Module 3 – Scheduling

Processes Scheduling - CPU Scheduling: Pre-emptive, non-pre-emptive - Multiprocessor scheduling – Deadlocks - Resource allocation and management - Deadlock handling mechanisms: prevention, avoidance, detection, recovery

Process Scheduling

- The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization.
- The objective of time sharing is to switch a CPU core among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on a core.
- Each CPU core can run one process at a time.
- For a system with a single CPU core, there will never be more than one process running at a time, whereas a multicore system can run multiple processes at one time.
- If there are more processes than cores, excess processes will have to wait until a core is free and can be rescheduled.
- The number of processes currently in memory is known as the degree of multiprogramming.

Scheduling Queues

As processes enter the system, they are put into a ready queue, where they are ready and waiting to execute on a CPU's core,

This queue is generally stored as a linked list; a ready-queue header contains pointers to the first PCB in the list, and each PCB includes a pointer field that points to the next PCB in the ready queue.

When a process is allocated a CPU core, it executes for a while and eventually terminates, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.

Suppose the process makes an I/O request to a device such as a disk. Since devices run significantly slower than processors, the process will have to wait for the I/O to become available.

Processes that are waiting for a certain event to occur such as completion of I/O are placed in a wait queue.

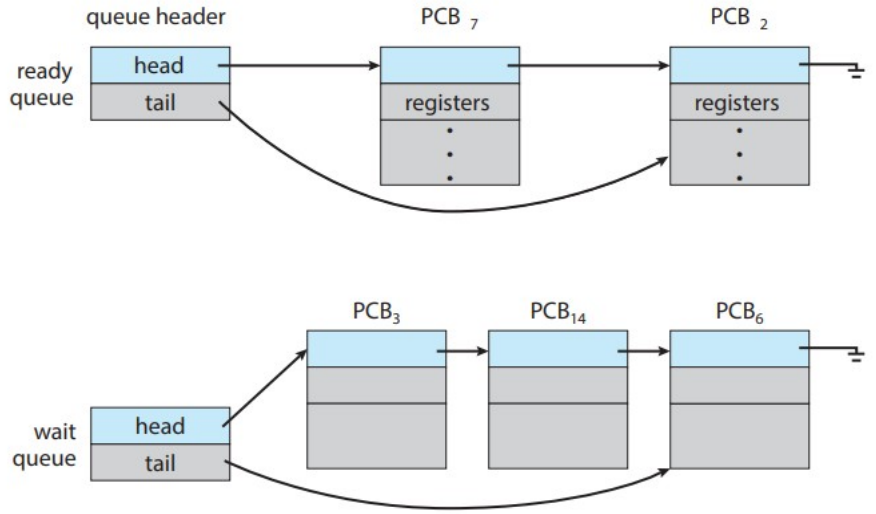


Figure 3.4 The ready queue and wait queues.

Scheduling Queues

A common representation of process scheduling is a queueing diagram.

Two types of queues are present: the ready queue and a set of wait queues.

The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue.

It waits there until it is selected for execution, or dispatched.

Once the process is allocated a CPU core and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O wait queue.
- The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.
- The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.

A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Scheduling Queues

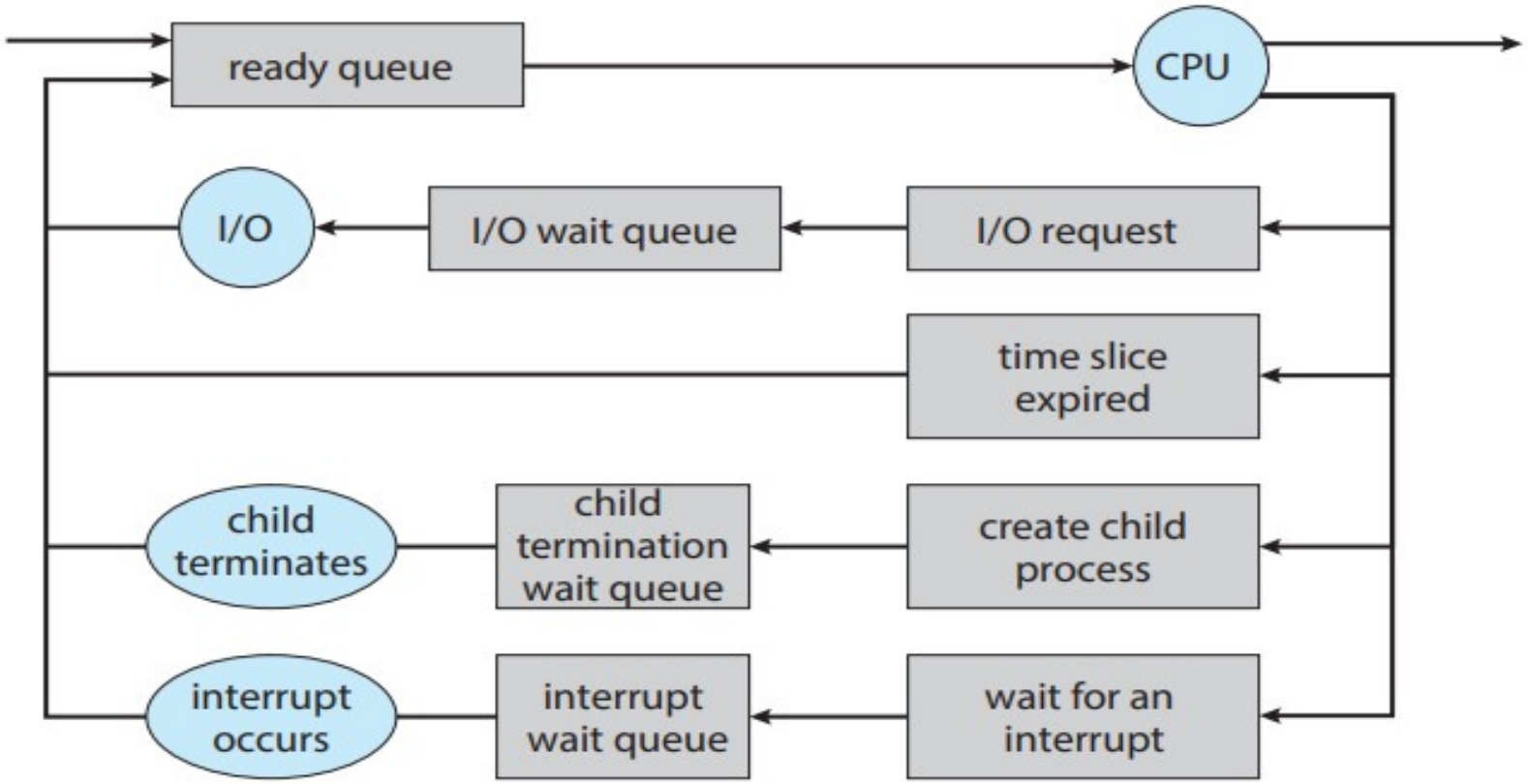


Figure 3.5 Queueing-diagram representation of process scheduling.

