MODULE - 2

Multithreaded Programming

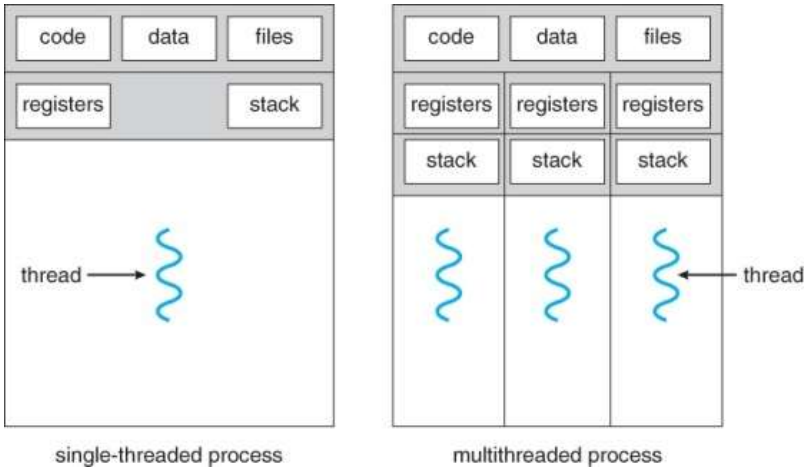
A thread is a ***basic unit of CPU utilization***. Sometimes it is also called as Light Weight Process (LWP).

A thread ***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.

Processes can be of two types:

1. Single threaded process
2. Multiple threaded process

Single threaded process will have one stream of execution, where as a multithreaded process contains multiple stream of execution as shown below.



Multithreading: A process is divided into smaller tasks and each task is called a thread. ***The use of multiple threads in a single program, all running at the same time and performing different tasks is called multithreading***

Ex: A browser is a multithreaded application where you can scroll a page while it is downloading an image, play animation or sound parallel, print a page on the background, while also saving the page on the local system.

1. Differentiate between
   a. Process and threads
   b. User level threads and Kernel Level threads
2. Explain benefits of multithreaded programming [8M]

Benefits of multithreaded programming

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 allow user interaction in one thread while an image was being loaded in another thread.
2. **Resource sharing**: Unlike processes, *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.***
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***.
4. **Scalability:**. A single-threaded process can only run on one processor, regardless how many are available. The *benefits of multithreading can be greatly increased in a multiprocessor architecture*, where threads may be running in parallel on different processors Multithreading on a multi-CPU machine ***increases parallelism***.

Multicore Programming

A multicore system places multiple computing cores on a single chip, where each core appears as a separate processor to the operating system

On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time

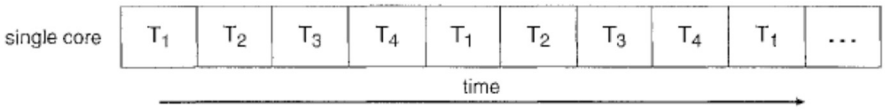


Figure 4.3 Concurrent execution on a single-core system.

On a system with multiple cores, however, concurrency means that the threads can run in parallel, as the system can assign a separate thread to each core

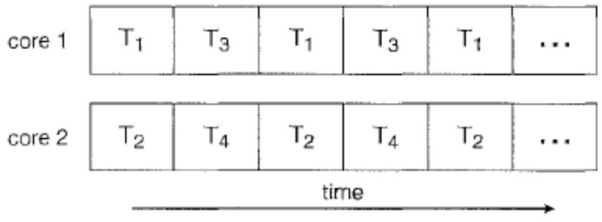


Figure 4.4 Parallel execution on a multicore system.

The trend towards multicore systems has placed pressure on system designers as well as application programmers to make better use of the multiple computing cores. For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded to take advantage of multicore systems.

Multithreading Models

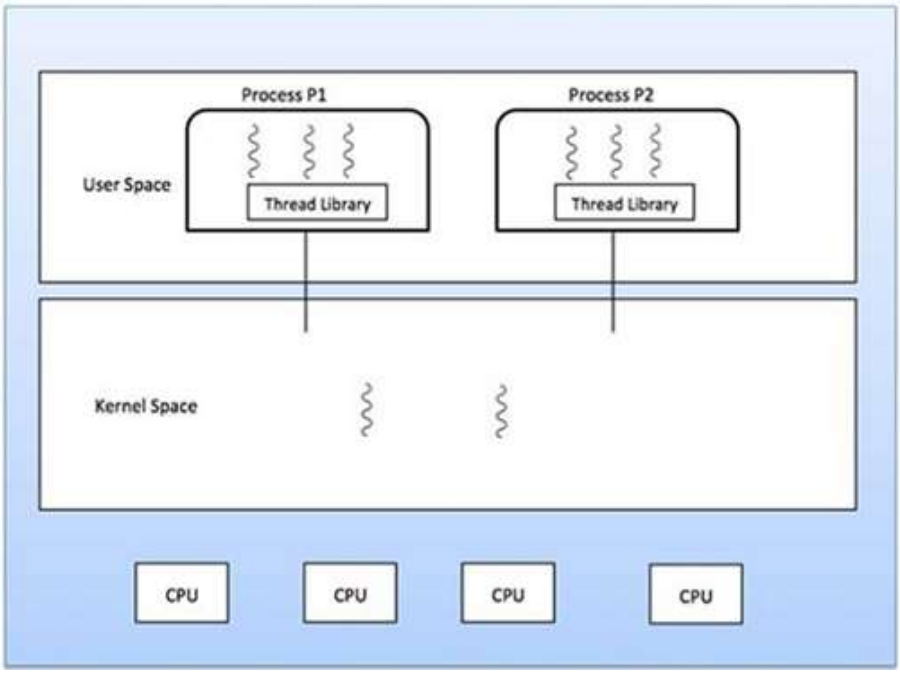
1. **Discuss three common ways of establishing relationship between user thread and kernel thread** [6M]
2. Why is thread called LWP? Describe any one threading model and cite an OS that implements it. Also explain one of the many threading issues [7M]
3. Why is thread called a LWP? **Explain different threading models.** Bring out the concept of thread pool [7M]
4. Write short note on user level thread and kernel level thread

There are two types of threads in a system.

1. **User thread**
2. **Kernel thread**

User Threads

User threads are supported in the user space and the kernel doesn't know about it. The ***thread library*** contains code for *creating and destroying* threads, for *passing message and data* between threads, for *scheduling* thread execution and for *saving and restoring thread contexts*.



Advantages of User Level thread

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages:

- Cannot be scheduled for execution directly on the processors.

