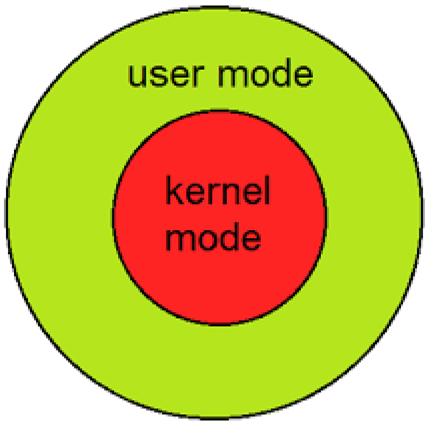
MODULE-2: OPERATING SYSTEMS PRINCIPLES

2.1 System Calls

- System call provides an interface to the services made available by an OS.

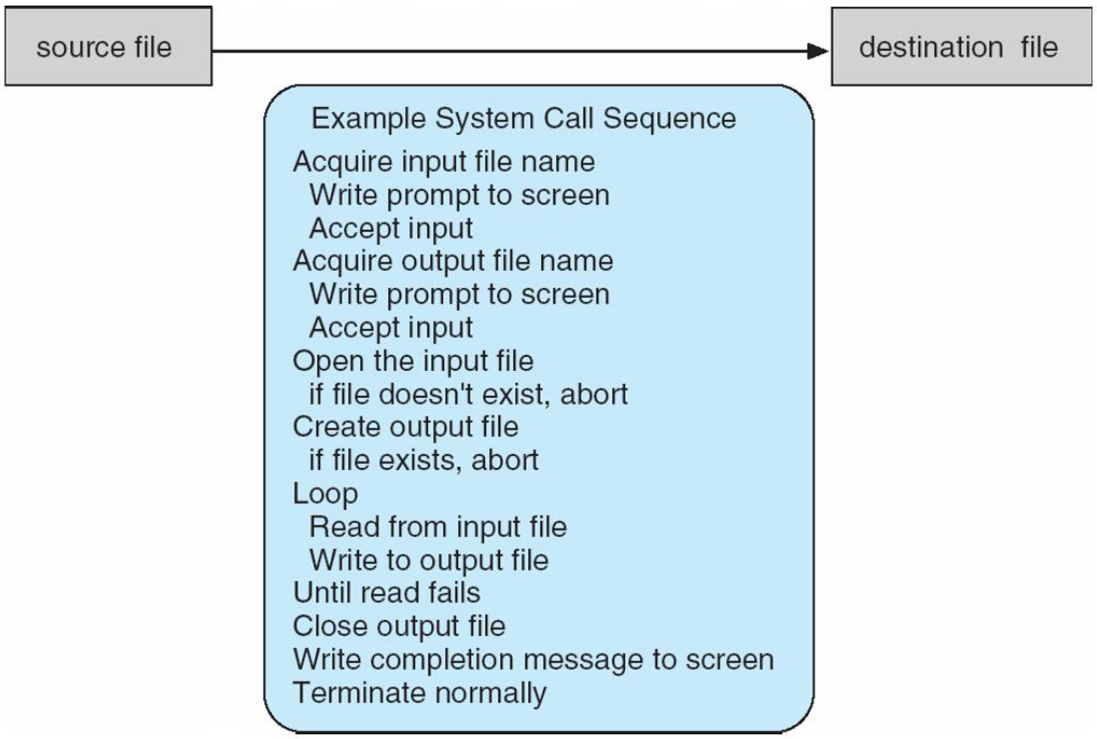


- System call is the programmatic way of in which a computer program requests a service from the kernel of the OS.
- These calls are generally available as routines written in C and C++.
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call.
- Three most common APIs are:
    1. Win32 API for Windows,
    2. POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X),
    3. Java API for the Java virtual machine (JVM)

Example of System Calls

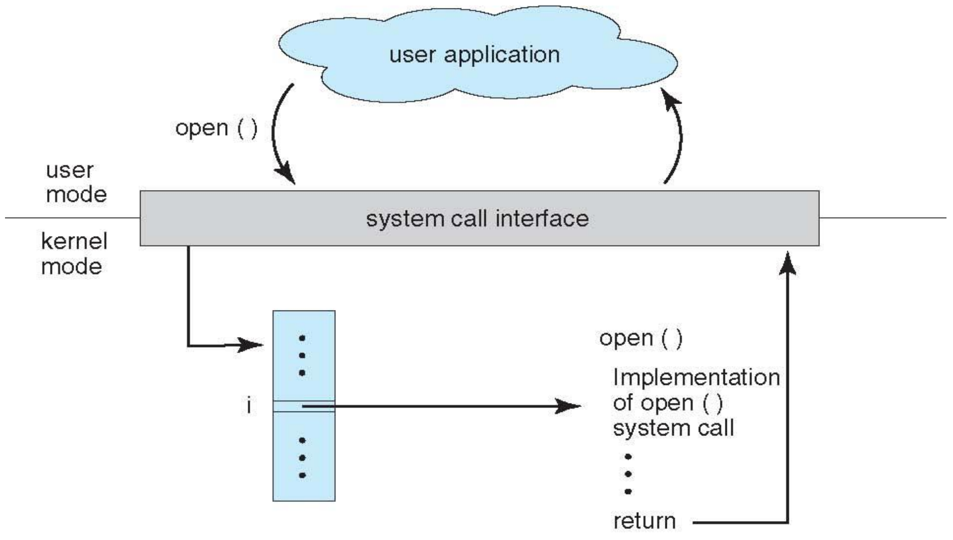
System calls sequence to copy the contents of one file to another file.