CPU Scheduling

A process migrates among the ready queue and various wait queues throughout its lifetime.

The role of the CPU scheduler is to select from among the processes that are in the ready queue and allocate a CPU core to one of them.

An intermediate form of scheduling, known as swapping, is to remove a process from memory and later, the process can be reintroduced into memory, and its execution can be continued where it left off.

This scheme is known as swapping because a process can be “swapped out” from memory to disk, where its current status is saved, and later “swapped in” from disk back to memory, where its status is restored.

CPU Scheduling

In a simple computer system, the CPU then just sits idle waiting for the completion of some I/O request. All this waiting time is wasted; and with multiprogramming, this time can be used productively.

Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.

Every time one process has to wait, another process can take over use of the CPU.

On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.

CPU – I/O Burst Cycle

The process execution consists of a cycle of CPU execution and I/O wait.

Processes alternate between these two states.

Process execution begins with a CPU burst followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.

Eventually, the final CPU burst ends with a system request to terminate execution (Figure 5.1)

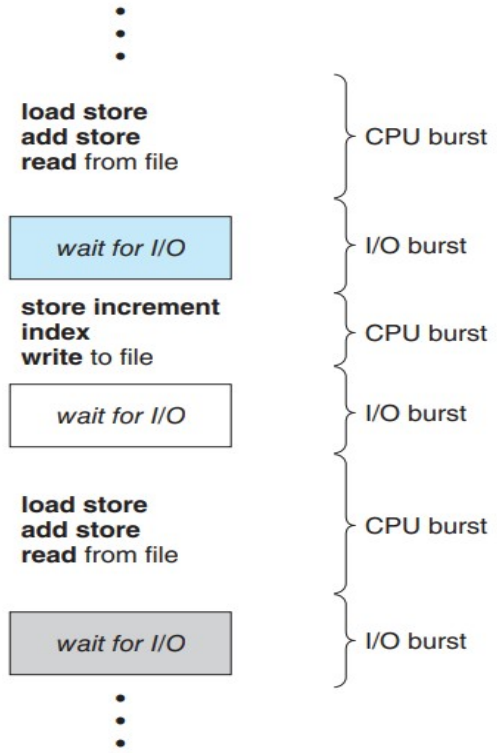


Figure 5.1 Alternating sequence of CPU and I/O bursts.

Pre-emptive and Non-preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process if exists in the ready queue must be selected for execution. When scheduling takes place under circumstances 1 and 4, the scheduling scheme is said to be non-preemptive or cooperative. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.

However, for situations 2 and 3, it is preemptive. All modern operating systems including Windows, macOS, Linux, and UNIX use preemptive scheduling algorithms.

Dispatcher

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

- Switching context from one process to another
- Switching to user mode
- Jumping to the proper location in the user program to resume that program

The dispatcher should be as fast as possible, since it is invoked during every context switch. T r the dispatcher to stop one process and start anot wn as the dispatch latency and is illustrated in Figu

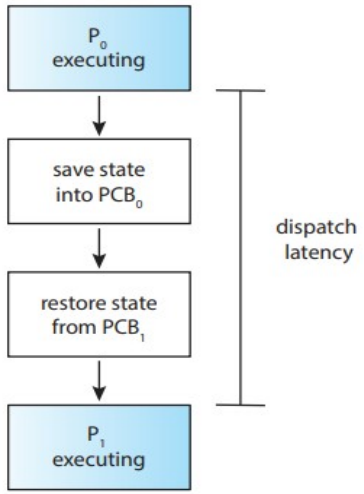


Figure 5.3 The role of the dispatcher.

Scheduling Criteria

CPU utilization: The CPU should be kept as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).

Throughput: If the CPU is busy executing processes, then work is being done. The measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process over several seconds; for short transactions, it may be tens of processes per second.

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

Waiting time: The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

Response time: The time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

CPU Scheduling Algorithms

1. First-Come, First-Served Scheduling(FCFS)
2. Shortest Job First Scheduling(SJF)
3. Shortest Remaining Time First(SRTF)
4. Priority Scheduling
5. Round Robin Scheduling(RR)

Scheduling Algorithms - First-Come, First-Served Scheduling (FCFS)

The simplest CPU-scheduling algorithm is the first-come first-serve (FCFS) scheduling algorithm.

With this scheme, the process that requests the CPU first is allocated the CPU first.

The implementation of the FCFS policy is easily managed with a FIFO queue.

When a process enters the ready queue, its PCB is linked onto the tail of the queue.

When the CPU is free, it is allocated to the process at the head of the queue.

The running process is then removed from the queue.

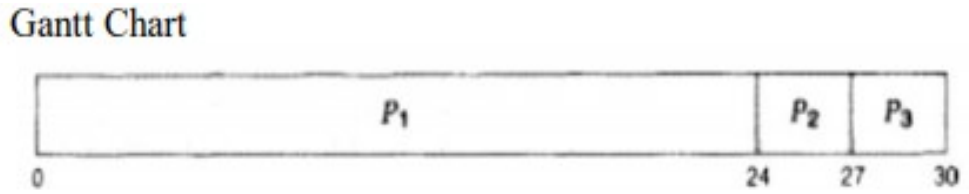
The average waiting time under the FCFS policy is often quite long.

First-Come, First-Served Scheduling

- The process that requests the CPU first is allocated the CPU first.
- It is a non-preemptive scheduling technique.

