Process Scheduling

- The two main objectives of the process scheduling system are to keep the CPU busy at all times and to deliver "acceptable" response times for all programs, particularly for interactive ones.
- The process scheduler must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU.

Scheduling Queues

- All processes are stored in the **job queue.**
- Processes in the Ready state are placed in the **ready queue.**
- Processes waiting for a device to become available or to deliver data are placed in **device queues**. There is generally a separate device queue for each device.
- Other queues may also be created and used as needed.

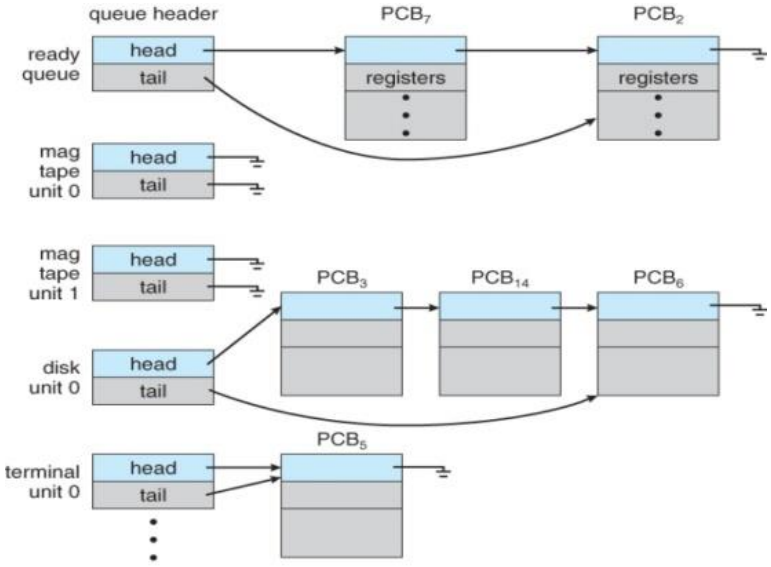


Figure 3.5 - The ready queue and various I/O device queues

Schedulers

- A **long-term scheduler** is typical of a batch system or a very heavily loaded system. It runs infrequently, (such as when one process ends selecting one more to be loaded in from disk in its place), and can afford to take the time to implement intelligent and advanced scheduling algorithms.
- The **short-term scheduler**, or CPU Scheduler, runs very frequently, on the order of 100 milliseconds, and must very quickly swap one process out of the CPU and swap in another one.
- Some systems also employ a **medium-term scheduler**. When system loads get high, this scheduler will swap one or more processes out of the ready queue system for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system. See the differences in Figures 3.7 and 3.8 below.
- An efficient scheduling system will select a good **process mix** of **CPU-bound** processes and **I/O bound** processes.

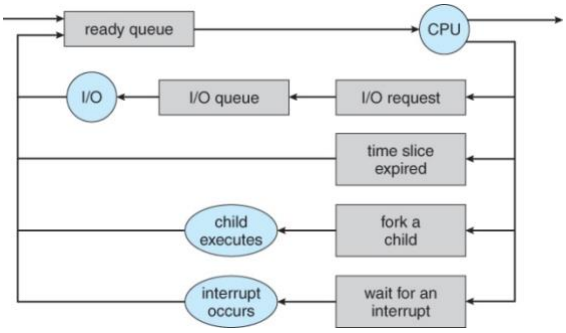


Figure 3.6 - Queueing-diagram representation of process scheduling

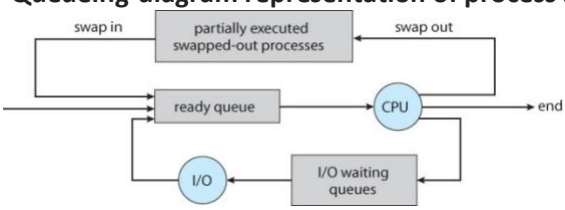


Figure 3.7 - Addition of a medium-term scheduling to the queueing diagram

Context Switch

- Whenever an interrupt arrives, the CPU must do a **state-save** of the currently running process, then switch into kernel mode to handle the interrupt, and then do a **state-restore** of the interrupted process.
- Similarly, a **context switch** occurs when the time slice for one process has expired and a new process is to be loaded from the ready queue. This will be instigated by a timer interrupt, which will then cause the current process's state to be saved and the new process's state to be restored.
- Context switching happens VERY frequently, and the overhead of doing the switching is just lost CPU time, so context switches (state saves & restores) need to be as fast as possible. Some hardware has special provisions for speeding this up, such as a single machine instruction for saving or restoring all registers at once.

Operations on Processes

Process Creation

Processes may create other processes through appropriate system calls, such as **fork** or **spawn**. The process which does the creating is termed the **parent** of the other process, which is termed its **child**.

Each process is given an integer identifier, termed its **process identifier**, or PID. The parent PID (PPID) is also stored for each process.

There are two options for the parent process after creating the child:

1. Wait for the child process to terminate before proceeding. The parent makes a wait( ) system call, for either a specific child or for any child, which causes the parent process to block until the wait( ) returns. UNIX shells normally wait for their children to complete before issuing a new prompt.
2. Run concurrently with the child, continuing to process without waiting.

Two possibilities for the address space of the child relative to the parent:

1. The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behavior of the **fork** system call in UNIX.
2. The child process may have a new program loaded into its address space, with all new code and data segments.

Figures 3.10 and 3.11 below shows the fork and exec process on a UNIX system. Note that the **fork** system call returns the PID of the processes child to each process - It returns a zero to the child process and a non-zero child PID to the parent, so the return value indicates which process is which. Process IDs can be looked up any time for the current process or its direct parent using the getpid( ) and getppid( ) system calls respectively.

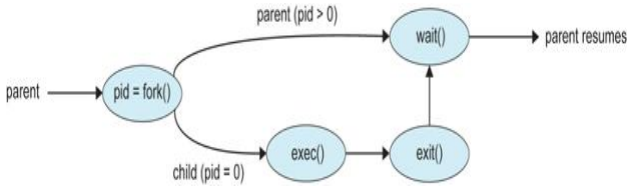


Figure 3.10 - Process creation using the fork( ) system call

Process Termination

- Processes may request their own termination by making the **exit( )** system call, typically returning an int. This int is passed along to the parent if it is doing a **wait( )**, and is typically zero on successful completion and some non-zero code in the event of problems.
- Processes may also be terminated by the system for a variety of reasons, including:
- The inability of the system to deliver necessary system resources.
- In response to a KILL command, or other un handled process interrupt.
- A parent may kill its children if the task assigned to them is no longer needed.
- If the parent exits, the system may or may not allow the child to continue without a parent. (On UNIX systems, orphaned processes are generally inherited by init, which then proceeds to kill them. The UNIX ***nohup*** command allows a child to continue executing after its parent has exited.)

