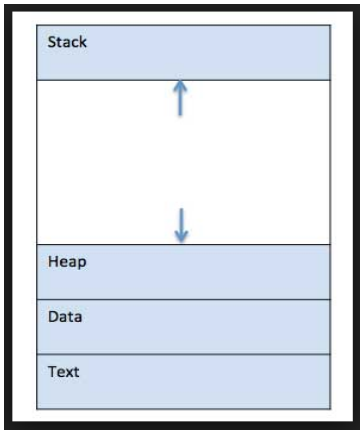
Module-2

PROCESS

A process is basically a program in execution. The execution of a process must progress in a sequential fashion. A process is defined as an entity which represents the basic unit of work to be implemented in the system. To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program. When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory:



1.Stack: The process Stack contains the temporary data such as method/function parameters, return address, and local variables.

2.Heap: This is a dynamically allocated memory to a process during its runtime.

3.Text: This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.

4. Data :This section contains the global and static variables.

PROCESS STATE DIAGRAM

As a process executes, it changes state. The sate of a process is defined in part by the current activity of that process. A process may be in one of the following states:

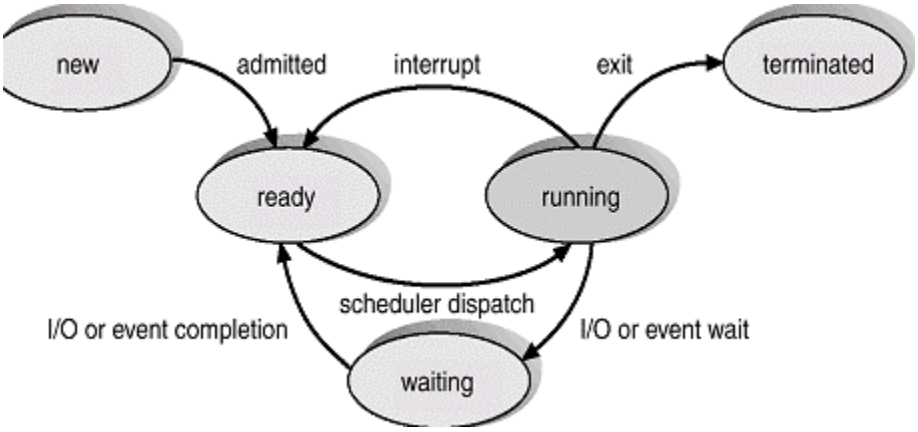
New: The process is being created

Running: Instructions are being executed

Waiting: The process is waiting for some event to occur

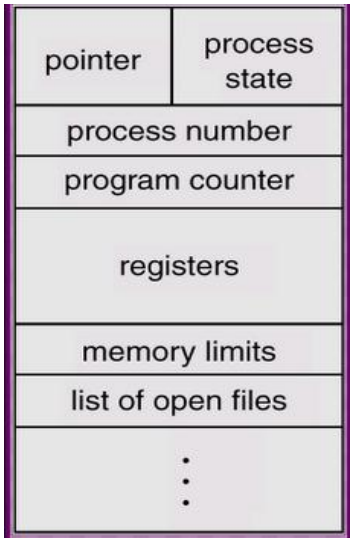
Ready: The process is waiting to be assigned to a processor

Terminated: The process has finished execution



Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process.it is also called as task control block (TCB).



It contains many pieces of information associated with a specific process, including these:

1. Process state
2. Program counter
3. CPU registers
4. CPU scheduling information
5. Memory management information
6. Accounting information

7. I/O status information

1. Process state: The current state of the process i.e., whether it is ready, running, waiting and so on.

2. Program counterprogram Counter is a pointer to the address of the next instruction to be executed for this process.

3.CPU registers: Various CPU registers where process need to be stored for execution for running state.

4.CPU scheduling information: This information includes a process priority, pointers to scheduling queues, and other scheduling parameters.

5. Memory management information: This includes the information of page table, memory limits, and Segment table depending on memory used by the operating system.

6. Accounting information : This includes the amount of CPU used for process execution, time limits, execution ID etc.

7.I/O status information: This information includes the list of I/O devices allocated to the process, a list of open files and so on.

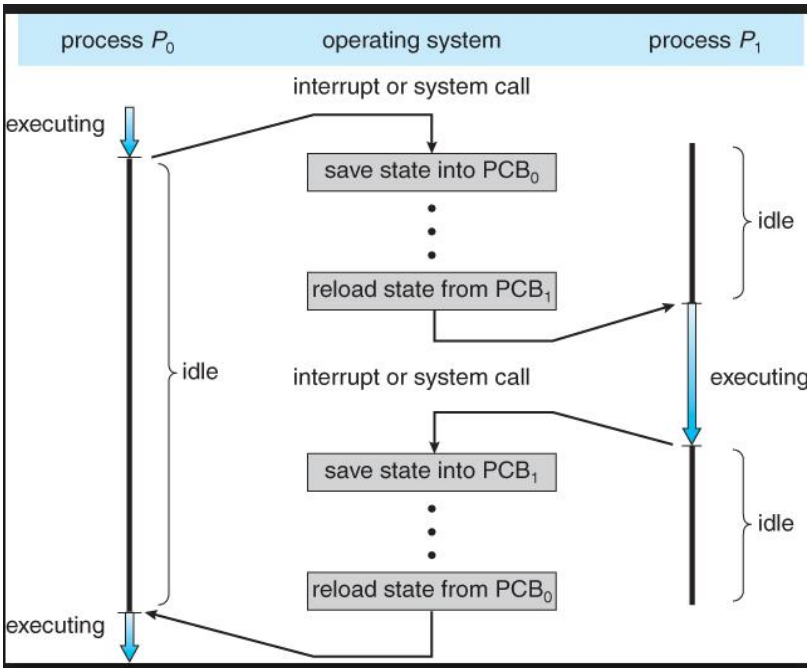


Fig: showing CPU switch from process to process

Process scheduling

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy. Process scheduling is an essential part of Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues:

- Job queue - This queue keeps all the processes in the system.
- Ready queue - This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- Device queues - The processes which are blocked due to unavailability of an I/O device constitute this queue.

