2.9. CPU SCHEDULING

2.9.1 Basic Concepts

Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.)

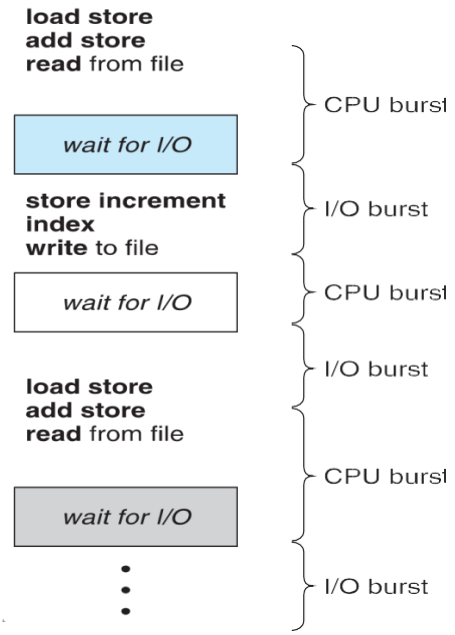
In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.

A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.

The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions, and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.

CPU-I/O Burst Cycle

- Almost all processes alternate between two states in a continuing **cycle**, as shown in Figure below **:**
- A CPU burst of performing calculations, and
- An I/O burst, waiting for data transfer in or out of the system.



CPU Scheduler

- Whenever the CPU becomes idle, it is the job of the CPU Scheduler (a.k.a. the short-term scheduler) to select another process from the ready queue to run next.
- The storage structure for the ready queue and the algorithm used to select the next process are not necessarily a FIFO queue. There are several alternatives to choose from, as well as numerous adjustable parameters for each algorithm, which is the basic subject of this entire chapter.

Preemptive Scheduling

CPU scheduling decisions take place under one of four conditions:

1. When a process switches from the running state to the waiting state, such as for an I/O request or invocation of the wait( ) system call.
2. When a process switches from the running state to the ready state, for example in response to an interrupt.
3. When a process switches from the waiting state to the ready state, say at completion of I/O or a return from wait( ).
4. When a process terminates.

- For conditions 1 and 4 there is no choice - A new process must be selected.
- For conditions 2 and 3 there is a choice - To either continue running the current process, or select a different one.
- If scheduling takes place only under conditions 1 and 4, the system is said to be **non-preemptive**, or **cooperative**. Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes. Otherwise the system is said to be **preemptive.**
- Windows used non-preemptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Macs used non-preemptive prior to OSX, and pre-emptive since then. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt.

Note that pre-emptive scheduling can cause problems when two processes share data, because one process may get interrupted in the middle of updating shared data structures. Chapter 5 examined this issue in greater detail.

- Preemption can also be a problem if the kernel is busy implementing a system call (e.g. updating critical kernel data structures) when the preemption occurs. Most modern

UNIX deal with this problem by making the process wait until the system call has either completed or blocked before allowing the preemption Unfortunately this solution is problematic for real-time systems, as real-time response can no longer be guaranteed.

- Some critical sections of code protect themselves from con currency problems by disabling interrupts before entering the critical section and re-enabling interrupts on exiting the section. Needless to say, this should only be done in rare situations, and only on very short pieces of code that will finish quickly, (usually just a few machine instructions.)

Dispatcher

The **dispatcher** is the module that gives control of the CPU to the process selected by the scheduler. This function involves:

- Switching context.
- Switching to user mode.
- Jumping to the proper location in the newly loaded program.
- The dispatcher needs to be as fast as possible, as it is run on every context switch. The time consumed by the dispatcher is known as **dispatch latency.**

2.9.2 Scheduling Criteria

There are several different criteria to consider when trying to select the "best" scheduling algorithm for a particular situation and environment, including:

CPU utilization

- Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

Throughput

- Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.

Turnaround time

- Time required for a particular process to complete, from submission time to completion. (Wall clock time.)

Waiting time

- How much time processes spend in the ready queue waiting their turn to get on the CPU.

(**Load average** - The average number of processes sitting in the ready queue waiting their turn to get into the CPU. Reported in 1-minute, 5-minute, and 15-minute averages by "uptime" and "who".)

Response time

- The time taken in an interactive program from the issuance of a command to the **commence** of a response to that command.

In general one wants to optimize the average value of a criteria (Maximize CPU utilization and throughput, and minimize all the others.) However sometimes one wants to do something different, such as to minimize the maximum response time.

Sometimes it is most desirable to minimize the **variance** of a criteria than the actual value.

I.e. users are more accepting of a consistent predictable system than an inconsistent one, even if it is a little bit slower.

2.9.3 Scheduling Algorithms

The following subsections will explain several common scheduling strategies, looking at only a single CPU burst each for a small number of processes. Obviously real systems have to deal with a lot more simultaneous processes executing their CPU-I/O burst cycles.

1. First-Come First-Serve Scheduling, FCFS

- FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine.
- Unfortunately, however, FCFS can yield some very long average wait times, particularly if the first process to get there takes a long time. For example, consider the following three processes:

Process	Burst Time
P1	24
P2	3
P3	3

- In the Gantt chart below, process P1 arrives first. The average waiting time for the three processes is (0 + 24 + 27) / 3 = 17.0 ms.

In the Gantt chart below, the same three processes have an average wait time of (0 + 3 + 6) / 3 = 3.0 ms. The total run time for the three bursts is the same, but in the second case two of the three finish much quicker, and the other process is only delayed by a short amount.

- FCFS can also block the system in a busy dynamic system in another way, known as the **convoy effect**. When one CPU intensive process blocks the CPU, a number of I/O intensive processes can get backed up behind it, leaving the I/O devices idle. When the CPU hog finally relinquishes the CPU, then the I/O processes pass through the CPU quickly, leaving the CPU idle while everyone queues up for I/O, and then the cycle repeats itself when the CPU intensive process gets back to the ready queue.
- Calculate Waiting time, average waiting time, turn around time, average turn around time

2. Shortest-Job-First Scheduling, SJF

- The idea behind the SJF algorithm is to pick the quickest fastest little job that needs to be done, get it out of the way first, and then pick the next smallest fastest job to do next.

(Technically this algorithm picks a process based on the next shortest **CPU burst**, not the overall process time.)

- For example, the Gantt chart below is based upon the following CPU burst times, (and the assumption that all jobs arrive at the same time.)

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

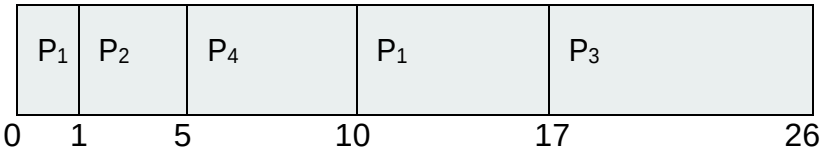
- In the case above the average wait time is (0 + 3 + 9 + 16) / 4 = 7.0 ms, (as opposed to 10.25 ms for FCFS for the same processes.)
- SJF can be proven to be the fastest scheduling algorithm, but it suffers from one important problem: How do you know how long the next CPU burst is going to be?
- For long-term batch jobs this can be done based upon the limits that users set for their jobs when they submit them, which encourages them to set low limits, but risks their having to re-submit the job if they set the limit too low. However that does not work for short-term CPU scheduling on an interactive system.

SJF can be either preemptive or non-preemptive. Preemption occurs when a new process arrives in the ready queue that has a predicted burst time shorter than the time remaining in the process whose burst is currently on the CPU. Preemptive SJF is sometimes referred to as

3. Shortest remaining time first scheduling.

- For example, the following Gantt chart is based upon the following data:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
p4	3	5



The average wait time in this case is ((5 - 3) + (10 - 1) + (17 - 2)) / 4 = 26 / 4 = 6.5 ms. (As opposed to 7.75 ms for non-preemptive SJF or 8.75 for FCFS.)

Calculate Waiting time, average waiting time, turn around time, average turn around time

3. Priority Scheduling

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

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

0 1 6 16 18 19

- 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 be either preemptive or non-preemptive.

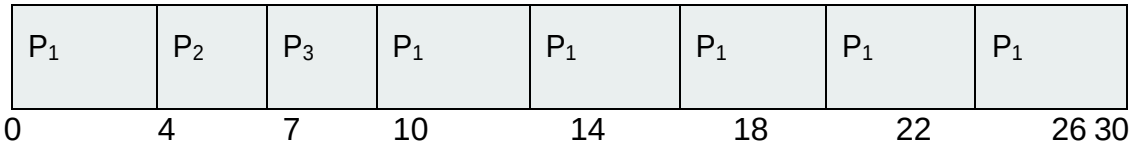
- Priority scheduling can suffer from a major problem known as **indefinite blocking**, or **starvation**, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.
- If this problem is allowed to occur, then processes will either run eventually 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.
- Calculate Waiting time, average waiting time, turn around time, average turn around time

4. Round Robin Scheduling

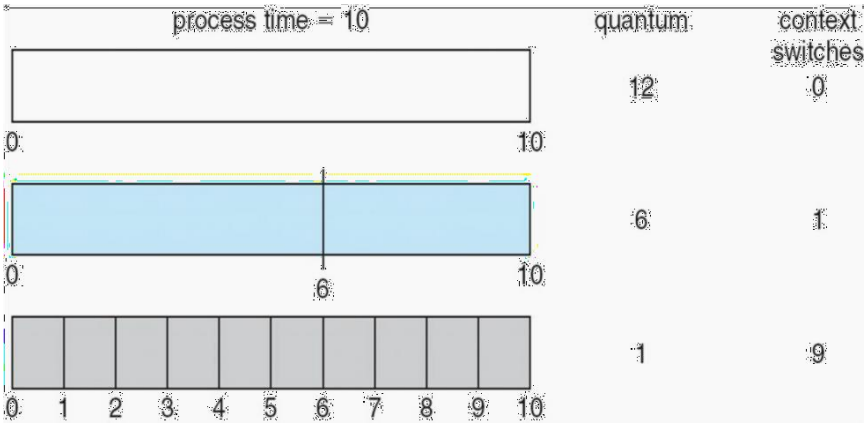
- Round robin scheduling is similar to FCFS scheduling, except that CPU bursts are assigned with limits called **time quantum**.
- When a process is given the CPU, a timer is set for whatever value has been set for a time quantum.
- If the process finishes its burst before the time quantum timer expires,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

Process	Burst Time
P1	24
P2	3
P3	3



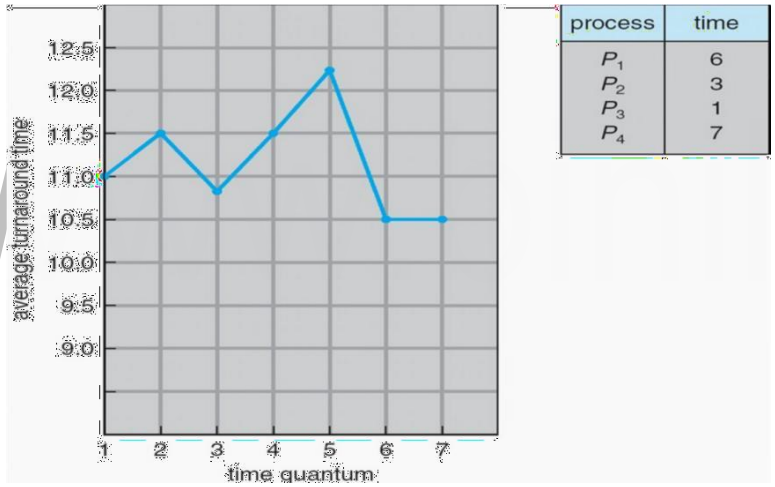
- The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally.

- **BUT**, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches there are. Most modern systems use time quantum between 10 and 100 milliseconds, and context switch times on the order of 10 microseconds, so the overhead is small relative to the time quantum.



The way in which a smaller time quantum increases context switches.

• Turnaround time also varies with quantum time, in a non-apparent manner. Consider, for example the processes shown in Figure 6.5:



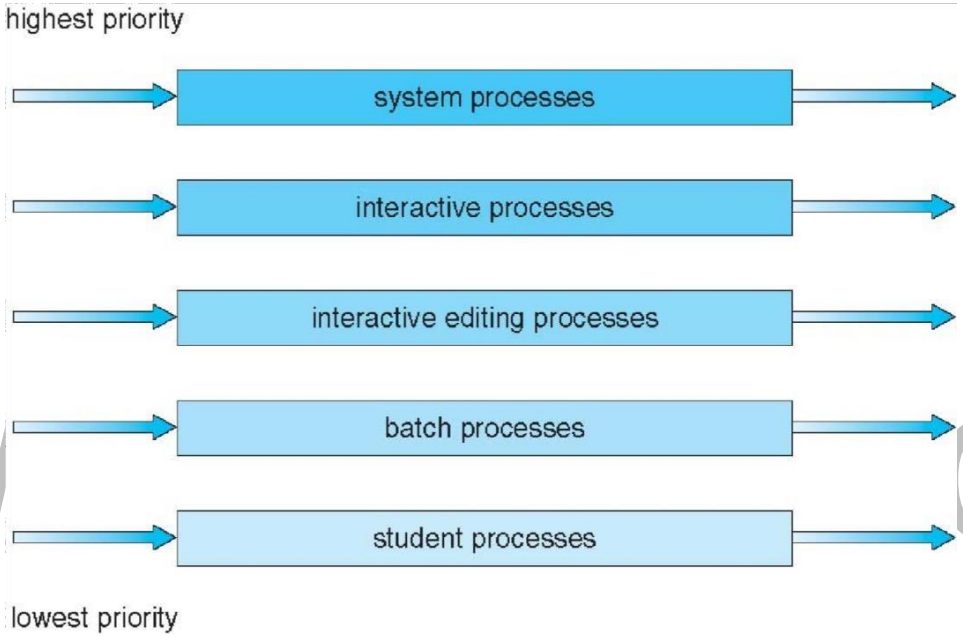
The way in which turnaround time varies with the time quantum.

In general, turnaround time is minimized if most processes finish their next cpu burst within one time quantum. For example, with three processes of 10 ms bursts each, the average turnaround time for 1 ms quantum is 29, and for 10 ms quantum it reduces to 20. However, if it is made too large, then RR just degenerates to FCFS. A rule of thumb is that 80% of CPU bursts should be smaller than the time quantum.

Calculate Waiting time, average waiting time, turn around time, average turn around time

5. Multilevel Queue Scheduling

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



6. Multilevel Feedback-Queue Scheduling

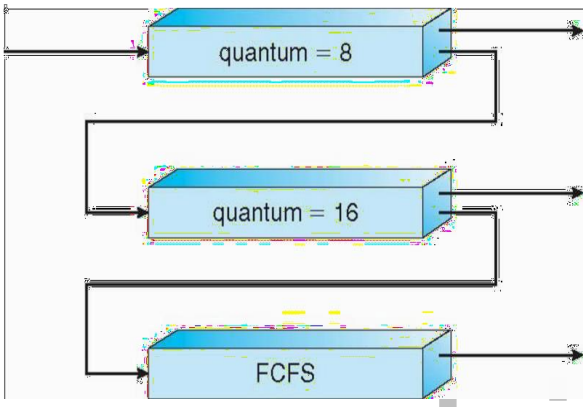
Multilevel feedback queue scheduling is similar to the ordinary multilevel queue scheduling described above, except jobs may be moved from one queue to another for a variety of reasons:

If the characteristics of a job change between CPU-intensive and I/O intensive, then it may be appropriate to switch a job from one queue to another.

Aging can also be incorporated, so that a job that has waited for a long time can get bumped up into a higher priority queue for a while.

Multilevel feedback queue scheduling is the most flexible, because it can be tuned for any situation. But it is also the most complex to implement because of all the adjustable parameters. Some of the parameters which define one of these systems include:

1. The number of queues.
2. The scheduling algorithm for each queue.
3. The methods used to upgrade or demote processes from one queue to another. (Which may be different.)
4. The method used to determine which queue a process enters initially.



2.10 DEADLOCKS

2.10.1 System Model

For the purposes of deadlock discussion, a system can be modelled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.

- Resource categories may include memory, printers, CPUs, open files, tape drives, CDROMS, etc.
- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- Some categories may have a single resource.
- In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:
- **Request** - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For example the system calls open( ), malloc( ), new( ), and request( ).
- **Use** - The process uses the resource, e.g. prints to the printer or reads from the file.
- **Release** - The process relinquishes the resource. so that it becomes available for other processes. For example, close( ), free( ), delete( ), and release( ).
- For all kernel-managed resources, the kernel keeps track of what resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available. Application-managed resources can be controlled using mutexes or wait( ) and signal( ) calls, (i.e. binary or counting semaphores.)
- A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)

2.10.2 Deadlock Characterization

Necessary Conditions

There are four conditions that are necessary to achieve deadlock:

1. Mutual Exclusion - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.

2. Hold and Wait - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.

3. No preemption - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.

4. Circular Wait - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)].

Resource-Allocation Graph

In some cases deadlocks can be understood more clearly through the use of **Resource-Allocation Graphs**, having the following properties:

A set of resource categories, { R1, R2, R3, . . ., RN }, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. two dots might represent two laser printers.)

A set of processes, { P1, P2, P3, . . ., PN }

Request Edges

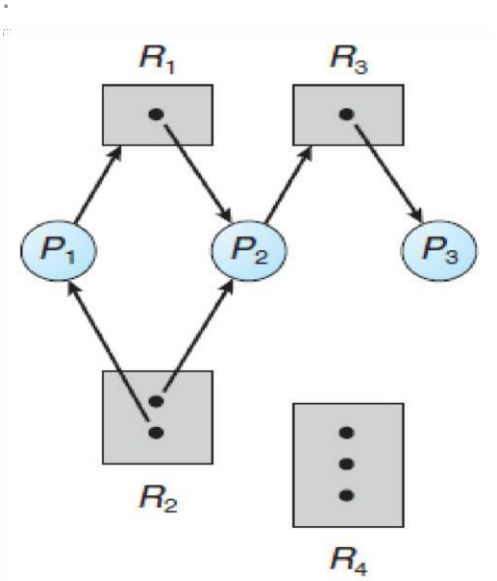
- A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.

Assignment Edges

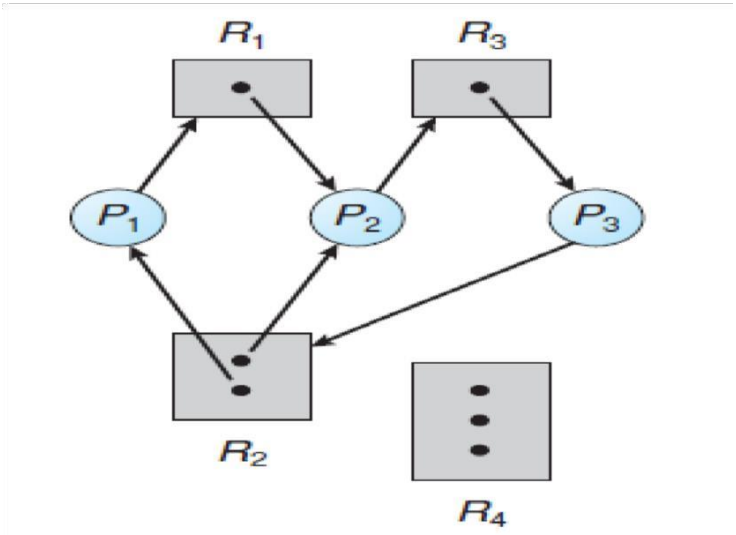
- A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.

Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.)

For example:

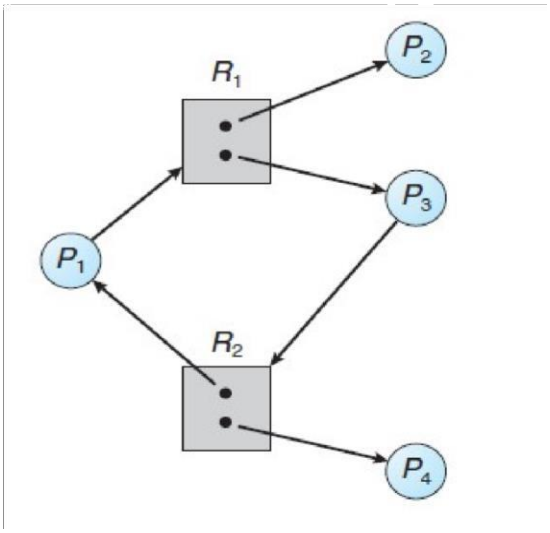


- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are **directed** graphs.) See the example in Figure above.
- If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, then a deadlock exists.
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock, but does not guarantee one. Consider, for example, Figures below:



Resource allocation graph with a deadlock

Resource allocation graph with a cycle but no deadlock



2.10.3 Methods for Handling Deadlocks

Generally there are three ways of handling deadlocks:

1. Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.

2. Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.

3. Ignore the problem all together – If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.

- In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future. (Ranging from a simple worst-case maximum to a complete resource request and release plan for each process, depending on the particular algorithm.)
- Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources, neither of which is an attractive alternative.
- If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes. Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

2.10.4 Deadlock Prevention

Deadlocks can be prevented by preventing at least one of the four required conditions:

Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
- Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
- Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
- Either of the methods described above can lead to starvation if a process requires one or more popular resources.

No Preemption

- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
- One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
- Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.

- Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In other words, in order to request resource Rj, a process must first release all Ri such that i>= j.
- One big challenge in this scheme is determining the relative ordering of the different resources

2.10.5 Deadlock Avoidance

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.
- This requires more information about each process, AND tends to lead to low device utilization. (I.e. it is a conservative approach.)
- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation **state** is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

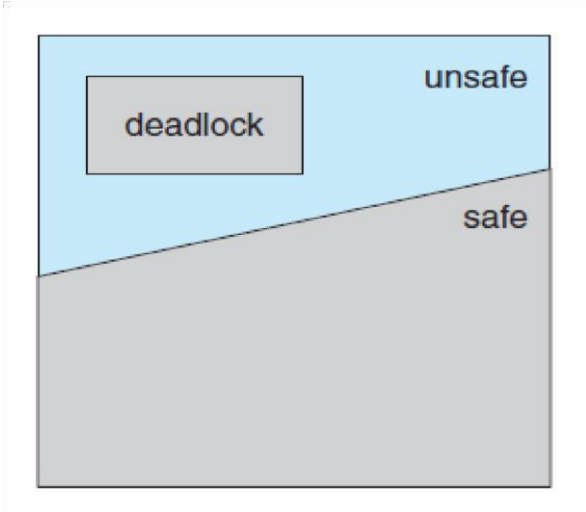
Safe State

A state is **safe** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.

More formally, a state is safe if there exists a **safe sequence** of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the

resources currently allocated to Pi and all processes Pj where j <i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)

• If a safe sequence does not exist, then the system is in an unsafe state, which **may** lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)



Safe, unsafe, and deadlocked state spaces.

For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

•

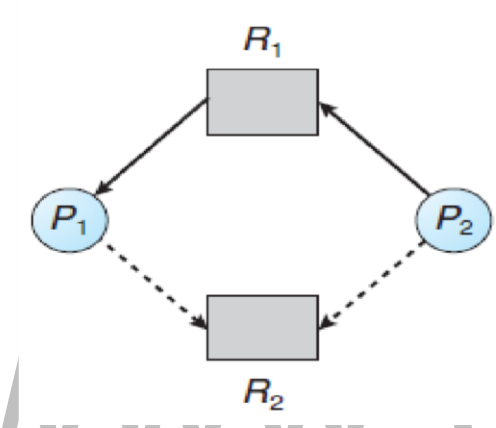
• What happens to the above table if process P2 requests and is granted one more tape drive?

• Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

Resource-Allocation Graph Algorithm

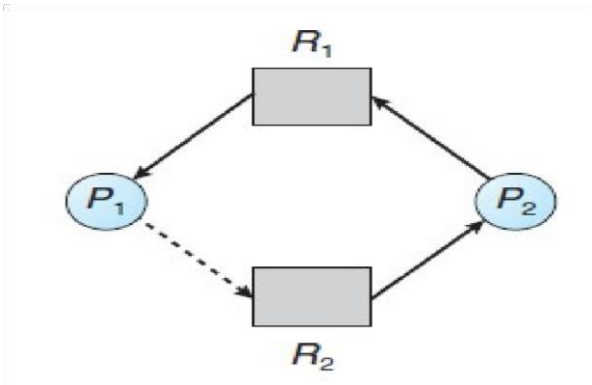
If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.

- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with **claim edges**, noted by dashed lines, which point from a process to a resource that it may request in the future.
- In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)
- When a process makes a request, the claim edge Pi->Rj is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge.
- This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.
- Consider for example what happens when process P2 requests resource R2:



Resource allocation graph for deadlock avoidance

The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.



An unsafe state in a resource allocation graph

Banker's Algorithm

For resource categories that contain more than one instance the resourceallocation graph method does not work, and more complex (and less efficient) methods must be chosen.

- The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)
- When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.
- When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.
- The banker's algorithm relies on several key data structures: (where n is the number of processes and m is the number of resource categories.)

Available[m] indicates how many resources are currently available of each type.

Max[n][m] indicates the maximum demand of each process of each resource.

Allocation[n][m] indicates the number of each resource category allocated to each process.

Need[n][m] indicates the remaining resources needed of each type for each process. (Note that **Need[i][j] = Max[i][j] - Allocation[i][j] for all i, j.**)

Safety Algorithm

In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.

This algorithm determines if the current state of a system is safe, according to the following steps:

1. Let **Work** and **Finish** be vectors of length m and n respectively.
2. Work is a working copy of the available resources, which will be modified during the analysis.

3. Finish is a vector of boolean indicating whether a particular process can finish. (or has finished so far in the analysis.)

4. Initialize Work to Available, and Finish to false for all elements.

5. Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.

6. Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.

7. If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.

Resource-Request Algorithm (The Bankers Algorithm)

Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself.

This algorithm determines if a new request is safe, and grants it only if it is safe to do so.

When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:

1. Let Request[n][m] indicate the number of resources of each type currently requested by processes. If Request[i] > Need[i] for any process i, raise an error condition.

3. If Request[i] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.

Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely.The procedure for granting a request (or pretending to for testing purposes) is:

- Available = Available - Request
- Allocation = Allocation + Request
- Need = Need - Request

2.10.6 Deadlock Detection

- If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.

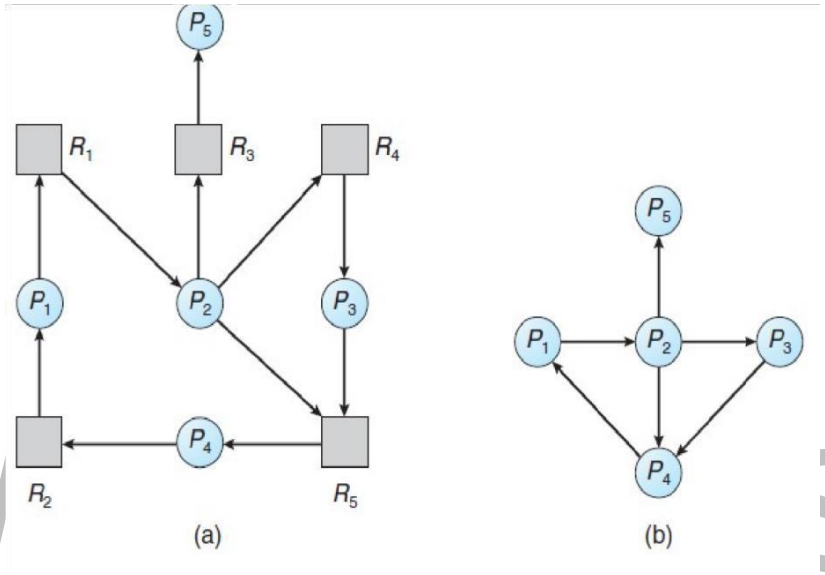
- In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.

Single Instance of Each Resource Type

If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a **wait-for graph**.

A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.

An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.



(a) Resource allocation graph. **(b) Corresponding wait-for graph**

As before, cycles in the wait-for graph indicate deadlocks.

This algorithm must maintain the wait-for graph, and periodically search it for cycles.

Several Instances of a Resource Type

The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:

In step 1, the Banker's Algorithm sets Finish[i] to false for all i. The algorithm presented here sets Finish[i] to false only if Allocation[i] is not zero. If the currently allocated resources for this process are zero, the algorithm sets Finish[i] to true. This is essentially assuming that IF all of the other processes can finish, then this process can finish

also. Furthermore, this algorithm is specifically looking for which processes are involved in a deadlock situation, and a process that does not have any resources allocated cannot be involved in a deadlock, and so can be removed from any further consideration.

Steps 2 and 3 are unchanged

In step 4, the basic Banker's Algorithm says that if Finish[i] == true for all i, that there is no deadlock. This algorithm is more specific, by stating that if Finish[i] == false for any process Pi, then that process is specifically involved in the deadlock which has been detected.

2.10.7 Recovery From Deadlock

- There are three basic approaches to recovery from deadlock:
- Inform the system operator, and allow him/her to take manual intervention. • Terminate one or more processes involved in the deadlock
- Preempt resources.

Process Termination

Two basic approaches, both of which recover resources allocated to terminated processes:

- Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
- Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.

In the latter case there are many factors that can go into deciding which processes to terminate next:

- Process priorities.
- How long the process has been running, and how close it is to finishing.
- How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
- How many more resources does the process need to complete. • How many processes will need to be terminated
- Whether the process is interactive or batch.
- (Whether or not the process has made non-restorable changes to any resource.)

Resource Preemption

When preempting resources to relieve deadlock, there are three important issues to be addressed:

1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)

Starvation - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get pre-empt.

2.4 INTERPROCESS COMMUNICATION

- A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.
- A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.
- Advantages of cooperating process

  i) Information sharing

  ii) Computation speedup

  iii) Modularity

  iv) Convenience**.**

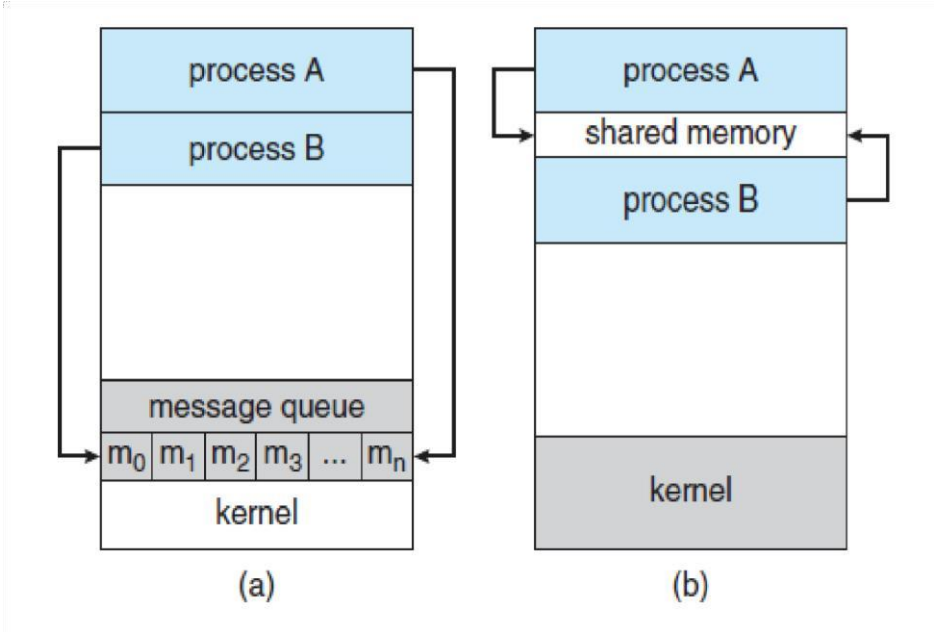
DEFINITION:

An Inter process communication is a mechanism that allows the cooperating process to exchange data and communication among each other.

There are two fundamental models of Inter process communication

- **Shared Memory model**
- **Message passing model**

- In shared memory model a region of memory is shared by the cooperating process. Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.
- Message passing is easier to implement in a distributed system than shared memory.
- The shared memory is faster than that of message passing.



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

2.4.1 Shared-Memory Systems

- Inter process communication using shared memory requires communicating processes to establish a region of shared memory.
- Shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space. They can then exchange information by reading and writing data in the shared areas.

EXAMPLE: PRODUCER – CONSUMER PROCESS:

- A **producer** process produces information that is consumed by a **consumer** process.
- One solution to the producer–consumer problem uses shared memory
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item.
- Two types of buffers can be used.

- Bounded Buffer.
- Unbounded Buffer.
- The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
- The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

CODE FOR PRODUCER PROCESS:

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


The producer process has local variable next produced in which the new item to be produced is stored.

The consumer process has a local variable next consumed in which the item to be consumed is stored.

This scheme allows at most BUFFER SIZE − 1 items in the buffer at the same time.

CODE FOR CONSUMER PROCESS

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

2.4.2 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
- Direct or indirect communication (naming)
- Synchronous or asynchronous communication
- Automatic or explicit buffering.

2.4.2.1 Naming

Each process that wants to communicate must explicitly name the recipient or sender of the communication. Direct communication can be done in two ways symmetric addressing and asymmetric addressing.

- With **direct communication** the sender must know the name of the receiver to which it wishes to send a message.
- There is a one-to-one link between every sender-receiver pair.
- For **symmetric** communication, the receiver must also know the specific name of the sender from which it wishes to receive messages.

  send(P, message)—Send a message to process P.
  receive(Q, message)—Receive a message from process Q.
- For **asymmetric** communications, this is not necessary.

  send(P, message)—Send a message to process P.
  receive(id, message)—Receive a message from any process
- **Indirect communication** uses shared mailboxes, or ports.

  send(A, message)—Send a message to mailbox A.
  receive(A, message)—Receive a message from mailbox A.
- 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.

The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.

3.4.2.2 Synchronization

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

- Blocking send. The sending process is blocked until the message is received by the receiving process or by the mailbox
- Nonblocking send. The sending process sends the message and resumes Operation
- Blocking receive. The receiver blocks until a message is available.
- Nonblocking receive. The receiver retrieves either a valid message or a null.

3.4.2.3 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.

2.9 MONITORS

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

{

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

Initialization code (…) { … }

}

}

2.9.1 Monitor Usage

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

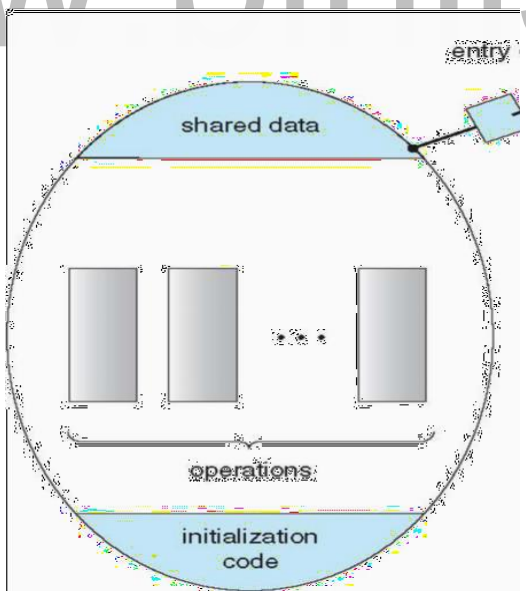
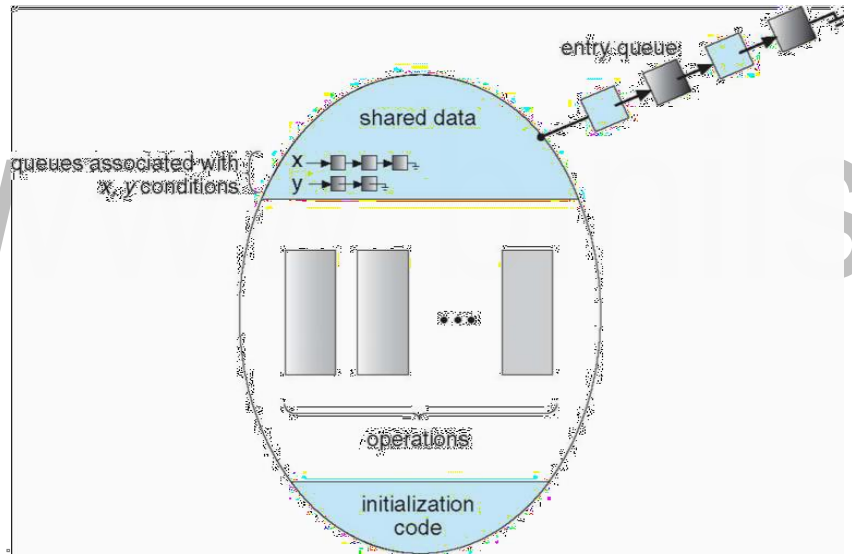


Figure shows a schematic of a monitor, with an entry queue of processes waiting their turn to execute monitor operations (methods.)

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



Monitor with condition variables

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

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

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

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

2.9.2 Dining-Philosophers Solution Using Monitors

- This solution to the dining philosophers uses monitors, and the restriction that a philosopher may only pick up chopsticks when both are available. There are also two key data structures in use in this solution:

1. enum { THINKING, HUNGRY,EATING } state[5]; A philosopher may only set their state to eating when neither of their adjacent neighbors is eating. (state[(i + 1) % 5] != EATING && state[(i + 4) % 5] != EATING).
2. Conditionself[5]; This condition is used to delay a hungry philosopher who is unable to acquire chopsticks.

In the following solution philosophers share a monitor, DiningPhilosophers, and eat using the following sequence of operations:

3. DiningPhilosophers.pickup( ) - Acquires chopsticks, which may block the process.
4. eat
5. DiningPhilosophers.putdown( ) - Releases the chopsticks.

```
monitorDiningPhilosophers
{
enum { THINKING; HUNGRY, EATING) state [5] ;
condition self [5];
void pickup (inti) {      state[i] = HUNGRY;
test(i);
if (state[i] != EATING) self[i].wait; }
void putdown (inti)
{      state[i] = THINKING;
```

```
            //test left and right neighborstest((i+4)%5);
test((i + 1) % 5);
}
void test (inti) {      if((state[(i+4)%5]!=EATING)&&      (state[i] == HUNGRY) &&
     (state[(i+1)%5] != EATING) )
{          state[i] = EATING ;
self[i].signal ();
     } }      initialization_code()
{     for (inti = 0; i< 5; i++)
state[i] = THINKING;
   }
}
```

2.9.3 Implementing a Monitor Using Semaphores

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

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

Variables **semaphoremutex;**

// (initially = 1) semaphore next;

// (initially = 0) intnext_count = 0;

Each procedure ***F*** will be replaced by **wait(mutex);**

…

body ofF;

… if (next_count> 0) signal(next) else signal(mutex);

Mutual exclusion within a monitor is ensured

For each condition variable ***x***, we have: **semaphore x_sem; // (initially = 0) intx_count = 0;**

The operation x.waitcan be implemented as:

x_count++; if (next_count> 0) signal(next); else

signal(mutex); wait(x_sem); x_count--;

The operation **x.signal**can be implemented as:

if (x_count> 0) { next_count++;

signal(x_sem); wait(next);

next_count--;

}

2.9.4 Resuming Processes Within a Monitor

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


```
monitorResourceAllocator
{boolean busy; condition x; void acquire(int time) {if (busy)
x.wait(time); busy= TRUE;
} void release() { busy = FALSE;
x.signal();
} initialization code() { busy = FALSE;
}
}
```

A monitor to allocate a single resource.

Unfortunately the use of monitors to restrict access to resources still only works if programmers make the requisite acquire and release calls properly. One option would be to

place the resource allocation code into the monitor, thereby eliminating the option for programmers to bypass or ignore the monitor, but then that would substitute the monitor's scheduling algorithms for whatever other scheduling algorithms may have been chosen for that particular resource.

2.3 OPERATIONS ON PROCESSES

The operating system must provide a mechanism for process creation and termination. The process can be created and deleted dynamically by the operating system. The Operations on the process includes

- Process creation
- Process Termination

2.3.1 Process Creation

During Execution a process may create several new processes.

- The creating process is called as the **parent process** and the newly created process is called as the **child process**.
  - Processes may create other processes through appropriate system calls, such as **fork** or **spawn**.
  - The operating systems identify the processes according to their unique process identifier.

:

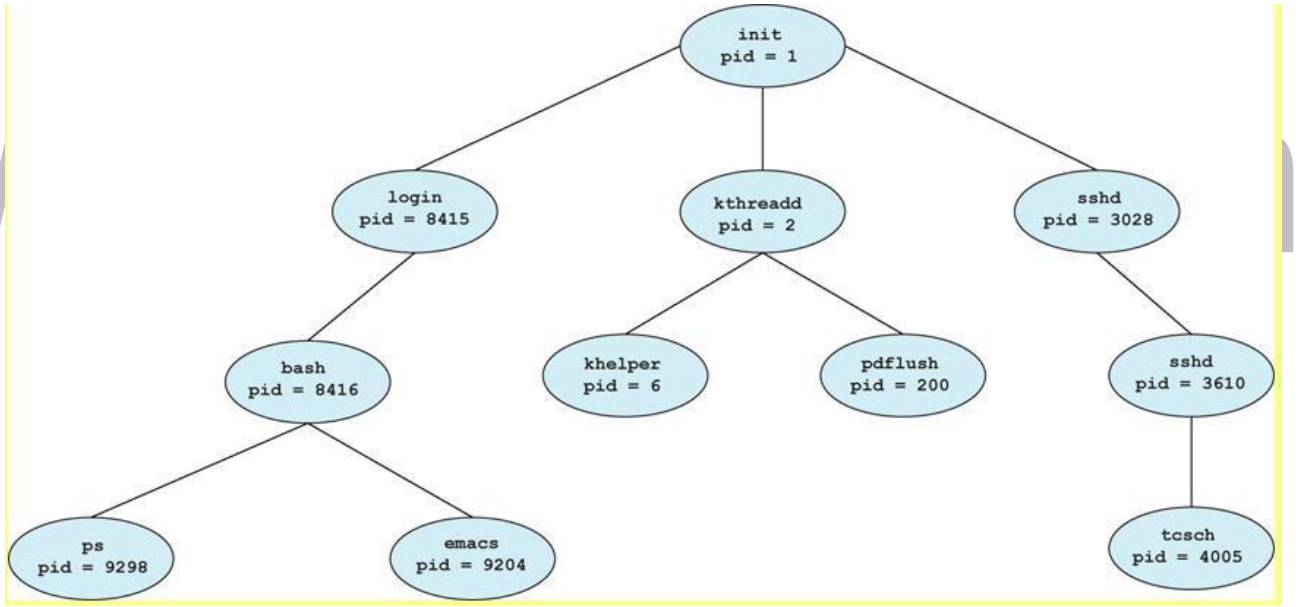


Fig: A tree of processes on a typical Linux system

- The init process serves as the root parent process for all the user process.
- Once the system has booted, the init process can also create various user processes, such as a web or printserver, an ssh server.
- The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel

- The sshd process is responsible for managing clients that connect to the system by using ssh(Secure shell)
- The login process is responsible for managing clients that directly log onto the system
- The command ps –el will list complete information for all processes currently active in the system.

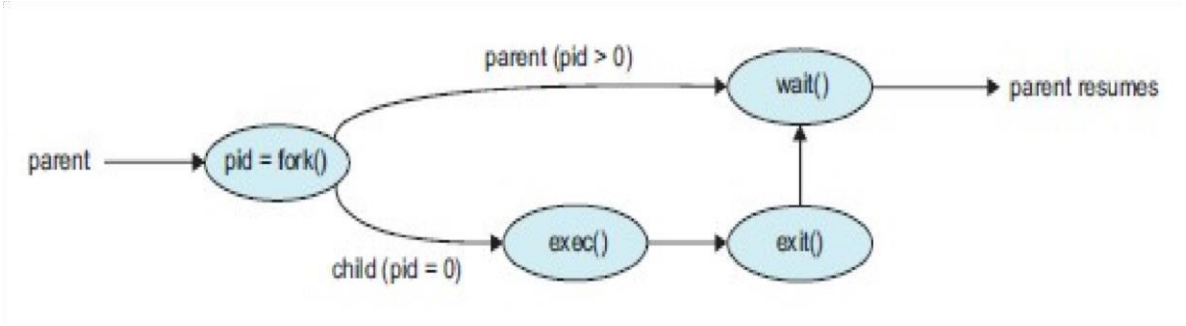
When a process creates a new process, two possibilities for execution exist:

- The parent continues to execute concurrently with its children.
- The parent waits until some or all of its children have terminated

There are also two address-space possibilities for the new process:

- The child process is a duplicate of the parent process (it has the same program as the parent).
- The child process has a new program loaded into it.
- The return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
    - After a fork() system call, one of the two processes typically uses the exec() system call to replace the process's memory space with a new program.
    - A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.

Depending on system implementation, a child process may receive some amount of shared resources with its parent. Child processes may or may not be limited to a subset of the resources originally allocated to the parent, preventing runaway children from consuming all of a certain system resource.




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

int main()
{
pid_t pid;

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

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

    return 0;
}
```

Creating a separate process using the UNIX fork( )system call.

Process creation using the fork( ) system call

2.3.2 Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.
- At that point, the process may return a status value (typically an integer) to its parent process.
- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system

- **A parent may terminate the execution of one of its children for a variety of reasons, such as**
- The child has exceeded its usage of some of the resources that it has been allocated.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.
- Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon is referred to as **cascading termination.**
- A parent process may wait for the termination of a child process by using the wait() system call
- This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid t pid;
int status;
pid = wait(&status);
```


  - A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie** process.

2.1 PROCESSES

2.1 Process Concept

- A process is an instance of a program in execution.
- Batch systems work in terms of "jobs". Many modern process concepts are still expressed in terms of jobs, (e.g. job scheduling), and the two terms are often used interchangeably.

2.1.1 The Process

Process memory is divided into four sections as shown in Figure below:

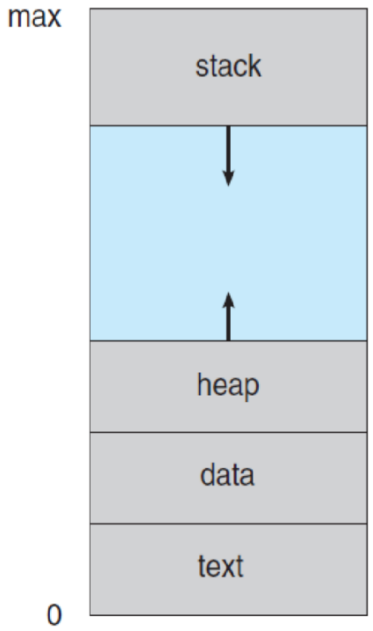


Figure 3.1 Process in memory.

- The text section comprises the compiled program code, read in from non-volatile storage when the program is launched.
- The data section stores global and static variables, allocated and initialized prior to executing main.
- The heap is used for 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 (at function entrance or elsewhere, depending on the language), and the space is freed up when the variables go out of scope. Note that the

stack is also used for function return values, and the exact mechanisms of stack management may be language specific.

- Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.
- When processes are swapped out of memory and later restored, additional information must also be stored and restored. Key among them are the program counter and the value of all program registers.

2.1.2 Process State

Processes may be in one of 5 states, as shown in Figure below.

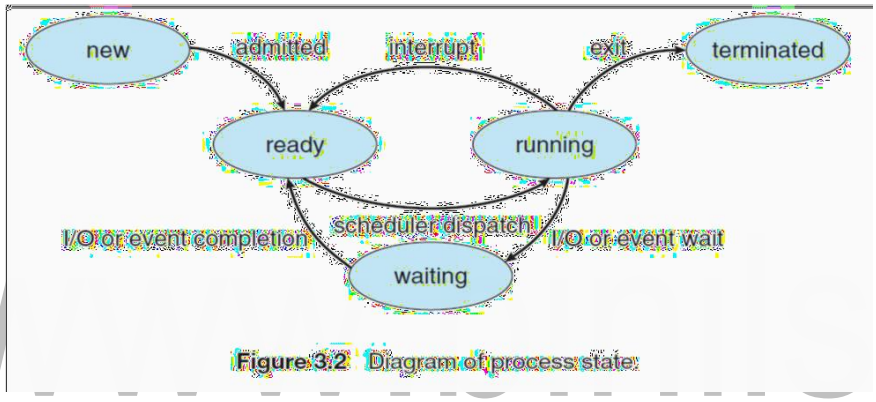


Figure 3.2 Diagram of process state

New - The process is in the stage of being created.

Ready - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.

Running - The CPU is working on this process's instructions.

Waiting - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.

Terminated - The process has completed.

The load average reported by the "w" command indicate the average number of processes in the "Ready" state over the last 1, 5, and 15 minutes, i.e. processes who have everything they need to run but cannot because the CPU is busy doing something else.

Some systems may have other states besides the ones listed here.

2.1.3 Process Control Block

For each process there is a Process Control Block, PCB, which stores the following (types of) process-specific information, as illustrated in Figure (Specific details may vary from system to system.)

Process State - Running, waiting, etc., as discussed above.

Process ID, and parent process ID.

CPU registers and Program Counter - These need to be saved and restored when swapping processes in and out of the CPU.

CPU-Scheduling information - Such as priority information and pointers to scheduling queues.

Memory-Management information - E.g. page tables or segment tables.

Accounting information - user and kernel CPU time consumed, account numbers, limits, etc.

I/O Status information - Devices allocated, open file tables, etc.

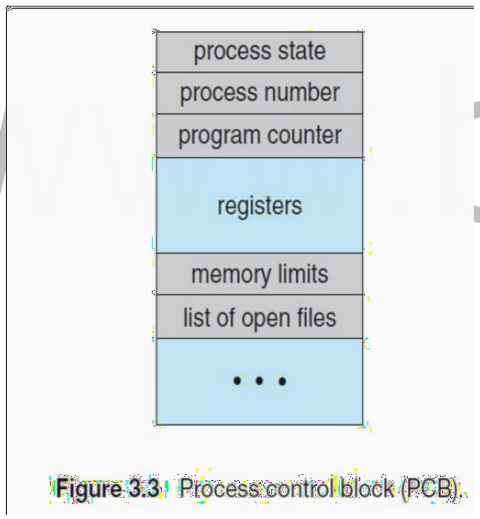


Figure 3.3 Process control block (PCB).

2.1.4 Threads

Modern systems allow a single process to have multiple threads of execution, which execute concurrently.

2.2 Process Scheduling

The two main objectives of the process scheduling system are to keep the CPU busy at all times and to deliver "acceptable" response times for all programs, particularly for interactive ones.

The process scheduler must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU.

2.2.1 Scheduling Queues

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

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

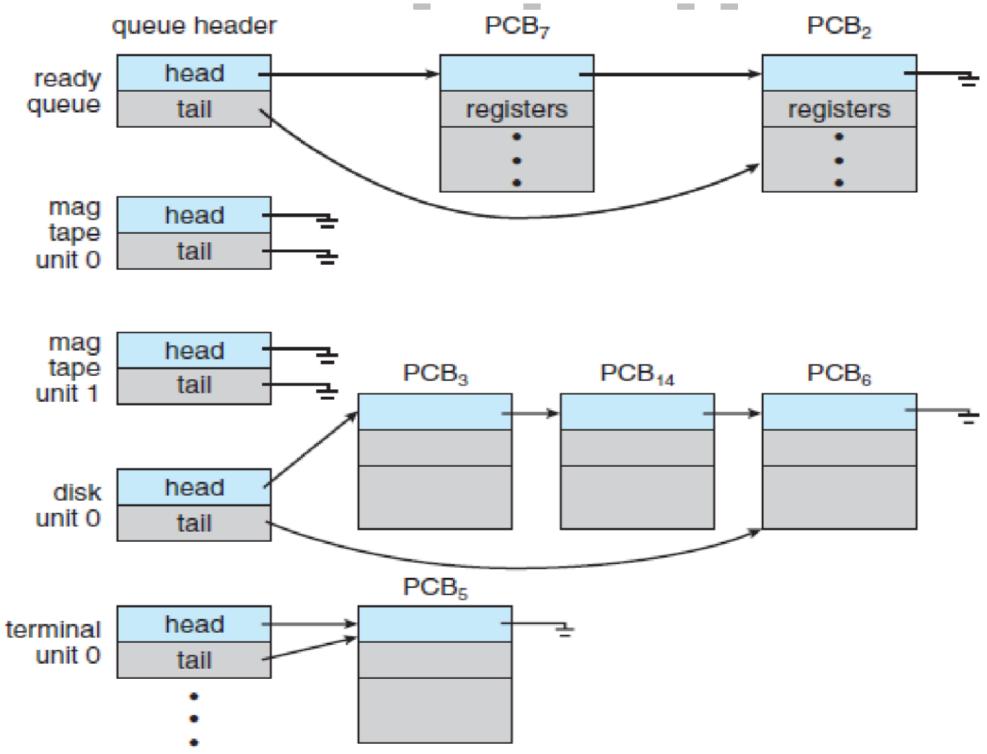


Figure 3.5 The ready queue and various I/O device queues.

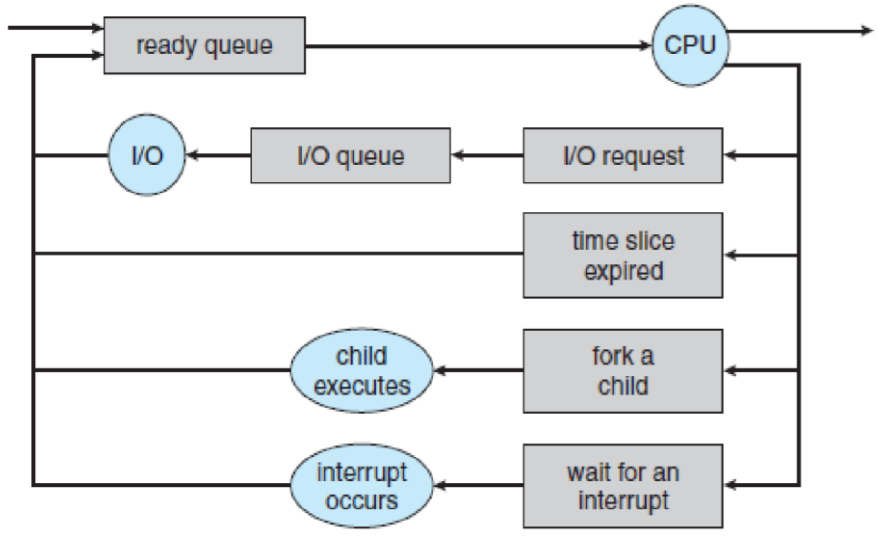
2.2.2 Schedulers

• A **long-term scheduler** is typical of a batch system or a very heavily loaded system. It runs infrequently, It is also called a job scheduler. It selects processes from the queue and loads them into memory for execution.

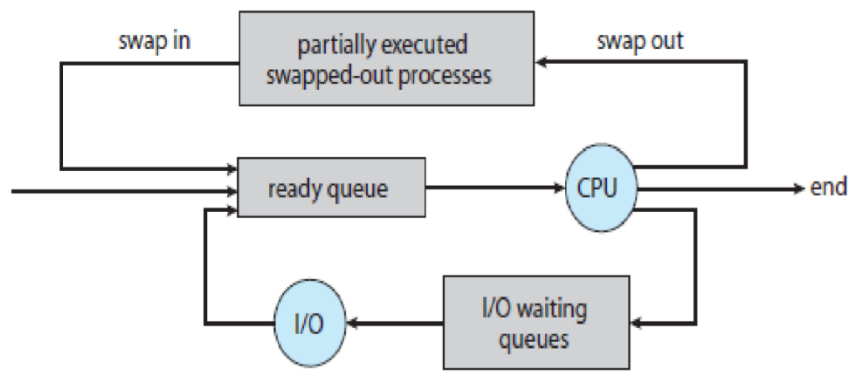
• The **short-term scheduler**, or CPU Scheduler, runs very frequently, on the order of 100 milliseconds, and must very quickly swap one process out of the CPU and swap in another one. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

• Some systems also employ a **medium-term scheduler**. When system loads get high, this scheduler will swap one or more processes out of the ready queue system for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system. See the differences in Figures below.

• An efficient scheduling system will select a good **process mix** of **CPU-bound** processes and **I/O bound** processes.



Queueing-diagram representation of process scheduling

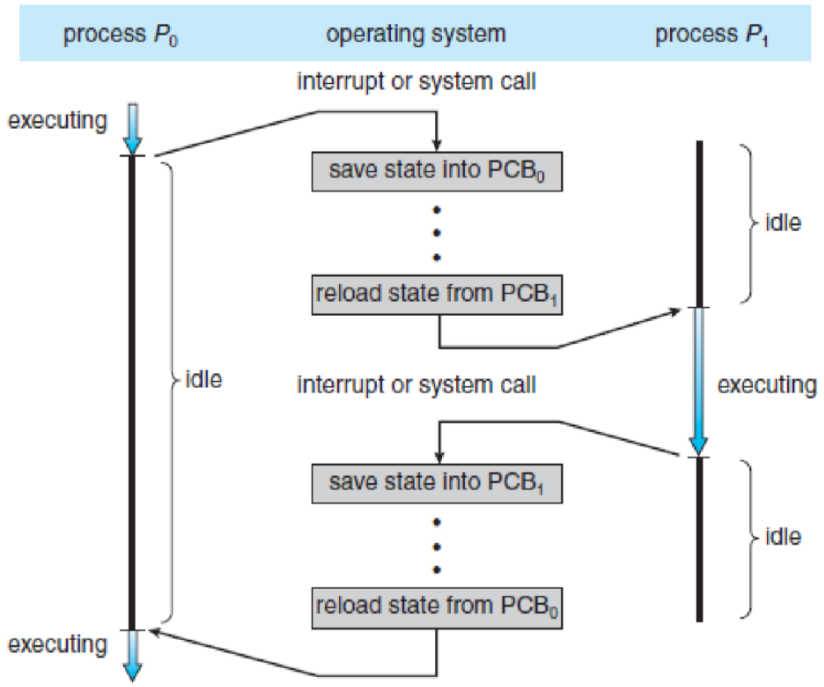


Addition of a medium-term scheduling to the queuing diagram

2.2.3 Context Switch

Definition: Switching the CPU between processes is called context switch. A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block

- Whenever an interrupt arrives, the CPU must do a **state-save** of the currently running process, then switch into kernel mode to handle the interrupt, and then do a **state-restore** of the interrupted process.
- Similarly, a **context switch** occurs when the time slice for one process has expired and a new process is to be loaded from the ready queue. This will be initiated by a timer interrupt, which will then cause the current process's state to be saved and the new process's state to be restored.
- Saving and restoring states involves saving and restoring all of the registers and program counter(s), as well as the process control blocks described above.
- Context switching happens VERY VERY frequently, and the overhead of doing the switching is just lost CPU time, so context switches (state saves & restores) need to be as fast as possible.



2.8 PROCESS SYNCHRONIZATION

2.8.1 Background

Cooperating processes (those that can effect or be effected by other simultaneously running processes) with the producer-consumer problem as an example

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 :

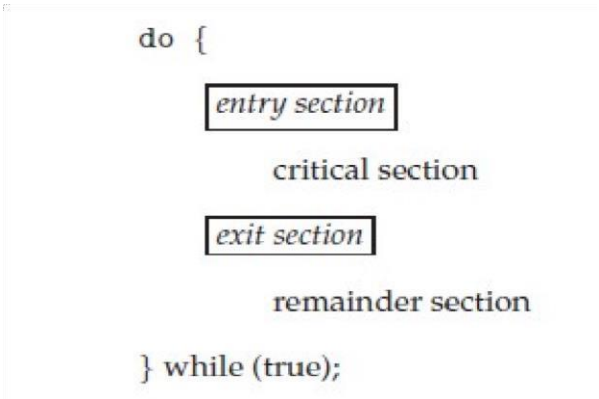
```
itemnextConsumed;
while( true ) {
/* Wait for an item to become available */
while( in == out ) ;
/* Do nothing */
/* Get the next available item */
nextConsumed = buffer[ out ] ;
out = (out + 1) % BUFFER_SIZE;
/* Consume the item in nextConsumed
( Do something with it ) */
}
```

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:

2.8.2 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.
- The code preceding the critical section, and which controls access to the critical section, is termed the entry section.
- The code following the critical section is termed the exit 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

2.8.3 Solution to critical section problem

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

Mutual Exclusion

- Only one process at a time can be executing in their critical section.

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.)

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.)

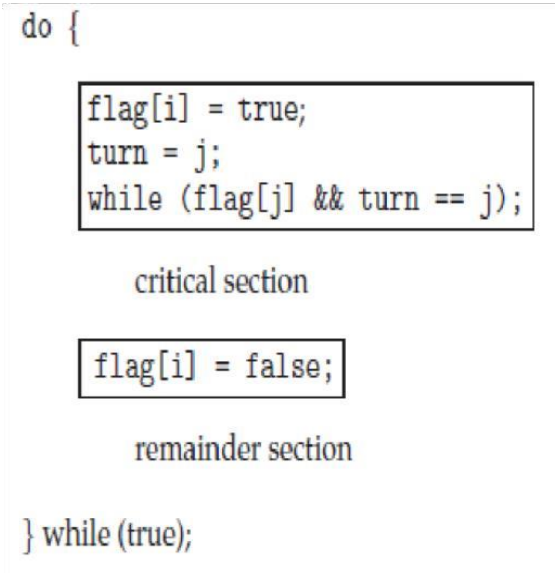
- We assume that all processes proceed at a non-zero speed, but no assumptions can be made regarding the **relative** speed of one process versus another.
- Kernel processes can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures 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 realtime 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.

2.8.4 Peterson's Solution

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



To prove that the solution is correct, we must examine the three conditions listed above:

- **Mutual exclusion** - If one process is executing their critical section when the other wishes to do so, the second process will become blocked by the flag of the first process. If both processes attempt to enter at the same time, the last process to execute "turn = j" will be blocked.
- **Progress** - Each process can only be blocked at the while if the other process wants to use the critical section (flag[j] = = true), AND it is the other process's turn to use the critical section (turn = = j). If both of those conditions are true, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, will set flag[j] to false, releasing process i. The shared variable turn assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.
- **Bounded Waiting** - As each process enters their entry section, they set the turn variable to be the other processes turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.

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

2.8.5 Synchronization Hardware

- To generalize the solution(s) expressed above, each process when entering their critical section must set some sort of **lock**, to prevent other processes from

entering their critical sections simultaneously, and must release the lock when exiting their critical section, to allow other processes to proceed. Obviously it must be possible to attain the lock only when no other process has already set a lock. Specific implementations of this general procedure can get quite complicated, and may include hardware solutions as outlined in this section.

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

booleantest_and_set (boolean *target)

{

booleanrv = *target;

*target = TRUE;

return rv: }

Solution using test-and-set:

```
do
{       while (test_and_set(&lock))
        ; /* do nothing */              /* critical section */
lock = false;
          /* remainder section */
    } while (true);
int compare _and_swap(int *value, int expected, intnew_value)
{       int temp = *value;
if (*value == expected)
*value = new_value;
return temp;
```

```
}
```

Solution using compare-and swap:

```
do {
while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */        /* critical section */
lock = 0;
      /* remainder section */
   } while (true);
```

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

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

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

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

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

Bounded-waiting mutual exclusion with TestAndSet( ).

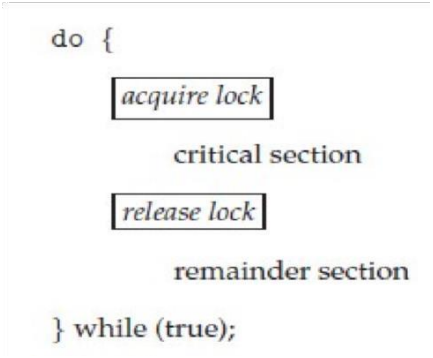
The key feature of the above algorithm is that a process blocks on the AND of the critical section being locked and that this process is in the waiting state. When exiting a critical section, the exiting process does not just unlock the critical section and let the other processes have a free-for-all trying to get in.

Rather it first looks in an orderly progression (starting with the next process on the list) for a process that has been waiting, and if it finds one, then it releases that particular process from its waiting state, without unlocking the critical section, thereby allowing a specific process into the critical section while continuing to block all the others. Only if there are no other processes currently waiting is the general lock removed, allowing the next process to come along access to the critical section.

2.8.6 Mutex Locks

- The hardware solutions presented above are often difficult for ordinary programmers to access, particularly on multi-processor machines, and particularly because they are often platform-dependent.
- Therefore most systems offer a software API equivalent called **mutex locks** or simply **mutexes.** (For mutual exclusion)

- The terminology when using mutexes is to **acquire** a lock prior to entering a critical section, and to **release it when exiting, as shown in Figure :**




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


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

2.8.7 Semaphores

A more robust alternative to simple mutexes is to use **semaphores**, which are integer variables for which only two (atomic) operations are defined, the wait and signal operations, as shown in the following figure.

```
wait(S){   while(S<=0)     ; // busy wait   S--; } signal(S){
  S++;
}
```

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.

Semaphore Usage

In practice, semaphores can take on one of two forms:

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

- **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:

```
typedefstruct{          int value;
struct process*list;
  }semaphore; wait(semaphore *S){   S->value--;
if (S->value < 0) {
add this process to S->list;      block();
  }
}signal(semaphore *S){   S->value++;
if (S->value <= 0) {
```

```
remove a process P from S->list;      wakeup(P);
  }
}
```

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

Deadlocks and Starvation

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

- Let ***S*** and ***Q*** be two semaphores initialized to 1

  P_0 P_1

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

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

Priority Inversion

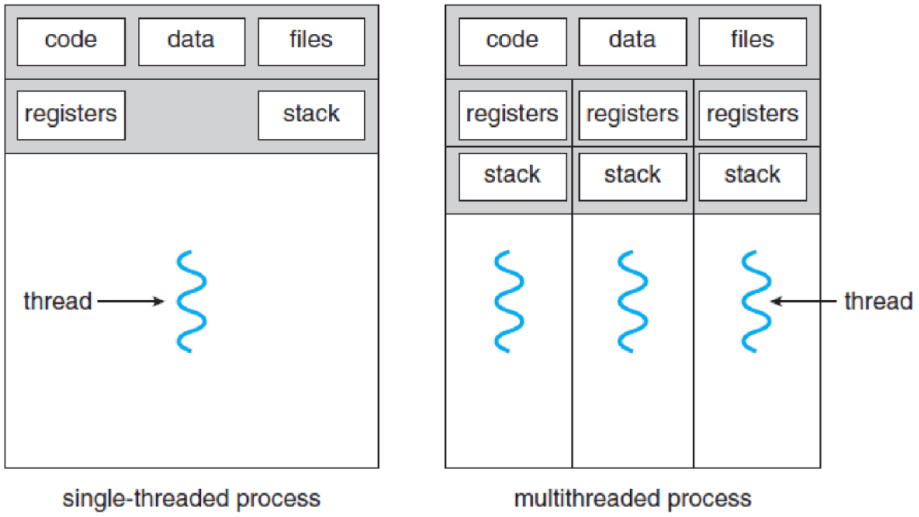
- A challenging scheduling problem arises when a high-priority process gets blocked waiting for a resource that is currently held by a low-priority process.
- If the low-priority process gets pre-empted by one or more medium-priority processes, then the high-priority process is essentially made to wait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

2.7 THREADS

2.7.1 Overview

A **thread** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)

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

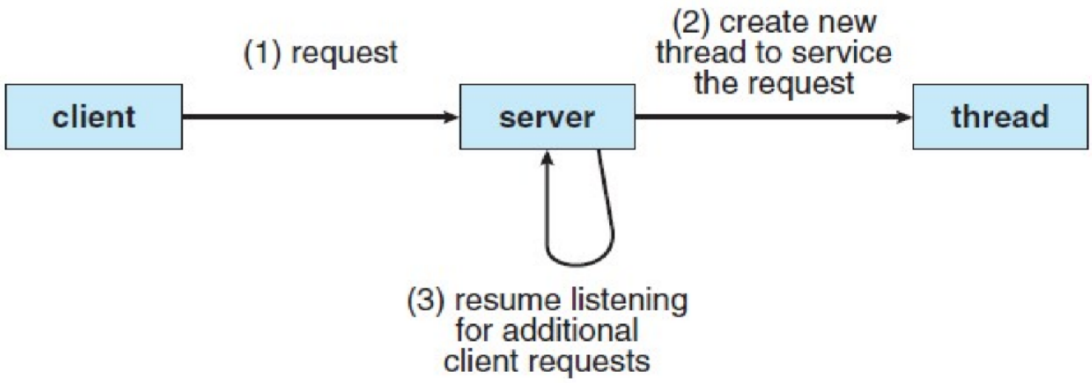


Single-threaded and multithreaded processes

2.7.1.1 Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
- Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork

off separate processes for every incoming request. (The latter is how this sort of thing was done before the concept of threads was developed. A daemon would listen at a port, fork off a child for every incoming request to be processed, and then go back to listening to the port.)



Multithreaded server architecture

2.7.1.2 Benefits

There are four major categories of benefits to multi-threading:

1. Responsiveness - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.

2. Resource sharing - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.

3. Economy - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.

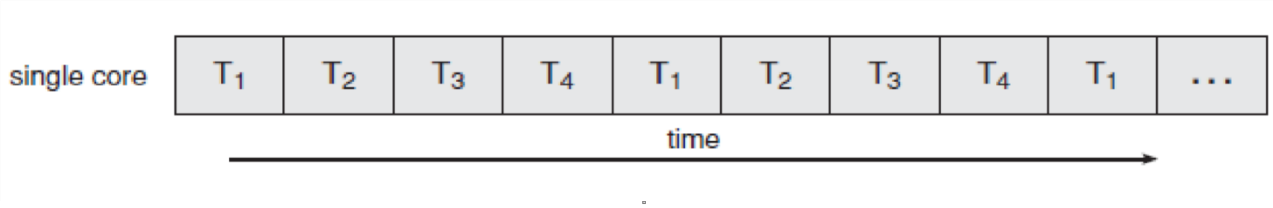
4. Scalability, i.e. Utilization of multiprocessor architectures - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors.

(Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold.)

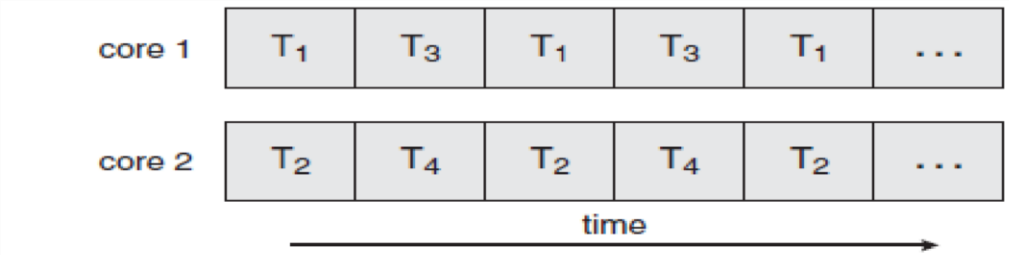
2.7.2 Multicore Programming

- A recent trend in computer architecture is to produce chips with multiple **cores**, or CPUs on a single chip.

- A multi-threaded application running on a traditional single-core chip would have to interleave the threads, as shown in Figure.
- On a multi-core chip, however, the threads could be spread across the available cores, allowing true parallel processing, as shown in figure



Concurrent execution on a single-core system.



Parallel execution on a multicore system

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

2.7.2.1 Programming Challenges

For application programmers, there are five areas where multi-core chips present new challenges:

- **Identifying tasks** - Examining applications to find activities that can be performed concurrently.
- **Balance** - Finding tasks to run concurrently that provide equal value. I.e. don't waste a thread on trivial tasks.
- **Data splitting** - To prevent the threads from interfering with one another.
- **Data dependency** - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.
- **Testing and debugging** - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

2.7.2.2 Types of Parallelism

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

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

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

2.7.3 Multithreading Models

There are two types of threads to be managed in a modern system: User threads and kernel threads.

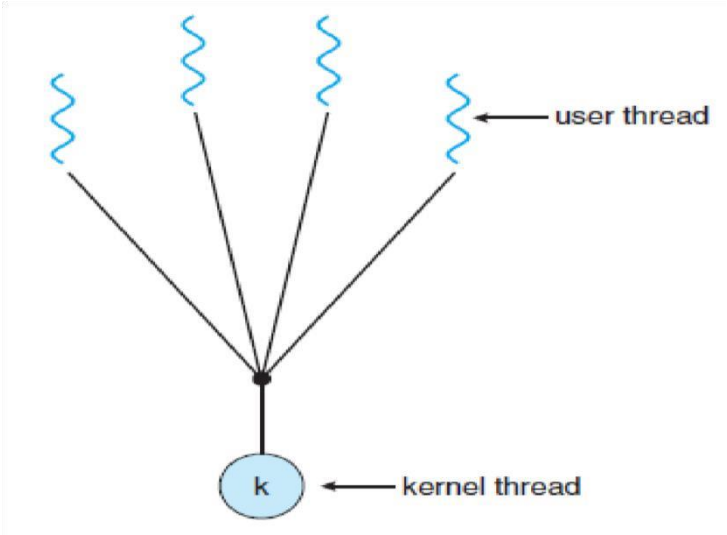
User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.

Kernel threads are supported within the kernel of the OS itself. All modern OSes support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.

2.7.3.1 Many-To-One Model

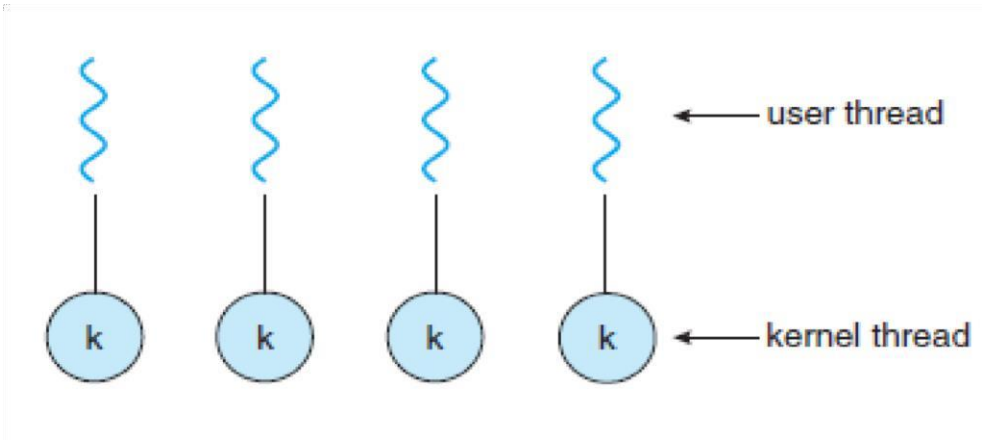
- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is very efficient.
- However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue.
- Because a single kernel thread can operate only on a single CPU, the many-to-one model does not allow individual processes to be split across multiple CPUs.
- Green threads for Solaris and GNU Portable Threads implement the many-to one model in the past, but few systems continue to do so today.



Many-to-one model

2.7.3.2 One-To-One Model

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

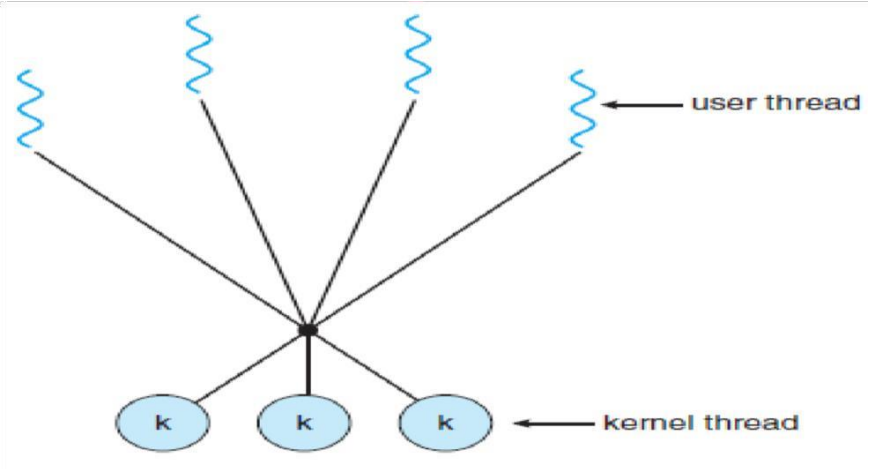


One-to-one model

2.7.3.3 Many-To-Many Model

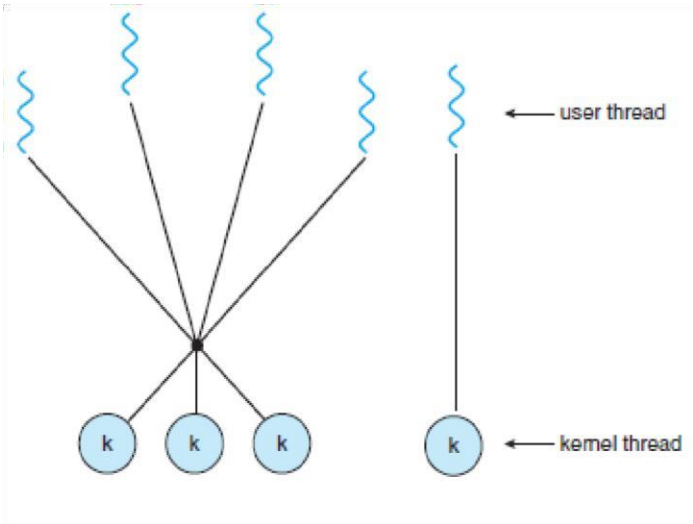
- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to one and many-to-one models.
- Users have no restrictions on the number of threads created.

- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.



Many-to-many model

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



Two-level model

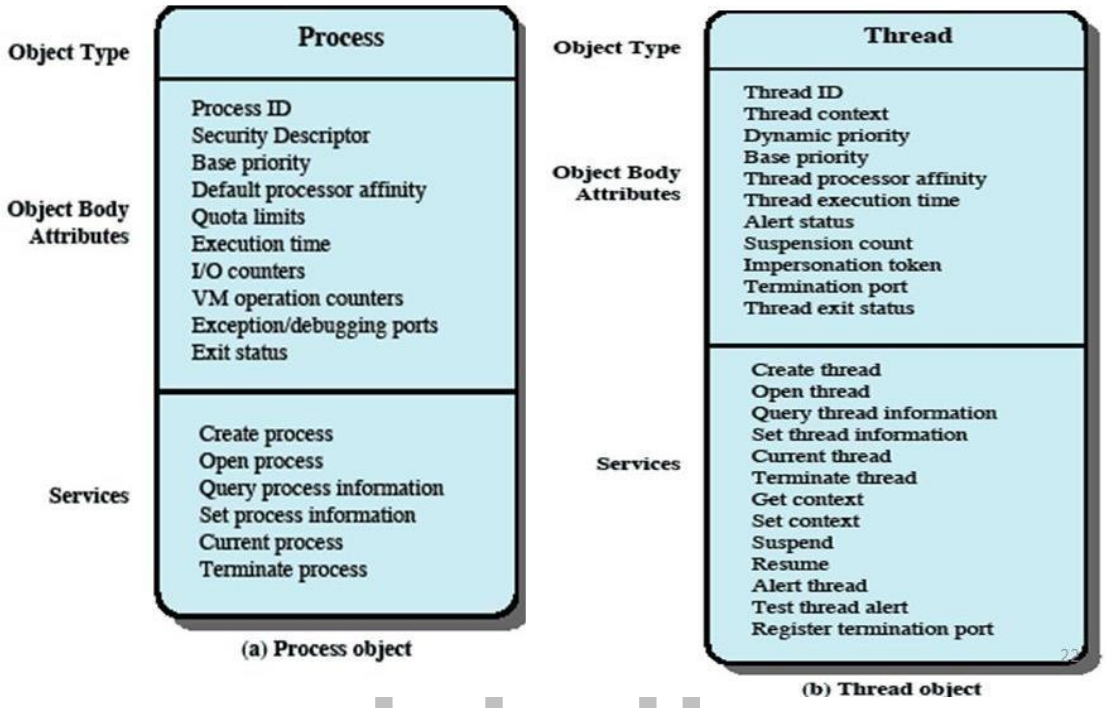
2.7.4 Windows 7 Threads and SMP Management

Windows process design is driven by the need to provide support for a variety of OS environments. Processes supported by different OS environments differ in a number of ways, including the following:

- How processes are named
- Whether threads are provided within processes
- How processes are represented
- How process resources are protected
- What mechanisms are used for interprocess communication and synchronization
- How processes are related to each other

Important characteristics of Windows processes are the following:

- Windows processes are implemented as objects.
- A process can be created as new process, or as a copy of an existing process.
- An executable process may contain one or more threads.
- Both process and thread objects have built-in synchronization capabilities.
- Figure illustrates the way in which a process relates to the resources it controls or uses.
- Each process is assigned a security access



(a) Process object

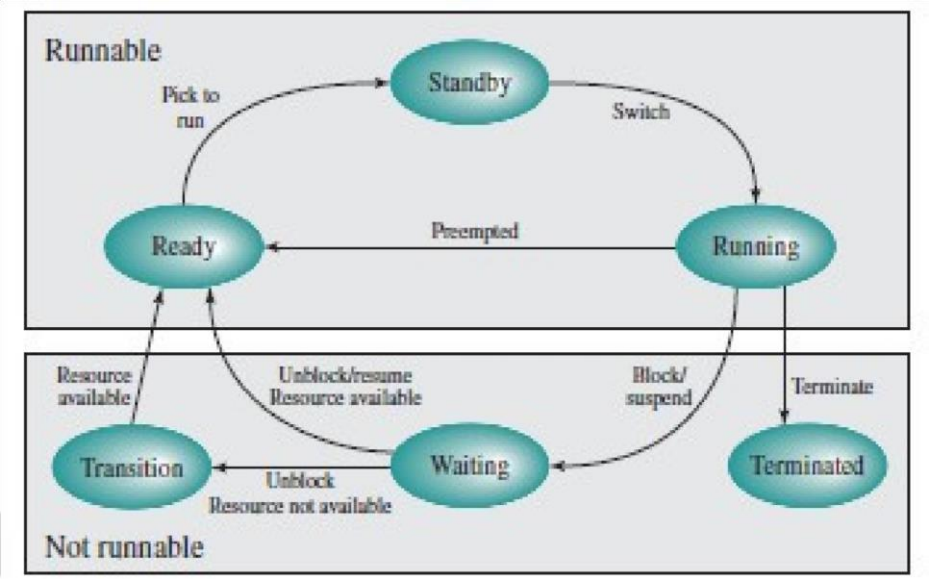
(b) Thread object

2.7.4.1 Thread States

An existing Windows thread is in one of six states

- **Ready:** A ready thread may be scheduled for execution. The Kernel dispatcher keeps track of all ready threads and schedules them in priority order.
- **Standby:** A standby thread has been selected to run next on a particular processor. The thread waits in this state until that processor is made available. If the standby thread's priority is high enough, the running thread on that processor may be preempted in favor of the standby thread. Otherwise, the standby thread waits until the running thread blocks or exhausts its time slice.
- **Running:** Once the Kernel dispatcher performs a thread switch, the standby thread enters the Running state and begins execution and continues execution until it is preempted by a higher priority thread, exhausts its time slice, blocks, or terminates. In the first two cases, it goes back to the Ready state.
- **Waiting:** A thread enters the Waiting state when (1) it is blocked on an event (e.g., I/O), (2) it voluntarily waits for synchronization purposes, or (3) an environment subsystem directs the thread to suspend itself. When the waiting condition is satisfied, the thread moves to the Ready state if all of its resources are available.

- **Transition:** A thread enters this state after waiting if it is ready to run but the resources are not available. For example, the thread's stack may be paged out of memory. When the resources are available, the thread goes to the Ready state.
- **Terminated:** A thread can be terminated by itself, by another thread, or when its parent process terminates. Once housekeeping chores are completed, the thread is removed from the system, or it may be retained by the Executive 6 for future reinitialization.



2.7.4.2 Symmetric Multiprocessing Support

- Windows supports SMP hardware configurations. The threads of any process, including those of the executive, can run on any processor.
- In the absence of affinity restrictions, explained in the next paragraph, the kernel dispatcher assigns a ready thread to the next available processor.
- This assures that no processor is idle or is executing a lower-priority thread when a higher priority thread is ready.
- Multiple threads from the same process can be executing simultaneously on multiple processors.
- As a default, the kernel dispatcher uses the policy of **soft affinity** in assigning threads to processors:
- The dispatcher tries to assign a ready thread to the same processor it last ran on.
- This helps reuse data still in that processor's memory caches from the previous execution of the thread.
- It is possible for an application to restrict its thread execution only to certain processors (**hard affinity**).