When a process terminates, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process becomes an orphan. (Processes which are trying to terminate but which cannot because their parent is not waiting for them are termed **zombies**. These are eventually inherited by init as orphans and killed off. Note that modern UNIX shells do not produce as many orphans and zombies as older systems used to.)

Interprocess Communication

Independent Processes operating concurrently on a systems are those that can neither affect other processes or be affected by other processes.

Cooperating Processes are those that can affect or be affected by other processes. There are several reasons why cooperating processes are allowed:

1. Information Sharing - There may be several processes which need access to the same file for example. (e.g. pipelines.)
2. Computation speedup - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously (particularly when multiple processors are involved.)
3. Modularity - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)
4. Convenience - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.

Cooperating processes require some type of inter-process communication, which is most commonly one of two types: Shared Memory systems or Message Passing systems. Figure 3.13 illustrates the difference between the two systems:

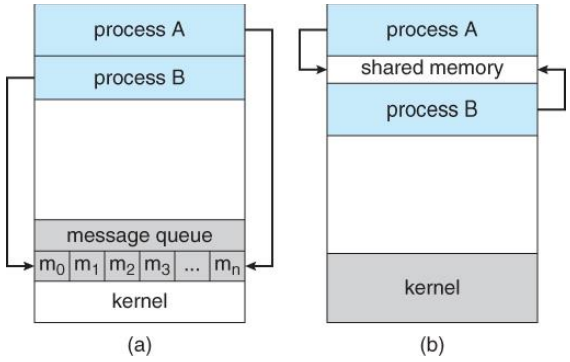


Figure 3.12 - Communications models: (a) Message passing. (b) Shared memory.

1. Shared Memory is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. However it is more complicated to set up, and doesn't work as well across multiple computers. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.
2. Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved.

Shared-Memory Systems

- In general the memory to be shared in a shared-memory system is initially within the address space of a particular process, which needs to make system calls in order to make that memory publicly available to one or more other processes.
- Other processes which wish to use the shared memory must then make their own system calls to attach the shared memory area onto their address space.
- Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

Producer-Consumer Example Using Shared Memory

- This is a classic example, in which one process is producing data and another process is consuming the data. (In this example in the order in which it is produced, although that could vary.)
- The data is passed via an intermediary buffer, which may be either unbounded or bounded. With a bounded buffer the producer may have to wait until there is space available in the buffer, but with an unbounded buffer the producer will never need to wait. The consumer may need to wait in either case until there is data available.
- This example uses shared memory and a circular queue. Note in the code below that only the producer changes "in", and only the consumer changes "out", and that they can never be accessing the same array location at the same time.

First the following data is set up in the shared memory area:

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

- Then the producer process. Note that the buffer is full when "in" is one less than "out" in a circular sense:

```
// Code from Figure 3.13
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;
}
```

- Then the consumer process. Note that the buffer is empty when "in" is equal to "out":

```
// Code from Figure 3.14
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 ) */

}
```

Message-Passing Systems

Message passing systems must support at a minimum system calls for "send message" and "receive message".

A communication link must be established between the cooperating processes before messages can be sent.

There are three key issues to be resolved in message passing systems as further explored in the next three subsections:

1. Direct or indirect communication (naming)
2. Synchronous or asynchronous communication
3. Automatic or explicit buffering.

Naming

- With **direct communication** the sender must know the name of the receiver to which it wishes to send a message.
- There is a one-to-one link between every sender-receiver pair.
- For **symmetric** communication, the receiver must also know the specific name of the sender from which it wishes to receive messages.
  For **asymmetric** communications, this is not necessary.
- **Indirect communication** uses shared mailboxes, or ports.
- Multiple processes can share the same mailbox or boxes.
- Only one process can read any given message in a mailbox. Initially the process that creates the mailbox is the owner, and is the only one allowed to read mail in the mailbox, although this privilege may be transferred.
- (Of course the process that reads the message can immediately turn around and place an identical message back in the box for someone else to read, but that may put it at the back end of a queue of messages.)
- The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.

Synchronization

Either the sending or receiving of messages (or neither or both) may be either **blocking** or **non-blocking**.

Buffering

Messages are passed via queues, which may have one of three capacity configurations:

Zero capacity - Messages cannot be stored in the queue, so senders must block until receivers accept the messages.

Bounded capacity- There is a certain pre-determined finite capacity in the queue. Senders must block if the queue is full, until space becomes available in the queue, but may be either blocking or non-blocking otherwise.

Unbounded capacity - The queue has a theoretical infinite capacity, so senders are never forced to block.

Communication in Client-Server Systems

Sockets

- A **socket** is an endpoint for communication.
- Two processes communicating over a network often use a pair of connected sockets as a communication channel. Software that is designed for client-server operation may also use sockets for communication between two processes running on the same computer - For example the UI for a database program may communicate with the back-end database manager using sockets. (If the program were developed this way from the beginning, it makes it very easy to port it from a single-computer system to a networked application.)
- A socket is identified by an IP address concatenated with a port number, e.g. 200.100.50.5:80.

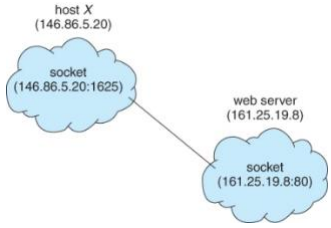


Figure 3.20 - Communication using sockets

- Port numbers below 1024 are considered to be *well-known*, and are generally reserved for common Internet services. For example, telnet servers listen to port 23, ftp servers to port 21, and web servers to port 80.
- General purpose user sockets are assigned unused ports over 1024 by the operating system in response to system calls such as socket( ) or soctkepair( ).

