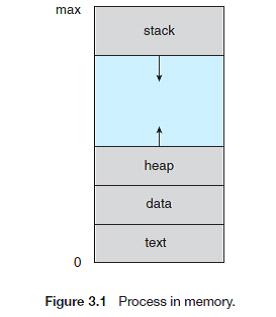
**2.1 Process Concepts**

**2.1.1 Definition of Process:**

A process is a program in execution. It also includes the current activity as represented by the value of the

“program counter” the contents of the processes register.

A process generally includes “stack” which contain temporary data and a “data section” which contains global variables. It contains “heap” which is memory that is dynamically allocated during process runtime.

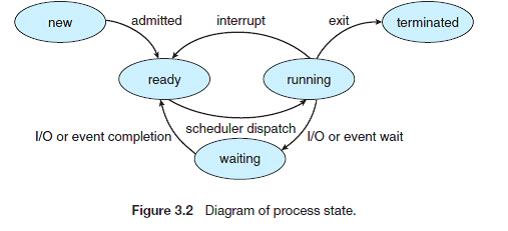


* A process is more than the program code, which is sometimes known as the **text section.**
* It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers.
* A process generally also includes the process **stack,** which contains temporary data (such as function
* parameters, return addresses, and local variables), and a **data section,** which contains global variables.
* A process may also include a **heap,** which is memory that is dynamically allocated during process run time

**2.1.2 Process States:**

As a process executes, it changes *state*

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

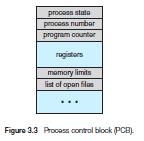


**2.1.3 Process Control Block:**

Each process is represented in the operating system by a **process control block (PCB)**—also called a *task* *control block.*

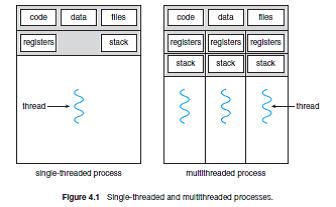
It includes:

* **Process state.** The state may be new, ready, running, waiting, halted, and so on.
* **Program counter.** The counter indicates the address of the next instruction to be executed forthis process.
* **CPU registers.** The registers vary in number and type, depending on the computer architecture.They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward
* **CPU-scheduling information.** This information includes a process priority, pointers toscheduling queues, and any other scheduling parameters.
* **Memory-management information.** This information may include such information as thevalue of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
* **Accounting information.** This information includes the amount of CPU and real time used, timelimits, account members, job or process numbers, and so on.
* **I/O status information.** This information includes the list of I/O devices allocated to the process,a list of open files, and so on.



**2.2 Thread:**

A thread is a flow of execution through the process code, with its own program counter, system registers and stack. A thread is also called a light weight process.



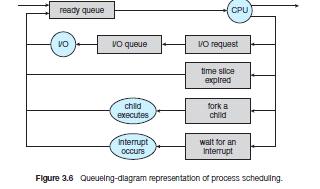
**2.3 Process Scheduling:**

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

**2.3.1 Scheduling Queues:**

* As processes enter the system, they are put into a **job queue,** which consists of all processes in the system.
* The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.** This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

27



**2.3.2 Schedulers:**

* **Long-term scheduler** (or job scheduler) - selects which processes should be brought into theready queue.
* **Short-term scheduler** (or CPU scheduler) - selects which process should be executed next andallocates CPU.
* **Medium term scheduler** - Medium term scheduling is part of the swapping. It removes theprocesses from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.

**2.3.3 Context Switch:**

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch.** When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

**2.3.5 Dispatcher:**

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

This function involves the following:

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

**2.3.6 Scheduling Criteria:**

In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

* **CPU utilization.** We want to keep the CPU as busy as possible. Conceptually, CPU utilization canrange 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 used system).
* **Throughput.** If the CPU is busy executing processes, then work is being done. One measure of work isthe number of processes that are completed per time unit, called *throughput.* For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.
* **Turnaround time.** From the point of view of a particular process, the important criterion is how long ittakes to execute that process. The interval from the time of submission of a process to the time of completion is the *turnaround time.* Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
* **Waiting time.** The CPU scheduling algorithm does not affect the amount of time during which aprocess 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.** In an interactive system, turnaround time may not be the best criterion. Often, a processcan produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called *response time,* is the time it takes to start responding, not the

29

time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

**2.3.7 Scheduling Algorithms:**

There are many different CPU scheduling algorithms.

* ***First Come First Serve Scheduling***:

The process that requests the CPU first is allocated the CPU first. The implementation of FCFS is early managed with the FIFO queue.

