UNIT-2(Concurrent Processes)

PROCESS CONCEPT

Early computer systems allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, current day computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a **process**, which is a program in execution. A process is the unit of work in a modern time-sharing system. Even on a single-user system such as Microsoft Windows, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package.

A process is a program in execution. It is an instance of a class. Process is not as same as program code but a lot more than it. A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity. Attributes held by process include hardware state, memory, CPU etc.

Process memory is divided into four sections for efficient working:

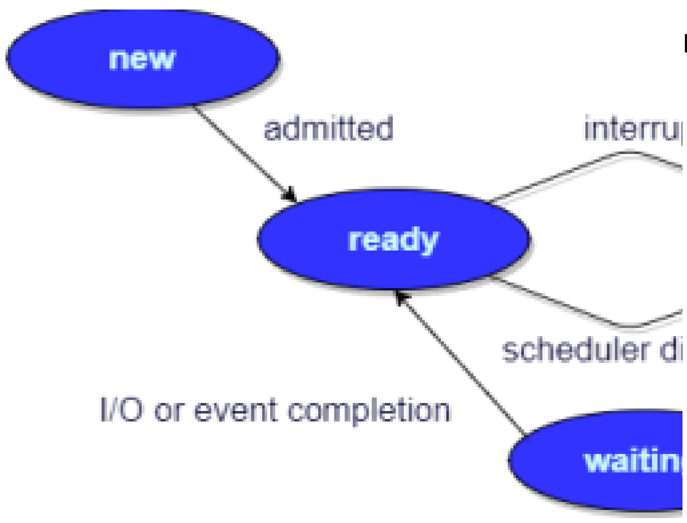
- The text section is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- The data section is made up the global and static variables, allocated and initialized prior to executing the main.
- The heap is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The stack is used for local variables. Space on the stack is reserved for local variables when they are declared.

PROCESS STATE

Processes can be any 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(such as an I/O completion or reception of a signal).
- **Terminated** - The process has finished execution.

Suspend



Resume Suspend Resume

UNIT-2(Concurrent Processes)

Some More Insight to Process State

Created: When a process is first created, it occupies the "created" or "new" state. In this state, the process awaits admission to the "ready" state. This admission will be approved or delayed by a long-term, or admission, scheduler.

Ready: A "ready" process has been loaded into main memory and is awaiting execution on a CPU (to be context switched onto the CPU by the dispatcher, or short-term scheduler). There may be many "ready" processes at any one point of the systems execution - for example, in a one processor system, only one process can be executing at any one time, and all other "concurrently executing" processes will be waiting for execution.

Running: A process moves into the running state when it is chosen for execution. The process's instructions are executed by one of the CPUs (or cores) of the system. There is at most one running process per CPU or core.

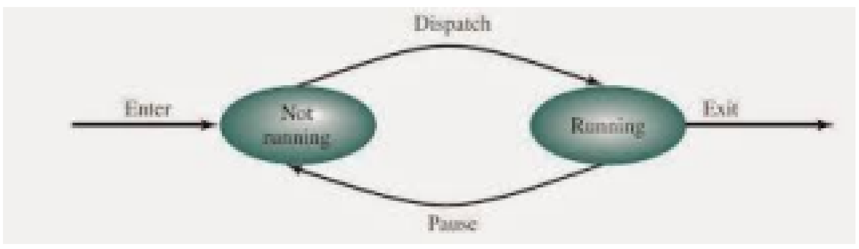
Waiting: A process that is blocked on some event (such as I/O operation completion or a signal).

Terminated: A process may be terminated, either from the "running" state by completing its execution or by explicitly being killed. In either of these cases, the process moves to the "terminated" state. If a process is not removed from memory after entering this state, this state may also be called zombie.

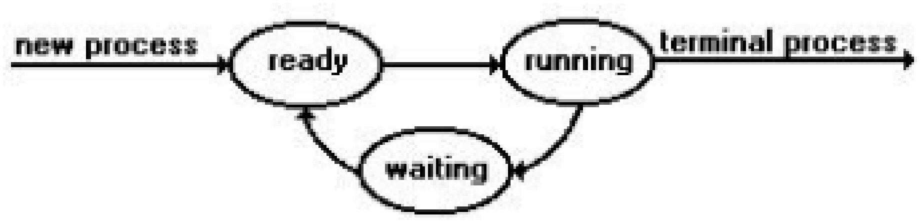
Additional process states

Two additional states are available for processes in systems that support virtual memory. In both of these states, processes are "stored" on secondary memory (typically a hard disk).

Two-State Model

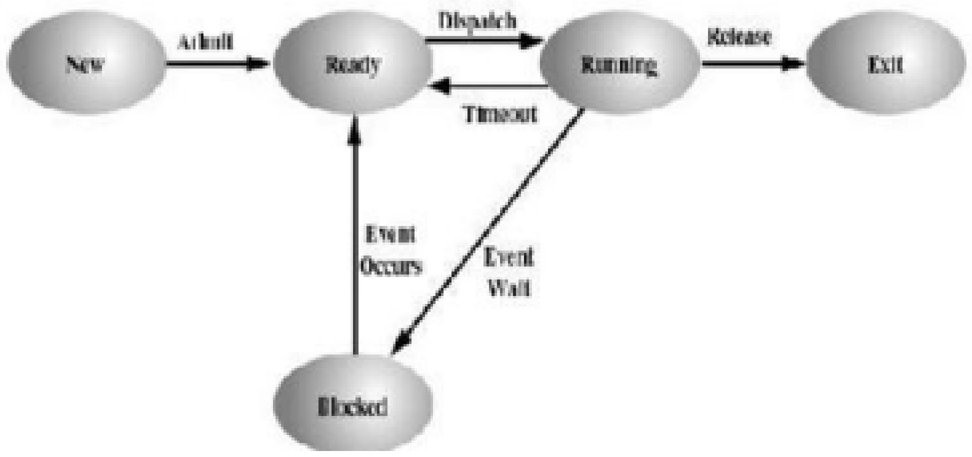


Three-State Model



UNIT-2(Concurrent Processes)

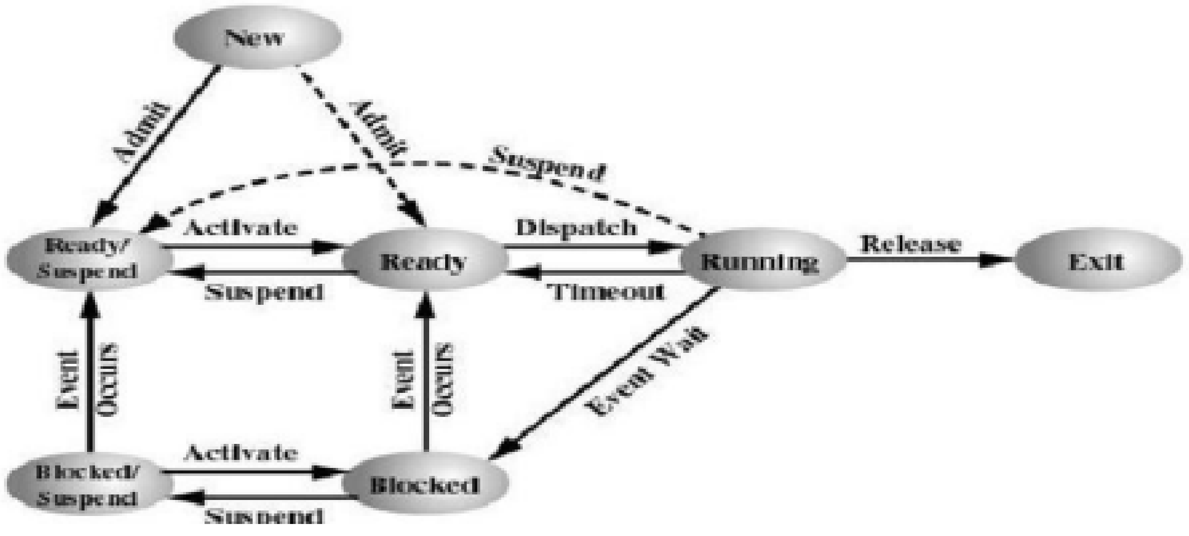
Five-State Model



Seven State Process Model

Seven state process model contains seven states for execution of processes:

1. **New : –** contains the processes which are newly coming for execution.
2. **Ready : –** contains the processes which are in main memory and available for execution. 3. **Running : –** contains the process which is running or executing.
4. **Exit : –** contains the processes which are completely executed.
5. **Blocked : –** contains the processes which are in main memory and awaiting an event occurrence.
6. **Blocked Suspend : –** contains the processes which are in secondary memory and awaiting an event occurrence.
7. **Ready Suspend : –** contains the processes which are is in secondary memory but is available for execution as soon as it is loaded into main memory.



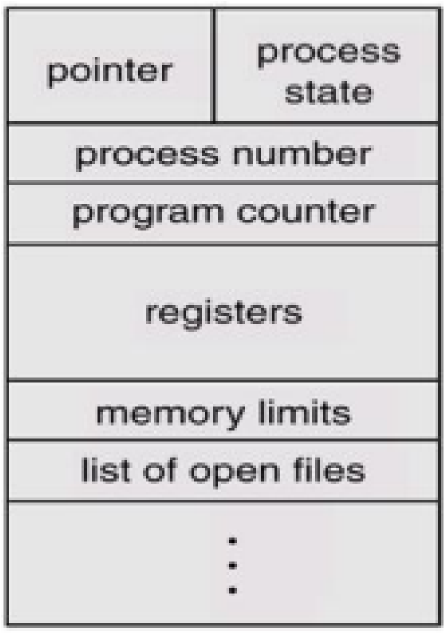
UNIT-2(Concurrent Processes)

Process Control Block

Each process is represented in the operating system by a **process control block (PCB)**—also called a *task control block*. A PCB is shown in Figure. It contains many pieces of information associated with a specific process, including these:
Information associated with each process.

- Process ID
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process state: The state may be new, ready, running, waiting, halted, and so on. *Program counter:* The counter indicates the address of the next instruction to be executed for this process.
CPU registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.



CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
Memory-management information: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
Accounting information: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
I/O status information: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

UNIT-2(Concurrent Processes)

Types of Processes

I/O- bound process: Spends more time doing I/O than computations, many short CPU bursts.

CPU-bound process: Spends more time doing computations; few very long CPU bursts.

Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these operating systems must provide a mechanism for process creation and termination.

- **Process Creation:**

Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
Resource sharing

- Parent and children share all resources.
- Children share subset of parent’s resources.
- Parent and child share no resources.

Execution

- Parent and children execute concurrently.

• Parent waits until children terminate.

Address space

• Child duplicate of parent.

• Child has a program loaded into it.

UNIX examples

▪ **fork** system call creates new process

▪ **fork** returns 0 to child , process id of child for parent

▪ **exec** system call used after a **fork** to replace the process' memory space with a new program.