Communication channels via sockets may be of one of two major forms:

1. **Connection-oriented (TCP, Transmission Control Protocol)** connections emulate a telephone connection. All packets sent down the connection are guaranteed to arrive in good condition at the other end, and to be delivered to the receiving process in the order in which they were sent. The TCP layer of the network protocol takes steps to verify all packets sent, re-send packets if necessary, and arrange the received packets in the proper order before delivering them to the receiving process. There is a certain amount of overhead involved in this procedure, and if one packet is missing or delayed, then any packets which follow will have to wait until the errant packet is delivered before they can continue their journey.

2. **Connectionless (UDP, User Datagram Protocol)** emulate individual telegrams. There is no guarantee that any particular packet will get through undamaged (or at all), and no guarantee that the packets will get delivered in any particular order. There may even be duplicate packets delivered, depending on how the intermediary connections are configured. UDP transmissions are much faster than TCP, but applications must implement their own error checking and recovery procedures.

Sockets are considered a low-level communications channel, and processes may often choose to use something at a higher level

Remote Procedure Calls, RPC

- The general concept of RPC is to make procedure calls similarly to calling on ordinary local procedures, except the procedure being called lies on a remote machine.
- Implementation involves ***stubs*** on either end of the connection.
- The local process calls on the stub, much as it would call upon a local procedure.
- The RPC system packages up (marshals) the parameters to the procedure call, and transmits them to the remote system.
- On the remote side, the RPC daemon accepts the parameters and calls upon the appropriate remote procedure to perform the requested work.
- Any results to be returned are then packaged up and sent back by the RPC system to the local system, which then unpackages them and returns the results to the local calling procedure.
- One potential difficulty is the formatting of data on local versus remote systems. (e.g. big-endian versus little-endian.) The resolution of this problem generally involves an agreed-upon intermediary format, such as XDR (external data representation.)
- Another issue is identifying which procedure on the remote system a particular RPC is destined for.
- Remote procedures are identified by *ports*, though not the same ports as the socket ports.
- One solution is for the calling procedure to know the port number they wish to communicate with on the remote system. This is problematic, as the port number would be compiled into the code, and it makes it break down if the remote system changes their port numbers.
- More commonly a *matchmaker* process is employed, which acts like a telephone directory service. The local process must first contact the matchmaker on the remote system (at a well-known port number), which looks up the desired port number and returns it. The local process can then use that information to contact the desired remote procedure. This operation involves an extra step, but is much more flexible. An example of the matchmaker process is illustrated in Figure 3.21 below.
- One common example of a system based on RPC calls is a networked file system. Messages are passed to read, write, delete, rename, or check status, as might be made for ordinary local disk access requests.

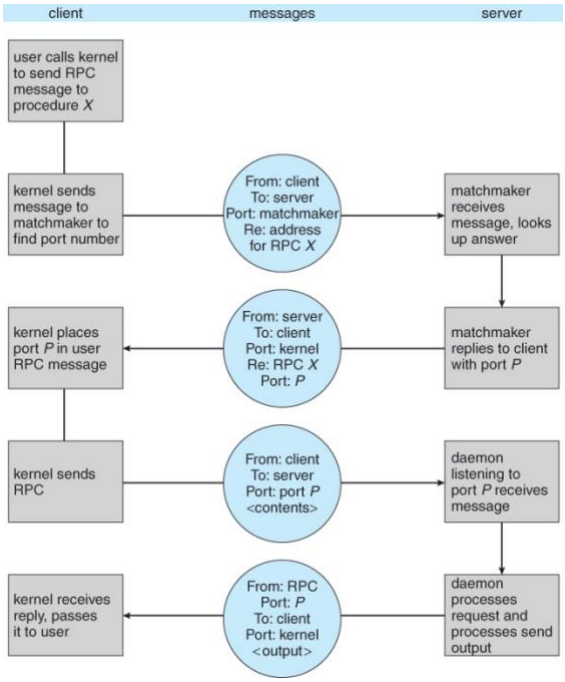


Figure 3.23 - Execution of a remote procedure call (RPC).

Pipes

Pipes are one of the earliest and simplest channels of communications between (UNIX) processes.

There are four key considerations in implementing pipes:

Unidirectional or Bidirectional communication?

Is bidirectional communication half-duplex or full-duplex?

Must a relationship such as parent-child exist between the processes?

Can pipes communicate over a network, or only on the same machine?

Ordinary Pipes

- Ordinary pipes are uni-directional, with a reading end and a writing end. (If bidirectional communications are needed, then a second pipe is required.)
- In UNIX ordinary pipes are created with the system call "int pipe(int fd [ ])".
- The return value is 0 on success, -1 if an error occurs.
- The int array must be allocated before the call, and the values are filled in by the pipe system call:
- fd[0] is filled in with a file descriptor for the reading end of the pipe
- fd[1] is filled in with a file descriptor for the writing end of the pipe
- UNIX pipes are accessible as files, using standard read( ) and write( ) system calls.
- Ordinary pipes are only accessible within the process that created them.
- Typically a parent creates the pipe before forking off a child.
- When the child inherits open files from its parent, including the pipe file(s), a channel of communication is established.
- Each process (parent and child) should first close the ends of the pipe that they are not using. For example, if the parent is writing to the pipe and the child is reading, then the parent should close the reading end of its pipe after the fork and the child should close the writing end.

- Figure 3.22 shows an ordinary pipe in UNIX, and Figure 3.23 shows code in which it is used.

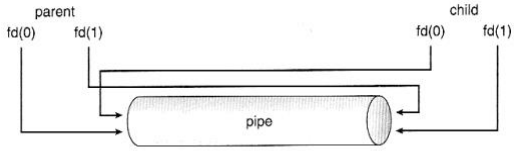


Figure 3.22 File descriptors for an ordinary pipe.

Figure 3.24

- Ordinary pipes in Windows are very similar
  - Windows terms them ***anonymous*** pipes
  - They are still limited to parent-child relationships.
  - They are read from and written to as files.
  - They are created with CreatePipe( ) function, which takes additional arguments.
  - In Windows it is necessary to specify what resources a child inherits, such as pipes.