Kernel Threads

Kernel threads exist in the kernel space and is managed by the OS. The application need not bother about kernel thread creation, destruction, scheduling, etc. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages:

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

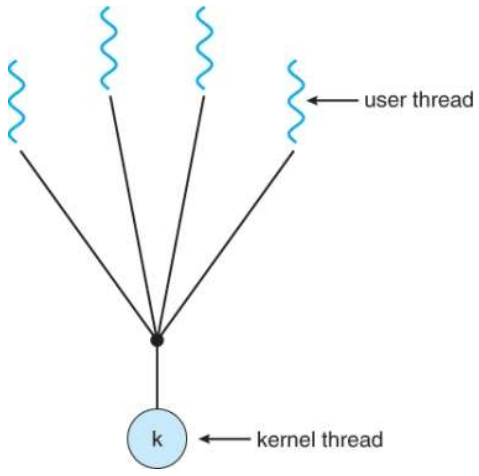
Multithreading Models

Some operating system provide a combined user level thread and Kernel level thread facility. Multithreading models are three types:

1. **Many to one**
2. **One to one**
3. **Many to Many**

Many-to-One Model

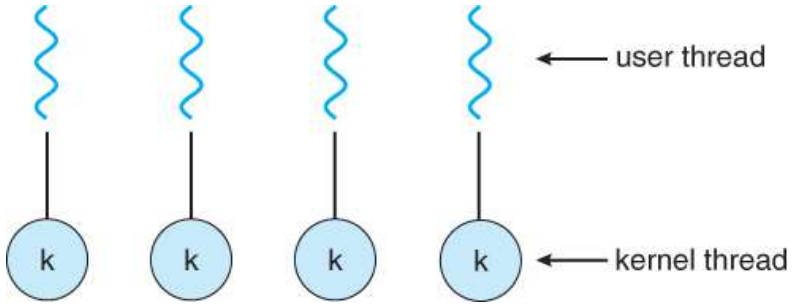
- The many-to-one model maps many user-level threads to one kernel thread.



- Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call.
- Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- Solaris OS uses many-to-one model for its “green threads” and so does GNU portable threads

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. More kernel threads ➔ reduced performance. Hence systems may restrict the number of threads that can be created
- OS that support one-to-one model➔ Linux and Windows OS

Many-to-Many Model

- The many-to-many model multiplexes many user-level threads *to a smaller or equal number of kernel threads*.
- Developers *can create as many user threads* as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- When a thread performs a blocking system call, the kernel can schedule another thread for execution.

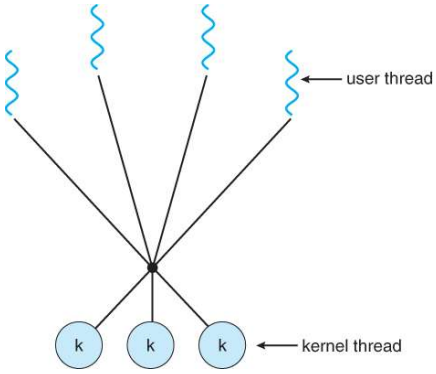


fig (1)

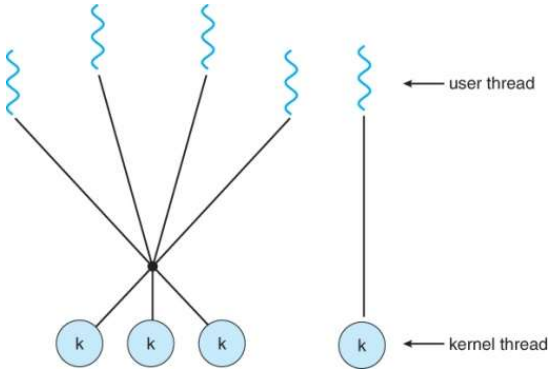


fig (2)

One popular 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 sometimes referred to as the *two-level model* (as shown in fig 2 above).

Supported by IRIX, HP-UX, and Tru64 UNIX operating systems

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.

1. 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.
2. 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.

Thread libraries provide support for

- creating and destroying threads, passing messages and data between threads, scheduling thread execution, saving and restoring thread contexts

Three main thread libraries are in use today:

(1) **POSIX Pthreads**,- Pthreads, the threads extension of the POSIX standard, may be provided as either a user- or kernel-level library.
(2) **Win32**,- is a kernel-level library available on Windows systems.
(3) **Java**.- The Java thread API allows threads to be created and managed directly in Java programs.

1. Write short note on pthreads [4M]

Pthreads

- The POSIX standard (IEEE 1003.1c) defines the *specification* for pThreads, not the *implementation*. pThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.
- Global variables are shared amongst all threads. One thread can wait for the others to rejoin before continuing.
- pThreads begin execution in a specified function, in this example the runner( ) function

Ex: A multithreaded program that computes summation of non-negative integers in a separate thread

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

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

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


/* get the default attributes */
pthread_attr_init(&attr);

/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);

/* now wait for the thread to exit */
pthread_join(tid,NULL);

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

}
```

```c
void *runner(void *param)
{
int i = 1, upper = atoi(param);