NOTE:



2.2 System Call Implementation

- Typically, a **number** is associated with **each system call.**
- **System-call interface** maintains a table indexed according to these numbers.
- The system call interface **invokes the intended system call in OS kernel** and **returns status of the system call and any return values**.



NOTE:

The caller need not know anything about how the system call is implemented. Just needs to obey the API and understand what the OS will do as a result call.

2.3 Types of System Call

- System calls can be grouped roughly into five major categories:
    1. **Process Control**
    2. **File Manipulation**
    3. **I/O Device Management**
    4. **Information Maintenance**
    5. **Communications**

1. Process Control

- end, abort
- load, execute
- fork
- create process, terminate process
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory

2. File Manipulation

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

3. I/O Device Management

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

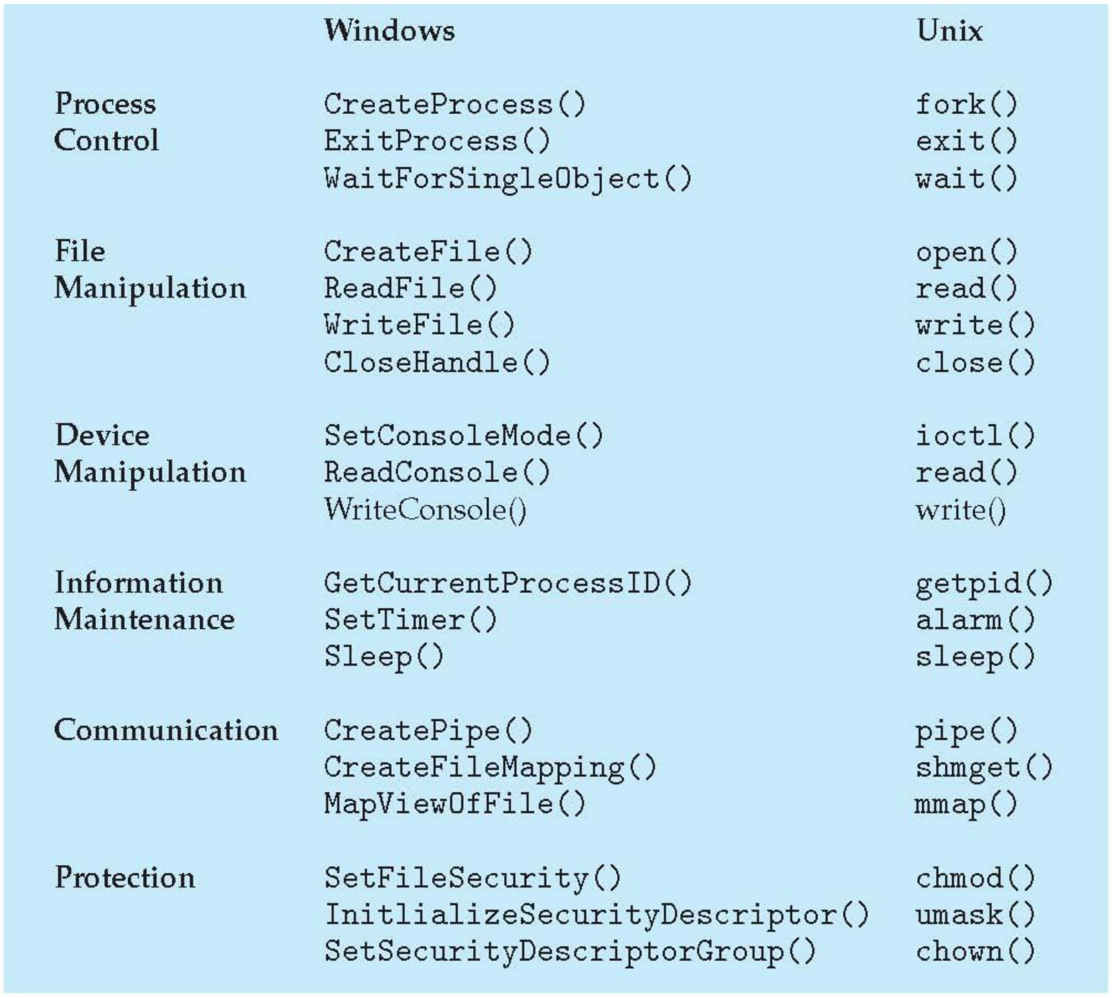
4. Information Maintenance

- 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

5. Communications

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

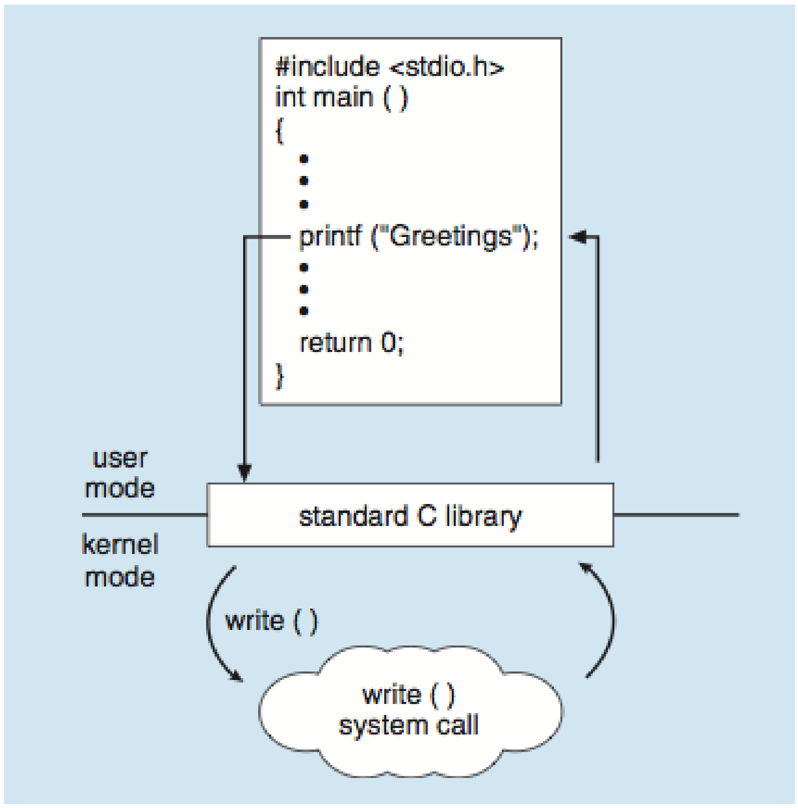