Named Pipes

- Named pipes support bidirectional communication, communication between non parent-child related processes, and persistence after the process which created them exits. Multiple processes can also share a named pipe, typically one reader and multiple writers.
- In UNIX, named pipes are termed fifos, and appear as ordinary files in the file system.
- (Recognizable by a "p" as the first character of a long listing, e.g. /dev/initctl)
- Created with mkfifo( ) and manipulated with read( ), write( ), open( ), close( ), etc.
- UNIX named pipes are bidirectional, but half-duplex, so two pipes are still typically used for bidirectional communications.
- UNIX named pipes still require that all processes be running on the same machine. Otherwise sockets are used.
- Windows named pipes provide richer communications.
- Full-duplex is supported.
- Processes may reside on the same or different machines
- Created and manipulated using CreateNamedPipe( ), ConnectNamedPipe( ), ReadFile( ),and WriteFile( ).

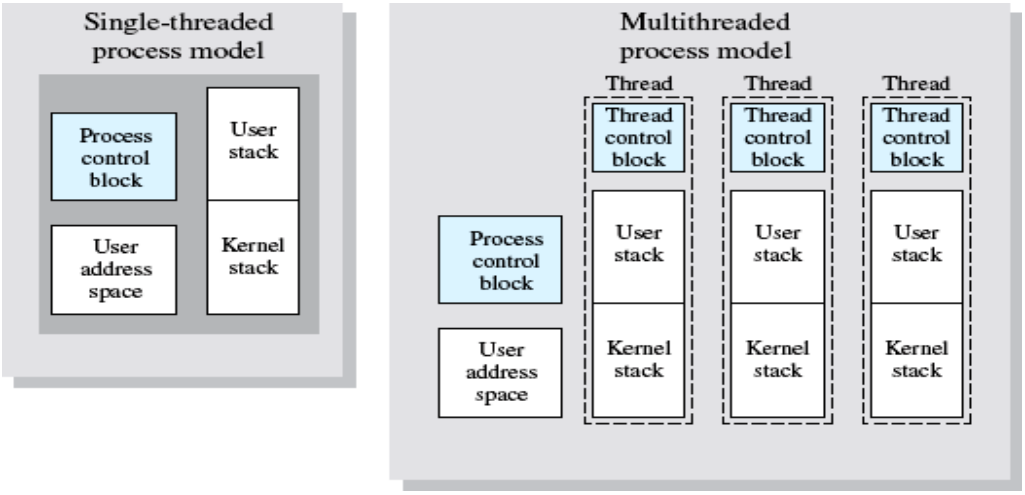
Thread

A thread, sometimes called a lightweight process (LWP), is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or heavyweight) process has a single thread of control. If the process has multiple threads of control, it can do more than one task at a time.

Motivation

Many software packages that run on modern desktop PCs are multithreaded.An application typically is implemented as a separate process with several thread of control.

Single-threaded and multithreaded



Ex: A web browser might have one thread display images or text while another thread retrieves data from the network. A word processor may have a thread for displaying graphics, another thread for reading keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

In certain situations a single application may be required to perform several similar tasks. For example, aweb server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps hundreds) of clients concurrently accessing it. If the web server ran as a traditional single- threaded process, it would be able to service only one client at a time.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is very heavyweight, as was shown in the previous chapter. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient for one process that contains multiple threads to serve the samepurpose. This approach would multithread the web-server process. The server would create a separate thread that would listen for client requests; when a request was made, rather than creating another process, it would create another thread to service the request.

Threads also play a vital role in remote procedure call (RPC) systems. RPCs allow inter-process communication by providing a communication mechanism similar to ordinary function or procedure calls. Typically, RPC servers are multithreaded. When a server receives a message, it services the message using a separate thread. This allows the server to service several concurrent requests.

Benefits

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

1. **Responsiveness:** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user

interaction in one thread while an image is being loaded in another thread.

2. **Resource sharing:** By default, threads share the memory and the resources of the process to which they belong. The benefit of code sharing is that it allows an application to have several different threads of activity all within the same address space.
3. **Economy**: Allocating memory and resources for process creation is costly. Alternatively, because threads share resources of the process to which they belong, it is more economical to create and context switch threads. It can be difficult to gauge empirically the difference in overhead for creating and maintaining a process rather than a thread, but in general it is much more time consuming to create and manage processes than threads. In Solaris 2, creating a process is about 30 times slower than is creating a thread, and context switching is about five times slower.
4. **Utilization of multiprocessor architectures**: The benefits of multithreading can be greatly increased in a multiprocessor architecture, where each thread may be running in parallel on a different processor. A single- threaded process can only run on one CPU, no matter how many are available.

Multithreading on a multi-CPU machine increases concurrency. In a single processor architecture, the CPU generally moves between each thread so quickly as to create an illusion of parallelism, but in reality only one thread is running at a time.

The OS supports the threads that can provided in following two levels:

User-Level Threads

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

Advantages:

- User-level threads do not require modification to operating systems.
- Simple Representation: Each thread is represented simply by a PC, registers, stack and a small controlblock, all stored in the user process address space.
- Simple Management: This simply means that creating a thread, switching between threads andsynchronization between threads can all be done without intervention of the kernel.
- Fast and Efficient: Thread switching is not much more expensive than a procedure call.

Disadvantages:

- There is a lack of coordination between threads and operating system kernel.
- User-level threads require non-blocking systems call i.e., a multithreaded kernel.

Kernel-Level Threads

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

- Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a processhaving large number of threads than process having small number of threads.
- Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

- The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of timesslower than that of user-level threads.

Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity.

Multithreading Models

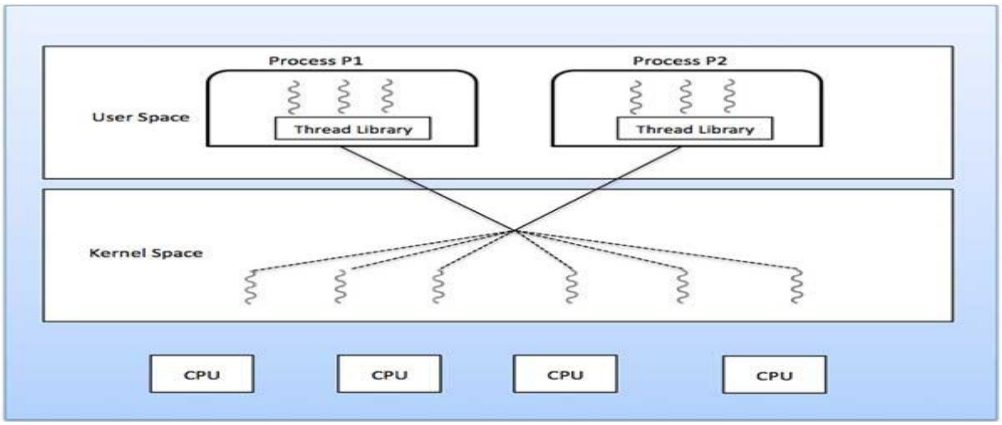
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