Example:

Process	Burst Time	Waiting time	TAT
P1	24	0	24
P2	3	24	27
P3	3	27	30

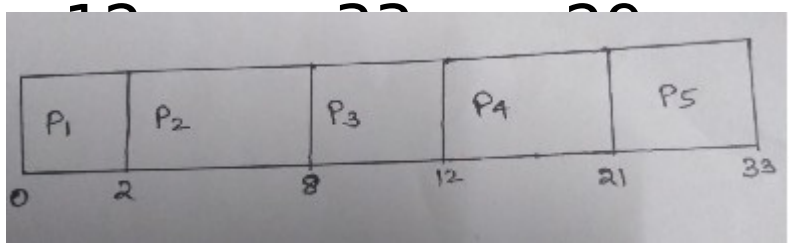


Average waiting time = (0+24+27) / 3 = 17 ms
Average Turnaround time = (24+27+30) / 3 = 27 ms

First-Come, First-Served Scheduling

Example:

Process	ArrivalTime	Burst Time	C.T	TAT	WT
P1	0	2	2	2	0
P2	1	6	8	7	1
P3	2	4	12	10	6
P4	3	9	21	18	9
P5	4			17	



Turn Around Time = Completion Time - Arrival Time

Waiting Time = Turnaround time - Burst Time

Average Waiting Time= (0+1+6+9+17)/5=33/5=6.6 ms

Average TurnAroundtime=(2+7+10+18+29)/5=68/5=13.6 ms

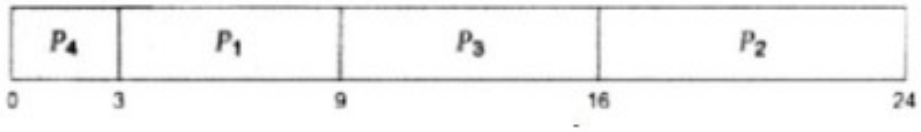
Shortest Job First Scheduling

- It is a non-preemptive scheduling technique.
- The CPU is assigned to the process that has the smallest next CPU burst.
- If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.

Example:

Process	Burst Time	WT	TAT
P1	6	3	9
P2	8	16	24
P3	7	9	16
P4	3	0	3

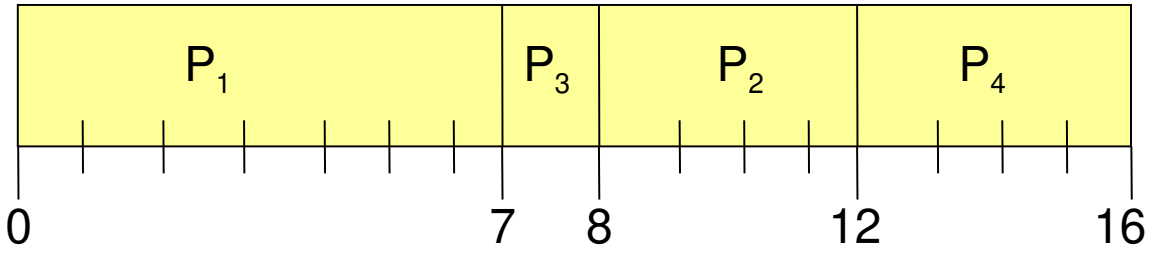
Gantt Chart



Average waiting time is (3 + 16 + 9 + 0)/4 = 7 ms
Average turnaround time = (3+9+16+24) / 4 = 13 ms

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time	C.T	TAT	WT
P_1	0	7	7	7	0
P_2	2	4	12	10	6
P_3	4	1	8	4	3
P_4	5	4	16	11	7

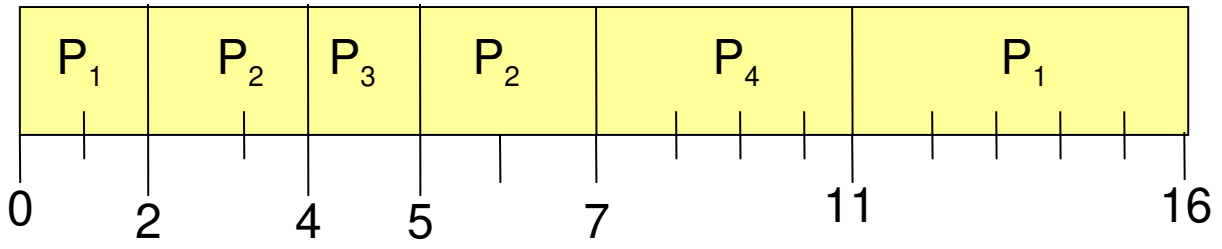


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4 ms
- Average TAT=(7+10+4+11)/4=32/4=8 ms

Shortest Remaining Time First (SRTF) Scheduling - Preemptive SJF

Process	A.T	B.T	C.T			TAT	WT
P_1	0	7	16		16	9	
P_2	2	4	7 5	1			
P_3	4	1	5	1	0		
P_4	5	4	11	6	2		

SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3 ms
- Average Turnaroundtime=(16+5+1+6)/4=28/4=7ms

Priority Scheduling

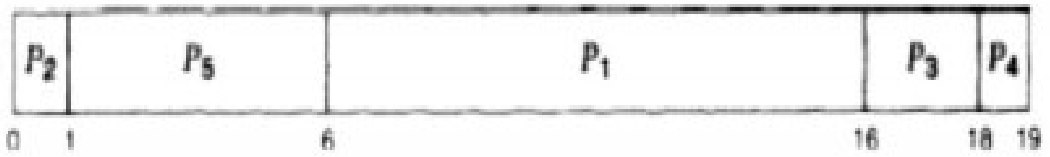
- Priority scheduling can be either preemptive or non-preemptive.
- When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
- A major problem with priority scheduling algorithms is indefinite blocking, or starvation.
- A process that is ready to run but waiting for the CPU can be considered blocked.
- A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
- A solution to the problem of indefinite blockage of low-priority processes is aging. Aging involves gradually increasing the priority of processes that wait in the system for a long time.

Priority Scheduling

- A priority is associated with each process, and the CPU is allocated to the process with the highest priority. (smallest integer=highest priority).

