TOPIC1: Process Concept:

TOPIC-2 :Process scheduling,

2:1 Basic concepts,

2:2 Scheduling criteria,

2:3 Scheduling algorithms,

2:4 Multiple processor scheduling

TOPIC-3 :Operations on processes,

TOPIC-4 Inter-process communication,

TOPIC-5:Communication in client server systems.

TOPIC-6:Multithreaded Programming:

6:1 : Multithreading models,

6:2: Thread libraries,

6:3 Threading issues.

6:4 Thread scheduling.

TOPIC:7

7:1 Inter-process Communication: Race conditions,

7:2 Critical Regions,

7:3 Mutual exclusion with busy waiting,

7:4 Sleep and wakeup, Semaphores,

7:5 Mutexes,

7:6 Monitors,

7:7 Message passing,

7:8 Barriers,

7:9Classical IPC Problems - Dining philosophers problem, Readers and writers problem

TOPIC:1 Process Concept: Process scheduling, Operations on processes, Inter-process communication, Communication in client server systems.

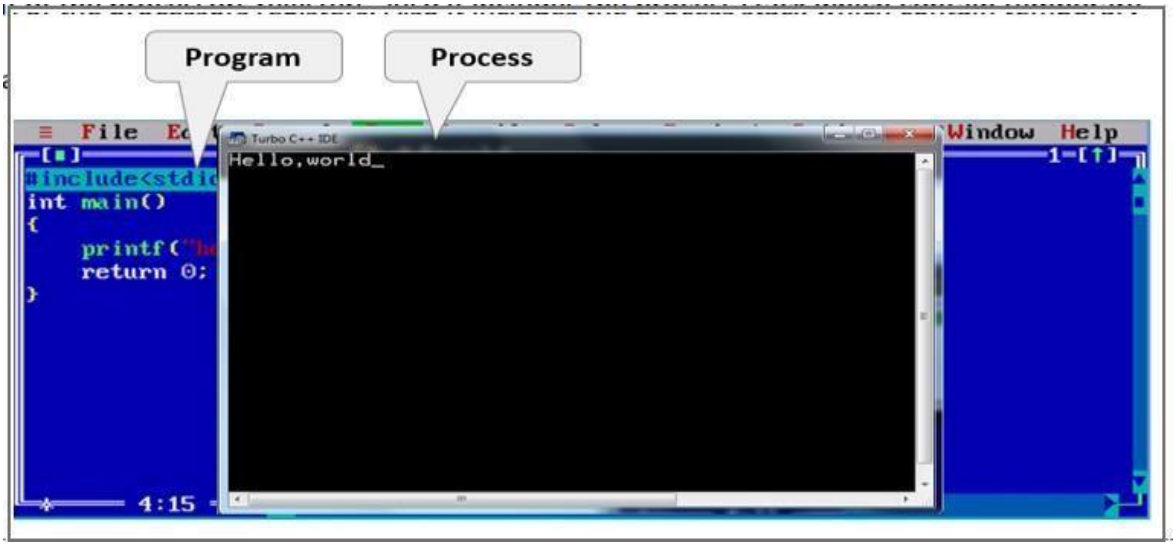
Process

A process is a program at the time of execution.

OR

Process: A process or task is an instance of a program in execution. The execution of a process must programs in a sequential manner. At any time at most one instruction is executed.

The process includes the current activity as represented by the value of the program counter and the content of the processors registers. Also it includes the process stack which contain temporary data (such as method parameters return address and local variables) & a data section which contain global variables.



Differences between Process and Program

Process	Program
Process is a dynamic object	Program is a static object
Process is sequence of instruction execution	Program is a sequence of instructions
Process loaded in to main memory	Program loaded into secondary storage devices
Time span of process is limited	Time span of program is unlimited
Process is a active entity	Program is a passive entity

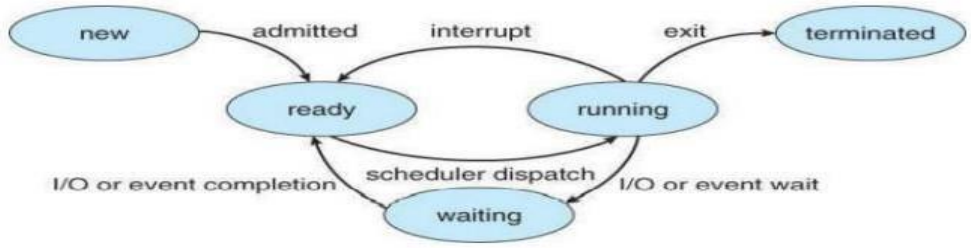
1.1

Process state: As a process executes, it changes state. The state of a process is defined by the correct activity of that process. Each process may be in one of the following states.

- **New:** The process is being created.
- **Ready:** The process is waiting to be assigned to a processor.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur.
- **Terminated:** The process has finished execution.

Only one process can be running in any processor at any time, But many process may be in ready and waiting states. The ready processes are loaded into a "ready queue".

Diagram of process state



a) **New ->Ready** : OS creates process and prepares the process to be executed,thenOSmoved the process into readyqueue.

b) **Ready->Running** : OS selects one of the Jobs from ready Queue and move themfro ready to Running.

c) **Running->Terminated** : When the Execution of a process has Completed, OSterminatesthatprocess from running state. Sometimes OS terminates the process for someother reasons including Time exceeded, memory unavailable, access violation, protection Error, I/O failure and soon.

d) **Running->Ready** : When the time slot of the processor expired (or) If the processorreceivedanyinterrupt signal, the OS shifted Running -> ReadyState.

e) **Running -> Waiting** : A process is put into the waiting state, if the process need an event occur (or) an I/O Devicerequire.

f) **Waiting->Ready** : A process in the waiting state is moved to ready state when the eventforwhichit has beenCompleted.

1:2 PROCESS CONTROL BLOCK:[pcb]

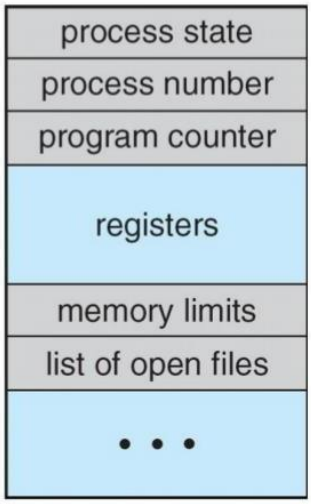
Process Control Block is a data structure that contains information of the process related to it. The process control block is also known as a task control block, entry of the process table, etc.

It is very important for process management as the data structuring for processes is done in terms of the PCB. It also defines the current state of the operating system.

Structure of the Process Control Block

The process control stores many data items that are needed for efficient process management. Some of these data items are explained with the help of the given diagram.

Each process is represented in the operating system by a process control block also called a task cont rol block.(PCB) Also called a task control block.



The following are the data items −

Process State
This specifies the process state i.e. new, ready, running, waiting or terminated.

Process Number
This shows the number of the particular process.

Program Counter
This contains the address of the next instruction that needs to be executed in the process.

Registers
This specifies the registers that are used by the process. They may include accumulators, index registers, stack pointers, general purpose registers etc.

List of Open Files

These are the different files that are associated with the process

CPU Scheduling Information

The process priority, pointers to scheduling queues etc. is the CPU scheduling information that is contained in the PCB. This may also include any other scheduling parameters.

Memory Management Information

The memory management information includes the page tables or the segment tables depending on the memory system used. It also contains the value of the base registers, limit registers etc.

I/O Status Information

This information includes the list of I/O devices used by the process, the list of files etc.

Accounting information

The time limits, account numbers, amount of CPU used, process numbers etc. are all a part of the PCB accounting information.

Location of the Process Control Block

The process control block is kept in a memory area that is protected from the normal user access. This is done because it contains important process information. Some of the operating.systems place the PCB at the beginning of the kernel stack for the process as it is a safe location.

TOPIC:2 Process scheduling:

PROCESS SCHEDULING:

CPU is always busy in **Multiprogramming**. Because CPU switches from one job to another job. But in **simple computers** CPU sit idle until the I/O request granted.

scheduling is a important OS function. All resources are scheduled before use.(cpu, memory, devices…..)

Process scheduling is an essential part of a 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

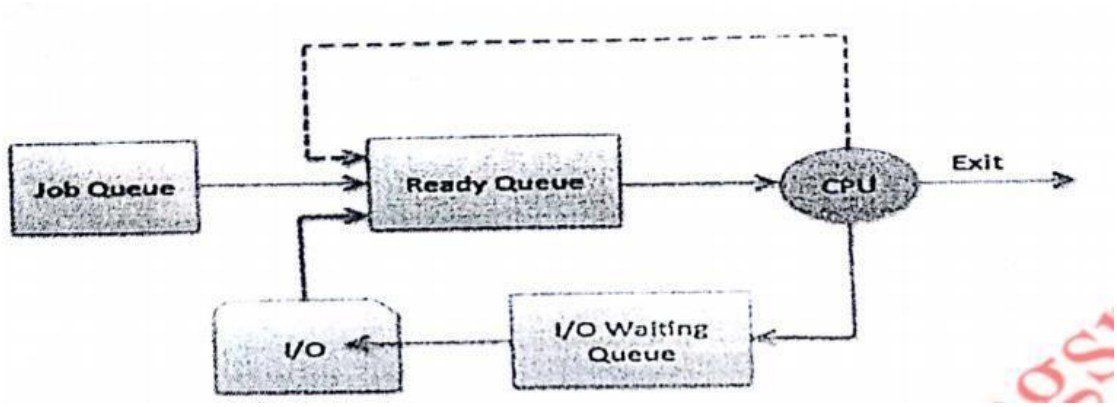
Scheduling Objectives

Maximize throughput.
Maximize number of users receiving acceptable response times.
Be predictable.
Balance resource use.
Avoid indefinite postponement.
Enforce Priorities.
Give preference to processes holding key resources

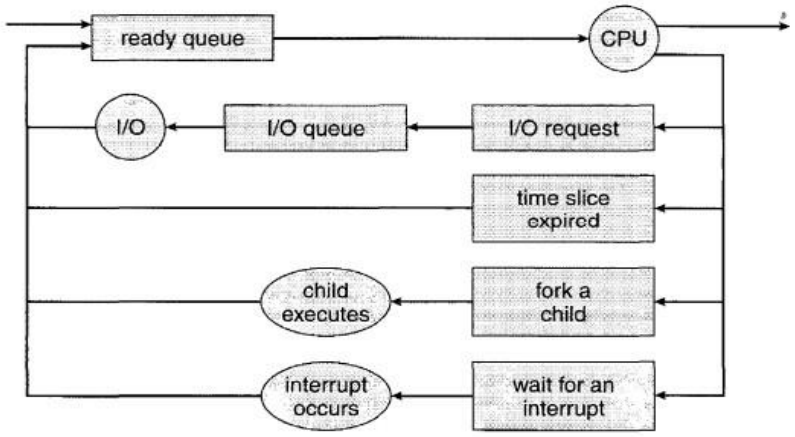
SCHEDULING QUEUES: people live in rooms. Process are present in rooms knows as queues. There are 3types

1. **job queue**: when processes enter the system, they are put into a **job queue**, which consists all processes in the system. Processes in the job queue reside on mass storage and await the allocation of main memory.
2. **ready queue**: if a process is present in main memory and is ready to be allocated to cpu forexecution, is kept in **readyqueue.**
3. **device queue**: if a process is present in waiting state (or) waiting for an i/o event to complete is said to bein device queue.(or)

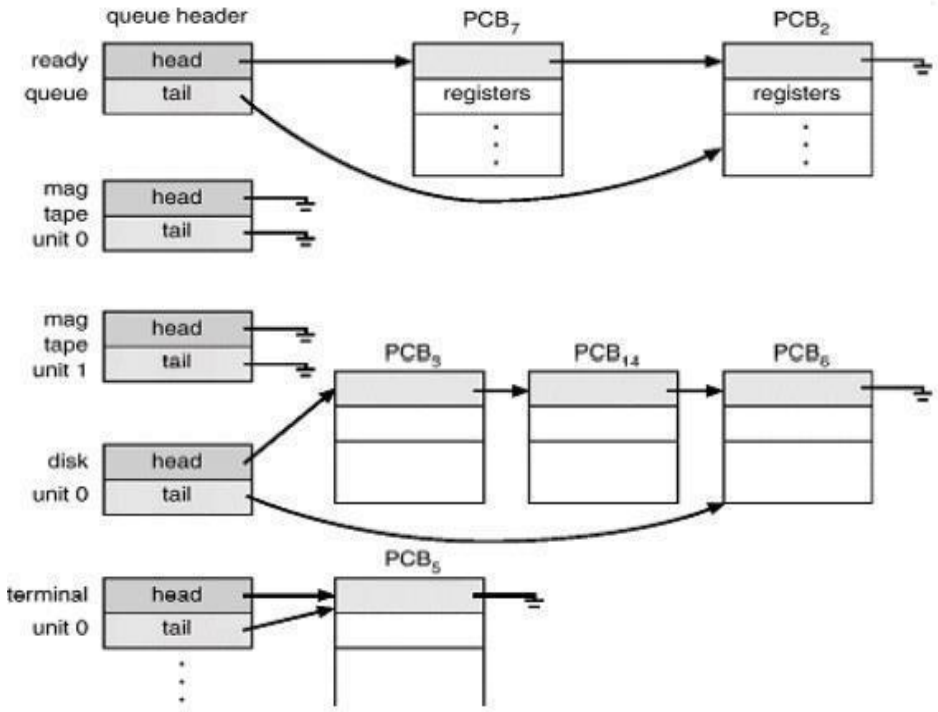
The processes waiting for a particular I/O device is called device queue.



Scheduling queues: As processes enter the system, they are put into a job queue. This queue consists of all process in the system. The process that are residing in main memory and are ready & waiting to execute or kept on a list called ready queue.



This queue is generally stored as a linked list. A ready queue header contains pointers to the first & final PCB in the list. The PCB includes a pointer field that points to the next PCB in the ready queue. The lists of processes waiting for a particular I/O device are kept on a list called device queue. Each device has its own device queue. A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution & is given the CPU.



SCHEDULERS:

A process migrates between the various scheduling queues throughout its life-time purposes. The OS must select for scheduling processes from these queues in some fashion. This selection process is carried out by the appropriate scheduler. In a batch system, more processes are submittedand then executed immediately. So these processes are spooled to a mass storage device like disk, where they are kept for later execution.