* ***Shortest Job First Scheduling:***

The algorithm associates each process the length of the processes next CPU burst.

* ***Priority Scheduling***:

A priority is associated with each process at and CPU is allocated to the process with the highest priority. The layer CPU burst, the lower the priority and vice versa.

* ***Round Robin Scheduling***:

It is designed for time sharing system. It is similar to FCFS Scheduling but preemption is added to switch between processes.

* ***Multi -level queue Scheduling***:

At partition the ready queue into several separate queues. The processes are permanently assigned to one queue generally based on some property or process type.

* ***Multi- level feedback queue Scheduling:***

It allows a process to move between queues. The idea is to separate process according to the characteristics of their CPU bursts.

**Examples:**

* **First Come First Serve Scheduling**:

|  |  |
| --- | --- |
| **Process** | **Burst Time** |
| *P1* | 24 |
| *P2* | 3 |
| *P3* | 3 |

Suppose that the processes arrive in the order: *P1* , *P2* , *P3*

**The Gantt Chart for the schedule is:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | P1 |  | P2 |  |  | P3 |  |
|  |  |  |  |  |  |  |  |
| 0 | | 24 | | 27 | | 30 | |

• Waiting time for *P1* = 0; *P2* = 24; *P3* = 27

* + Average waiting time: (0 + 24 + 27)/3 = 17
* **Shortest Job First Scheduling:**

|  |  |  |
| --- | --- | --- |
| Process | Arrival Time | Burst Time |
| *P1* | 0.0 | 6 |
| *P2* | 2.0 | 8 |
| *P3* | 4.0 | 7 |
| *P4* | 5.0 | 3 |

**The Gantt Chart for the schedule is:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | P4 |  | P1 |  | P3 |  |  | P2 |  |
|  |  |  |  |  |  |  |  |  |  |
| 0 | | 3 | | 9 | | 16 | | 24 | |

* + Average waiting time = (3 + 16 + 9 + 0) / 4 = 7
* **Priority Scheduling**:

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

Priority scheduling Gantt Chart

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | P1 |  | P2 |  | P1 |  | P3 | P4 | |  |
| 0 | 1 | | 6 | | 16 | | 18 | | 19 | |

Average waiting time = 8.2 msec

* **Round Robin Scheduling:**

|  |  |
| --- | --- |
| Process | Burst Time |
| *P1* | 24 |
| *P2* | 3 |
| *P3* | 3 |

32

Time quantum=4

The Gantt chart is:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | P1 |  | P2 |  | P3 |  | P1 |  | P1 |  |  | P1 |  | P1 |  | P1 |  |
| 0 | | 4 | | 7 | | 10 | | 14 | | 18 | | 22 | | 26 | | 30 | |

* Typically, higher average turnaround than SJF, but better *response*
* q should be large compared to context switch time
* q usually 10ms to 100ms, context switch < 10 usec

**Process Synchronization:**

**Race condition:**

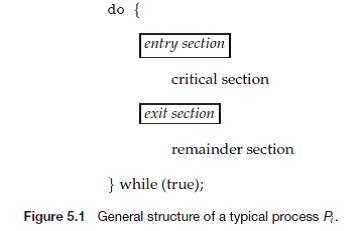
The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

**2.6 Critical Section Problem**:

Consider a system consisting of *n* processes {PQ, PI, ..., *P,,~\}.* 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.

Each process must request permission to enter its critical

section. The section of code implementing this request is the **entry section.** The critical section may be followed by an **exit section.** The remaining code is the **remainder section.**



The critical section problem is to design a protocol that the processes can use to cooperate. The solution to the critical section problems must satisfy the requirements

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

**2.7 Peterson’s Solution:**

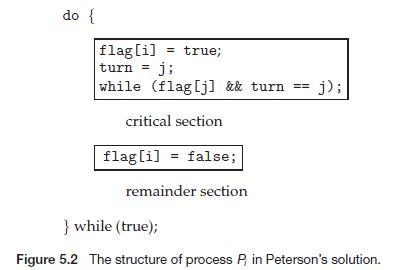
A classic software based solution to the critical section problem known as Peterson’s solution. The solution provides a good algorithmic description of solving the critical section problem and illustrates some of the complexities involved in designing software that address the requirements of mutual exclusion, progress and bounded writing requirements.

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



34

Peterson's solution requires two data items to be shared between the two processes:



The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process P; is allowed to execute in its critical section. The flag array is used to indicate if a process *is ready* to enter its critical section. For example, if f lag[i] is true, this value indicates that P; is ready to enter its critical section.