Example :

Process	Burst Time	Priority	WT	TAT
P1	10	3	6	16
P2	1	1	0	1
P3	2	4	16	18
P4	1	5	18	19
P5	5	2	1	6

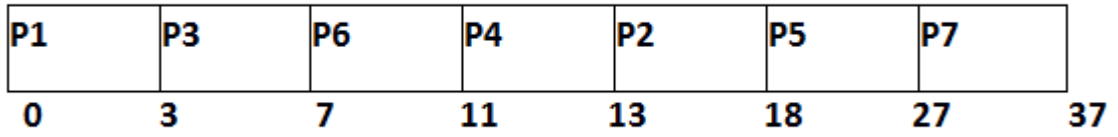


Average waiting time
=41/5=8.2 ms

Non Preemptive Priority Scheduling

Example :

Process	B.T	Priority	A.T	C.T	TAT	W.T
P1	3	2	0	3	3	0
P2	5	6	2	18	16	11
P3	4	3	1	7	6	2
P4	2	5	4	13	9	7
P5	9	7	6	27	21	12
P6	4	4	5	11	6	2
P7	10	10	7	37	30	20



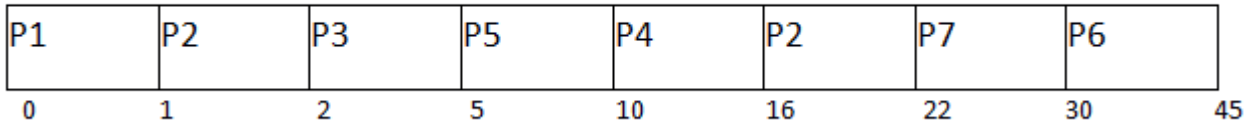
Average TAT=91/7=13 ms

Average WT=54/7=7.7ms

Preemptive Priority Scheduling

Example :

Process	B.T	Priority	A.T	C.T	TAT	W.T
P1	1 0	2	0	1	1	0
P2	7 6	6	1	22	21	14
P3	3 0	3	2	5	3	0
P4	6 0	5	3	16	13	7
P5	5 0	4	4	10	6	1
P6	15	10	5	45	40	25
P7	8	9	6	30	24	16



Average TAT=108/7= 15.4ms

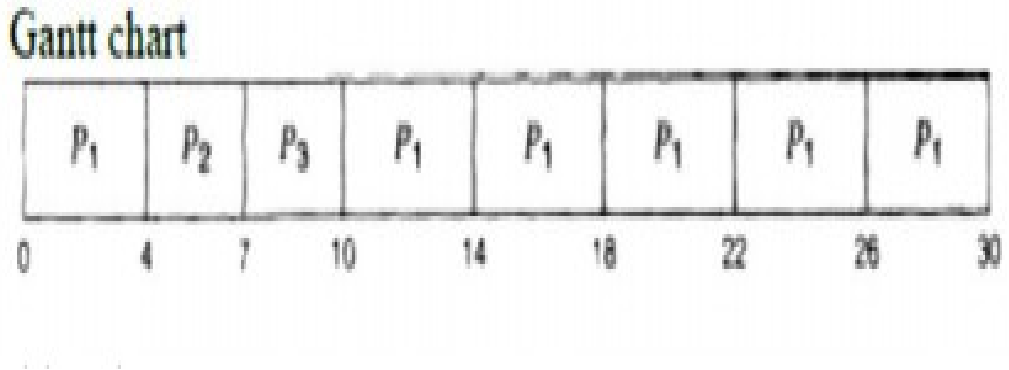
Average WT=63/7=9ms

Round-Robin Scheduling

- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to switch between processes.
- A small unit of time, called a time quantum (or time slice), is defined.
- The ready queue is treated as a circular queue.

Example: **Time Quantum = 4 ms**

Process	Burst Time
P1	24 20 16 12 8 4 0
P2	3 0
P3	3 0

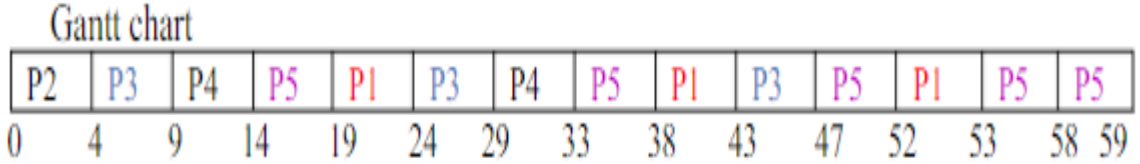


Waiting time
P1 = 26 – 20 = 6
P2 = 4
P3 = 7
Average waiting time is (6+4+7 / 3 = 5.66 ms) ,17/3 = 5.66

Round-Robin Scheduling

Process	B.T	A.T	C.T	TAT	WT
P1	11 6 1	0	5	53	48	37
P2	4 0	0	4	4	0
P3	14 9 4	0	0	47	47	33
P4	9 4 0	1	33	32	23
P5	21 16 11 6 1 0	2	59	57	36

Timequantum=5 ms



Average TAT=188/5=37.6 ms

Average WT=129/5=25 .8 ms

Multiprocessor Scheduling

If multiple CPUs are available, load sharing, where multiple threads may run in parallel, becomes possible, however scheduling issues become correspondingly more complex.

The term multiprocessor referred to systems that provided multiple physical processors, where each processor contained one single-core CPU.

Multiprocessor now applies to the following system architectures:

- Multicore CPUs
- Multithreaded cores
- NUMA systems
- Heterogeneous multiprocessing

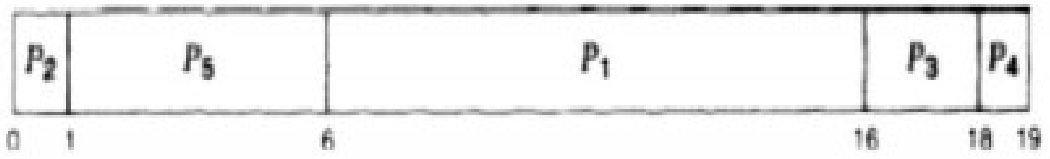
In the first three architectures, the processors are identical, homogeneous in terms of their functionality but in the last architecture the processors are not identical in their capabilities.

Priority Scheduling

- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.(smallest integer=highest priority).

Example :

Process	Burst Time	Priority	WT	TAT
P1	10	3	6	16
P2	1	1	0	1
P3	2	4	16	18
P4	1	5	18	19
P5	5	2	1	6



Average waiting time
=41/5=8.2 ms

Approches to Multiprocessor Scheduling

The approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor, the master server.

The other processors execute only user code.

This asymmetric multiprocessing is simple because only one core accesses the system data structures, reducing the need for data sharing.

The downfall of this approach is the master server becomes a potential bottleneck where overall system performance may be reduced.

The standard approach for supporting multiprocessors is symmetric multiprocessing (SMP), where each processor is self-scheduling.

Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a thread to run. This provides two possible strategies for organizing the threads eligible to be scheduled:

1. All threads may be in a common ready queue.
2. Each processor may have its own private queue of threads.

Approaches to Multiprocessor Scheduling

These two strategies are contrasted in Figure 5.11.

If we select the first option, we have a possible race condition on the shared ready queue and therefore must ensure that two separate processors do not choose to schedule the same thread and that threads are not lost from the queue.