        if (upper > 0) {
                for (i = 1; i < upper; i++)
                        {
                                printf("\nThread #1: %d + %d = ", sum, i);
                                sum += i;
                                printf("%d", sum);
                        }
        }
        pthread_exit(0);
}
```

The C program shown above demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a nonnegative integer in a separate thread. When this program begins, a single thread of control begins in main (). After some initialization, main () creates a second thread that begins control in the runner () function. Both threads share the global data sum.

All Pthreads programs must include the **pthread. h** header file. The statement **pthread_t tid** declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The **pthread_attr_t attr** declaration represents the attributes for the thread. We set the attributes in the function call **pthread_attr_init (&attr).**

A separate thread is created with the **pthread_create ()** function call. This function receives as parameters, a reference to the thread id, its attributes, name of the function which has to be loaded on a separate thread and finally a set of arguments (if any) to the called function.

After creating the summation thread the parent thread will wait for it to complete by calling the pthread_join () function. The summation thread will complete when it calls the function pthread_exit (). Once the summation thread has returned the parent thread will output the value of the shared data sum.

Win32 Threads

The technique for creating threads using the Win32 thread library is similar to the Pthreads technique in several ways. The implementation of the same task using Win32 threads is shown below (next page)

We must include the **windows.h** header file when using the Win32 API

Threads are created in the Win32 API using the **CreateThread** () function, and-just as in Pthreads-a set of attributes for the thread is passed to this function. These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state.

Once the summation thread is created, the parent must wait for it to complete before outputting the value of Sum, as the value is set by the summation thread. The parent can be made to wait until thread completes using **WaitForSingleObject**() function, which causes the creating thread to block until the summation thread has exited.

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

DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function*/
DWORD WINAPI Summation(LPVOID Param) {
  DWORD Upper = *(DWORD*)Param;
  for (DWORD i = 0; i <= Upper; i++) {
    Sum += i;
  }
  return 0;
}
```

```
int main(int argc, char *argv[]) {
  DWORD ThreadId;
  HANDLE ThreadHandle;
  int Param;

  // create the thread
  ThreadHandle = CreateThread(
    NULL, // default security attributes
    0, // default stack size
    Summation, // thread function
    &Param, // parameter to thread function
    0, // default creation flags
    &ThreadId); // returns the thread identifier

  if (ThreadHandle != NULL) {
    // now wait for the thread to finish
    WaitForSingleObject(ThreadHandle, INFINITE);

    // close thr thread handle
    CloseHandle(ThreadHandle);

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

Java Threads

- The creation of new Threads requires ***Objects that implement the Runnable Interface***, which means they contain a method "**public void run( )**" . Any descendant of the Thread class will naturally contain such a method. (In practice the run( ) method must be overridden / provided for the thread to have any practical functionality.)
- Creating a Thread Object does not start the thread running - To do that the program must call the Thread's "**start()**" method. Start( ) allocates and initializes memory for the Thread, and then calls the run( ) method. (Programmers do not call run( ) directly.)
- Because Java ***does not support global variables***, Threads must be passed a reference to a shared Object in order to share data, in this example the "Sum" Object.

```
class Sum
{
  private int sum;

  public int getSum() {
    return sum;
  }

  public void setSum(int sum) {
    this.sum = sum;
  }
}
```

```
class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
    this.upper = upper;
    this.sumValue = sumValue;
  }

  public void run() {
    int sum = 0;
    for (int i = 0; i <= upper; i++)
      sum += i;
    sumValue.setSum(sum);
  }
}

