Module III
OPERATING SYSTEM

CPU SCHEDULING

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

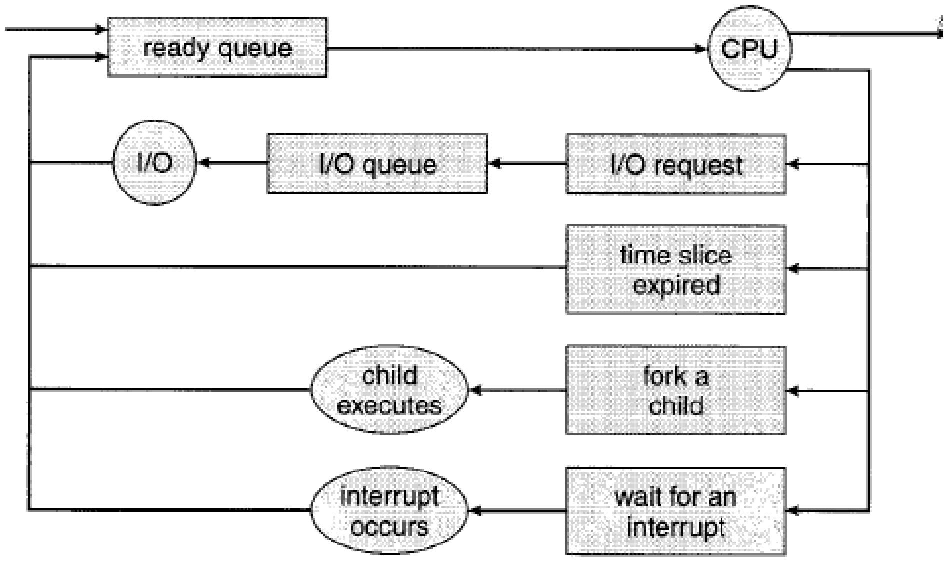


Figure: Queuing-diagram representation of process scheduling

Dispatcher

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

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

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 I/O request or invocation of wait for the termination of one of the child processes).
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, completion of I/O).
4. When a process terminates.

In circumstances 1 and 4, there is no choice in terms of scheduling. A new process(if one exists in the ready queue) must be selected for execution. There is a choice, however in circumstances 2 and 3.

When Scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is non-preemptive; otherwise the scheduling scheme is preemptive.

Non-Preemptive Scheduling

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems It is the only method that can be used on certain hardware platforms, because It does not require the special hardware(for example: a timer) needed for preemptive scheduling.

When a process switches from the running state to the waiting state (for example, I/O request, or invocation of wait for the termination of one of the child processes) or when a process terminates is called non-preemptive scheduling.

Preemptive Scheduling

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

When a process switches from the running state to the ready state (for example, when an interrupt occurs) or when a process switches from the waiting state to the ready state (for example, completion of I/O) is called preemptive scheduling.

Preemptive	Non-preemptive
When a process switches from running state to ready state (interrupts)	When a process switches from running state to waiting state(I/O request)
When a process switches waiting state to ready state (completion of I/O)	When a process terminate

SCHEDULING CRITERIA

- **CPU Utilization**:- we wants to keep the CPU as busy as possible. Its range from 0 to 100 percent. In real time system it is 40 percent (lightly loaded system) to 90 percent (for a heavily used system)
- **Throughput**:- the number of process completed per time unit. For a long process this rate may be one process per hour, for short transactions it may be 10 processes per second.
- **Turn around time**:- It is the interval from the time of submission of a process to the time of completion. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time**:- the amount of time that a process spends waiting in the ready queue. It is the sum of the periods spends waiting in the ready queue.
- **Response time**:- It is the time between the submission of the process and will produce the first output (to start responding).

Note:-We want to maximize CPU utilization and throughput and to minimize turnaround time, waiting time and response time.

SCHEDULING ALGORITHMS

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. Scheduling of processes is done to finish the work on time. Below are different times with respect to a process.

Arrival Time:- Time at which the process arrives in the ready queue.
Completion Time:- Time at which process completes its execution.
Burst Time:-Time required by a process for CPU execution.
Turn Around Time:-Time Difference between completion time and arrival time.