Schedulers : There are 3 schedulers

1. Long term scheduler.
2. Medium term scheduler
3. Short term scheduler.

Scheduler duties:

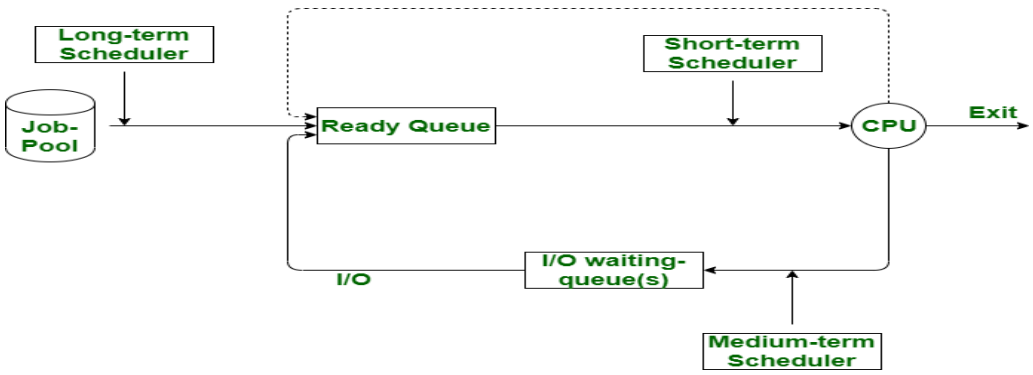
- Maintains the queue.
- Select the process from queues assign to CPU.

Types of schedulers

1. Long term scheduler:
select the jobs from the job pool and loaded these jobs into main memory (ready queue).
Long term scheduler is also called job scheduler.

2. Short term scheduler:
select the process from ready queue, and allocates it to the cpu.
If a process requires an I/O device, which is not present available then process enters device queue.
short term scheduler maintains ready queue, device queue. Also called as cpu scheduler.

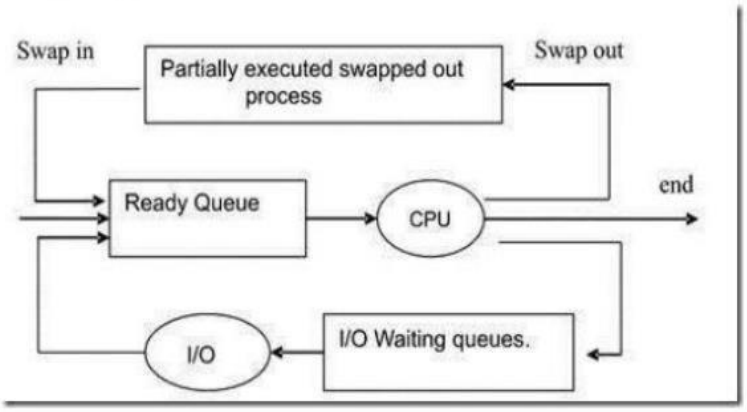
3. Medium term scheduler: if process request an I/O device in the middle of the execution, then the process removed from the main memory and loaded into the waiting queue. When the I/O operation completed, then the job moved from waiting queue to ready queue. These two operations performed by medium term scheduler.



1. **Long term scheduler:** Long term scheduler selects process from the disk & loads them into memory for execution. It controls the degreeof multi-programming i.e. no. of processes in memory. It executes less frequently than other schedulers. If the degree of multiprogramming is stable than the average rate of process creation is equal to the average departure rate of processes leaving the system. So, the long term scheduler is needed to be invoked only when a process leaves the system. Due to longer intervals between executions it can afford to take more time to decide which process should be selected for execution. Most processes in the CPU are either I/O bound or CPU bound. An I/O bound process (an interactive 'C' program is one that spends most of its time in I/O operation than it spends in doing I/O operation. A CPU bound process is one that spends more of its time in doing computations than I/O operations (complex sorting program). It is important that the long term scheduler should select a good mix of I/O bound & CPU bound processes.

2. **Short - term scheduler:** The short term scheduler selects among the process that are ready to execute & allocates the CPU to one of them. The primary distinction between these two schedulers is the frequency of their execution. The short-term scheduler must select a new process for the CPU quite frequently. It must execute at least one in 100ms. Due to the short duration of time between executions, it must be very fast.

3. **Medium - term scheduler:** some operating systems introduce an additional intermediate level of scheduling known as medium - term scheduler. The main idea behind this scheduler is that sometimes it is advantageous to remove processes from memory & thus reduce the degree of multiprogramming. At some later time, the process can be reintroduced into memory & its execution can be continued from where it had left off. This is called as swapping. The process is swapped out & swapped in later by medium term scheduler. Swapping is necessary to improve theprocess miss or due to some change in memory requirements, the available memory limit is exceeded which requires some memory to be freed up.



DIFFERENCE BETWEEN CPU SHEDLULERS

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: Assume, main memory contains more than one process. If cpu is executing a process, if time expires or if a high priority process enters into main memory, then the scheduler saves information about current process in the PCB and switches to execute the another process. The concept of moving CPU by scheduler from one process to other process is known as context switch.

Non-Preemptive Scheduling: **CPU** is assigned to one process, CPU do not release until the competition of that process. The CPU will assigned to some other process only after the previous process has finished.

Preemptive scheduling: here CPU can release the processes even in the middle of the execution. CPU received a signal from process p2. OS compares the priorities of p1 ,p2. If p1>p2, CPU continues the execution of p1. If p1<p2 CPU preempt p1 and assigned to p2.

Dispatcher: The main job of dispatcher is switching the cpu from one process to another process. Dispatcher connects the cpu to the process selected by the short term scheduler.

Dispatcher latency: The time it takes by the dispatcher to stop one process and start another process is known as dispatcher latency. If the dispatcher latency is increasing, then the degree of multiprogramming decreases.

SCHEDULING CRITERIA:

1. **Throughput**: how many jobs are completed by the cpu with in a timeperiod.
2. **Turn around time** : The time interval between the submission of the process and time of the completion is turn around time.

TAT = Waiting time in ready queue + executing time + waiting time in waiting queue for I/O.

3. **Waiting time**: The time spent by the process to wait for cpu to beallocated.
4. **Response time**: Time duration between the submission and firstresponse.
5. **Cpu Utilization**: CPU is costly device, it must be kept as busy aspossible.

Eg: CPU efficiency is 90% means it is busy for 90 units, 10 units idle.

What is Context Switching in Operating System

In the Operating System, there are cases when you have to bring back the process that is in the running state to some other state like ready state or wait/block state. If the running process wants to perform some I/O operation, then you have to remove the process from the running state and then put the process in the I/O queue.

Sometimes, the process might be using a round-robin scheduling algorithm where after every fixed time quantum, the process has to come back to the ready state from the running state. So, these process switchings are done with the help of Context Switching. In this blog, we will learn about the concept of Context Switching in the Operating System and we will also learn about the advantages and disadvantages of Context Switching. So, let's get started.

What is Context Switching?

A context switching is a process that involves switching of the CPU from one process or task to another. In this phenomenon, the execution of the process that is present in the running state is suspended by the kernel and another process that is present in the ready state is executed by the CPU.

It is one of the essential features of the multitasking operating system. The processes are switched so fastly that it gives an illusion to the user that all the processes are being executed at the same time.

But the context switching process involved a number of steps that need to be followed. You can't directly switch a process from the running state to the ready state. You have to save the context of that process.

If you are not saving the context of any process P then after some time, when the process P comes in the CPU for execution again, then the process will start executing from starting. But in reality, it should continue from that point where it left the CPU in its previous execution.

So, the context of the process should be saved before putting any other process in the running state. A context is the contents of a CPU's registers and program counter at any point in time.

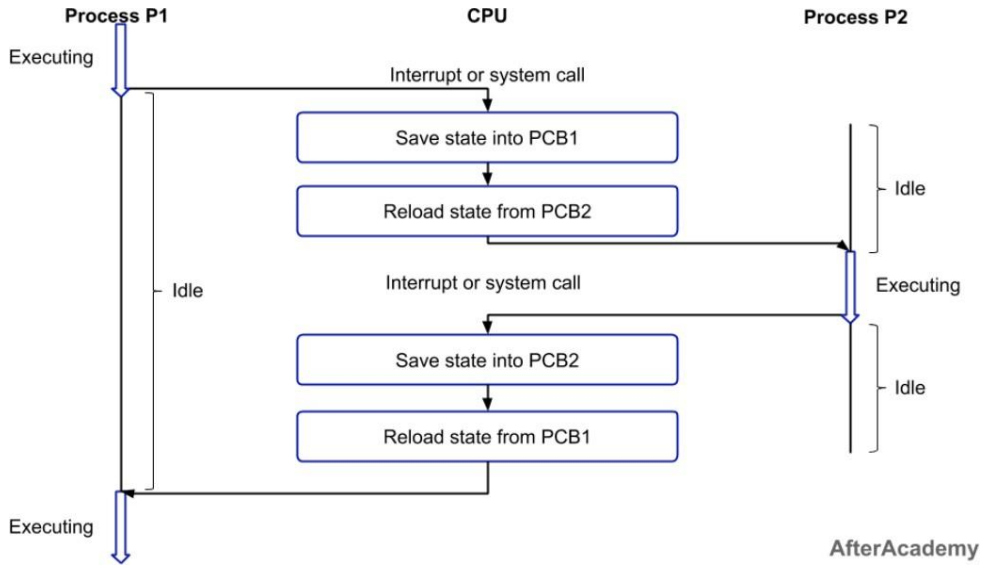
Context switching can happen due to the following reasons:•

When a process of high priority comes in the ready state. In this case, the execution of the running process should be stopped and the higher priority process should be given the CPU for execution. When an interruption occurs then the process in the running state should be stopped and the CPU should handle the interrupt before doing something else.

When a transition between the user mode and kernel mode is required then you have to perform the context switching.

Steps involved in Context Switching

The process of context switching involves a number of steps. The following diagram depicts the process of context switching between the two processes P1 and P2.



In the above figure, you can see that initially, the process P1 is in the running state and the process P2 is in the ready state. Now, when some interruption occurs then you have to switch the process P1 from running to the ready state after saving the context and the process P2 from ready to running state. The following steps will be performed:

1. Firstly, the context of the process P1 i.e. the process present in the running state will be saved in the Process Control Block of process P1 i.e. PCB1.
2. Now, you have to move the PCB1 to the relevant queue i.e. ready queue, I/O queue, waiting queue, etc.
3. From the ready state, select the new process that is to be executed i.e. the process P2.
4. Now, update the Process Control Block of process P2 i.e. PCB2 by setting the process state to running. If the process P2 was earlier executed by the CPU, then you can get the position of last executed instruction so that you can resume the execution of P2.
5. Similarly, if you want to execute the process P1 again, then you have to follow the same steps as mentioned above(from step 1 to 4).

For context switching to happen, two processes are at least required in general, and in the case of the round-robin algorithm, you can perform context switching with the help of one process only.

The time involved in the context switching of one process by other is called the Context Switching Time

2.1 Process sheduling algorithems or cpu sheduling algorithems.

CPU Scheduling Algorithm:

CPU Scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated first to the CPU. There are four types of CPU scheduling that exist.

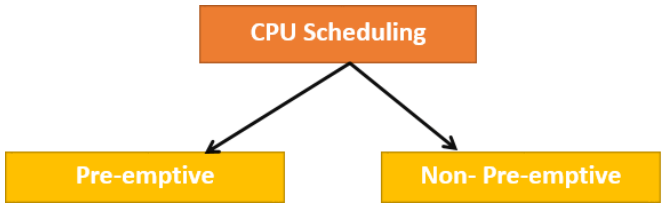
A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms.

●

CPU Scheduling is a process of determining which process will own CPU for execution while another process is on hold. The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS at least select one of the processes available in the ready queue for execution. The selection process will be carried out by the CPU scheduler. It selects one of the processes in memory that are ready for execution

Types of CPU Scheduling

Here are two kinds of Scheduling methods:



Preemptive Scheduling

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

Non-Preemptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling.

When scheduling is Preemptive or Non-Preemptive?

To determine if scheduling is preemptive or non-preemptive, consider these four parameters:

1.
A process switches from the running to the waiting state.
2.
Specific process switches from the running state to the ready state.
3.
Specific process switches from the waiting state to the ready state.
4.
Process finished its execution and terminated.

Only conditions 1 and 4 apply, the scheduling is called non- preemptive.
All other scheduling are preemptive.

Important CPU scheduling Terminologies

- **Burst Time/Execution Time:** It is a time required by the process to complete execution. It is also called running time.
- **Arrival Time:** when a process enters in a ready state
- **Finish Time:** when process complete and exit from a system
- **Multiprogramming:** A number of programs which can be present in memory at the same time.
- **Jobs:** It is a type of program without any kind of user interaction.
- **User:** It is a kind of program having user interaction.
- **Process:** It is the reference that is used for both job and user.
- **CPU/IO burst cycle:** Characterizes process execution, which alternates between CPU and I/O activity. CPU times are usually shorter than the time of I/O.

CPU Scheduling Criteria

A CPU scheduling algorithm tries to maximize and minimize the following:

What is Dispatcher?

It is a module that provides control of the CPU to the process. The Dispatcher should be fast so that it can run on every context switch. Dispatch latency is the amount of time needed by the CPU scheduler to stop one process and start another.

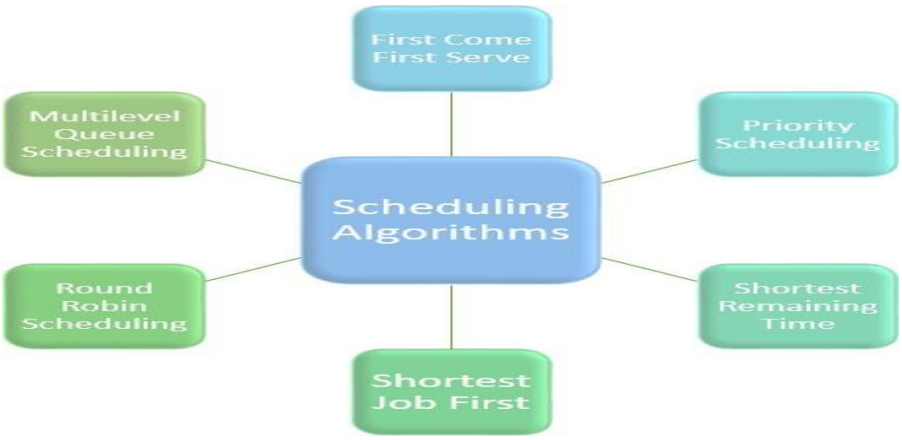
Functions performed by Dispatcher:

- Context Switching
- Switching to user mode
- Moving to the correct location in the newly loaded program.

Types of CPU scheduling Algorithm

There are mainly six types of process scheduling algorithms

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



Sheduling algorithems

1. First Come First Serve

First Come First Serve is the full form of FCFS. It is the easiest and most simple CPU scheduling algorithm. In this type of algorithm, the process which requests the CPU gets the CPU allocation first. This scheduling method can be managed with a FIFO queue.

As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue. So, when CPU becomes free, it should be assigned to the process at the beginning of the queue.

Characteristics of FCFS method:

•
It offers non-preemptive and pre-emptive scheduling algorithm.
•
Jobs are always executed on a first-come, first-serve basis
•
It is easy to implement and use.
•

However, this method is poor in performance, and the general wait time is quite high.

First Come First Serve (FCFS)

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

In FCFS Scheduling,

- The process which arrives first in the ready queue is firstly assigned the CPU.
- In case of a tie, process with smaller process id is executed first.
- It is always non-preemptive in nature.

divantages-

- It is simple and easy to understand.
- It can be easily implemented using queue data structure.
- It does not lead to starvation.