public class Driver
{
  public static void main(String[] args) {
    if (args.length > 0) {
      if (Integer.parseInt(args[0]) < 0)
        System.err.println(args[0] + " must be >= 0.");
      else {
        // create the object to be shared
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        Thread thrd = new Thread(new Summation(upper, sumObject));
        thrd.start();
        try {
          thrd.join();
          System.out.println
                  ("The sum of "+upper+" is "+sumObject.getSum());
        } catch (InterruptedException ie) { }
      }
    }
    else
      System.err.println("Usage: Summation <integer value>"); }
}
```

Figure 4.11 Java program for the summation of a non-negative integer.

Threading Issues

1. Why is thread called LWP? Describe any one threading model and cite an OS that implements it. Also explain one of the many threading issues [7M]
2. Discuss any 3 threading issues that comes with multithreaded programs [6M]
3. Why is thread called a LWP? **Explain different threading models.** Bring out the concept of thread pool [7M]

Threading Issues

There are a variety of issues to consider with multithreaded programming –

- Semantics of fork() and exec() system calls
- Thread cancellation
- Signal handling
- Thread pooling
- Thread-specific data
- Scheduler Activations

Semantics of fork() and exec() system calls

- Normally when fork() is called, a separate, duplicate process is created. The issue of fork() in a multi-threaded program is: when a multithreaded program invokes a fork()
  - → Should all threads be duplicated?
  - → Should only the thread that made the call to fork() be duplicated?
- On some systems, *different versions of fork() exist depending on the desired behavior*. Some UNIX systems have fork1() and forkall()
  - → fork1() only duplicates the calling thread
  - → forkall() duplicates all of the threads in a process
- The exec() call replaces the entire process including all the threads. As such, if fork() is called with exec(), there is no need to duplicate all the threads in the calling process. If only fork() is invoked, all the threads may be duplicated.

Thread Cancellation

- Thread cancellation is the act of terminating a thread before it has completed - Example - clicking the stop button on your web browser will stop the thread that is rendering the web page
- The cancellation of threads may take place in 2 different ways:
  1. **Asynchronous cancellation -** cancels the thread immediately. If the thread was in the middle of some important task such as updating data, asynchronous cancellation can leave the system in an inconsistent state.
  2. **Deferred cancellation -** sets a flag indicating the thread should cancel itself when it is convenient. The target thread checks the flag periodically to determine if it is scheduled for termination. If the flag is set, the thread terminates itself. Deferred cancellation allows thread to terminate itself in an orderly fashion

Signal Handling

- Signals are used in UNIX systems to *notify a process that a particular event has occurred*. All signals, whether synchronous or asynchronous [synchronous – signals delivered to the same process that caused the signal, asynchronous – signals generated from outside of the process], follow the same pattern:
  - → A signal is generated by the occurrence of a particular event.
  - → A generated signal is delivered to a process
  - → Once delivered, the signal must be handled
- Handling signals in a *single threaded program is straight forward* –The one (and only) thread in the process receives and handles the signal.
- However in multi-threaded program, when a signal arrives where should the signal be delivered?
  In general 4 options exist
  - → *Deliver the signal to the thread to which the signal applies.*
  - → *Deliver the signal to every thread in the process.*
  - → *Deliver the signal to certain threads in the process.*
  - → *Assign a specific thread to receive all signals for the process*
- The *best choice may depend on the type of signal involved*. Ex: - Divide by 0 is an example of a synchronous signal that might be sent to a process. CTRL-Z on command terminal is an example of an asynchronous signal that might be sent to a process (i.e. to all threads thus forcing all threads to terminate)
- Unix provides the function **kill()** and **pthread_kill()** *to deliver signals* to processes and threads
- Windows does not support signals, but provides a similar feature with **Asynchronous Procedure Calls**

Thread Pools

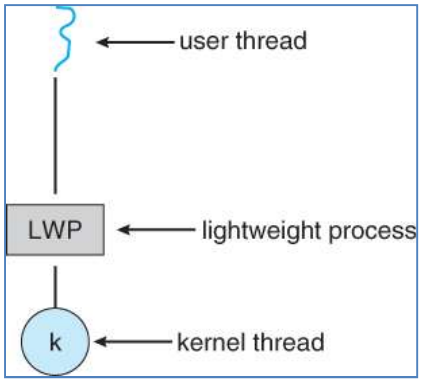
- In applications where threads are repeatedly being created/destroyed thread pools might provide a performance benefit - Example: A server that spawns a new thread each time a client connects to the system and discards that thread when the client disconnects
- A thread pool is a *group of threads that have been pre-created and are available* to do work as needed
  - → Threads may be *created when the process starts*
  - → A thread may be *kept in a queue until it is needed*
  - → After a thread finishes, it is *placed back into a queue* until it is needed again
  - → *Avoids the extra time needed* to create new threads when they're needed
- **Advantages**
- Typically faster to service a request with an existing thread than create a new thread
- **Drawbacks** - Bounds the number of threads in a process
  - → The only threads available are those in the thread pool
  - → If the thread pool is empty, then the process must wait for a thread to re-enter the pool before it can assign work to a thread

Thread Specific Data

- Most data is shared among threads in a process. However, in some applications it may be useful for each thread to have its own copy of data
- Most major thread libraries (pThreads, Win32, Java) provide support for thread-specific data, known as **thread-local storage** or **TLS.** This data is more like static data than local variables, because it does not cease to exist when the function ends.

Scheduler Activations

- Scheduler Activation is the technique used for communication between the user-thread library and the kernel.
- It works as follows:
  - → the kernel must inform an application about certain events. This procedure is known as an upcall.
  - → Upcalls are handled by the thread library with an upcall handler, and upcall handlers must run on a virtual processor.
- Example - The kernel triggers an upcall occurs when an application thread is about to block. The kernel makes an upcall to the thread library informing that a thread is about to block and also informs the specific ID of the thread. The upcall handler handles this thread, by saving the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running.



- The upcall handler then schedules another thread that is eligible to run on the virtual processor. When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. Thus assigns the thread to the available virtual processor.

Module – 2 : Process Scheduling

CPU scheduling is the basis of multi-programmed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

Basic Concepts

Earlier computer systems allowed only one process to execute at a time. This process would have complete control over CPU and had access to all system resources. In contrast modern computer systems allow multiple programs to be loaded into memory and executed concurrently. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, this time is used productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.

CPU-I/O Burst Cycle

→ Any process execution consists of a **cycle** of **CPU execution** and **I/0 wait**. Processes alternate between these two states.
→ Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/0 burst, and so on.
→ Eventually, the final CPU burst ends with a system request to terminate execution (as shown in figure 1).

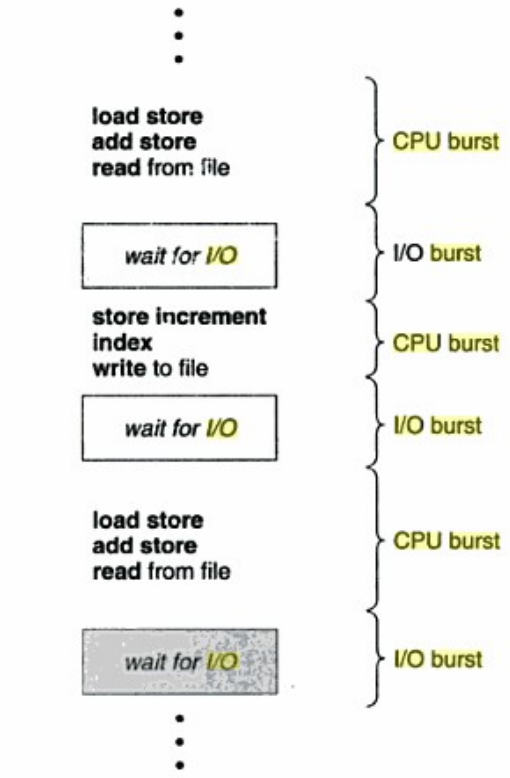


Figure 1: Alternating sequence of CPU and I/O bursts

→ An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

CPU Scheduler

1. **What is Scheduler? What is a dispatcher?**
2. **Differentiate the following with examples**
   a. **Preemptive and non preemptive scheduling**
   b. **I/O bound and CPU bound**
   c. **Scheduler and dispatcher**
3. **Differentiate Pre-emptive and Non-preemptive scheduling giving the application of each of them**
4. **Describe throughput and response time in multiprogramming system [2M]**
5. **Explain different criteria that must be kept in mind when choosing scheduling algorithms [5M]**

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Preemptive & Non-Preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process *switches from the running state to the waiting state* (for example, as the result of an I/0 request or an invocation of *wait* for the termination of one of the child processes).
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/0)
4. When a *process terminates*.

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

→ When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative; otherwise, it is preemptive.

→ Under **nonpreemptive scheduling,** once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

→ With **preemptive scheduling**, a *process under execution may be forced to preempt the CPU* as a result of an interrupt, or an arrival of higher priority process

Non-Preemptive Scheduling	Preemptive Scheduling
1. Once a process has been allotted the CPU, then the CPU cannot be taken away from that process	1. In preemptive scheduling, the CPU can be taken away before the process completes execution
2. No preference is given when a higher priority process arrives	2. When a higher priority task arrives, the current process is forced to release the CPU
3. All processes are treated fairly	3. Certain processes are given higher priority based on several criteria – shorter time to execute, importance of the process, etc
4. Ex: First Come First Served	4. Ex: Priority Scheduling, Round Robin Scheduling

Dispatcher

→ Another component involved in the CPU-scheduling function is the dispatcher.

→ The dispatcher is ***the module that gives control of the CPU to the process selected by the short-term scheduler***.

→ This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

→ The dispatcher should be as fast as possible, since it is invoked during every process switch.

→ The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Scheduling criteria

A scheduler algorithm is evaluated against some widely accepted performance criteria:

1. **CPU utilization**: It is defined as the ***average faction of time during which CPU is kept busy executing either user programs or system modules***. A good scheduling algorithm succeeds ni keeping CPU busy at all times.
2. **Throughput**: Throughput is the ***average amount of work completed per unit time***. One wy to measure throughput is by mean of the number of processes completed in unit time.
3. **Turnaround time**: It is the ***total time elapsed from the time the process is submitted (or created) to the time the process is completed***. 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/0.
4. **Waiting time**: It is the ***total time spent by the process while waiting in suspended state or ready state (in the ready queue)*** in a multiprogrammed environment.
5. **Response time**: For an interactive system, turnaround time may not be a good judging criterion. Response time refers to ***the time from the submission of a request until the first response is produced*** . Response time is the

time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

Scheduling Algorithms

1. Explain round robin scheduling policy [6M]
2. What is the criterion used to select the time quantum in case of round-robin scheduling algorithm? Explain it with a suitable example.
3. Explain multilevel feedback queue scheduling with example.

First-Come, First-Served (FCFS) Scheduling

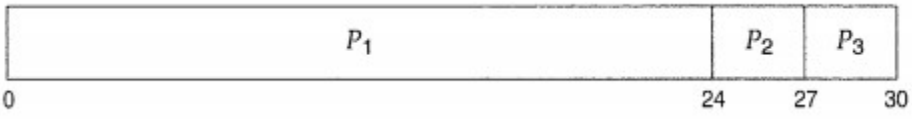
→ The FCFS scheduling is the *simplest and most straight-forward scheduling algorithm*. It is a fair scheduling scheme where the *process that requests the CPU first is allocated the CPU first*.

→ FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following **Gantt chart,** *which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:*



The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2 , and 27 milliseconds for process P3 . Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.

The average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes CPU burst times vary greatly.

Suppose If the processes arrive in the order P2, P3, P1, the average waiting time will differ as follows:

The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds.

→ FCFS scheduling algorithm is nonpreemptive.

Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/0.

→ The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

Benefits:

- Fair scheduling scheme
- Easy to understand and implement

Drawbacks

FCFS policy may sometimes introduce the Convoy Effect - **A condition where many shorter/ faster jobs are forced to wait for longer [CPU intensive] jobs to complete, thereby reducing the overall performance of the system**

Convoy Effect Ex: When one CPU intensive process blocks the CPU, a number of I/O intensive processes can get backed up behind it, leaving the I/O devices idle. When the CPU hog finally relinquishes the CPU, then the I/O processes pass through the CPU quickly, leaving the CPU idle while everyone queues up for I/O, and then the cycle repeats itself when the CPU intensive process gets back to the ready queue

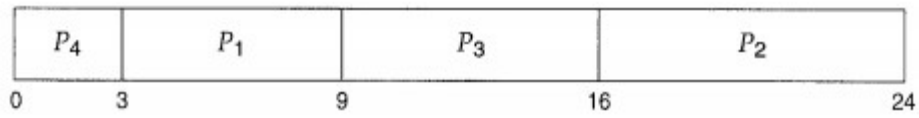
Shortest-Job-First Scheduling

→ This algorithm associates with each process the length of the process's next CPU burst.
→ When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
→ In this scheduling method scheduling ***depends on the length of the next CPU burst of a process, rather than its total length.***
→ Sometimes it is also referred to as **shortest-next-CPU-burst algorithm**.

Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Gantt chart using SJF scheduling is shown below.



The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0) /4 = 7 milliseconds.

By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes.

One of the MAJOR CHALLENGES with SJF scheduling is knowing the length of the next CPU burst

→ For long-term (job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job. Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response. SJF scheduling is used frequently in long-term scheduling

→ It is difficult to implement SJF scheduling for short-term CPU scheduling because there is no way to know the length of the next CPU burst.

One approach is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but **we may be able to predict its value**. We expect that the next CPU burst will be similar in length to the previous ones. ***By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.***

The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. We can define the exponential average with the following formula. Let t_n be the length of the nth actual CPU burst and Γ_n be the predicted value for the n^{th} CPU burst, then we can compute the predicted value for the $n+1^{th}$ CPU burst as:

$$\Gamma_{n+1} = \alpha\, t_n + (1- \alpha)\, \Gamma_n.$$

where α, $0 \leq \alpha \leq 1$,

If $\alpha = 0$, then $\Gamma_{n+1} = \Gamma_n$ and recent history has no effect (current conditions are assumed to be transient).

If $\alpha = 1$, then $\Gamma_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant).

The SJF algorithm can be either preemptive or nonpreemptive.

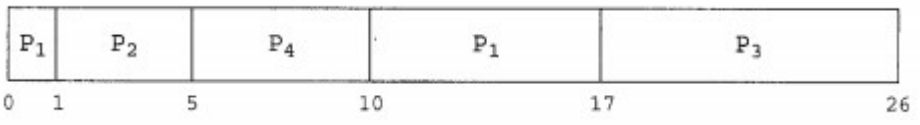
When a new process arrives at the ready queue with a shorter CPU burst than what is left of the currently executing process, a **preemptive SJF algorithm will preempt the currently executing process iff** the execution length of the newly added process is smaller than the remaining execution time of the running process, whereas a **nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.**

Preemptive SJF scheduling is sometimes called **Shortest-Remaining-Time-First** scheduling.

Consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled. The average waiting time for this example is [(10- 1) + (1 - 1) + (17- 2) +(5-3)]/ 4 = 26/4 = 6.5 milliseconds.

Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

Priority Scheduling

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

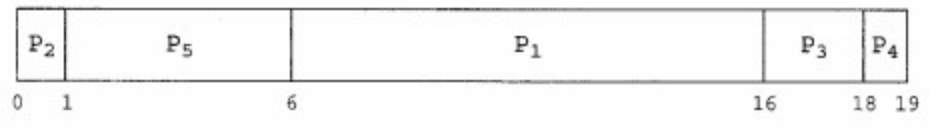
- → Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
- → Some systems use low numbers to represent low priority; others use low numbers for high priority.

Consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, · · ·, P5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

In this scenario low number represents high priority.

Using priority scheduling, we get the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally.

- Internally defined priorities use some measurable quantity to compute the priority of a process. Ex:*time limits, memory requirements, the number of open files, and the ratio of average I/0 burst to average CPU burst* , etc
- External priorities: are set by criteria outside the operating system, Ex: *importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political factors.*

Priority scheduling can be either preemptive or nonpreemptive.

- When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. *A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.*
- *A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.*

A major problem with priority scheduling algorithms is **indefinite blocking or starvation**.

A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm **can leave some low priority processes waiting indefinitely**. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

A solution to the problem of indefinite blockage of low-priority processes is **aging**.

Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.

Round-Robin Scheduling

The round-robin (RR) scheduling algorithm is designed especially for time sharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.

Methodology:

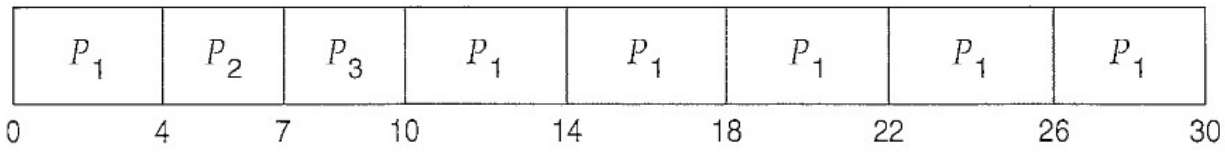