Examples of Windows and Unix System Calls



	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Example -- Standard C Library

C program invoking printf() library call, which calls write() system call.



2.4 Process Management

Process:

- **A** program in execution.

Threads:

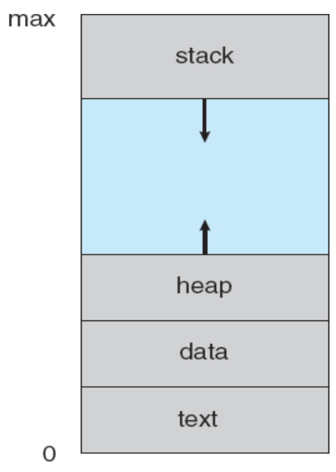
- A thread is unit of execution within a process.
- A process can have one thread to many threads.

The Process

- Process memory is divided into **four sections**.
    - The program code, also called **text section**
    - Current activity including **program counter**, processor registers
    - **Stack** containing temporary data

        - Function parameters, return addresses, local variables
    - **Data section** containing global variables
    - **Heap** containing memory dynamically allocated during run time

A Process in memory



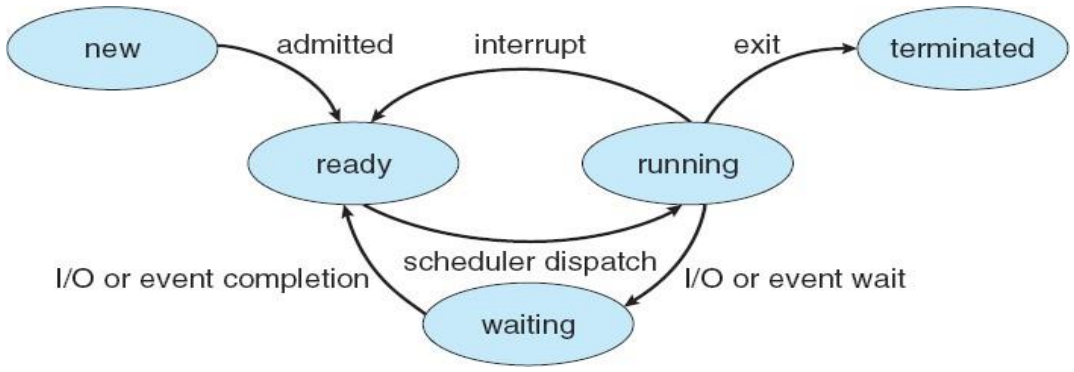
- A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk (often called an **executable file**).
- A process is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory.
- Two common techniques for loading executable files are
    - double-clicking an icon representing the executable file
    - entering the name of the executable file on the command line

Process State:

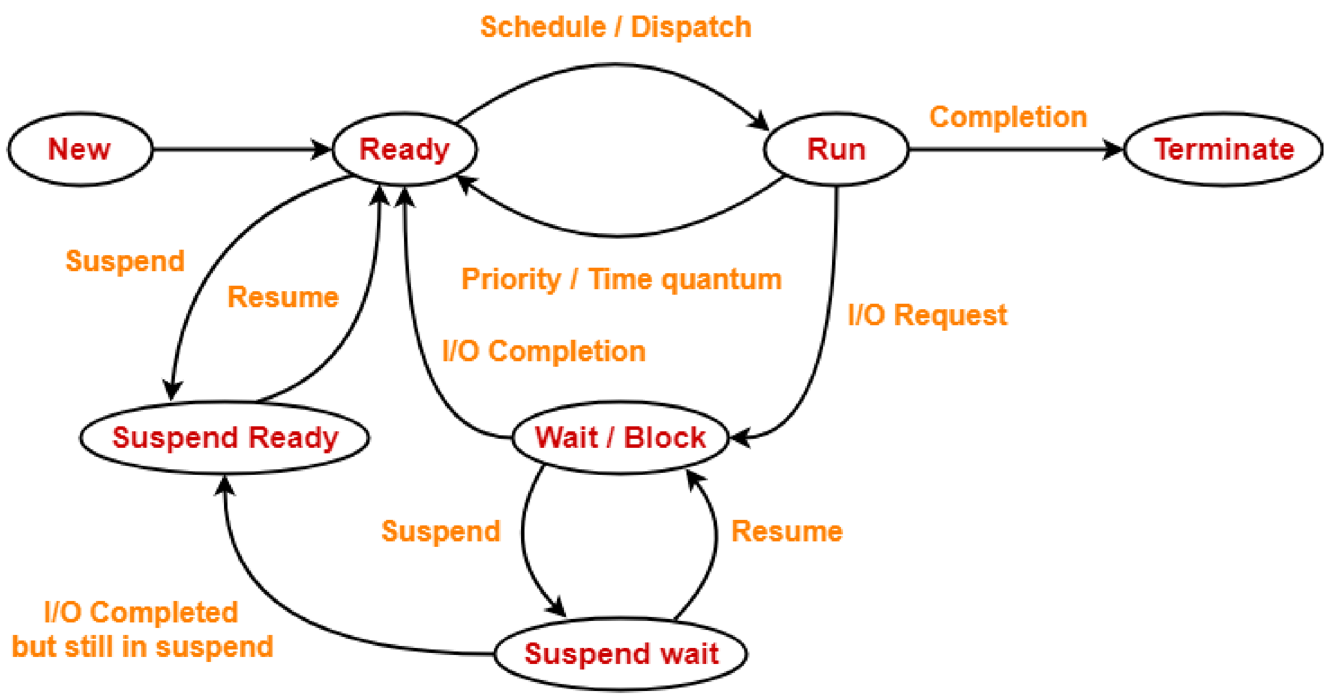
- As process executes, it changes state.
- The state of a process is defined in part by the current activity of that process.
- Each process may be in one of the following states:
    - **NEW** – The process being created.
    - **RUNNING** – Instructions are being executed.

- **WAITING** – The process is waiting for some event to occur. (Such as an I/O completion or reception of signal)
- **READY** – The process is waiting to be assigned to a processor.
- **TERMINATED** – The process has finished execution.

Process State Transition Diagram



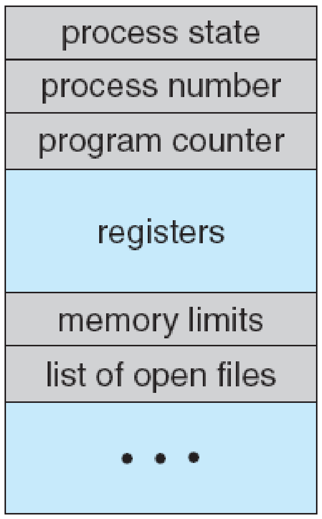
- It is important that only one process can be *running* on any processor at any instant.
- Many processes may be *ready* and *waiting.*
- **Process** go to **suspend state** if they do not satisfy the requirement/ MM is unable to handle more **process** due to storage then some of the **processes** go to **suspend state**(secondary memory) for the finite amount of time.
- When the resource is sufficient/space is available in MM they are resume & come back to MM.