To enter the critical section, process P, first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

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

35

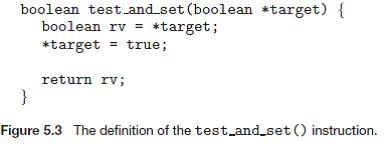
To prove properties 2 and 3, we note that a process P, can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] == true and turn == j; this loop is the only one possible. If P; is not ready to enter the critical section, then flag [j] == false,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

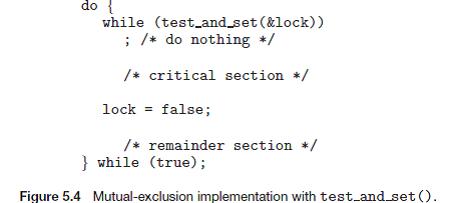
**2.8 Synchronization hardware:**

Hardware features can make any programming task easier and improve system efficiency. In this section, we present some simple hardware instructions that are available on many systems and show how they can be used effectively in solving the critical section problem.

In general, we can state that any solution to the critical-section problem requires a simple tool—a lock. Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section Hardware features can make any programming task easier and improve system efficiency. In this section, we present some simple hardware instructions that are available on many systems and show how they can be used effectively in solving the critical-section problem.

The important characteristic of TestAndSet() instruction is that this instruction is executed atomically. Thus, if two TestAndSet C) instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false.



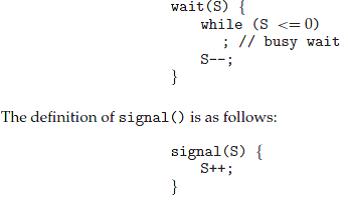


**2.9 Semaphores:**

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal ().

The wait() operation was originally termed P (from the Dutch *probercn,* "to test"); signal () was originally called V (from *verhogen,* "to increment").

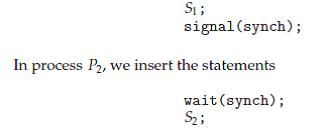
The definition of wait() is as follows:



Operating systems often distinguish between counting and binary semaphores. The value of a **counting** **semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range onlybetween 0 and 1. On some systems, binary semaphores are known as **mutex locks,** as they are locks that provide *mutual* t'.rclusion.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: *P\* with a statement Si and *Pi* with a statement *Si.* Suppose we require that *So* be executed only after Si has completed. We can implement this scheme readily by letting Pi and

*Pi* share a common semaphore synch, initialized to 0, and by inserting the statements



in process P2. Because synch is initialized to 0, P2 will execute S2 only after *P\* has invoked signal (synch), which is after statement Si has been executed.

**Implementation:**

The main disadvantage of the semaphore definition given here is that it requires **busy waiting.** While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** because the process "spins" while waiting for the lock. (Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful; they are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.)

To overcome the need for busy waiting, we can modify the definition of the wait () and signal () semaphore operations. When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can *block* itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as a "C" struct:

typedef struct { int value;

struct process \*list; } semaphore;

Each semaphore has an integer value and a list of processes l i s t . When a process must wait on a semaphore, it is added to the list of processes. A signal () operation removes one process from the list of waiting processes and awakens that process.

The wait () semaphore operation can now be defined as

wait(semaphore \*S) { S->value—;

if (S->value < 0) {

add this process to S->list;

block();

}

}

The signal 0 semaphore operation can now be defined as # signal(semaphore \*S) {

S->value++;

if (S->value <= 0) { remove a process *P* from S->list;

wakeup(P);

}

}

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

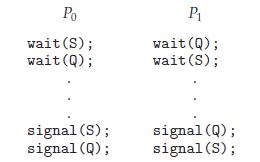
Note that, although under the classical definition of semaphores with busy waiting the semaphore value is never negative, this implementation may have negative semaphore values. If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the waitO operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list in a way that ensures bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use *any* queuing strategy. Correct usage of semaphores does not depend on a particular queuing strategy for the semaphore lists.

**Deadlocks and Starvation**

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked.**

To illustrate this, we consider a system consisting of two processes, PQ and Pi, each accessing two semaphores, S and Q, set to the value 1:



Suppose that *P$* executes wait (S) and then Pi executes wait (Q). When Po executes wait(Q), it must wait until Pi executes signal(Q). Similarly, when Pi executes wait(S), it must wait until Po executes signal(S). Since these signal () operations cannot be executed, Po and Pi are deadlocked.