- A small unit of time, called a **time quantum** or **time slice**, is defined.
- The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. One of the 2 things will happen
    - The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.
    - The CPU burst of the currently running process is longer than 1 time quantum; the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail o£ the ready queue. The CPU scheduler will then select the next process in the ready queue.

→ The average waiting time under the RR policy is often long.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If time quantum = 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2. Process P2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is as follows:



P1 waits for 6 milliseconds (10- 4), P2 waits for 4 milliseconds, and P3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

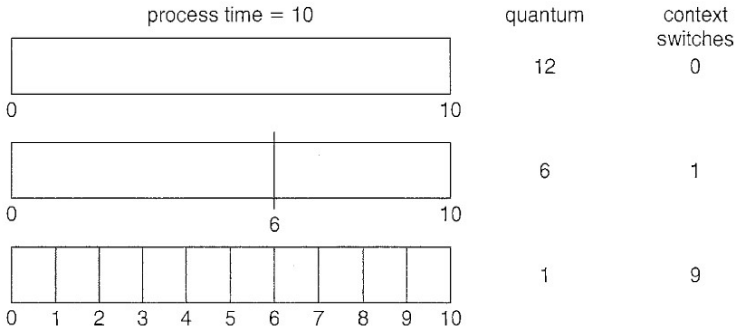
Features:

→ In the RR scheduling algorithm, *no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process).* If a process's CPU burst exceeds 1 time quantum, that process is preempted and is p11t back in the ready queue. The RR scheduling algorithm is thus preemptive.

→ **The performance of the RR algorithm depends heavily on the size of the time quantum.**

- If the **time quantum is extremely large**, the RR policy is the same as the FCFS policy.
- If the **time quantum is extremely small** (say, 1 millisecond), the RR approach is called **processor sharing** and (in theory) creates the appearance that each of n processes has its own processor running at 1/n the speed of the real processor. A lower value of time quantum *also results in too much context switching*

Ex: Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in. less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (In the figure below).



An adequate size of the time quantum ***should be sufficiently large w.r.t. the context switch time*** (at most 10 microseconds). Context switch time should be approximately 10% of the time quantum. Most modern systems have time quantum ranging from 10ms to 100ms.

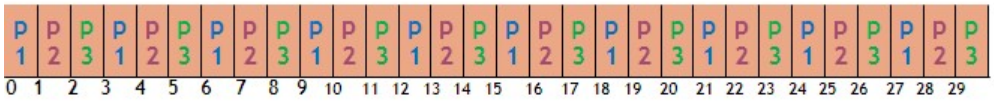
→ **Turnaround time also depends on the size of the time quantum.**

The average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.

Process	CPU Burst
P1	10
P2	10
P3	10

TIME QUANTUM TOO LOW

Quantum : 1 ms



Average Turnaround time = (28+29+30)/3 = 29 ms

If the time quantum is increased to 10ms,

Process	CPU Burst
P1	10
P2	10
P3	10

Quantum : 10 ms

Average turnaround time = (10+20+30)/3 = 20 ms

Multilevel Queue Scheduling

→ Another class of scheduling algorithms has been created ***for situations in which processes are easily classified into different groups.***

For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes

→ A multilevel queue scheduling algorithm partitions the ready queue into several separate queues (Figure 3).

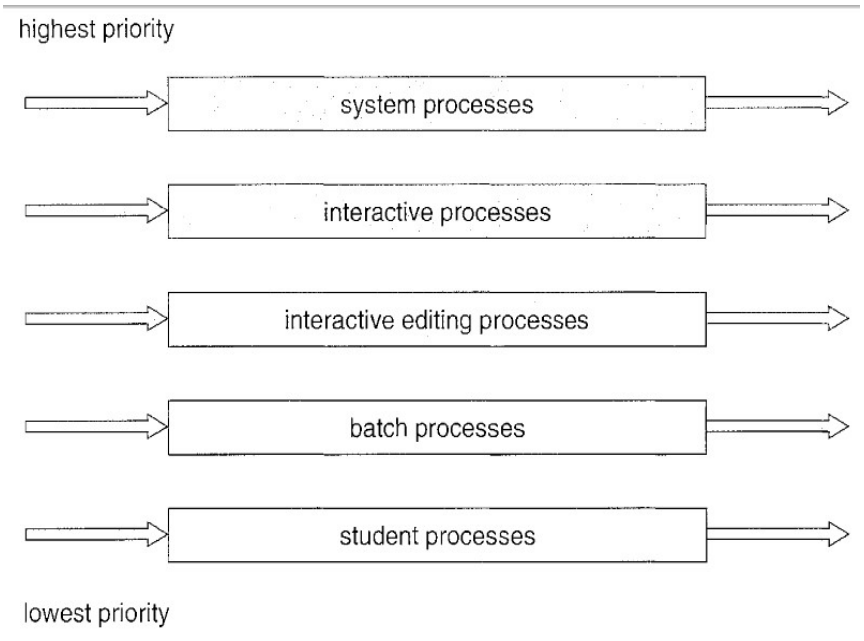


Figure 3: Multilevel queue scheduling

Features:

→ The ***processes are permanently assigned to one queue***, generally based on some property of the process, such as memory size, process priority, or process type.

→ ***Each queue has its own scheduling algorithm***. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

→ ***Each queue has absolute priority over lower-priority queues***. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

→ ***Another possibility is to time-slice among the queues***. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

Multilevel Feedback Queue Scheduling

→ Unlike multilevel queue, the multilevel feedback queue scheduling algorithm, in contrast, ***allows a process to move between queues.***

→ The idea is to ***separate processes according to the characteristics of their CPU bursts***.

→ If a ***process uses too much CPU time, it will be moved to a lower-priority queue***. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.

→ In addition, a ***process that waits too long in a lower-priority queue may be moved to a higher-priority queue***. This form of aging prevents starvation.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 4).

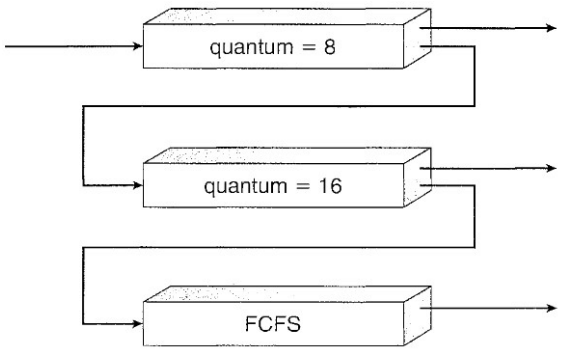


Figure 4: Multilevel feedback queues.

The scheduler first executes all processes in queue 0. *Only when queue 0 is empty will it execute processes in queue 1*. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/0 burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher priority queue
- The method used to determine when to demote a process to a lower priority queue
- The method used to determine which queue a process will enter when that process needs service

Consider the following set of processes with arrival time:

Process	Burst time	Arrival time
P_1	10	0
P_2	1	0
P_3	2	1
P_4	4	2
P_5	3	2

i) Draw Gantt charts using FCFS, SJF preemptive and non preemptive scheduling.
ii) Calculate the average waiting time for each of the scheduling algorithm. **(08 Marks)**

Following is the snapshot of a cpu

Process	CPU Burst	Arrival time
P_1	10	0
P_2	29	1
P_3	03	2
P_4	07	3

Draw Gantt charts and calculate the waiting and turnaround time using FCFS, SJF and RR with time quantum 10 scheduling algorithms. **(09 Marks)**

b. Consider the following set of processes. Assume the length of the CPU burst time is given in milli seconds.

Process	Arrival Time	Burst Time	Priority
P_1	0	10	3
P_2	0	1	1
P_3	3	2	3
P_4	5	1	4
P_5	10	5	2

Draw Gantt charts illustrating the execution of these processes using FCFS and pre-emptive priority scheduling algorithms. Assume highest priority = 1 and lowest priority = 4. Also, calculate average waiting time and average turn around time of both the algorithms.

(06 Marks)

a. **Consider the following set of processes, with CPU burst time (in ms) :**

Process	Arrival time (in ms)	Burst time
P0	0	6
P1	1	3
P2	2	1
P3	3	4

i) **Draw the Gantt chart illustrating the execution of above processes using Shortest – Remaining – Time First (SRTF) and non-preemptive Shortest-Job-First (SJF).**

ii) **Find the turn around time for each process for SRTF and SJF, hence show that, SRTF is faster than SJF.** **(10 Marks)**

b. **Consider the following data about processes**

Process	Arrival Time	Burst Time	Priority
P_1	0	7	3
P_2	3	2	2
P_3	4	3	1
P_4	4	1	1
P_5	5	3	3

i) Draw charts to illustrate execution using SRTF, preemptive priority and RR (TS = 1msec).

ii) Compute waiting time in each of the cases.

iii) Which of them provide minimal average waiting time and turnaround time?

iv) Find out the time at which there are maximum numbers of processes in the ready queue in the above scenario? **(10 Marks)**

c. Consider a system running 10 I/0 bound tasks and one CPU bound task. Assume I/0 bound tasks issue an I/0 once for every msecs of CPU computing and that each I/0 operation takes 10msecs to complete. Also assume that the context switching overhead is 0.1msec. and that all processes are long running tasks. Comment on the CPU utilization for a RR scheduler when TS = 1msec and TS = 10msec. **(04 Marks)**

c. Suppose the following jobs arrive for processing at the times indicated. Each job will run the listed amount of time.

Job	1	2	3
Arrival time	0.0	0.4	1.0
Burst time	8	4	1

i) Give a Gantt chart illustrating the execution of these jobs, using the non pre-emptive FCFS and SJF scheduling algorithms.

ii) What is turn around time and waiting time of each job for the above algorithms?

iii) Compute average turn around time if CPU is left idle for the first 1 unit and then SJF is used. (Job1 and Job2 will wait during this time) **(10 Marks)**

Thread scheduling

On systems that support user and kernel-level threads, it is the kernel-level threads-not processes-that are being scheduled by the operating system. User-level threads are managed by a thread library and to run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).

Contention Scope

- ***Contention scope*** refers to the scope in which threads compete for the use of physical CPUs.
- On systems implementing many-to-one and many-to-many threads, ***Process Contention Scope, PCS,*** occurs, because competition occurs between threads that are part of the same process.
- ***System Contention Scope, SCS,*** involves the system scheduler scheduling kernel threads to run on one or more CPUs. Systems implementing one-to-one threads, use only SCS.

- The Pthread library provides for specifying scope contention:
    - PTHREAD_SCOPE_PROCESS schedules threads using PCS, by scheduling user threads onto available LWPs using the many-to-many model.
    - PTHREAD_SCOPE_SYSTEM schedules threads using SCS, by binding user threads to particular LWPs, effectively implementing a one-to-one model.
- getscope and setscope methods provide for determining and setting the scope contention respectively:

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
   int i, scope;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
       printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
       printf("PTHREAD_SCOPE_SYSTEM");
      else
       fprintf(stderr, "Illegal scope value.\n");
    }

   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */

   pthread_exit(0);
}
```

Figure 5.9 Pthread scheduling API.

Multiple-processor scheduling

When multiple processors are available, then the scheduling gets more complicated, because *now there is more than one CPU which must be kept busy and in effective use at all times*.

Approaches to Multiple-Processor Scheduling

- One approach to multi-processor scheduling is ***asymmetric multiprocessing,*** in which one processor is the master, controlling all activities and running all kernel code, while the other runs only user code. This approach is relatively simple, as there is no need to share critical system data.
- Another approach is ***symmetric multiprocessing, SMP,*** where each processor schedules its own jobs, either from a common ready queue or from separate ready queues for each processor. Regardless, scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.

Processor Affinity

- Processors contain cache memory, which speeds up repeated accesses to the same memory locations.
- If a process were to switch from one processor to another each time it got a time slice, the data in the cache (for that process) would have to be invalidated and re-loaded from main memory, thereby cancels out the benefit of the cache.
- Therefore SMP systems attempt to keep processes on the same processor, via ***processor affinity. Soft affinity*** occurs when the system attempts to keep processes on the same processor but makes no guarantees. Linux and some other OSes support ***hard affinity,*** in which a process specifies that it is not to be moved between processors.

The main-memory architecture of a system can affect processor affinity issues. In the traditional systems, each processor has equal access to memory and I/O. As more processors are added, the processor bus becomes a limitation for system performance.

System designers use non-uniform memory access (NUMA) to increase processor speed without increasing the load on the processor bus. The architecture is non-uniform because each processor is close to some parts of memory and farther from other parts of memory. The processor quickly gains access to the memory it is close to, while it can take longer to gain access to memory that is farther away.