Unix Program

```
#include <stdio.h >
main(int argc, char *argv[])
 {
 int pid;
 pid=fork(); /* fork another process */
 if (pid == 0)
 { /* child */
 exclp("/bin/ls","ls",NULL);
 }
 else
 { /* parent */
 wait(NULL); /* parent waits for child */
 printf("Child complete\n");
 exit(0);
 }
 }
```

Process Termination:

• Process executes last statement and asks the operating system to delete it (**exit**). o Output data from child to parent (via **wait**).

o Process' resources are deallocated by operating system.

• Parent may terminate execution of children processes (**abort**).

o Child has exceeded allocated resources.

o Task assigned to child is no longer required.

o Parent is exiting.

▪ Operating system does not allow child to continue if its parent terminates. ▪ Cascading termination.

o In Unix, if parent exits children are assigned **init** as parent

Context Switch

When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process. Context-switch time is overhead; the system does no useful work while switching. It is time dependent on hardware support.

When you return back to a process, you should resume where you left off. For all practical purposes, this process should never know there was a switch, and it should look like this was the only process in the system.

To implement this, on a context switch, you have to

- save the context of the current process
- select the next process to run
- Restore the context of this new process.

Context of a process

- Program Counter
- Stack Pointer
- Registers
- Code + Data + Stack (also called Address Space)
- Other state information maintained by the OS for the process (open files, scheduling info, I/O devices being used etc.)

All this information is usually stored in a structure called Process Control Block (PCB).

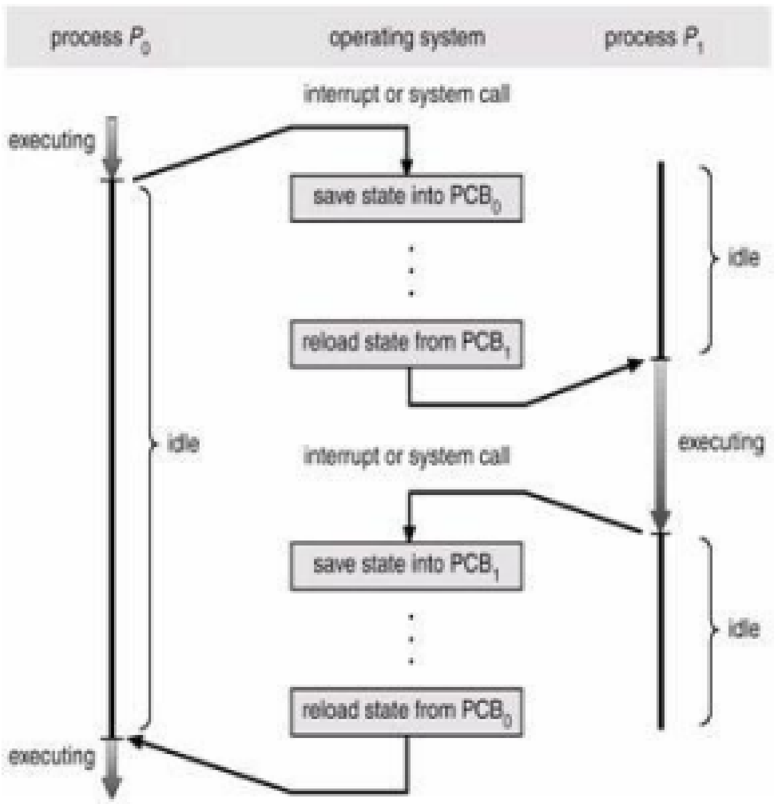
```
context_switch()
{
Push registers onto stack
Save ptrs to code and data.
Save stack pointer

Pick next process to execute

Restore stack ptr of that process /* You have now switched the stack */
Restore ptrs to code and data.
Pop registers

Return
}
```

CPU switch from process to process



Process Scheduling Queues

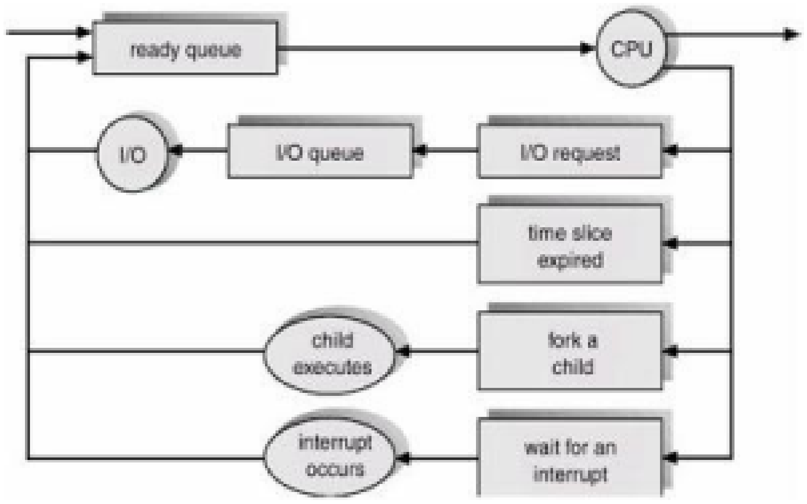
The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. A contemporary computer system maintains many scheduling queues. Here is a brief description of some of these queues:

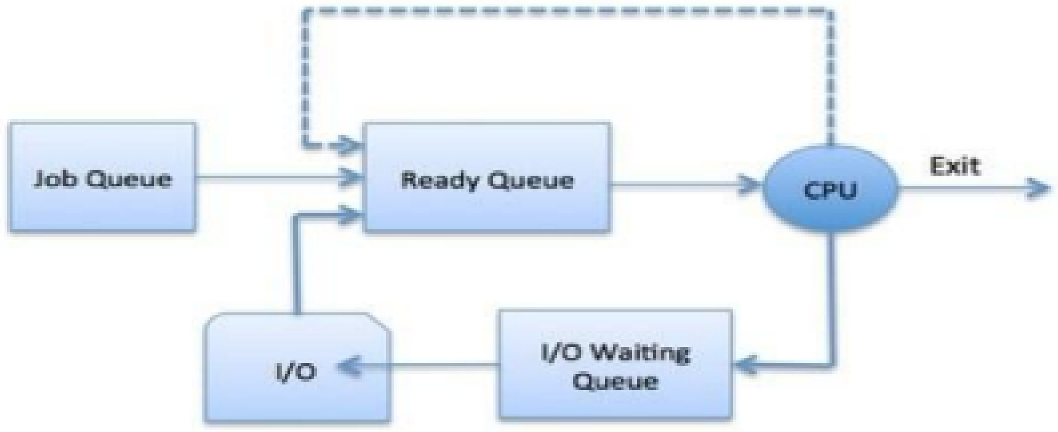
Job Queue: As processes enter the system, they are put into a job queue. This queue consists of all processes in the system.

Ready Queue: The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB is extended to include a pointer field that points to the next PCB in the ready queue.

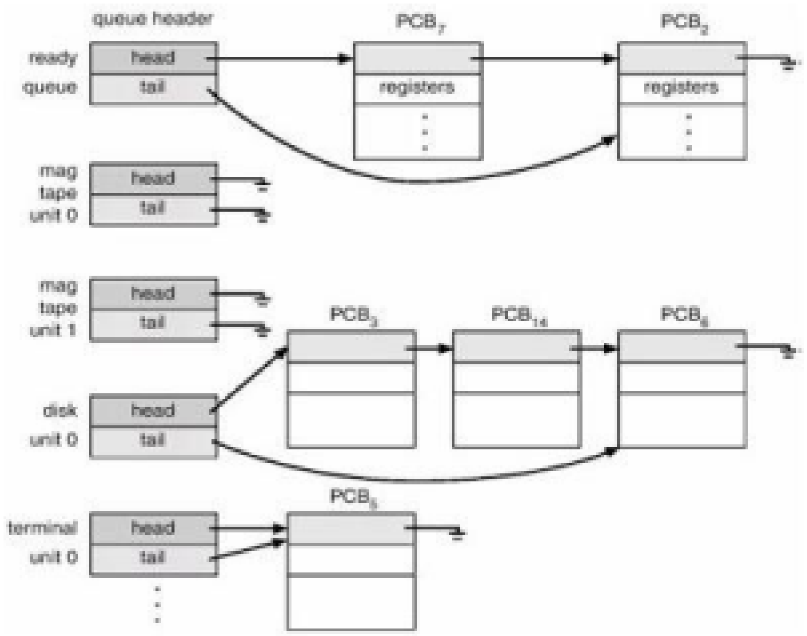
Device Queue: When a process is allocated the CPU, it executes for a while, and eventually quits, is interrupted or waits for a particular event, such as completion of an I/O request. In the case of an I/O request, the device may be busy with the I/O request of some other process, hence the list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.

Representation of a process scheduling





Ready queue and various I/O device queues



Scheduling

Scheduling refers to the way processes are assigned to run on the available CPUs, since there are typically many more processes running than there are available CPUs. This assignment is carried out by software known as a **scheduler** and **dispatcher**.

The CPU scheduling decisions may take place when a process:

1. switches from the running to the waiting state
2. switches from the running to the ready state
3. switches from the waiting to the ready state
4. terminates

· Types of operating system schedulers

Operating systems may feature up to 3 distinct types of a long-term scheduler (also known as an admission scheduler or high-level scheduler), a mid-term or medium-term scheduler and a short term scheduler. The names suggest the relative frequency with which these functions are performed.

Long-term scheduler

The long-term, or admission scheduler, decides which jobs or processes are to be admitted to the ready queue. Thus, this scheduler dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time - i.e.: whether a high or low amount of processes are to be executed concurrently, and how the split between IO intensive and CPU intensive processes is to be handled.

Mid-term scheduler

The mid-term scheduler temporarily removes processes from main memory and places them on secondary memory (such as a disk drive) or vice versa. This is commonly referred to as "swapping out" or "swapping in".

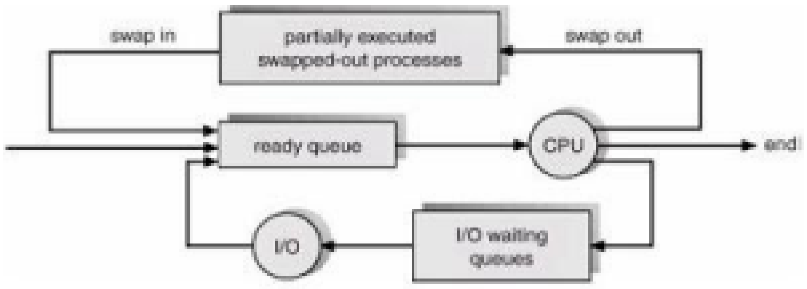
The mid-term scheduler may decide to swap out

1. A process which has not been active for some time.
2. A process which has a low priority.
3. A process which is page faulting frequently.
4. A process which is taking up a large amount of memory in order to free up main memory for other processes.