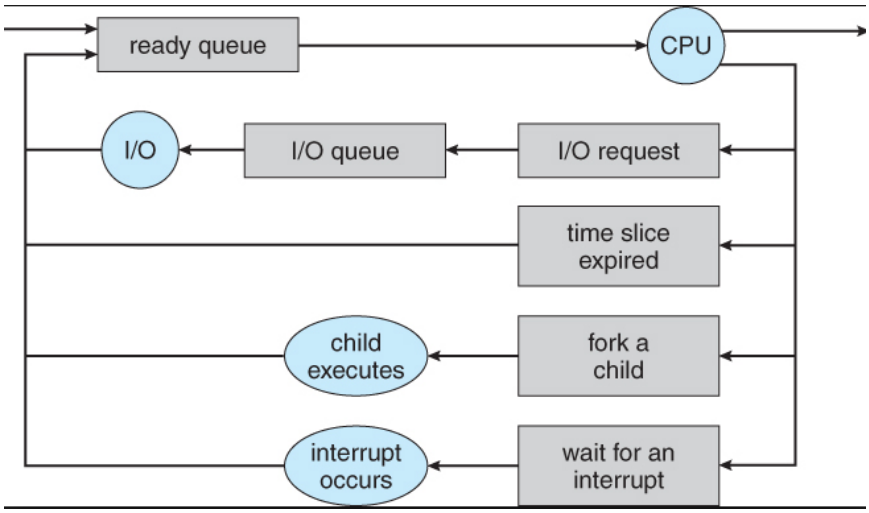


Fig: queuing diagram representation of process scheduling

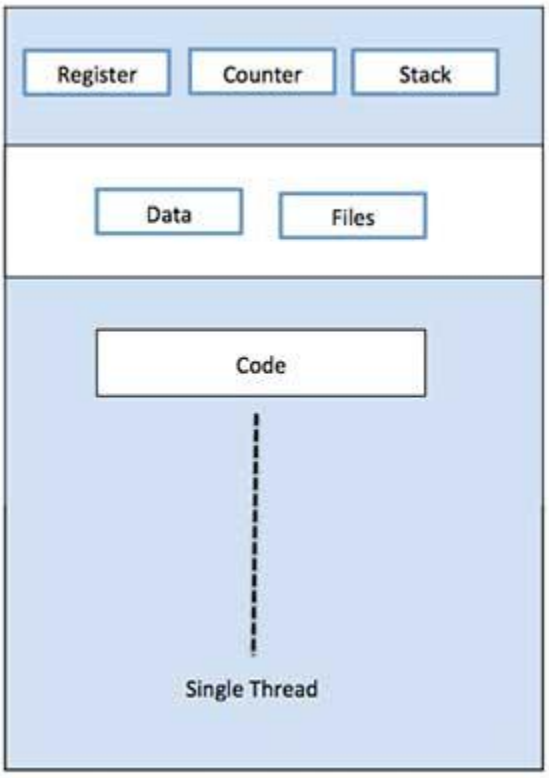
Threads

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

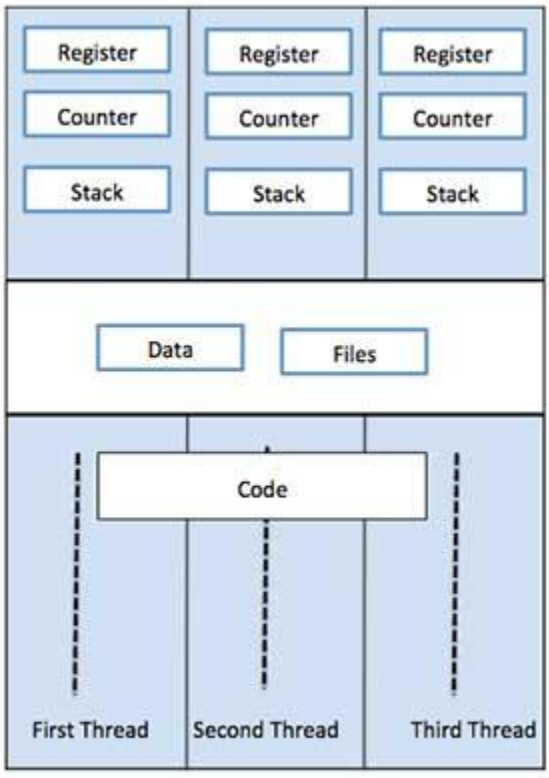
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.



Single Process P with single thread



Single Process P with three threads

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

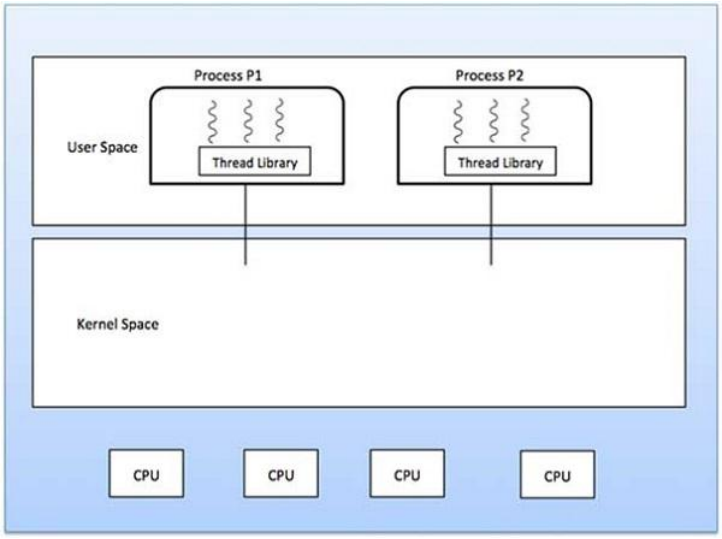
Types of Thread

Threads are implemented in following two ways −

- **User Level Threads** − User managed threads.
- **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.