The second option permits each processor to schedule threads from its private run queue and therefore does not suffer from the possible performance problems associated with a shared run queue. Thus, it i h on systems supporting SMP.

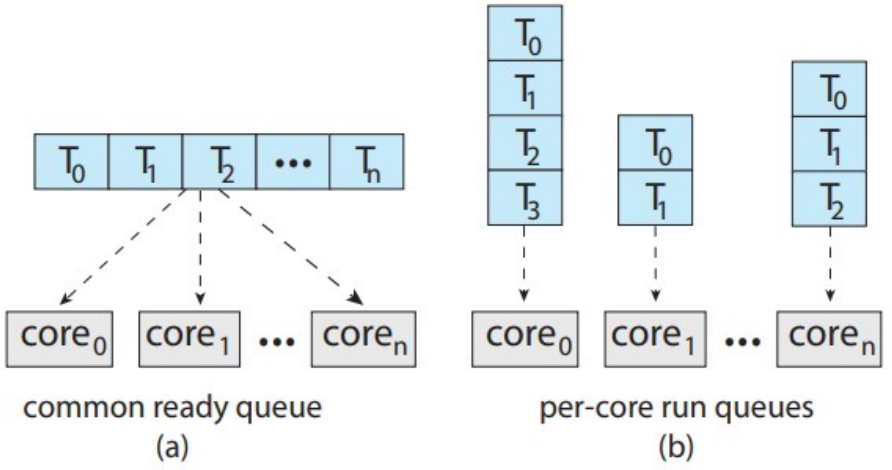


Figure 5.11 Organization of ready queues.

Multicore Processors

The computer hardware now places multiple computing cores on the same physical chip, resulting in a multicore processor.

Each core maintains its architectural state and thus appears to the operating system to be a separate logical CPU.

SMP systems that use multicore processors are faster and consume less power than systems in which each CPU has its own physical chip.

Multicore processors may complicate scheduling issues. Let's consider how this can happen.

When a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a memory stall, occurs because modern processors operate at much faster speeds than memory. However, a memory stall can also occur because of a cache miss (accessing data that are not in cache memory). Figure 5.12 illustrates a memory stall. In this scenario, percent of its time waiting for data to become

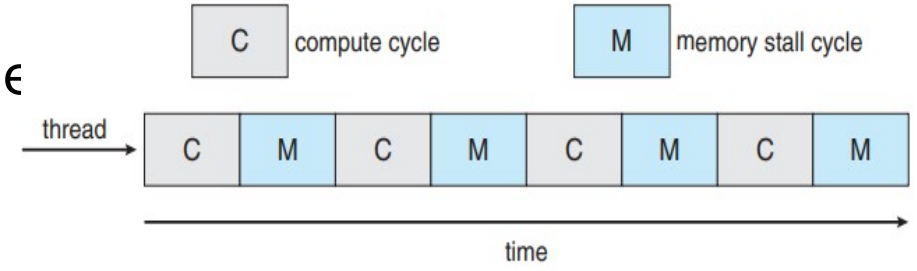


Figure 5.12 Memory stall.

Multithreaded cores

To remedy this situation, many recent hardware designs have implemented multithreaded processing cores in which two (or more) hardware threads are assigned to each core.

If one hardware thread stalls while waiting for memory, the core can switch to another thread.

Figure 5.13 illustrates a dual-threaded processing core on which the execution of thread 0 and the execution of thread 1 are interleaved.

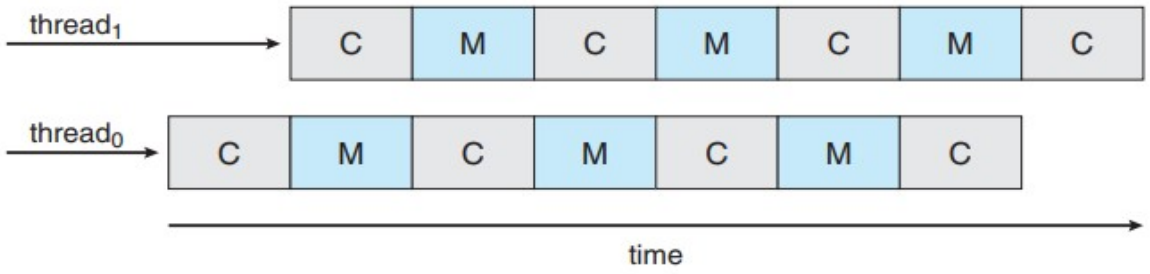


Figure 5.13 Multithreaded multicore system.

Multithreaded cores

From an operating system perspective, each hardware thread maintains its architectural state, such as instruction pointer and register set, and thus appears as a logical CPU that is available to run a software thread.

This technique known as chip multithreading (CMT) is illustrated in Figure 5.14.

Here, the processor contains four computing cores, with each core containing two hardware threads. From the perspective of the operating system, there are eight logical C

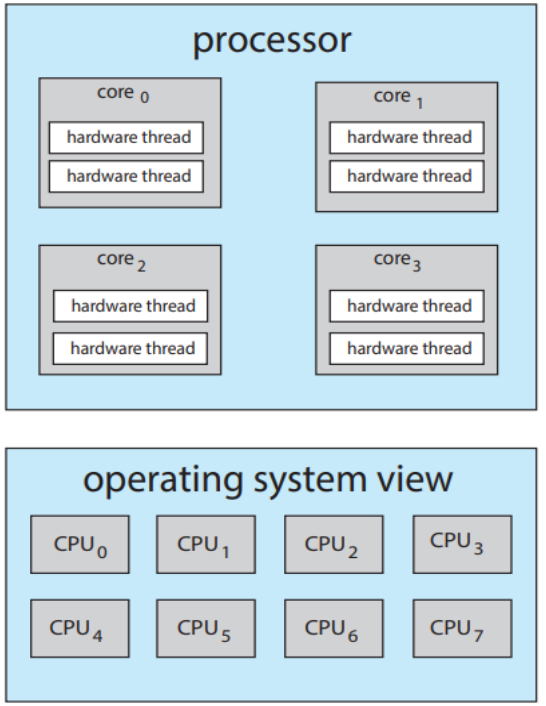


Figure 5.14 Chip multithreading.

Multithreaded cores

There are two ways to multithread a processing core: coarse grained and fine-grained multithreading.