We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are *resource acquisition and release.* However, other types of events may result in deadlocks Another problem related to deadlocks is **indefinite blocking,** or **starvation,** a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

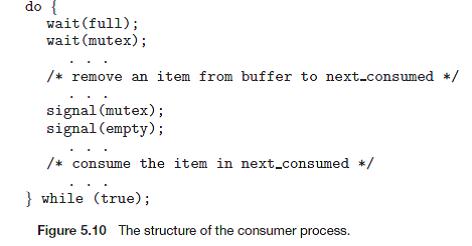
**2.10 Classical problems of Synchronization:**

We present a number of synchronization problems as example of a large class of concurrency control problems. These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we can use semaphores for synchronization.

**2.10.1 The Bounded-Buffer Problem:**

We assume that the pool consists of *n* buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and f u l l semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore f u l l is initialized to the value 0.

Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.



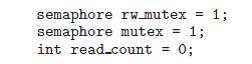
**2.10.2 The Readers-Writers Problem:**

A database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as **readers** and to the latter as writers. Obviously, if two readers access the shared data simultaneously, no adverse affects will result. However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue.

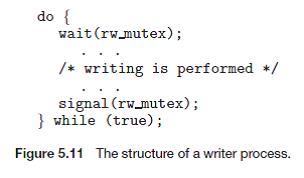
To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database. This synchronization problem is referred to as the *readers-writers problem.* Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The *second* readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we present a solution to the first readers-writers problem. Refer to the bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers-writers problem.

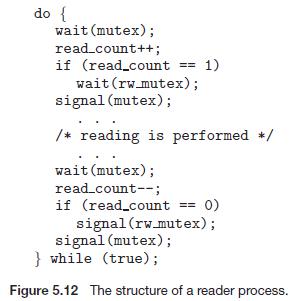
In the solution to the first readers-writers problem, the reader processes share the following data structures:



The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object.



If a writer is in the critical section and *n* readers are waiting, then one reader is queued on wrt, and n — 1 readers are queued on mutex. Also observe that, when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler. The readers-writers problem and its solutions has been generalized to provide **reader-writer** locks on some systems. Acquiring a reader-writer lock requires specifying the mode of the lock: either *read* or *write* access. When a process only wishes to read shared data, it requests the reader-wrriter lock in readmode; a process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader-writer lock in read mode; only one process may acquire the lock for writing as exclusive access is required for writers.



Reader-writer locks are most useful in the following situations:

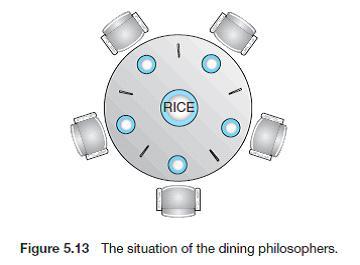
• In applications where it is easy to identify which processes only read shared data and which threads only write shared data.

• In applications that have more readers than writers. This is because reader writer locks generally require more overhead to establish than semaphores or mutual exclusion locks, and the overhead for setting up a reader-writer lock is compensated by the increased concurrency of allowing multiple readers.

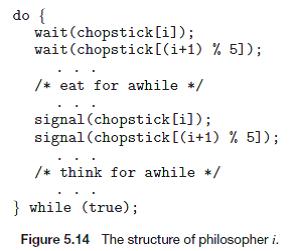
**2.10.3 The Dining-Philosophers Problem:**

The *dining-philosophers problem* is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are semaphore chopstick[5]; where all the elements of chopstick are initialized to 1.



Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0.

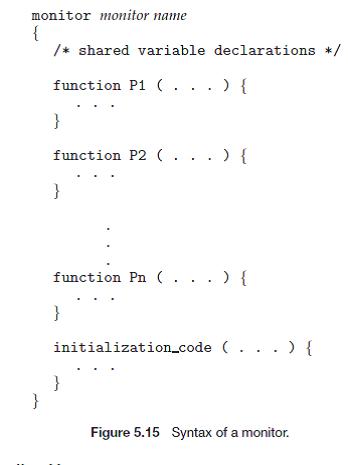


When each philosopher tries to grab her right chopstick, she will be delayed forever.

* + Several possible remedies to the deadlock problem are listed next.
  + Allow at most four philosophers to be sitting simultaneously at the table.
  + Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).
  + Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

1. **Monitors:**

The various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical section problem. Similar problems may arise in the other synchronization models. To deal with such errors, researchers have developed high level language constructs. In this section, we describe one fundamental high level synchronization construct the Monitor type. Solutions using monitors



* Usage
* Dining philosophers solutions using monitors
* Implementing a monitor using semaphores
* Resuming processes within a monitor

**Usage**

The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly. However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition:*

condition x, y;

