Process Description and Control

Process - Process states - Process description - Process control - Processes and Threads
Uniprocessor Scheduling: Types of Processor Scheduling - - Overview of Multiprocessor Scheduling and Real time scheduling- Principles of concurrency - Mutual exclusion- Semaphores – Monitors-
Deadlock and Starvation: Principles of deadlock - Deadlock Prevention - Deadlock

Detection - Deadlock Avoidance.

Process

- A process is a program in execution. A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers.
- A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file). In contrast, a process is an

active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

- Although two processes may be associated with the same program, they are always treated as separate processes.
- For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap,
and stack sections vary.

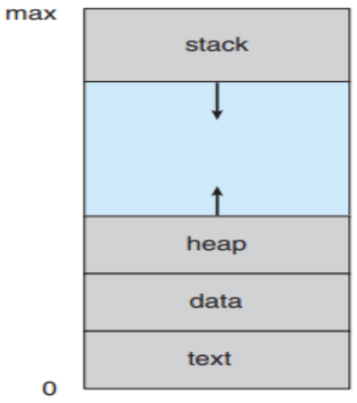


Figure 3.1 Process in memory.

Section	Description
Text	Contains the program code
Program Counter(PC)	Contains address of next instruction
Data	Contains global variables
Heap	Memory that is dynamically allocated during process run time
Stack	Contains temporary data (such as function parameters, return addresses, and local variables)

Process State

The state may be new, ready, running, waiting, halted, and so on.

Process States

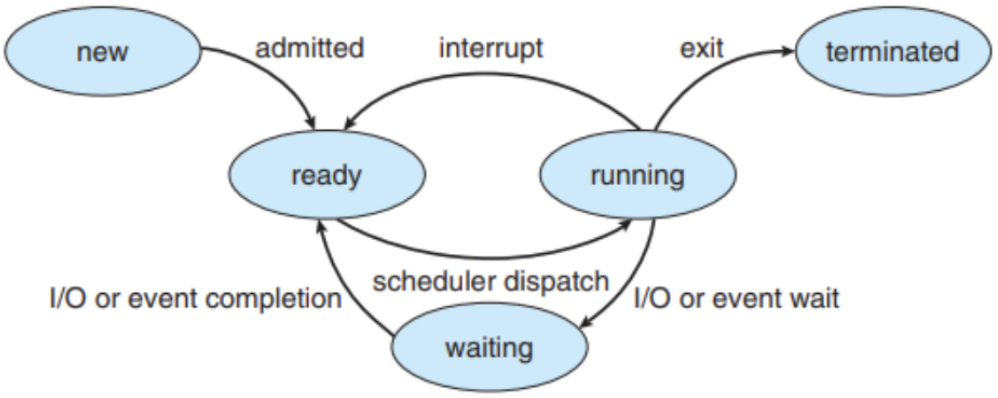


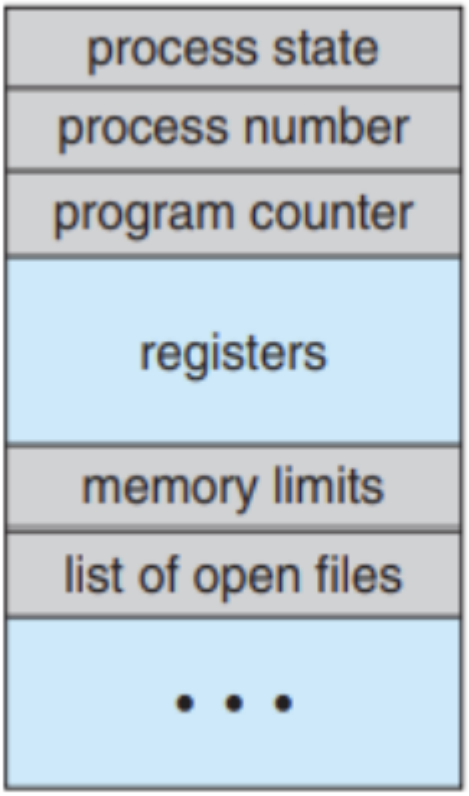
Figure 3.2 Diagram of process state.

- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready**. The process is waiting to be assigned to a processor.
- **Terminated**. The process has finished execution.

What is the difference between process and program?