With coarse-grained multithreading, a thread executes on a core until a long-latency event such as a memory stall occurs. Because of the delay caused by the long-latency event, the core must switch to another thread to begin execution. However, the cost of switching between threads is high, since the instruction pipeline must be flushed before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions.

Fine-grained (or interleaved) multithreading switches between threads at a much finer level of granularity— typically at the boundary of an instruction cycle. However, the architectural design of fine-grained systems includes logic for thread switching. As a result, the cost of switching between threads is small.

It is important to note that the resources of the physical core (such as caches and pipelines) must be shared among its hardware threads, and therefore a processing core can only execute one hardware thread at a time.

Multithreaded cores

Consequently, a multithreaded, multicore processor actually requires two different levels of scheduling, as shown in Figure 5.15, which illustrates a dual-threaded processing core.

On one level are the scheduling decisions that must be made by the operating system as it chooses which software thread to run on each hardware thread (logical CPU).

A second level of scheduling specifies how each core decides which hardware thread to run. The approach is to use a simple round-robin algorithm to schedule a hardware thread to the processing core.

Assume that a CPU has two processing cores, and each core has two hardware threads.

If two software threads are running on this system, they can be running either on the same core or on separate cores.

If they are both scheduled to run on the same core, they have to share processor resources and thus are likely to proceed more slowly cheduled on separate cores.

If the operating system is aware of the level of pr aring, it can schedule software threads onto logical processors tha rces.

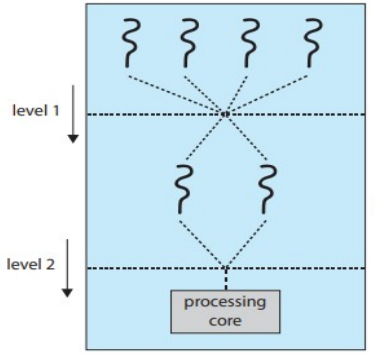


Figure 5.15 Two levels of scheduling.

NUMA

An architecture featuring non-uniform memory access (NUMA) where there are two physical processor chips each with their own CPU and local memory.

Although a system interconnect allows all CPUs in a NUMA system to share one physical address space, a CPU has faster access to its local memory than to memory local to another CPU.

If the operating system's CPU scheduler and memory-placement algorithms are NUMA-aware and work together, then a thread that has been scheduled onto a part emory closest to where the CPU resides, tl stest possible memory access.

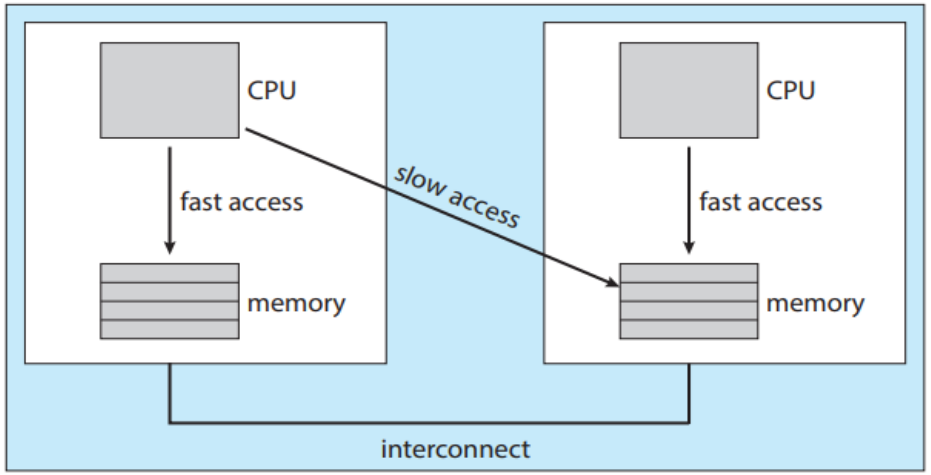


Figure 5.16 NUMA and CPU scheduling.

Heterogeneous Multiprocessing systems

The systems are now designed using cores that run the same instruction set, yet vary in terms of their clock speed and power management, including the ability to adjust the power consumption of a core to the point of idling the core. Such systems are known as heterogeneous multiprocessing (HMP).

The intention behind HMP is to better manage power consumption by assigning tasks to certain cores based upon the specific demands of the task.

For ARM processors that support it, this type of architecture is known as big. LITTLE where higher-peformance big cores are combined with energy efficient LITTLE cores.

Big cores consume greater energy and therefore should only be used for short periods of time. Likewise, little cores use less energy and can therefore be used for longer periods.

There are several advantages to this approach. By combining a number of slower cores with faster ones, a CPU scheduler can assign tasks that do not require high performance, but may need to run for longer periods, (such as background tasks) to little cores, thereby helping to preserve a battery charge. Similarly, interactive applications which require more processing power, but may run for shorter durations, can be assigned to big cores.

Deadlocks

A deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

A system consists of a finite number of resources to be distributed among a number of processes.

A process must request a resource before using it and must release the resource after using it.

A process may request as many resources as it requires to carry out its designated task.

Obviously, the number of resources requested may not exceed the total number of resources available in the system.

A process in operating system uses resources in the following way.:

1. Request a resource: The process requests the resource. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. Use the resource: The process can operate on the resource
3. Release the resource: The process releases the resource.

A system table records whether each resource is free or allocated.

For each resource that is allocated, the table also records the process to which it is allocated.

If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

Deadlock Characterization

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. Hold and wait: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. No preemption: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. Circular wait: A set {T0, T1, ..., Tn} of waiting threads must exist such that T0 is waiting for a resource held by T1, T1 is waiting for a resource held by T2, ..., Tn−1 is waiting for a resource held by Tn, and Tn is waiting for a resource held by T0.

Resource Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph.

This graph consists of a set of vertices V and a set of edges E.

The set of vertices V is partitioned into two different types of nodes: T = {T1, T2, ..., Tn}, the set consisting of all the active threads in the system, and R = {R1, R2, ..., Rm}, the set consisting of all resource types in the system.

A directed edge from thread Ti to resource type Rj is denoted by Ti → Rj; it signifies that thread Ti has requested an instance of resource type Rj and is currently waiting for that resource.

A directed edge from resource type Rj to thread Ti is denoted by Rj → Ti ; it signifies that an instance of resource type Rj has been allocated to thread Ti .

A directed edge Ti → Rj is called a request edge; a directed edge Rj → Ti is called an assignment edge.

Each thread Ti is represented as a circle and each resource type Rj as a rectangle.