Disadvantages-It does not consider the priority or burst time of the processes.

- It suffers from **convoy effect**.

Convoy Effect

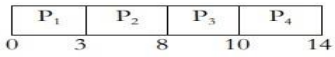
In convoy effect,

- Consider processes with higher burst time arrived before the processes with smaller burst time.

Process	CPU time
P_1	3
P_2	5
P_3	2
P_4	4

Using FCFS algorithm find the average waiting time and average turnaround time if the order is P_1, P_2, P_3, P_4.

Solution: If the process arrived in the order P_1, P_2, P_3, P_4 then according to the FCFS the Gantt chart will be:



The waiting time for process $P_1 = 0$, $P_2 = 3$, $P_3 = 8$, $P_4 = 10$ then the turnaround time for process $P_1 = 0 + 3 = 3$, $P_2 = 3 + 5 = 8$, $P_3 = 8 + 2 = 10$, $P_4 = 10 + 4 = 14$.

Then average waiting time = $(0 + 3 + 8 + 10)/4 = 21/4 = 5.25$

Average turnaround time = $(3 + 8 + 10 + 14)/4 = 35/4 = 8.75$

The FCFS algorithm is non preemptive means once the CPU has been allocated to a process then the process keeps the CPU until the release the CPU either by terminating or requesting I/O.

2. SJF Scheduling | SRTF | CPU Scheduling.

In SJF Scheduling,

- Out of all the available processes, CPU is assigned to the process having smallest burst time
- In case of a tie, it is broken by **FCFS Scheduling**.

- SJF Scheduling can be used in both preemptive and non-preemptive mode.
- Preemptive mode of Shortest Job First is called as **Shortest Remaining Time First (SRTF)**.

Advantages-

- SRTF is optimal and guarantees the minimum average waiting time.
- It provides a standard for other algorithms since no other algorithm performs better than it.

Disadvantages-

- It can not be implemented practically since burst time of the processes can not be known in advance.
- It leads to starvation for processes with larger burst time.
- Priorities can not be set for the processes.
- Processes with larger burst time have poor response time.

Shortest Remaining Time

The full form of SRT is Shortest remaining time. It is also known as SJF preemptive scheduling. In this method, the process will be allocated to the task, which is closest to its completion. This method prevents a newer ready state process from holding the completion of an older process.

characteristics of SRT scheduling method:

•

This method is mostly applied in batch environments where short jobs are required to be given preference.

•

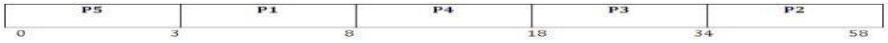
This is not an ideal method to implement it in a shared system where the required CPU time is unknown.

• Associate with each process as the length of its next CPU burst. So that operating system usethese lengths, which helps to schedule the process with the shortest possible time.

2) Shortest Job First Scheduling (SJF):

Which process having the smallest CPU burst time, CPU is assigned to that process . If two process having the same CPU burst time, FCFS is used.

PROCESS	CPU BURST TIME
P1	5
P2	24
P3	16
P4	10
P5	3



P5 having the least CPU burst time (3ms). CPU assigned to that (P5). After completion of P5 short term scheduler search for nest (P1)…….

Average Waiting Time :

Average Waiting Time :

Formula = Staring Time - Arrival Time
waiting Time for P1 => 3-0 = 3
waiting Time for P2 -> 34-0 = 34
waiting Time for P3 => 18-0 = 18
waiting Time for P4 =>8-0=8
waiting time for P5=0
Average waiting time => (3+34+18+8+0)/5 => 63/5 =12.6 ms

Average Turn Around Time :

Formula = waiting Time + burst Time

Turn Around Time for P1 => 3+5 =8
Turn Around for P2 => 34+24 =58
Turn Around for P3 => 18+16 = 34

Turn Around Time for P4 => 8+10 =18
Turn Around Time for P5 => 0+3 = 3
Average Turn around time => (8+58+34+18+3)/5 => 121/5 = 24.2 ms

Average Response Time :

Average Response Time :

Formula : First Response - Arrival Time

First Response time for P1 =>3-0 = 3
First Response time for P2 => 34-0 = 34
First Response time for P3 => 18-0 = 18
First Response time for P4 => 8-0 = 8
First Response time for P5 = 0
Average Response Time => (3+34+18+8+0)/5 => 63/5 = 12.6 ms
SJF is Non primitive scheduling algorithm
Advantages : Least average waiting time
Least average turn around time Least average response time
Average waiting time (FCFS) = 25 ms
Average waiting time (SJF) = 12.6 ms 50% time saved in SJF.
Disadvantages:

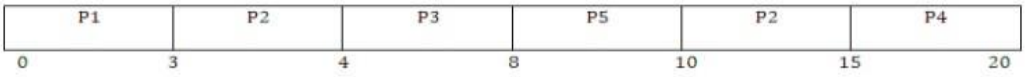
- Knowing the length of the next CPU burst time is difficult.
- Aging (Big Jobs are waiting for long time for CPU)

3) Shortest Remaining Time First (SRTF):

This is primitive scheduling algorithm.

Short term scheduler always chooses the process that has term shortest remaining time. When a new process joins the ready queue , short term scheduler compare the remaining time of executing process and new process. If the new process has the least CPU burst time, The scheduler selects that job and connect to CPU. Otherwise continue the old process.

PROCESS	BURST TIME	ARRIVAL TIME
P1	3	0
P2	6	2
P3	4	4
P4	5	6
P5	2	8



P1 arrives at time 0, P1 executing First , P2 arrives at time 2. Compare P1 remaining time and P2 (3-2 = 1) and 6. So, continue P1 after P1, executing P2, at time 4, P3 arrives, compare P2 remaining time (6-1=5) and 4 (4<5) .So, executing P3 at time 6, P4 arrives. Compare P3 remaining time and P4 (4-2=2) and 5 (2<5). So, continue P3 , after P3, ready queue consisting P5 is the least out of three. So execute P5, next P2, P4.

FORMULA : Finish time - Arrival
Time Finish Time for P1 => 3-0 = 3
Finish Time for P2 => 15-2 = 13
Finish Time for P3 => 8-4 =4
Finish Time for P4 => 20-6 = 14
Finish Time for P5 => 10-8 = 2

Average Turn around time –> 36/5 – 7.2 ms.

Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.

Priority Scheduling-

In Priority Scheduling,

- Out of all the available processes, CPU is assigned to the process having the highest priority.
- In case of a tie, it is broken by FCFS Scheduling.

- Priority Scheduling can be used in both preemptive and non-preemptive mode.

Advantages-

- It considers the priority of the processes and allows the important processes to run first
- Priority scheduling in preemptive mode is best suited for real time operating system.

Disadvantages-

- Processes with lesser priority may starve for CPU.
- There is no idea of response time and waiting time.

Important Notes-

Note-01:

- The waiting time for the process having the highest priority will always be zero in preemptive mode.
- The waiting time for the process having the highest priority may not be zero in non-preemptive mode.

Note-02:

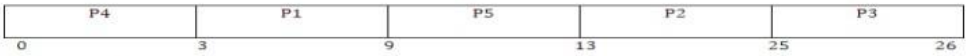
Priority scheduling in preemptive and non-preemptive mode behaves exactly same under following conditions-

- The arrival time of all the processes is same
- All the processes become available

5) PRIORITY SCHEDULING :

PROCESS	BURST TIME	PRIORITY
P1	6	2
P2	12	4
P3	1	5
P4	3	1
P5	4	3

P4 has the highest priority. Allocate the CPU to process P4 first next P1, P5, P2, P3.



AVERAGE WAITING TIME :

Waiting time for P1 => 3-0 =3
Waiting time for P2 => 13-0 = 13
Waiting time for P3 => 25-0 = 25
Waiting time for P4 => 0
Waiting time for P5 => 9-0 =9

Average waiting time => (3+13+25+0+9)/5 = 10 ms

AVERAGE TURN AROUND TIME :

Turn around time for P1 =>3+6 = 9
Turn around time for P2 => 13+12= 25
Turn around time for P3 => 25+1 = 26
Turn around time for P4 => 0+3= 3
Turn around time for P5 => 9+4 = 13

Average Turn around time => (9+25+26+3+13)/5 = 15.2 ms

Disadvantage: Starvation

Starvation means only high priority process are executing, but low priority process are waiting for the CPU for the longest period of the time.

Multiple – processor scheduling:
When multiple processes are available, then the scheduling gets more complicated, because there is more than one CPU which must be kept busy and in effective use at all times.
Load sharing resolves around balancing the load between multiple processors. Multi processor systems may be heterogeneous (It contains different kinds of CPU's) (or) Homogeneous(all the same kind of CPU).

1) Approaches to multiple-processor scheduling

a)Asymmetric multiprocessing
One processor is the master, controlling all activities and running all kernel code, while the other runs only user code.

b)Symmetric multiprocessing:
Each processor schedules its own job. Each processor may have its own private queue of ready processes.

4. Round-Robin Scheduling

Round robin is the oldest, simplest scheduling algorithm. The name of this algorithm comes from the round-robin principle, where each person gets an equal share of something in turn. It is mostly used for scheduling algorithms in multitasking. This algorithm method helps for starvation free execution of processes.

Characteristics of Round-Robin Scheduling

- Round robin is a hybrid model which is clock-driven
- Time slice should be minimum, which is assigned for a specific task to be processed. However, it may vary for different processes.
- It is a real time system which responds to the event within a specific time limit.

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted anc other process executes for a given time period.
- Context switching is used to save states of preempted processes.

Round Robin(RR) scheduling algorithm is mainly designed for time-sharing systems. This algorithm is similar to FCFS scheduling, but in Round Robin(RR) scheduling, preemption is added which enables the system to switch between processes.

- A fixed time is allotted to each process, called a **quantum**, for execution.
- Once a process is executed for the given time period that process is preempted and another process executes for the given time period.
- Context switching is used to save states of preempted processes.
- This algorithm is simple and easy to implement and the most important is thing is this algorithm is starvation-free as all processes get a fair share of CPU.

- It is important to note here that the length of time quantum is generally from 10 to 100 milliseconds in length.

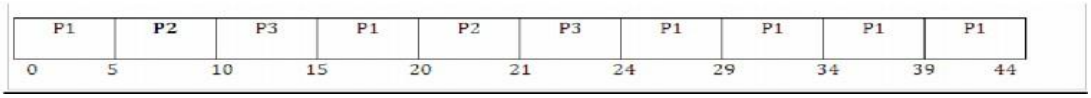
Some important characteristics of the Round Robin(RR) Algorithm are as follows:

1. Round Robin Scheduling algorithm resides under the category of Preemptive Algorithms.
2. This algorithm is one of the oldest, easiest, and fairest algorithm.
3. This Algorithm is a real-time algorithm because it responds to the event within a specific time limit.
4. In this algorithm, the time slice should be the minimum that is assigned to a specific task that needs to be processed. Though it may vary for different operating systems.

4)ROUND ROBIN SCHEDULING ALGORITHM :

It is designed especially for time sharing systems. Here CPU switches between the processes. When the time quantum expired, the CPU switched to another job. A small unit of time, called a time quantum or time slice. A time quantum is generally from 10 to 100 ms. The time quantum is generally depending on OS. Here ready queue is a circular queue. CPU scheduler picks the first process from ready queue, sets timer to interrupt after one time quantum and dispatches the process.

PROCESS	BURST TIME
P1	30
P2	6
P3	8



AVERAGE WAITING TIME :

Waiting time for P1 => 0+(15-5)+(24-20) => 0+10+4 = 14
Waiting time for P2 => 5+(20-10) => 5+10 = 15
Waiting time for P3 => 10+(21-15) => 10+6 = 16
Average waiting time => (14+15+16)/3 = 15 ms.

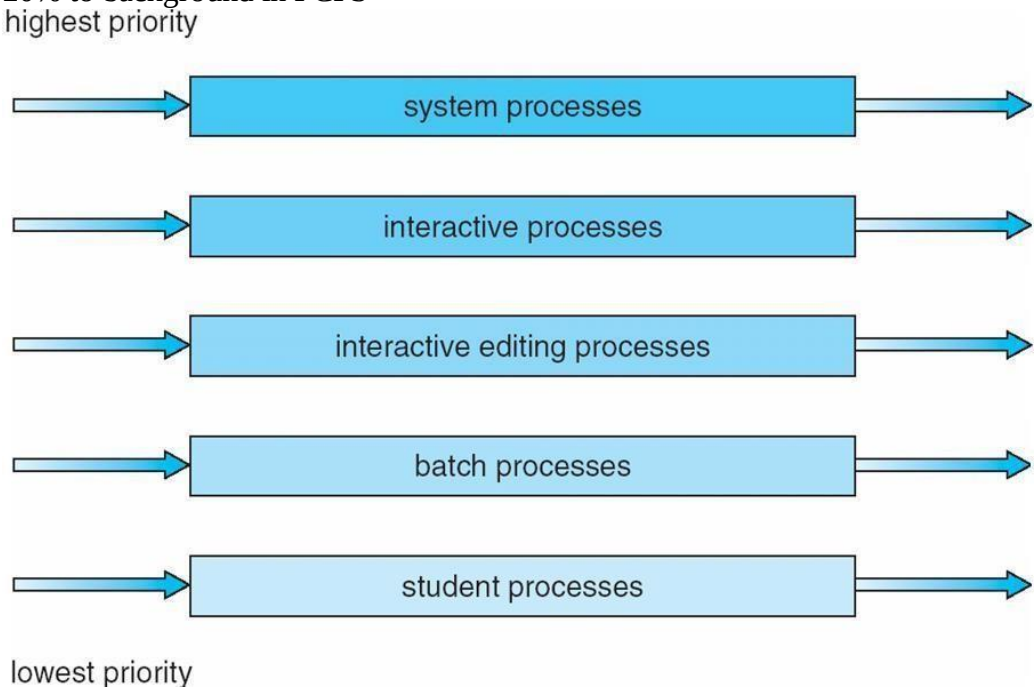
AVERAGE TURN AROUND TIME :

FORMULA : Turn around time = waiting time + burst Time
Turn around time for P1 => 14+30 =44
Turn around time for P2 => 15+6 = 21
Turn around time for P3 => 16+8 = 24
Average turn around time => (44+21+24)/3 = 29.66 ms

5. Multilevel Queue (MLQ) CPU Scheduling

Multilevel Queue

Ready queue is partitioned into separate queues:
foreground (interactive)
background (batch)
Each queue has its own scheduling algorithm
foreground – RR
background – FCFS
Scheduling must be done between the queues
Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes;
i.e., 80% to foreground in RR
20% to background in FCFS



6. Multilevel Feedback Queue

A process can move between the various queues; aging can be implemented this way
Multilevel-feedback-queue scheduler defined by the following parameters:
number of queues
scheduling algorithms for each queue
method used to determine when to upgrade a process
method used to determine when to demote a process
method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

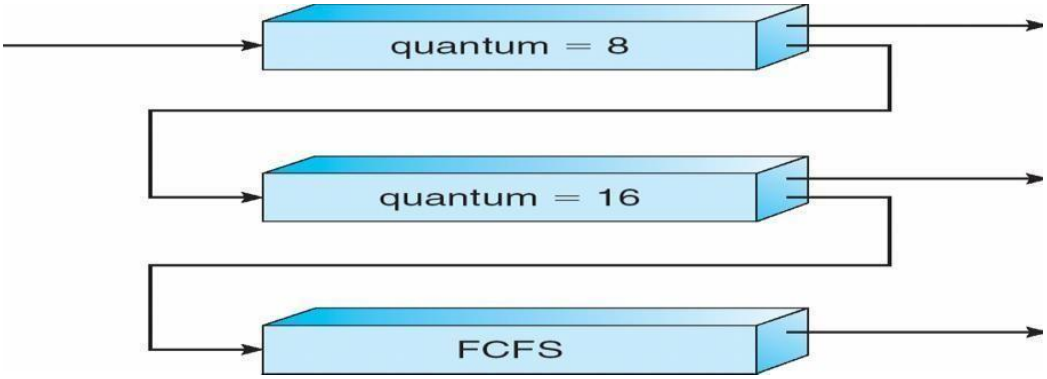
Three queues:
$Q0$ – RR with time quantum 8 milliseconds
$Q1$ – RR time quantum 16 milliseconds
$Q2$ – FCFS

Scheduling

A new job enters queue *Q0* which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue *Q*1.