Advantages

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

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

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

Disadvantages

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

Multithreading Models

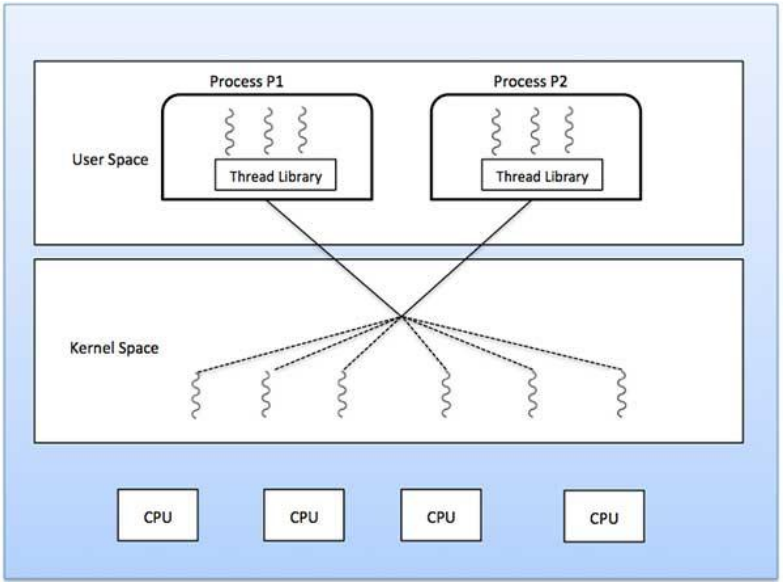
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

- Many to many relationship.
- Many to one relationship.
- 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.

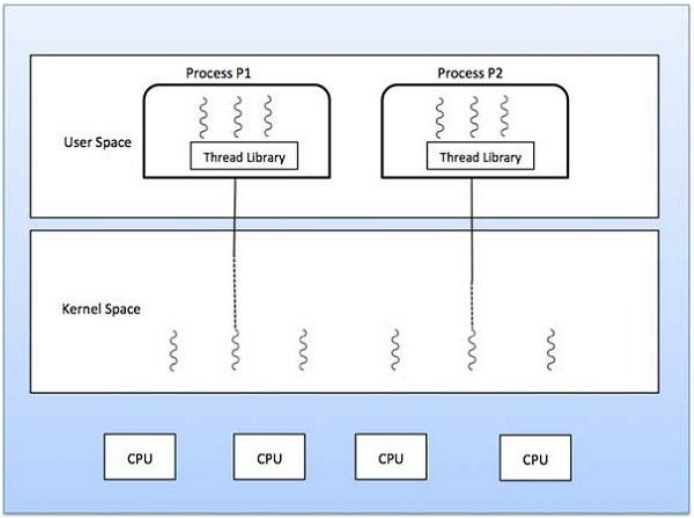
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model 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.

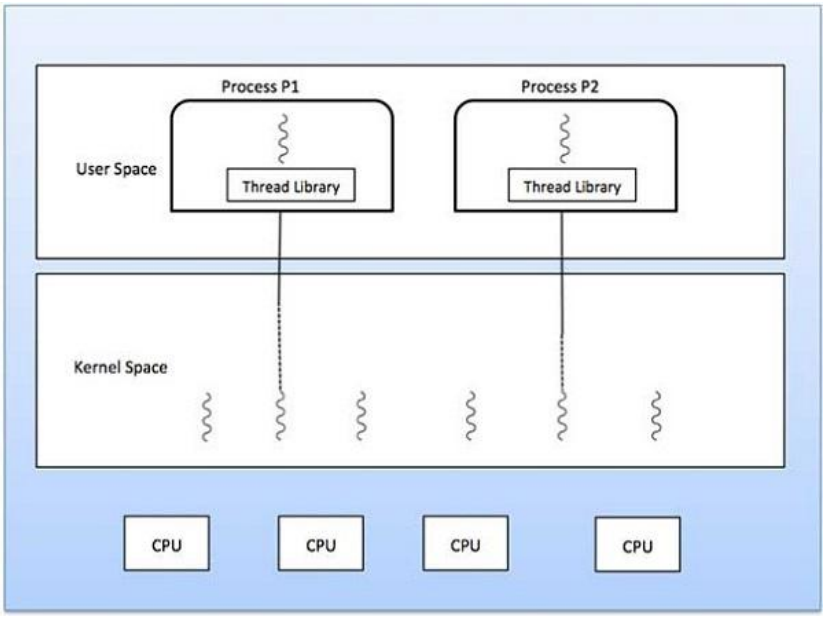


One to One Model

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.

Schedulers

Schedulers are special system software which handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types:

1 .Long-Term Scheduler

2. Short-Term Scheduler

3. Medium-Term Scheduler

1. Long-Term Scheduler

It is also called a job scheduler. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

2. Short-Term Scheduler

It is also called as CPU scheduler. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running.state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them. Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

3. Medium-Term Scheduler

Medium-term scheduling is a part of swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in charge of handling the swapped out-processes. A running process may become suspended if it makes an I/O request. A suspended process cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Comparison among Schedulers

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features. When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers.