Resource Allocation Graph

When thread Ti requests an instance of resource type Rj , a request edge is inserted in the resource-allocation graph.

When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge.

When the thread no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure 8.4 depicts the following situation

- The sets T, R, and E:
  - T = {T1, T2, T3}
  - R = {R1, R2, R3, R4}
  - E = {T1 → R1, T2 → R3, R1 → T2, R2 → T2, R2 → T1, R3 → T3}
- Resource instances:
  - One instance of resource type R1
  - Two instances of resource type R2
  - One instance of resource type R3
  - Three instances of resource type R4
- Thread states:
  - Thread T1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
  - Thread T2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
  - Thread T3 is holding an instance of R3

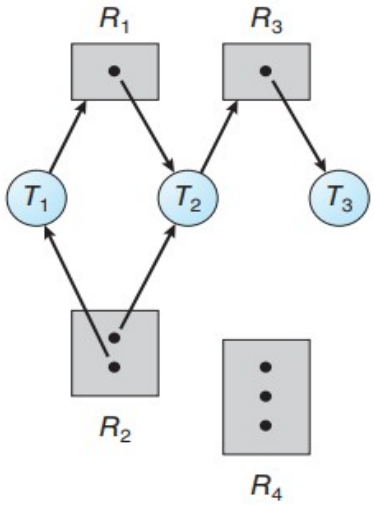


Figure 8.4 Resource-allocation graph.

Resource Allocation Graph

- It can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Resource Allocation Graph

To illustrate this concept, consider the resource-allocation graph depicted in Figure 8.4. Suppose that thread T3 requests an instance of resource type R2.

Since no resource instance is currently available, we add a request edge T3 → R2 to the graph (Figure 8.5).

At this point, two minimal cycles exist in the system:

T1 → R1 → T2 → R3 → T3 → R2 → T1

T2 → R3 → T3 → R2 → T2

Threads T1, T2, and T3 are deadlocked.

Thread T2 is waiting for the resource R3, which is held by thread T3.

Thread T3 is waiting for either thread T1 or thread T2 to r

In addition, thread T thread T2 to release r

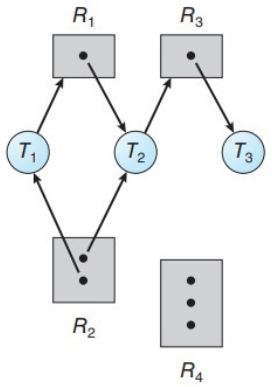


Figure 8.4 Resource-allocation graph.

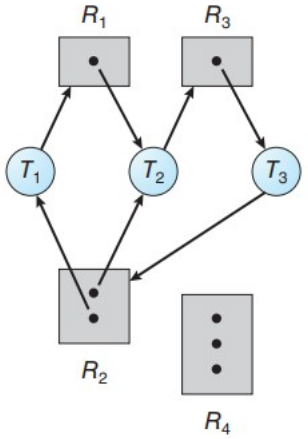


Figure 8.5 Resource-allocation graph with a deadlock.

Resource Allocation Graph

Now consider the resource-allocation graph in Figure 8.6.

In this example, we also have a cycle: T1 → R1 → T3 → R2 → T1

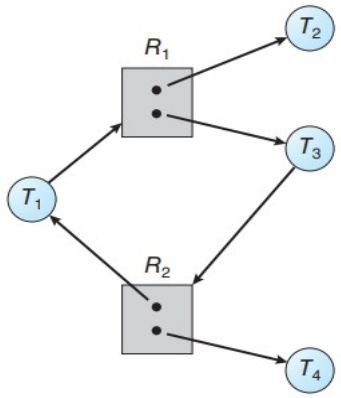
However, there is no deadlock.

Observe that thread T4 may release its instance of resource type R2.

That resource can then be allocated to T3, breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state.

If there is a cycle, then the system may or may not be in a deadlocked state.



ure 8.6 Resource-allocation graph with a cycle but no deadlock.

Methods for Handling Deadlock

- Deadlock prevention
- Deadlock avoidance
- Deadlock Recovery

Deadlock Prevention

Mutual Exclusion:

The mutual-exclusion condition must hold. That is, at least one resource must be non-sharable.

Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock.

Read-only files are a good example of a sharable resource.

If several threads attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.

A thread never needs to wait for a sharable resource.

In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

For example, a mutex lock cannot be simultaneously shared by several threads.

Deadlock Prevention

Hold and Wait:

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a thread requests a resource, it does not hold any other resources.

One protocol that we can use requires each thread to request and be allocated all its resources before it begins execution. This is impractical for most applications due to the dynamic nature of requesting resources.

An alternative protocol allows a thread to request resources only when it has none. A thread may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Both these protocols have two main disadvantages.

First, resource utilization may be low, since resources may be allocated but unused for a long period. For example, a thread may be allocated a mutex lock for its entire execution, yet only require it for a short duration.

Second, starvation is possible. A thread that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other thread.

Deadlock Prevention

No Preemption:

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a thread is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the thread must wait), then all resources the thread is currently holding are preempted. In other words, these resources are implicitly released.

The preempted resources are added to the list of resources for which the thread is waiting. The thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a thread requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other thread that is waiting for additional resources. If so, we preempt the desired resources from the waiting thread and allocate them to the requesting thread.

If the resources are neither available nor held by a waiting thread, the requesting thread must wait. While it is waiting, some of its resources may be preempted, but only if another thread requests them.

A thread can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

Deadlock Prevention

Circular Wait:

The circular-wait condition presents a practical solution by invalidating one of the necessary conditions.

One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each thread requests resources in an increasing order of enumeration.

To illustrate, we let R = {R1, R2, ..., Rm} be the set of resource types.

We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.

Formally, we define a one-to-one function F: R → N, where N is the set of natural numbers.

We can accomplish this scheme in an application program by developing an ordering among all synchronization objects in the system.

Deadlock Avoidance

A method for avoiding deadlocks is to require additional information about how resources are to be requested.

For example, in a system with resources R1 and R2, the system need to know that process P will request first R1 and then R2 before releasing both resources, whereas process Q will request R2 and then R1.

With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.

In making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

The various algorithms that use this approach differ in the amount and type of information required.

The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.

Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence.

A sequence of processes is a safe sequence for the current allocation state if, for each Ti , the resource requests that Ti can still make can be satisfied by the currently available resources plus the resources held by all Tj , with j < i.

In this situation, if the resources that Ti needs are not immediately available, then Ti can wait until all Tj have finished.