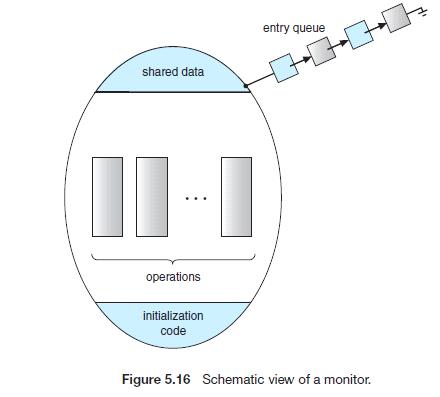
The only operations that can be invoked on a condition variable are wait () and signal(). The operation

x.waitO ;

means that the process invoking this operation is suspended until another process invokes

x . signal ( ) ;

The x. signal () operation resumes exactly one suspended process. If no process is suspended, then the signal () operation has no effect; that is, the state of x is the same as if the operation had never been executed. Contrast this operation with the signal () operation associated with semaphores, which always affects the state of the semaphore.



Now suppose that, when the x. **s** ignal () operation is invoked by a process P, there is a suspended process Q associated with condition x. Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. Note, however, that both processes can conceptually continue with their execution.

Two possibilities exist:

1. **Signal and wait.** *P*either waits until Q leaves the monitor or waits for another condition.
2. **Signal and continue.** *Q*either waits until*P*leaves the monitor or waits for another condition.

There are reasonable arguments in favor of adopting either option. On the one hand, since *P* was already executing in the monitor, the *signal-and-continue* method seems more reasonable. On the other hand, if we allow thread P to continue, then by the time *Q* is resumed, the logical condition for which *Q* was waiting may no longer hold. A compromise between these two choices was adopted in the language Concurrent Pascal. When thread P executes the signal operation, it immediately leaves the monitor. Hence, *Q* is immediately resumed.

**Dining philosophers solutions using monitors**

We now illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. For this purpose, we introduce the following data structure:

enum {thinking, hungry, eating} state [5] ;

Philosopher *i* can set the variable s t a t e [i] = eating only if her two neighbors are not eating: ( state [(i+4) °/» 5] != eating) and ( state [(i+1) % 5] != eating).

We also need to declare

condition self [5];

where philosopher can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor dp. Each philosopher, before starting to eat, must invoke the operation pi ckup (). This may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the

putdownO operation. Thus, philosopher *i* must invoke the operations pi ckup () and putdownO in the

following sequence:

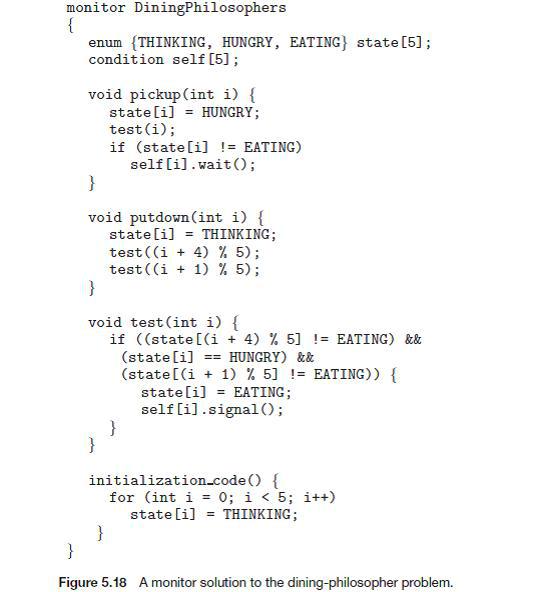
dp.pickup(i);

..

Eat

..

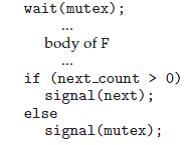
dp.putdown(i);



**Implementing a Monitor Using Semaphores**

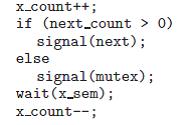
We now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a semaphore mutex (initialized to 1) is provided.

A process must execute wait (mutex) before entering the monitor and must execute signal (mutex) after leaving the monitor. Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0, on which the signaling processes may suspend themselves. An integer variable next-count is also provided to count the number of processes suspended on next. Thus, each external procedure F is replaced by

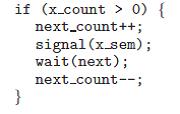


Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented. For each condition x, we introduce a semaphore x\_sem and an integer variable x\_count, both initialized to 0. The operation x. wait () can now be implemented as



The operation x. signal () can be implemented as



**Resuming Processes within a Monitor**

We turn now to the subject of process-resumption order within a monitor. If several processes are suspended on condition x, and an x. signal () operation is executed by some process, then how do we determine which of the suspended processes should be resumed next.