1. Both are same beast with different name or when this beast is sleeping (not executing) it is called program and when it is executing becomes process.
2. Program is a static object whereas a process is a dynamic object.
3. A program resides in secondary storage whereas a process resides in main memory.
4. The span time of a program is unlimited but the span time of a process is limited.
5. A process is an 'active' entity whereas a program is a 'passive' entity.
6. A program is an algorithm expressed in programming language whereas a process is expressed in assembly language or machine language.

Process Control Block (PCB)



- **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 information as 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.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **status information:** The information includes the list of I/O devices allocated to this process, a list of open files, and so on

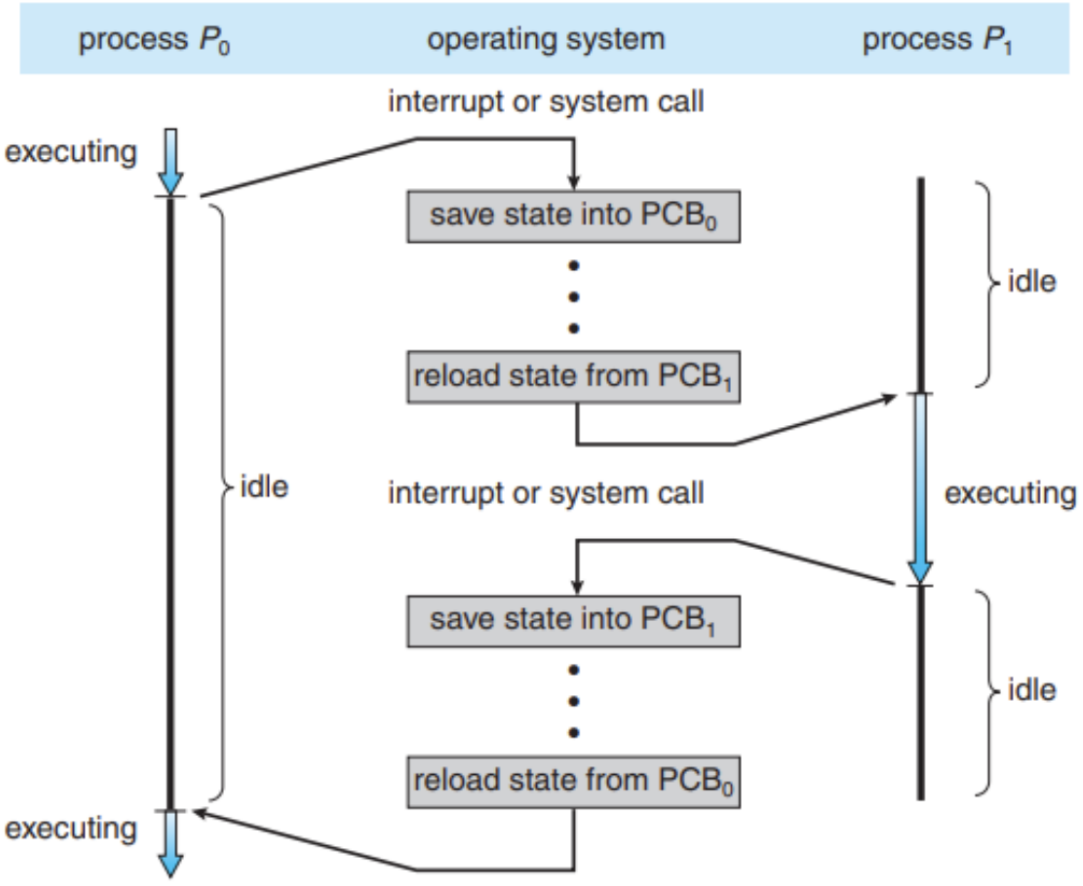
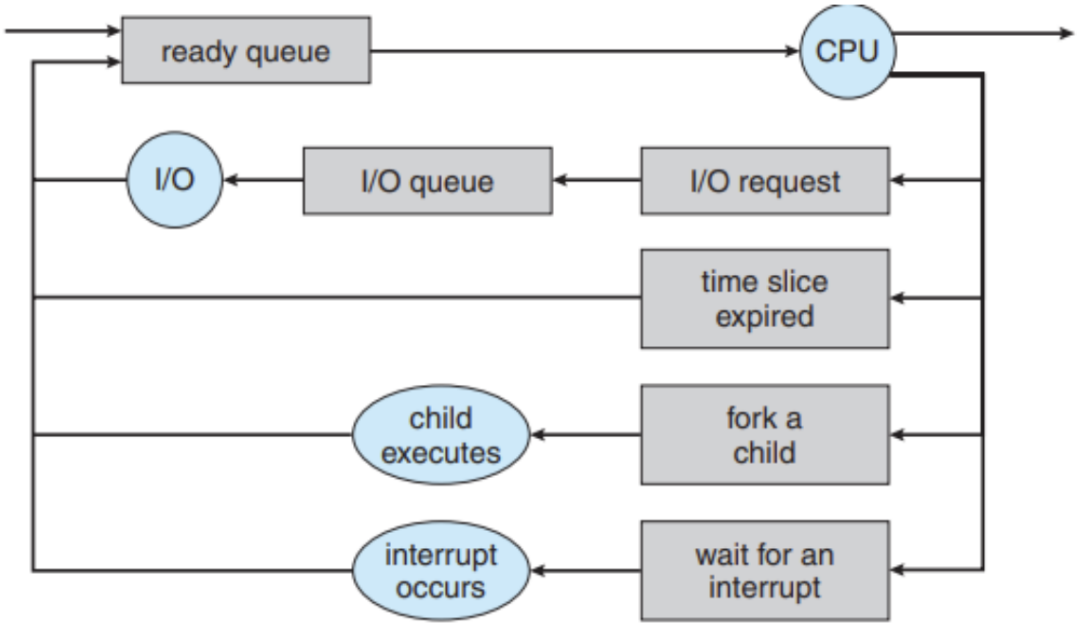