Process State Diagram

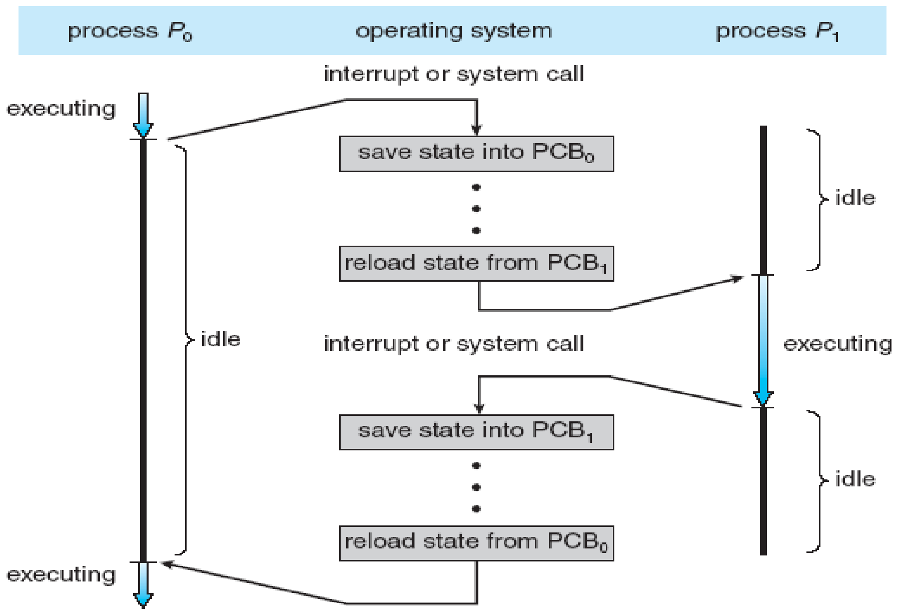
Process Control Block

- Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**.
- The PCB simply serves as the repository for any information that may vary from process to process.
- It 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 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.

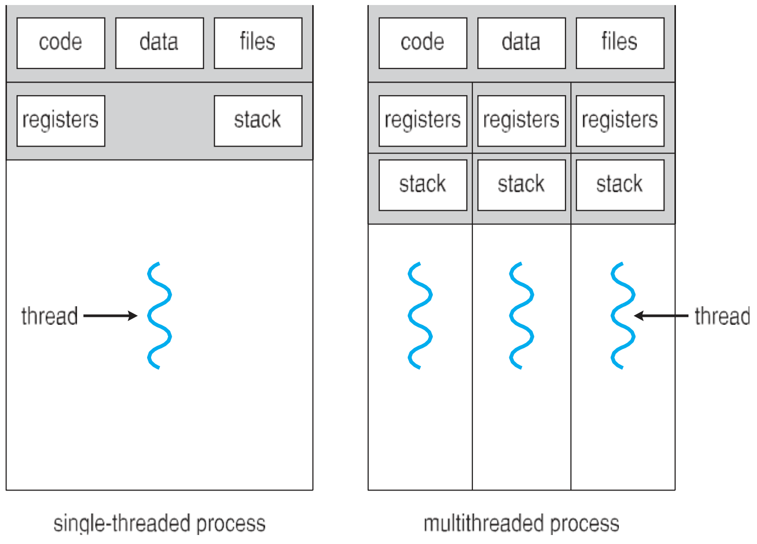


- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
    - **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.

2.5 Threads

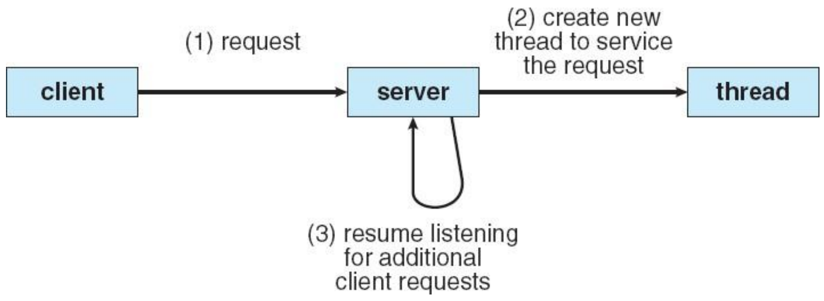
- A thread is a basic **unit of CPU utilization**.
- It comprises a thread ID, a program counter, a register set, and a stack.
- Threads are popular way to **improve performance** through parallelism by reducing the **overhead of process switching**.
- A thread sometimes called a **light weight process.**
- 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 (or ***heavyweight) 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.
- Threads have been successfully used in implementing network servers.

Single and Multithreaded Processes



Motivation

- Most modern applications are multithreaded.
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
    - Update display
    - Fetch data
    - Spell checking
- Answer a network request
- Process creation is **heavy-weight** while thread creation is **light-weight**. When a request is made, rather than creating another process, the server **creates a new thread to service the request** and resume listening for additional requests.


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

- Threads also play a vital role in remote procedure call (RPC) systems.
- Can simplify code, increase efficiency
- Most operating system kernels are generally multithreaded.

Benefits

The benefits of multithreaded programming can be broken down into four major categories:

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.

Resource sharing.

- Processes can only 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.

Economy.

- Allocating memory and resources for process creation is costly.

Scalability.

- The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

User Threads Vs Kernel Threads

- Support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.
- User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed

directly by the operating system.

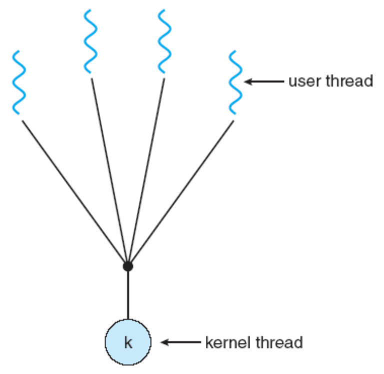
User threads	**Kernel threads**
User threads are supported above the Kernel and are implemented by a thread library at the user level	Kernel threads are supported directly by the operating system
Thread creation & scheduling are done in the user space, without kernel. Therefore they are fast to create and manage.	Thread creation, scheduling and management are done by the operating system.
Blocking system call will cause the entire process to block	If the thread performs a blocking system call, the kernel can schedule another thread in the application for execution

- Virtually all contemporary operating systems—including Windows, Linux, Mac OS X, and Solaris— support kernel threads.

Multithreading Models

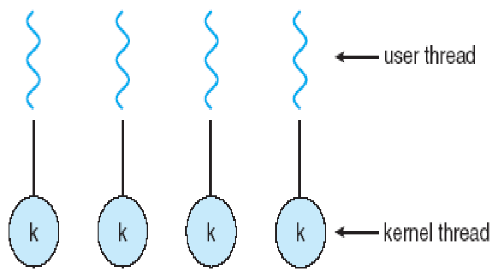
- **Many-to-One**
- **One-to-One**
- **Many-to-Many**

Many-to-One Model



- The many-to-one model maps many user-level threads to one kernel thread.
- Thread management is done by the thread library in user space, so it is efficient.
- The entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.
- **Green threads**—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model.
- However, very few systems continue to use this model because of its inability to take advantage of multiple processing cores.

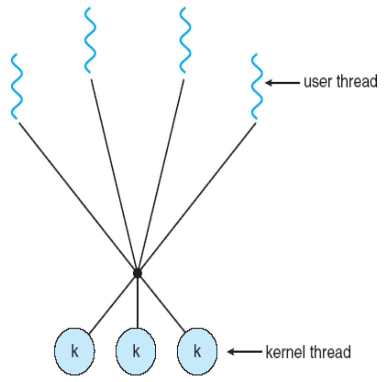
One–to-one model



- 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.
- **Drawback** - creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.
- Linux, along with the family of Windows operating systems, implement the one- to-one model.

Many-to-Many Model



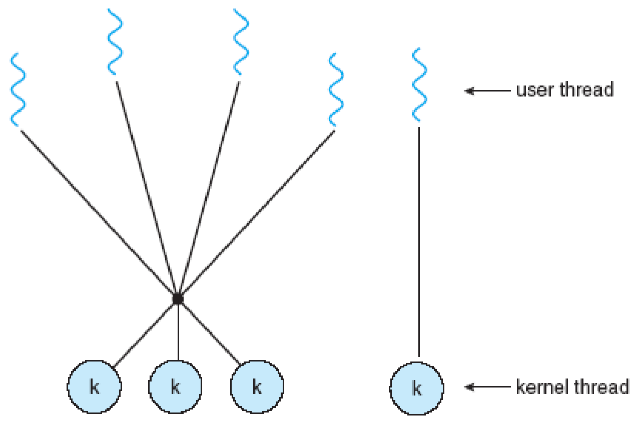
- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine
- Many-to-one model allows the developer to create as many user threads as she wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time.

Drawback

✓ Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.

✓ Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

- ✓ 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.**



2.6 Inter process communication

- Processes executing concurrently in the operating system 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.
- There are several reasons for providing an environment that allows process cooperation:

1. Information sharing. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such 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. Notice that such a speedup can be achieved only if the computer has multiple processing cores.

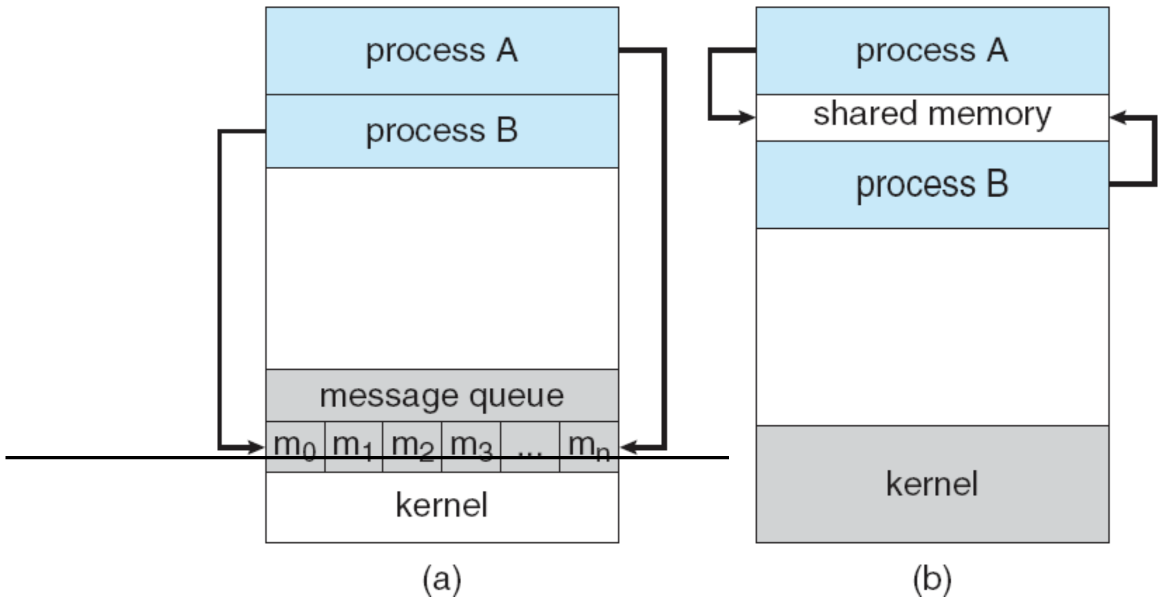
3. **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads,.