Turn Around Time = Completion Time - Arrival Time

Waiting Time(W.T):-Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time - Burst Time

There are six popular process scheduling algorithms

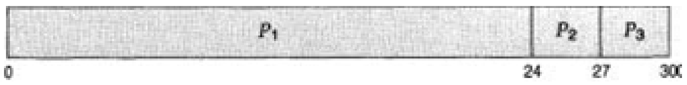
1. First Come First Serve(FCFS) Scheduling
2. Shortest-Job-First(SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

FIRST COME FIRST SERVE(FCFS) SCHEDULING

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

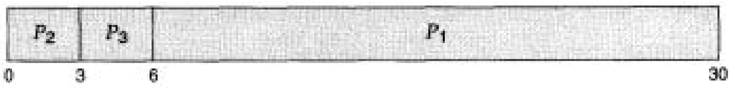
Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

Process: P1, P2, P3
Burst Time: 24, 3, 3



If the processes arrive in the order PI, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart:

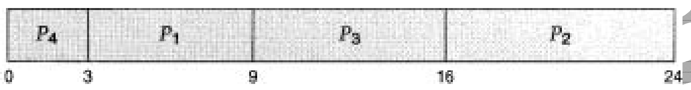
The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2, and 27 milliseconds for process P3. Thus, the average waiting time is (0 + 24 +27) / 3 = 17 milliseconds. If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:
The average waiting time is now (6 + 0 + 3)/3 That is 3 milliseconds.



SHORTEST-JOB-FIRST(SJF) SCHEDULING

This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it 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.

Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:
Process: P1, P2, P3, P4
Burst Times: 6, 7, 8, 3
Using SJF scheduling, we get the following Gantt chart:

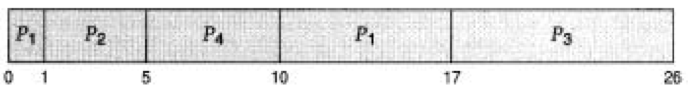


The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. If we were using the FCFS scheduling scheme, then the average waiting time would be 10.25 milliseconds. The SJF algorithm may be either preemptive or non-preemptive.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. By moving a short process before a long one, the waiting time of the short process decreases more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

Shortest-Remaining-Time-First scheduling (SRTF)

The Preemptive SJF scheduling is called Shortest-Remaining-Time-First scheduling. Consider the following four processes, with the length of the CPU-burst time given in milliseconds:
Process: P1, P2, P3, P4
Arrival Time: 0, 1, 2, 3
Burst Time: 8, 4, 9, 5



If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the above Gantt chart.

Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.

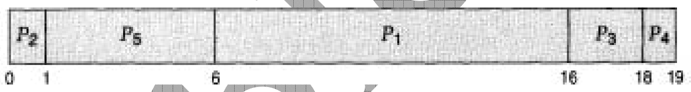
The average waiting time for this example is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5 milliseconds. A non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds

PRIORITY SCHEDULING

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa. Priorities are generally some fixed range of numbers, such as 0 to 7, or 0 to 4,095. We use low numbers to represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order PI, P2, ..., Ps, with the length of the CPU-burst time given in milliseconds:

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



Using priority scheduling, we would schedule these processes according to the above Gantt chart:
The average waiting time is 8.2 milliseconds.

A major problem with priority-scheduling algorithms is indefinite blocking (or starvation). A process that is ready to run but lacking the CPU can be considered blocked-waiting for the CPU. A priority-scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU.

A solution to the problem of indefinite blockage of low-priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could decrement the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority 127 process to age to priority 0 processes.

ROUND ROBIN(RR) SCHEDULING

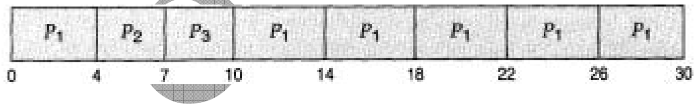
The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum (or time slice), is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

Process: P1, P2, P3

Burst Time: 24, 3, 3

If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2. Since process P2 does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is



The average waiting time is 17/3 = 5.66 milliseconds. In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row. If a process' CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is preemptive.

MULTILEVEL QUEUE SCHEDULING

A multilevel queue-scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example: The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm. An example of a multilevel queue-scheduling algorithm with five queues:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

MULTILEVEL FEEDBACK QUEUE SCHEDULING

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

A multilevel feedback queue scheduler is defined by the following parameters:

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

Multiple Processor Scheduling

If several identical processors are available, then load sharing can occur. It would be possible to provide a separate queue for each processor. In this case, one processor could be idle, with an empty queue, while another processor was very busy. To prevent this situation, we use a common ready queue. All processes go into one queue and are scheduled onto any available processor. In such a scheme, one of two scheduling approaches may be used. In one approach, each processor is self-scheduling.