When they have finished, Ti can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate.

When Ti terminates, Ti+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

Safe State

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks.

An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.

In an unsafe state, the operating system cannot prevent threads from requesting resources in such a way that a deadlock occurs.

The behavior of the threads controls unsafe states.

Safe State

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks.

An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.

In an unsafe state, the operating system cannot prevent threads from requesting resources in such a way that a deadlock occurs.

The behavior of the threads controls unsafe states.

Bankers Algorithm

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.

This number may not exceed the total number of resources in the system.

When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.

If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Bankers Algorithm

The following data structures are required, where n is the number of process in the system and m is the number of resource types:

- Available: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Rj are available.
- Max: An n × m matrix defines the maximum demand of each thread. If Max[i][j] equals k, then thread Ti may request at most k instances of resource type Rj .
- Allocation: An n × m matrix defines the number of resources of each type currently allocated to each thread. If Allocation[i][j] equals k, then thread Ti is currently allocated k instances of resource type Rj .
- Need: An n × m matrix indicates the remaining resource need of each thread. If Need[i][j] equals k, then thread Ti may need k more instances of resource type Rj to complete its task. Note that Need[i][j] equals Max[i][j] − Allocation[i][j].

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available and Finish[i] = false for i = 0, 1, ..., n − 1. 2.
2. Find an index i such that both
    a. Finish[i] == false
    b. Needi ≤ Work If no such i exists, go to step 4.
3. Work = Work + Allocationi Finish[i] = true Go to step 2.
4. If Finish[i] == true for all i, then the system is in a safe state.

Resource - Request Algorithm

The algorithm for determining whether requests can be safely granted. Let Requesti be the request vector for thread Ti . If Requesti [j] == k, then thread Ti wants k instances of resource type Rj . When a request for resources is made by thread Ti , the following actions are taken:

1. If Requesti ≤ Needi , go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.
2. If Requesti ≤ Available, go to step 3. Otherwise, Ti must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread Ti by modifying the state as follows:

    Available = Available–Request[i]

    Allocation[i] = Allocation[i] + Request[i]

    Need[i] = Need[i] –Request[i]

If the resulting resource-allocation state is safe, the transaction is completed and thread Ti

Example- Bankers Algorithm

Consider a system with five threads T0 through T4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that the following snapshot represents the current state of the system:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 1 1	2 2 2	
T_4	0 0 2	4 3 3	

1. Compute the contents of Need matrix.
2. Is the system in a safe state? **If Yes, then what is the safe sequence?**
3. Determine the total amount of resources of each type?
4. What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C? - (1,0,2)

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.

In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

The two requirements that pertain to systems with only a single instance of each resource type and systems with several instances of each resource type.

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.

This graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

An edge from Ti to Tj in a wait-for graph implies that thread Ti is waiting for thread Tj to release a resource that Ti needs.

An edge Ti → Tj exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges Ti → Rq a . In Figure 8.11, a resource-allocation graph and the corresponding

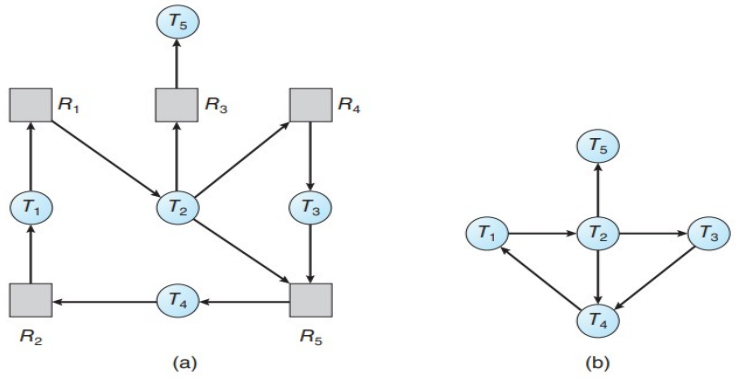


Figure 8.11 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle, and it is a necessary and sufficient conditions for deadlock. To detect deadlocks, the system needs to maintain

Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

The deadlock detection algorithm is applicable to such a system and this algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.

- Available: A vector of length m indicates the number of available resources of each type.
- Allocation: An n × m matrix defines the number of resources of each type currently allocated to each thread.
- Request: An n × m matrix indicates the current request of each thread. If Request[i][j] equals k, then thread Ti is requesting k more

Several Instances of a Resource Type

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available. For i = 0, 1, ..., n–1, if Allocationi ≠ 0, then Finish[i] = false. Otherwise, Finish[i] = true.
2. Find an index i such that both
    a. Finish[i] == false
    b. Requesti ≤ Work If no such i exists, go to step 4.
3. Work = Work + Allocationi Finish[i] = true Go to step 2.
4. If Finish[i] == false for some i, 0 ≤ i < n, then the system is in a deadlocked state. Moreover, if Finish[i] == false, then thread Ti is deadlocked.

This algorithm requires an order of m × n2 operations to detect whether

Several Instances of a Resource Type

Consider a system with five threads T0 through T4 and three resource types A, B, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. The following snapshot represents the current state of the system:

	Allocation	*Request*	*Available*
	A B C	A B C	A B C
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. The sequence results in <T0, T2, T3, T1, T4>. Suppose now ead T2 makes one additional request for an instance of type C. The Request

	Request
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

modified as follows

We claim that the system is now deadlocked. The number of available resources is not

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available.

- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically.

There are two options for breaking a deadlock.

- One is simply to abort one or more threads to break the circular wait.
- The other is to preempt some resources from one or more of the deadlocked threads.

Process and Thread Termination

To eliminate deadlocks by aborting a process or thread, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- Abort all deadlocked processes: This method will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Process and Thread Termination

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it may leave that file in an incorrect state.

Similarly, if the process was in the midst of updating shared data while holding a mutex lock, the system must restore the status of the lock as being available, although no guarantees can be made regarding the integrity of the shared data.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated.

This determination is a policy decision, similar to CPU-scheduling decisions. Those processes should be terminated whose termination will incur the minimum cost.

Process and Thread Termination

Many factors may affect which process is chosen, includes:

1. What the priority of the process is?
2. How long the process has computed and how much longer the process will compute before completing its designated task ?
3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt) ?
4. How many more resources the process needs in order to complete ?
5. How many processes will need to be terminated?

Resource Preemption

To eliminate deadlocks using resource preemption, preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1.Selecting a victim: Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.

2. Rollback: If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes