PERATING SYSTEMS

II B .TECH, II-SEM CSE

UNIT II

Process Concept: Process ,Process in memory ,Process States, PCB, Process Scheduling ,Context Switching ,Process Schedulers, Process Creation ,Process Termination, Inter Process Communication. Example IPC Systems, Communication in Client-Server Systems.

Threads: overview, Multicore Programming, Multithreading Models, Thread Libraries, Implicit Threading, Threading Issues.

Process Synchronization: The critical-section problem, Peterson_s Solution, Synchronization Hardware, Mutex Locks, Semaphores, Classic problems of synchronization, Monitors, Synchronization examples, Alternative approaches.

CPU Scheduling: Scheduling-Criteria, Scheduling Algorithms, Thread Scheduling, Multiple-Processor Scheduling, Real-Time CPU Scheduling, Algorithm Evaluation.

Processes : Process Concept, Process Scheduling, Operations on Processes, Inter process Communication, Examples of IPC Systems, Communication in Client-Server Systems

Process -- a program in execution, which forms the basis of all computation

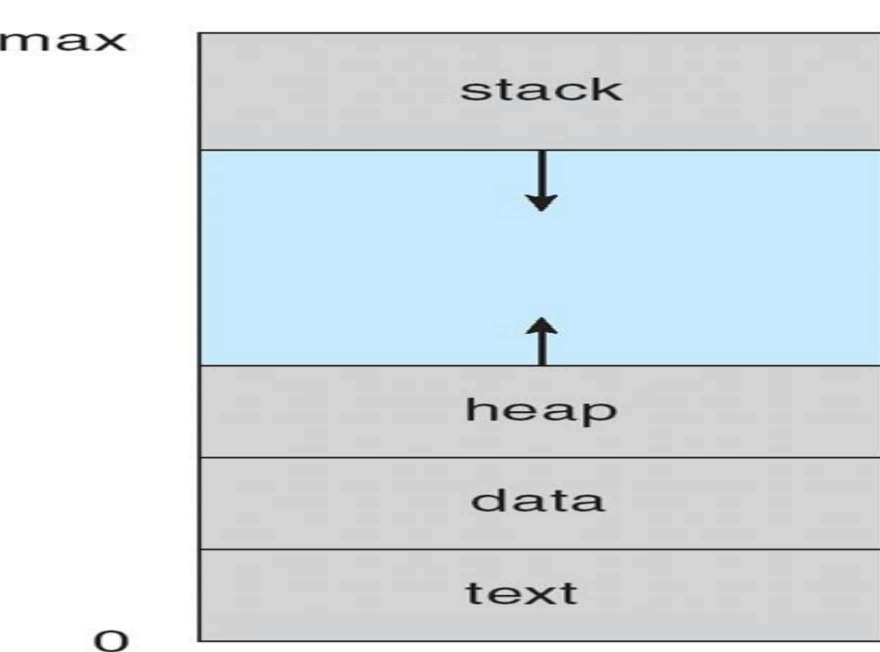
1.15 Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
  - Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section

The Process

- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
- Program is passive entity, process is active
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

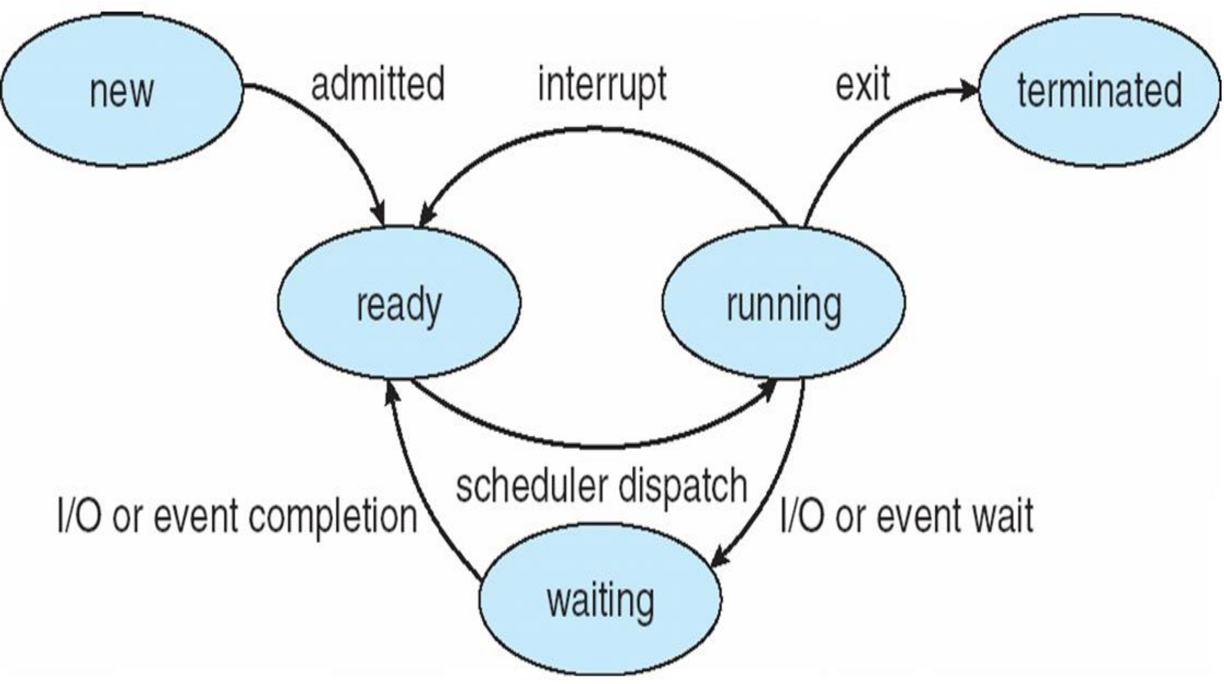
Process in Memory



Process State

- As a process executes, it changes *state*
    - **new:** The process is being created
    - **running:** Instructions are being executed
    - **waiting:** The process is waiting for some event to occur
    - **ready:** The process is waiting to be assigned to a processor
    - **terminated:** The process has finished execution

Diagram of Process State



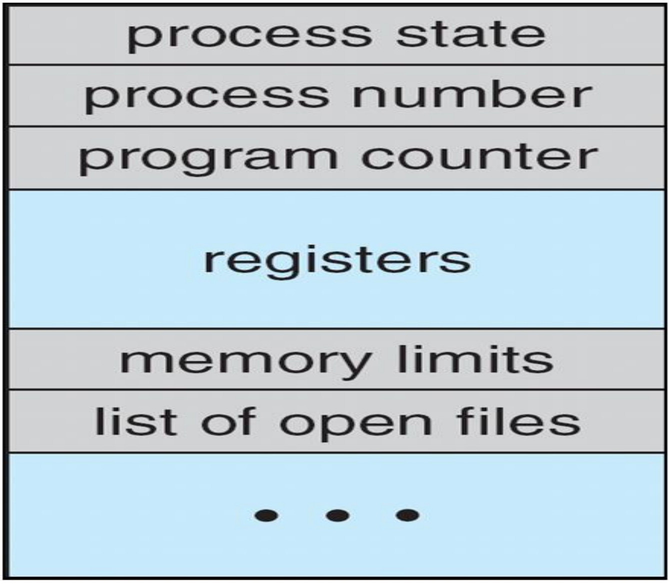
Process Control Block (PCB)

Information associated with each process

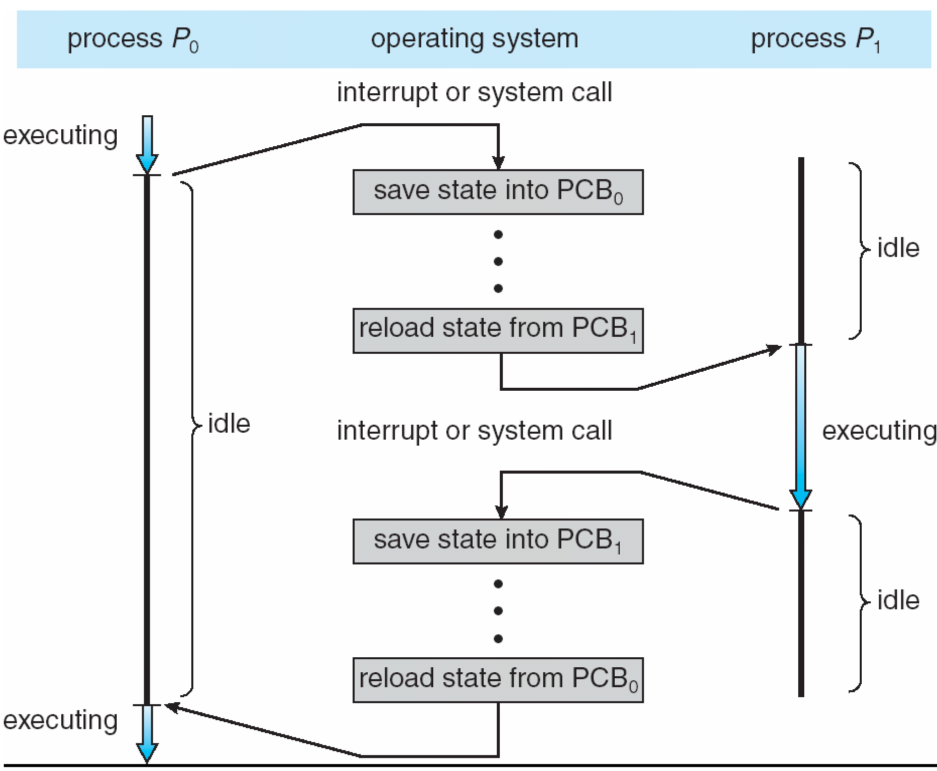
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information

- I/O status information

Process Control Block (PCB)



CPU Switch From Process to Process(CONTEXT-SWITCHING)

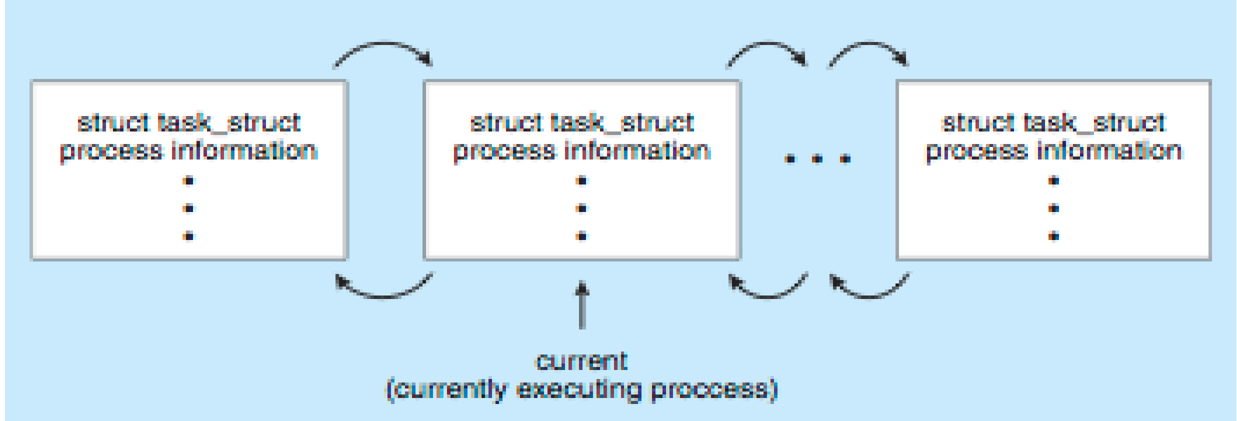


1.16 Process Scheduling

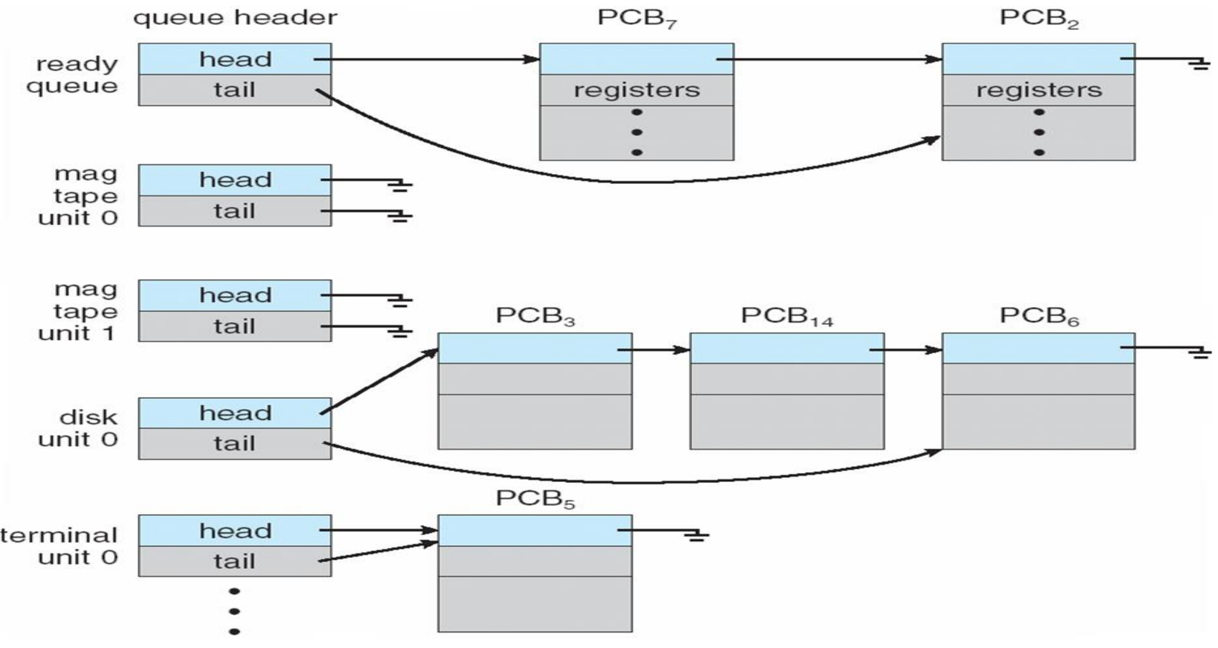
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on CPU
- Maintains scheduling queues of processes
    - **Job queue** – set of all processes in the system
    - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
    - **Device queues** – set of processes waiting for an I/O device
    - Processes migrate among the various queues

Process Representation in Linux

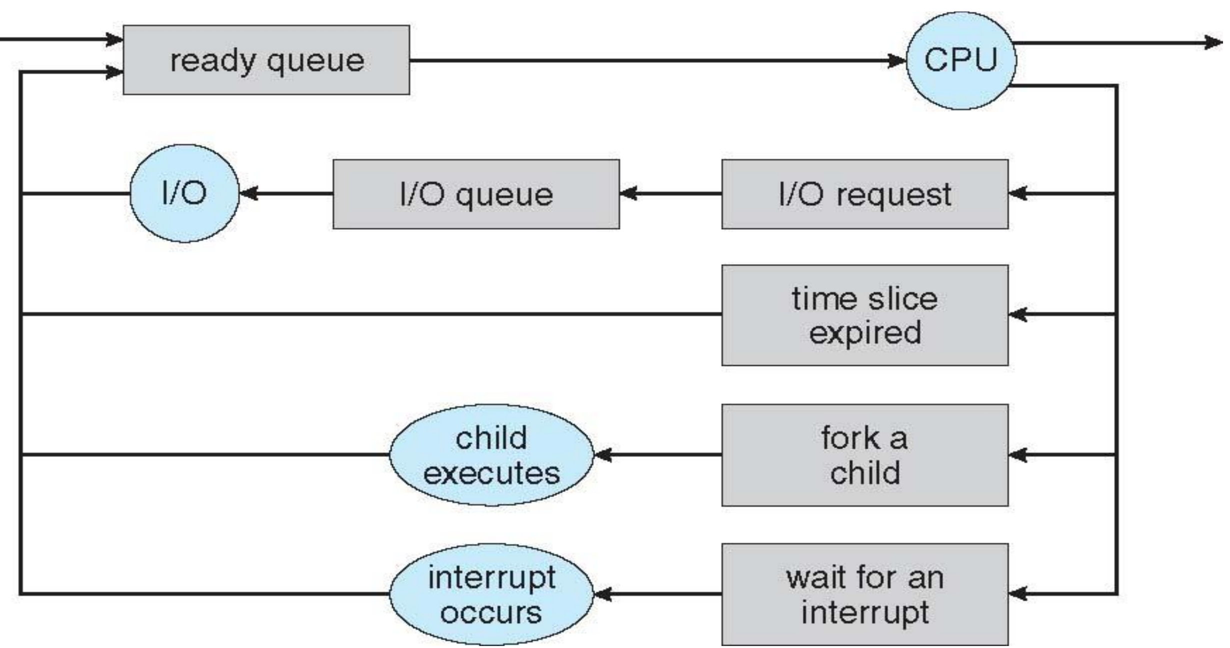
- Represented by the C structure
- task_struct
  pid t pid; /* process identifier */
  long state; /* state of the process */
  unsigned int time slice /* scheduling information */ struct task struct *parent; /* this process's parent */ struct list head children; /* this process's children */ struct files struct *files; /* list of open files */ struct mm struct *mm; /* address space of this pro */



Ready Queue And Various I/O Device Queues

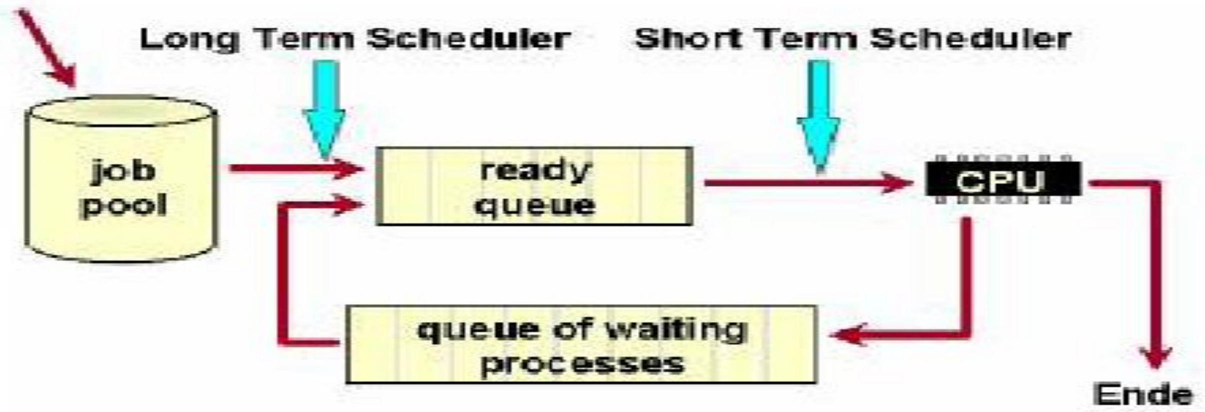


Representation of Process Scheduling



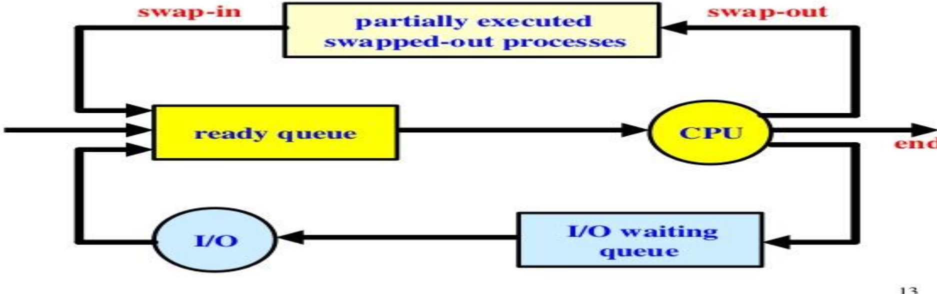
Schedulers

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU
    - Sometimes the only scheduler in a system



- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- 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

Addition of Medium Term Scheduling



Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB -> longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

1.17 Operations on Processes :

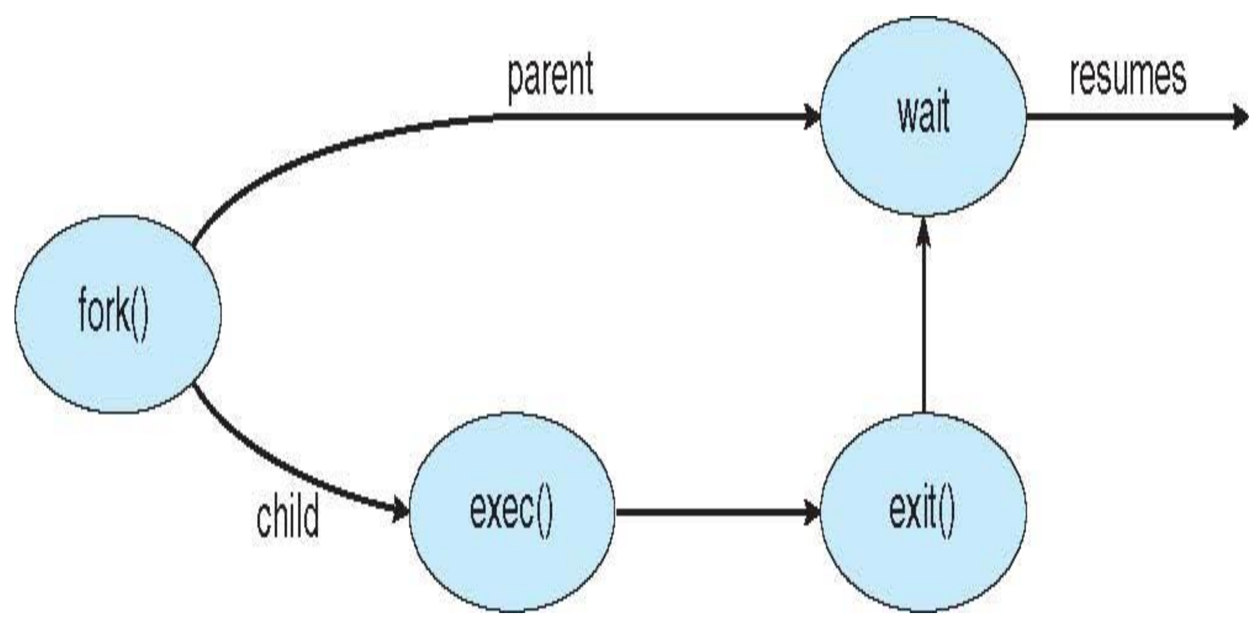
Process Creation

- Process Creation : 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
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources

  Execution

  - Parent and children execute concurrently
  - Parent waits until children terminate
- **Address space**
  - Child duplicate of parent
  - Child has a program loaded into it
- **UNIX examples**
  - fork system call creates new process
  - exec system call used after a fork to replace the process' memory space with a new program

Process Creation

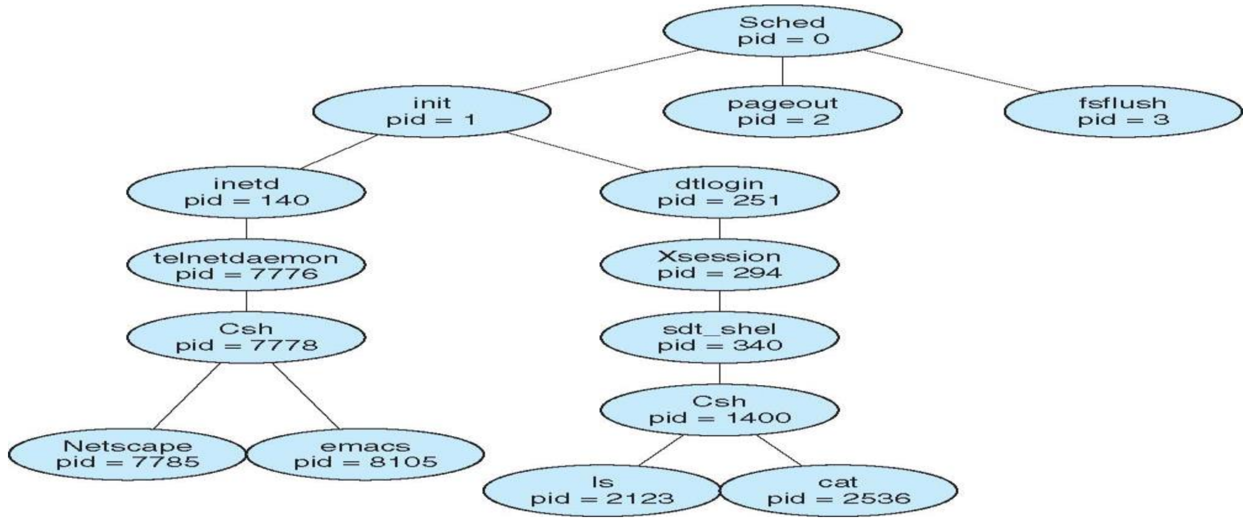


C Program Forking Separate Process

```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child */
```

```
                wait (NULL);

                printf ("Child Complete");

            }

            return 0;

}
```

A Tree of Processes on Solaris



Process Termination

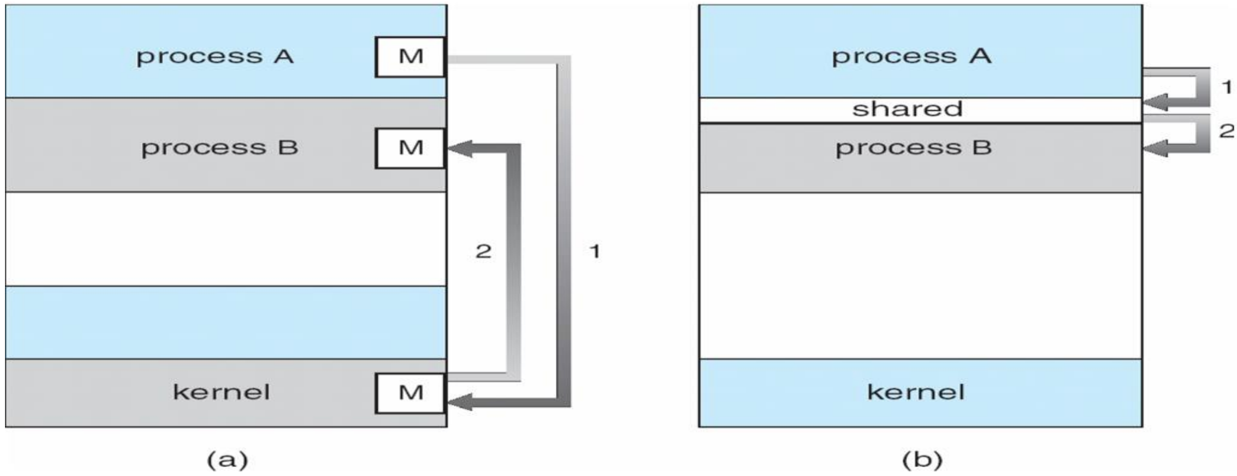
- Process executes last statement and asks the operating system to delete it (exit)
    - Output data from child to parent (via wait)
    - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort)
    - Child has exceeded allocated resources
    - Task assigned to child is no longer required
    - If parent is exiting
        - Some operating system do not allow child to continue if its parent terminates
            - All children terminated - cascading termination

1.18 Inter process Communication

- Processes within a system may be independent or cooperating

- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
    - Information sharing
    - Computation speedup
    - Modularity
    - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
    - Shared memory
    - Message passing

Communications Models



Cooperating Processes

- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
    - Information sharing
    - Computation speed-up
    - Modularity
    - Convenience

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
    - *unbounded-buffer* places no practical limit on the size of the buffer
    - *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer

Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10

typedef struct {

.

} item;

item buffer[BUFFER_SIZE];

int in = 0;

int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
while(true)

{
   /* Produce an item */

        while (((in = (in + 1) % BUFFER SIZE count)  == out)

                              ;   /* do nothing -- no free buffers */

                             buffer[in] = item;

                             in = (in + 1) % BUFFER SIZE;

    }
```

Bounded Buffer – Consumer

```
while (true)

{

          while (in == out)
```

```
        ; // do nothing -- nothing to consume

                // remove an item from the buffer

                item = buffer[out];

                out = (out + 1) % BUFFER SIZE;

            return item;

}
```

InterprocessCommunication –
Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
    - send(*message*) – message size fixed or variable
    - receive(*message*)
- If *P* and *Q* wish to communicate, they need to:
    - establish a *communication link* between them
    - exchange messages via send/receive
- Implementation of communication link
    - physical (e.g., shared memory, hardware bus)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:

- send (*P, message*) – send a message to process P
- receive(Q, *message*) – receive a message from process Q
- Properties of communication link
- Links are established automatically
- A link is associated with exactly one pair of communicating processes
- Between each pair there exists exactly one link
- The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
    - Each mailbox has a unique id
    - Processes can communicate only if they share a mailbox
    - Properties of communication link
    - Link established only if processes share a common mailbox
    - A link may be associated with many processes
    - Each pair of processes may share several communication links
    - Link may be unidirectional or bi-directional
- **Operations**
    - create a new mailbox
    - send and receive messages through mailbox
    - destroy a mailbox
    - ➢ **Primitives are defined as:**

      send(*A, message*) – send a message to mailbox A

      receive(*A, message*) – receive a message from mailbox A
- **Mailbox sharing**
    - P_1, P_2, and P_3 share mailbox A
    - P_1, sends; P_2 and P_3 receive
    - Who gets the message?

- **Solutions**
    - Allow a link to be associated with at most two processes
    - Allow only one process at a time to execute a receive operation
    - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered synchronous
    - **Blocking send** has the sender block until the message is received
    - **Blocking receive** has the receiver block until a message is available
    - Non-blocking is considered **asynchronous**
    - **Non-blocking send** has the sender send the message and continue
    - **Non-blocking receive** has the receiver receive a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways

    1. Zero capacity – 0 messages
    Sender must wait for receiver (rendezvous)

    2.Bounded capacity – finite length of *n* messages
    Sender must wait if link full

    3.Unbounded capacity – infinite length
    Sender never waits

1.19 Examples of IPC Systems - POSIX

- **POSIX Shared Memory**
    - **Process first creates shared memory segment**

segment id = shmget(IPC PRIVATE, size, S IRUSR | S IWUSR);

- Process wanting access to that shared memory must attach to it

shared memory = (char *) shmat(id, NULL, 0);

- Now the process could write to the shared memory

sprintf(shared memory, "Writing to shared memory");

- When done a process can detach the shared memory from its address space

shmdt(shared memory);

Examples of IPC Systems – Mach

Mach communication is message based

- Even system calls are messages
- Each task gets two mailboxes at creation- Kernel and Notify
- Only three system calls needed for message transfer

msg_send(), msg_receive(), msg_rpc()

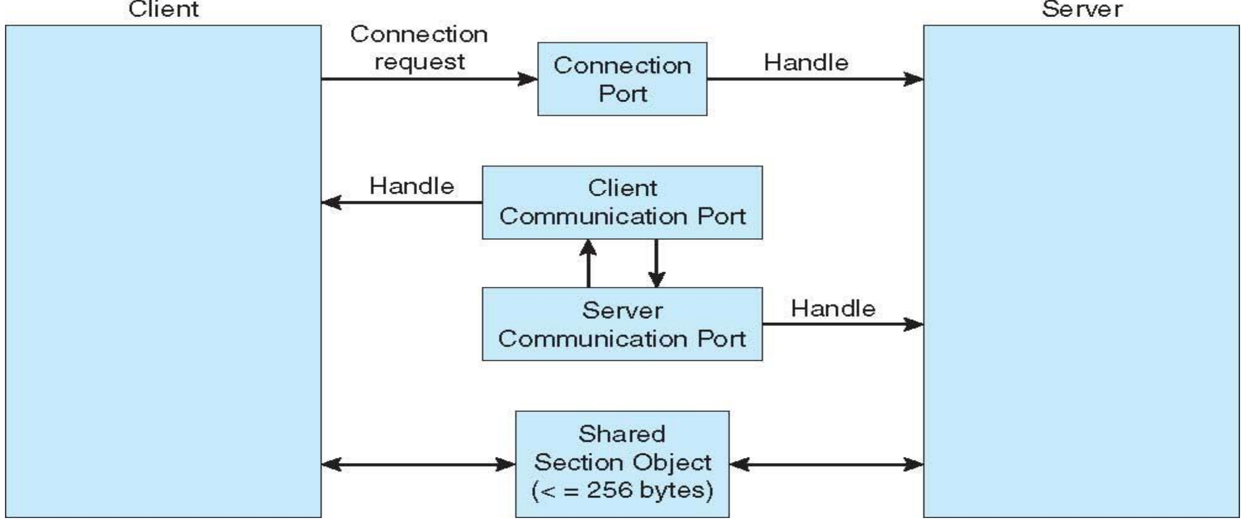
- Mailboxes needed for commuication, created via

port_allocate()

Examples of IPC Systems – Windows XP

- Message-passing centric via local procedure call (LPC) facility
    - Only works between processes on the same system
    - Uses ports (like mailboxes) to establish and maintain communication channels
    - Communication works as follows:
        - The client opens a handle to the subsystem's connection port object.
        - The client sends a connection request.
        - The server creates two private communication ports and returns the handle to one of them to the client.
        - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Local Procedure Calls in Windows XP



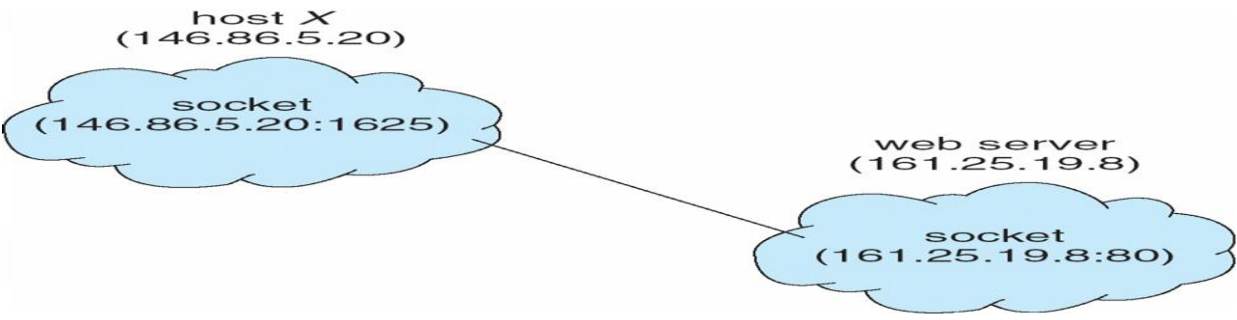
1.20 Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)

Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets

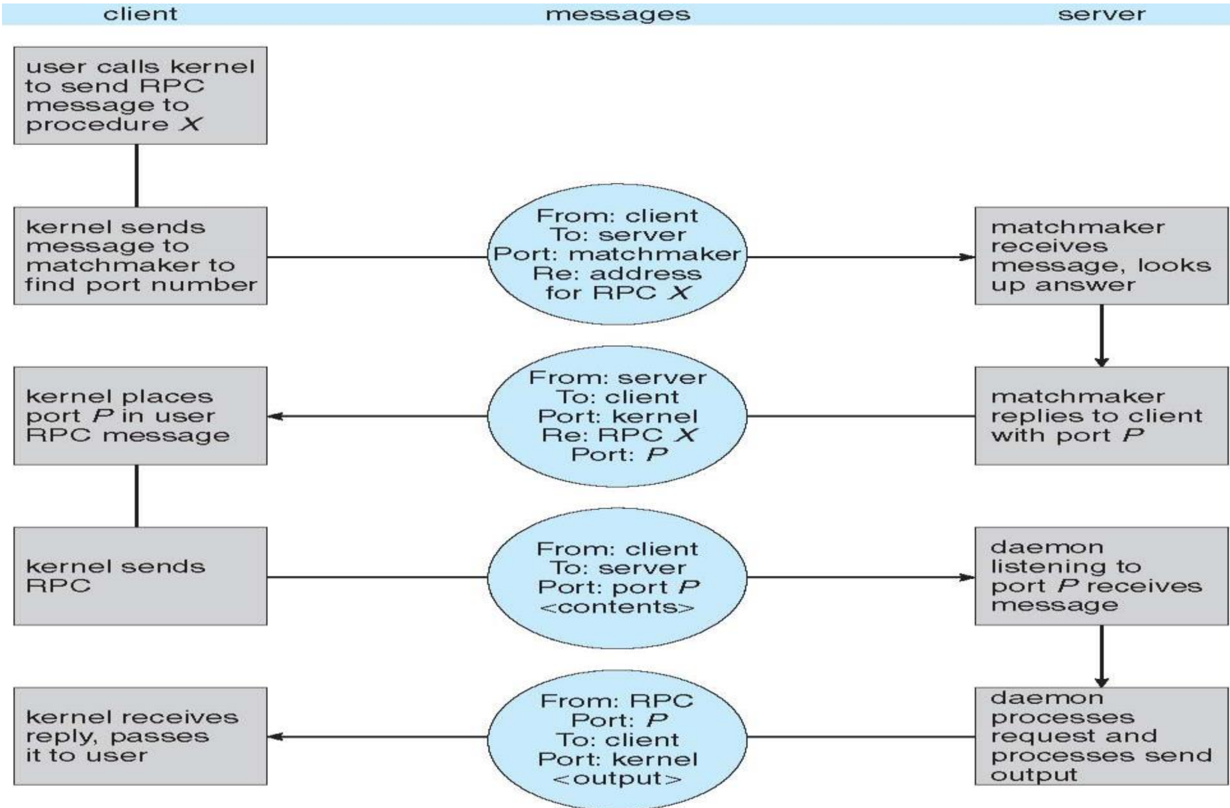
Socket Communication



Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- Stubs – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

Execution of RPC



Pipes

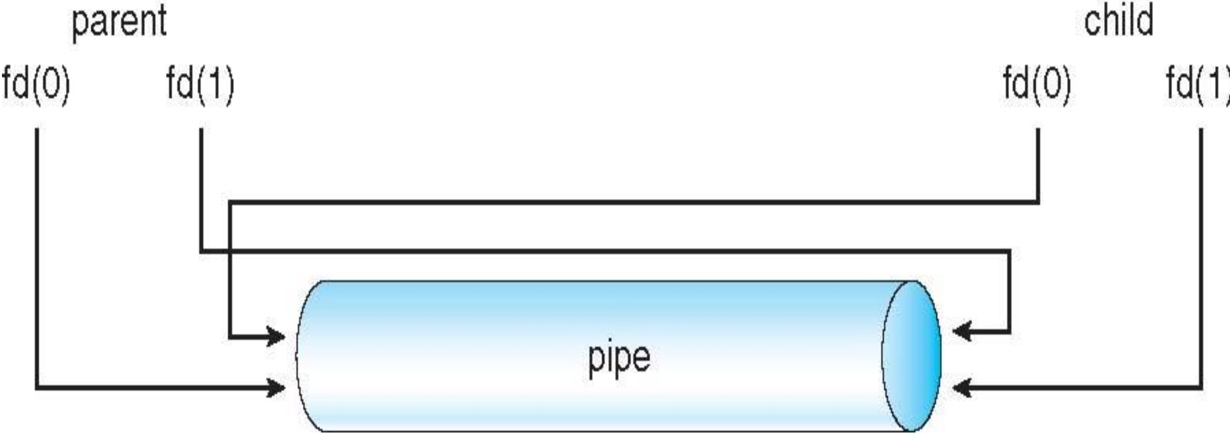
- Acts as a conduit allowing two processes to communicate

- **Issues**
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e. parent-child) between the communicating processes?
  - Can the pipes be used over a network?

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the *write-end* of the pipe)
- Consumer reads from the other end (the *read-end* of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes

Ordinary Pipes



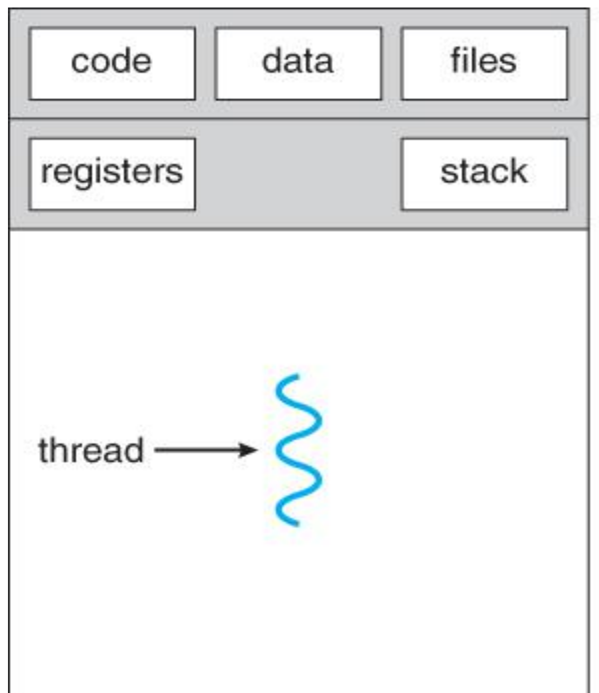
Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

CHAPTER-4 : Threads

4.1 Overview

- A ***thread*** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)
- Traditional (heavyweight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- As shown in Figure 4.1, multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.



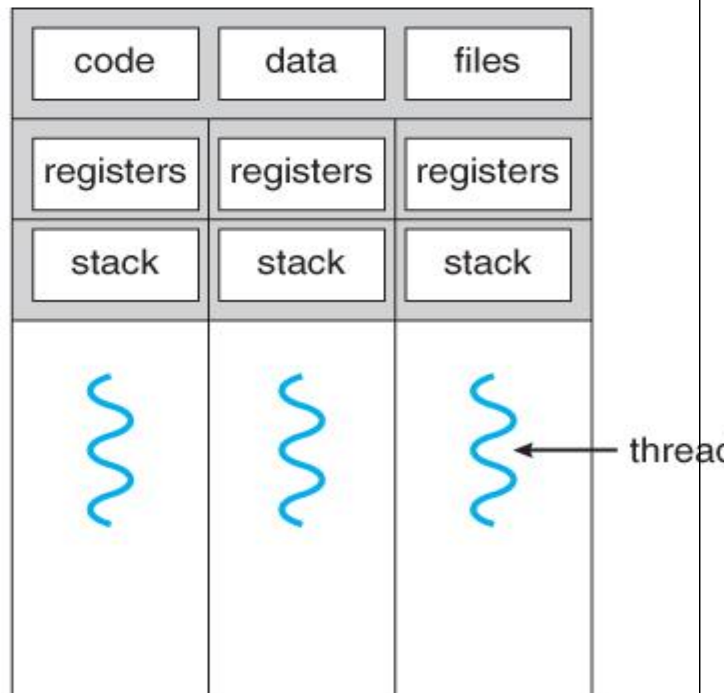


Figure 4.1 - Single-threaded and multithreaded processes

4.1.1 Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while

yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.

- Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request. (The latter is how this sort of thing was done before the concept of threads was developed. A daemon would listen at a port, fork off a child for every incoming request to be processed, and then go back to listening to the port.)

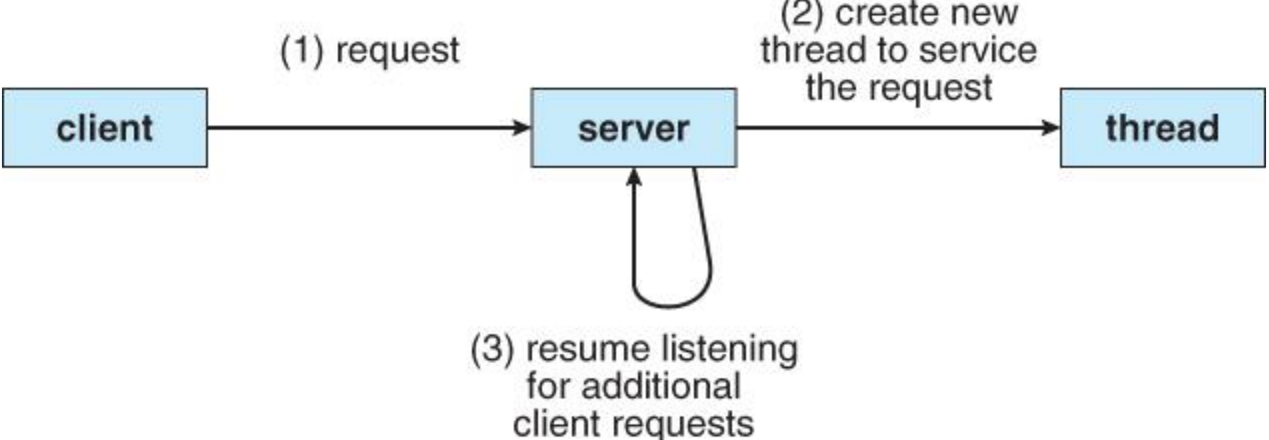


Figure 4.2 - Multithreaded server architecture

4.1.2 Benefits

- There are four major categories of benefits to multi-threading:
    1. **Responsiveness** - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
    2. **Resource sharing** - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
    3. **Economy** - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.
    4. **Scalability**, i.e. Utilization of multiprocessor architectures - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors. (Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold.)

4.2 Multicore Programming

- A recent trend in computer architecture is to produce chips with multiple ***cores***, or CPUs on a single chip.
- A multi-threaded application running on a traditional single-core chip would have to interleave the threads, as shown in Figure 4.3. On a multi-core chip, however, the threads could be spread across the available cores, allowing true parallel processing, as shown in Figure 4.4.

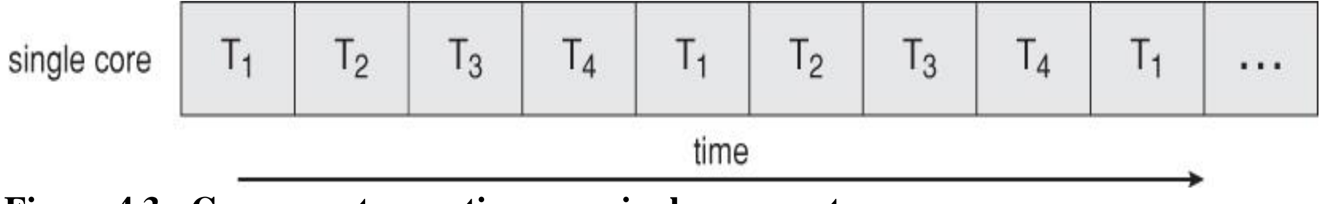


Figure 4.3 - Concurrent execution on a single-core system.

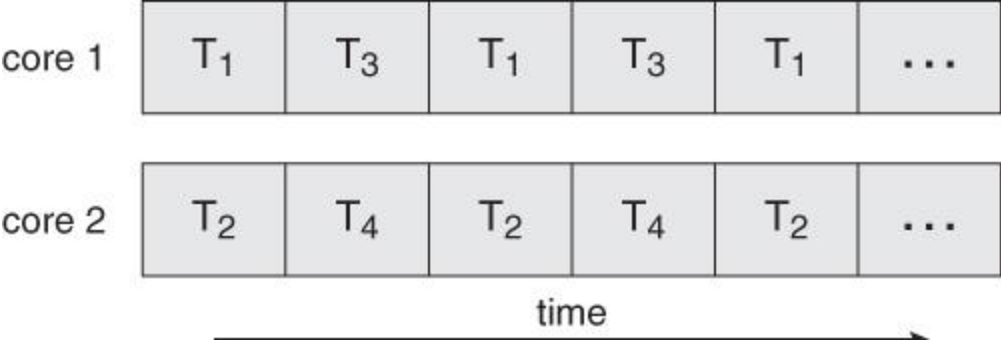


Figure 4.4 - Parallel execution on a multicore system

- For operating systems, multi-core chips require new scheduling algorithms to make better use of the multiple cores available.
- As multi-threading becomes more pervasive and more important (thousands instead of tens of threads), CPUs have been developed to support more simultaneous threads per core in hardware.

4.2.1 Programming Challenges

- For application programmers, there are five areas where multi-core chips present new challenges:
    1. **Identifying tasks** - Examining applications to find activities that can be performed concurrently.
    2. **Balance** - Finding tasks to run concurrently that provide equal value. I.e. don't waste a thread on trivial tasks.
    3. **Data splitting** - To prevent the threads from interfering with one another.

4. **Data dependency** - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.
5. **Testing and debugging** - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

4.2.2 Types of Parallelism

In theory there are two different ways to parallelize the workload:

1. **Data parallelism** divides the data up amongst multiple cores (threads), and performs the same task on each subset of the data. For example dividing a large image up into pieces and performing the same digital image processing on each piece on different cores.
2. **Task parallelism** divides the different tasks to be performed among the different cores and performs them simultaneously.

In practice no program is ever divided up solely by one or the other of these, but instead by some sort of hybrid combination.

4.3 Multithreading Models

- There are two types of threads to be managed in a modern system: **User threads and kernel threads.**
- User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.
- Kernel threads are supported within the kernel of the OS itself. All modern OS support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.
- In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.

4.3.1 Many-To-One Model

- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is very efficient.
- However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue.

- Because a single kernel thread can operate only on a single CPU, the many-to-one model does not allow individual processes to be split across multiple CPUs.
- Green threads for Solaris and GNU Portable Threads implement the many-to-one model in the past, but few systems continue to do so today.

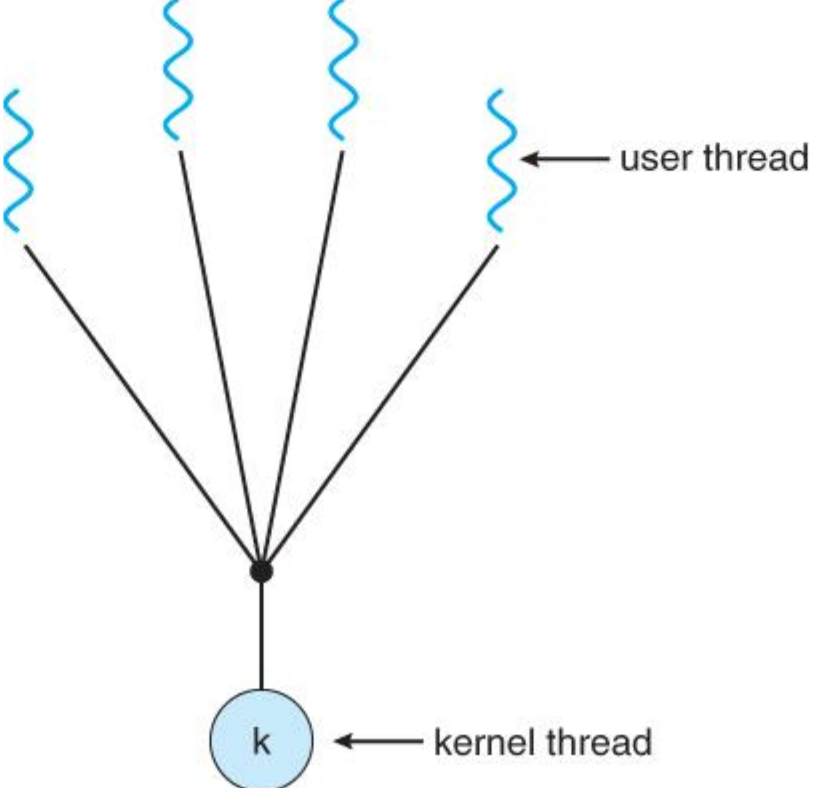


Figure 4.5 - Many-to-one model

4.3.2 One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each user thread.
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.

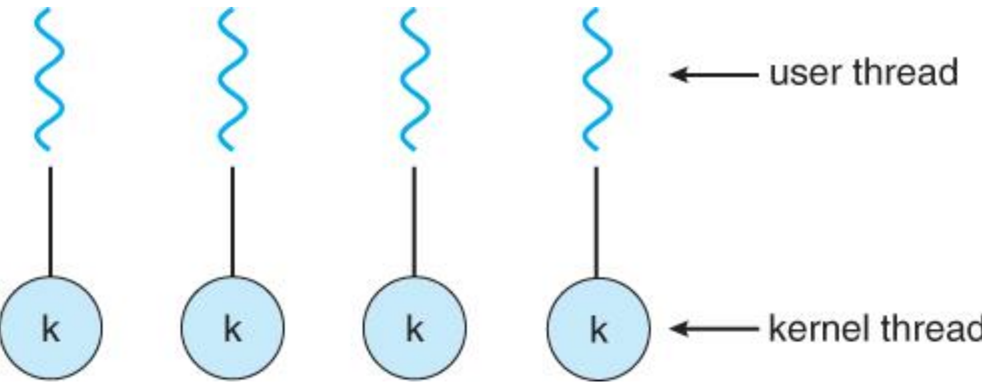


Figure 4.6 - One-to-one model

4.3.3 Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.

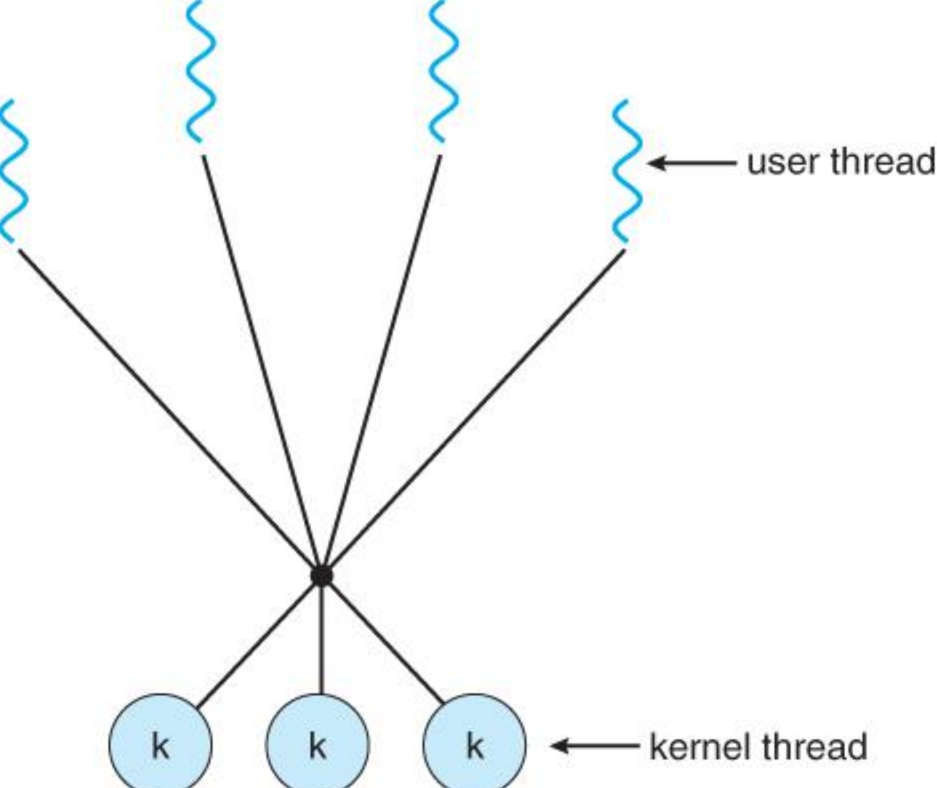


Figure 4.7 - Many-to-many model

- One popular variation of the many-to-many model is the two-tier model, which allows either many-to-many or one-to-one operation.
- IRIX, HP-UX, and Tru64 UNIX use the two-tier model, as did Solaris prior to Solaris 9.

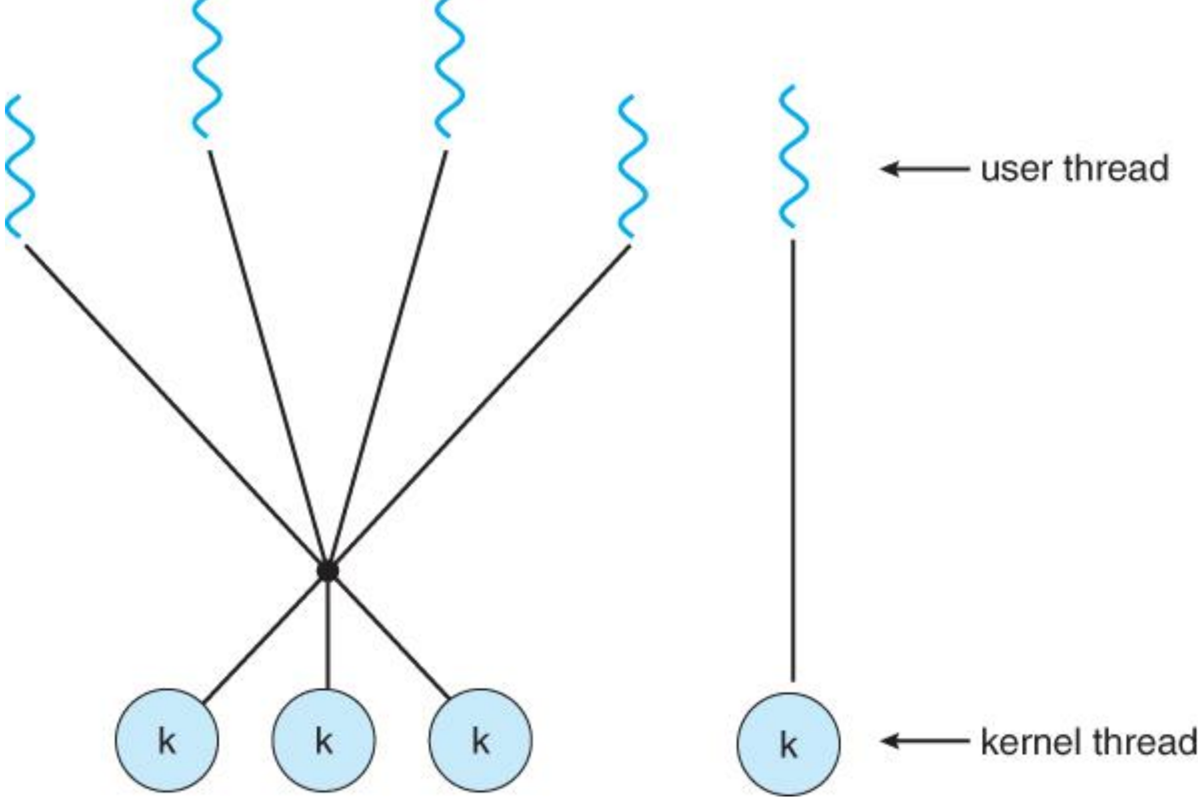


Figure 4.8 - Two-level model

4.4 Thread Libraries

- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space. The former involves API functions implemented solely within user space, with no kernel support. The latter involves system calls, and requires a kernel with thread library support.
- There are three main thread libraries in use today:
    1. **POSIX Pthreads** - may be provided as either a user or kernel library, as an extension to the POSIX standard.
    2. **Win32 threads** - provided as a kernel-level library on Windows systems.
    3. **Java threads** - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

- The following sections will demonstrate the use of threads in all three systems for calculating the sum of integers from 0 to N in a separate thread, and storing the result in a variable "sum".

4.4.1 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:

```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 thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }

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

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

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

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

  pthread_exit(0);
}
```

Figure 4.6 Multithreaded C program using the Pthreads API.

Figure 4.9

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Figure 4.10 Pthread code for joining ten threads.

4.4.2 Windows Threads

- Similar to Pthreads. Examine the code example to see the differences, which are mostly syntactic & nomenclature:

```
#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;
   /* perform some basic error checking */
   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }

   // 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 the thread handle
      CloseHandle(ThreadHandle);

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

Figure 4.7 Multithreaded C program using the Win32 API.

Figure 4.11

4.4.3 Java Threads

- ALL Java programs use Threads - even "common" single-threaded ones.
- 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.
- Note that the JVM runs on top of a native OS, and that the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision is JVM implementation dependant, and may be one-to-one, many-to-many, or many to one. (On a UNIX system the JVM normally uses PThreads and on a Windows system it normally uses windows threads.)

```java
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 4.8 Java program for the summation of a non-negative integer.

Figure 4.12

4.5 Implicit Threading

Shifts the burden of addressing the programming challenges outlined in section 4.2.1 above from the application programmer to the compiler and run-time libraries.

4.5.1 Thread Pools

- Creating new threads every time one is needed and then deleting it when it is done can be inefficient, and can also lead to a very large (unlimited) number of threads being created.
- An alternative solution is to create a number of threads when the process first starts, and put those threads into a ***thread pool***.
  - Threads are allocated from the pool as needed, and returned to the pool when no longer needed.
  - When no threads are available in the pool, the process may have to wait until one becomes available.
- The (maximum) number of threads available in a thread pool may be determined by adjustable parameters, possibly dynamically in response to changing system loads.
- Win32 provides thread pools through the "Pool Function" function. Java also provides support for thread pools through the java.util.concurrent package, and Apple supports thread pools under the Grand Central Dispatch architecture.

4.5.2 OpenMP

- OpenMP is a set of compiler directives available for C, C++, or FORTRAN programs that instruct the compiler to automatically generate parallel code where appropriate.
- For example, the directive:

```
#pragma omp parallel
 {
        /* some parallel code here */
 }
```

Would cause the compiler to create as many threads as the machine has cores available (e.g. 4 on a quad-core machine), and to run the parallel block of code, (known as a **parallel region**) on each of the threads.

- Another sample directive is "**#pragma omp parallel for**", which causes the for loop immediately following it to be parallelized, dividing the iterations up amongst the available cores.

4.5.3 Grand Central Dispatch, GCD

- GCD is an extension to C and C++ available on Apple's OSX and iOS operating systems to support parallelism.
- Similar to OpenMP, users of GCD define **blocks** of code to be executed either serially or in parallel by placing a carat just before an opening curly brace, i.e.

  {printf("I am a block.\n"); }

- GCD schedules blocks by placing them on one of several **dispatch queues**.
  - Blocks placed on a serial queue are removed one by one. The next block cannot be removed for scheduling until the previous block has completed.
  - There are three concurrent queues, corresponding roughly to low, medium, or high priority. Blocks are also removed from these queues one by one, but several may be removed and dispatched without waiting for others to finish first, depending on the availability of threads.
- Internally GCD manages a pool of POSIX threads which may fluctuate in size depending on load conditions.

4.5.4 Other Approaches

There are several other approaches available, including Microsoft's Threading Building Blocks (TBB) and other products, and Java's util.concurrent package.

4.6 Threading Issues

4.6.1 The fork ( ) and exec ( ) System Calls

- Q: If one thread forks, is the entire process copied, or is the new process single-threaded?
- A: System dependant.
- A: If the new process execs right away, there is no need to copy all the other threads. If it doesn't, then the entire process should be copied.
- A: Many versions of UNIX provide multiple versions of the fork call for this purpose.

4.6.2 Signal Handling

- Q: When a multi-threaded process receives a signal, to what thread should that signal be delivered?
- A: There are four major options:
  1. Deliver the signal to the thread to which the signal applies.
  2. Deliver the signal to every thread in the process.

3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals in a process.

- The best choice may depend on which specific signal is involved.
- UNIX allows individual threads to indicate which signals they are accepting and which they are ignoring. However the signal can only be delivered to one thread, which is generally the first thread that is accepting that particular signal.
- UNIX provides two separate system calls, **kill (pid, signal)** and **pthread_kill (tid, signal)**, for delivering signals to processes or specific threads respectively.
- Windows does not support signals, but they can be emulated using Asynchronous Procedure Calls (APCs). APCs are delivered to specific threads, not processes.

4.6.3 Thread Cancellation

- Threads that are no longer needed may be cancelled by another thread in one of two ways:
    1. **Asynchronous Cancellation** cancels the thread immediately.
    2. **Deferred Cancellation** sets a flag indicating the thread should cancel itself when it is convenient. It is then up to the cancelled thread to check this flag periodically and exit nicely when it sees the flag set.
- (Shared) resource allocation and inter-thread data transfers can be problematic with asynchronous cancellation.

4.6.4 Thread-Local Storage (was 4.4.5 Thread-Specific Data)

- Most data is shared among threads, and this is one of the major benefits of using threads in the first place.
- However sometimes threads need thread-specific data also.
- Most major thread libraries (PThreads, Win32, Java) provide support for thread-specific data, known as **thread-local storage** or **TLS.** Note that this is more like static data than local variables, because it does not cease to exist when the function ends.

4.6.5 Scheduler Activations

- Many implementations of threads provide a virtual processor as an interface between the user thread and the kernel thread, particularly for the many-to-many or two-tier models.
- This virtual processor is known as a "Lightweight Process", LWP.
    - There is a one-to-one correspondence between LWPs and kernel threads.
    - The number of kernel threads available, (and hence the number of LWPs) may change dynamically.

    - The application (user level thread library) maps user threads onto available LWPs.
    - Kernel threads are scheduled onto the real processor(s) by the OS.
    - The kernel communicates to the user-level thread library when certain events occur (such as a thread about to block) via an ***upcall***, which is handled in the thread library by an ***upcall handler***. The upcall also provides a new LWP for the upcall handler to run on, which it can then use to reschedule the user thread that is about to become blocked. The OS will also issue up calls when a thread becomes unblocked, so the thread library can make appropriate adjustments.
- If the kernel thread blocks, then the LWP blocks, which blocks the user thread.
- Ideally there should be at least as many LWPs available as there could be concurrently blocked kernel threads. Otherwise if all LWPs are blocked, then user threads will have to wait for one to become available.

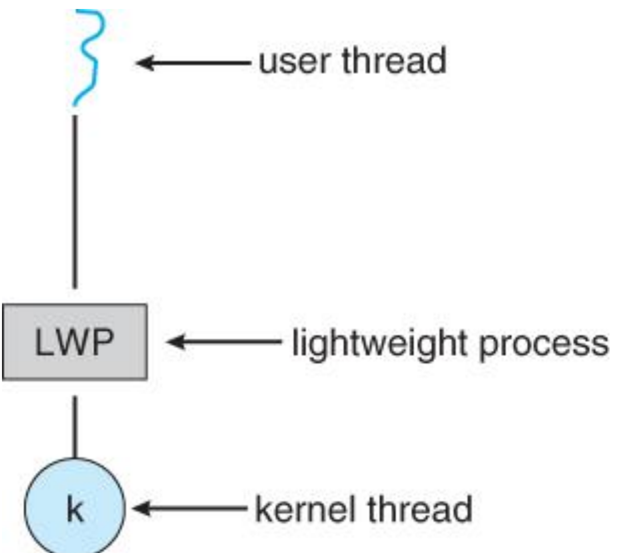


Figure 4.13 - Lightweight process (LWP)

4.7 Operating-System Examples

4.7.1 Windows XP Threads

- The Win32 API thread library supports the one-to-one thread model
- Win32 also provides the ***fiber*** library, which supports the many-to-many model.
- Win32 thread components include:
    - Thread ID
    - Registers
    - A user stack used in user mode, and a kernel stack used in kernel mode.
    - A private storage area used by various run-time libraries and dynamic link libraries (DLLs).

- The key data structures for Windows threads are the ETHREAD (executive thread block), KTHREAD (kernel thread block), and the TEB (thread environment block). The ETHREAD and KTHREAD structures exist entirely within kernel space, and hence are only accessible by the kernel, whereas the TEB lies within user space, as illustrated in Figure 4.10:

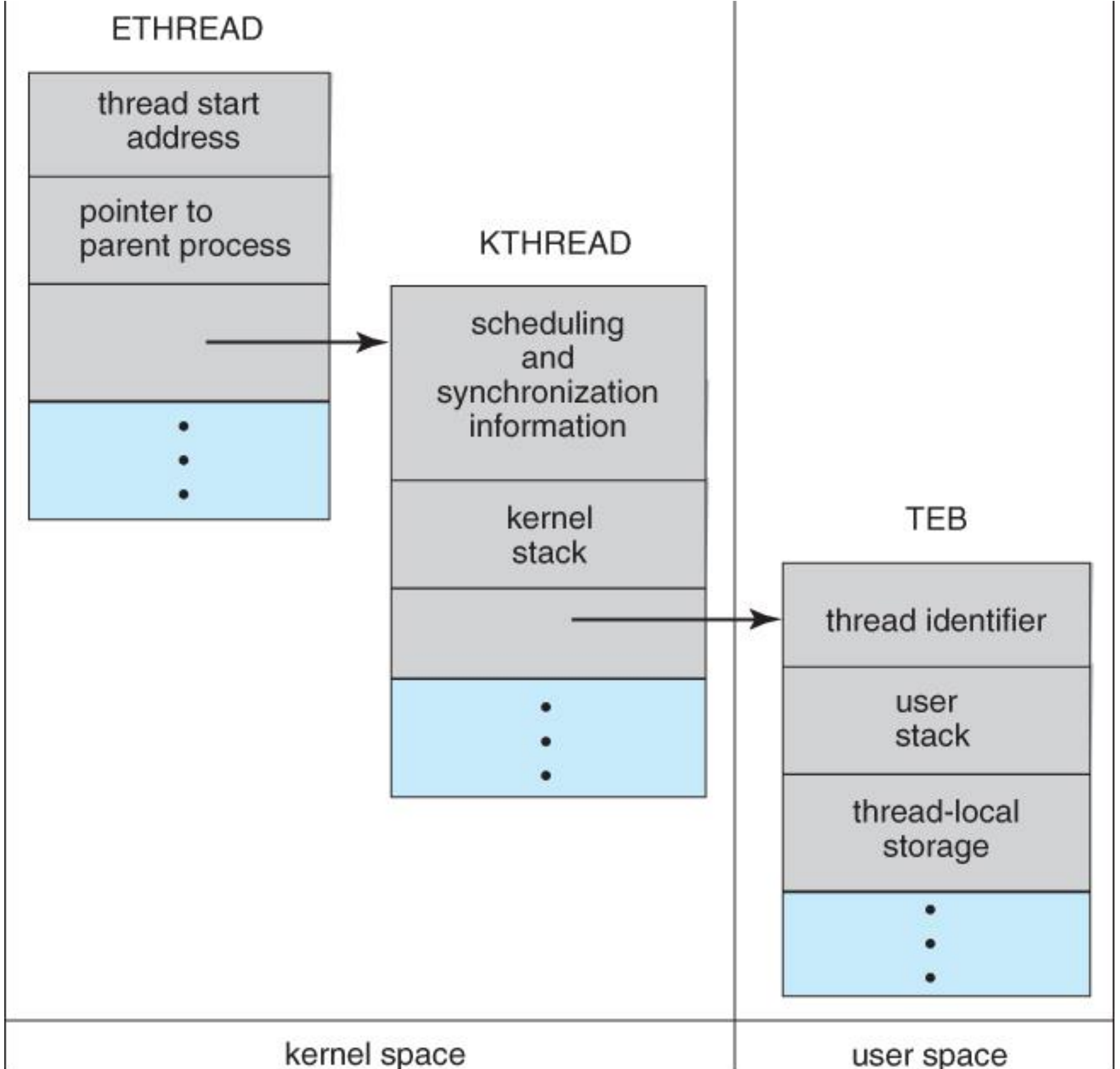


Figure 4.14 - Data structures of a Windows thread

4.7.2 Linux Threads

- Linux does not distinguish between processes and threads - It uses the more generic term "tasks".

- The traditional fork ( ) system call completely duplicates a process (task), as described earlier.
- An alternative system call, clone( ) allows for varying degrees of sharing between the parent and child tasks, controlled by flags such as those shown in the following table:

flag	**Meaning**
CLONE_FS	File-system information is shared
CLONE_VM	The same memory space is shared
CLONE_SIGHAND	Signal handlers are shared
CLONE_FILES	The set of open files is shared

- Calling clone( )with no flags set is equivalent to fork( ). Calling clone( ) with CLONE_FS, CLONE_VM, CLONE_SIGHAND, and CLONE_FILES is equivalent to creating a thread, as all of these data structures will be shared.
- Linux implements this using a structure **task_struct**, which essentially provides a level of indirection to task resources. When the flags are not set, then the resources pointed to by the structure are copied, but if the flags are set, then only the pointers to the resources are copied, and hence the resources are shared. (Think of a deep copy versus a shallow copy in OO programming.)
- Several distributions of Linux now support the NPTL (Native POXIS Thread Library)
    - POSIX compliant.
    - Support for SMP (symmetric multiprocessing), NUMA (non-uniform memory access), and multicore processors.
    - Support for hundreds to thousands of threads.

CHAPTER-5 : Process Synchronization

5.1 Background

- Recall that back in Chapter 3 we looked at cooperating processes (those that can effect or be effected by other simultaneously running processes), and as an example, we used the producer-consumer cooperating processes:

Producer code from chapter 3:

```
item nextProduced;

while( true ) {

/* Produce an item and store it in nextProduced */
nextProduced = makeNewItem( . . . );

/* Wait for space to become available */
while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
        ; /* Do nothing */

/* And then store the item and repeat the loop. */
buffer[ in ] = nextProduced;
in = ( in + 1 ) % BUFFER_SIZE;

}
```

Consumer code from chapter 3:

```
item nextConsumed;

while( true ) {

/* Wait for an item to become available */
while( in == out )
      ; /* Do nothing */

/* Get the next available item */
nextConsumed = buffer[ out ];
out = ( out + 1 ) % BUFFER_SIZE;

/* Consume the item in nextConsumed
     ( Do something with it ) */
```

```
}
```

- The only problem with the above code is that the maximum number of items which can be placed into the buffer is BUFFER_SIZE - 1. One slot is unavailable because there always has to be a gap between the producer and the consumer.
- We could try to overcome this deficiency by introducing a counter variable, as shown in the following code segments:

Producer Process:

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

Consumer Process:

```
while (true)
{
      while (counter == 0)
         ; /* do nothing */
      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;
      /* consume the item in nextConsumed */
}
```

- Unfortunately we have now introduced a new problem, because both the producer and the consumer are adjusting the value of the variable counter, which can lead to a condition known as a ***race condition***. In this condition a piece of code may or may not work correctly, depending on which of two simultaneous processes executes first, and more importantly if one of the processes gets interrupted such that the other process runs between important steps of the first process. (Bank balance example discussed in class.)

- The particular problem above comes from the producer executing "counter++" at the same time the consumer is executing "counter--". If one process gets part way through making the update and then the other process butts in, the value of counter can get left in an incorrect state.
- But, you might say, "Each of those are single instructions - How can they get interrupted halfway through?" The answer is that although they are single instructions in C++, they are actually three steps each at the hardware level: (1) Fetch counter from memory into a register, (2) increment or decrement the register, and (3) Store the new value of counter back to memory. If the instructions from the two processes get interleaved, there could be serious problems, such as illustrated by the following:

Producer:

$$\begin{aligned} ®ister_1 = \texttt{counter} \\ ®ister_1 = register_1 + 1 \\ &\texttt{counter} = register_1 \end{aligned}$$

Consumer:

$$\begin{aligned} ®ister_2 = \texttt{counter} \\ ®ister_2 = register_2 - 1 \\ &\texttt{counter} = register_2 \end{aligned}$$

Interleaving:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

- **Exercise:** What would be the resulting value of counter if the order of statements T4 and T5 were reversed? (What **should** the value of counter be after one producer and one consumer, assuming the original value was 5?)
- Note that race conditions are ***notoriously difficult*** to identify and debug, because by their very nature they only occur on rare occasions, and only when the timing is just exactly right. (or wrong! :-)) Race conditions are also very difficult to reproduce. :-(

- Obviously the solution is to only allow one process at a time to manipulate the value "counter". This is a very common occurrence among cooperating processes, so lets look at some ways in which this is done, as well as some classic problems in this area.

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**

To guard against the race condition , it is to be ensured that only one process at a time can be manipulating the variable counter and processes be synchronized in some manner.

5.2 The Critical-Section Problem

- The producer-consumer problem described above is a specific example of a more general situation known as the ***critical section*** problem. The general idea is that in a number of cooperating processes, each has a critical section of code, with the following conditions and terminologies:
    - Only one process in the group can be allowed to execute in their critical section at any one time. If one process is already executing their critical section and another process wishes to do so, then the second process must be made to wait until the first process has completed their critical section work.
    - The code preceding the critical section, and which controls access to the critical section, is termed the entry section. It acts like a carefully controlled locking door.
    - The code following the critical section is termed the exit section. It generally releases the lock on someone else's door, or at least lets the world know that they are no longer in their critical section.
    - The rest of the code not included in either the critical section or the entry or exit sections is termed the remainder section.

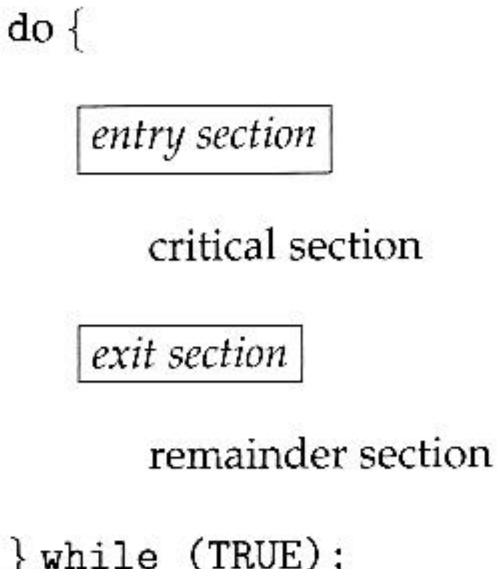


Figure 5.1 - General structure of a typical process Pi

In a system of processes (P0, P1, P2…. Pn), each process has a segment of code called a critical section, in which the process may be changing common variables, updating a table, writing a file etc.

When one process is executing in its critical section, no other process is to be allowed to execute in its critical section

Solution is to design a protocol that the processes can use to cooperate. Implemented by following sections:

- **Entry section**- includes implementation of code to grant permission to a process to enter its critical section.
- **Exit section**- follows the critical section.
- **Remainder Section**- remaining code of the protocol

- A solution to the critical section problem must satisfy the following three conditions:
  1. **Mutual Exclusion** - Only one process at a time can be executing in their critical section.
  2. **Progress** - If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then only the processes not in their remainder sections can participate in the decision, and the decision cannot be postponed indefinitely. (I.e. processes cannot be blocked forever waiting to get into their critical sections.)
  3. **Bounded Waiting** - There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their

critical section and before that request is granted. (I.e. a process requesting entry into their critical section will get a turn eventually, and there is a limit as to how many other processes get to go first.)

- We assume that all processes proceed at a non-zero speed, but no assumptions can be made regarding the ***relative*** speed of one process versus another.
- Kernel processes can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures such as open file tables or virtual memory management. Accordingly kernels can take on one of two forms:
  - Non-preemptive kernels do not allow processes to be interrupted while in kernel mode. This eliminates the possibility of kernel-mode race conditions, but requires kernel mode operations to complete very quickly, and can be problematic for real-time systems, because timing cannot be guaranteed.
  - Preemptive kernels allow for real-time operations, but must be carefully written to avoid race conditions. This can be especially tricky on SMP systems, in which multiple kernel processes may be running simultaneously on different processors.

Non-preemptive kernels include Windows XP, 2000, traditional UNIX, and Linux prior to 2.6; Preemptive kernels include Linux 2.6 and later, and some commercial UNIXes such as Solaris and IRIX.

5.3 Peterson's Solution

- Peterson's Solution is a classic software-based solution to the critical section problem. It is unfortunately not guaranteed to work on modern hardware, due to vagaries of load and store operations, but it illustrates a number of important concepts.
- Peterson's solution is based on two processes, P0 and P1, which alternate between their critical sections and remainder sections. For convenience of discussion, "this" process is Pi, and the "other" process is Pj. (I.e. j = 1 - i)
- Peterson's solution requires two shared data items:
  - **int turn** - Indicates whose turn it is to enter into the critical section. If turn = = i, then process i is allowed into their critical section.
  - **boolean flag[2]** - Indicates when a process ***wants to*** enter into their critical section. When process i wants to enter their critical section, it sets flag[i] to true.
- In the following diagram, the entry and exit sections are enclosed in boxes.
  - In the entry section, process i first raises a flag indicating a desire to enter the critical section.
  - Then turn is set to ***j*** to allow the ***other*** process to enter their critical section ***if process j so desires.***

- The while loop is a busy loop (notice the semicolon at the end), which makes process i wait as long as process j has the turn and wants to enter the critical section.
- Process i lowers the flag[i] in the exit section, allowing process j to continue if it has been waiting.

```
do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Figure 5.2 - The structure of process Pi in Peterson's solution.

- To prove that the solution is correct, we must examine the three conditions listed above:
  1. **Mutual exclusion** - If one process is executing their critical section when the other wishes to do so, the second process will become blocked by the flag of the first process. If both processes attempt to enter at the same time, the last process to execute "turn = j" will be blocked.
  2. **Progress** - Each process can only be blocked at the while if the other process wants to use the critical section (flag[j] = = true), AND it is the other process's turn to use the critical section (turn = = j). If both of those conditions are true, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, will set flag[j]

to false, releasing process i. The shared variable turn assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.

3. **Bounded Waiting** - As each process enters their entry section, they set the turn variable to be the other processes turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.

- Note that the instruction "turn = j" is ***atomic,*** that is it is a single machine instruction which cannot be interrupted.

Properties followed by this solution:

1. Mutual Exclusion: This condition is followed, explained in above example.
2. Progress: It is definitely followed as whichever process needs critical section, will make the INTERESTED value as true.
3. Bounded Waiting: This property is also followed as whichever process can make the TURN variable first, will get into critical section.
4. Platform Neutrality: yes because the solution is in user mode.

Disadvantage:

1. This solution works for 2 processes, but this solution is best scheme in user mode for critical section.
2. This is also a busy waiting solution so CPU time is wasted. And because of that “SPIN LOCK” problem can come. And this problem can come in any of the busy waiting solution.

5.4 Synchronization Hardware

- To generalize the solution(s) expressed above, each process when entering their critical section must set some sort of ***lock***, to prevent other processes from entering their critical sections simultaneously, and must release the lock when exiting their critical section, to allow other processes to proceed. Obviously it must be possible to attain the lock only when no other process has already set a lock. Specific implementations of this general procedure can get quite complicated, and may include hardware solutions as outlined in this section.
- One simple solution to the critical section problem is to simply prevent a process from being interrupted while in their critical section, which is the approach taken by non preemptive kernels. Unfortunately this does not work

well in multiprocessor environments, due to the difficulties in disabling and the re-enabling interrupts on all processors. There is also a question as to how this approach affects timing if the clock interrupt is disabled.

- Another approach is for hardware to provide certain ***atomic*** operations. These operations are guaranteed to operate as a single instruction, without interruption. One such operation is the "Test and Set", which simultaneously sets a boolean lock variable and returns its previous value, as shown in Figures 5.3 and 5.4:

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Figure 5.3 The definition of the TestAndSet() instruction.

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Figure 5.4 Mutual-exclusion implementation with TestAndSet().

Figures 5.3 and 5.4 illustrate "test_and_set( )" function

- Another variation on the test-and-set is an atomic swap of two booleans, as shown in Figures 5.5 and 5.6:

```
int compare_and_swap(int *value, int expected, int new_value) {
  int temp = *value;

  if (*value == expected)
    *value = new_value;

  return temp;
}
```

Figure 5.5 The definition of the compare_and_swap() instruction.

```
do {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ; /* do nothing */

    /* critical section */

  lock = 0;

    /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the compare_and_swap() instruction.

- The above examples satisfy the mutual exclusion requirement, but unfortunately do not guarantee bounded waiting. If there are multiple processes trying to get into their critical sections, there is no guarantee of what order they will enter, and any one process could have the bad luck to wait forever until they got their turn in the critical section. (Since there is no guarantee as to the relative ***rates*** of the processes, a very fast process could theoretically release the lock, whip through their remainder section, and re-lock the lock before a slower process got a chance. As more and more processes are involved vying for the same resource, the odds of a slow process getting locked out completely increase.)
- Figure 5.7 illustrates a solution using test-and-set that does satisfy this requirement, using two shared data structures, `boolean`

`lock` and `boolean waiting[ N ]`, where N is the number of processes in contention for critical sections:

```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
}while (TRUE);
```

Figure 5.7 Bounded-waiting mutual exclusion with TestAndSet( ).

- The key feature of the above algorithm is that a process blocks on the AND of the critical section being locked and that this process is in the waiting state. When exiting a critical section, the exiting process does not just unlock the critical section and let the other processes have a free-for-all trying to get in. Rather it first looks in an orderly progression (starting with the next process on the list) for a process that has been waiting, and if it finds one, then it releases that particular process from its waiting state, without unlocking the critical section, thereby allowing a specific process into the critical section while continuing to block all the others. Only if there are no other processes currently waiting is the general lock removed, allowing the next process to come along access to the critical section.

- Unfortunately, hardware level locks are especially difficult to implement in multi-processor architectures. Discussion of such issues is left to books on advanced computer architecture.

5.5 Mutex Locks

- The hardware solutions presented above are often difficult for ordinary programmers to access, particularly on multi-processor machines, and particularly because they are often platform-dependent.
- Therefore most systems offer a software API equivalent called ***mutex locks*** or simply ***mutexes.*** (For mutual exclusion)
- The terminology when using mutexes is to ***acquire*** a lock prior to entering a critical section, and to ***release*** **it when exiting, as shown in Figure 5.8:**

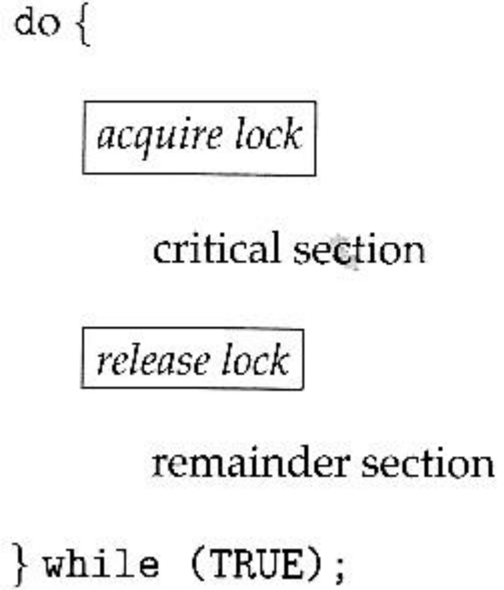


Figure 5.8 - Solution to the critical-section problem using mutex locks

- Just as with hardware locks, the acquire step will block the process if the lock is in use by another process, and both the acquire and release operations are atomic.
- Acquire and release can be implemented as shown here, based on a boolean variable "available":

Acquire:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

Release:

```
release() {
   available = true;
}
```

- One problem with the implementation shown here, (and in the hardware solutions presented earlier), is the busy loop used to block processes in the acquire phase. These types of locks are referred to as ***spinlocks***, because the CPU just sits and spins while blocking the process.
- Spinlocks are wasteful of cpu cycles, and are a really bad idea on single-cpu single-threaded machines, because the spinlock blocks the entire computer, and doesn't allow any other process to release the lock. (Until the scheduler kicks the spinning process off of the cpu.)
- On the other hand, spinlocks do not incur the overhead of a context switch, so they are effectively used on multi-threaded machines when it is expected that the lock will be released after a short time.

5.6 Semaphores

- A more robust alternative to simple mutexes is to use ***semaphores***, which are integer variables for which only two (atomic) operations are defined, the wait and signal operations, as shown in the following figure.
- Note that not only must the variable-changing steps (S-- and S++) be indivisible, it is also necessary that for the wait operation when the test proves false that there be no interruptions before S gets decremented. It IS okay, however, for the busy loop to be interrupted when the test is true, which prevents the system from hanging forever.

Wait:

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

Signal:

```
signal(S) {
    S++;
}
```

5.6.1 Semaphore Usage

- In practice, semaphores can take on one of two forms:
    - **Binary semaphores** can take on one of two values, 0 or 1. They can be used to solve the critical section problem as described above, and can be used as mutexes on systems that do not provide a separate mutex mechanism.. The use of mutexes for this purpose is shown in Figure 6.9 (from the 8th edition) below.

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Mutual-exclusion implementation with semaphores. (From 8th edition.)

-
    - **Counting semaphores** can take on any integer value, and are usually used to count the number remaining of some limited resource. The counter is initialized to the number of such resources available in the system, and whenever the counting semaphore is greater than zero, then

a process can enter a critical section and use one of the resources. When the counter gets to zero (or negative in some implementations), then the process blocks until another process frees up a resource and increments the counting semaphore with a signal call. (The binary semaphore can be seen as just a special case where the number of resources initially available is just one.)

- Semaphores can also be used to synchronize certain operations between processes. For example, suppose it is important that process P1 execute statement S1 before process P2 executes statement S2.
    - First we create a semaphore named synch that is shared by the two processes, and initialize it to zero.
    - Then in process P1 we insert the code:

```
S1;
signal( synch );
```

    - and in process P2 we insert the code:

```
wait( synch );
S2;
```

    - Because synch was initialized to 0, process P2 will block on the wait until after P1 executes the call to signal.

5.6.2 Semaphore Implementation

- The big problem with semaphores as described above is the busy loop in the wait call, which consumes CPU cycles without doing any useful work. This type of lock is known as a ***spinlock***, because the lock just sits there and spins while it waits. While this is generally a bad thing, it does have the advantage of not invoking context switches, and so it is sometimes used in multi-processing systems when the wait time is expected to be short - One thread spins on one processor while another completes their critical section on another processor.
- An alternative approach is to block a process when it is forced to wait for an available semaphore, and swap it out of the CPU. In this implementation each semaphore needs to maintain a list of processes that are blocked waiting for it, so that one of the processes can be woken up and swapped back in when the semaphore becomes available. (Whether it gets swapped back into the CPU immediately or whether it needs to hang out in the ready queue for a while is a scheduling problem.)
- The new definition of a semaphore and the corresponding wait and signal operations are shown as follows:

Semaphore Structure:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Wait Operation:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
}
```

Signal Operation:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

- Note that in this implementation the value of the semaphore can actually become negative, in which case its magnitude is the number of processes waiting for that semaphore. This is a result of decrementing the counter before checking its value.
- Key to the success of semaphores is that the wait and signal operations be atomic, that is no other process can execute a wait or signal on the same semaphore at the same time. (Other processes could be allowed to do other things, including working with other semaphores, they just can't have access to **this** semaphore.) On single processors this can be implemented by disabling interrupts during the execution of wait and signal; Multiprocessor systems have to use more complex methods, including the use of spinlocking.

5.6.3 Deadlocks and Starvation

- One important problem that can arise when using semaphores to block processes waiting for a limited resource is the problem of ***deadlocks***, which occur when multiple processes are blocked, each waiting for a resource that can only be freed by one of the other (blocked) processes, as illustrated in the following example. (Deadlocks are covered more completely in chapter 7.)

```
     P0                P1

wait(S);          wait(Q);
wait(Q);          wait(S);
     .                 .
     .                 .
     .                 .
signal(S);        signal(Q);
signal(Q);        signal(S);
```

- Another problem to consider is that of ***starvation***, in which one or more processes gets blocked forever, and never get a chance to take their turn in the critical section. For example, in the semaphores above, we did not specify the algorithms for adding processes to the waiting queue in the semaphore in the wait( ) call, or selecting one to be removed from the queue in the signal( ) call. If the method chosen is a FIFO queue, then every process will eventually get their turn, but if a LIFO queue is implemented instead, then the first process to start waiting could starve.

5.6.4 Priority Inversion

- A challenging scheduling problem arises when a high-priority process gets blocked waiting for a resource that is currently held by a low-priority process.
- If the low-priority process gets pre-empted by one or more medium-priority processes, then the high-priority process is essentially made to wait for the medium priority processes to finish before the low-priority process can release the needed resource, causing a ***priority inversion.*** If there are enough medium-priority processes, then the high-priority process may be forced to wait for a very long time.
- One solution is a ***priority-inheritance protocol,*** in which a low-priority process holding a resource for which a high-priority process is waiting will temporarily

inherit the high priority from the waiting process. This prevents the medium-priority processes from preempting the low-priority process until it releases the resource, blocking the priority inversion problem.

- The book has an interesting discussion of how a priority inversion almost doomed the Mars Pathfinder mission, and how the problem was solved when the priority inversion was stopped. Full details are available online

5.7 Classic Problems of Synchronization

The following classic problems are used to test virtually every new proposed synchronization algorithm.

5.7.1 The Bounded-Buffer Problem

- This is a generalization of the producer-consumer problem wherein access is controlled to a shared group of buffers of a limited size.
- In this solution, the two counting semaphores "full" and "empty" keep track of the current number of full and empty buffers respectively (and initialized to 0 and N respectively.) The binary semaphore mutex controls access to the critical section. The producer and consumer processes are nearly identical - One can think of the producer as producing full buffers, and the consumer producing empty buffers.

```
do {
      . . .
   // produce an item in nextp
      . . .
   wait(empty);
   wait(mutex);
      . . .
   // add nextp to buffer
      . . .
   signal(mutex);
   signal(full);
}while (TRUE);
```

Figure 5.9 The structure of the producer process.

```
do {
   wait(full);
   wait(mutex);
      . . .
   // remove an item from buffer to nextc
      . . .
   signal(mutex);
   signal(empty);
      . . .
   // consume the item in nextc
      . . .
}while (TRUE);
```

Figure 5.10 The structure of the consumer process.

Figures 5.9 and 5.10 use variables next_produced and next_consumed

5.7.2 The Readers-Writers Problem

- In the readers-writers problem there are some processes (termed readers) who only read the shared data, and never change it, and there are other processes (termed writers) who may change the data in addition to or instead

of reading it. There is no limit to how many readers can access the data simultaneously, but when a writer accesses the data, it needs exclusive access.

- There are several variations to the readers-writers problem, most centered around relative priorities of readers versus writers.
    - The *first* readers-writers problem gives priority to readers. In this problem, if a reader wants access to the data, and there is not already a writer accessing it, then access is granted to the reader. A solution to this problem can lead to starvation of the writers, as there could always be more readers coming along to access the data. (A steady stream of readers will jump ahead of waiting writers as long as there is currently already another reader accessing the data, because the writer is forced to wait until the data is idle, which may never happen if there are enough readers.)
    - The *second* readers-writers problem gives priority to the writers. In this problem, when a writer wants access to the data it jumps to the head of the queue - All waiting readers are blocked, and the writer gets access to the data as soon as it becomes available. In this solution the readers may be starved by a steady stream of writers.
- The following code is an example of the first readers-writers problem, and involves an important counter and two binary semaphores:
    - `readcount` is used by the reader processes, to count the number of readers currently accessing the data.
    - `mutex` is a semaphore used only by the readers for controlled access to `readcount`.
    - `rw_mutex` is a semaphore used to block and release the writers. The first reader to access the data will set this lock and the last reader to exit will release it; The remaining readers do not touch `rw_mutex`. (Eighth edition called this variable `wrt`.)
    - Note that the first reader to come along will block on `rw_mutex` if there is currently a writer accessing the data, and that all following readers will only block on `mutex` for their turn to increment `readcount`.

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Figure 5.11 The structure of a writer process.

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Figure 5.12 The structure of a reader process.

- Some hardware implementations provide specific reader-writer locks, which are accessed using an argument specifying whether access is requested for reading or writing. The use of reader-writer locks is beneficial for situation in which: (1) processes can be easily identified as either readers or writers, and (2) there are significantly more readers than writers, making the additional overhead of the reader-writer lock pay off in terms of increased concurrency of the readers.

5.7.3 The Dining-Philosophers Problem

- The dining philosophers problem is a classic synchronization problem involving the allocation of limited resources amongst a group of processes in a deadlock-free and starvation-free manner:
    - Consider five philosophers sitting around a table, in which there are five chopsticks evenly distributed and an endless bowl of rice in the center, as shown in the diagram below. (There is exactly one chopstick between each pair of dining philosophers.)
    - These philosophers spend their lives alternating between two activities: eating and thinking.
    - When it is time for a philosopher to eat, it must first acquire two chopsticks - one from their left and one from their right.
    - When a philosopher thinks, it puts down both chopsticks in their original locations.

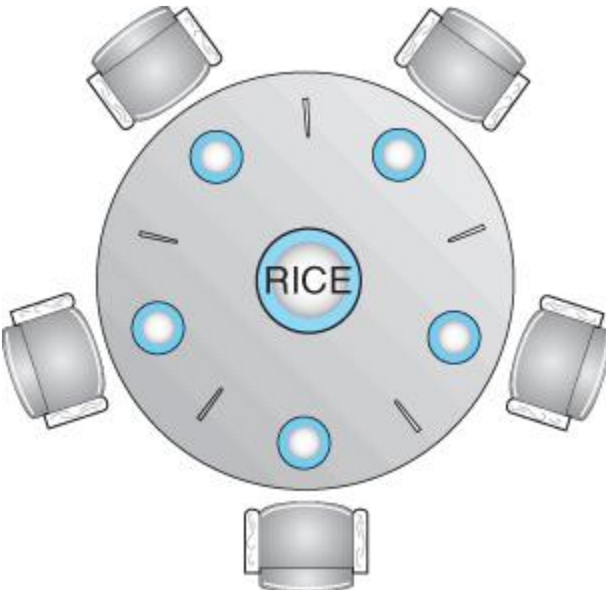


Figure 5.13 - The situation of the dining philosophers

- One possible solution, as shown in the following code section, is to use a set of five semaphores (chopsticks[5]), and to have each hungry philosopher first wait on their left chopstick (chopsticks[i]), and then wait on their right chopstick (chopsticks[(i + 1) % 5])
- But suppose that all five philosophers get hungry at the same time, and each starts by picking up their left chopstick. They then look for their right chopstick, but because it is unavailable, they wait for it, forever, and eventually all the philosophers starve due to the resulting deadlock.

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
}while (TRUE);
```

Figure 5.14 - The structure of philosopher i.

- Some potential solutions to the problem include:
    - Only allow four philosophers to dine at the same time. (Limited simultaneous processes.)
    - Allow philosophers to pick up chopsticks only when both are available, in a critical section. (All or nothing allocation of critical resources.)
    - Use an asymmetric solution, in which odd philosophers pick up their left chopstick first and even philosophers pick up their right chopstick first. (Will this solution always work? What if there are an even number of philosophers?)
- Note carefully that a deadlock-free solution to the dining philosophers problem does not necessarily guarantee a starvation-free one. (While some or even most of the philosophers may be able to get on with their normal lives of eating and thinking, there may be one unlucky soul who never seems to be able to get both chopsticks at the same time. :

(OR)

Classical Problems of Synchronization

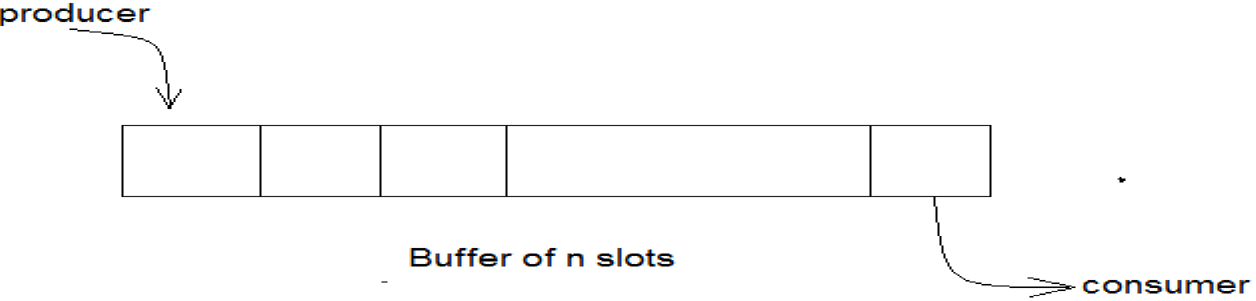
1. Bounded Buffer (Producer-Consumer) Problem
2. Dining Philosophers Problem
3. The Readers Writers Problem

Bounded Buffer Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

What is the Problem Statement?

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.
There needs to be a way to make the producer and consumer work in an independent manner.

Solution

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- `mutex`, a **binary semaphore** which is used to acquire and release the lock.initial value is 1.
- `empty`, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- `full`, a **counting semaphore** whose initial value is `0`.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

The Producer Operation

The pseudocode of the producer function looks like this:

```
do
{
    // wait until empty > 0 and then
decrement 'empty'
    wait(empty);
    // acquire lock
    wait(mutex);

    /* perform the insert operation in a slot
*/
   // release lock
    signal(mutex);
    // increment 'full'
    signal(full);
}
```

- Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.

- Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

The Consumer Operation

The pseudocode for the consumer function looks like this:

```
do
{
    // wait until full > 0 and then decrement 'full'
    wait(full);
    // acquire the lock
    wait(mutex);

    /* perform the remove operation in a slot */

    // release the lock
    signal(mutex);
    // increment 'empty'
    signal(empty);
}
while(TRUE);
```

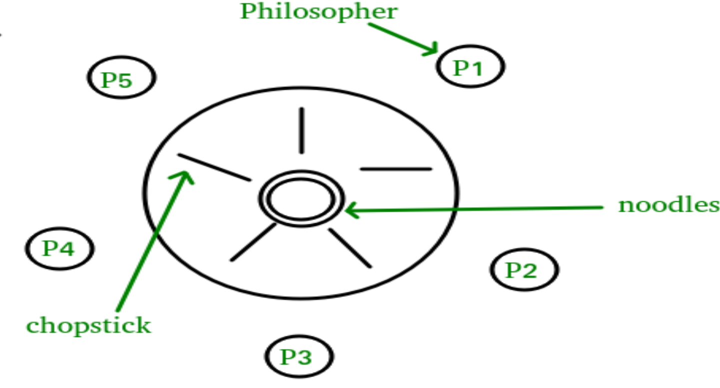
- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

Dining Philosophers Problem

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

What is the Problem Statement?

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



- There is one chopstick between each philosopher
- A philosopher must pick up its two nearest chopsticks in order to eat
- A philosopher must pick up first one chopstick, then the second one, not both at once

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Here's the Solution

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, `stick[5]`, for each of the five chopsticks. The code for each philosopher looks like:

```
while(TRUE)
{
    wait(stick[i]);
    /*
        mod is used because if i=5, next
        chopstick is 1 (dining table is circular)
    */
    wait(stick[(i+1) % 5]);

    /* eat */
    signal(stick[i]);

    signal(stick[(i+1) % 5]);
    /* think */}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

Readers Writer Problem

- Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

- The Problem Statement
- There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

- The Solution

- From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.
- Here, we use one **mutex** `m` and a **semaphore** `w`. An integer variable `read_count` is used to maintain the number of readers currently accessing the resource. The variable `read_count` is initialized to `0`. A value of `1` is given initially to `m` and `w`.
- Instead of having the process to acquire lock on the shared resource, we use the mutex `m` to make the process to acquire and release lock whenever it is updating the `read_count` variable.

  The code for the **writer** process looks like this:

```
while(TRUE)
{
    wait(w);

   /* perform the write operation */

   signal(w);
}
```

And, the code for the **reader** process looks like this:

```
while(TRUE)
{
    //acquire lock
    wait(m);
    read_count++;
    if(read_count == 1)
        wait(w);
        //release lock
    signal(m);

    /* perform the reading operation */
```

```
    // acquire lock
    wait(m);
    read_count--;
    if(read_count == 0)
        signal(w);

    // release lock
    signal(m);
}
```

Here is the Code uncoded(explained)

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- When a reader wants to access the resource, first it increments the **read_count**value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

5.8 Monitors

- Semaphores can be very useful for solving concurrency problems, ***but only if programmers use them properly.*** If even one process fails to abide by the proper use of semaphores, either accidentally or deliberately, then the whole system breaks down. (And since concurrency problems are by definition rare events, the problem code may easily go unnoticed and/or be heinous to debug.)
- For this reason a higher-level language construct has been developed, called ***monitors***.

5.8.1 Monitor Usage

- Monitor is a programming language construct that controls access to shared data
  synchronization code added by the compiler
  synchronization enforced by the runtime
- Monitor is an abstract data type (ADT) that encapsulates
  shared data structures
  procedures that operate on the shared data structures
  synchronization between the concurrent procedure invocations
- Protects the shared data structures inside the monitor from outside access.
- Guarantees that monitor procedures (or operations) can only legitimately update the shared data.
- A monitor is essentially a class, in which all data is private, and with the special restriction that only one method within any given monitor object may be active at the same time. An additional restriction is that monitor methods may only access the shared data within the monitor and any data passed to them as parameters. I.e. they cannot access any data external to the monitor.

```
monitor monitor name
{
  // shared variable declarations

  procedure P1 ( . . . ) {
    . . .
  }

  procedure P2 ( . . . ) {
    . . .
  }

        .
        .
        .
  procedure Pn ( . . . ) {
    . . .
  }

  initialization code ( . . . ) {
    . . .
  }
}
```

Figure 5.15 - Syntax of a monitor.

- Figure 5.16 shows a schematic of a monitor, with an entry queue of processes waiting their turn to execute monitor operations (methods).

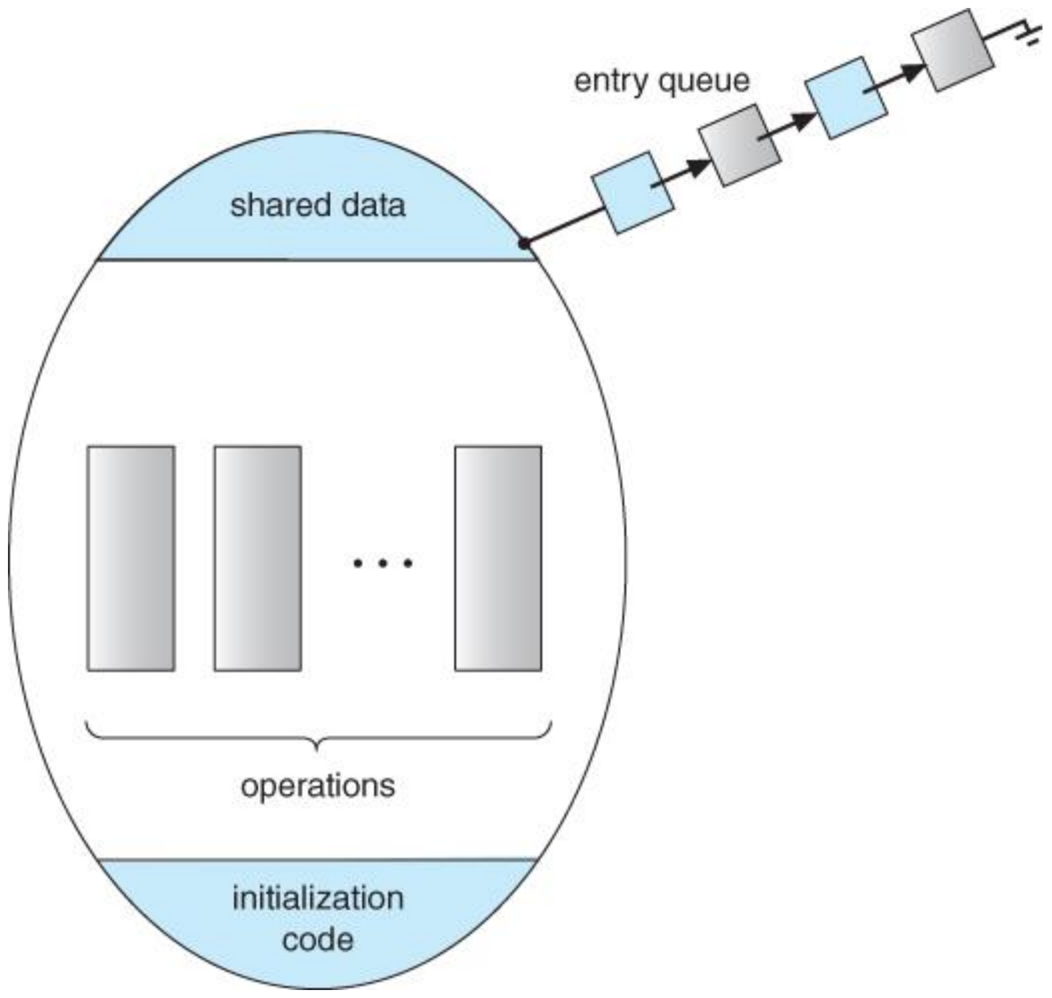


Figure 5.16 - Schematic view of a monitor

- In order to fully realize the potential of monitors, we need to introduce one additional new data type, known as a ***condition***.
  - A variable of type condition has only two legal operations, ***wait*** and ***signal****.* I.e. if X was defined as type condition, then legal operations would be X.wait( ) and X.signal( )
  - The wait operation blocks a process until some other process calls signal, and adds the blocked process onto a list associated with that condition.
  - The signal process does nothing if there are no processes waiting on that condition. Otherwise it wakes up exactly one process from the condition's list of waiting processes. (Contrast this with counting semaphores, which always affect the semaphore on a signal call.)
- Figure 6.18 below illustrates a monitor that includes condition variables within its data space. Note that the condition variables, along with the list of processes currently waiting for the conditions, are in the data space of the monitor - The processes on these lists are not "in" the monitor, in the sense that they are not executing any code in the monitor.