Real-Time Scheduling

This scheduling facility needed to support real-time computing within a general-purpose computer system. Real-time computing is divided into two types.

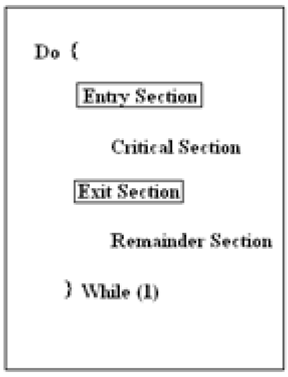
The **Hard real-time** systems are required to complete a critical task within a guaranteed amount of time. Generally, a process is submitted along with a statement of the amount of time in which it needs to complete or perform I/O. The scheduler then either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible. This is known as resource reservation.

The **Soft real-time** computing is less restrictive. It requires that critical processes receive priority over less fortunate ones. Although adding soft real-time functionality to a time-sharing system may cause an unfair allocation of resources and may result in longer delays, or even starvation, for some processes, it is at least possible to achieve.

PROCESS SYNCHRONIZATION

A situation, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable counter. To make this, we require some form of synchronization of the processes.

CRITICAL SECTION PROBLEM



```
Do {
      Entry Section
            Critical Section
      Exit Section
            Remainder Section
   } While (1)
```

Consider a system consisting of n processes {P0, P1, ..., Pn-1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. When one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. A picture (right side) showing general structure of a process Pi.

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

- **Mutual Exclusion**: If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress**: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
- **Bounded Waiting**: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

SYNCHRONIZATION HARDWARE

Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.

Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors. This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.

Mutex Locks

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.

As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released. As the resource is locked while a process executes its critical section hence no other process can access it.

- A lock is one form of hardware support for mutual exclusion.
- If a shared resource has a locked hardware lock, it is already in use by another process.
- If it is not locked, a process may freely
- Lock it for itself;
- Use it;
- Unlock it when it finishes.

Problem: Race conditions

Test-and-Set:

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

Is a hardware implementation for testing for the lock and resetting it to locked.

If test shows unlocked, the process may proceed.

- Acts as an atomic operation
- Permits
- Busy waits
- Spinning
- Spinlocks

Atomic Swap:

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.

Performs three operations atomically

- Swap current lock value with temp locked lock
- Examine new value of temp lock
- If locked, repeat
- If unlocked, proceed into critical section/region

Utilizes a temporary variable

CLASSICAL PROBLEMS OF SYNCHRONIZATION

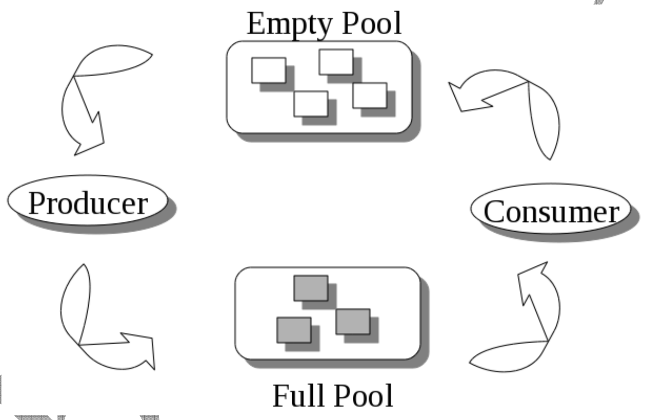
Some synchronization problems, such as the Shared Buffer Problem are fairly simple to solve using reliable synchronization facilities such as Dijkstra's Semaphore. Other problems that programmers must solve are more complex, in fact, some of them are just down right tricky to solve correctly such that there is never a problem with Starvation, Deadlock or Data Inconsistency. Fortunately, there are a lot of really smart people who have already tackled these problems and we have known solutions to them. These problems are ones that frequently arise, so we discuss them and their solutions in fairly generic terms and refer to them as the Classic Synchronization Problems. One of the biggest challenges that the programmer must solve is to correctly identify their problem as an instance of one of the classic problems. It may require thinking about the problem or framing it in a less than obvious way, so that a known solution may be used. The advantage to using a using known solution is assurance that it is correct. Some instances of deadlock, can occur only in rare conditions and are difficult to detect. So for any programming task using synchronization, it is best to reference a known solution to one of the classic problems.