At *Q*1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue *Q*2.

Multilevel Feedback Queues



Now let us suppose that queue 1 and 2 follow round robin with time quantum 4 and 8 respectively and queue 3 follow FCFS.One implementation of MFQS is given below –

1. When a process starts executing then it first enters queue 1.
2. In queue 1 process executes for 4 unit and if it completes in this 4 unit or it gives CPU for I/O operation in this 4 unit than the priority of this process does not change and if it again comes in the ready queue than it again starts its execution in Queue 1.
3. If a process in queue 1 does not complete in 4 unit then its priority gets reduced and it shifted to queue 2.
4. Above points 2 and 3 are also true for queue 2 processes but the time quantum is 8 unit.In a general case if a process does not complete in a time quantum than it is shifted to the lower priority queue.
5. In the last queue, processes are scheduled in FCFS manner.
6. A process in lower priority queue can only execute only when higher priority queues are empty.
7. A process running in the lower priority queue is interrupted by a process arriving in the higher priority queue.

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

Problems in the above implementation – A process in the lower priority queue can suffer from starvation due to some short processes taking all the CPU time.

Solution – A simple solution can be to boost the priority of all the process after

regular intervals and place them all in the highest priority queue.

What is the need of such complex Scheduling?

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

•
To optimize turnaround time algorithms like SJF is needed which require the running time of processes to schedule them. But the running time of the process is not known in advance. MFQS runs a process for a time quantum and then it can change its priority(if it is a long process). Thus it learns from past behavior of the

process and then predicts its future behavior.This way it tries to run shorter process first thus optimizing turnaround time.

•
MFQS also reduces the response time.

Example –

Consider a system which has a CPU bound process, which requires the burst time of 40 seconds.The multilevel Feed Back Queue scheduling algorithm is used and the queue time quantum ‘2’ seconds and in each level it is incremented by ‘5’ seconds.Then how many times the process will be interrupted and on which queue the process will terminate the execution?

Solution –

Process P needs 40 Seconds for total execution.
At Queue 1 it is executed for 2 seconds and then interrupted and shifted to queue 2.
At Queue 2 it is executed for 7 seconds and then interrupted and shifted to queue 3.
At Queue 3 it is executed for 12 seconds and then interrupted and shifted to queue 4.
At Queue 4 it is executed for 17 seconds and then interrupted and shifted to queue 5.
At Queue 5 it executes for 2 seconds and then it completes.
Hence the process is interrupted 4 times and completes on queue 5.

Advantages:

1.
It is more flexible.
2.
It allows different processes to move between different queues.
3.
It prevents starvation by moving a process that waits too long for lower priority queue to the higher priority queue.

Disadvantages:

1.
For the selection of the best scheduler, it require some other means to select the values.
2.
It produces more CPU overheads.
3.
It is most complex algorithm.

2:4 Multiple processor scheduling

Basics of Multi-processor Scheduling

- In multiple-processor scheduling multiple CPU's are available and hence Load Sharing becomes possible.
- Its more complex than single processor system.
- In multi-processor system, system are HOMOGENEOUS we can use any processor available to run any process in the queue.

Approaches to Multiple-Processor Scheduling

- There are two approaches
  1. Asymmetric Multiprocessing
  2. Symmetric Multiprocessing

Asymmetric Multiprocessing

1. **Asymmetric Multiprocessing**
   - when all the scheduling decisions and I/O processing are handled by a single processor which is called the Master Server and the other processors executes only the user code. This is simple and reduces the need of data sharing. This entire scenario is called Asymmetric Multiprocessing.

Asymmetric Multiprocessing

Asymmetric Multiprocessing

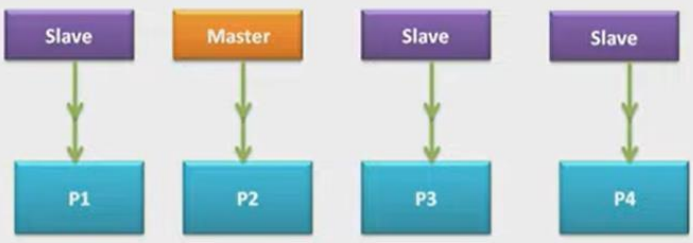


Figure 1. Asymmetric Multi-processing

Symmetric Multiprocessing

2. Symmetric Multiprocessing

- All processes may be in a common ready queue or each processor may have its own private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

Symmetric Multiprocessing

Symmetric Multiprocessing

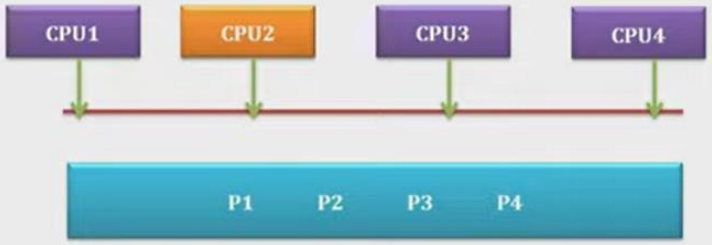


Figure 2. Symmetric Multi-processing

Processor Affinity

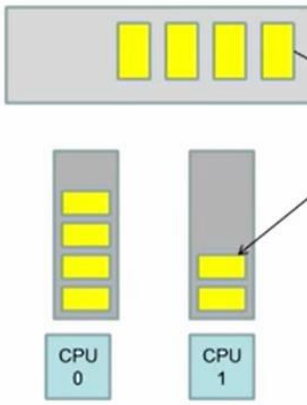
- **PROCESSOR AFFINITY**
- systems try to avoid migration of processes from one processor to another and try to keep a process running on the same processor. This is known as PROCESSOR AFFINITY.
- **Two types of Affinity**
  1. Soft Affinity
  2. Hard Affinity

1. **Soft Affinity** – When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.

2. **Hard Affinity** – Some system such as Linux also provide some system calls that support Hard Affinity which allows a process to migrate between processors.

3RD APPROACH

Hybrid Approach



- Use local and global queues
- Load balancing across queues feasible
- Locality achieved by processor affinity wrt the local queues
- Similar approach followed in Linux 2.6

Load Balancing

- Load Balancing is the phenomena which keeps the workload equally distributed across all processors in an SMP system.
- Load balancing is necessary only on systems where each processor has its own private queue of process which are eligible to execute.
- **There are two general approaches to load balancing**

Approaches to load balancing

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

TOPIC:3 OPERATIONS ON PROCESES:

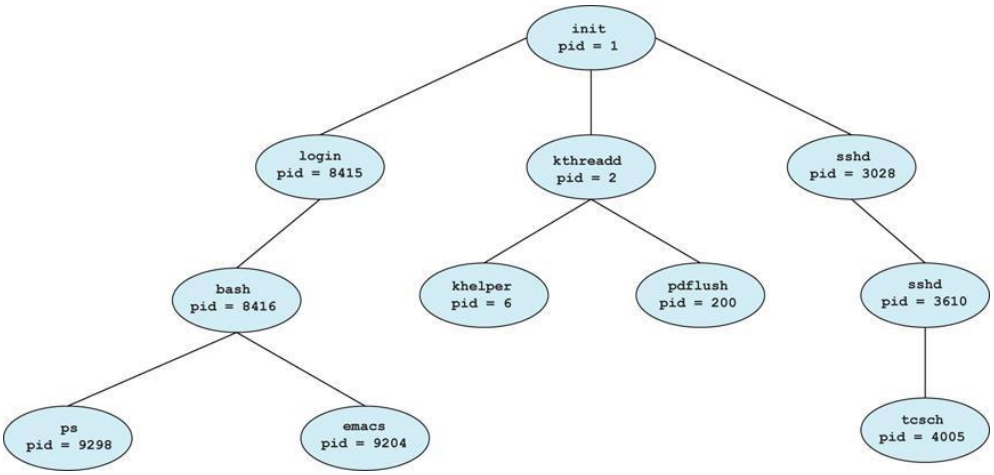
Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems

Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes. Most operating systems (including UNIX and the Windows family of operating systems) identify processes according to a unique process identifier (or pid), which is typically an integer number. Figure illustrates a typical process tree for the Solaris operating system, showing the name of each process and its pid.

In Solaris, the process at the top of the tree is the sched process, with pid of 0. The sched process creates several children processes—including pageout and f sf lush. These processes are responsible for managing memory and file systems. The sched process also creates the ini t process, which serves as the root parent process for all user processes. In Figure , we see two children of ini t — inetd and dtlogin. inetd is responsible for networking services such as telne t and ftp; dtlogin is the process representing a user login screen. When a user logs in, dtlogin creates an X-windows session (Xsession), which in turns creates the sdt_shel process. Below sdt_shel, a user's command-line shell—the C-shell or csh—is created. It is this commandline interface where the user then invokes various child processes, such as the Is and cat commands. We also see a csh process with pid of 7778 representing a user who has logged onto the system using telnet. This user has started the Netscape browser (pid of 7785) and the emacs editor (pid of 8105).



On UNIX, a listing of processes can be obtained using the ps command. For example, entering the command ps -e l will list complete information for all processes currently active in the system. It is easy to construct a process tree similar to what is shown in Figure by recursively tracing parent processes all the way to the ini t process. In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operatiiig system, or it may be constrained to a subset of the resources of the parent process.

The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses. In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process. For example, consider a process whose function is to display the contents of a file—say, img.jpg—on the screen of a terminal. When it is created, it will get, as an input from its parent process, the name of the file img.jpg, and it will use that file name, open the file, and write the contents out. It may also get the name of the output device. Some operating systems pass resources to child processes. On such a system, the new process may get two open files, img.jpg and the terminal device, and may simply transfer the datum between the two. When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.

2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).

2. The child process has a new program loaded into it. To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier.

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

int main()
{
pid_t pid;

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

    if (pid < 0) {/* error occurred */
       fprintf(stderr, "Fork Failed");
       exit(-1);
    }
    else if (pid == 0) {/* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else {/* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
       exit(0);
    }
}
```

A new process is created by the forkO system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the f ork(), with one difference: The return code for the forkO is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent. Typically, the execO system call is used after a forkO system call by one of the two processes to replace the process's memory space with a new program. The exec ()

system call loads a binary file into memory (destroying the memory image of the program containing the execO system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait () system call to move itself off the ready queue until the termination of the child.

The UNIX system calls previously described. We now have two different processes running a copy of the same program. The value of pid for the child process is zero; that for the parent is an integer value greater than zero. The child process overlays its address space with the UNIX command /bin/Is (used to get a directory listing) using the execlpO system call (execlpO is a version of the execO system call). The parent waits for the child process to complete with the wait () system call. When the child process completes (by either implicitly or explicitly invoking exit ()) the parent process resumes from the call to wait (), where it completes using the exit () system call.. As an alternative example, we next consider process creation in Windows. Processes are created in the Win32 API using the CreateProcessO function, which is similar to f ork () in that a parent creates a new child process. However, whereas f ork () has the child process inheriting the address space of its parent, CreateProcess () requires loading a specified program into the address space of the child process at process creation. Furthermore, whereas fork () is passed no parameters, CreateProcess 0 expects no fewer than ten parameters

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // allocate memory
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof (si) ;
    ZeroMemory(&pi, sizeof(pi));

    // create child process
    if (!CreateProcess(NULL, // use command line
     "C:\\WINDOWS\\system32\\mspaint.exe", // command line
     NULL, // don't inherit process handle
     NULL, // don't inherit thread handle
     FALSE, // disable handle inheritance
     0, //no creation flags
     NULL, // use parent's environment block
     NULL, // use parent's existing directory
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
```

Two parameters passed to CreateProcess () are instances of the STARTUPINFO and PROCESSJNFORMATION structures. STARTUPINFO specifies many properties of the new process, such as window size and appearance and handles to standard input and output files. The PROCESSJNFORMATION structure contains a handle and the identifiers to the newly created process and its thread. We invoke the ZeroMemoryO function to allocate memory for each of these structures before proceeding with CreateProcess (). The first two parameters

passed to CreateProcess () are the application name and command line parameters. If the application name is NULL (which in this case it is), the command line parameter specifies the application to load. In this instance we are loading the Microsoft Windows mspaint.exeapplication.

Beyond these two initial parameters, we use the default parameters for inheriting process and thread handles as well as specifying no creation flags. We also use the parent's existing environment block and starting directory. Last, we provide two pointers to the STARTUPINFO and PROCESS-INFORMATION structures created at the beginning of the program.The parent process waits for the child to complete by invoking the waitO system call. The equivalent of this in Win32 is WaitForSingleObj ect (), which is passed a handle of the child process—pi . hProcess— that it is waiting for to complete. Once the child process exits, control returns from the WaitForSingleOb j ect () function in the parent process.

Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system. Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcessO in Win32). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs. Note that a parent needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent. A parent may terminate the execution of one of its children for a variety of reasons, such as these:

• The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)

• The task assigned to the child is no longer required.

• The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. Some systems, including VMS, do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system.

To illustrate process execution and termination, consider that, in UNIX, we can terminate a process by using the exitQ system call; its parent process may wait for the termination of a child process by using the waitO system call. The wait () system call returns the process identifier of a terminated child so that the parent can tell which of its possibly many children has terminated. If the parent terminates, however, all its children have assigned as their new parent the ini t process. Thus, the children still have a parent to collect their status and execution statistics.

TOPIC:4 Inter Process Communication (IPC):

A process can be of two types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently, in reality, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity. Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other through both:

1. Shared Memory
2. Message passing

Figure 1 below shows a basic structure of communication between processes via the shared memory method and via the message passing method.