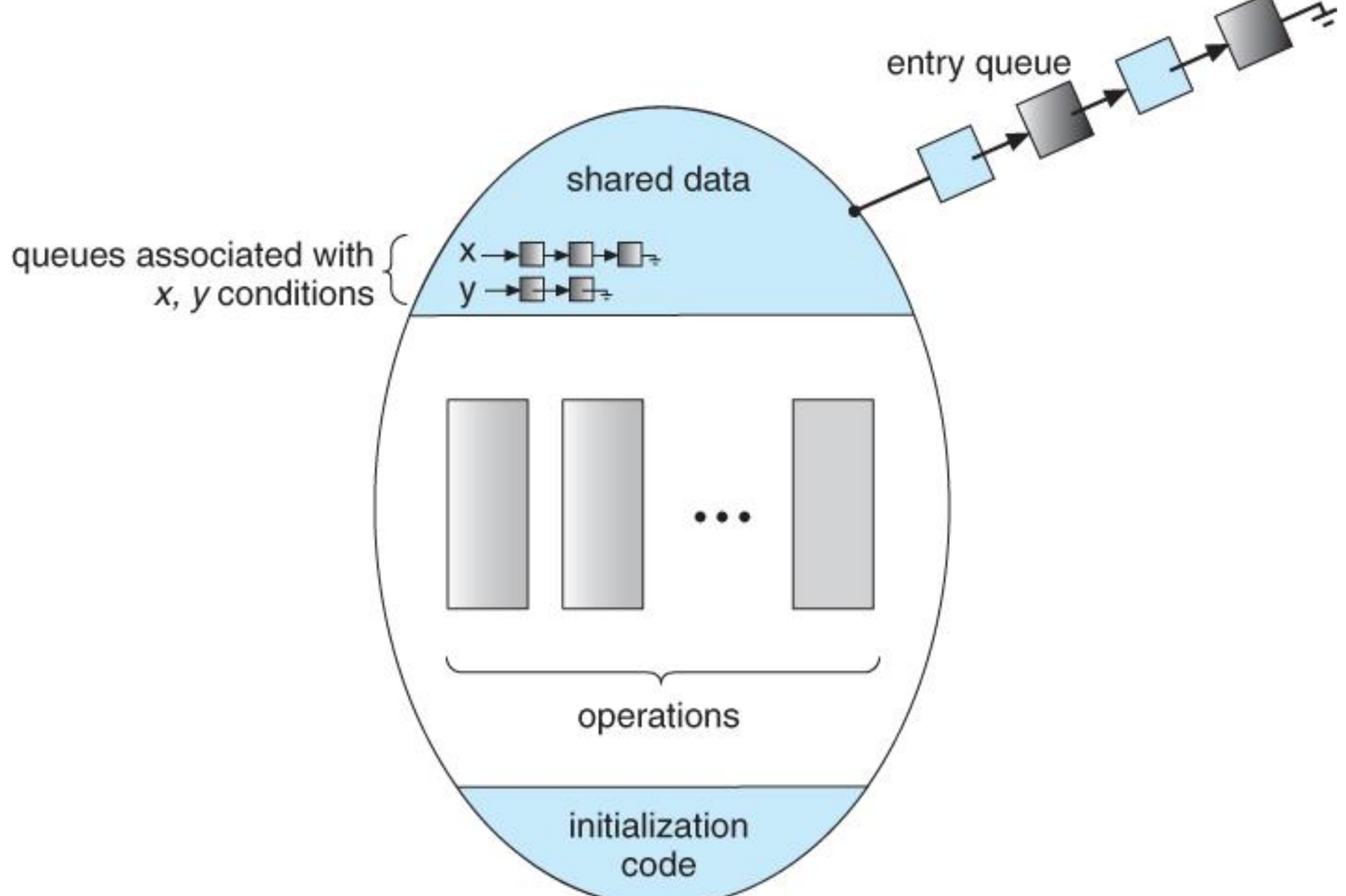


Figure 5.17 - Monitor with condition variables

- But now there is a potential problem - If process P within the monitor issues a signal that would wake up process Q also within the monitor, then there would be two processes running simultaneously within the monitor, violating the exclusion requirement. Accordingly there are two possible solutions to this dilemma:

Signal and wait - When process P issues the signal to wake up process Q, P then waits, either for Q to leave the monitor or on some other condition.

Signal and continue - When P issues the signal, Q waits, either for P to exit the monitor or for some other condition.

There are arguments for and against either choice. Concurrent Pascal offers a third alternative - The signal call causes the signaling process to immediately exit the monitor, so that the waiting process can then wake up and proceed.

- Java and C# (C sharp) offer monitors bulit-in to the language. Erlang offers similar but different constructs.

Bounded Buffer Using Monitors

```
Monitor bounded_buffer {

   Resource buffer[N];

   // condition variables

   Condition empty, full;

   void producer (Resource R) {

      while (buffer full)

         empty.wait( );

      // add R to buffer array

      full.signal( );

   }

Void  consumer ( ) {

      while (buffer empty)

         full.wait( );

      // get Resource from buffer

      empty.signal( );

      return R;

   }

} // end monitor
```

5.8.2 Dining-Philosophers Solution Using Monitors

- This solution to the dining philosophers uses monitors, and the restriction that a philosopher may only pick up chopsticks when both are available. There are also two key data structures in use in this solution:
    1. **`enum { THINKING, HUNGRY,EATING } state[ 5 ];`** A philosopher may only set their state to eating when neither of their adjacent neighbors is eating. (state[(i + 1) % 5] != EATING && state[(i + 4) % 5] != EATING).
    2. **`condition self[ 5 ];`** This condition is used to delay a hungry philosopher who is unable to acquire chopsticks.

- In the following solution philosophers share a monitor, DiningPhilosophers, and eat using the following sequence of operations:
    1. DiningPhilosophers.pickup( ) - Acquires chopsticks, which may block the process.
    2. eat
    3. DiningPhilosophers.putdown( ) - Releases the chopsticks.

```
monitor DiningPhilosophers
{
  enum {THINKING, HUNGRY, EATING} state[5];
  condition self[5];

  void pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
      self[i].wait();
  }

  void putdown(int i) {
    state[i] = THINKING;
    test((i + 4) % 5);
    test((i + 1) % 5);
  }

  void test(int i) {
    if ((state[(i + 4) % 5] != EATING) &&
     (state[i] == HUNGRY) &&
     (state[(i + 1) % 5] != EATING)) {
       state[i] = EATING;
       self[i].signal();
    }
  }

  initialization_code() {
    for (int i = 0; i < 5; i++)
      state[i] = THINKING;
  }
}
```

Figure 5.18 A monitor solution to the dining-philosopher problem.

5.8.3 Implementing a Monitor Using Semaphores

- One possible implementation of a monitor uses a semaphore "mutex" to control mutual exclusionary access to the monitor, and a counting semaphore

"next" on which processes can suspend themselves after they are already "inside" the monitor (in conjunction with condition variables, see below.) The integer next_count keeps track of how many processes are waiting in the next queue. Externally accessible monitor processes are then implemented as:

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

- Condition variables can be implemented using semaphores as well. For a condition x, a semaphore "x_sem" and an integer "x_count" are introduced, both initialized to zero. The wait and signal methods are then implemented as follows. (This approach to the condition implements the signal-and-wait option described above for ensuring that only one process at a time is active inside the monitor.)

Wait:

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

Signal:

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

5.8.4 Resuming Processes Within a Monitor

- When there are multiple processes waiting on the same condition within a monitor, how does one decide which one to wake up in response to a signal on that condition? One obvious approach is FCFS, and this may be suitable in many cases.
- Another alternative is to assign (integer) priorities, and to wake up the process with the smallest (best) priority.
- Figure 5.19 illustrates the use of such a condition within a monitor used for resource allocation. Processes wishing to access this resource must specify the time they expect to use it using the acquire(time) method, and must call the release( ) method when they are done with the resource.

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = TRUE;
   }

   void release() {
      busy = FALSE;
      x.signal();
   }

   initialization_code() {
      busy = FALSE;
   }
}
```

Figure 5.19 - A monitor to allocate a single resource.

- Unfortunately the use of monitors to restrict access to resources still only works if programmers make the requisite acquire and release calls properly. One option would be to place the resource allocation code into the monitor, thereby eliminating the option for programmers to bypass or ignore the monitor, but then that would substitute the monitor's scheduling algorithms for whatever other scheduling algorithms may have been chosen for that particular resource. Chapter 14 on Protection presents more advanced methods for enforcing "nice" cooperation among processes contending for shared resources.
- Concurrent Pascal, Mesa, C#, and Java all implement monitors as described here. Erlang provides concurrency support using a similar mechanism.

5.9 Synchronization Examples

This section looks at how synchronization is handled in a number of different systems.

5.9.1 Synchronization in Windows

Figure 5.20 - Mutex dispatcher object

5.9.2 Synchronization in Linux

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

5.9.3 Synchronization in Solaris

- Solaris controls access to critical sections using five tools: semaphores, condition variables, adaptive mutexes, reader-writer locks, and turnstiles. The first two are as described above, and the other three are described here:

Adaptive Mutexes

- Adaptive mutexes are basically binary semaphores that are implemented differently depending upon the conditions:
    - On a single processor system, the semaphore sleeps when it is blocked, until the block is released.
    - On a multi-processor system, if the thread that is blocking the semaphore is running on the same processor as the thread that is blocked, or if the blocking thread is not running at all, then the blocked thread sleeps just like a single processor system.
    - However if the blocking thread is currently running on a different processor than the blocked thread, then the blocked thread does a spinlock, under the assumption that the block will soon be released.
    - Adaptive mutexes are only used for protecting short critical sections, where the benefit of not doing context switching is worth a short bit of spinlocking. Otherwise traditional semaphores and condition variables are used.

Reader-Writer Locks

- Reader-writer locks are used only for protecting longer sections of code which are accessed frequently but which are changed infrequently.

Turnstiles

- A ***turnstile*** is a queue of threads waiting on a lock.
- Each synchronized object which has threads blocked waiting for access to it needs a separate turnstile. For efficiency, however, the turnstile is associated with the thread currently holding the object, rather than the object itself.
- In order to prevent ***priority inversion,*** the thread holding a lock for an object will temporarily acquire the highest priority of any process in the turnstile waiting for the blocked object. This is called a ***priority-inheritance protocol.***
- User threads are controlled the same as for kernel threads, except that the priority-inheritance protocol does not apply.

5.10 Alternate Approaches

- Transactional Memory
- OpenMP
- Functional Programming Langua

Transactional Memory

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.

```
void update()
{
/*    read/write   memory    */
}
```

OpenMP

OpenMP

- OpenMP is a set of compiler directives and API that support parallel progamming.

```
void  update(int  value)
{
        #pragma omp  critical
        {
                count  += value
        }
}
```

The code contained within the #pragma omp critical directive is treated as a critical section and performed atomically.

Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
- Variables are treated as immutable and cannot change state once they have been assigned a value.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races

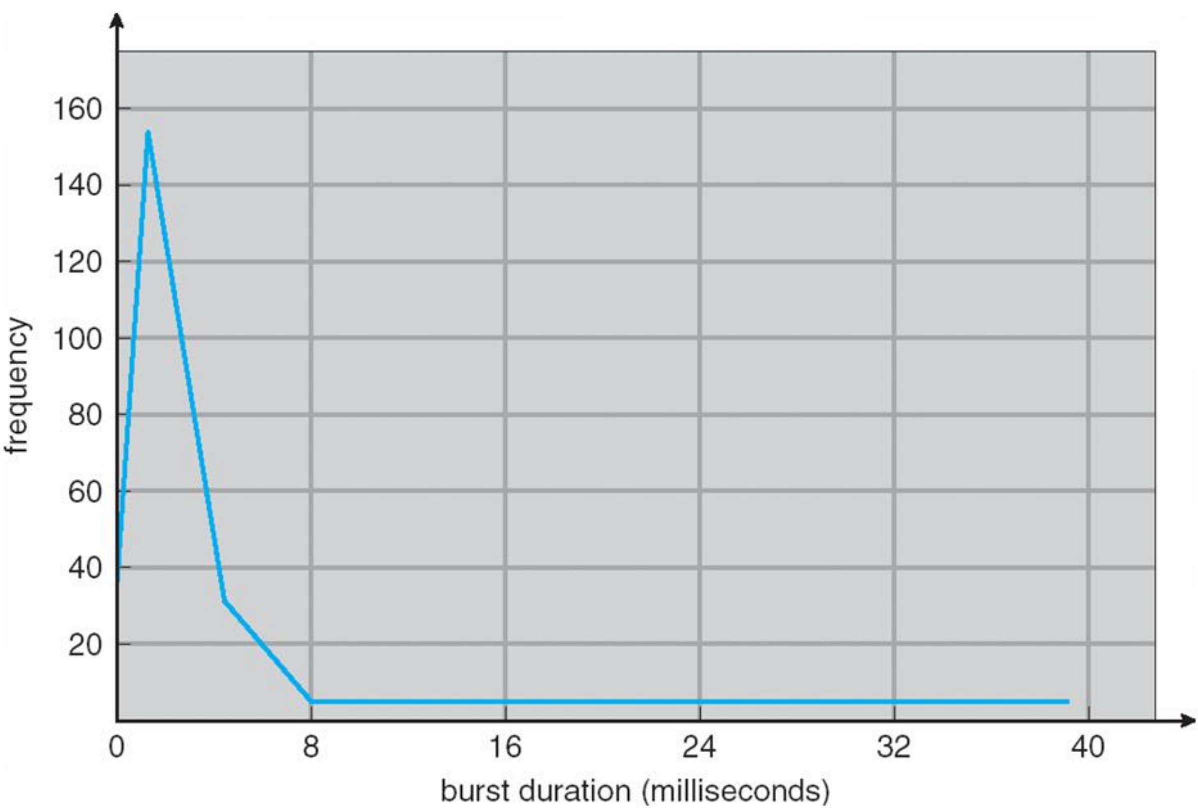
Chapter-6

CPU Scheduling

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

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 from among the processes in memory that are ready to execute, and allocates the CPU to one of them

Histogram of CPU-burst Times



Indicates there are less no.of longer cpu burst processes and more no.of smaller cpu bursts

CPU 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:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program from where it left last time.

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time taken by the dispatcher to stop one process and start another process is known as the **Dispatch Latency**. Dispatch Latency can be explained using the below figure:

Types of CPU 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 I/O request or 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, completion of I/O).
4. When a process **terminates**.

In circumstances 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 in circumstances 2 and 3.

When Scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is **non-preemptive**; otherwise the scheduling scheme is **preemptive**.

Non-Preemptive Scheduling

Under non-preemptive 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.

This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems.

It is the only method that can be used on certain hardware platforms, because It does not require the special hardware(for example: a timer) needed for preemptive scheduling.

Preemptive Scheduling

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

CPU Scheduling: Scheduling Criteria

There are many different criterias to check when considering the **"best"** scheduling algorithm, they are:

CPU Utilization

To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

Throughput

It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

Turnaround Time

It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).

Waiting Time

The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

Load Average

It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

Response Time

Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

Scheduling Algorithm Optimization

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Scheduling Algorithms

To decide which process to execute first and which process to execute last to achieve maximum CPU utilisation, computer scientists have defined some algorithms, they are:

1. First Come First Serve(FCFS) Scheduling
2. Shortest-Job-First(SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

First Come First Serve Scheduling

In the "First come first serve" scheduling algorithm, as the name suggests, the process which arrives first, gets executed first, or we can say that the process which requests the CPU first, gets the CPU allocated first.

- First Come First Serve, is just like **FIFO**(First in First out) Queue data structure, where the data element which is added to the queue first, is the one who leaves the queue first.
- This is used in Batch Systems.
- It's **easy to understand and implement** programmatically, using a Queue data structure, where a new process enters through the **tail** of the queue, and the scheduler selects process from the **head** of the queue.
- A perfect real life example of FCFS scheduling is **buying tickets at ticket counter**.

Calculating Average Waiting Time

For every scheduling algorithm, **Average waiting time** is a crucial parameter to judge it's performance.

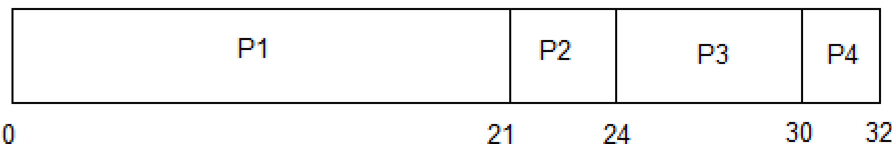
AWT or Average waiting time is the average of the waiting times of the processes in the queue, waiting for the scheduler to pick them for execution.

Lower the Average Waiting Time, better the scheduling algorithm.

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with **Arrival Time** `0`, and given **Burst Time**, let's find the average waiting time using the FCFS scheduling algorithm.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms



This is the GANTT chart for the above processes

The average waiting time will be `18.75 ms`

For the above given processes, first **P1** will be provided with the CPU resources,

- Hence, waiting time for **P1** will be `0`
- **P1** requires `21 ms` for completion, hence waiting time for **P2** will be `21 ms`
- Similarly, waiting time for process **P3** will be execution time of **P1** + execution time for **P2**, which will be `(21 + 3) ms` = `24 ms`.
- For process **P4** it will be the sum of execution times of **P1**, **P2** and **P3**.

The **GANTT chart** above perfectly represents the waiting time for each process.

Completion Time: Time taken for the execution to complete, starting from arrival time.

Turn Around Time: Time taken to complete after arrival. In simple words, it is the difference between the Completion time and the Arrival time.

Turn Around Time = Completion Time – Arrival Time

Waiting Time: Total time the process has to wait before it's execution begins. It is the difference between the Turn Around time and the Burst time of the process.

Waiting Time = Turn Around Time – Burst Time

Ex2:

FCFS (Example)

Process	Duration	Oder	Arrival Time
P1	24	1	0
P2	3	2	0
P3	4	3	0

Gantt Chart :

P1(24) P2(3) P3(4)

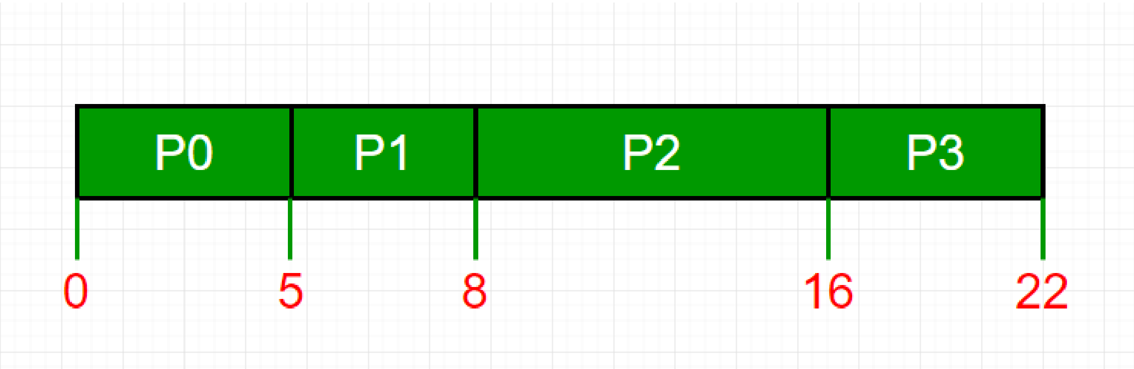
P1 waiting time : 0
P2 waiting time : 24
P3 waiting time : 27

The Average waiting time :

(0+24+27)/3 = 17

Ex3:

Processes	Burst time	Arrival Time	Service Time
P0	5	0	0
P1	3	1	5
P2	8	2	8
P3	6	3	16



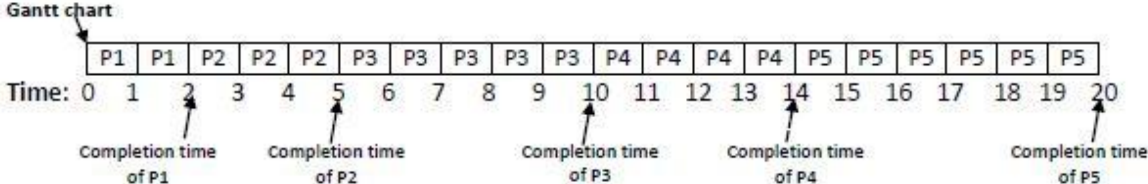

```
1. Process      Wait Time : Service Time - Arrival Time
2.     P0                          0 - 0    = 0
3.     P1                          5 - 1    = 4
4.     P2                          8 - 2    = 6
5.     P3                          16 - 3   = 13
6.
7. Average Wait Time: (0 + 4 + 6 + 13) / 4 = 5.75
```

Service Time : Service time means amount of time after which a process can start execution. It is summation of burst time of previous processes

Q. Consider the following processes with burst time (CPU Execution time). Calculate the average waiting time and average turnaround time?

Process id	Arrival time	Burst time/CPU execution time
P1	0	2
P2	1	3
P3	2	5
P4	3	4
P5	4	6

Sol.



Turnaround time= Completion time – Arrival time

Waiting time= Turnaround time – Burst time

Process id	Arrival time	Burst time	Completion time	Turnaround time	Waiting time
P1	0	2	2	2-0=2	2-2=0
P2	1	3	5	5-1=4	4-3=1
P3	2	5	10	10-2=8	8-5=3
P4	3	4	14	14-3=11	11-4=7
P5	4	6	20	20-4=16	16-6=10

Average turnaround time= $\sum_{i=0}^{n} \text{Turnaround time(i)/n}$ where, n= no. of process

Average waiting time= $\sum_{i=0}^{n} \text{Wating time(i)/n}$ where, n= no. of process

Average turnaround time= 2+4+8+11+16/5 =41/5 =8.2

Average waiting time= 0+1+3+7+10/5 = 21/5 =4.2

Let's take an example of The FCFS scheduling algorithm. In the Following schedule, there are 5 processes with process ID **P0, P1, P2, P3 and P4**. P0 arrives at time 0, P1 at time 1, P2 at time 2, P3 arrives at time 3 and Process P4 arrives at time 4 in the ready queue. The processes and their respective Arrival and Burst time are given in the following table.

The Turnaround time and the waiting time are calculated by using the following formula.

1. Turn Around Time = Completion Time - Arrival Time
2. Waiting Time = Turnaround time - Burst Time

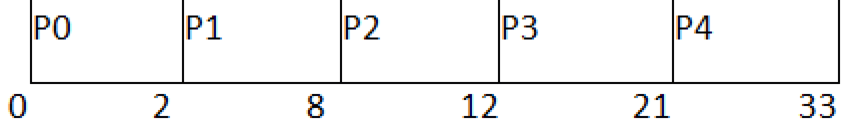
Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting time
0	0	2	2	2	0
1	1	6	8	7	1
2	2	4	12	8	4

3	3	9	21	18	9
4	4	12	33	29	17

The average waiting Time is determined by summing the respective waiting time of all the processes and divided the sum by the total number of processes.

Avg Waiting Time=31/5

(Gantt chart)



Advantages of FCFS

- Simple
- Easy
- First come, First serve

Problems or disadvantages with FCFS Scheduling

Below we have a few shortcomings or problems with the FCFS scheduling algorithm:

1. It is **Non Pre-emptive** algorithm, which means the **process priority** doesn't matter.

   If a process with very least priority is being executed, more like **daily routine backup** process, which takes more time, and all of a sudden some other high priority process arrives, like **interrupt to avoid system crash**, the high priority process will have to wait, and hence in this case, the system will crash, just because of improper process scheduling.

2. Not optimal Average Waiting Time.

3. Resources utilization in parallel is not possible, which leads to **Convoy Effect**, and hence poor resource(CPU, I/O etc) utilization.

What is Convoy Effect?

Convoy Effect is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time.

This essentially leads to poor utilization of resources and hence poor performance.

Shortest Job First(SJF) Scheduling

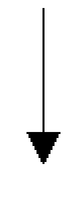
Shortest Job First scheduling works on the process with the shortest **burst time** or **duration** first.

- This is the best approach to minimize waiting time.
- This is used in Batch Systems.
- It is of two types:
    1. Non Pre-emptive
    2. Pre-emptive
- To successfully implement it, the burst time/duration time of the processes should be known to the processor in advance, which is practically not feasible all the time.
- This scheduling algorithm is optimal if all the jobs/processes are available at the same time. (either Arrival time is `0` for all, or Arrival time is same for all)

Non Pre-emptive Shortest Job First

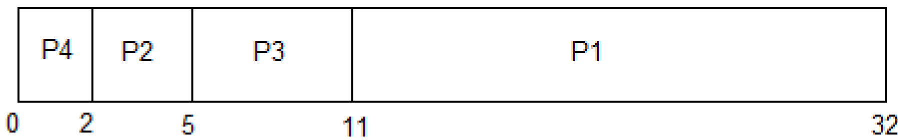
Consider the below processes available in the ready queue for execution, with **arrival time** as `0` for all and given **burst times**.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2



In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT chart will be following :



Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

As you can see in the GANTT chart above, the process P4 will be picked up first as it has the shortest burst time, then **P2**, followed by **P3** and at last **P1**.

We scheduled the same set of processes using the First come first serve algorithm in the previous tutorial, and got average waiting time to be `18.75 ms`, whereas with SJF, the average waiting time comes out `4.5 ms`.

Ex:

As an example, consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:

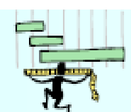
P_4	P_1	P_3	P_2

0 3 9 16 24

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. If we were using the **FCFS** scheduling scheme, then 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. By moving a short process before a long one, the waiting time of the short process decreases more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

Shortest Job First



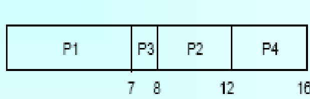
- Associate with each process the length of its next CPU burst.
  - **Non-preemptive:** once CPU is given the process, it cannot be preempted until it completes this CPU burst.
  - **Preemptive:** if a new process arrives with CPU burst length less than the remaining time of the currently executing process then switch them. Also known as **Shortest Remaining Time First**.

Non-preemptive Shortest Job First

- Example:

Process:	p1	p2	p3	p4
Arrival time:	0	2	4	5
Burst time:	7	4	1	4

- Schedule:



Waiting time:	0	6	3	7

Average waiting time: 4.

Preemptive Shortest Job First

- Example:

Process:	p1	p2	p3	p4
Arrival time:	0	2	4	5
Burst time:	7	4	1	4

- Schedule:

P1	P2	P3	P2	P4	P1
2	4	5	7	11	16

Waiting time:	9	1	0	2

Average waiting time: 3.

Problem with Non Pre-emptive SJF

If the **arrival time** for processes are different, which means all the processes are not available in the ready queue at time `0`, and some jobs arrive after some time, in such

situation, sometimes process with short burst time have to wait for the current process's execution to finish, because in Non Pre-emptive SJF, on arrival of a process with short duration, the existing job/process's execution is not halted/stopped to execute the short job first.

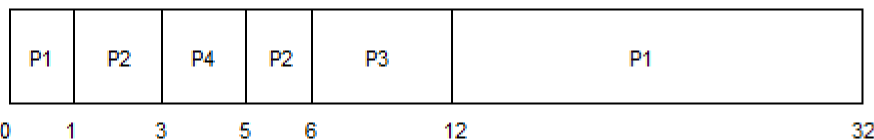
This leads to the problem of **Starvation**, where a shorter process has to wait for a long time until the current longer process gets executed. This happens if shorter jobs keep coming, but this can be solved using the concept of **aging**.

Pre-emptive Shortest Job First

In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with **short burst time** arrives, the existing process is preempted or removed from execution, and the shorter job is executed first.

PROCESS	BURST TIME	ARRIVAL TIME
P1	21	0
P2	3	1
P3	6	2
P4	2	3

The GANTT chart for Preemptive Shortest Job First Scheduling will be,



The average waiting time will be, ((5-3) + (6-2) + (12-1))/4 = 4.25 ms

The average waiting time for preemptive shortest job first scheduling is less than both, non-preemptive SJF scheduling and FCFS scheduling.

As you can see in the **GANTT chart** above, as **P1** arrives first, hence it's execution starts immediately, but just after `1 ms`, process **P2** arrives with a **burst time** of `3 ms`which is less than the burst time of **P1**, hence the process **P1**(1 ms done, 20 ms left) is preemptied and process **P2** is executed.

As **P2** is getting executed, after `1 ms`, **P3** arrives, but it has a burst time greater than that of **P2**, hence execution of **P2** continues. But after another millisecond, **P4** arrives with a burst time of `2 ms`, as a result **P2**(2 ms done, 1 ms left) is preemptied and **P4** is executed.

After the completion of **P4**, process **P2** is picked up and finishes, then **P2** will get executed and at last **P1**.

The Pre-emptive SJF is also known as **Shortest Remaining Time First**, because at any given point of time, the job with the shortest remaining time is executed first.

Shortest Remaining Time First (SRTF) Scheduling Algorithm

This Algorithm is the **preemptive version** of **SJF scheduling**. In SRTF, the execution of the process can be stopped after certain amount of time. At the arrival of every process, the short term scheduler schedules the process with the least remaining burst time among the list of available processes and the running process.

Once all the processes are available in the **ready queue**, No preemption will be done and the algorithm will work as **SJF scheduling**. The context of the process is saved in the **Process Control Block** when the process is removed from the execution and the next process is scheduled. This PCB is accessed on the **next execution** of this process.

Example

In this Example, there are five jobs P1, P2, P3, P4, P5 and P6. Their arrival time and burst time are given below in the table.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time	Response Time
1	0	8	20	20	12	0
2	1	4	10	9	5	1
3	2	2	4	2	0	2
4	3	1	5	2	1	4
5	4	3	13	9	6	10
6	5	2	7	2	0	5

Avg Waiting Time = 24/6

The Gantt chart is prepared according to the arrival and burst time given in the table.

P1	P2	P3	P3	P4	P6	P2	P5	P1	
0	1	2	3	4	5	7	10	13	20

1. Since, at time 0, the only available process is P1 with CPU burst time 8. This is the only available process in the list therefore it is scheduled.
2. The next process arrives at time unit 1. Since the algorithm we are using is SRTF which is a preemptive one, the current execution is stopped and the scheduler checks for the process with the least burst time.
   Till now, there are two processes available in the ready queue. The OS has executed P1 for one unit of time till now; the remaining burst time of P1 is 7 units. The burst time of Process P2 is 4 units. Hence Process P2 is scheduled on the CPU according to the algorithm.
3. The next process P3 arrives at time unit 2. At this time, the execution of process P3 is stopped and the process with the least remaining burst time is searched. Since the process P3 has 2 unit of burst time hence it will be given priority over others.
4. The Next Process P4 arrives at time unit 3. At this arrival, the scheduler will stop the execution of P4 and check which process is having least burst time among the available processes (P1, P2, P3 and P4). P1 and P2 are having the remaining burst time 7 units and 3 units respectively.

   P3 and P4 are having the remaining burst time 1 unit each. Since, both are equal hence the scheduling will be done according to their arrival time. P3 arrives earlier than P4 and therefore it will be scheduled again.
5. The Next Process P5 arrives at time unit 4. Till this time, the Process P3 has completed its execution and it is no more in the list. The scheduler will compare the remaining burst time of all the available processes. Since the burst time of process P4 is 1 which is least among all hence this will be scheduled.
6. The Next Process P6 arrives at time unit 5, till this time, the Process P4 has completed its execution. We have 4 available processes till now, that are P1 (7), P2 (3), P5 (3) and P6 (2). The Burst time of P6 is the least among all hence P6 is scheduled. Since, now, all the processes are available hence the algorithm will now work same as SJF. P6 will be executed till its completion and then the process with the least remaining time will be scheduled.

SRTF GATE 2011 Example

If we talk about scheduling algorithm from the GATE point of view, they generally ask simple numerical questions about finding the average waiting

time and Turnaround Time. Let's discuss the question asked in GATE 2011 on SRTF.

Q. Given the arrival time and burst time of 3 jobs in the table below. Calculate the Average waiting time of the system.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	0	9	13	13	4
2	1	4	5	4	0
3	2	9	22	20	11

There are three jobs P1, P2 and P3. P1 arrives at time unit 0; it will be scheduled first for the time until the next process arrives. P2 arrives at 1 unit of time. Its burst time is 4 units which is least among the jobs in the queue. Hence it will be scheduled next.

At time 2, P3 will arrive with burst time 9. Since remaining burst time of P2 is 3 units which are least among the available jobs. Hence the processor will continue its execution till its completion. Because all the jobs have been arrived so no preemption will be done now and all the jobs will be executed till the completion according to SJF.

Avg Waiting Time = (4+0+11)/3 = 5 units

Priority Scheduling

In Priority scheduling, there is a priority number assigned to each process. In some systems, the lower the number, the higher the priority. While, in the others, the higher the number, the higher will be the priority. The Process with the higher priority among the available processes is given the CPU. There are two types of priority scheduling algorithm exists. One is **Preemptive** priority scheduling while the other is **Non Preemptive** Priority scheduling.

The priority number assigned to each of the process may or may not vary. If the priority number doesn't change itself throughout the process, it is called **static priority**, while if it keeps changing itself at the regular intervals, it is called **dynamic priority**.

Non Preemptive Priority Scheduling

In the Non Preemptive Priority scheduling, The Processes are scheduled according to the priority number assigned to them. Once the process gets scheduled, it will run till the completion. Generally, the lower the priority number, the higher is the priority of the process. The people might get confused with the priority numbers, hence in the GATE, there clearly mention which one is the highest priority and which one is the lowest one.

Example

In the Example, there are 7 processes P1, P2, P3, P4, P5, P6 and P7. Their priorities, Arrival Time and burst time are given in the table.

Process ID	Priority	Arrival Time	Burst Time
1	2	0	3
2	6	2	5
3	3	1	4
4	5	4	2
5	7	6	9
6	4	5	4
7	10	7	10

We can prepare the Gantt chart according to the Non Preemptive priority scheduling.

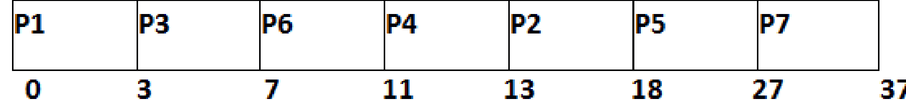
The Process P1 arrives at time 0 with the burst time of 3 units and the priority number 2. Since No other process has arrived till now hence the OS will schedule it immediately.

Meanwhile the execution of P1, two more Processes P2 and P3 are arrived. Since the priority of P3 is 3 hence the CPU will execute P3 over P2.

Meanwhile the execution of P3, All the processes get available in the ready queue. The Process with the lowest priority number will be given the priority. Since P6 has priority number assigned as 4 hence it will be executed just after P3.

After P6, P4 has the least priority number among the available processes; it will get executed for the whole burst time.

Since all the jobs are available in the ready queue hence All the Jobs will get executed according to their priorities. If two jobs have similar priority number assigned to them, the one with the least arrival time will be executed.



From the GANTT Chart prepared, we can determine the completion time of every process. The turnaround time, waiting time and response time will be determined.

1. Turn Around Time = Completion Time - Arrival Time
2. Waiting Time = Turn Around Time - Burst Time

Process Id	Priority	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time	Respon se Time
1	2	0	3	3	3	0	0
2	6	2	5	18	16	11	13
3	3	1	4	7	6	2	3
4	5	4	2	13	9	7	11
5	7	6	9	27	21	12	18
6	4	5	4	11	6	2	7
7	10	7	10	37	30	20	27

Avg Waiting Time = (0+11+2+7+12+2+18)/7 = 54/7 units

Preemptive Priority Scheduling

In Preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The One with the highest priority among all the available processes will be given the CPU next.

The difference between preemptive priority scheduling and non preemptive priority scheduling is that, in the preemptive priority scheduling, the job which is being executed can be stopped at the arrival of a higher priority job.

Once all the jobs get available in the ready queue, the algorithm will behave as non-preemptive priority scheduling, which means the job scheduled will run till the completion and no preemption will be done.

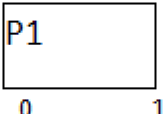
Example

There are 7 processes P1, P2, P3, P4, P5, P6 and P7 given. Their respective priorities, Arrival Times and Burst times are given in the table below.

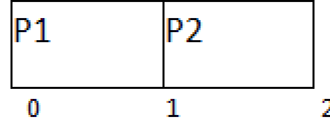
Process Id	Priority	Arrival Time	Burst Time
1	2(L)	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10(H)	5	15
7	9	15	8

GANTT chart Preparation

At time 0, P1 arrives with the burst time of 1 units and priority 2. Since no other process is available hence this will be scheduled till next job arrives or its completion (whichever is lesser).



At time 1, P2 arrives. P1 has completed its execution and no other process is available at this time hence the Operating system has to schedule it regardless of the priority assigned to it.



The Next process P3 arrives at time unit 2, the priority of P3 is higher to P2. Hence the execution of P2 will be stopped and P3 will be scheduled on the cpu .

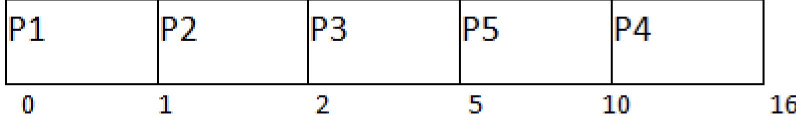
P1	P2	P3

0 1 2 5

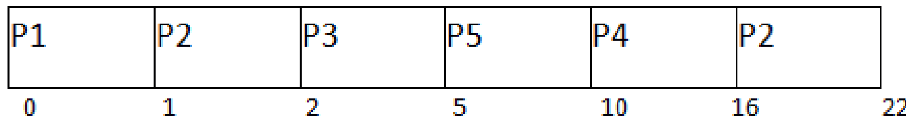
During the execution of P3, three more processes P4, P5 and P6 becomes available. Since, all these three have the priority lower to the process in execution so PS can't preempt the process. P3 will complete its execution and then P5 will be scheduled with the priority highest among the available processes.

P1	P2	P3	P5	
0	1	2	5	10

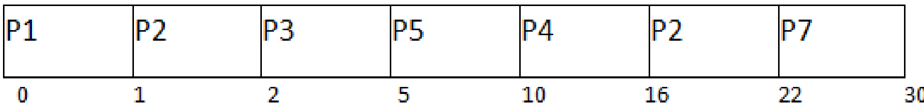
Meanwhile the execution of P5, all the processes got available in the ready queue. At this point, the algorithm will start behaving as Non Preemptive Priority Scheduling. Hence now, once all the processes get available in the ready queue, the OS just took the process with the highest priority and execute that process till completion. In this case, P4 will be scheduled and will be executed till the completion.



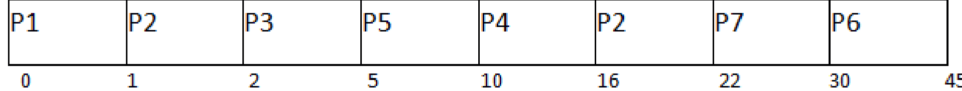
Since P4 is completed, the other process with the highest priority available in the ready queue is P2. Hence P2 will be scheduled next.



P2 is given the CPU till the completion. Since its remaining burst time is 6 units hence P7 will be scheduled after this.



The only remaining process is P6 with the least priority, the Operating System has no choice unless of executing it. This will be executed at the last.



The Completion Time of each process is determined with the help of GANTT chart. The turnaround time and the waiting time can be calculated by the following formula.

1. Turnaround Time = Completion Time - Arrival Time
2. Waiting Time = Turn Around Time - Burst Time

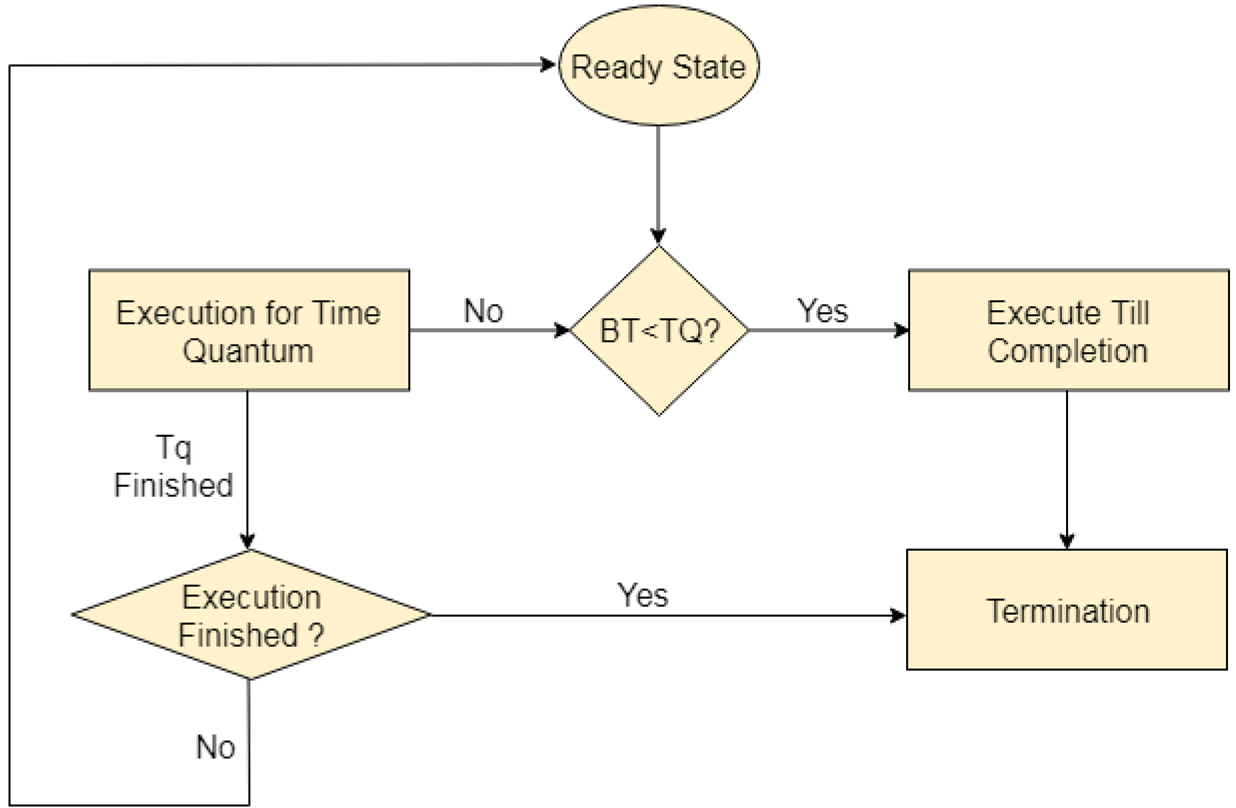
Process Id	Priority	Arrival Time	Burst Time	Completion Time	Turn around Time	Waiting Time
1	2	0	1	1	1	0
2	6	1	7	22	21	14
3	3	2	3	5	3	0
4	5	3	6	16	13	7
5	4	4	5	10	6	1
6	10	5	15	45	40	25
7	9	6	8	30	24	16

Avg Waiting Time = (0+14+0+7+1+25+16)/7 = 63/7 = 9 units

Round Robin Scheduling Algorithm

Round Robin scheduling algorithm is one of the most popular scheduling algorithm which can actually be implemented in most of the operating systems. This is the **preemptive version** of first come first serve scheduling. The Algorithm focuses on Time Sharing. In this algorithm, every process gets

executed in a **cyclic way**. A certain time slice is defined in the system which is called time **quantum**. Each process present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will **terminate** else the process will go back to the **ready queue** and waits for the next turn to complete the execution.



Advantages

1. It can be actually implementable in the system because it is not depending on the burst time.
2. It doesn't suffer from the problem of starvation or convoy effect.
3. All the jobs get a fare allocation of CPU.

Disadvantages

1. The higher the time quantum, the higher the response time in the system.
2. The lower the time quantum, the higher the context switching overhead in the system.
3. Deciding a perfect time quantum is really a very difficult task in the system.

RR Scheduling Example

In the following example, there are six processes named as P1, P2, P3, P4, P5 and P6. Their arrival time and burst time are given below in the table. The time quantum of the system is 4 units.

Process ID	Arrival Time	Burst Time
1	0	5
2	1	6
3	2	3
4	3	1
5	4	5
6	6	4

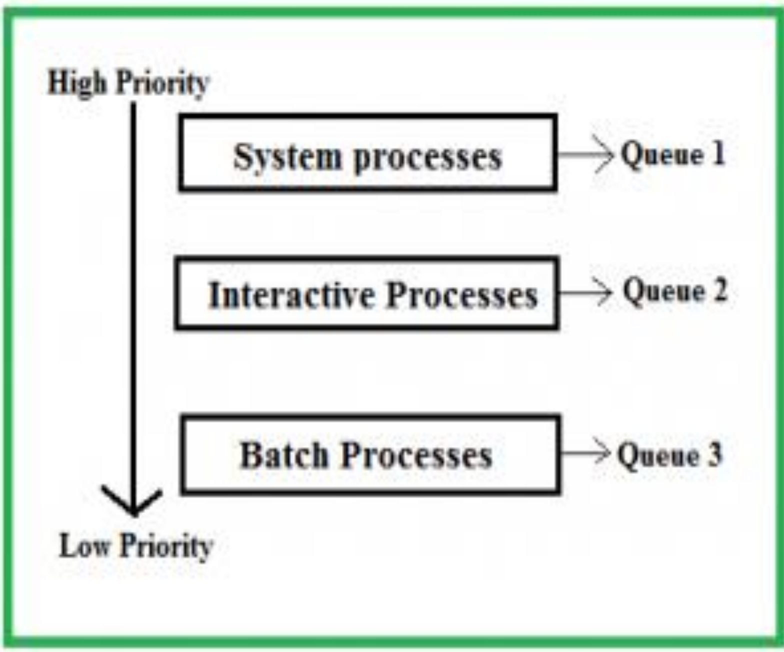
According to the algorithm, we have to maintain the ready queue and the Gantt chart. The structure of both the data structures will be changed after every scheduling.

Note: Refer notebook

Multilevel Queue (MLQ) CPU Scheduling

It may happen that processes in the ready queue can be divided into different classes where each class has its own scheduling needs. For example, a common division is a **foreground (interactive)** process and **background (batch)** processes.These two classes have different scheduling needs. For this kind of situation Multilevel Queue Scheduling is used.Now, let us see how it works.

Ready Queue is divided into separate queues for each class of processes. For example, let us take three different types of process System processes, Interactive processes and Batch Processes. All three process have there own queue. Now,look at the below figure.



All three different type of processes have there own queue. Each queue have its own Scheduling algorithm. For example, queue 1 and queue 2 uses **Round Robin** while queue 3 can use **FCFS** to schedule there processes.

Scheduling among the queues : What will happen if all the queues have some processes? Which process should get the cpu? To determine this Scheduling among the queues is necessary. There are two ways to do so –

1. **Fixed priority preemptive scheduling method** – Each queue has absolute priority over lower priority queue. Let us consider following priority order **queue 1 > queue 2 > queue 3**.According to this algorithm no process in the batch queue(queue 3) can run unless queue 1 and 2 are empty. If any batch process (queue 3) is running and any system (queue 1) or Interactive process(queue 2) entered the ready queue the batch process is preempted.
2. **Time slicing** – In this method each queue gets certain portion of CPU time and can use it to schedule its own processes.For instance, queue 1 takes 50 percent of CPU time queue 2 takes 30 percent and queue 3 gets 20 percent of CPU time.

Example Problem :
Consider below table of four processes under Multilevel queue scheduling.Queue number denotes the queue of the process.

Process	Arrival Time	CPU Burst Time	Queue Number
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

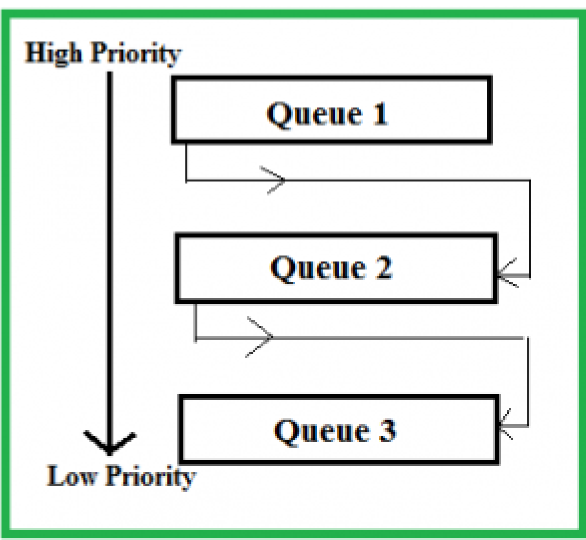