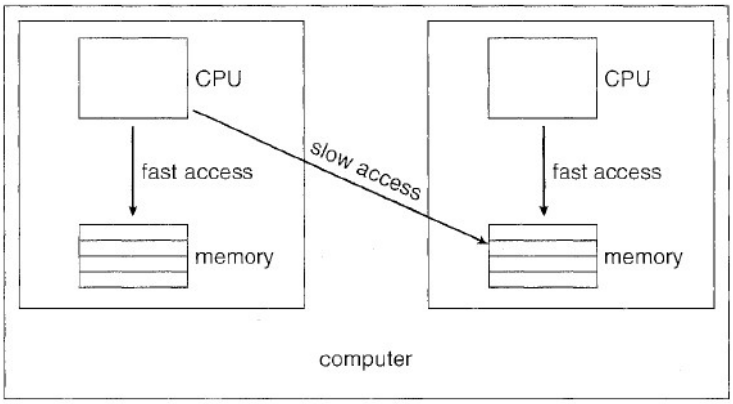


Figure 5: NUMA and CPU scheduling

Typically, this occurs in systems containing combined CPU and memory boards. The CPUs on a board can access the memory on that board with less delay than they can access memory on other boards in the system.

Therefore, if a process has an affinity for a particular CPU, then it should preferentially be assigned memory storage in "local" fast access areas.

Load Balancing

An important goal in a multiprocessor system is to balance the load between processors, so that one processor won't be sitting idle while another is overloaded.

Systems using a common ready queue are naturally self-balancing, and do not need any special handling. Most systems, however, maintain separate ready queues for each processor.

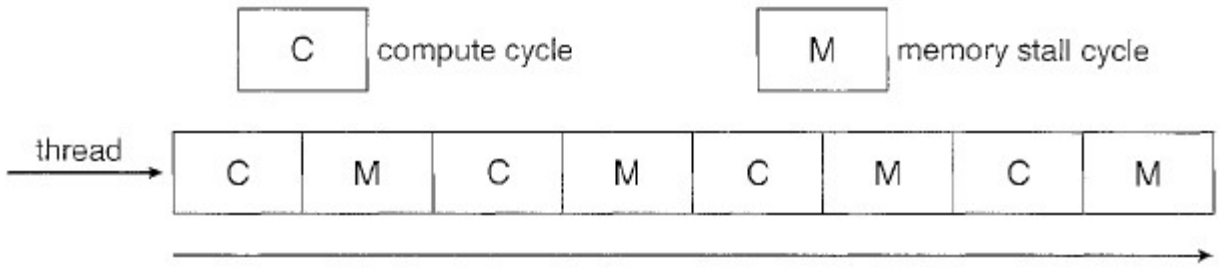
- Balancing can be achieved through either ***push migration*** or ***pull migration:***
    - ***Push migration*** involves a separate process that runs periodically, and moves processes from heavily loaded processors onto less loaded ones.
    - ***Pull migration*** involves idle processors taking processes from the ready queues of other processors.
    - Push and pull migration are not mutually exclusive.
- Note that moving processes from processor to processor to achieve load balancing works against the principle of processor affinity, and if not carefully managed, the savings gained by balancing the system can be lost in rebuilding caches. One option is to only allow migration when imbalance surpasses a given threshold.

Multicore Processors

A recent trend in computer hardware has been to place multiple processor cores on the same physical chip, resulting in a multicore processor. In multi-core CPUs, where 1 processor contains 2 or more cores, each processing core has its own arithmetic and logic unit, floating point unit, set of registers, pipeline, as well as some amount of cache. However multi-core CPUs also share some resources between the cores (e.g. L3-Cache, memory controller).

One of the problems with multicore processors is that when a processor accesses memory, it spends considerable amount of time waiting for data to become available. This situation is called ***memory stall*** (may occur due to reasons such as cache miss).

Therefore in multicore processor systems, Compute cycles can be blocked by the time needed to access memory, whenever the needed data is not already present in the cache. Figure below illustrates a memory stall.



In this scenario, the processor can spend up to 50 percent of its time waiting for data to become available from memory.

- By assigning multiple kernel threads to a single processor, memory stall can be avoided (or reduced) by running one thread on the processor while the other thread waits for memory.

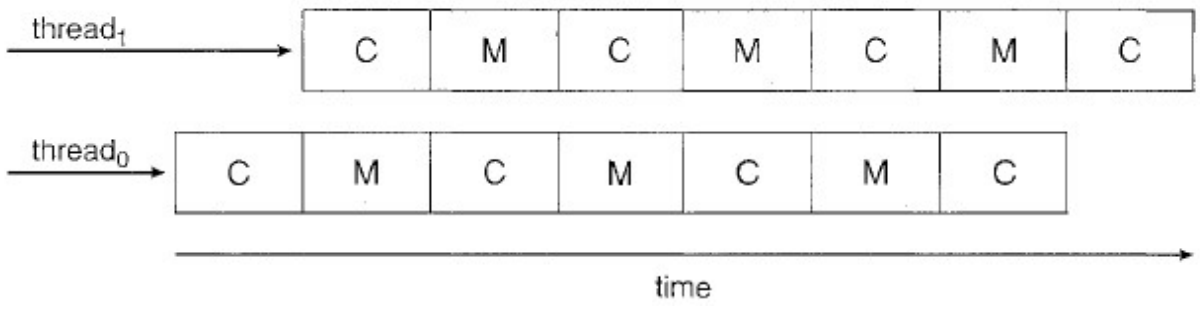


Figure: Multithreaded multicore system

In general, there are two ways to multithread a processor: **coarse-grained and fine-grained** multithreading.

Fine grained multithreading:

- switches from one thread to the other at each instruction – the execution of more threads is interleaved (often the switching is performed taking turns, skipping one thread if there is a stall)
- Advantage: It can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Drawback: It slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads

Coarse grained multithreading:

- switching from one thread to another occurs only when there are long stalls – e.g., for a miss on the second level cache.
- Two threads share many system resources (e.g., architectural registers) ð the switching from one thread to the next requires different clock cycles to save the context.
- Advantage:
    - Relieves need to have very fast thread-switching
    - Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall
- Disadvantage:
    - when there is one stall it is necessary to empty the pipeline before starting the new thread and later refill the pipeline with the instructions of the new thread.

Virtualization and Scheduling

- Virtualization adds another layer of complexity and scheduling.
- Typically there is one host operating system operating on "real" processor(s) and a number of guest operating systems operating on virtual processors.
- The Host OS creates some number of virtual processors and presents them to the guest OSes as if they were real processors.
- The guest OSes don't realize their processors are virtual, and make scheduling decisions on the assumption of real processors.
- As a result, interactive and especially real-time performance can be severely compromised on guest systems. The time-of-day clock will also frequently show delays.