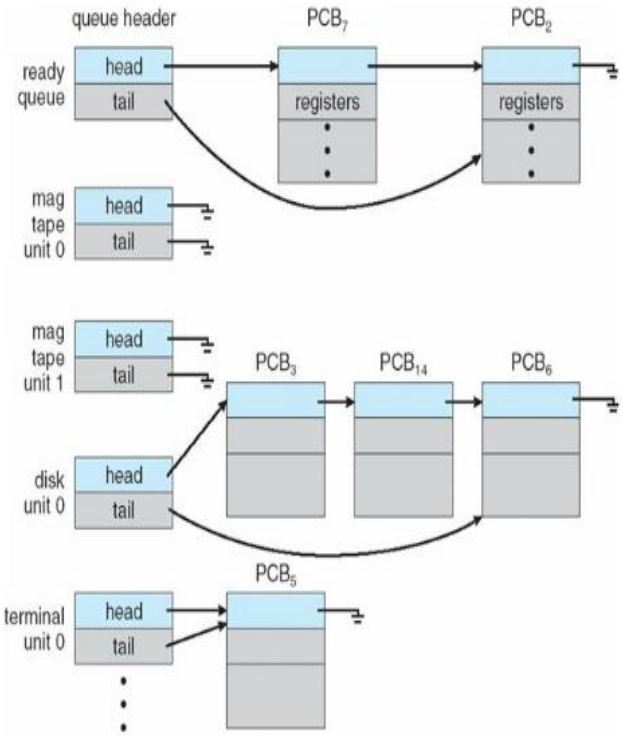
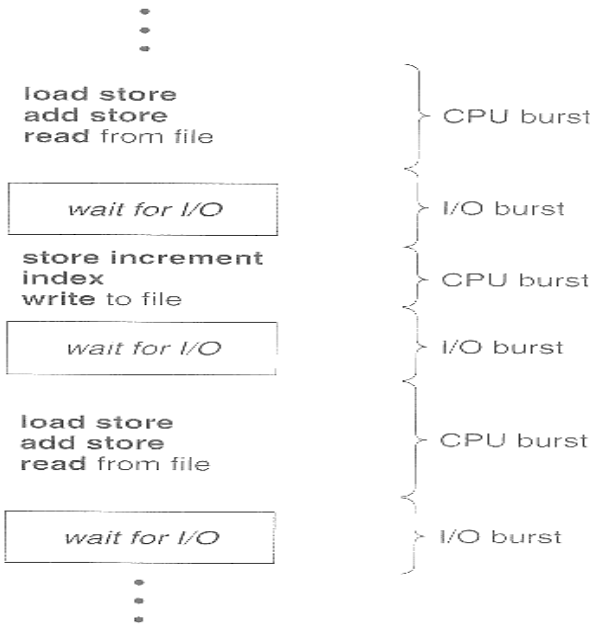


Fig: the ready queue and various I/O device queues

CPU-I/O Burst Cycle:

All most all processes alternate between two states in a continuing cycle, as shown in figure below:

- A CPU burst of performing calculations, and
- An I/O burst, waiting for data transfer in or out of the system.



Alternating sequence CPU and I/O bursts

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.

Dispatcher

Another component that is involved in the CPU-scheduling function is the dispatcher, which is the module that gives control of the CPU to the process selected by the short-term scheduler. It receives control in kernel mode as the result of an interrupt or system call. The functions of a dispatcher involve the following:

- Context switches, in which the dispatcher saves the state (also known as context) of the process or thread that was previously running; the dispatcher then loads the initial or previously saved state of the new process.
- Switching to user mode.
- Jumping to the proper location in the user program to restart that program indicated by its new state.

The dispatcher should be as fast as possible, since it is invoked during every process switch. During the context switches, the processor is virtually idle for a fraction of time, thus unnecessary context switches should be avoided. The time it takes for the dispatcher to stop one process and start another is known as the ***dispatch latency***.

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

There are six popular process scheduling algorithms which we are going to discuss in this chapter:

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

CPU Scheduling deals with the problem of deciding which of the process in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms.

1. Preemptive Scheduling algorithms

2. Non Preemptive Scheduling algorithms

1. Preemptive Scheduling algorithms

Preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state

2. Non Preemptive scheduling algorithms

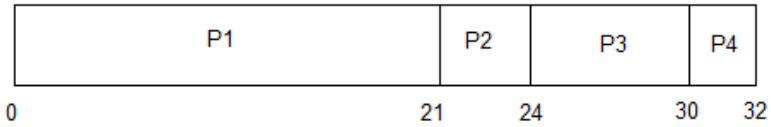
Once CPU given to the process it cannot be preempted until completes its CPU burst time.

First Come First Serve (FCFS) Scheduling

- Jobs are executed on first come, first served basis.
- It is a non-preemptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance, as average wait time is high.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

GANTT chart



Completion time:

P1=21

P2=24

P3=30

P4=32

Turnaround time: Completion time-Arrival time

P1=21-0=21

P2=24-0=24

P3=30-0=30

P4=32-0=32

Average Turnaround time=(P1+P2+P3+P4)/4=(21+24+30+32)/4=26.73 ms

Waiting time: Turnaround time-burst time

P1=21-21=0

P2=24-3=21

P3=30-6=24

P4=32-2=30

Average Waiting time = (P1+P2+P3+P4)/4= (0+21+24+30)/4=18.75 ms

Shortest Job Next (SJN) or Shortest job first(SJF)

Associate with each process the length of its next burst. Use these lengths to schedule the process with the shortest time. Two schemes:

1. Preemptive Scheduling algorithms: Preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state

2. Non Preemptive scheduling algorithms: Once CPU given to the process it cannot be preempted until completes its CPU burst time.

Ex: Sjf preemptive

process	arrival time	burst time
p1	0	7
p2	2	4
p3	4	1
p4	5	4

gantt chart:

P1	P2	P3	P2	P4	p1

0 2 4 5 7 11 16

Completion time:

P1=16

P2=7

P3=5

P4=11

Turnaround time: Completion time-Arrival time

P1=16-0=16

P2=7-2=5

P3=5-4=1

P4=11-5=6

Average Turnaround time= (P1+P2+P3+P4)/4= (16+5+1+6)/4= 7 ms

Waiting time: Turnaround time-burst time

P1=16-7=9

P2=5-4=1

P3=1-1=0

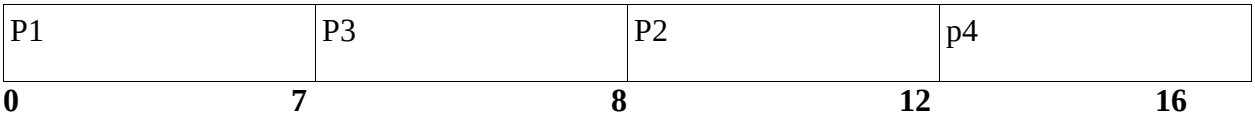
P4=6-4=2

Average Waiting time = (P1+P2+P3+P4)/4= (9+1+0+2)/4=3 ms

Ex:Non Preemptive Sjf

process	arrival time	burst time
p1	**0**	**7**
p2	**2**	**4**
p3	**4**	**1**
p4	**5**	**4**

Gantt chart:



Completion time:

P1=7

P2=12

P3=8

P4=16

Turnaround time: Completion time-Arrival time

P1=7-0=7

P2=12-2=10

P3=8-4=4

P4=16-5=11

Average Turnaround time= (P1+P2+P3+P4)/4= (7+10+4+11)/4= 8 ms

Waiting time: Turnaround time-burst time

P1=7-7=0

P2=10-4=6

P3=4-1=3

P4=11-4=7

Average Waiting time = (P1+P2+P3+P4)/4= (0+6+3+7)/4=4 ms

Round Robin

Round robin scheduling is similar to FCFS scheduling, except that CPU bursts are assigned with limits called **time quantum**.

- When a process is given the CPU, a timer is set for whatever value has been set for a time quantum.
- If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU just like the normal FCFS algorithm.
- If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue.
- The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on.
- RR scheduling can give the effect of all processors sharing the CPU equally, although the average wait time can be longer than with other scheduling algorithms. In the following example the average wait time is 5.66 ms

process	**burst time**
p1	**24**
p2	**3**
p3	**3**

time quantum=3 ms

Gantt chart:

P1	P2	P3	P1	P1	P1	P1	p1

0 4 7 10 14 18 22 26 30

Completion time:

p1=30

p2=7

p3=10

Turnaround time: Completion time-Arrival time

P1=30-0=30

P2=7-0=7

P3=10-0=10

Average Turnaround time= (P1+P2+P3)/3= (30+7+10)/3= 15.6 ms

Waiting time: Turnaround time-burst time

P1=30-24=6

P2=7-3=4

P3=10-3=7

Average Waiting time = (P1+P2+P3)/3= (6+4+7)/3=5.6 ms

Priority Scheduling

- Priority scheduling is a more general case of SJF, in which each job is assigned a priority and the job with the highest priority gets scheduled first. (SJF uses the inverse of the next expected burst time as its priority - The smaller the expected burst, the higher the priority.)
- Note that in practice, priorities are implemented using integers within a fixed range, but there is no agreed-upon convention as to whether "high" priorities use large numbers or small numbers. This book uses low number for high priorities, with 0 being the highest possible priority.
- For example, the following Gantt chart is based upon these process burst times and priorities, and yields an average waiting time of 8.2 ms:

process	burst time	priority
p1	**10**	**3**
p2	**1**	**1**
p3	**2**	**4**
p4	**1**	**5**
p5	**5**	**2**

Gantt chart:

Completion time:

P1=16

P2=1

P3=18

P4=19

p5=6

Turnaround time: Completion time-Arrival time

P1=16-0=16

P2=1-0=1

P3=18-0=18

P4=19-0=19

p5=6-0=6

Average Turnaround time=(P1+P2+P3+P4+p5)/5=(16+1+18+19+6)/5

=12 ms

Waiting time: Turnaround time-burst time

P1=16-10=6

P2=1-1=0

P3=18-2=16

P4=19-1=18

p5=6-5=1

Average Waiting time = (P1+P2+P3+P4+p5)/5

= (6+0+16+18+1)/5=8.2 ms

Priorities can be assigned either internally or externally. Internal priorities are assigned by the OS using criteria such as averageburst time, ratio of CPUtoI/O activity, system resource use, and other factors available to the kernel. External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc.

- Priority scheduling can be either preemptive or non-preemptive.
- Priority scheduling can suffer from a major problem known as **indefinite blocking**, or **starvation**, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.

One common solution to this problem is **aging**, in which priorities of jobs increase the longer they wait. Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

Multiple-Level Queues Scheduling :

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.

- Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

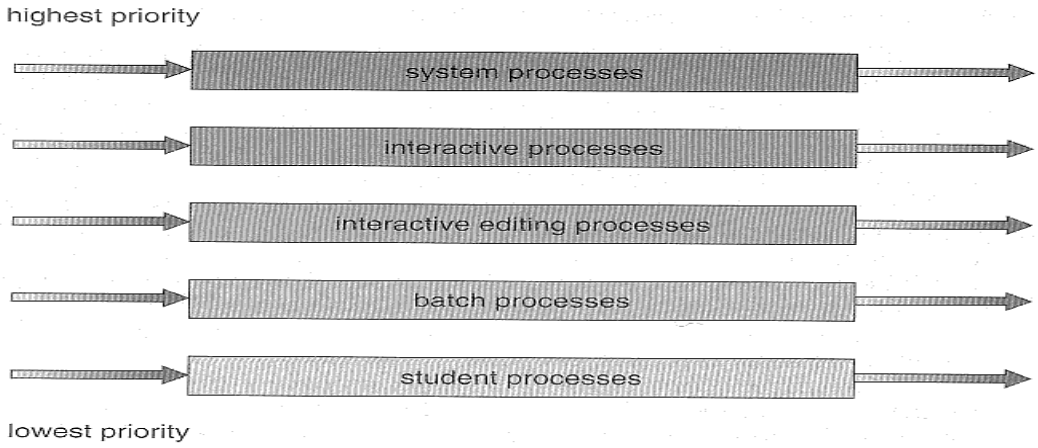


Figure 5.6 Multilevel queue scheduling.

Multilevel feedback queue-scheduling

- Multilevel feedback queue-scheduling Algorithm allows a process to move between queues. It uses many ready queues and associate a different priority with each queue.
- The Algorithm chooses to process with highest priority from the occupied queue and run that process either preemptively or unpreemptively.
- If the process uses too much CPU time it will moved to a lower-priority queue. Similarly, a process that wait too long in the lower-priority queue may be moved to a higher-priority queue may be moved to a highest-priority queue.
- Note that this form of aging prevents starvation.

Example:

- A process entering the ready queue is placed in queue 0.
- If it does not finish within 8 milliseconds time, it is moved to the tail of queue 1.
- If it does not complete, it is preempted and placed into queue 2.

- Processes in queue 2 run on a FCFS basis, only when queue 2 run on a FCFS basis, only when queue 0 and queue 1 are empty.