An operating system can implement both methods of communication. First, we will discuss the shared memory methods of communication and then message passing. Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.
Let's discuss an example of communication between processes using the shared memory method.

i) Shared Memory Method

Ex: Producer-Consumer problem

There are two processes: Producer and Consumer. The producer produces some items and the Consumer consumes that item. The two processes share a common space or memory location known as a buffer where the item produced by the Producer is stored and from which the Consumer consumes the item if needed. There are two versions of this problem: the first one is known as the unbounded buffer problem in which the Producer can keep on producing items and there is no limit on the size of the buffer, the second one is known as the bounded buffer problem in which the Producer can produce up to a certain number of items before it starts waiting for Consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then the producer will start producing items. If the total produced item is equal to the size of the buffer, the producer will wait to get it consumed by the Consumer. Similarly, the consumer will first check for the availability of the item. If no item is available, the Consumer will wait for the Producer to produce it. If there are items available, Consumer will consume them. The pseudo-code to demonstrate is provided below:

Shared Data between the two Processes

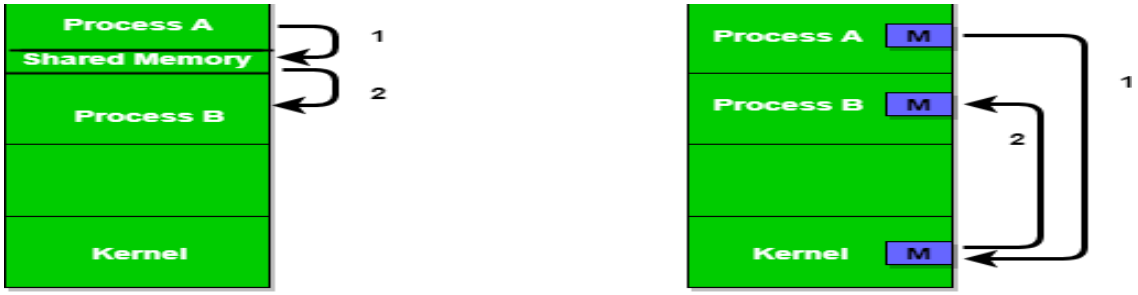


Figure 1 - Shared Memory and Message Passing

```
#define buff_max 25
#define mod %

        struct item{

                // different member of the produced data
                // or consumed data
                ------
        }

        // An array is needed for holding the items.
        // This is the shared place which will be
        // access by both process
        // item shared_buff [ buff_max ];

        // Two variables which will keep track of
        // the indexes of the items produced by producer
        // and consumer The free index points to
        // the next free index. The full index points to
        // the first full index.
        int free_index = 0;
        int full_index = 0;
```

Producer Process Code

```
item nextProduced;

        while(1){

                // check if there is no space
                // for production.
                // if so keep waiting.
                while((free_index+1) mod buff_max == full_index);

                shared_buff[free_index] = nextProduced;
                free_index = (free_index + 1) mod buff_max;
        }
```

Consumer Process Code

```
item nextConsumed;

        while(1){
```

```
        // check if there is an available
        // item for consumption.
        // if not keep on waiting for
        // get them produced.
        while((free_index == full_index);
```

In the above code, the Producer will start producing again when the (free_index+1) mod buff max will be free because if it it not free, this implies that there are still items that can be consumed by the Consumer so there is no need to produce more. Similarly, if free index and full index point to the same index, this implies that there are no items to consume.

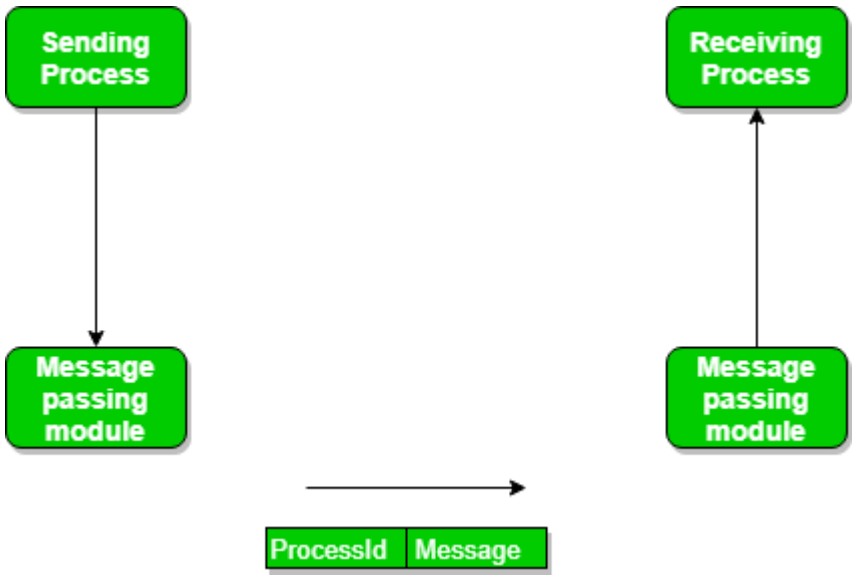
```
        nextConsumed = shared_buff[full_index];
        full_index = (full_index + 1) mod buff_max;
    }
```

ii) Messaging Passing Method

Now, We will start our discussion of the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives.

We need at least two primitives:
– **send**(message, destination) or **send**(message)
– **receive**(message, host) or **receive**(message)



The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for an OS designer but complicated for a programmer and if it is of variable size then it is easy for a programmer but complicated for the OS designer. A standard message can have two parts: **header and body.**

The **header part** is used for storing message type, destination id, source id, message length, and control information. The control information contains information like what to do if runs

out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

Message Passing through Communication Link.

Direct and Indirect Communication link

Now, We will start our discussion about the methods of implementing communication links. While implementing the link, there are some questions that need to be kept in mind like :

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

A link has some capacity that determines the number of messages that can reside in it temporarily for which every link has a queue associated with it which can be of zero capacity, bounded capacity, or unbounded capacity. In zero capacity, the sender waits until the receiver informs the sender that it has received the message. In non-zero capacity cases, a process does not know whether a message has been received or not after the send operation. For this, the sender must communicate with the receiver explicitly. Implementation of the link depends on the situation, it can be either a direct communication link or an in-directed communication link.

Direct Communication links are implemented when the processes use a specific process identifier for the communication, but it is hard to identify the sender ahead of time.

For example the print server.

In-direct Communication is done via a shared mailbox (port), which consists of a queue of messages. The sender keeps the message in mailbox and the receiver picks them up.

Message Passing through Exchanging the Messages.

Synchronous and Asynchronous Message Passing:

A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation. IPC is possible between the processes on same computer as well as on the processes running on different computer i.e. in networked/distributed system. In both cases, the process may or may not be blocked while sending a message or attempting to receive a message so message passing may be blocking or non-blocking. Blocking is considered **synchronous** and **blocking send** means the sender will be blocked until the message is received by receiver. Similarly, **blocking receive** has the receiver block until a message is available. Non-blocking is considered **asynchronous** and Non-blocking send has the sender sends the message and continue. Similarly, Non-blocking receive has the receiver receive a valid message or null. After a careful analysis, we can come to a conclusion that for a sender it is more natural to be non-blocking after message passing as there may be a need to send the message to different processes. However, the sender expects acknowledgment from the receiver in case the send fails. Similarly, it is more natural for a receiver to be blocking after issuing the receive as the information from the received message may be used for further execution. At the same time, if the message send keep on failing, the receiver will have to wait indefinitely. That is why we also consider the other possibility of message passing. There are basically three preferred combinations:

- Blocking send and blocking receive
- Non-blocking send and Non-blocking receive

- Non-blocking send and Blocking receive (Mostly used)

In Direct message passing, The process which wants to communicate must explicitly name the recipient or sender of the communication.
e.g. **send(p1, message)** means send the message to p1.
Similarly, **receive(p2, message)** means to receive the message from p2.
In this method of communication, the communication link gets established automatically, which can be either unidirectional or bidirectional, but one link can be used between one pair of the sender and receiver and one pair of sender and receiver should not possess more than one pair of links. Symmetry and asymmetry between sending and receiving can also be implemented i.e. either both processes will name each other for sending and receiving the messages or only the sender will name the receiver for sending the message and there is no need for the receiver for naming the sender for receiving the message. The problem with this method of communication is that if the name of one process changes, this method will not work.
In Indirect message passing, processes use mailboxes (also referred to as ports) for sending and receiving messages. Each mailbox has a unique id and processes can communicate only if they share a mailbox. Link established only if processes share a common mailbox and a single link can be associated with many processes. Each pair of processes can share several communication links and these links may be unidirectional or bi-directional. Suppose two processes want to communicate through Indirect message passing, the required operations are: create a mailbox, use this mailbox for sending and receiving messages, then destroy the mailbox. The standard primitives used are: **send(A, message)** which means send the message to mailbox A. The primitive for the receiving the message also works in the same way e.g. **received (A, message)**. There is a problem with this mailbox implementation. Suppose there are more than two processes sharing the same mailbox and suppose the process p1 sends a message to the mailbox, which process will be the receiver? This can be solved by either enforcing that only two processes can share a single mailbox or enforcing that only one process is allowed to execute the receive at a given time or select any process randomly and notify the sender about the receiver. A mailbox can be made private to a single sender/receiver pair and can also be shared between multiple sender/receiver pairs. Port is an implementation of such mailbox that can have multiple senders and a single receiver. It is used in client/server applications (in this case the server is the receiver). The port is owned by the receiving process and created by OS on the request of the receiver process and can be destroyed either on request of the same receiver processor when the receiver terminates itself. Enforcing that only one process is allowed to execute the receive can be done using the concept of mutual exclusion. **Mutex mailbox** is created which is shared by n process. The sender is non-blocking and sends the message. The first process which executes the receive will enter in the critical section and all other processes will be blocking and will wait. Now, let's discuss the Producer-Consumer problem using the message passing concept. The producer places items (inside messages) in the mailbox and the consumer can consume an item when at least one message present in the mailbox. The code is given below:

Producer Code

```
void Producer(void){

        int item;
        Message m;

        while(1){

            receive(Consumer, &m);
            item = produce();
```

```
                build_message(&m , item ) ;
                send(Consumer, &m);
            }
        }
```

Consumer Code

```
void Consumer(void){

            int item;
            Message m;

            while(1){

                receive(Producer, &m);
                item = extracted_item();
                send(Producer, &m);
                consume_item(item);
            }
        }
```

Examples of IPC systems

1. Posix : uses shared memory method.
2. Mach : uses message passing
3. Windows XP : uses message passing using local procedural calls

Communication in client/server Architecture:

There are various mechanism:

- Pipe
- Socket
- Remote Procedural calls (RPCs)

TOPIC-5:Communication in client server systems:

Client/Server communication involves two components, namely a client and a server. They are usually multiple clients in communication with a single server. The clients send requests to the server and the server responds to the client requests.

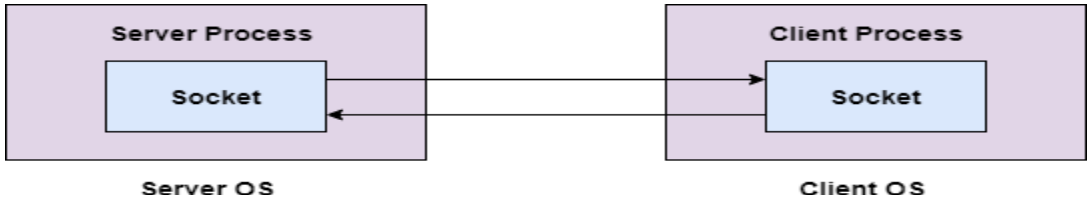
There are three main methods to client/server communication. These are given as follows −

Sockets

Sockets facilitate communication between two processes on the same machine or different machines. They are used in a client/server framework and consist of the IP address and port number. Many application protocols use sockets for data connection and data transfer between a client and a server.

Socket communication is quite low-level as sockets only transfer an unstructured byte stream across processes. The structure on the byte stream is imposed by the client and server applications.

A diagram that illustrates sockets is as follows −

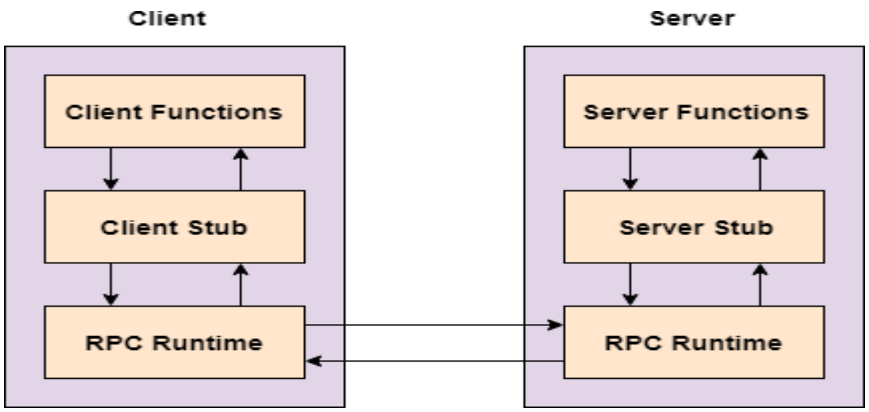


Remote Procedure Calls

These are interprocess communication techniques that are used for client-server based applications. A remote procedure call is also known as a subroutine call or a function call.

A client has a request that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client.

A diagram that illustrates remote procedure calls is given as follows −



Pipes

These are interprocess communication methods that contain two end points. Data is entered from one end of the pipe by a process and consumed from the other end by the other process.

The two different types of pipes are ordinary pipes and named pipes. Ordinary pipes only allow one way communication. For two way communication, two pipes are required. Ordinary pipes have a parent child relationship between the processes as the pipes can only be accessed by processes that created or inherited them.

Named pipes are more powerful than ordinary pipes and allow two way communication. These pipes exist even after the processes using them have terminated. They need to be explicitly deleted when not required anymore.

A diagram that demonstrates pipes are given as follows −

TOPIC-6:Multithreaded Programming:

6:1 : Multithreading models,

6:2: Thread libraries,

6:3 Threading issues.

6:4 Thread scheduling.

Threads:

A process is divide into number of light weight process, each light weight process is said to be a Thread. The Thread has a program counter (Keeps track of which instruction to execute next), registers (holds its current working variables), stack (execution History).

Thread States:

1. bornState : A thread is justcreated.
2. readystate : The thread is waiting forCPU.
3. running : System assigns the processor to thethread.
4. sleep : A sleeping thread becomes ready after the designated sleep timeexpires.
5. dead : The Execution of the threadfinished.

Eg: Word processor.

Typing, Formatting, Spell check, saving are threads.