1. Many to many relationship.
2. Many to one relationship.
3. One to one relationship.

Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

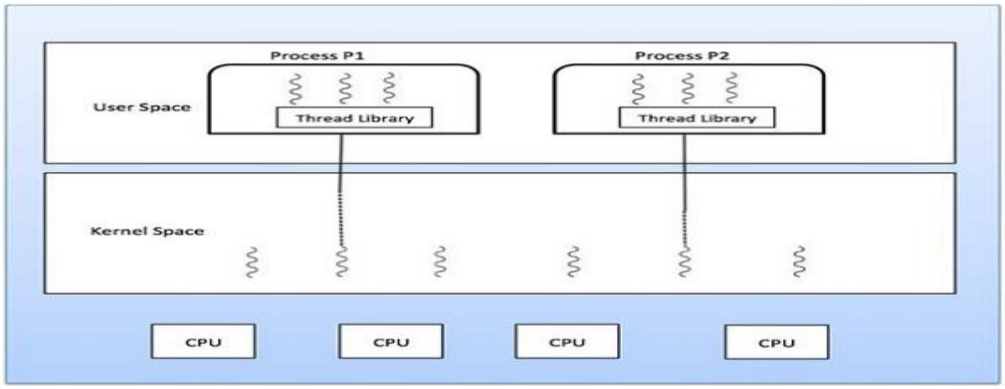
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Many to One Model

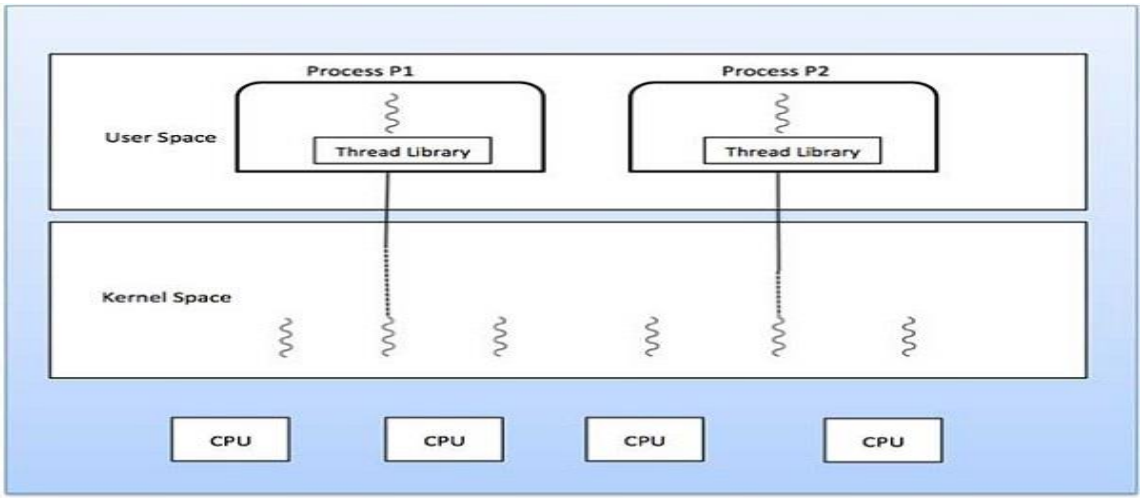
Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.
Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

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.

Scheduling Criteria

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

- **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 Algorithms

We'll discuss four major scheduling algorithms here which are following :

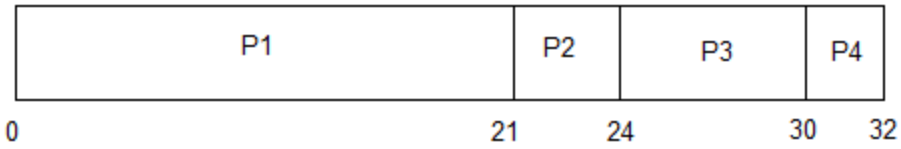
1. First Come First Serve(FCFS) Scheduling
2. Shortest-Job-First(SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling
5. Multilevel Queue Scheduling

First Come First Serve(FCFS) Scheduling

- Jobs are executed on first come, first serve basis.
- Easy to understand and implement.
- Poor in performance as average wait time is high.

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

Shortest-Job-First(SJF) Scheduling

- Best approach to minimize waiting time.
- Actual time taken by the process is already known to processor.
- Impossible to implement.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

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

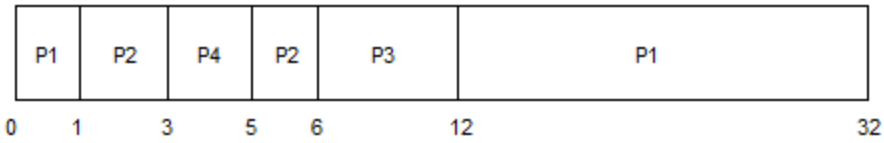
Hence the GANTT chart will be following :

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

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 preemptied.

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.

Priority Scheduling

- Priority is assigned for each process.
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS manner.

Priority can be decided based on memory requirements, time requirements or any other resource requirement.

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,

The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

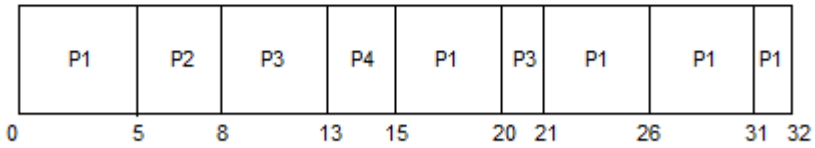
Round Robin(RR) Scheduling

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preemptied and other process executes for given time period.

- Context switching is used to save states of preemptied processes.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

Multilevel Queue Scheduling

- When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments.
- Scheduling must also be done between queues, that is scheduling one queue to get time relative to other queues. Two common options are strict priority (no job in a lower priority queue runs until all higher priority queues are empty) and round-robin (each queue gets a time slice in turn, possibly of different sizes.)
- Note that under this algorithm jobs cannot switch from queue to queue - Once they are assigned a queue, that is their queue until they finish.