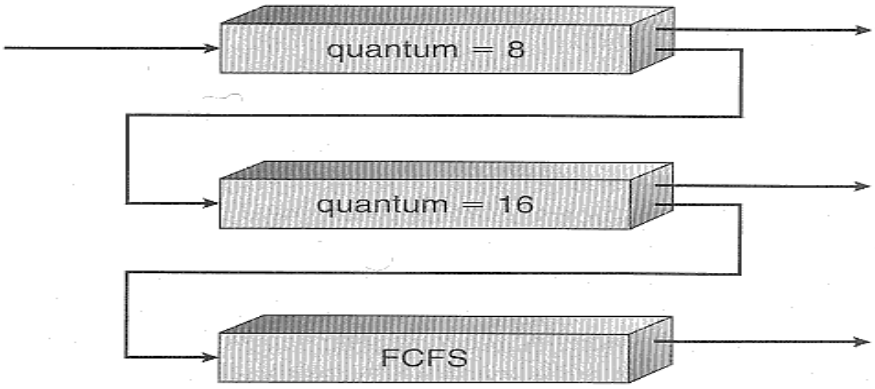


Figure 5.7 Multilevel feedback queues.

Algorithm Evaluation

The first step in determining which algorithm (and what parameter settings within that algorithm) is optimal for a particular operating environment is to determine what criteria are to be used, what goals are to be targeted, and what constraints if any must be applied. For example, one might want to "maximize CPU utilization, subject to a maximum response time of 1 second".

Once criteria have been established, then different algorithms can be analyzed and a "best choice" determined. The following sections outline some different methods for determining the "best choice".

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.

Deterministic Modeling

For FCFS the process would be executed in the following order, with the following wait times

Process	Burst Time	Wait Time
P1	9	0
P2	33	9
P3	2	42
P4	5	44
P5	14	49

Therefore, the average waiting time is ((0 + 9 + 42 + 44 + 49) / 5) = 28.80 milliseconds

For SJF (non preempted) the processes would run as follows

Process	Burst Time	Wait Time
P3	2	0
P4	5	2
P1	9	7
P5	14	16
P2	33	30

The average waiting time is ((0 + 2 + 7 + 16 + 30) / 5) = 11 milliseconds

For RR the jobs would execute as follows

Process	CPU Time
P1	8
P2	8
P3	2
P4	5
P5	8
P1	1
P2	8
P5	6
P2	8
P2	8
P2	1

The waiting time for each process is as follows

P1 : 0 + 23 = 23
P2 : 8 + 16 + 6 = 30
P3 : 16
P4 : 18
P5 : 23 + 9 = 32

Therefore, the average waiting time is ((23 + 30 + 16 + 18 + 32) / 5) = 23.80

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.

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:

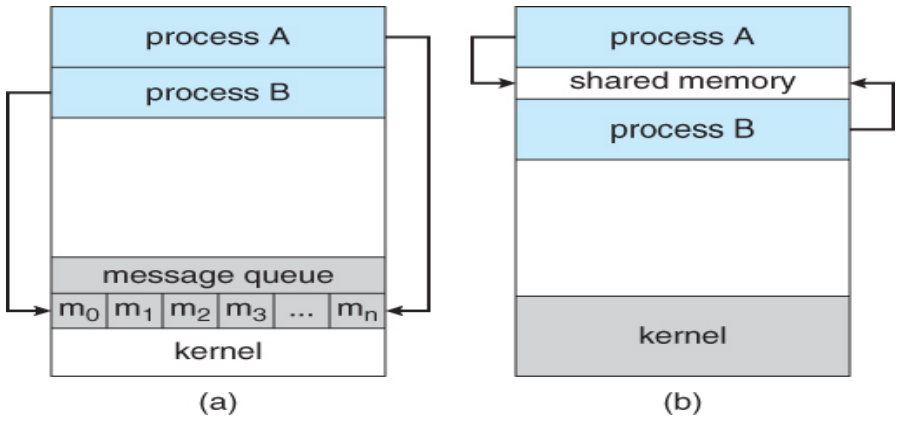
Information Sharing - There may be several processes which need access to the same file for example. (e.g. pipelines.)

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.)

Modularity - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)

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**. Below Figure illustrates the difference between the two systems:



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

Interprocess Communication – Shared Memory

● An area of memory shared among the processes that wish to communicate

● The communication is under the control of the processes not the operating system.

● Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

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;
```

- **Bounded-Buffer – Producer**

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

- **Bounded Buffer – Consumer**

```
item next_consumed;
while (true) {
while (in == out)
; /* do nothing */
next_consumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
/* consume the item in next consumed */
}
```

Interprocess Communication – 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)

● receive(message)

● The message size is either fixed or variable

Implementation of communication link

● Physical:

- Shared memory
- Hardware bus
- Network

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 (port)

● 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
    - P1, P2, and P3 share mailbox A
    - P1, sends; P2 and P3 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 -- the sender is blocked until the message is received

● Blocking receive -- the receiver is blocked until a message is available

● Non-blocking is considered asynchronous

● Non-blocking send -- the sender sends the message and continue

● Non-blocking receive -- the receiver receives:

● A valid message, or

● Null message

● Different combinations possible

● If both send and receive are blocking, we have a rendezvous

Communications in Client-Server Systems

● Sockets

● Remote Procedure Calls

● Pipes

Sockets

● A socket is defined as an endpoint for communication

● Concatenation of IP address and port – a number included at start of message packet to differentiate network services on a host

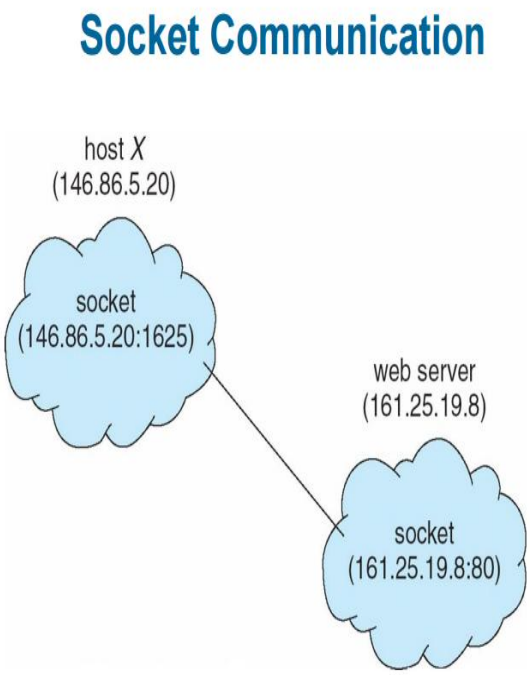
● The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8

● Communication consists between a pair of sockets

● All ports below 1024 are well known, used for standard services

● Special IP address 127.0.0.1 (loopback) to refer to system on which process is running.

Socket Communication