Differences between Process and Thread

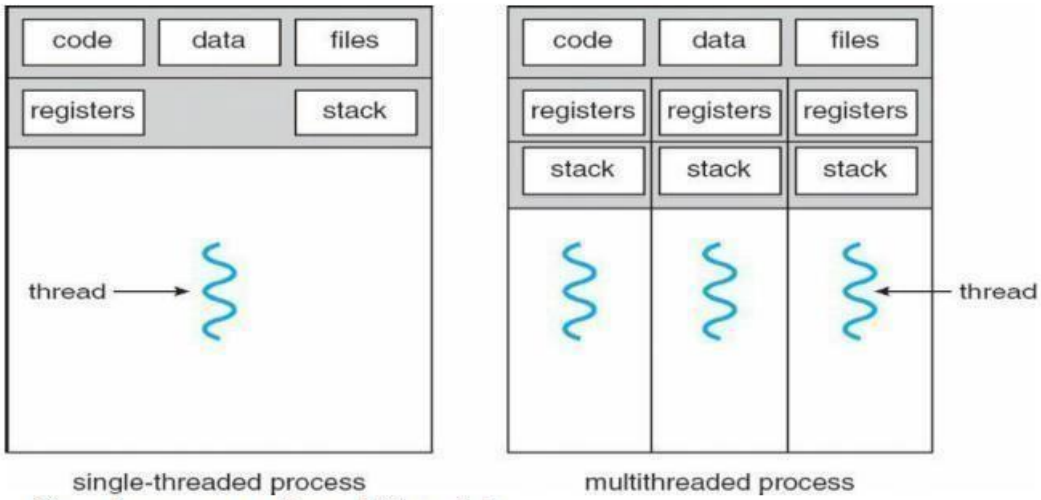
Process	Thread
Process takes more time to create.	Thread takes less time to create.
it takes more time to complete execution & terminate.	Less time to terminate.
Execution is very slow.	Execution is very fast.
It takes more time to switch b/w two processes.	It takes less time to switch b/w two threads.
Communication b/w two processes is difficult .	Communication b/w two threads is easy.
Process can't share the same memory area.	Threads can share same memory area.
System calls are requested to communicate each other.	System calls are not required.
Process is loosely coupled.	Threads are tightly coupled.
It requires more resources to execute.	Requires few resources to execute.

Multithreading

A process is divided into number of smaller tasks each task is called a Thread. Number of Threads with in a Process execute at a time is called Multithreading.
If a program, is multithreaded, even when some portion of it is blocked, the whole program is not blocked.The rest of the program continues working If multiple CPU's are available. Multithreading gives best performance.If we have only a single thread, number of CPU's available, No performance benefits achieved.

- Process creation is heavy-weight while thread creation is light-weight

□ Can simplify code, increase efficiency



□ Kernels are generally multithreaded

CODE- Contains instruction
DATA- holds global variable **FILES-**
opening and closing files
REGISTER- contain information about CPU state
STACK-parameters, local variables, functions
Types Of Threads:

1) **User Threads** : Thread creation, scheduling, management happen in user space by Thread Library. user threads are faster to create and manage. If a user thread performs a system call, which blocks it, all the other threads in that process one also automatically blocked, whole process is blocked.

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 areblocking.
- Multithreaded application cannot take advantage ofmultiprocessing.

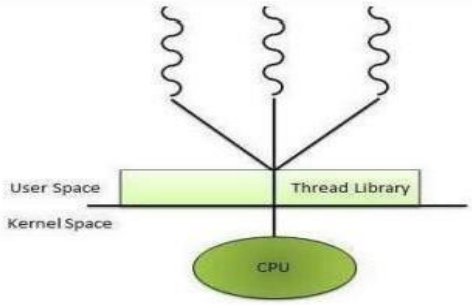
2) Kernel Threads: kernel creates, schedules, manages these threads .these threads are slower, manage. If one thread in a process blocked, over all process need not be blocked.

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 sameprocess.
- Kernel routines themselves can multithreaded.

Disadvantages

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



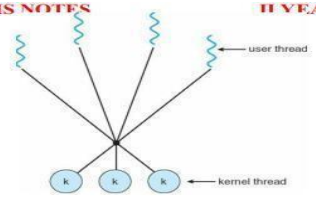
Multithreading Models

Some operating system provides 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

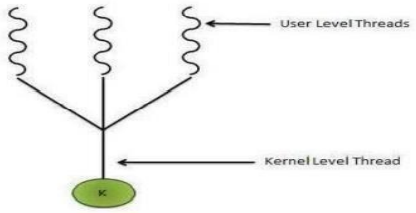
In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine. Following diagram shows the many to many model. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



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. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time,so multiple threads are unable to run in parallel on multiprocessors.

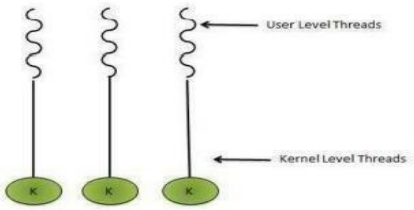
If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship modes.



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 another thread to run when a thread makes a blocking system call. It support multiple thread 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.



User level thread	Kernel level thread		
User thread are implemented by users.	kernel threads are implemented by OS.		
OS doesn't recognized user level threads.	Kernel threads are recognized by OS.		
Implementation of User threads is easy.	Implementation of Kernel thread is complicated.		
Context switch time is less.	Context switch time is more.		
Context switch requires no hardware support.	Hardware support is needed.		
If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.		
User level threads are designed as dependent threads.	Kernel level threads are designed as independent threads.	Example : Java thread,	Example : Window

6:2 Thread Libraries

A thread library provides the programmer an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main thread libraries are in use today:

1. POSIX Pthreads,
2. Win32, and
3. Java. Pthreads,

the threads extension of the POSIX standard, may be provided as either a user- or kernel-level library. The Win32 thread library is a kernel-level library available on Windows systems. The Java thread API allows thread creation and management directly in Java programs. However, because in most instances the JVM is running on top of a host operating system, the Java thread API is typically implemented using a thread library available on the host system. This means that on Windows systems, Java threads are typically implemented using the Win32 API; UNIX and Linux systems often use Pthreads.

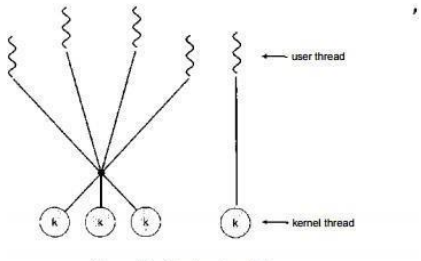


Figure 4.5 Two-level model.

In the remainder of this section, we describe basic thread creation using these three thread libraries. As an illustrative example, we design a multithreaded program that performs the summation of a non-negative integer in a separate thread using the well known summation function. For example, if N were 5, this function would represent the summation from 0 to 5, which is 15. Each of the three programs will be run with the upper bounds of the summation entered on the command line; thus, if the user enters 8, the summation of the integer values from 0 to 8 will be output.

Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a specification for thread behavior, not an implementation. Operating system designers may implement the specification in any way they wish. Numerous systems implement the Pthreads specification, including Solaris, Linux, Mac OS X, and Tru64 UNIX. Shareware implementations are available in the public domain for the various Windows operating systems as well. In a Pthreads program, separate threads begin execution in a specified function, this is the runner () function. When this program begins, a single thread of control begins in main().

After some initialization, main() creates a second thread that begins control in the runner () function. Both threads share the global data sum. Let's look more closely at this program. All Pthreads programs must include the pthread.h header file. The statement pthreadjt tid declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The pthread_attr_t attr declaration represents the attributes for the thread. We set the attributes in the function call pthread_attr_init C&attr). Because we did not explicitly set any attributes, we use the default attributes provided. A separate thread is created with the pthread_create () function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution-—in this case, the runner () function. Last, we pass the integer parameter that was provided on the command line, argv [1]. At this point, the program has two threads: the initial (or parent) thread in main() and the summation (or child) thread performing the summation operation in the runner () function. After creating the summation thread, the parent thread will wait for it to complete by calling the pthread_join() function. The summation thread will complete when it calls the function pthread.exit() . Once the summation thread has returned, the parent thread will output the value of the shared data sum.

Win32 Threads

The technique for creating threads using the Win32 thread library is similar to the Pthreads technique in several ways. we must include the windows.h header file when using the Win32 API. Just as in the Pthreads, data shared by the separate threads—in this case, Sum—are declared globally (the DWORD data type is an unsigned 32-bit integer. We also define the SummationO function that is to be performed in a separate thread. This function is passed a pointer to a void, which Win32 defines as

LPVOID. The thread performing this function sets the global data Sum to the value of the summation from 0 to the parameter passed to SummationO.
Threads are created in the Win32 API using the CreateThreadO function and—just as in Pthreads—a set of attributes for the thread is passed to this function. These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state. In this program, we use the default values for these attributes (which do not initially set the thread to a suspended state and instead make it eligible to be run by the CPU scheduler). Once the summation thread is created, the parent must wait for it to complete before outputting the value of Sum, as the value is set by the summation thread. Recall that the Pthread program had the parent thread wait for the summation thread using the pthread join() statement. We perform the equivalent of this in the Win32 API using the WaitForSingleObject () function, which causes the creating thread to block until the summation thread has exited.

Java Threads

Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads. All Java programs comprise at least a single thread, even a simple Java program consisting of only a main() method runs as a single thread in the JVM. There are two techniques for creating threads in a Java program. One approach is to create a new class that is derived from the Thread class and to override its run() method. An alternative—and more commonly used— technique is to define a class that implements the Runnable interface.

6:3 Threading Issues

4.6.1 The fork( ) and exec( ) System Calls

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

4.6.2 Signal Handling

- Q: When a multi-threaded process receives a signal, to what thread should that signal be delivered?
- A: There are four major options:

    1. Deliver the signal to the thread to which the signal applies.
    2. Deliver the signal to every thread in the process.
    3. Deliver the signal to certain threads in the process.
    4. Assign a specific thread to receive all signals in a process.

- The best choice may depend on which specific signal is involved.
- UNIX allows individual threads to indicate which signals they are accepting and which they are ignoring. However the signal can only be

delivered to one thread, which is generally the first thread that is accepting that particular signal.

- UNIX provides two separate system calls, **kill(pid, signal)** and **pthread_kill(tid, signal)**, for delivering signals to processes or specific threads respectively.
- Windows does not support signals, but they can be emulated using Asynchronous Procedure Calls (APCs). APCs are delivered to specific threads, not processes.

4.6.3 Thread Cancellation

- Threads that are no longer needed may be cancelled by another thread in one of two ways:

    1. **Asynchronous Cancellation** cancels the thread immediately.
    2. **Deferred Cancellation** sets a flag indicating the thread should cancel itself when it is convenient. It is then up to the cancelled thread to check this flag periodically and exit nicely when it sees the flag set.

- (Shared) resource allocation and inter-thread data transfers can be problematic with asynchronous cancellation.

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

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

4.6.5 Scheduler Activations

- Many implementations of threads provide a virtual processor as an interface between the user thread and the kernel thread, particularly for the many-to-many or two-tier models.
- This virtual processor is known as a "Lightweight Process", LWP.

    - There is a one-to-one correspondence between LWPs and kernel threads.
    - The number of kernel threads available, (and hence the number of LWPs) may change dynamically.
    - The application (user level thread library) maps user threads onto available LWPs.
    - kernel threads are scheduled onto the real processor(s) by the OS.
    - The kernel communicates to the user-level thread library when certain events occur (such as a thread about to block) via an **upcall**, which is handled in the thread library by an **upcall**

handler. The upcall also provides a new LWP for the upcall handler to run on, which it can then use to reschedule the user thread that is about to become blocked. The OS will also issue upcalls when a thread becomes unblocked, so the thread library can make appropriate adjustments.

- If the kernel thread blocks, then the LWP blocks, which blocks the user thread.
- Ideally there should be at least as many LWPs available as there could be concurrently blocked kernel threads. Otherwise if all LWPs are blocked, then user threads will have to wait for one to become available.

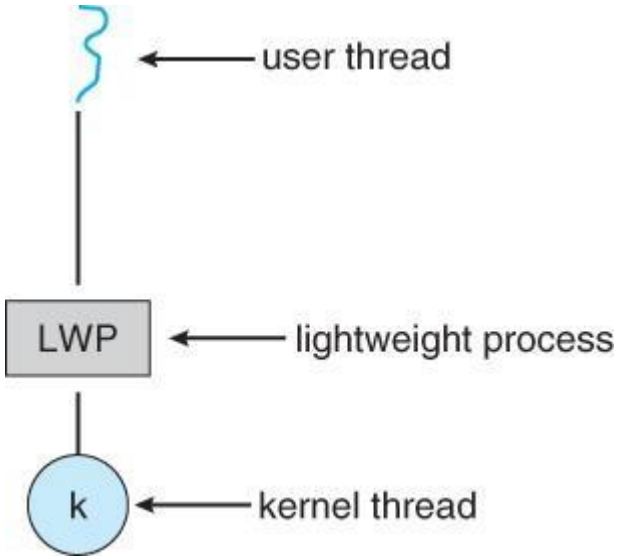


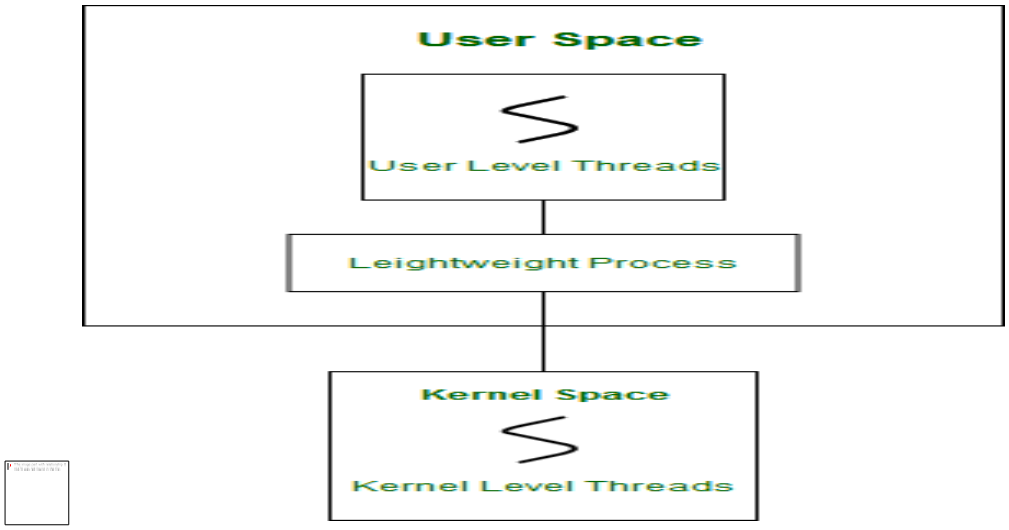
Figure 4.13 - Lightweight process (LWP)

6:4 Thread scheduling

Scheduling of threads involves two boundary scheduling,

- Scheduling of user level threads (ULT) to kernel level threads (KLT) via leightweight process (LWP) by the application developer.
- Scheduling of kernel level threads by the system scheduler to perform different unique os functions.

Leightweight Process (LWP) :
Light-weight process are threads in the user space that acts as an interface for the ULT to access the physical CPU resources. Thread library schedules which thread of a process to run on which LWP and how long. The number of LWP created by the thread library depends on the type of application. In the case of an I/O bound application, the number of LWP depends on the number of user-level threads. This is because when an LWP is blocked on an I/O operation, then to invoke the other ULT the thread library needs to create and schedule another LWP. Thus, in an I/O bound application, the number of LWP is equal to the number of the ULT. In the case of a CPU bound application, it depends only on the application. Each LWP is attached to a separate kernel-level thread.



In real-time, the first boundary of thread scheduling is beyond specifying the scheduling policy and the priority. It requires two controls to be specified for the User level threads: Contention scope, and Allocation domain. These are explained as following below.

1. Contention Scope :
The word contention here refers to the competition or fight among the User level threads to access the kernel resources. Thus, this control defines the extent to which contention takes place. It is defined by the application developer using the thread library. Depending upon the extent of contention it is classified as **Process Contention Scope** and **System Contention Scope**.

1. **Process Contention Scope (PCS) –**
   The contention takes place among threads **within a same process**. The thread library schedules the high-prioritized PCS thread to access the resources via available LWPs (priority as specified by the application developer during thread creation).
2. **System Contention Scope (SCS) –**
   The contention takes place among **all threads in the system**. In this case, every SCS thread

is associated to each LWP by the thread library and are scheduled by the system scheduler to access the kernel resources.

In LINUX and UNIX operating systems, the POSIX Pthread library provides a function Pthread_attr_setscope to define the type of contention scope for a thread during its creation.

3.

int Pthread_attr_setscope(pthread_attr_t *attr, int scope)

4.

The first parameter denotes to which thread within the process the scope is defined.
The second parameter defines the scope of contention for the thread pointed. It takes two values.

5.

PTHREAD_SCOPE_SYSTEM

PTHREAD_SCOPE_PROCESS

6.

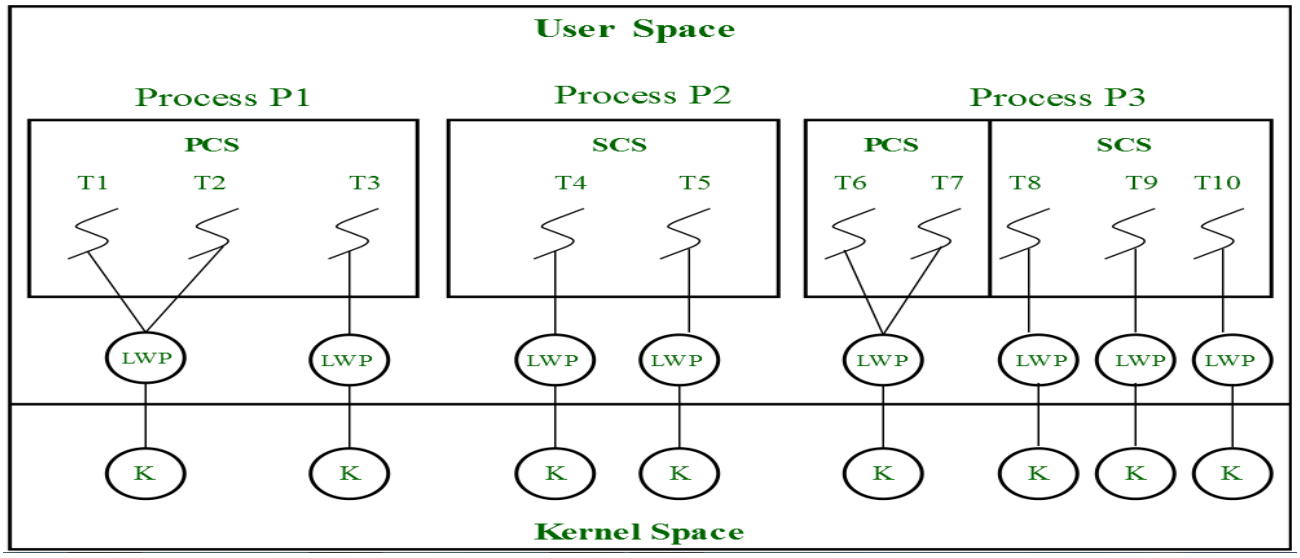
If the scope value specified is not supported by the system, then the function returns ENOTSUP.

7.

2. Allocation Domain :

The allocation domain is **a set of one or more resources** for which a thread is competing. In a multicore system, there may be one or more allocation domains where each consists of one or more cores. One ULT can be a part of one or more allocation domain. Due to this high complexity in dealing with hardware and software architectural interfaces, this control is not specified. But by default, the multicore system will have an interface that affects the allocation domain of a thread.

Consider a scenario, an operating system with three process P1, P2, P3 and 10 user level threads (T1 to T10) with a single allocation domain. 100% of CPU resources will be distributed among all the three processes. The amount of CPU resources allocated to each process and to each thread depends on the contention scope, scheduling policy and priority of each thread defined by the application developer using thread library and also depends on the system scheduler. These User level threads are of a different contention scope.



In this case, the contention for allocation domain takes place as follows,

1. **Process P1:**
   All PCS threads T1, T2, T3 of Process P1 will compete among themselves. The PCS threads of the same process can share one or more LWP. T1 and T2 share an LWP and T3 are allocated to a separate LWP. Between T1 and T2 allocation of kernel resources via LWP is based on preemptive priority scheduling by the thread library. A Thread with a high priority will preempt low priority threads. Whereas, thread T1 of process p1 cannot preempt thread T3 of process p3 even if the priority of T1 is greater than the priority of T3. If the priority is equal, then the allocation of ULT to available LWPs is based on the scheduling policy of threads by the system scheduler(not by thread library, in this case).
2. **Process P2:**
   Both SCS threads T4 and T5 of process P2 will compete with processes P1 as a whole and with SCS threads T8, T9, T10 of process P3. The system scheduler will schedule the kernel resources among P1, T4, T5, T8, T9, T10, and PCS threads (T6, T7) of process P3 considering each as a separate process. Here, the Thread library has no control of scheduling the ULT to the kernel resources.
3. **Process P3:**
   Combination of PCS and SCS threads. Consider if the system scheduler allocates 50% of CPU resources to process P3, then 25% of resources is for process scoped threads and the remaining 25% for system scoped threads. The PCS threads T6 and T7 will be allocated to access the 25% resources based on the priority by the thread library. The SCS threads T8, T9, T10 will divide the 25% resources among themselves and access the kernel resources via separate LWP and KLT. The SCS scheduling is by the system scheduler.

Note:
For every system call to access the kernel resources, a Kernel Level thread is created and associated to separate LWP by the system scheduler.

Number of Kernel Level Threads = Total Number of LWP

Total Number of LWP = Number of LWP for SCS + Number of LWP for PCS

Number of LWP for SCS = Number of SCS threads

Number of LWP for PCS = Depends on application developer

Here,

Number of SCS threads = 5

Number of LWP for PCS = 3

Number of SCS threads = 5

Number of LWP for SCS = 5

Total Number of LWP = 8 (=5+3)

Number of Kernel Level Threads = 8

Advantages of PCS over SCS :

- If all threads are PCS, then context switching, synchronization, scheduling everything takes place within the userspace. This reduces system calls and achieves better performance.
- PCS is cheaper than SCS.
- PCS threads share one or more available LWPs. For every SCS thread, a separate LWP is associated.For every system call, a separate KLT is created.
- The number of KLT and LWPs created highly depends on the number of SCS threads created. This increases the kernel complexity of handling scheduling and synchronization. Thereby, results in a limitation over SCS thread creation, stating that, the number of SCS threads to be smaller than the number of PCS threads.

- If the system has more than one allocation domain, then scheduling and synchronization of resources becomes more tedious. Issues arise when an SCS thread is a part of more than one allocation domain, the system has to handle n number of interfaces.

The second boundary of thread scheduling involves CPU scheduling by the system scheduler. The scheduler considers each kernel-level thread as a separate process and provides access to the kernel resources.

Difference between

USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are implemented by users.	kernel threads are implemented by OS.
OS doesn't recognized user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Hardware support is needed.
If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
Example : Java thread, POSIX threads.	Example : Window Solaris.

TOPIC:7

7:1 Inter-process Communication: Race conditions,

7:2 Critical Regions,

7:3 Mutual exclusion with busy waiting,

7:4 Sleep and wakeup, Semaphores,

7:5 Mutexes,

7:6Monitors,

7:7 Message passing,

7:8Barriers,

7:9 Classical IPC Problems - Dining philosophers problem, Readers and writers problem

Topic 7;1:Inter-process Communication: Race conditions:

Background

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

  Producer code from chapter 3:

-

```
item nextProduced;

while( true ) {

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

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

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

}
```

Consumer code from chapter 3:

```
item nextConsumed;

while( true ) {

     /* Wait for an item to become
     available */
```

```
        while( in == out )
            ; /* Do nothing */

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

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

    }
```

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

Producer Process:

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

Consumer Process:

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

- Unfortunately we have now introduced a new problem, because both the producer and the consumer are adjusting the value of the variable counter, which can lead to a condition known as a **race condition**. In this condition a piece of code may or may not

work correctly, depending on which of two simultaneous processes executes first, and more importantly if one of the processes gets interrupted such that the other process runs between important steps of the first process. (Bank balance example discussed in class.)

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

Producer:

$$register_1 = \texttt{counter}$$
$$register_1 = register_1 + 1$$
$$\texttt{counter} = register_1$$

Consumer:

$$register_2 = \texttt{counter}$$
$$register_2 = register_2 - 1$$
$$\texttt{counter} = register_2$$

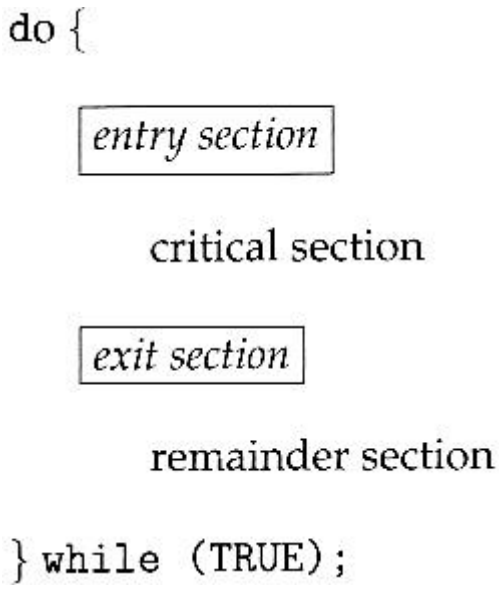
Interleaving:

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

Topic 7:2 The Critical-Section Problemor Critical Regions,

- The producer-consumer problem described above is a specific example of a more general situation known as the **critical section** problem. The general idea is that in a number of cooperating processes, each has a critical section of code, with the following conditions and terminologies:
  - Only one process in the group can be allowed to execute in their critical section at any one time. If one process is already executing their critical section and another process wishes to do so, then the second process must be made to wait until the first process has completed their critical section work.
  - The code preceding the critical section, and which controls access to the critical section, is termed the entry section. It acts like a carefully controlled locking door.

- The code following the critical section is termed the exit section. It generally releases the lock on someone else's door, or at least lets the world know that they are no longer in their critical section.
- The rest of the code not included in either the critical section or the entry or exit sections is termed the remainder section.



- **Figure 5.1 - General structure of a typical process Pi**

- A solution to the critical section problem must satisfy the following three conditions:
  1. **Mutual Exclusion** - Only one process at a time can be executing in their critical section.
  2. **Progress** - If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then only the processes not in their remainder sections can participate in the decision, and the decision cannot be postponed indefinitely. (I.e. processes cannot be blocked forever waiting to get into their critical sections.)
  3. **Bounded Waiting** - There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted. (I.e. a process requesting entry into their critical section will get a turn eventually, and there is a limit as to how many other processes get to go first.)
- We assume that all processes proceed at a non-zero speed, but no assumptions can be made regarding the **relative** speed of one process versus another.
- Kernel processes can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures such as open file tables or virtual memory management. Accordingly kernels can take on one of two forms:

  1. Non-preemptive kernels do not allow processes to be interrupted while in kernel mode. This eliminates the possibility of kernel-mode race conditions, but requires kernel mode operations to complete very quickly, and can be problematic for real-time systems, because timing cannot be guaranteed.

2. Preemptive kernels allow for real-time operations, but must be carefully written to avoid race conditions. This can be especially tricky on SMP systems, in which multiple kernel processes may be running simultaneously on different processors.

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

Peterson's Solution

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

  - In the entry section, process i first raises a flag indicating a desire to enter the critical section.
  - Then turn is set to **j** to allow the **other** process to enter their critical section **if process j so desires.**
  - The while loop is a busy loop (notice the semicolon at the end), which makes process i wait as long as process j has the turn and wants to enter the critical section.
  - Process i lowers the flag[i] in the exit section, allowing process j to continue if it has been waiting.

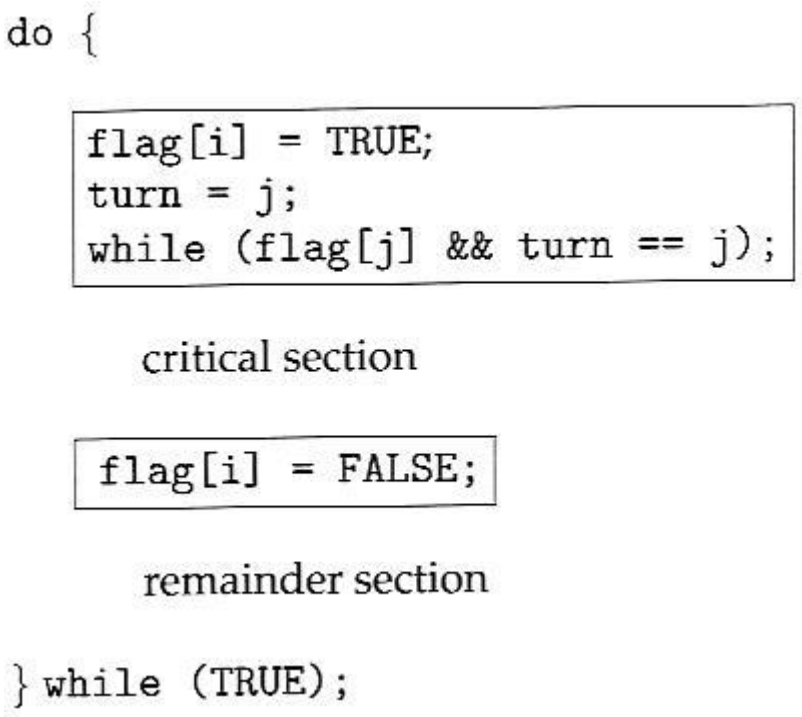


Figure 5.2 - The structure of process Pi in Peterson's solution.

- To prove that the solution is correct, we must examine the three conditions listed above:
    1. **Mutual exclusion** - If one process is executing their critical section when the other wishes to do so, the second process will become blocked by the flag of the first process. If both processes attempt to enter at the same time, the last process to execute "turn = j" will be blocked.
    2. **Progress** - Each process can only be blocked at the while if the other process wants to use the critical section (flag[j] = = true), AND it is the other process's turn to use the critical section (turn = = j). If both of those conditions are true, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, will set flag[j] to false, releasing process i. The shared variable turn assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.
    3. **Bounded Waiting** - As each process enters their entry section, they set the turn variable to be the other processes turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.
- Note that the instruction "turn = j" is **atomic,** that is it is a single machine instruction which cannot be interrupted.

Synchronization Hardware sollution to cs probel

- To generalize the solution(s) expressed above, each process when entering their critical section must set some sort of **lock**, to prevent other processes from entering their critical sections simultaneously, and must release the lock when exiting their critical section, to allow other processes to proceed. Obviously it must be possible to attain the lock only

when no other process has already set a lock. Specific implementations of this general procedure can get quite complicated, and may include hardware solutions as outlined in this section.

- One simple solution to the critical section problem is to simply prevent a process from being interrupted while in their critical section, which is the approach taken by non preemptive kernels. Unfortunately this does not work well in multiprocessor environments, due to the difficulties in disabling and the re-enabling interrupts on all processors. There is also a question as to how this approach affects timing if the clock interrupt is disabled.
- Another approach is for hardware to provide certain **atomic** operations. These operations are guaranteed to operate as a single instruction, without interruption. One such operation is the "Test and Set", which simultaneously sets a boolean lock variable and returns its previous value, as shown in Figures 5.3 and 5.4:

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

Figure 5.3 The definition of the TestAndSet() instruction.

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

      // critical section

   lock = FALSE;

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

Figure 5.4 Mutual-exclusion implementation with TestAndSet().

Figures 5.3 and 5.4 illustrate "test_and_set( )" function

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

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

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

   return temp;
}
```

Figure 5.5 The definition of the `compare_and_swap()` instruction.

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

      /* critical section */

   lock = 0;

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

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

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

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

      // critical section

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

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

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

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

Topic 7:3 Mutual Exclusion with Busy Waiting (Software approach)

- *Mutual exclusion* is a mechanism to ensure that only one process (or person) is doing certain things at one time, thus avoid data inconsistency. All others should be prevented from modifying shared data until the current process finishes
- Strict Alternation (see Fig. 2.11)
  - the two processes strictly alternate in entering their CR
  - the integer variable **turn,** initially 0, keeps track of whose turn is to enter the critical region
  - **busy waiting**, continuously testing a variable until some value appears, a lock that uses busy waiting is called a **spin lock**

- both processes are executing in their noncritical regions
- process 0 finishes its noncritical region and goes back to the top of its loop
- unfortunately, it is not permitted to enter its CR, **turn** is 1 and process 1 is busy with its nonCR
- this algorithm does avoid all races
- but violates condition 3

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Figure 2.11: A proposed solution to the CR problem. (a) Process 0, (b)Process1

Topic 7:4

In order to read and write, both of them are usinga buffer. The code that simulates the sleep and wake mechanism in terms of providing the solution to producer consumer problem is shown below.

```
1.  #define N 100 //maximum slots in buffer
2.  #define count=0 //items in the buffer
3.  void producer (void)
4.  {
5.     int item;
6.     while(True)
7.     {
8.        item = produce_item(); //producer produces an item
9.        if(count == N) //if the buffer is full then the producer will sleep
10.          Sleep();
11.       insert_item (item); //the item is inserted into buffer
12.       countcount=count+1;
13.       if(count==1) //The producer will wake up the
```

```
14.             //consumer if there is at least 1 item in the buffer
15.             wake-up(consumer);
16.         }
17.     }
18.
19.     void consumer (void)
20.     {
21.         int item;
22.         while(True)
23.         {
24.             {
25.                 if(count == 0) //The consumer will sleep if the buffer is empty.
26.                 sleep();
27.                 item = remove_item();
28.                 countcount = count - 1;
29.                 if(count == N-1) //if there is at least one slot available in the buffer
30.                 //then the consumer will wake up producer
31.                 wake-up(producer);
32.                 consume_item(item); //the item is read by consumer.
33.             }
34.         }
35.     }
```

The producer produces the item and inserts it into the buffer. The value of the global variable count got increased at each insertion. If the buffer is filled completely and no slot is available then the producer will sleep, otherwise it keep inserting.

On the consumer's end, the value of count got decreased by 1 at each consumption. If the buffer is empty at any point of time then the consumer will sleep otherwise, it keeps consuming the items and decreasing the value of count by 1.

The consumer will be waked up by the producer if there is at least 1 item available in the buffer which is to be consumed. The producer will be waked up by the consumer if there is at least one slot available in the buffer so that the producer can write that.

Well, the problem arises in the case when the consumer got preempted just before it was about to sleep. Now the consumer is neither sleeping nor consuming. Since the producer is not aware of the fact that consumer is not actually sleeping therefore it keep waking the consumer while the consumer is not responding since it is not sleeping.

This leads to the wastage of system calls. When the consumer get scheduled again, it will sleep because it was about to sleep when it was preempted.

The producer keep writing in the buffer and it got filled after some time. The producer will also sleep at that time keeping in the mind that the consumer will wake him up when there is a slot available in the buffer.

The consumer is also sleeping and not aware with the fact that the producer will wake him up.

Sleep and Wake

(Producer Consumer problem)

Let's examine the basic model that is sleep and wake. Assume that we have two system calls as **sleep** and **wake**. The process which calls sleep will get blocked while the process which calls will get waked up.

There is a popular example called **producer consumer problem** which is the most popular problem simulating **sleep and wake** mechanism.

The concept of sleep and wake is very simple. If the critical section is not empty then the process will go and sleep. It will be waked up by the other process which is currently executing inside the critical section so that the process can get inside the critical section.

In producer consumer problem, let us say there are two processes, one process writes something while the other process reads that. The process which is writing something is called **producer** while the process which is reading is called **consumer**.

This is a kind of deadlock where neither producer nor consumer is active and waiting for each other to wake them up. This is a serious problem which needs to be addressed.

Topic:7:5 Mutexes

Mutex Locks

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

```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (TRUE);
```

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

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

Acquire:

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

Release:

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

Topic 7:6 Semaphores:

Semaphores

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

Wait:

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

Signal:

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

Semaphore Usage

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

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

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

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

- **Counting semaphores** can take on any integer value, and are usually used to count the number remaining of some limited resource. The counter is initialized to the number of such resources available in the system, and whenever the counting semaphore is greater than zero, then a process can enter a critical section and use one of the resources. When the counter gets to zero (or negative in some implementations), then the process blocks until another process frees up a resource and increments the counting semaphore with a signal call. (The binary semaphore can be seen as just a special case where the number of resources initially available is just one.)
- Semaphores can also be used to synchronize certain operations between processes. For example, suppose it is important that process P1 execute statement S1 before process P2 executes statement S2.
  - First we create a semaphore named synch that is shared by the two processes, and initialize it to zero.
    - Then in process P1 we insert the code:

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

-
- and in process P2 we insert the code:

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

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

Semaphore Implementation

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

Semaphore Structure:

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

Wait Operation:

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

Signal Operation:

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

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

Deadlocks and Starvation

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

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

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

Priority Inversion

- A challenging scheduling problem arises when a high-priority process gets blocked waiting for a resource that is currently held by a low-priority process.
- If the low-priority process gets pre-empted by one or more medium-priority processes, then the high-priority process is essentially made to wait for the medium priority processes to finish before the low-priority process can release the needed resource, causing a **priority inversion.** If there are enough medium-priority processes, then the high-priority process may be forced to wait for a very long time.
- One solution is a **priority-inheritance protocol,** in which a low-priority process holding a resource for which a high-priority process is waiting will temporarily inherit the high priority from the waiting process. This prevents the medium-priority processes from preempting the low-priority process until it releases the resource, blocking the priority inversion problem.
- The book has an interesting discussion of how a priority inversion almost doomed the Mars Pathfinder mission, and how the problem was solved when the priority inversion was stopped.

Topic:7:7 Monitors

Monitors

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

5.8.1 Monitor Usage

- A monitor is essentially a class, in which all data is private, and with the special restriction that only one method within any given monitor object may be active at the same time. An additional restriction is that monitor methods may only access the shared data within the monitor and any data passed to them as parameters. I.e. they cannot access any data external to the monitor.

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

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

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

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

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

Figure 5.15 - Syntax of a monitor.

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

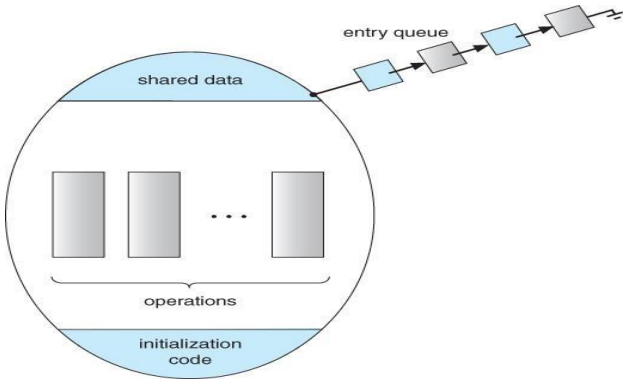


Figure 5.16 - Schematic view of a monitor

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

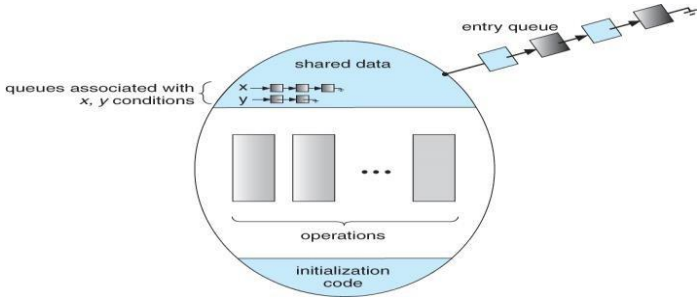


Figure 5.17 - Monitor with condition variables

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

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

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

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

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

Implementing a Monitor Using Semaphores

- One possible implementation of a monitor uses a semaphore "mutex" to control mutual exclusionary access to the monitor, and a counting semaphore "next" on which processes can suspend themselves after they are already "inside" the monitor (in conjunction with condition variables, see below.) The integer next_count keeps track of how many processes are waiting in the next queue. Externally accessible monitor processes are then implemented as:

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

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

Wait:

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

Signal:

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

Difference between Monitors and Semaphore

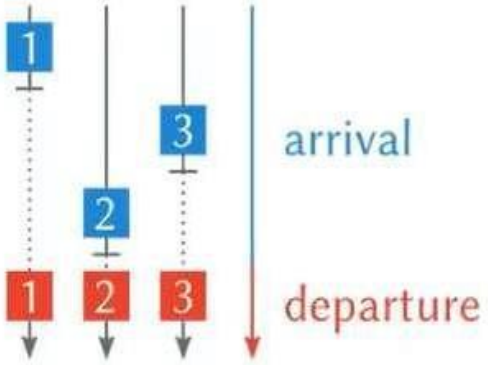
Monitors	Semaphore
We can use condition variables only in the monitors.	In semaphore, we can use condition variables anywhere in the program, but we cannot use conditions variables in a semaphore.
In monitors, wait always block the caller.	In semaphore, wait does not always block the caller.
The monitors are comprised of the shared variables and the procedures which operate the shared variable.	The semaphore S value means the number of shared resources that are present in the system.

Condition variables are present in the monitor.	Condition variables are not present in the semaphore.

Topic: 7:8 Barriers:

Barriers

- In parallel computing, a **barrier** is a type of synchronization method. A barrier can be used to synchronize a group of processes.
- A barrier means that the processes must stop at the barrier and cannot be allowed to proceed until all processes from the group reach the barrier.
- When the last process reaches the barrier, all processes can resume execution and continue from the barrier.



Topic:7:9 Classic Problems of Synchronization

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

5.7.1 The Bounded-Buffer Problem

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

nearly identical - One can think of the producer as producing full buffers, and the consumer producing empty buffers.

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

Figure 5.9 The structure of the producer process.

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

Figure 5.10 The structure of the consumer process.

Figures 5.9 and 5.10 use variables next_produced and next_consumed

The Readers-Writers Problem

- In the readers-writers problem there are some processes (termed readers) who only read the shared data, and never change it, and there are other processes (termed writers) who may change the data in addition to or instead of reading it. There is no limit to how many readers can access the data simultaneously, but when a writer accesses the data, it needs exclusive access.
- There are several variations to the readers-writers problem, most centered around relative priorities of readers versus writers.
    - The first readers-writers problem gives priority to readers. In this problem, if a reader wants access to the data, and there is not already a writer accessing it, then access is granted to the reader. A solution to this problem can lead to starvation of the writers, as there could always be more readers coming along to access the data. (A steady stream of readers will jump ahead of waiting writers as long as there is currently already another reader accessing the data, because the writer is forced to wait until the data is idle, which may never happen if there are enough readers.)

- The second readers-writers problem gives priority to the writers. In this problem, when a writer wants access to the data it jumps to the head of the queue - All waiting readers are blocked, and the writer gets access to the data as soon as it becomes available. In this solution the readers may be starved by a steady stream of writers.

- The following code is an example of the first readers-writers problem, and involves an important counter and two binary semaphores:

  - readcount is used by the reader processes, to count the number of readers currently accessing the data.
  - mutex is a semaphore used only by the readers for controlled access to readcount.
  - rw_mutex is a semaphore used to block and release the writers. The first reader to access the data will set this lock and the last reader to exit will release it; The remaining readers do not touch rw_mutex. (Eighth edition called this variable wrt.)
  - Note that the first reader to come along will block on rw_mutex if there is currently a writer accessing the data, and that all following readers will only block on mutex for their turn to increment readcount.

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

Figure 5.11 The structure of a writer process.

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

Figure 5.12 The structure of a reader process.

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

The Dining-Philosophers Problem

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

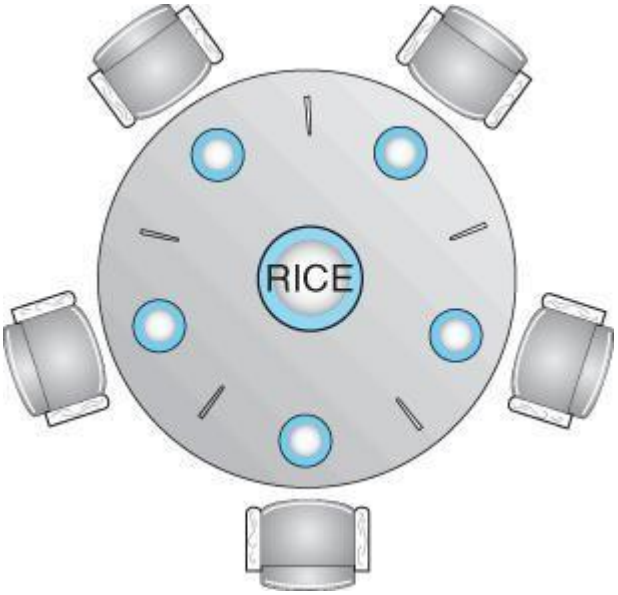


Figure 5.13 - The situation of the dining philosophers

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

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

Figure 5.14 - The structure of philosopher i.

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