4. 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.

- Cooperating processes require an **interprocess communication** **(IPC)** mechanism that will allow them to exchange data and information.
- There are two fundamental models of interprocess communication:
    - **Shared memory**
    - **Message passing**

- 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.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

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



Shared Memory systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- 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.
- **Producer–Consumer problem**
- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
- 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.
- 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

Message Passing System

- 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

Naming

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

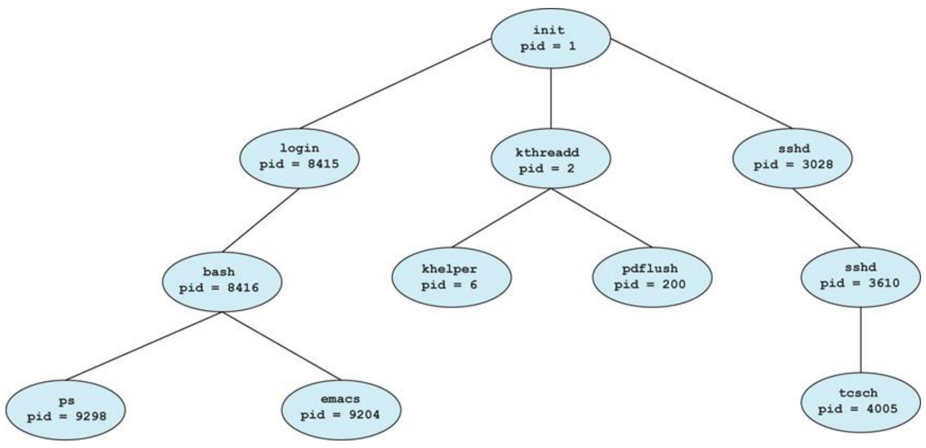
Direct Communication

- Each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:

- send(P, message)—Send a message to process P. receive(Q, message)—Receive a message from process Q.

2.7 Operations on Processes

- A process may create several new processes, via a create-process system call, during the course of execution.
- The **creating process** is called a **parent process**, and 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**.



A tree of processes on a typical Linux system

- Most operating systems (including UNIX, Linux, and Windows) identify processes according to a **unique process identifier** (or pid), which is typically an **integer number**.
- 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.

Resource sharing options

- Parent and children share all resources
- Children share subset of parent’s resources
- Parent and child share no resources

Execution options

- Parent and children execute concurrently
- Parent waits until children terminate

Process Creation

- Running a new program involves 2 steps:

  1. Fork : Clone the current process; Identical copy of parent but it is a different process; PID is different

  2. Exec : Replace current program with a different one; PID does not change

- Fork system call is used to **clone a running process** the **cloned process is a child** of the parent.
- The parent and child are **identical processes**
  - Have exactly the same stack (sequence of function calls)
  - Have same virtual memory
  - Have same set of files
- The cloned child **differs** from the parent in:
  - The process ID
  - The return value from fork in the parent and child is slightly different
  - This is used to tell the difference between parent and child process
- After a new child process created, both processes will execute the next instruction following the fork() system call.

- **NOTE:** It takes no parameters and returns an integer value. Below are different values returned by fork().
- ***Negative Value***: creation of a child process was **unsuccessful.**
  Zero: Newly created child process.
  Positive value: Returned to parent or caller. The value contains process ID of newly created child process.
- After a fork() system call, one of the two processes typically uses the **exec()** system call to replace the process with a new program.
- The **exec()** system call loads a binary file into memory and starts its execution.

- Forking processes has the following **advantages**
  1. **Fault tolerance**: If the child crashes parent process is unaffected. Highly used in web servers and databases to improve resilience
  2. **Security:** Parent and child can have different security settings for better auditing
  3. **Low latency**: Since memory and I/O streams are identical start-up overhead is minimized.
  4. **Concurrency and Performance**: Parent and child can collaborate to perform different operations to improve performance.

Example:

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

#include <stdio.h>

#include <unistd.h>

int main()
```

```
{

pid t pid;

/* fork a child process */

pid = fork();

if (pid < 0) { /* error occurred */

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0) { /* child process */

execlp("/bin/ls","ls",NULL);

}

else { /* parent process */

/* parent will wait for the child to complete */

wait(NULL);

printf("Child Complete");

}

return 0;

}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
    - Returns status data from child to parent (via **wait())**
    - Process' resources are deallocated by operating system – physical and virtual memory, open files, and I/O buffers,etc
- 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 operating systems does not allow a child to continue if its parent terminates

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
- **Cascading termination**: All children, grandchildren, etc. are terminated.
- The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the wait()system call. The call returns status information and the pid of the terminated process.
    - pid = wait(&status);
- If no parent waiting (did not invoke wait()) process is a zombie.
- If parent terminated without invoking wait , process is an orphan.