```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Remote Procedure Calls

● Remote procedure call (RPC) abstracts procedure calls between processes on networked systems

● Again uses ports for service differentiation

● 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

● On Windows, stub code compile from specification written in Microsoft Interface Definition Language (MIDL)

● Data representation handled via External Data Representation (XDL) format to account for different architectures

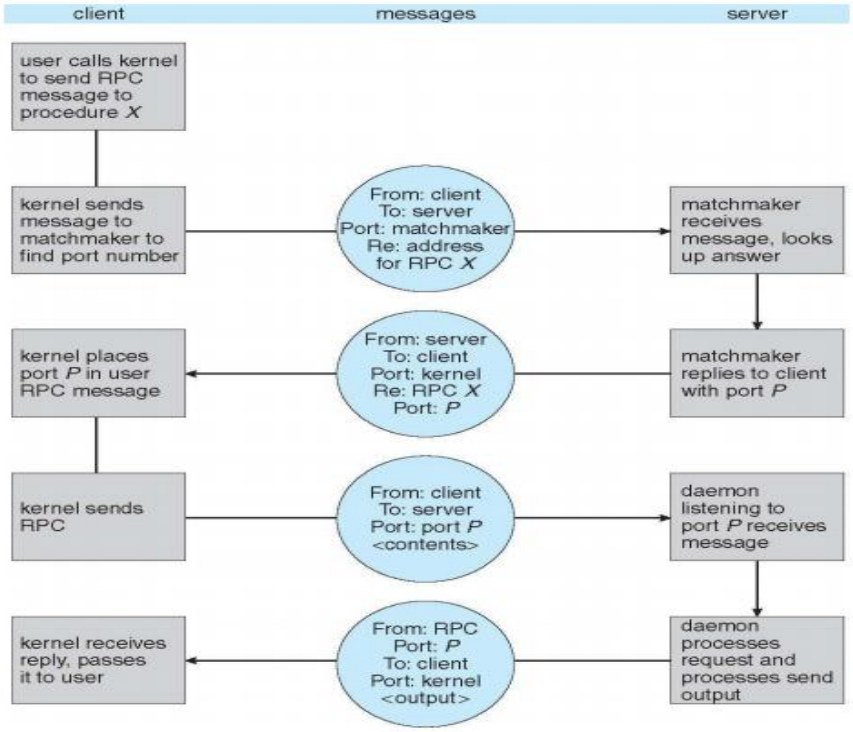
● Big-endian and little-endian

● Remote communication has more failure scenarios than local

● Messages can be delivered exactly once rather than at most once

● OS typically provides a rendezvous (or matchmaker) service to connect client and server

Execution of RPC



Pipes

● Acts as a conduit allowing two processes to communicate

● Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.

● Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

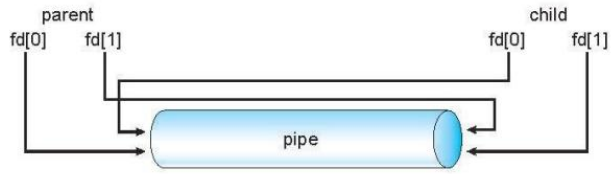
● Ordinary Pipes allow communication in standard producer-consumer style

● 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



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

Process Synchronization

- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples

Back Ground

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Bounded Bounded-Buffer – Producer &Consumer

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

Consumer

```
item nextConsumed;
```

```
while (1) {
while (in == out)
; /* do nothing */
nextConsumed = buffer[out];
out = (out + 1) %
BUFFER_SIZE;
}
```

Race Condition

count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1 = count {register1 = 5}

S1: producer execute register1 = register1 + 1 {register1 = 6}

S2: consumer execute register2 = count {register2 = 5}

S3: consumer execute register2 = register2 - 1 {register2 = 4}

S4: producer execute count = register1 {count = 6 }

S5: consumer execute count = register2 {count = 4}

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process Pi is executing in its critical section, hen no other processes can be executing in their critical sections

2. **Progress** - If no process is executing in its critical section and here exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed

- No assumption concerning relative speed of the N processes

Peterson's Solution

Two process solution

- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

Algorithm for Process Pi

```
do{
flag[i]=TRUE;
 turn = j;
 while(flag[j] && turn == j);
        CRITICAL SECTION
 flag[i] = FALSE;
 REMAINDER SECTION
 } while (TRUE);
```

Synchronization Hardware

- Many systems provide hardware support for critical section code „
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

TestAndSet Instruction

Definition:

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

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {
while ( TestAndSet (&lock ))
; /* do nothing
// critical section
lock = FALSE;
// remainder section
} while ( TRUE);
```

Mutual-exclusion implementation with the compare and swap().

```
int compare and swap(int *value, int expected, int new value) {
int temp = *value;
if (*value == expected)
*value = new value;
return temp;
}
```

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);

Mutual-exclusion implementation with the compare and swap() instruction.

Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore S – integer variable.
- Two standard operations modify S: wait() and signal()

  Originally called P() and V()
- Less complicated.
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    while S <= 0
    ; // no-op
    S--;
    }
  - signal (S) {
    S++;
    }

Semaphore as General Synchronization Tool

Counting semaphore – integer value can range over an unrestricted domain „
Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement. Also known as mutex locks „

Can implement a counting semaphore S as a binary semaphore „
Provides mutual exclusion

Semaphore S; // initialized to 1
wait (S);

Critical Section
signal (S);

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
- Could now have busy waiting in critical section implementation

  But implementation code is short

  Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:

value (of type integer)

pointer to next record in the list

Two operations:

block – place the process invoking the operation on the appropriate waiting queue.

wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Implementation of wait:

```
wait (S){
        value--;
        if (value < 0) {
        add this process to waiting queue
        block(); }
}
```

Implementation of signal:

```
Signal (S){
                value++;
```

```
        if (value <= 0) {
        remove a process P from the waiting queue
        wakeup(P); }
}
```

Deadlock and Starvation

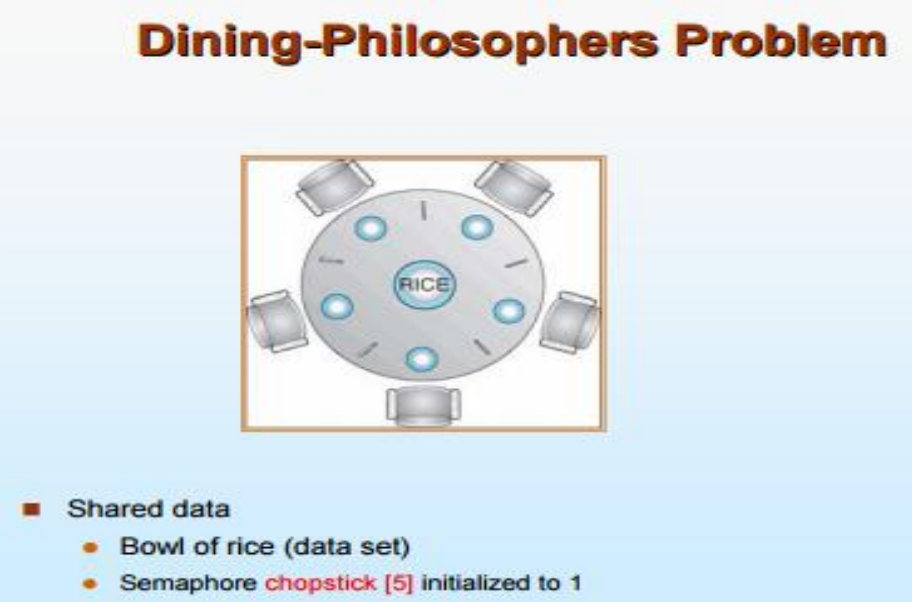
- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P0	P1
wait (S);	wait (Q);
wait (Q);	wait (S);
. .	
. .	
. .	
signal (S);	signal (Q);
signal (Q);	signal (S);

Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem



The structure of Philosopher i:

```
Do {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );
// eat
        ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
// think
} while (true) ;
```

Problems with Semaphores

Correct use of semaphore operations:

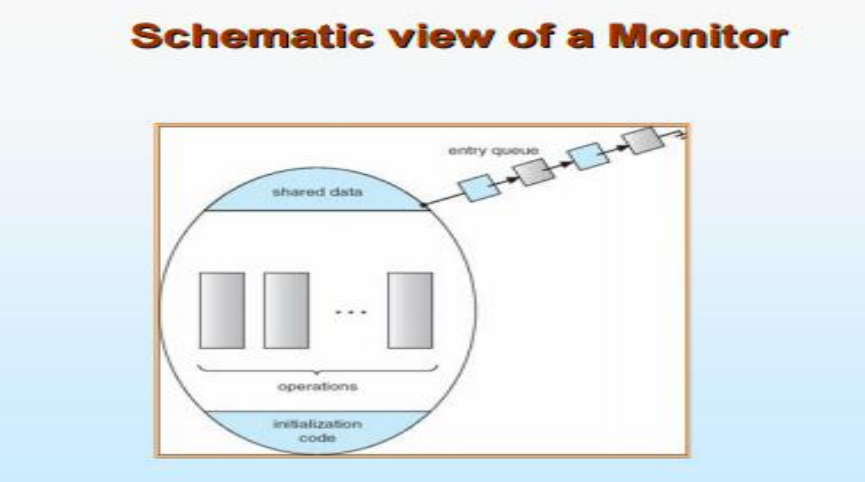
- signal (mutex) …. wait (mutex)
- wait (mutex) … wait (mutex)
- Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time monitor monitor-name

```
{
// shared variable declarations
procedure P1 (…) { …. }
…
```

procedure Pn (…) {……}
Initialization code (….) { … }
…
}
}

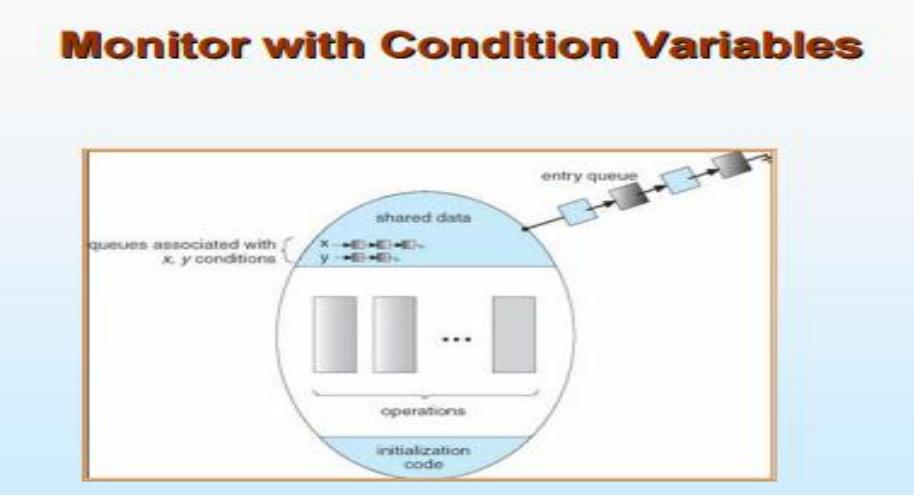


Condition Variables

- „ condition x, y;
- „ Two operations on a condition variable:

  x.wait () – a process that invokes the operation is suspended.

  x.signal () – resumes one of processes (if any) that invoked x.wait ()



Solution to Dining Philosophers

monitor DP
{
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 left and right neighbors
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;
}
}
```