Priority of queue 1 is greater than queue 2. queue 1 uses Round Robin (Time Quantum = 2) and queue 2 uses FCFS.

Below is the **gantt chart** of the problem :

At starting both queues have process so process in queue 1 (P1, P2) runs first (because of higher priority) in the round robin fashion and completes after 7 units then process in queue 2 (P3) starts running (as there is no process in queue 1) but while it is running P4 comes in queue 1 and interrupts P3 and start running for 5 second and after its completion P3 takes the CPU and completes its execution.

Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling

his Scheduling is like Multilevel Queue(MLQ) Scheduling but in this process can move between the queues. **Multilevel Feedback Queue Scheduling (MLFQ)** keep analyzing the behavior (time of execution) of processes and according to which it changes its priority.Now, look at the diagram and explanation below to understand it properly.



Well, above implementation may differ for example the last queue can also follow Round-robin Scheduling.

Problems in the above implementation – A process in the lower priority queue can suffer from starvation due to some short processes taking all the CPU time.
Solution – A simple solution can be to boost the priority of all the process after regular intervals and place them all in the highest priority queue.
What is the need of such complex Scheduling?

- Firstly, it is more flexible than the multilevel queue scheduling.

- To optimize turnaround time algorithms like SJF is needed which require the running time of processes to schedule them. But the running time of the process is not known in advance. MFQS runs a process for a time quantum and then it can change its priority(if it is a long process). Thus it learns from past behavior of the process and then predicts its future behavior.This way it tries to run shorter process first thus optimizing turnaround time.
- MFQS also reduces the response time.
  - **Example** –
    Consider a system which has a CPU bound process, which requires the burst time of 40 seconds.The multilevel Feed Back Queue scheduling algorithm is used and the queue time quantum '2' seconds and in each level it is incremented by '5' seconds.Then how many times the process will be interrupted and on which queue the process will terminate the execution?
  - **Solution** –
    Process P needs 40 Seconds for total execution.
    At Queue 1 it is executed for 2 seconds and then interrupted and shifted to queue 2.
    At Queue 2 it is executed for 7 seconds and then interrupted and shifted to queue 3.
    At Queue 3 it is executed for 12 seconds and then interrupted and shifted to queue 4.
    At Queue 4 it is executed for 17 seconds and then interrupted and shifted to queue 5.
    At Queue 5 it executes for 2 seconds and then it completes.
    Hence the process is interrupted 4 times and completes on queue 5.

Multiple-Processor Scheduling in Operating System

In multiple-processor scheduling **multiple CPU's** are available and hence **Load Sharing** becomes possible. However multiple processor scheduling is more **complex** as compared to single processor scheduling. In multiple processor scheduling there are cases when the processors are identical i.e. HOMOGENEOUS, in terms of their functionality, we can use any processor available to run any process in the queue.

Approaches to Multiple-Processor Scheduling –

One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the **Master Server** and the other processors executes only the **user code**. This is simple and reduces the need of data sharing. This entire scenario is called **Asymmetric Multiprocessing**.
A second approach uses **Symmetric Multiprocessing** where each processor is **self scheduling**. All processes may be in a common ready queue or each processor may have its own private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

Processor Affinity –

Processor Affinity means a processes has an **affinity** for the processor on which it is currently running.
When a process runs on a specific processor there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor and as a result successive memory access by the process are often satisfied in the cache memory. Now if the process migrates to another processor, the contents of the cache memory must be invalidated for

the first processor and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most of the SMP(symmetric multiprocessing) systems try to avoid migration of processes from one processor to another and try to keep a process running on the same processor. This is known as **PROCESSOR AFFINITY**.
There are two types of processor affinity:

1. **Soft Affinity –** When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity –** Some systems such as Linux also provide some system calls that support Hard Affinity which allows a process to migrate between processors.

Load Balancing –

Load Balancing is the **phenomena** which keeps the **workload** evenly **distributed** across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of process which are eligible to execute. Load balancing is unnecessary because once a processor becomes idle it immediately extracts a runnable process from the common run queue. On SMP(symmetric multiprocessing), it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor else one or more processor will sit idle while other processors have high workloads along with lists of processors awaiting the CPU.
There are two general approaches to load balancing :

1. **Push Migration –** In push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes load on each processors by moving the processes from overloaded to idle or less busy processors.
2. **Pull Migration –** Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Multicore Processors –

In multicore processors **multiple processor** cores are places on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor. **SMP systems** that use multicore processors are faster and consume **less power** than systems in which each processor has its own physical chip.
However multicore processors may **complicate** the scheduling problems. When processor accesses memory then it spends a significant amount of time waiting for the data to become available. This situation is called **MEMORY STALL**. It occurs for various reasons such as cache miss, which is accessing the data that is not in the cache memory. In such cases the processor can spend upto fifty percent of its time waiting for data to become available from the memory. To solve this problem recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, core can switch to another thread.
There are two ways to multithread a processor :

1. **Coarse-Grained Multithreading –** In coarse grained multithreading a thread executes on a processor until a long latency event such as a memory stall occurs, because of the delay caused by the long latency event, the processor must switch to another thread to begin execution. The cost of switching between threads is high as the instruction pipeline must be

terminated before the other thread can begin execution on the processor core. Once this new thread begins execution it begins filling the pipeline with its instructions.

2. **Fine-Grained Multithreading** – This multithreading switches between threads at a much finer level mainly at the boundary of an instruction cycle. The architectural design of fine grained systems include logic for thread switching and as a result the cost of switching between threads is small.

Virtualization and Threading –

In this type of **multiple-processor** scheduling even a single CPU system acts like a multiple-processor system. In a system with Virtualization, the virtualization presents one or more virtual CPU's to each of virtual machines running on the system and then schedules the use of physical CPU'S among the virtual machines. Most virtualized environments have one host operating system and many guest operating systems. The host operating system creates and manages the virtual machines and each virtual machine has a guest operating system installed and applications running within that guest.Each guest operating system may be assigned for specific use cases,applications, and users,including time sharing or even real-time operation. Any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will be negatively impacted by the virtualization. In a time sharing operating system that tries to allot 100 milliseconds to each time slice to give users a reasonable response time. A given 100 millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more which results in a very poor response time for users logged into that virtual machine. The net effect of such scheduling layering is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all cycles and that they are scheduling all of those cycles.Commonly, the time-of-day clocks in virtual machines are incorrect because timers take no longer to trigger than they would on dedicated CPU's.

Evaluation of Process Scheduling Algorithms

In the section above we looked at various scheduling algorithms. But how do we decide which one to use?

The first thing we need to decide is how we will evaluate the algorithms. To do this we need to decide on the relative importance of the factors we listed above (Fairness, Efficiency, Response Times, Turnaround and Throughput). Only once we have decided on our evaluation method can we carry out the evaluation.

Deterministic Modeling

This evaluation method takes a predetermined workload and evaluates each algorithm using that workload.

Assume we are presented with the following processes, which all arrive at time zero.

Process	Burst Time
P1	9
P2	33
P3	2
P4	5
P5	14

Which of the following algorithms will perform best on this workload?

First Come First Served (FCFS), Non Preemptive Shortest Job First (SJF) and Round Robin (RR). Assume a quantum of 8 milliseconds.

Before looking at the answers, try to calculate the figures for each algorithm.

The advantages of deterministic modeling is that it is exact and fast to compute. The disadvantage is that it is only applicable to the workload that you use to test. As an example, use the above workload but assume P1 only has a burst time of 8 milliseconds. What does this do to the average waiting time?

Of course, the workload might be typical and scale up but generally deterministic modeling is too specific and requires too much knowledge about the workload.

Queuing Models

Another method of evaluating scheduling algorithms is to use queuing theory. Using data from real processes we can arrive at a probability distribution for the length of a burst time and the I/O times for a process. We can now generate these times with a certain distribution.

We can also generate arrival times for processes (arrival time distribution).

If we define a queue for the CPU and a queue for each I/O device we can test the various scheduling algorithms using queuing theory.

Knowing the arrival rates and the service rates we can calculate various figures such as average queue length, average wait time, CPU utilization etc.

One useful formula is ***Little's Formula.***

$n = \lambda w$

Where

n is the average queue length
λ is the average arrival rate for new processes (e.g. five a second)
w is the average waiting time in the queue

Knowing two of these values we can, obviously, calculate the third. For example, if we know that eight processes arrive every second and there are normally sixteen processes in the queue we can compute that the average waiting time per process is two seconds.

The main disadvantage of using queuing models is that it is not always easy to define realistic distribution times and we have to make assumptions. This results in the model only being an approximation of what actually happens.

Simulations

Rather than using queuing models we simulate a computer. A Variable, representing a clock is incremented. At each increment the state of the simulation is updated.

Statistics are gathered at each clock tick so that the system performance can be analysed.

The data to drive the simulation can be generated in the same way as the queuing model, although this leads to similar problems.

Alternatively, we can use trace data. This is data collected from real processes on real machines and is fed into the simulation. This can often provide good results and good comparisons over a range of scheduling algorithms.

However, simulations can take a long time to run, can take a long time to implement and the trace data may be difficult to collect and require large amounts of storage.

Implementation

The best way to compare algorithms is to implement them on real machines. This will give the best results but does have a number of disadvantages.

· It is expensive as the algorithm has to be written and then implemented on real hardware.
· If typical workloads are to be monitored, the scheduling algorithm must be used in a live situation. Users may not be happy with an environment that is constantly changing.
· If we find a scheduling algorithm that performs well there is no guarantee that this state will continue if the workload or environment changes.

Algorithm Evaluation

How do we select a CPU scheduling algorithm for a particular system?

There are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult. The first problem is defining the criteria to be used in selecting an algorithm. Criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these measures. Our criteria may include several measures, such as:

- Maximizing CPU utilization under the constraint that the maximum response time is 1 second
- Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time Once the selection criteria have been defined, we want to evaluate the algorithms under consideration.We next describe the various evaluation methods we can use.

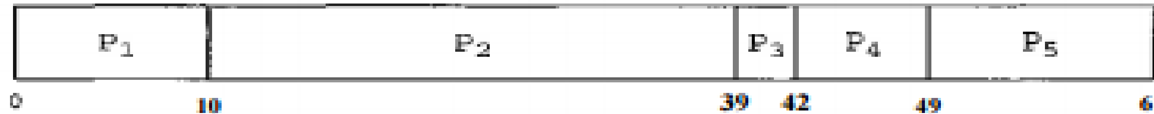
Deterministic Modeling

One major class of evaluation methods is analytic evaluation. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload. One type of analytic evaluation is deterministic modeling. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload. For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	10
P_2	29
P_3	3
Pi	7
P_5	12

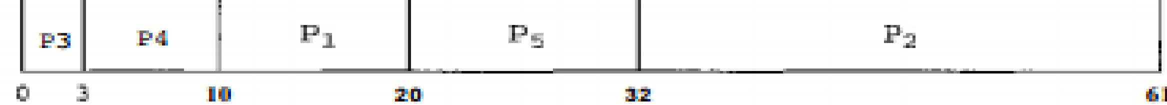
Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

For the FCFS algorithm, we would execute the processes as



The waiting time is 0 milliseconds for process P_1, 10 milliseconds for process P_2, 39 milliseconds for process P3, 42 milliseconds for process P4, and 49 milliseconds for process P_5. Thus, the average waiting time is (0 + 10 + 39 + 42 + 49)/5= 28 milliseconds.

With nonpreemptive SJF scheduling, we execute the processes as



The waiting time is 10 milliseconds for process P_1, 32 milliseconds for process P_2, 0 milliseconds for process P3, 3 milliseconds for process P4, and 20 milliseconds for process P5. Thus, the average waiting time is (10 + 32 + 0 + 3 + 20)/5 = 13 milliseconds.

With the RR algorithm, we execute the processes as

The waiting time is 0 milliseconds for process P_1, 32 milliseconds for process P_2, 20 milliseconds for process P_3, 23 milliseconds for process P4, and 40 milliseconds for process P5. Thus, the average waiting time is (0 + 32 + 20 + 23 + 40)/5 = 23 milliseconds.

We see that, *in this case*, the average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.

Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms. However, it requires exact numbers for input, and its answers apply only to those cases. The main uses of deterministic modeling are in describing scheduling algorithms and providing examples.

In cases where we are running the same program over and over again and can measure the program's processing

requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm. Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately.

For example, it can be shown that, for the environment described (all processes and their times available at time 0), the SJF policy will always result in the minimum waiting time.

Queueing Models

On many systems, the processes that are run vary from day to day, so there is no static set of processes (or times) to use for deterministic modeling. What can be determined, however, is the distribution of CPU and I/O bursts. These distributions can be measured and then approximated or simply estimated. The result is a mathematical formula describing the probability of a particular CPU burst. Commonly, this distribution is exponential and is described by its mean. Similarly, we can describe the distribution of times when processes arrive in the system (the arrival-time distribution). From these two distributions, it is possible to compute the average throughput, utilization, waiting time, and so on for most algorithms. The computer system is described as a network of servers.

Each server has a queue of waiting processes. The CPU is a server with its ready queue, as is the I/O system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called queueing-network analysis. As an example, let n be the average queue length (excluding the process being serviced), let W be the average waiting time in the queue, and let X be the average arrival rate for new processes in the queue (such as three processes per second).

We expect that during the time W that a process waits, \ x W new processes will arrive in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus, This equation, known as Little's formula, is particularly useful

because it is valid for any scheduling algorithm and arrival distribution. We can use Little's formula to compute one of the three variables, if we know the other two.

For example, if we know that 7 processes arrive every second (on average), and that there are normally 14 processes in the queue, then we can compute the average waiting time per process as 2 seconds. Queueing analysis can be useful in comparing scheduling algorithms, but it also has limitations. At the moment, the classes of algorithms and distributions that can be handled are fairly limited.

The mathematics of complicated algorithms and distributions can be difficult to work with. Thus, arrival and service distributions are often defined in mathematically tractable —but unrealistic—ways. It is also generally necessary to make a number of independent assumptions, which may not be accurate. As a result of these difficulties, queueing models are often only approximations of real systems, and the accuracy of the computed results may be questionable.

Simulations

To get a more accurate evaluation of scheduling algorithms, we can use simulations. Running simulations involves programming a model of the computer system. Software data structures represent the major components of the system. The simulator has a variable representing a clock; as this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed. The data to drive the simulation can be generated in several ways. The most common method uses a random-number generator, which is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions.

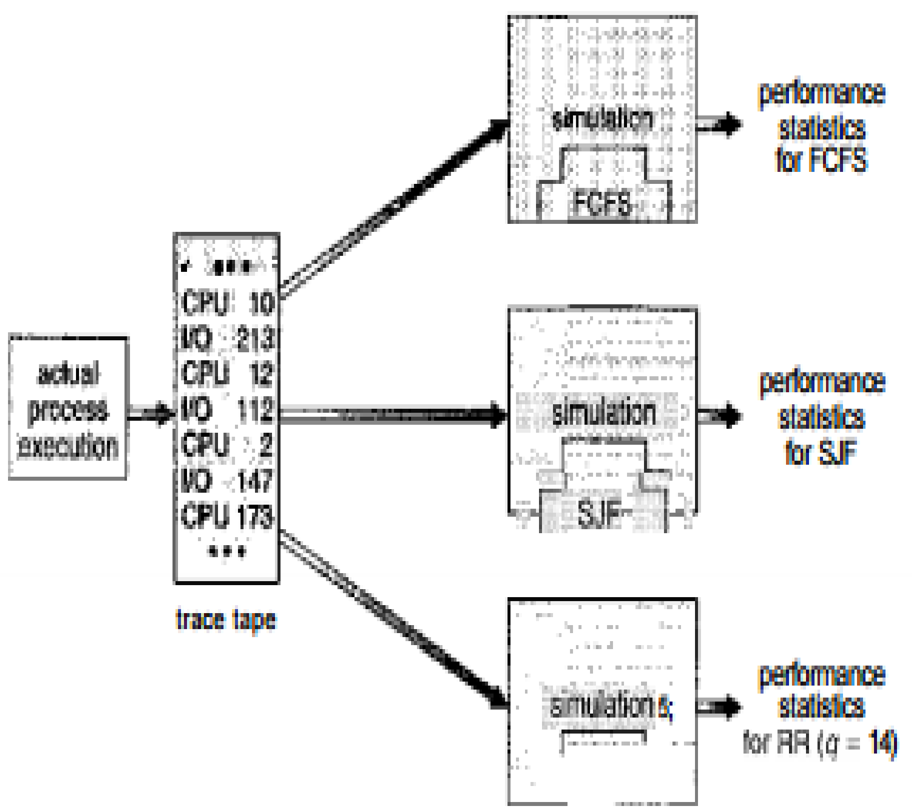


Figure 5.15 Evaluation of CPU schedulers by simulation.

The distributions can be defined mathematically (uniform, exponential, Poisson) or empirically. If a distribution is to be defined empirically, measurements of the actual system under study are taken. The results define the distribution of events in the real system; this distribution can then be used to drive the simulation. A distribution-driven simulation may be inaccurate, however, because of relationships between successive events in the real system. The frequency distribution indicates only how many instances of each event occur; it does not indicate anything about the order of their occurrence.

To correct this problem, we can use trace tapes. We create a trace tape by monitoring the real system and recording the sequence of actual events (Figure 5.15). We then use this sequence to drive the simulation. Trace tapes provide an

excellent way to compare two algorithms on exactly the same set of real inputs. This method can produce accurate results for its inputs.

Simulations can be expensive, often requiring hours of computer time. A more detailed simulation provides more accurate results, but it also requires more computer time. In addition, trace tapes can require large amounts of storage space. Finally, the design, coding, and debugging of the simulator can be a major task.

Implementation Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions. The major difficulty with this approach is the high cost.

The expense is incurred not only in coding the algorithm and modifying the operating system to support it (along with its required data structures) but also in the reaction of the users to a constantly changing operating system. Most users are not interested in building a better operating system; they merely want to get their processes executed and use their results. A constantly changing operating system does not help the users to get their work done. Another difficulty is that the environment in which the algorithm is used will change.

The environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler. If short processes are given priority, then users may break larger processes into sets of smaller processes. If interactive processes are given priority over noninteractive processes, then users may switch to interactive use. For example, researchers designed one system that classified interactive and noninteractive processes automatically by looking at the amount of terminal I/O.

If a process did not input or output to the terminal in a 1-second interval, the process was classified as noninteractive and was moved to a lower-priority queue. In response to this policy, one programmer modified his programs to write an arbitrary character to the terminal at regular intervals of less than 1 second. The system gave his programs a high priority, even though the terminal output was completely meaningless. The most flexible scheduling algorithms are those that can be altered by the system managers

or by the users so that they can be tuned for a specific application or set of applications. For instance, a workstation that performs high-end graphical applications may have scheduling needs different from those of a web server or file server.

Some operating systems— particularly several versions of UNIX—allow the system manager to fine-tune the scheduling parameters for a particular system configuration. For example, Solaris provides the dispadmin command to allow the system administrator to modify the parameters of the scheduling classes . Another approach is to use APIs that modify the priority of a process or thread. The Java, /POSIX, and /WinAPI/ provide such functions. The downfall of this approach is that performance tuning a system or application most often does not result in improved performance in more general situations.