Multilevel Feedback-Queue Scheduling

- Multilevel feedback queue scheduling is similar to the ordinary multilevel queue scheduling described above, except jobs may be moved from one queue to another for a variety of reasons:
  - If the characteristics of a job change between CPU-intensive and I/O intensive, then it may be appropriate to switch a job from one queue to another.
  - Aging can also be incorporated, so that a job that has waited for a long time can get bumped up into a higher priority queue for a while.

- Multilevel feedback queue scheduling is the most flexible, because it can be tuned for any situation. But it is also the most complex to implement because of all the adjustable parameters. Some of the parameters which define one of these systems include:
  - The number of queues.
  - The scheduling algorithm for each queue.
  - The methods used to upgrade or demote processes from one queue to another. (Which may be different.)
  - The method used to determine which queue a process enters initially.

Multiple-Processor Scheduling

- When multiple processors are available, then the scheduling gets more complicated, because now there is more than one CPU which must be kept busy and in effective use at all times.
- ***Load sharing*** revolves around balancing the load between multiple processors.
- Multi-processor systems may be **heterogeneous,** (different kinds of CPUs), or **homogenous,** (all the same kind of CPU). Even in the latter case there may be special scheduling constraints, such as devices which are connected via a private bus to only one of the CPUs. This book will restrict its discussion to homogenous systems.

Approaches to Multiple-Processor Scheduling

- One approach to multi-processor scheduling is ***asymmetric multiprocessing,*** in which one processor is the master, controlling all activities and running all kernel code, while the other runs only user code. This approach is relatively simple, as there is no need to share critical system data.
- Another approach is ***symmetric multiprocessing, SMP,*** where each processor schedules its own jobs, either from a common ready queue or from separate ready queues for each processor.
- Virtually all modern OSes support SMP, including XP, Win 2000, Solaris, Linux, and Mac OSX.

Processor Affinity

- Processors contain cache memory, which speeds up repeated accesses to the same memory locations.
- If a process were to switch from one processor to another each time it got a time slice, the data in the cache (for that process) would have to be invalidated and re-loaded from main memory, thereby obviating the benefit of the cache.
- Therefore SMP systems attempt to keep processes on the same processor, via ***processor affinity. Soft affinity*** occurs when the system attempts to keep processes on the same processor but makes no guarantees. Linux and some other OSes support ***hard affinity,*** in which a process specifies that it is not to be moved between processors.
- Main memory architecture can also affect process affinity, if particular CPUs have faster access to memory on the same chip or board than to other memory loaded elsewhere. (Non-Uniform Memory Access, NUMA.) As shown below, if a process has an affinity for a particular CPU, then it should preferentially be assigned memory storage in "local" fast access areas.

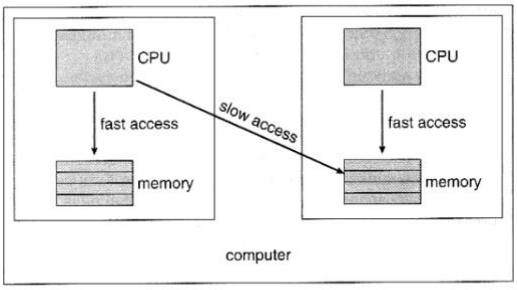


Figure 5.9 NUMA and CPU scheduling.

Load Balancing

- Obviously an important goal in a multiprocessor system is to balance the load between processors, so that one processor won't be sitting idle while another is overloaded.
- Systems using a common ready queue are naturally self-balancing, and do not need any special handling. Most systems, however, maintain separate ready queues for each processor.

Balancing can be achieved through either ***push migration*** or ***pull migration:***

Push migration involves a separate process that runs periodically, (e.g. every 200 milliseconds), and moves processes from heavily loaded processors onto less loaded ones.

Pull migration involves idle processors taking processes from the ready queues of other processors.

Push and pull migration are not mutually exclusive.

Note that moving processes from processor to processor to achieve load balancing works against the principle of processor affinity, and if not carefully managed, the savings gained by balancing the system can be lost in rebuilding caches. One option is to only allow migration when imbalance surpasses a given threshold.

Multicore Processors

Traditional SMP required multiple CPU chips to run multiple kernel threads concurrently.

Recent trends are to put multiple CPUs (cores) onto a single chip, which appear to the system as multiple processors.

Compute cycles can be blocked by the time needed to access memory, whenever the needed data is not already present in the cache. (Cache misses.) In Figure 5.10, as much as half of the CPU cycles are lost to memory stall.

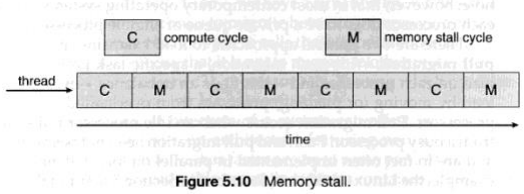


Figure 5.10 Memory stall.

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

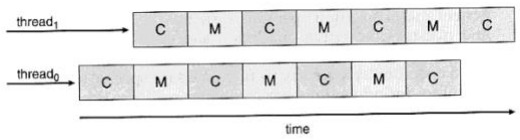


Figure 5.11 Multithreaded multicore system.

A dual-threaded dual-core system has four logical processors available to the operating system. The UltraSPARC T1 CPU has 8 cores per chip and 4 hardware threads per core, for a total of 32 logical processors per chip.

There are two ways to multi-thread a processor:

> ***Coarse-grained*** multithreading switches between threads only when one thread blocks, say on a memory read. Context switching is similar to process switching, with considerable overhead.
>
> ***Fine-grained*** multithreading occurs on smaller regular intervals, say on the boundary of instruction cycles. However the architecture is designed to support thread switching, so the overhead is relatively minor.

Note that for a multi-threaded multi-core system, there are **two** levels of scheduling, **at the kernel level:**

- The OS schedules which kernel thread(s) to assign to which logical processors, and when to make context switches using algorithms as described above.
- On a lower level, the hardware schedules logical processors on each physical core using some other algorithm.

The UltraSPARC T1 uses a simple round-robin method to schedule the 4 logical processors (kernel threads) on each physical core.

The Intel Itanium is a dual-core chip which uses a 7-level priority scheme (urgency) to determine which thread to schedule when one of 5 different events occurs.

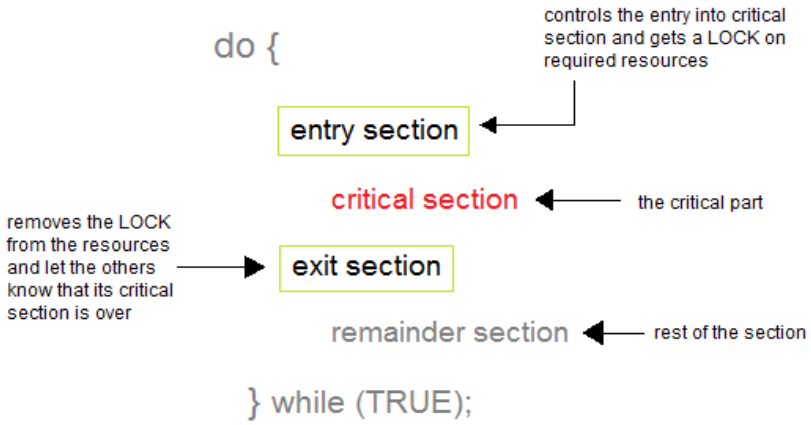
Virtualization and Scheduling

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

Critical Section Problem

- A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action.

- It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section.
- If any other process also wants to execute its critical section, it must wait until the first one finishes.
- The entry to the critical section is mainly handled by wait() function
- while the exit from the critical section is controlled by the signal() function.



Entry Section

In this section mainly the process requests for its entry in the critical section.

Exit Section

This section is followed by the critical section.

The solution to the Critical Section Problem

A solution to the critical section problem must satisfy the following three conditions:

1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, the system must grant the process permission to get into its critical section.

Race Condition

- At the time when more than one process is either executing the same code or accessing the same memory or any shared variable;
- In that condition, there is a possibility that the output or the value of the shared variable is wrong so for that purpose all the processes are doing the race to say that my output is correct.

- This condition is commonly known as **a race condition.**
- As several processes access and process the manipulations on the same data in a concurrent manner and due to which the outcome depends on the particular order in which the access of data takes place.
- Mainly this condition is a situation that may occur inside the **critical section**.
- Race condition in the critical section happens when the result of multiple thread execution differs according to the order in which the threads execute. But this condition is critical sections can be avoided if the critical section is treated as an atomic instruction. Proper thread synchronization using locks or atomic variables can also prevent race conditions.

A solution for the race condition should have following conditions:

1. No two processes may be simultaneously inside their critical section (mutual exclusion).
2. No process running outside its critical region may block other processes
3. No process should have to wait forever to enter critical region 4. No assumptions may be made about speeds or the number of CPUs.

Mutual Exclusion with Busy Waiting

Mutual Exclusion Solutions

- Busy waiting
- Sleep and Wakeup

Mutual Exclusion with Busy Waiting

- Disabling Interrupts
- Lock Variables
- Strict Alternation
- Peterson's Solution

Disabling Interrupt

Once a process enters the critical section, it disables all interrupts.

With interrupts turned off the CPU will not be switched to another process.

The process finishes its job in the critical section.

It is unwise to give user process the power to turn off interrupts. If the process never turned it on again, it might cause the end of the system.

Mutual Exclusion with Busy Waiting (Lock Variable)

It is a Software solution

There is a single shared variable (lock), initially 0.

If lock = 0, the process sets it to 1 and enters the critical section.

If lock = 1, the process will wait until it becomes 0. -- busy waiting –

repeat

while lock ≠0 do

; (no-operation)

lock = 1

Critical section

lock = 0
until false

1. Initially, lock = 0.
2. A process P1 tries to enter its critical section. A process P1 checks lock value (= 0).
3. Before updating lock = 1, Process P2 tries to enter its critical section. P2 checks lock value (still = 0). P2 sets lock = 1 and goes to its critical section.
4. P1 is rescheduled. P1 continues. P1 sets lock = 1 and goes to its critical section.
5. Now P1 and P2 are in the their critical sections at the same time.

It is violating condition 1

Mutual Exclusion with Busy Waiting (Strict Alternation)

- Take turns
- For processor Pi and Pj
- Variable turn can be i or j
- If turn = i, process Pi can enter its critical section.
- Once Pi finishes its job in its critical section,
- Pi sets turn = j, let process Pj enter its critical section.

repeat
while turn ≠ I do
; (no-operation)
Critical Section
turn = j;
Non-Critical Section
until false

Initially turn = 0

1. P0 is in its C.S. while P1 is in its non-critical section.
2. P0 finishes C.S. and sets turn = 1, P1 is still in its noncritical section.
3. P0 finishes its non-critical section and wants to go to its C.S. again but turn= 1.
4. P0 might need to wait forever to enter its C.S.

It is violating condition 2 and condition 3

Mutual Exclusion with Busy Waiting (Peterson's Solution)

```
#define false 0
#define true 1
 #define n 2
int turn
 int interested[n]
void enter_region(int process)
 {
```

```
  int other;
  other = 1 – process
  interested[process] = true
  turn = process;
 while (turn ==process && interested[other]==true) ;
  /*no operation */
}
 void leave_region(int process)
{
  interested[process] = false;
}

 void main()
{
  Repeat
 enter_region (int i)
   Critical Section
 leave_region (int i)
 Non-Critical Section
 until false
}
```

- Initially, neither process is in the critical section
- P0 calls enter_region (0)
- Set interested[0] = true;
- Set turn = 0
- go to critical section
- P1 calls enter_region(1) to get into its critical section
    - set interested[1] = true;
    - set turn = 1;
- . since interested[0] = true, it keeps looping for interested[0] = false
- . (finally) P0 finishes its critical section and calls leave_region(0)
    - set interested[0] = false
- P1 finds out interested[0] = false, P1 goes to its critical section

Sleep and Wakeup

- Instead of busy waiting, it goes to sleeping state.
- Once a process finishes its Critical section, It calls wakeup function which allows one of sleeping process to get into its critical section.

Semaphore

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes.
This integer variable is called a **semaphore**.
So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, **wait** and **signal** designated by P(S) and V(S) respectively.

```
In very simple words, the semaphore is a variable that can hold only a non-negative Integer value, share,
P(S): if S >= 1 then S := S - 1

    else <block and enqueue the process>;



V(S): if <some process is blocked on the queue>

      then <unblock a process>

    else S := S + 1;
```

The classical definitions of **wait** and **signal** are:

- **Wait**: This operation decrements the value of its argument S, as soon as it would become non-negative(greater than or equal to 1).
- This Operation mainly helps you to control the entry of a task into the critical section. In the case of the negative or zero value, no operation is executed.
- wait() operation was originally termed as P; so it is also known as **P(S) operation**.
- The definition of wait operation is as follows:

```
wait(S)

{

   while (S<=0);//no operation

   S--;

}
```

- When one process modifies the value of a semaphore then, no other process can simultaneously modify that same semaphore's value. In the above case the integer value of S(S<=0) as well as the possible modification that is S-- must be executed without any interruption.

  Signal: Increments the value of its argument S, as there is no more process blocked on the queue.This Operation is mainly used to control the exit of a task from the critical section.signal() operation was originally termed as V; so it is also known as **V(S) operation**. The definition of signal operation is as follows:

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

Also, note that all the modifications to the integer value of semaphore in the wait() and signal() operations must be executed indivisibly.

Properties of Semaphores

1. It's simple and always have a non-negative integer value.
2. Works with many processes.
3. Can have many different critical sections with different semaphores.
4. Each critical section has unique access semaphores.
5. Can permit multiple processes into the critical section at once, if desirable.
6.

Types of Semaphores

Semaphores are mainly of two types in Operating system:

1. **Binary Semaphore:**
   It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **Mutex**. A binary semaphore is initialized to 1 and only takes the values 0 and 1 during the execution of a program. In Binary Semaphore, the wait operation works only if the value of semaphore = 1, and the signal operation succeeds when the semaphore= 0. Binary Semaphores are easier to implement than counting semaphores.
2. **Counting Semaphores:**
   These are used to implement **bounded concurrency**. The Counting semaphores can range over an **unrestricted domain**. These can be used to control access to a given resource that consists of a finite number of Instances. Here the semaphore count is used to indicate the number of available resources. If the resources are added then the semaphore count automatically gets incremented and if the resources are removed, the count is decremented. Counting Semaphore has no mutual exclusion.

Advantages of Semaphores

- With the help of semaphores, there is a flexible management of resources.
- Semaphores are machine-independent and they should be run in the machine-independent code of the microkernel.
- Semaphores do not allow multiple processes to enter in the critical section.
- They allow more than one thread to access the critical section.
- As semaphores follow the mutual exclusion principle strictly and these are much more efficient than some other methods of synchronization.

- No wastage of resources in semaphores because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if any condition is fulfilled in order to allow a process to access the critical section.

Disadvantages of Semaphores

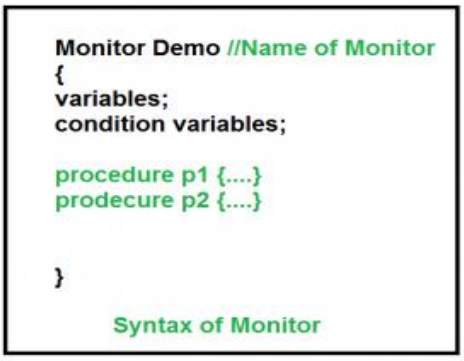
- One of the biggest limitations is that semaphores may lead to priority inversion; where low priority processes may access the critical section first and high priority processes may access the critical section later.
- To avoid deadlocks in the semaphore, the Wait and Signal operations are required to be executed in the correct order.
- Using semaphores at a large scale is impractical; as their use leads to loss of modularity and this happens because the wait() and signal() operations prevent the creation of the structured layout for the system.
- Their use is not enforced but is by convention only.
- With improper use, a process may block indefinitely. Such a situation is called **Deadlock**. We will be studying deadlocks in detail in coming lessons.

Monitors

The monitor is one of the ways to achieve Process synchronization.
The monitor is supported by programming languages to achieve mutual exclusion between processes..

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.

Syntax:



Condition Variables:
Two different operations are performed on the condition variables of the monitor.
Wait.
signal.
let say we have 2 condition variables
condition x, y; // Declaring variable

Wait operation
x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.
Note: Each condition variable has its unique block queue.

Signal operation
x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.
If (x block queue empty)

// Ignore signal

else

// Resume a process from block queue.

Advantages of Monitor:
Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphore.

Disadvantages of Monitor:
Monitors have to be implemented as part of the programming language .
The compiler must generate code for them.
This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes.
Some languages that do support monitors are Java,C#,Visual Basic,Ada and concurrent Euclid.

Barrier
In parallel computing, a barrier is a type of synchronization method.
A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.
Many collective routines and directive-based parallel languages impose implicit barriers.
For example, a parallel do loop in Fortran with OpenMP will not be allowed to continue on any thread until the last iteration is completed.
This is in case the program relies on the result of the loop immediately after its completion.
In message passing, any global communication (such as reduction or scatter) may imply a barrier.

Dynamic barriers
Classic barrier constructs define the set of participating processes/threads statically. This is usually done either at program startup or when a barrier like the Pthreads barrier is instantiated. This restricts the possible applications for which barriers can be used.

Classic Problems of Synchronisation
The classical problem depicting flaws of process synchronaization in systems where cooperating processes are present.
There are three classic problems of synchronization

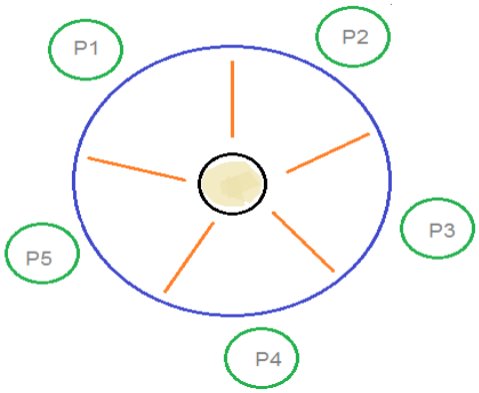
1. Bounded Buffer (Producer-Consumer) Problem
2. Dining Philosophers Problem
3. The Readers Writers Problem

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.

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.



Dining Philosophers Problem

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.

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.

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 themutex 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);
}
```

- 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.