Figure 3.4 Diagram showing CPU switch from process to process.

Process Scheduling Queues

- **Job Queue:** This queue consists of all processes in the system; those processes are entered to the system as new processes.
- **Ready Queue:** This queue consists of the processes that are residing in main memory and are ready and waiting to execute by CPU. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
- **Device Queue:** This queue consists of the processes that are waiting for a particular I/O device. Each device has its own device queue.



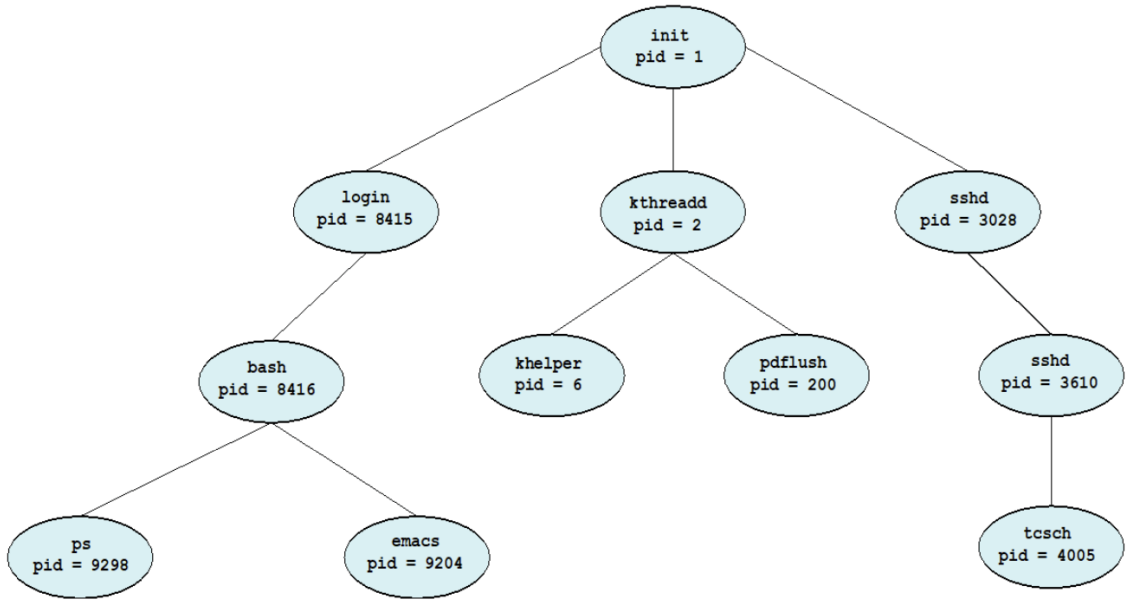
Parent and Child process

- **Parent** process create **children** processes, which, in turn create other processes, forming a
  tree of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- 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

Tree of Processes




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

int main() {
    pid_t pid;

    // Fork a child process
    pid = fork();

    // Check forking error
    if (pid < 0) {
        fprintf(stderr, "Fork failed\n");
```

```
        return 1;
    }

    // Child process
    if (pid == 0) {
        execlp("/bin/ls","ls",NULL);
    }
    // Parent process
    else {
        printf("Parent process is running\n");
    }

    return 0;
}
```

Process Termination

- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Terminologies :

1) Cascading Termination - if a parent process is terminated, all of its child processes are also terminated . parent process will wait until the child process is terminated if wait() is used.

2) Zombie process - A process is terminated but their process has not called wait()

3) Orphans - parent did not invoke wait() but terminated, by leaving the child processes

Example : Producer Consumer Problem

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.

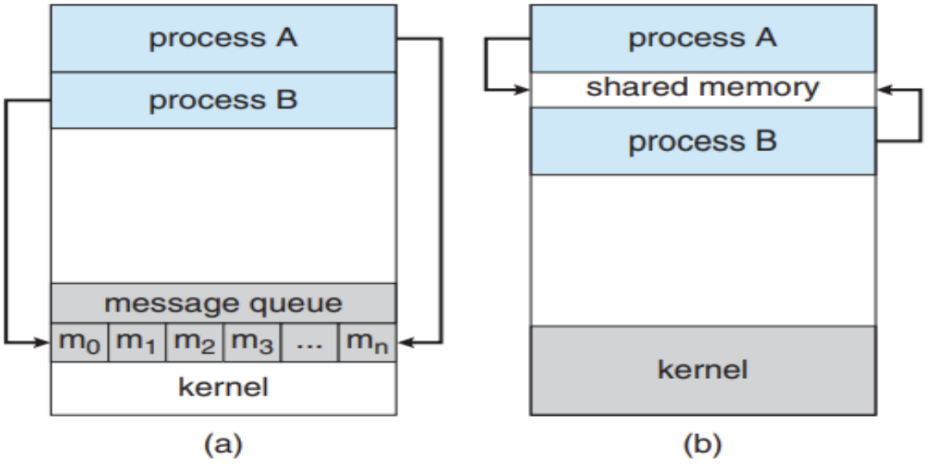


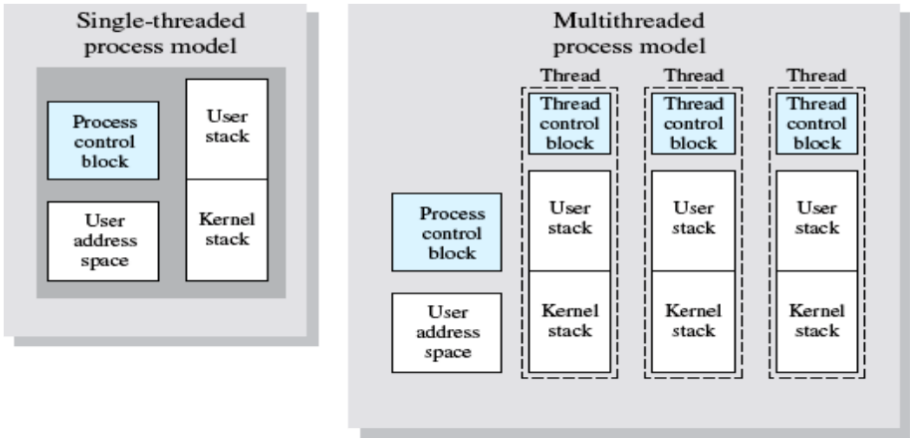
Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

Threads

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, 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 (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.

Single-threaded and multithreaded



Ex: 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 reading keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

BENEFITS OF THREADS

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. For instance, a multithreaded web browser could still allow user interaction in one thread while an image is being loaded in another thread.

2. Resource sharing: By default, threads share the memory and the resources of the process to which they belong.

3. Economy: Allocating memory and resources for process creation is costly. Alternatively, because threads share resources of the process to which they belong, it is more economical to create and context switch threads.

4. Utilization of multiprocessor architectures: The benefits of multithreading can be greatly increased in a multiprocessor architecture, where each thread may be running in parallel on a different processor.

Levels of threads

User-Level Threads

User-level threads implement in user-level libraries, rather than via systems calls, so thread switching does not need to call operating system and to cause interrupt to the kernel. In fact, the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.

Kernel-Level Threads

In this method, the kernel knows about and manages the threads. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. Operating Systems kernel provides system call to create and manage threads.

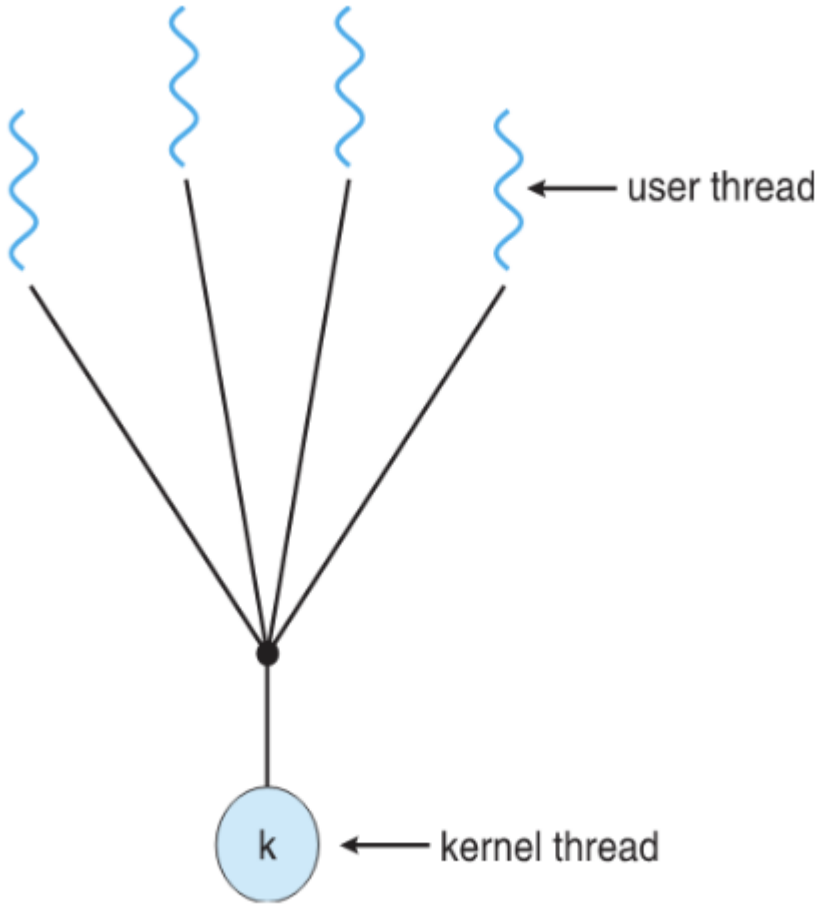
USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are implemented by users.	kernel threads are implemented by OS.
OS doesn't recognized user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Hardware support is needed.

Multi-threading

Many-to-One Model

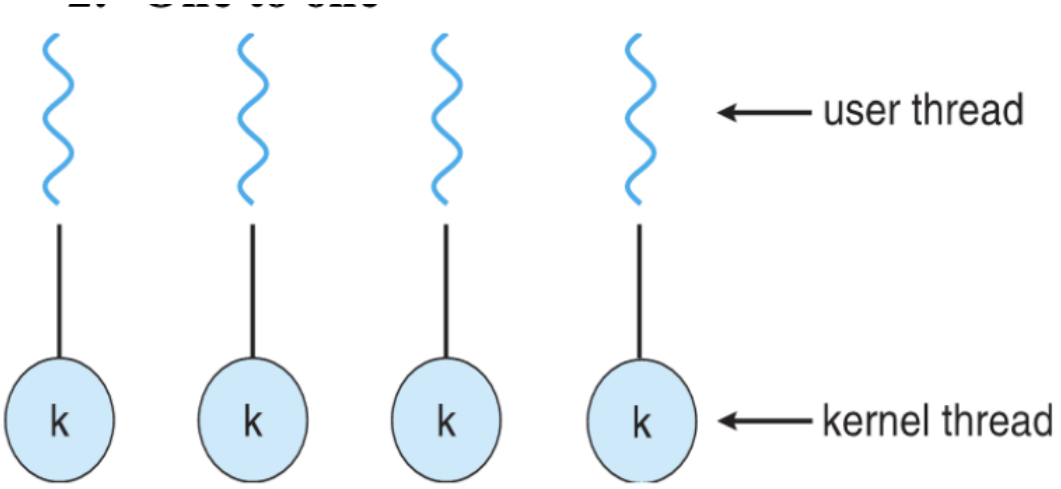
The many-to-one model maps many user-level threads to one kernel thread. Thread management is done in user space, so it is efficient, but the entire

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 multiprocessors



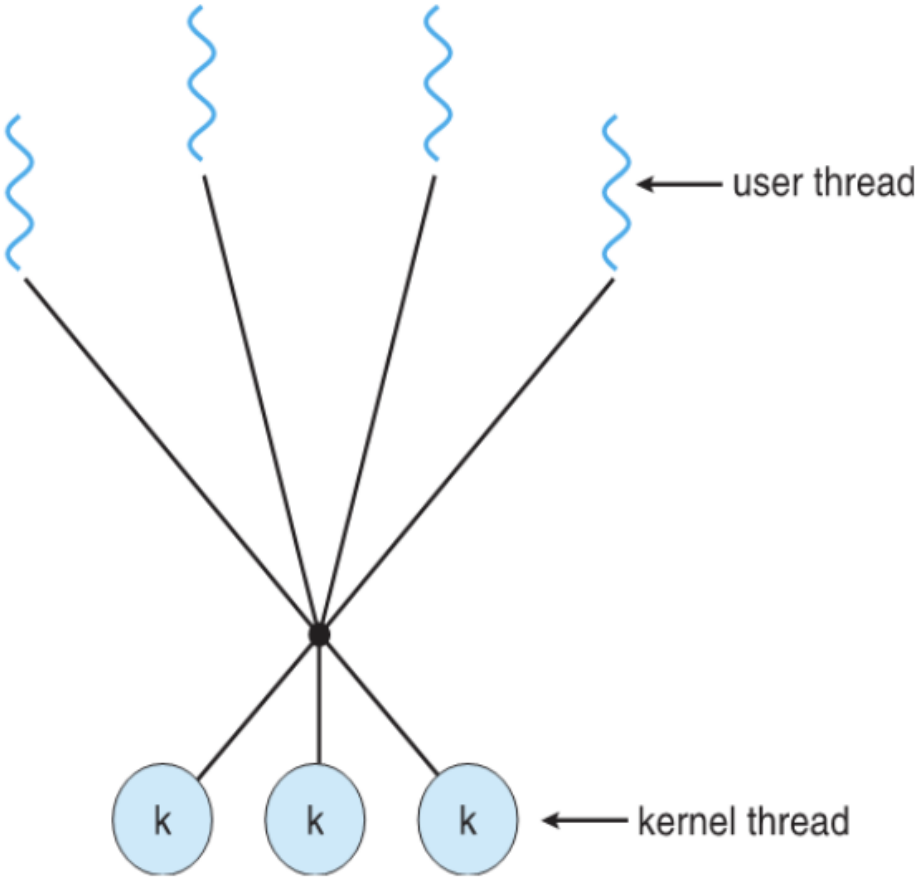
One-to-One Model

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. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.



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 (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor). Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.



PROCESS SYNCHRONIZATION

- Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- **Process synchronization** refers to the coordination of activities (or processes) in a concurrent system to ensure that they behave correctly when accessing shared resources or communicating with each other.
- Shared-memory solution to bounded-butter problem allows at most n – 1 items in buffer at the same time.
- A solution, where all N buffers are used is not simple.
- Suppose that we modify the producer-consumer code by adding a variable counter, initialized to 0 and increment it each time a new item is added to the buffer
- **Race condition:** The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data

depends upon which process finishes last.

- A race condition is a phenomenon that occurs in concurrent systems when the outcome of the execution depends on the relative timing or interleaving of multiple concurrent operations.
- To prevent race conditions, concurrent processes must be synchronized.

Schedulers

A **scheduler** is a decision maker that selects the processes from one scheduling queue to another or allocates CPU for execution. The Operating System has three types of scheduler:

1. Long-term scheduler or Job scheduler

2. Short-term scheduler or CPU scheduler

3. Medium-term scheduler or swapping scheduler

Long-term scheduler or Job scheduler

- the process is taken from job pool
- The long-term scheduler or job scheduler selects processes from discs and loads them into main memory for execution. It executes much less frequently.
- It controls the degree of multiprogramming (i.e., the number of processes in memory).
- Because of the longer interval between executions, the long-term scheduler can afford to take more time to select a process for execution.

Short-term scheduler or CPU scheduler

- The short-term scheduler or CPU scheduler selects a process from among the processes that are ready to execute and allocates the CPU.
- the process is taken from read state
- The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request.

Medium-term scheduler

- The medium-term scheduler schedules the processes as intermediate level of scheduling

- the process is taken from ready state

Processes can be described as either:

- I/O-bound process – spends more time doing I/O than computations, many short CPU bursts.
- CPU-bound process – spends more time doing computations; few very long CPU bursts.

Context Scheduling

❖ Context switching is a technique or method used by the operating system to
switch a process from one state to another to execute its function using CPUs in the system.

❖ When switching is performed in the system, it stores the old running process's
PCB in the form of registers and assigns the CPU to a new process by reloading the PCB of the new process to execute its tasks.

❖ 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 single instruction to load or store all registers). A typical speed is a few milliseconds.

CPU SCHEDULING ALGORITHMS

When scheduling is needed?

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)
2. When a process switches from the running state to the ready state (for example,
   when an interrupt occurs)

3. When a process switches from the waiting state to the ready state (for example,
   at completion of I/O)
4. When a process terminates

Two types of Scheduling
Preemptive Scheduling:

Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to ready state. The resources (mainly CPU cycles)
are allocated to the process for a limited amount of time and then taken away, and the
process is again placed back in the ready queue if that process still has CPU burst time
remaining.

- In preemptive scheduling, the operating system has the ability to interrupt a running process or thread and allocate the CPU to another process or thread.
- Preemptive scheduling ensures fairness and responsiveness in the system by allowing higher-priority tasks to preempt lower-priority tasks.
- When a higher-priority process becomes ready to execute, the operating system can preempt the currently running process and allocate the CPU to the higher-priority process.
- Common preemptive scheduling algorithms include Round Robin, Priority Scheduling, and Multilevel Queue Scheduling.

Non-Preemptive Scheduling:

Non-preemptive Scheduling is used when a process terminates, or a process switches from running to the waiting state. In this scheduling, once the resources
(CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state.

- In non-preemptive scheduling, a process or thread continues to run until it voluntarily relinquishes the CPU, such as by blocking on I/O or completing its execution.

- Once a process or thread starts executing, it will not be preempted by the operating system until it either completes its execution or enters a waiting state.
- Non-preemptive scheduling can lead to situations where a long-running process or a high-priority process monopolizes the CPU, causing lower-priority tasks to experience delays or starvation.
- Common non-preemptive scheduling algorithms include First Come, First Served (FCFS) and Shortest Job Next (SJN).

Scheduling Criteria

1. **CPU utilization :** We want to keep the CPU as busy as possible. Conceptually, CPU
   utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).
2. **Throughput :** It is the number of processes that are completed per time unit, called
   throughput. For long processes, this rate may be one process per hour; for short
   transactions, it may be ten processes per second.
3. **Turnaround time :** The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting
   to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
4. **Waiting time :** The CPU-scheduling algorithm does not affect the amount of time during
   which a process executes or does I/O. It affects only the amount of time that a
   process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
5. **Response time**

- In an interactive system, turnaround time may not be the best criterion.

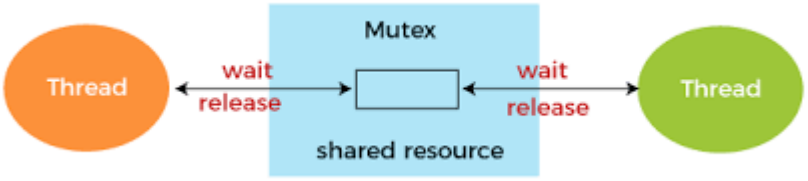
- Response time is the time from the submission of a request until the first response is produced. It is the time it takes to start responding, not the time it

  takes to output the response.

Mutex

Mutex is a specific kind of binary semaphore that is used to provide a locking mechanism. It stands for mutual exclusion Object. Mutex is mainly used to provide mutual exclusion to a specific portion of the code so that the process can execute and work with a particular section of the code at a particular time.

- A Mutex is a lock that we set before using a shared resource and release after using it.
- When the lock is set, no other thread can access the locked region of code.



The purpose of a mutex is to ensure that only one thread or process can access a shared resource at any given time, preventing data races, race conditions, and other forms of concurrent access conflicts. mutex locking uses functions such as `acquire()` and `release()`

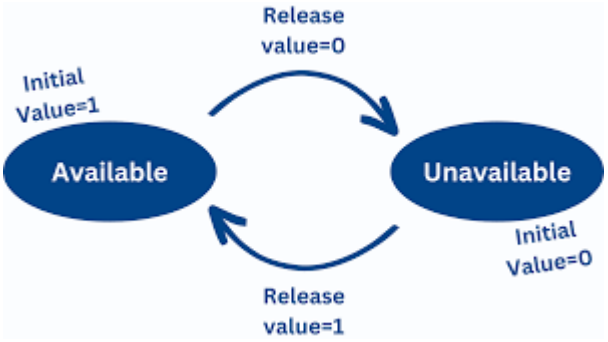
A mutex typically provides two fundamental operations:

1. **Locking**: A thread or process acquires the mutex lock before accessing the shared resource. If the mutex is not already locked by another thread or process, the locking operation succeeds, and the thread or process gains exclusive access to the resource. If the mutex is already locked, the locking operation will block the thread or process until the mutex becomes available.

2. **Unlocking**: After a thread or process finishes using the shared resource, it releases the mutex lock by invoking the unlocking operation. This allows other threads or processes that are waiting for the mutex to acquire it and access the resource.

Semaphores

Semaphores are just normal variables used to coordinate the activities of multiple processes in a computer system. They are used to enforce mutual exclusion, avoid race conditions, and implement synchronization between processes.



A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait** and **signal**.

The process of using Semaphores provides two operations: wait (P) and signal (V). The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore. When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.

The classical definition of wait in pseudocode is

```
wait(S) {

while (S <= 0)

; // no-op
```

```
S --;

}

The classical definitions of signal in pseudocode is

Signal(S){

S++;

}
```

Monitors

A monitor is a high-level synchronization construct used in concurrent programming to manage access to shared resources by multiple threads or processes. It provides a structured way to enforce mutual exclusion and coordinate the execution of concurrent activities.

- A monitor type is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor.
- The monitor type also declares the variables whose values define the state of an
  instance of that type, along with the bodies of functions that operate on those
  variables.
- A function defined within a monitor can access only those variables declared
  locally within the monitor and its formal parameters.
- The monitor construct ensures that only one process at a time is active within
  the monitor.

```
Monitor monitorName{
    variables_declaration;
```

```
    condition_variables;

    procedure p1{ ... };
    procedure p2{ ... };
    ...
    procedure pn{ ... };

    {
        initializing_code;
    }

}
```

Deadlocks

A set of process is in a deadlock state if each process in the set is waiting for an event that can be caused by only another process in the set. In other words, each member of the set of deadlock processes is waiting for a resource that can be released only by a deadlock process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The resources may be either physical or logical. Examples of physical resources are Printers, Hard Disc Drives, Memory Space, and CPU Cycles.

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that
can be caused only by one of the waiting processes. The event in question is the
execution of a signal() operation. When such a state is reached, these processes
are said to be
deadlocked.

Deadlock is a situation in concurrent programming where two or more processes or threads are unable to proceed because each is waiting for the other to release a resource, resulting in a standstill.

1. **Resource Allocation Deadlock**: In a multi-process system, processes may compete for resources such as memory, files, or network connections. If two or more processes acquire resources in a non-preemptive manner and hold one resource while waiting for another that is held by another process, a deadlock can occur. For example:
    - Process A holds Resource 1 and waits for Resource 2.
    - Process B holds Resource 2 and waits for Resource 1.
      Both processes are blocked, leading to a deadlock.
2. **Database Deadlock**: In database systems, transactions may acquire locks on database records or tables to ensure data consistency. If transactions acquire locks in a different order, deadlock can occur. For example:
    - Transaction T1 acquires a lock on Record 1 and waits to acquire a lock on Record 2.
    - Transaction T2 acquires a lock on Record 2 and waits to acquire a lock on Record 1.
      Both transactions are blocked, resulting in a deadlock.

METHODS OF HANDLING DEADLOCK

In general, there are four strategies of dealing with deadlock problem:

1. **Deadlock Prevention**: Prevent deadlock by resource scheduling so as to negate at least one of the four conditions.

2. **Deadlock Avoidance**: Avoid deadlock by careful resource scheduling.

3. **Deadlock Detection and Recovery**: Detect deadlock and when it occurs, take steps to recover.

4. **The Ostrich Approach:** Just ignore the deadlock problem altogether.

Indefinite blocking or starvation, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove
processes from the list associated with a semaphore in LIFO (last-in, first-out)
order.

Deadlock Characterization - necessary conditions

1. **Mutual exclusion.** At least one resource must be held in a nonsharable mode;
that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2.
Hold and wait. A process must be holding at least one resource and waiting to
acquire additional resources that are currently being held by other processes.

3.
No preemption. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4.
Circular wait. A set {P0, P1, ..., Pn} of waiting processes must exist such that
P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2,
..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource
held by P0.