Zombie Process

- A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process.
- A child process always first becomes a zombie before being removed from the process table.
- The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

In the following code, the child finishes its execution using exit() system call while the parent sleeps for 50 seconds, hence doesn't call wait() and the child process's entry still exists in the process table.

A C program to demonstrate Zombie Process.

```
// Child becomes Zombie as parent is sleeping when child process exits.

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

        // Fork returns process id in parent process
```

```
    pid_t child_pid = fork();
    // Parent process
    if (child_pid > 0)
        sleep(50);
    // Child process
    else
        exit(0);
    return 0;
}
```

Orphan Process

- A process whose **parent process no more exists** i.e. either finished or terminated **without waiting** for its child process to terminate is called an orphan process.
- In the following code, parent finishes execution and exits while the child process is still executing and is called an orphan process now.

However, the orphan process is soon adopted by init process, once its parent process dies.

```
int main()

{
    // Create a child process
    int pid = fork();

    if (pid > 0)
        printf("in parent process");
    // Note that pid is 0 in child process

    // and negative if fork() fails

    else if (pid == 0)

    {

        sleep(30);

        printf("in child process");

    }

    return 0;

}
```

// Program to demonstrate bottom to up execution of processes using fork() and wait()

```
#include <iostream>

#include <sys/wait.h> // for wait()

#include <unistd.h> // for fork()

int main()

{

    // creating 4 process using 2 fork calls

    // 1 parent : 2 child : 1 grand-child

    pid_t id1 = fork();

    pid_t id2 = fork();
```

```
// parent process

if (id1 > 0 && id2 > 0) {

        wait(NULL);

        wait(NULL);

        cout << "Parent Terminated" << endl;

}


// 1st child

else if (id1 == 0 && id2 > 0) {



        // sleep the process for 2 seconds

        // to ensure 2nd child executes first

        sleep(2);

        wait(NULL);

        cout << "1st child Terminated" << endl;

}

// second child

else if (id1 > 0 && id2 == 0) {

        // sleep the process for 1 second
```

```
                sleep(1);

                cout << "2nd Child Terminated" << endl;

        }

        // grand child

        else {

                cout << "Grand Child Terminated" << endl;

        }

        return 0;

}
```

Output:

```
Grand Child Terminated

2nd Child Terminated

1st child Terminated

Parent Terminated
```

Predict the Output of the following program.

```
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>
```

```
int main()

{


        // make two process which run same

        // program after this instruction

        fork();


        printf("Hello world!\n");

        return 0;

}
```

Output:

```
Hello world!

Hello world!
```

Calculate number of times hello is printed.

```
#include <stdio.h>

#include <sys/types.h>

int main()

{

        fork();
        fork();

        fork();

        printf("hello\n");

        return 0;

}
```

Output:

```
hello

hello

hello

hello

hello

hello

hello

hello
```

Number of times hello printed is equal to number of process created. Total Number of Processes = 2^n where n is number of fork system calls. So here n = 3, $2^3 = 8$

Predict the Output of the following program

```
#include <stdio.h>

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

```
#include <unistd.h>

void forkexample()

{

        // child process because return value zero

        if (fork() == 0)

                printf("Hello from Child!\n");



        // parent process because return value non-zero.

        else

                printf("Hello from Parent!\n");

}

int main()

{

        forkexample();

        return 0;

}
```

Output:

```
1.

Hello from Child!

Hello from Parent!
```

```
    (or)

2.

Hello from Parent!

Hello from Child!
```

Explanation:

- In the above code, a child process is created, fork() returns 0 in the child process and positive integer to the parent process.
- Here, two outputs are possible because the parent process and child process are running concurrently. So we don't know if OS first give control to which process a parent process or a child process.

Predict output of below program.

```
#include <stdio.h>

#include <unistd.h>

int main()

{

        fork();

        fork() && fork() || fork();

        fork();
        printf("forked\n");

        return 0;

}
```

How to execute zombie and orphan process in a single program?

NOTE:

- If a process's parent is alive but never executes a wait ( ), the process's return code will never be accepted and the process will remain a zombie.
- An orphan process is a process that is still executing, but whose parent has died.

C program to execute zombie and orphan process in a single program

```
#include <stdio.h>

int main()

{



        int x;

        x = fork();



        if (x > 0)

        printf("IN PARENT PROCESS\n PROCESS ID : %d\n", getpid());

        else if (x == 0) {

                sleep(5);
```

```
                    x = fork();
                    if (x > 0) {
printf("IN CHILD PROCESS\n PROCESS ID :%d\n PARENT PROCESS ID : %d\n", getpid(),
getppid());
while(1)
          sleep(1);
printf("IN CHILD PROCESS\nMY PARENT PROCESS ID : %d\n", getppid());
                    }
          else if (x == 0)
printf("IN CHILD'S CHILD PROCESS\n MY PARENT ID : %d\n", getppid());
          }
          return 0;
}
```