1. Bounded-Buffer Problem
2. Readers and Writers Problem
3. Dining-Philosophers Problem

Bounded Buffer (Producers and Consumers)

In this problem, two processes, one called the producer and the other called the consumer, run concurrently and share a common buffer. The producer generates items that it must pass to the consumer, who is to consume them. The producer passes items to the consumer through the buffer. However, the producer must be certain that it does not deposit an item into the buffer when the buffer is full, and the consumer must not extract an item from an empty buffer. The two processes also must not access the buffer at the same time, for if the consumer tries to extract an item from the slot into which the producer is depositing an item, the consumer might get only part of the item. Any solution to this problem must ensure none of the above three events occur.

A practical example of this problem is electronic mail. The process you use to send the mail must not insert the letter into a full mailbox (otherwise the recipient will never see it); similarly, the recipient must not read a letter from an empty mailbox (or he might obtain something meaningless but that looks like a letter). Also, the sender (producer) must not deposit a letter in the mailbox at the same time the recipient extracts a letter from the mailbox; otherwise, the state of the mailbox will be uncertain.



Because the buffer has a maximum size, this problem is often called the bounded buffer problem. A (less common) variant of it is the unbounded buffer problem, which assumes the buffer is infinite. This eliminates the problem of the producer having to worry about the buffer filling up, but the other two concerns must be dealt with. Producers produce a product and consumers consume the product, but both use of one of the containers each time.

Readers and Writers Problem

In this problem, a number of concurrent processes require access to some object (such as a file.) Some processes extract information from the object and are called readers; others change or insert information in the object and are called writers. The Bernstein conditions state that many readers may access the object concurrently, but if a writer is accessing the object, no other processes (readers or writers) may access the object. There are two possible policies for doing this:

1. Readers have priority over writers; that is, unless a writer has permission to access the object, any reader requesting access to the object will get it. Note this may result in a writer waiting indefinitely to access the object.

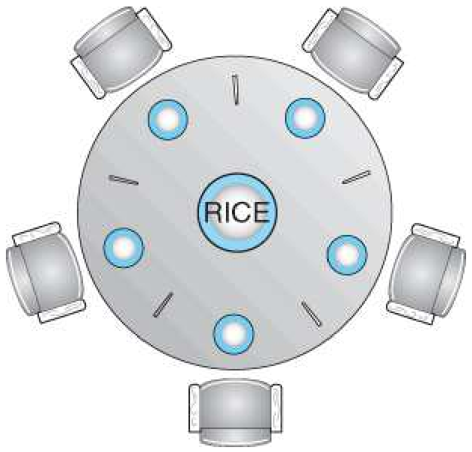
2. Writers have priority over readers; that is, when a writer wishes to access the object, only readers which have already obtained permission to access the object are allowed to complete their access; any readers that request access after the writer has done

so must wait until the writer is done. Note this may result in readers waiting indefinitely to access the object.

So there are two concerns: first, enforce the Bernstein conditions among the processes, and secondly, enforcing the appropriate policy of whether the readers or the writers have priority. A typical example of this occurs with databases, when several processes are accessing data; some will want only to read the data, others to change it. The database must implement some mechanism that solves the readers-writers problem. Writers have mutual exclusion, but multiple readers at the same time is allowed.

Dining-Philosophers Problem

In this problem, five philosophers sit around a circular table eating spaghetti and thinking. In front of each philosopher is a plate and to the left of each plate is a fork (so there are five forks, one to the right and one to the left of each philosopher's plate; see the figure). When a philosopher wishes to eat, he picks up the forks to the right and to the left of his plate. When done, he puts both forks back on the table. The problem is to ensure that no philosopher will be allowed to starve because he cannot ever pick up both forks.



There are two issues here: first, deadlock (where each philosopher picks up one fork so none can get the second) must never occur; and second, no set of philosophers should be able to act to prevent another philosopher from ever eating. A solution must prevent both.

FILE AND DATABASE SYSTEM, FILE SYSTEM, FUNCTIONS OF ORGANIZATION, ALLOCATION AND FREE SPACE MANAGEMENT.