Swapping the process back in later

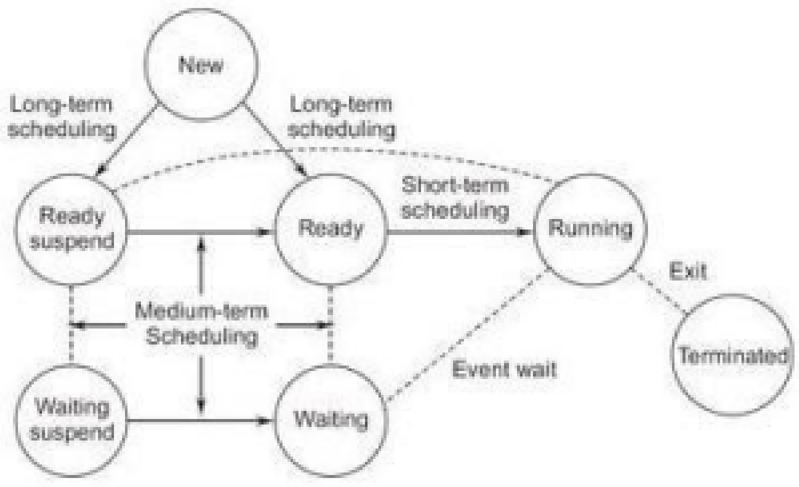
1. When more memory is available.
2. When the process has been unblocked and is no longer waiting for a resource

Action of a medium term scheduling



Short-term scheduler

The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in memory processes are to be executed (allocated a CPU) next following a clock interrupt, an IO interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive (also known as "voluntary" or "co-operative"), in which case the scheduler is unable to "force" processes off the CPU.



Dispatcher

Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

1. Switching context
2. Switching to user mode
3. Jumping to the proper location in the user program to restart that program The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

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

Types of Scheduling

Preemptive Scheduling:
Preemption (sometimes pre-emption) is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time. Such a change is known as a context switch. It is normally carried out by a privileged task or part of the system known as a preemptive scheduler, which has the power to preempt, or interrupt, and later resume, other tasks in the system.
In preemptive scheduling, a scheduler executes jobs in the following two situations.

1. When a process switches from the running state to the ready state. 2. When a

process switches from the waiting state to the ready state. **Non pre-emptive**

Scheduling

A scheduling discipline is nonpreemptive if, once a process has been given the CPU; the CPU cannot be taken away from that process.

Following are some characteristics of nonpreemptive scheduling
1. In non-preemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
2. In non-preemptive system, response times are more predictable because incoming high priority jobs cannot displace waiting jobs.
3. In non-preemptive scheduling, a scheduler executes jobs in the following two situations. o
   When a process switches from running state to the waiting state.
   - o When a process terminates.

Scheduling Criteria

CPU utilization: Keeping the CPU as busy as possible. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent.

Throughput: The number of processes that complete their execution per time unit. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called **throughput**. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

Turnaround time: The amount of time it takes to execute a particular process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Waiting time: Amount of time a process has been in the waiting queue. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It

affects only the amount of time that a process spends waiting in the ready queue.Waiting time is the sum of the periods spent waiting in the ready queue.

Response time: Amount of time it takes from when a request was submitted until the first response is produced, not output - for time-sharing environments. a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

Optimizing Criteria

Each scheduling algorithm must be evaluated from how it optimizes the following variables:

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

SCHEDULING ALGORITHM

First Come First Serve

- Non-preemptive scheduling algorithm.
- Follow first in first out (FIFO) method.
- FIFO strategy assigns priority to processes in the order in which they request the processor.
- When a process comes in, add its PCB to the tail of ready queue.
- Next process will be taken from head of the queue.
- Dispatcher selects first job in queue and this job runs to completion of CPU burst. • Running process does not give up the CPU until it terminates or it performs IO

Advantages

- Better for long processes
- Simple method (i.e., minimum overhead on processor)
- No starvation

Disadvantages

- Inappropriate for interactive systems
- Large fluctuations in average turnaround time are possible.
- Convoy effect occurs. Even very small process should wait for its turn to come to utilize the CPU. Short process behind long process results in lower CPU utilization.
- Throughput is not emphasized.

CONVOY EFFECT

CPU bound jobs will hold CPU until exit or I/O (but I/O rare for CPU-bound thread)

- long periods where no I/O requests issued, and CPU held
- Result: poor I/O device utilization

Consider: P1: CPU-bound
P2, P3, P4: I/O-bound

- P2, P3 and P4 could quickly finish their IO request → ready queue, waiting for CPU. • Note: IO devices are idle then.
- Then P1 finishes its CPU burst and move to an IO device.
- P2, P3, P4, which have short CPU bursts, finish quickly → back to IO queue. • Note: CPU is idle then.
- P1 moves then back to ready queue is gets allocated CPU time.
- Again P2, P3, P4 wait behind P1 when they request CPU time.

Example 1

PROCESS	**ARRIVAL TIME**	**BURST TIME**
P1	0	24
P2	0	3
P3	0	3

Gantt chart:

P_1	P_2	P_3

0 24 27 30

PROCESS	WAIT TIME	TURN AROUND TIME
P1	0	24
P2	24	27
P3	27	30

Total Wait Time: 0 + 24 + 27 = 51 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)=51/3 = 17 ms

Total Turn Around Time=24 + 27 + 30 = 81 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)=81 / 3 = 27 ms
Throughput=3 jobs/30 sec = 0.1 jobs/sec

Example 2

In the above example, if order of process arriving is p2, p3, p1 instead of p1, p2, p3 then

Gantt chart

P_2	P_3	P_1

0 3 6 30

PROCESS	WAIT TIME	TURN AROUND TIME
P1	6	30
P2	0	3
P3	3	6

Total Wait Time=6 + 0 + 3 = 9 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)=9/3 = 3 ms

Total Turn Around Time=30 + 3 + 6 = 39 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)= 39 / 3 = 13 ms

Throughput= 3 jobs/30 sec = 0.1 jobs/sec

Thus, it can be clearly stated that scheduling of processes can significantly reduce waiting and turn around time.

Example 3

PROCESS	ARRIVAL TIME	BURST TIME
P1	0	80
P2	0	20
P3	0	10
P4	0	20
P5	0	50

Gantt chart

P_1	P_2	P_3	P_4	P_5

0 80 100 110 130 180

PROCESS	WAIT TIME	TURN AROUND TIME
P1	0	80
P2	80	100
P3	100	110

P4	110	130
P5	130	180

Total Wait Time=0 + 80 + 100 + 110 + 130 = 420 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)=420/5 = 84 ms

Total Turn Around Time= 80 + 100 + 110 + 130 + 180 = 600 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)=600/5 = 120 ms

Throughput = 5 jobs/180 sec = 0.2778 jobs/sec

Example 4

PROCESS	ARRIVAL TIME	SERVICE TIME
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Gantt chart

P_1	P_2	P_3	P_4

0 8 12 21 26

PROCESS	WAIT TIME	TURN AROUND TIME
P1	0	8 – 0 = 8
P2	8 – 1 = 7	12 – 1 = 11

P3	12 – 2 = 10	21 – 2 = 19
P4	21 – 3 = 18	26 – 3 = 23

Total Wait Time= 0 + 7 + 10 + 18 = 35 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)=35/4 = 8.75 ms

Total Turn Around Time=8 + 11 + 19 + 23 = 61 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)=61/4 = 15.25 ms
Throughput=4 jobs/26 sec = 0.15385 jobs/sec

Shortest Job First

- Non-preemptive - once CPU given to the process it cannot be preempted until completes its CPU burst
- Ready queue is treated as a priority queue based on smallest CPU time requirement
    - Priorities are assigned in inverse order of time needed for completion of the entire job
    - If equal time of completion, then FCFS is used for assigning priority.
    - Arriving jobs inserted at proper position in queue
    - Dispatcher selects shortest job (1st in queue) and runs to completion
- Gives minimum average waiting time for a given set of processes.
- Minimizes average turnaround time.
- When multiple batch jobs are sitting in a queue with the same priority, the scheduler runs the shortest job first.
- It cannot be implemented at the level of short term CPU scheduling.
- A job exceeding the resource estimation is aborted.
- Store estimated value in PCB for the current burst, and compare with actual value.
- Exponential averaging is used to estimate the process' burst duration.

Advantages

- Minimizes average waiting time.
- Provably optimal w.r.t. average turnaround time
- Throughput is high.

Disadvantages

- In general, cannot be implemented.
    - Requires future knowledge
    - In practice, can't actually predict the length of next burst
- Can lead to unfairness or starvation
    - It may penalize processes with high service time requests. If the ready list is saturated, then processes with large service times tend to be left in the ready list while small processes receive service. In extreme case, where the system has little

idle time, processes with large service times will never be served. This total starvation of large processes may be a serious liability of this algorithm.

- Doesn't always minimize average turnaround time
- Elapsed time (i.e., execution-completed-time) must be recorded, it results an additional overhead on the processor.

Exponentially weighted average

We can estimate CPU burst length based on past by using an **Exponentially weighted average** Make an estimate based on the past behavior. Say the estimated time (burst) for a process is Γ_0, suppose the actual time is measured to be T_0. Update the estimate by taking a weighted sum of these two

ie. $\Gamma_1 = \alpha T_0 + (1-\alpha)\,\Gamma_0$

In general,

$\Gamma_{(n+1)} = \alpha T_n + (1-\alpha)\,\Gamma_n$ (Exponential average)

$\Gamma_{(n+1)}$: Predicted length of next CPU burst

Γ_n : Previous prediction

T_n :: Actual length of last CPU burst.

α : Weighting factor

$0 \leq \alpha \leq 1$

if $\alpha=0$

- Recent history not considered
- $\Gamma_{(n+1)} = \Gamma_n$

if $\alpha=1$

- Past history not considered
- Only the actual last CPU burst counts.
- $\Gamma_{(n+1)} = T_n$

if $\alpha=1/2$

- Implies weighted (older bursts get less and less weight).

If we expand the formula, we will get,

$\Gamma_{(n+1)} = \alpha\, T_n + (1-\alpha)\,\alpha\, T_{n-1} + \dots + (1-\alpha)^j\,\alpha\, T_{n-j} + \dots$

α and $(1 - \alpha)$ are ≤ 1, so each successive term has less weight than its predecessor.

So, older information has less weightage

Example 1

PROCESS	ARRIVAL TIME	BURST TIME
P1	0	24

P2	0	3
P3	0	3

Gantt chart

P_2	P_3	P_1

0 3 6 30

PROCESS	WAIT TIME	TURN AROUND TIME
P1	6	30
P2	0	3
P3	3	6

Total Wait Time=6 + 0 + 3 = 9 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)=9/3 = 3 ms

Total Turn Around Time= 30 + 3 + 6 = 39 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)=39 / 3 = 13 ms

Throughput= 3 jobs/30 sec = 0.1 jobs/sec

If solved using the approach of FCFS scheduling, then both average waiting time and average turn around time will increase. Refer Example 1 of FCFS.

Thus, SJF scheduling reduces average waiting time and average turn around time.

Example 2

PROCESS	ARRIVAL TIME	BURST TIME
P1	0	80
P2	0	20
P3	0	10
P4	0	20

P5	0	50

Gantt chart

P_3	P_2	P_4	P_5	P_1

0 10 30 50 100 180

PROCESS	WAIT TIME	TURN AROUND TIME
P1	100	180
P2	10	30
P3	0	10
P4	30	50
P5	50	100

Total Wait Time=100 + 10 + 0 + 30 + 50 = 190 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)=190/5 = 38 ms

Total Turn Around Time= 180 + 30 + 10 + 50 + 100 = 370 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)=370/5 = 74 ms
Throughput= 5 jobs/180 sec = 0.2778 jobs/sec

If solved using the approach of FCFS scheduling, then average waiting time will increase but average turn around time will decrease. Refer Example 3 of FCFS.

Thus, SJF scheduling reduces average waiting time and average turn around time.

Thus, it can be concluded that, average waiting time will be always reduced, but average turn around time may or may not reduce. i.e. It may increase

Example 3

PROCESS	ARRIVAL TIME	SERVICE TIME
P1	0	8
P2	1	4
P3	2	9

P4	3	5

Gantt chart

P_1	P_2	P_4	P_3

0 8 12 17 26

PROCESS	WAIT TIME	TURN AROUND TIME
P1	0	8 – 0 = 8
P2	8 – 1 = 7	12 – 1 = 11
P3	17 – 2 = 15	26 – 2 = 24
P4	12 – 3 = 9	17 – 3 = 14

Total Wait Time= 0 + 7 + 15 + 9 = 31 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)= 31/4 = 7.75 ms

Total Turn Around Time= 8 + 11 + 24 + 14 = 57 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)=57/4 = 14.25 ms

Throughput= 4 jobs/26 sec = 0.15385 jobs/sec

Example 4

PROCESS	ARRIVAL TIME	SERVICE TIME
P1	0	7
P2	2	4
P3	4	1

P4	5	4

Gantt chart

P_1	P_3	P_2	P_4

0 7 8 12 16

PROCESS	WAIT TIME	TURN AROUND TIME
P1	0	7 – 0 = 7
P2	8 – 2 = 6	12 – 2 = 10
P3	7 – 4 = 3	8 – 4 = 4
P4	12 – 5 = 7	16 – 5 = 11

Total Wait Time= 0 + 6 + 3 + 7 = 16 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)=16/4 = 4 ms

Total Turn Around Time= 7 + 10 + 4 + 11 = 32 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)=32/4 = 8 ms

Throughput=4 jobs/16 sec = 0.25 jobs/sec

Preemptive SJF (Shortest Remaining Time First)

• Shortest remaining time first(SRTF) is the preemptive version of the SJF algorithm. • When a process arrives to Request Queue, sort it in and select the SJF including the running process, possibly interrupting it (Remember: SJF schedules a new process only when the running is finished)

• If a new process arrives with CPU burst length less than remaining time of current executing process, preempt the CPU away from the currently executing process when a higher priority process is ready.

• On time sharing machines, this type of scheme is required because the CPU must be protected from a run-away low priority process.

• Give short jobs a higher priority – perceived response time is thus better. • Impossible to implement in interactive systems where required CPU time is not known. • It is often used in batch environments where short jobs need to give preference.

Example 1

PROCESS	ARRIVAL TIME	SERVICE TIME
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Gantt chart

P_1	P_2	P_4	P_1	P_3

0 1 5 10 17 26

PROCESS	WAIT TIME	TURN AROUND TIME
P1	10 – 1 = 9	17 – 0 = 17
P2	1 – 1 = 0	5 – 1 = 4
P3	17 – 2 = 15	26 – 2 = 24
P4	5 – 3 = 2	10 – 3 = 7

Total Wait Time= 9 + 0 + 15 + 2 = 26 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)= 26/4 = 6.5 ms

Total Turn Around Time= 17 + 4 + 24 + 7 = 46 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)= 46/4 = 11.5 ms

Throughput= 4 jobs/26 sec = 0.15385 jobs/sec

Example 2

PROCESS	ARRIVAL TIME	SERVICE TIME
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Gantt chart

P_1	P_2	P_3	P_2	P_4	P_1

0 2 4 5 7 11 16

PROCESS	WAIT TIME	TURN AROUND TIME
P1	11 – 2 = 9	16 – 0 = 16
P2	(2 – 2) + (11 – 4) = 7	7 – 2 = 5
P3	4 – 4 = 0	5 – 4 = 1
P4	7 – 5 = 2	11 – 5 = 6

Total Wait Time= 9 + 7 + 0 + 2 = 18 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)= 18/4 = 4.5 ms

Total Turn Around Time= 16 + 5 + 1 + 6 = 28 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)= 28/4 = 7 ms

Throughput= 4 jobs/16 sec = 0.25 jobs/sec

Example 3

PROCESS	ARRIVAL TIME	SERVICE TIME
P1	0	7
P2	2	6
P3	4	1
P4	6	2

Gantt chart

P_1	P_2	P_3	P_2	P_4	P_2	P_1

0 2 4 5 6 8 11 16

PROCESS	WAIT TIME	TURN AROUND TIME

P1	11 – 2 = 9	16 – 0 = 16
P2	(2 – 2) + (5 – 4) + (11 – 6) = 6	11 – 2 = 9
P3	4 – 4 = 0	5 – 4 = 1
P4	6 – 6 = 0	8 – 6 = 2

Total Wait Time= 9 + 6 + 0 + 0 = 15 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)= 15/4 = 3.75 ms

Total Turn Around Time=16 + 9 + 1 + 2 = 28 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)=28/4 = 7 ms

Throughput= 4 jobs/16 sec = 0.25 jobs/sec

Priority Scheduling

• The SJF algorithm is a special case of the general **priority scheduling algorithm**. • Priority scheduling can be either preemptive or non-preemptive.

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

• Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4095.

• There is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority.

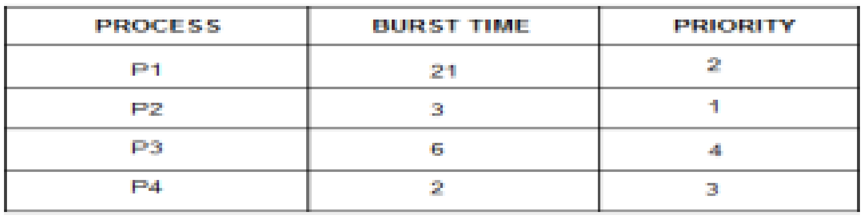
• Priority can be decided based on memory requirements, time requirements or any other resource requirement.

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

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

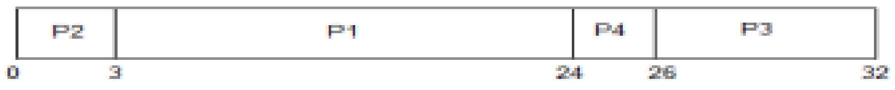
  - o If this problem is allowed to occur, then processes will either run eventually when the system load lightens (at say 2:00 a.m.), or will eventually get lost when the system is shut down or crashes. (There are rumors of jobs that have been stuck for years.)

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



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

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



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

Advantages:

- The priority of a process can be selected based on memory requirement, time requirement or user preference. For example, a high end game will have better graphics, that means the process which updates the screen in a game will have higher priority so as to achieve better graphics performance.

Disadvantages:

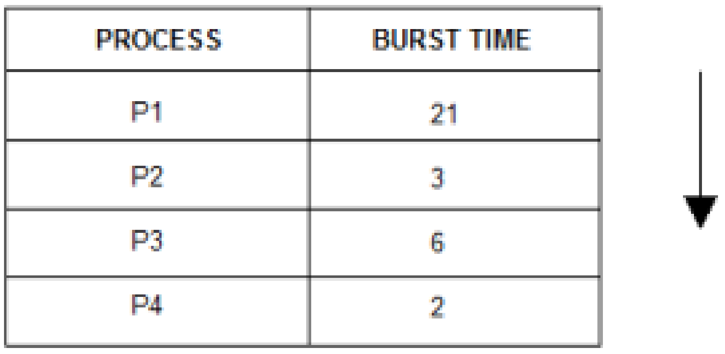
- A second scheduling algorithm is required to schedule the processes which have same priority.
- In preemptive priority scheduling, a higher priority process can execute ahead of an already executing lower priority process. If lower priority process keeps waiting for higher priority processes, starvation occurs.

Round Robin (RR)

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

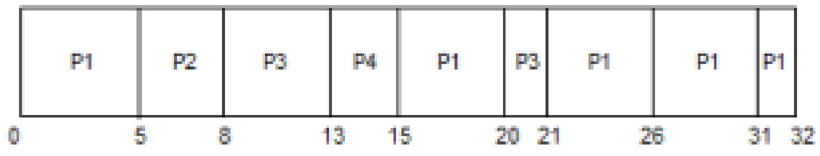
The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally.

- **BUT**, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches there are.



PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

Advantages:

- Each process is served by the CPU for a fixed time quantum, so all processes are given the same priority.
- Starvation doesn't occur because for each round robin cycle, every process is given a fixed time to execute. No process is left behind.

Disadvantages:

- The throughput in RR largely depends on the choice of the length of the time quantum. If time quantum is longer than needed, it tends to exhibit the same behavior as FCFS. • If time quantum is shorter than needed, the number of times that CPU switches from one process to another process, increases. This leads to decrease in CPU efficiency.

Multilevel Queue Scheduling

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

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

A multi-level queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm.

For example: separate queues might be used for foreground and background processes. The foreground queue might be scheduled by Round Robin algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. **For example:** The foreground queue may have absolute priority over the background queue.

Let us consider an example of a multilevel queue-scheduling algorithm with five queues:

1. System Processes
2. Interactive Processes
3. Interactive Editing Processes
4. Batch Processes
5. Student Processes

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

Multilevel Feedback Queue Scheduling

In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. This setup has the advantage of low scheduling overhead, but the disadvantage of being inflexible.

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to separate processes with different CPU-burst characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

An example of a multilevel feedback queue can be seen in the figure.

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

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

The definition of a multilevel feedback queue scheduler makes it the most general CPU scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it also requires some means of selecting values for all the parameters to define the best scheduler. Although a multilevel feedback queue is the **most general scheme**, it is also the **most complex**.

Multiple-Processor Scheduling

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

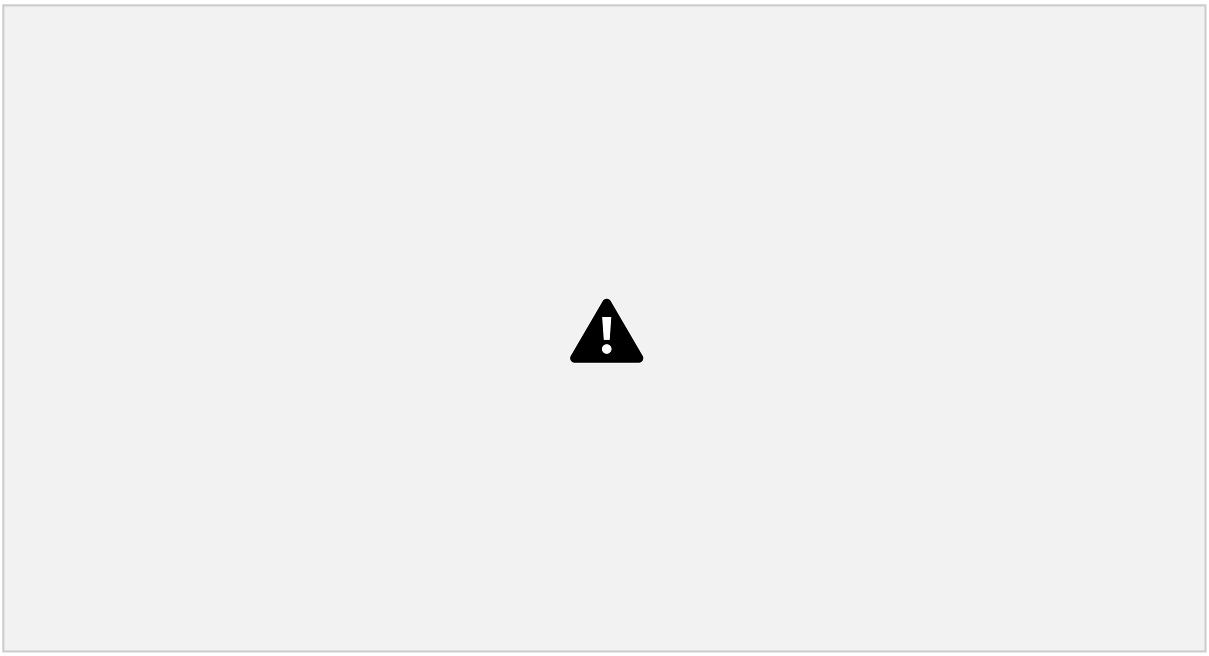
Approaches to Multiple-Processor Scheduling

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

Processor Affinity

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

a particular CPU, then it should preferentially be assigned memory storage in "local" fast access areas.



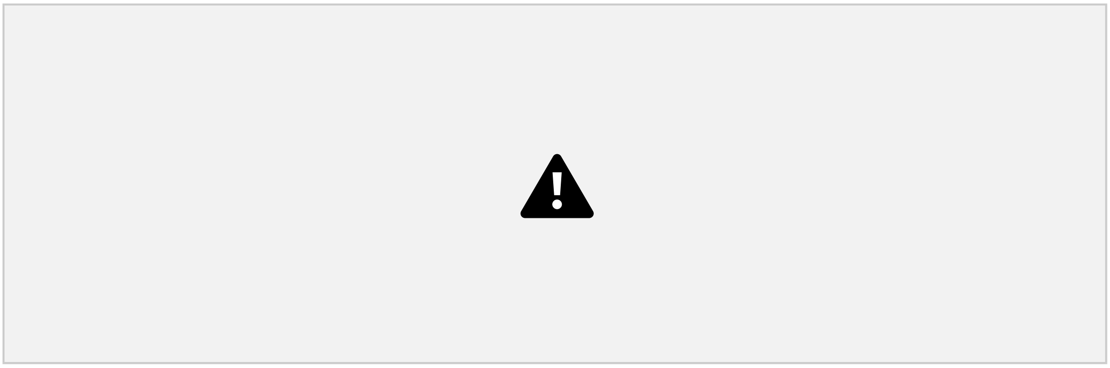
NUMA and CPU scheduling

Load Balancing

• Obviously an important goal in a multiprocessor system is to balance the load between processors, so that one processor won't be sitting idle while another is overloaded. • Systems using a common ready queue are naturally self-balancing, and do not need any special handling. Most systems, however, maintain separate ready queues for each processor.
• Balancing can be achieved through either ***push migration*** or ***pull migration:*** o ***Push migration*** involves a separate process that runs periodically, (e.g. every 200 milliseconds), and moves processes from heavily loaded processors onto less loaded ones.
o ***Pull migration*** involves idle processors taking processes from the ready queues of other processors.
o Push and pull migration are not mutually exclusive.
• Note that moving processes from processor to processor to achieve load balancing works against the principle of processor affinity, and if not carefully managed, the savings gained by balancing the system can be lost in rebuilding caches. One option is to only allow migration when imbalance surpasses a given threshold.

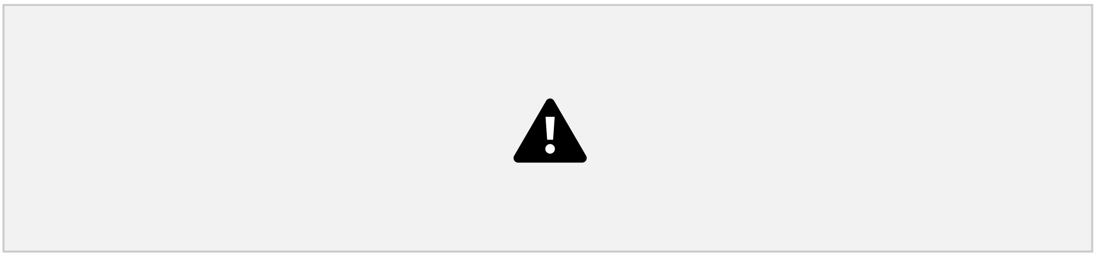
Multicore Processors

• Traditional SMP required multiple CPU chips to run multiple kernel threads concurrently. • Recent trends are to put multiple CPUs (cores) onto a single chip, which appear to the system as multiple processors.
• Compute cycles can be blocked by the time needed to access memory, whenever the needed data is not already present in the cache. (Cache misses.) In Figure 5.10, as much as half of the CPU cycles are lost to memory stall.



Memory stall

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



Multithreaded multicore system

- A dual-threaded dual-core system has four logical processors available to the operating system. The UltraSPARC T1 CPU has 8 cores per chip and 4 hardware threads per core, for a total of 32 logical processors per chip.
- There are two ways to multi-thread a processor:
    1. ***Coarse-grained*** multithreading switches between threads only when one thread blocks, say on a memory read. Context switching is similar to process switching, with considerable overhead.
    2. ***Fine-grained*** multithreading occurs on smaller regular intervals, say on the boundary of instruction cycles. However the architecture is designed to support thread switching, so the overhead is relatively minor.
- Note that for a multi-threaded multi-core system, there are **two** levels of scheduling, **at the kernel level:**
    - The OS schedules which kernel thread(s) to assign to which logical processors, and when to make context switches using algorithms as described above.
    - On a lower level, the hardware schedules logical processors on each physical core using some other algorithm.
        - The UltraSPARC T1 uses a simple round-robin method to schedule the 4 logical processors (kernel threads) on each physical core.
        - The Intel Itanium is a dual-core chip which uses a 7-level priority scheme (urgency) to determine which thread to schedule when one of 5 different events occurs.

Algorithm Evaluation

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

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

Deterministic Modeling

- If a specific workload is known, then the exact values for major criteria can be fairly easily calculated, and the "best" determined. For example, consider the following workload (with all processes arriving at time 0), and the resulting schedules determined by three different algorithms:

Process	**Burst Time**
P1	10
P2	29
P3	3
P4	7
P5	12

FCFS:

Non-preemptive SJF:

Round Robin:

• The average waiting times for FCFS, SJF, and RR are 28ms, 13ms, and 23ms respectively. • Deterministic modeling is fast and easy, but it requires specific known input, and the results only apply for that particular set of input. However by examining multiple similar cases, certain trends can be observed. (Like the fact that for processes arriving at the same time, SJF will always yield the shortest average wait time.)

Queuing Models

- Specific process data is often not available, particularly for future times. However a study of historical performance can often produce statistical descriptions of certain important parameters, such as the rate at which new processes arrive, the ratio of CPU bursts to I/O times, the distribution of CPU burst times and I/O burst times, etc.
- Armed with those probability distributions and some mathematical formulas, it is possible to calculate certain performance characteristics of individual waiting queues. For example, **Little's Formula** says that for an average queue length of N, with an average waiting time in the queue of W, and an average arrival of new jobs in the queue of Lambda, then these three terms can be related by:

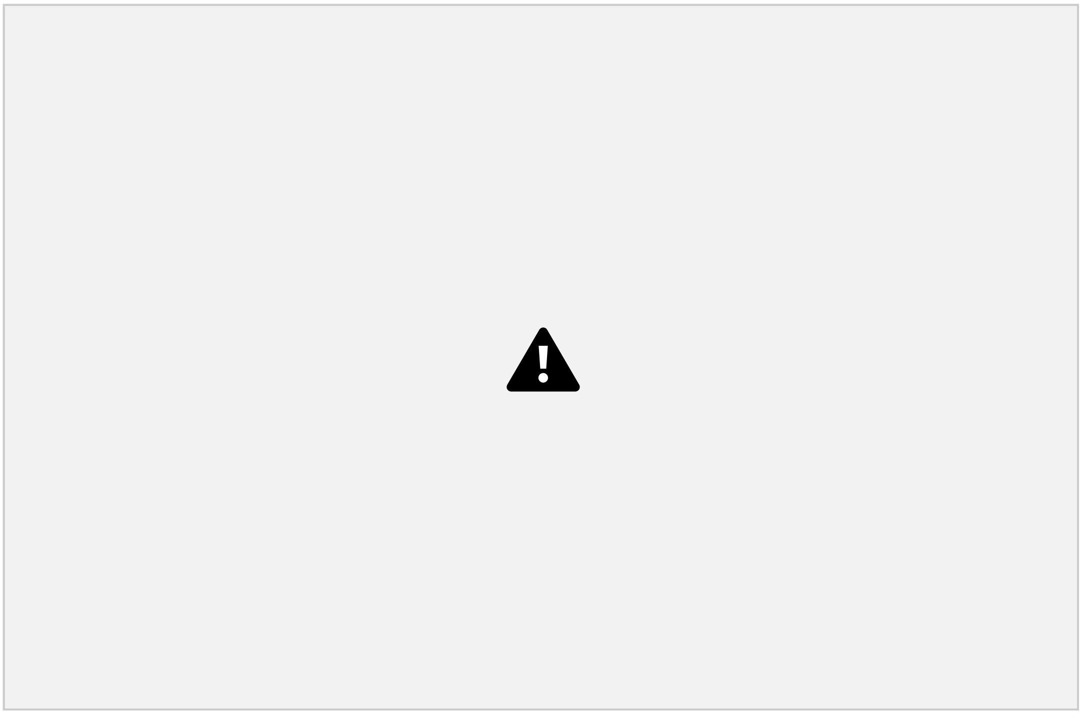
$$N = \lambda * W$$

- Queuing models treat the computer as a network of interconnected queues, each of which is described by its probability distribution statistics and formulas such as Little's formula.

Unfortunately real systems and modern scheduling algorithms are so complex as to make the mathematics intractable in many cases with real systems.

Simulations

- Another approach is to run computer simulations of the different proposed algorithms (and adjustment parameters) under different load conditions, and to analyze the results to determine the "best" choice of operation for a particular load pattern.
- Operating conditions for simulations are often randomly generated using distribution functions similar to those described above.
- A better alternative when possible is to generate ***trace tapes***, by monitoring and logging the performance of a real system under typical expected work loads. These are better because they provide a more accurate picture of system loads, and also because they allow multiple simulations to be run with the identical process load, and not just statistically equivalent loads. A compromise is to randomly determine system loads and then save the results into a file, so that all simulations can be run against identical randomly determined system loads.
- Although trace tapes provide more accurate input information, they can be difficult and expensive to collect and store, and their use increases the complexity of the simulations significantly. There is also some question as to whether the future performance of the new system will really match the past performance of the old system.


Evaluation of CPU schedulers by simulation

Implementation

- The only real way to determine how a proposed scheduling algorithm is going to operate is to implement it on a real system.
- For experimental algorithms and those under development, this can cause difficulties and resistance among users who don't care about developing OSes and are only trying to get their daily work done.

- Even in this case, the measured results may not be definitive, for at least two major reasons: (1) System work loads are not static, but change over time as new programs are installed, new users are added to the system, new hardware becomes available, new work projects get started, and even societal changes. (For example the explosion of the Internet has drastically changed the amount of network traffic that a system sees and the importance of handling it with rapid response times.) (2) As mentioned above, changing the scheduling system may have an impact on the work load and the ways in which users use the system

- Most modern systems provide some capability for the system administrator to adjust scheduling parameters, either on the fly or as the result of a reboot or a kernel rebuild.

Interprocess Communication

- **Independent Processes** operating concurrently on a systems are those that can neither affect other processes or be affected by other processes.
- **Cooperating Processes** are those that can affect or be affected by other processes. There are several reasons why cooperating processes are allowed:
  - Information Sharing - There may be several processes which need access to the same file for example. (e.g. pipelines.)
  - Computation speedup - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously (particularly when multiple processors are involved.)
  - Modularity - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)
  - Convenience - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.

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

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

- Shared Memory is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. However it is more complicated to set up, and doesn't work as well across multiple computers. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.

- Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved.

Shared-Memory Systems

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

Producer-Consumer Example Using Shared Memory

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

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

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

// Code from Producer

```
item nextProduced;

while( true ) {

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

/* Wait for space to become available */
```

```
while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
```

```
 ; /* Do nothing */

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

}
```

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

Code from Consumer

```
item nextConsumed;

while( true ) {

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

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

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

}
```

Message-Passing Systems

· Message passing systems must support at a minimum system calls for "send message" and "receive message".
· A communication link must be established between the cooperating processes before messages can be sent.
· There are three key issues to be resolved in message passing systems as further explored in the next three subsections:
  o Direct or indirect communication (naming)
  o Synchronous or asynchronous communication
  o Automatic or explicit buffering.

1. Naming

· With **direct communication** the sender must know the name of the receiver to which it wishes to send a message.
  o There is a one-to-one link between every sender-receiver pair.
  o For **symmetric** communication, the receiver must also know the specific name of the sender from which it wishes to receive messages. For **asymmetric** communications, this is not necessary.
· **Indirect communication** uses shared mailboxes, or ports.
  o Multiple processes can share the same mailbox or boxes.

- Only one process can read any given message in a mailbox. Initially the process that creates the mailbox is the owner, and is the only one allowed to read mail in the mailbox, although this privilege may be transferred.
    - (Of course the process that reads the message can immediately turn around and place an identical message back in the box for someone else to read, but that may put it at the back end of a queue of messages.)
- The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.

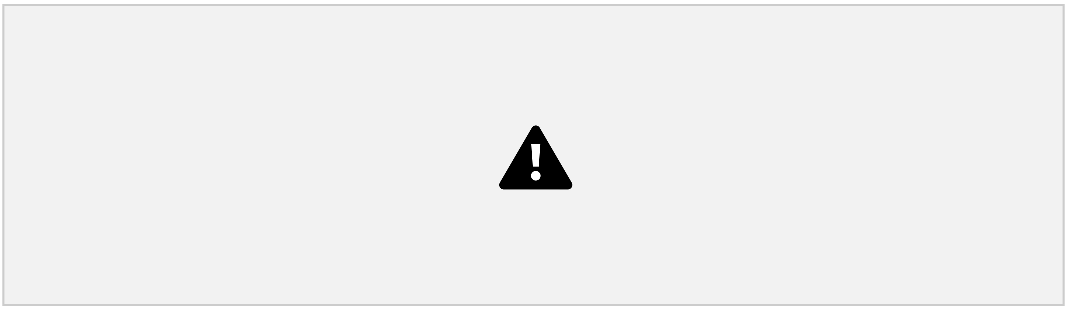
2. Synchronization

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

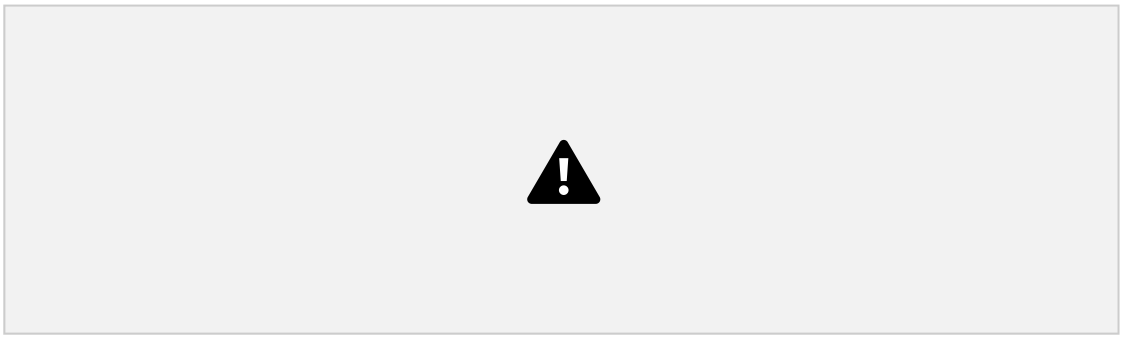
3. Buffering

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

1. **Zero capacity** - Messages cannot be stored in the queue, so senders must block until receivers accept the messages.
2. **Bounded capacity**- There is a certain pre-determined finite capacity in the queue. Senders must block if the queue is full, until space becomes available in the queue, but may be either blocking or non-blocking otherwise.
3. **Unbounded capacity** - The queue has a theoretical infinite capacity, so senders are never forced to block.



The Producer process using Message Passing


The Consumer process using Message Passing

Process Synchronization

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:

```
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:

```
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 ) */
 }
```

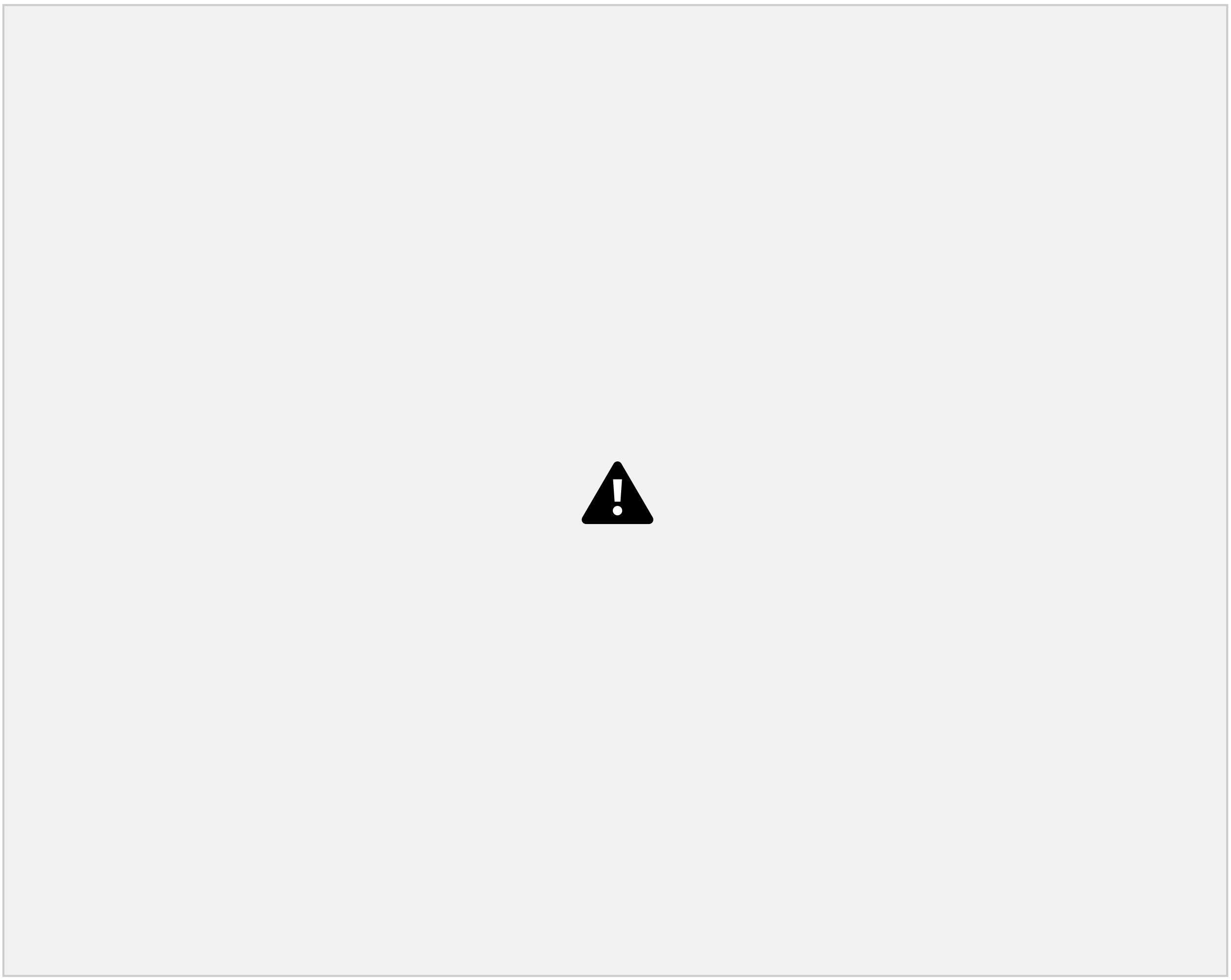
- The only problem with the above code is that the maximum number of items which can be placed into the buffer is BUFFER_SIZE - 1. One slot is unavailable because there always has to be a gap between the producer and the consumer.

- We could try to overcome this deficiency by introducing a counter variable, as shown in the following code segments:

- 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:



- **Exercise:** What would be the resulting value of counter if the order of statements T4 and T5 were reversed? (What **should** the value of counter be after one producer and one consumer, assuming the original value was 5?)
- Note that race conditions are ***notoriously difficult*** to identify and debug, because by their very nature they only occur on rare occasions, and only when the timing is just exactly right. (or wrong! :-)) Race conditions are also very difficult to reproduce. :-(
- Obviously the solution is to only allow one process at a time to manipulate the value "counter". This is a very common occurrence among cooperating processes, so lets look at

some ways in which this is done, as well as some classic problems in this area.

The Critical-Section Problem

Consider a system consisting of n processes {P_1,P_2,........,P_n}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The critical section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process P, is shown in Figure

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.

General structure of a typical process Pi

Solution Criteria for Critical Section problem

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

  3. **Bounded Waiting** - There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted. (I.e. a process requesting entry into their critical section will get a turn eventually, and there is a limit as to how many other processes get to go first.)
- We assume that all processes proceed at a non-zero speed, but no assumptions can be made regarding the ***relative*** speed of one process versus another.
- Kernel processes can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures such as open file tables or virtual memory management. Accordingly kernels can take on one of two forms:
  - Non-preemptive kernels do not allow processes to be interrupted while in kernel mode. This eliminates the possibility of kernel-mode race conditions, but requires kernel mode operations to complete very quickly, and can be problematic for real time systems, because timing cannot be guaranteed.
  - Preemptive kernels allow for real-time operations, but must be carefully written to

avoid race conditions. This can be especially tricky on SMP systems, in which multiple kernel processes may be running simultaneously on different processors.

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

Two-Process Solutions

In this section, we restrict our attention to algorithms that are applicable to only two processes at a time. The processes are numbered P_0 and P_1. For convenience, when presenting Pj, we use P_i, to denote the other process; that is, j = 1 - *i*.

Algorithm 1: (A Naive Solution)

Our first approach is to let the processes share a common integer variable *turn* initialized to 0 (or 1). If *turn* == *i,* then process P_i, is allowed to execute in its critical section. The structure of process P_i, is shown in algorithm

```
repeat
while turn != i do no-op;
 critical section
turn = j;
remainder section
until false;
```

The structure of process *Pi* in Algorithm1

This solution ensures that only one process at a time can be in its critical section. However, it does not satisfy the progress requirement, since it requires strict alternation of processes in the execution of the critical section". For example, if *turn*= = 0 and P_1 is ready to enter its critical section, P_1 cannot do so, even though P_0 may be in its remainder section.

Algorithm 2: (Dekker's Solution)

The problem with algorithm 1 is that it does not retain sufficient information about the state of each process; it remembers only which process is allowed to enter the critical section. To remedy this problem, we can replace the variable *turn* with the following array:

Boolean flag[2];

The elements of the array are initialized *to false.* If *flag[i]* is *true,* this value indicates that P_i, is *ready* to enter the critical section. The structure of process P_i is shown in algorithm below.

In this algorithm, process P_i first *sets flag[i]* to be *true,* signaling that it is ready to enter its critical section. Then, P_i checks to verify that process P_j is not also ready to enter its critical section. If P_j were ready, then P_i would wait until P_j had indicated that it no longer needed to be in the critical section (that is, until *flag[j] was false).* At this point, P_i would enter the critical section. On exiting the critical section, P_i would set *its flag* to be *false,* allowing the other process (if it is waiting) to enter its critical section.

```
repeat
flag[i] := true;
```

```
        while flag[j] do no-op;
         critical section
        flag[i] :=false;
         remainder section
        until false;
```

The structure of process *Pi* in Algorithm2

In this solution, the mutual-exclusion requirement is satisfied. Unfortunately, the progress requirement is not met. To illustrate this problem, we consider the following execution sequence:

T0: P0 *sets flag[0] = true*
TI: PI *sets flag[l] = true*

Now P_0 and P_1 are looping forever in their respective while statements. This algorithm is crucially dependent on the exact timing of the two processes. The sequence could have been derived in an environment where there are several processors executing concurrently, or where an interrupt (such as a timer interrupt) occurs immediately after step T_0 is executed, and the CPU is switched from one process to another.

Algorithm 3: (Peterson's Solution**)**

Next, we illustrate a classic software-based solution to the critical-section problem known as **Peterson's solution**. Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);
                critical section
        flag[i] = false;
                remainder section
} while (true);
```

The structure of process *Pi* in Peterson's solution.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered *P*0 and *P*1. For convenience, when presenting *Pi* , we use *Pj* to denote the other process; that is, j equals 1 – i.

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process *Pi* is allowed to execute in its critical section. For example, if flag[i] is true, this value indicates that *Pi* is ready to enter its critical section. To enter the critical section, process *Pi* first

sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately.
The eventual value of turn determines which of the two processes is allowed to enter its critical section first. We now prove that this solution is correct. We need to show that: • Mutual exclusion is preserved.

- The progress requirement is satisfied.
- The bounded-waiting requirement is met.

To prove property 1, we note that each *Pi* enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] ==flag[1] == true. These two observations imply that *P*0 and *P*1 could not have Successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes—say, *Pj*—must have successfully executed the while statement, whereas *Pi* had to execute at least one additional statement ("turn == j"). However, at that time, flag[j] == true and turn == j, and this condition will persist as long as *Pj* is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3,we note that a process *Pi* can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one possible. If *Pj* is not ready to enter the critical section, then flag[j] == false, and *Pi* can enter its critical section. If *Pj* has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then *Pi* will enter the critical section. If turn == j, then *Pj* will enter the critical section. However, once *Pj* exits its critical section, it will reset flag[j] to false, allowing *Pi* to enter its critical section. If *Pj* resets flag[j] to true, it must also set turn to i. Thus, since *Pi* does not change the value of the variable turn while executing the while statement, *Pi* will enter the critical section (progress) after at most one entry by *Pj* (bounded waiting).

Synchronization Hardware

Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.

In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.

Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.

This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.

Mutex Locks

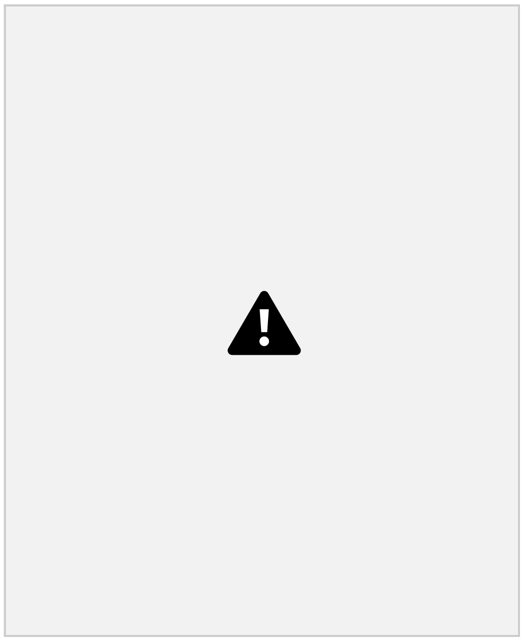
As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released.

As the resource is locked while a process executes its critical section hence no other process can access it.

Semaphores

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

- A more robust alternative to simple mutex 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.



The classical definition of wait and signal are :

· Wait : decrement the value of its argument S as soon as it would become non-negative. · Signal : increment the value of its argument, S as an individual operation.

Application of Semaphores

1. Solve the critical section problem
2. Decide the order of execution sequence of processes
3. Resource sharing if multiple instance of resource is present

Types of Semaphores

Semaphores are mainly of two types:

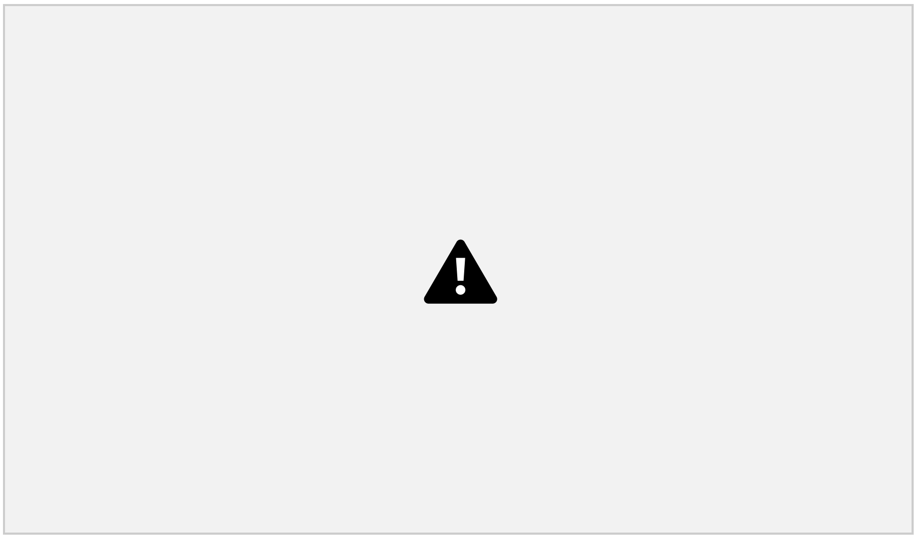
1. **Binary Semaphore**

   It is a special form of semaphore used for implementing mutual exclusion, hence it is often called *Mutex*. A binary semaphore is initialized to 1 and only takes the value 0 and 1 during execution of a program.

2. **Counting Semaphores**

   These are used to implement bounded concurrency.

- 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.



Mutual-exclusion implementation with semaphores.

  - **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 );
```

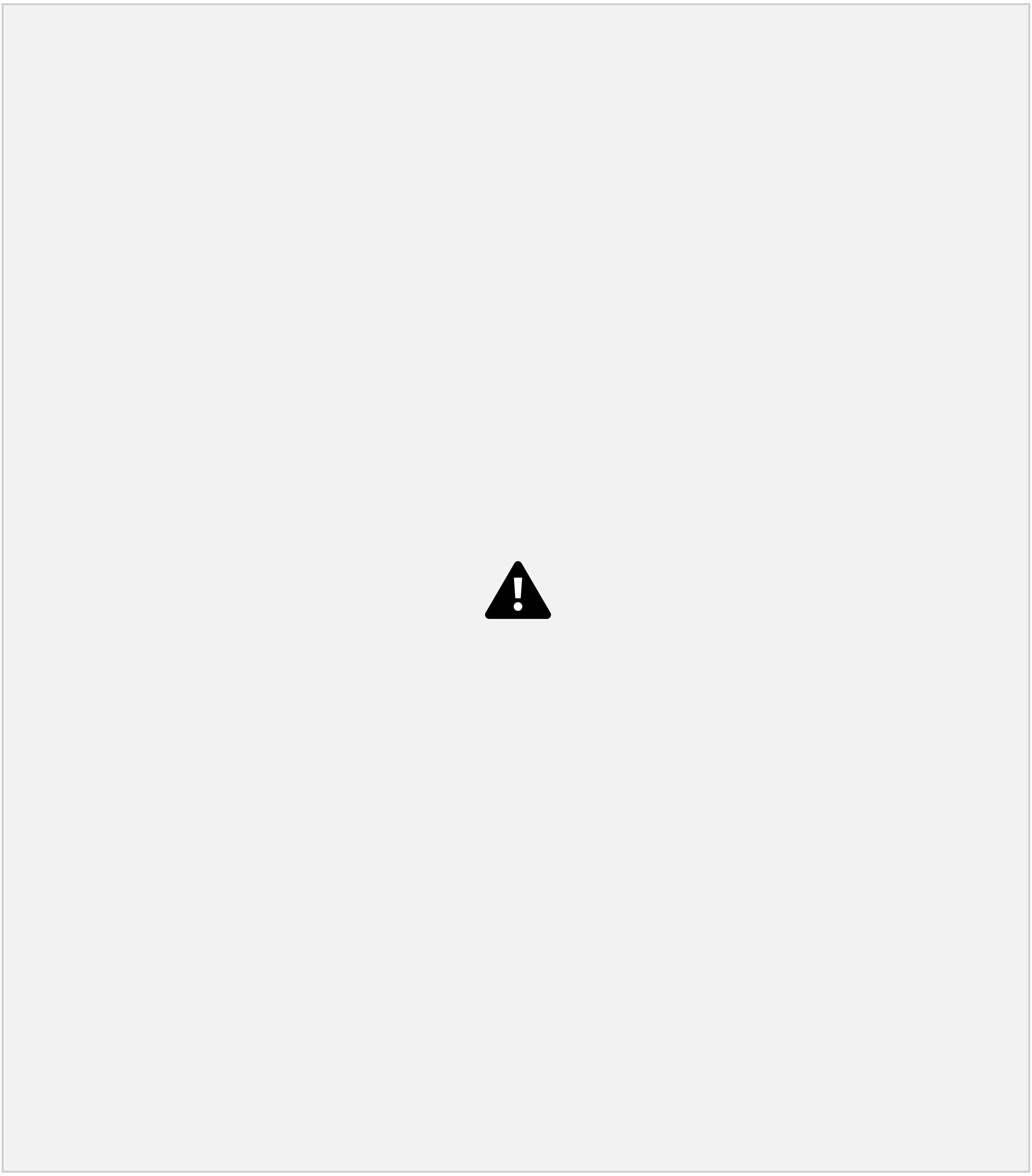
- and in process P2 we insert the code:

```
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

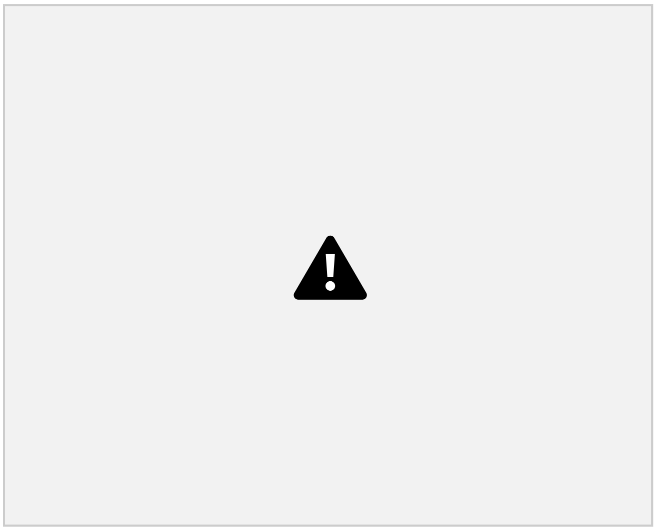
- The big problem with semaphores as described above is the busy loop in the wait call, which consumes CPU cycles without doing any useful work. This type of lock is known as a ***spinlock***, because the lock just sits there and spins while it waits. While this is generally a bad thing, it does have 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:



- 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.



- 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.

Limitations of Semaphores

1. Priority Inversion is a big limitation of semaphores.
2. Their use is not enforced, but is by convention only.
3. With improper use, a process may block indefinitely. Such a situation is called Deadlock.

Classical Problem of Synchronization

Following are some of the classical problem faced while process synchronaization in systems where cooperating processes are present.

1. Bounded Buffer Problem

- This problem is generalised in terms of the Producer-Consumer problem.
- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.

Problem Statement:

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

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

There needs to be a way to make the producer and consumer work in an independent manner.

Solution:

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

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

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

Producer Operation:

The pseudocode of the producer function looks like this:

```
do {
 wait(empty); // wait until empty>0 and then decrement 'empty'
 wait(mutex); // acquire lock
 /* perform the insert operation in a slot */
 signal(mutex); // release lock
 signal(full); // increment 'full'
 } while(TRUE)
```

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

Consumer Operation:

The pseudocode of the consumer function looks like this:

```
do {
 wait(full); // wait until full>0 and then decrement 'full'
 wait(mutex); // acquire the lock
 /* perform the remove operation
 in a slot */
 signal(mutex); // release the lock
 signal(empty); // increment 'empty'
 } while(TRUE);
```

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

2. The Readers Writers Problem

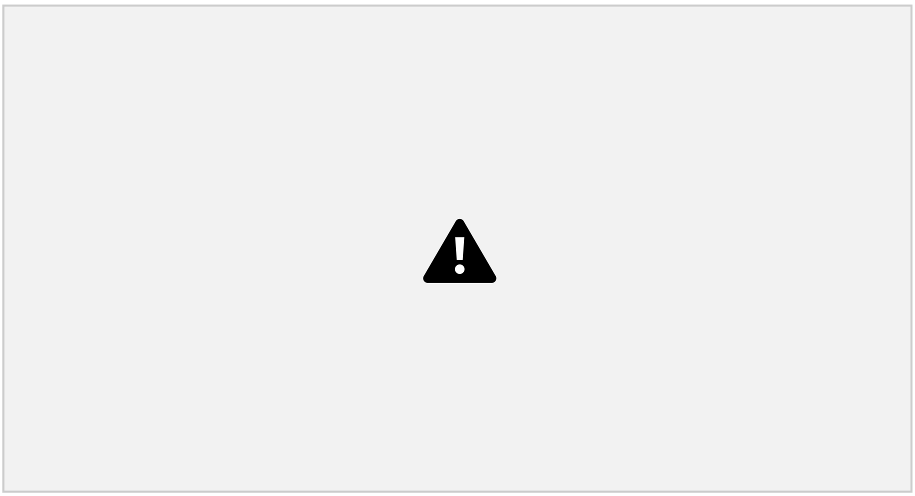
Readers writer problem is another example of a classic synchronization problem.

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

- There are various type of the readers-writers problem, most centred on relative priorities of readers and writers

Problem Statement:

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



Solution:

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one mutex **m** and a semaphore **w**. An integer variable **read_count** is used to maintain the number of readers currently accessing the resource. The variable **read_count** is initialized to 0. A value of 1 is given initially to **m** and **w**.

Instead of having the process to acquire lock on the shared resource, we use the mutex **m** to make the process to acquire and release lock whenever it is updating the **read_count** variable.

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

```
while(TRUE) {
wait(w);
/*perform the
write operation */
signal(w);
 }
```

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

```
while(TRUE) {
 wait(m); //acquire lock
```

```
read_count++;
if(read_count == 1)
wait(w);
signal(m); //release lock
/* perform the
reading operation */
wait(m); // acquire lock
read_count--;
if(read_count == 0)
signal(w);
signal(m); // release lock
}
```

Code Explained:

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

3. <u>**Dining Philosophers Problem**</u>

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

- The dining philosopher's problem involves the allocation of limited resources from a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks kept beside them and a bowl of rice in the centre, When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Problem Statement

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

Solution:

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

An array of five semaphores, **stick[5]**, for each of the five chopsticks.

The code for each philosopher looks like:

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

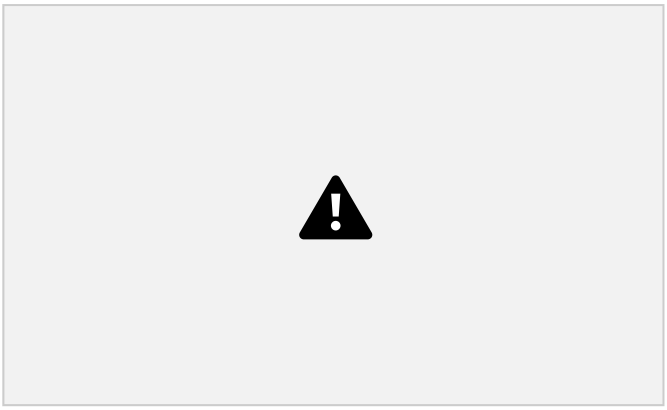
When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

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

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

4. The Sleeping Barber Problem

Another classical IPC problem takes place in a barber shop. The barber shop has one barber, one barber chair, and n chairs for waiting customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and falls asleep, as illustrated in [Figure 2.35]. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full). The problem is to program the barber and the customers without getting into race conditions.
This problem is similar to various queueing situations, such as a multiperson helpdesk with a computerized call waiting system for holding a limited number of incoming calls.



Our solution uses three semaphores, customers, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), barbers, the number of barbers (0 or 1) who are idle, waiting for customers, and mutex, which is used for mutual exclusion. We also need a variable, waiting, which also counts the waiting customers.
The reason for having waiting that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.

```
#define CHAIRS 5 /* # chairs for waiting customers */
```

```
typedef int semaphore; /* use your imagination */
semaphore customers = 0; /* # of customers waiting for service */
semaphore barbers = 0; /* # of barbers waiting for customers */
semaphore mutex = 1; /* for mutual exclusion */
int waiting = 0; /* customer are waiting (not being cut) */

void barber(void)
{
```

```
        while (TRUE) {
        down(&customers); /* go to sleep if # of customers is 0 */
        down(&mutex); /* acquire access to "waiting' */
                    waiting = waiting -1; /* decrement count of waiting customers */
                          up(&barbers); /* one barber is now read y to cut hair */
                up(&mutex); /* release 'waiting' */
                             cut_hair(); /* cut hair (outside critical region */
        }
}

void customer(void)
{
        down(&mutex); /* enter critical region */
         if (waiting < CHAIRS) { /* if there are no free chairs, leave */ waiting = waiting +
        1; /* increment count of waiting customers */ up(&customers); /* wake up barber if
                                        necessary */
                                 up(&mutex); /* release access to 'waiting'*/
                down(&barbers); /* go to sleep if # of free barbers is 0 */ get_haircut();
                /* be seated and be served */
        } else {
                                 up(&mutex); /* shop is full; do not wait */
        }
}
```