SYNCHRONIZATION

The system consisting of cooperatingsequential processes or threads, all running asynchronously and possiblysharing data. We illustrated this model with the producer-consumer problem,described how a bounded buffer could be used to enable processes to sharememory.

Solution allows at most BUFFER.SIZE - 1 items in the buffer at the sametime. Suppose we want to modify the algorithm to remedy this deficiency. Onepossibility is to add an integer variable counter, initialized to 0. counter isincremented every time we add a new item to the buffer and is decrementedevery time we remove one item from the buffer. The code for the producerprocess can be modified as follows:

```
while (true)
{
/* produce an item in nextProduced */
while (counter == BUFFER.SIZE)
; /* do nothing */
buffer[in] = nextProduced;
in = (in + 1) % BUFFER-SIZE;
counter++;
}
```

The code for the consumer process can be modified as follows:

```
while (true)
{
while (counter == 0)
; /* do nothing */
nextConsumed = buffer [out] ,-
out = (out + 1) % BUFFER_SIZE;
counter--;
/* consume the item in nextConsumed */
}
```

Although both the producer and consumer routines are correct separately,they may not function correctly when executed concurrently. As an illustration,suppose that the value of the variable counter is currently 5 and that theproducer and consumer processes execute the statements "counter++" and"counter—" concurrently.

We can show that the value of counter may be incorrect as follows. Notethat the statement "counter++" may be implemented in machine language (ona typical machine) as

Register1=- counter
Register1 = *register1* + 1
counter =*register1*

where *register1*is a local CPU register. Similarly, the statement "counter—" isimplemented as follows:

register2=counter
register2 = *register2*- 1
counter = *register2*

where again *register2*is a local CPU register. Even though *register1*and*register2*may be the same physical register (an accumulator, say), rememberthat the contents of this register will be saved and restored by the interrupthandler.

The concurrent execution of "counter++" and "counter—" is equivalentto a sequential execution where the lower-level statements presented previouslyare interleaved in some arbitrary order. One such interleaving is

Register1= counter *{register1 = 5}*
register 1= *register1*+ 1 {*register1*= 6}
register2. = counter {*register2* =5}
register2 = *registeri* — 1 {*register2* =4}
counter = *register1* {*counter* = 6}
counter = *register2* {*counter* = 4}

Notice that we have arrived at the incorrect state "counter == 4", indicatingthat four buffers are full, when, in fact, five buffers are full. If we reversed theorder of the statements, we would arrive at the incorrect state"counter —— 6".

We would arrive at this incorrect state because we allowed both processesto manipulate the variable counter concurrently. A situation like this, whereseveral processes access and manipulate the same data concurrently and theoutcome of the execution depends on the particular order in which the accesstakes place, is called a **race condition.** To guard against the race conditionabove, we need to ensure that only one process at a time can be manipulatingthe variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

1. The Critical-Section Problem

Consider a system consisting of *n* processes {PQ, PI, ..., *P,,~\}*. Each processhas a segment of code, called a **critical section,** in which the process maybe changing common variables, updating a table, writing a file, and so on.The important feature of the system is that, when one process is executing inits critical section, no other process is to be allowed to execute in its criticalsection. That is, no two processes are executing in their critical sections at thesame time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its criticalsection. The section of code implementing this request is the **entry section.** Thecritical section may be followed by an **exit section.** The remaining code is the**remainder section.** The general structure of a typical process P, is shown inFigure 6.1. The entry section and exit section are enclosed in boxes to highlightthese important segments of code.

```
do{
\\entry section
critical section
\\exit section
\\remainder section
} while (TRUE);
```

General structure of a typical process *Pi.*

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

1. Mutual exclusion. If process P; is executing in its critical section, then noother processes can be executing in their critical sections.

2. **Progress.** If no process is executing in its critical section and someprocesses wish to enter their critical sections, then only those processesthat are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selectioncannot be postponed indefinitely.

3. Bounded waiting. There exists a bound, or limit, on the number of timesthat other processes are allowed to enter their critical sections after aprocess has made a request to enter its critical section and before that request is granted.

2. Peterson's Solution

It is a classic software-based solution to the critical-sectionproblem known as **Peterson's solution.** Peterson's solution is restricted to two processes that alternate executionbetween their critical sections and remainder sections. The processes arenumbered Po and Pi. For convenience, when presenting P,-, we use *Pj* todenote the other process; that is, j equals 1 — i.

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

```
int turn;
boolean f l a g [2] •
```

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. Theflag 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 enterits critical section.

To enter the critical section, process P, first sets flag[i] to be true andthen sets turn to the value j, thereby asserting that if the other process wishesto enter the critical section, it can do so. If both processes try to enter at thesame time, turn will be set to both i and j at roughly the same time. Onlyone of these assignments will last; the other will occur but will be overwrittenimmediately. The eventual value of turn decides which of the two processesis allowed to enter its critical section first.

```
do
 {
flag[i] = TRUE;
turn = j ;
while (flag[j] turn == j ) ;
critical section
flag[i] = FALSE;
remainder section
} while (TRUE);
```

The structure of process *P-*, in Peterson's solution.

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 onlyif either flag[j] == false or turn -- i. Also note that, if both processescan 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 havesuccessfully executed their while statements at about the same time, since thevalue 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 persistas long as *Pj* is in its critical section, the result follows: Mutual exclusion ispreserved.

To prove properties 2 and 3, we note that a process P, can be prevented fromentering the critical section only if it is stuck in the while loop with the conditionflag [j] == true and turn == j; this loop is the only one possible. If P; is notready to enter the critical section, then flag [j] == false, and P; can enter itscritical section. If *Pj* has set flag [j] to true and is also executing in its whilestatement, then either turn == i or turn == j . If turn == i, then P, will enterthe 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, toenter 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 executingthe while statement, P,- will enter the critical section (progress) after at mostone entry by P/ (bounded waiting).

3. Synchronization Hardware

We have just described one software-based solution to the critical-sectionproblem. We explore several more solutions to thecritical-section problem using techniques ranging from hardware to softwarebasedAPIs available to application programmers. Hardware features can make any programming task easier and improve system efficiency. In this section, we present some simple hardware instructionsthat are available on many systems and show how they can be used effectivelyin solving the critical-section problem.

The critical-section problem could be solved simply in a uniprocessor environment, we could be sure that the current sequenceof instructions would be allowed to execute in order without preemption. Unfortunately, this solution is not as feasible in a multiprocessor environment.Disabling interrupts on a multiprocessor can be time consuming, as themessage is passed to all the processors. This message passing delays entry intoeach critical section, and system efficiency decreases.

```
boolean TestAndSet(boolean *target) {
boolean rv = *target;
*target = TRUE;
return rv;
```

The definition of the TestAndSet () instruction.

```
do {
while (TestAndSetLock(&lock) )
; // do nothing
// critical section
lock = FALSE;
// remainder section
}while (TRUE);
```

Mutual-exclusion implementation with TestAndSet ( ) .

The TestAndSet() instruction can be defined as shown in Figure . The important characteristic is that this instruction is executed atomically.Thus, if two TestAndSet C) instructions are executed simultaneously (each ona different CPU), they will be executed sequentially in some arbitrary order. Ifthe machine supports the TestAndSet () instruction, then we can implementmutual exclusion by declaring a Boolean variable lock, initialized to false.The structure of process P, is shown in above Figure.

The SwapO instruction, in contrast to the TestAndSet0 instruction,operates on the contents of two words; it is defined as shown in Figure .Like the TestAndSet 0 instruction, it is executed atomically. If the machinesupports the SwapO instruction, then mutual exclusion can be provided asfollows. A global Boolean variable lock is declared and is initialized to false.In addition, each process has a local Boolean variable key. The structure ofprocess *P*, is shown in Figure .

```
void Swap(boolean *a, boolean *b)
{
boolean temp = *a;
*a = *b;
*b = temp;
}
```

The definition of the Swap () instruction.

```
do { ,
key = TRUE;
while (key == TRUE)
Swap (&lock, &key) ,-
// critical section
lock = FALSE;
// remainder section
}while (TRUE);
```

Mutual-exclusion implementation with the SwapO instruction.

4. Semaphores

The various hardware-based solutions to the critical-section problem (usingthe TestAndSetC) and SwapO instructions) arecomplicated for application programmers to use. To overcome this difficulty,we can use a synchronization tool called a semaphore.

A semaphore S is an integer variable that, apart from initialization, isaccessed only through two standard atomic operations: wait () and signal ().The waitO operation was originally termed P (from the Dutch *probercn,* "totest"); signal () was originally called V (from *verhogen,* "to increment"). Thedefinition of wait() is as follows:

```
wait(S) {
while (S <= 0)
; // no-op
S--;
}
```

The definition of signal () is as follows:

```
signal(S) {
S + + ;
}
```

All the modifications to the integer value of the semaphore in the wait ()and signal() operations must be executed indivisibly. That is, when oneprocess modifies the semaphore value, no other process can simultaneouslymodify that same semaphore value.

In addition, in the case of wait(S), thetesting of the integer value of S (S < 0), and its possible modification (S—),must also be executed without interruption. We shall see how these operationscan be implemented in Section 6.5.2; first, let us see how semaphores can beused.

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 only between 0 and 1. On somesystems, binary semaphores are known as **mutex locks,** as they are locks thatprovide *mutual Ex*clusion.

We can use binary semaphores to deal with the critical-section problem formultiple processes. The *n* processes share a semaphore, mutex, initialized to 1.Each process P, is organized as shown in Figure.

```
do {
wait(mutex);
// critical section
signal(mutex);
// remainder section
}while (TRUE);
```

Mutual-exclusion implementation with semaphores.

Counting semaphores can be used to control access to a given resourceconsisting of a finite number of instances. The semaphore is initialized to thenumber of resources available. Each process that wishes to use a resourceperforms a waitQ operation on the semaphore (thereby decrementing thecount). When a process releases a resource, it performs a signal () operation(incrementing the count). When the count for the semaphore goes to 0, allresources are being used. After that, processes that wish to use a resource willblock until the count becomes greater than 0.

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 thattries to enter its critical section must loop continuously in the entry code. Thiscontinual looping is clearly a problem in a real multiprogramming system,where a single CPU is shared among many processes. Busy waiting wastesCPU cycles that some other process might be able to use productively. Thistype of semaphore is also called a **spinlock** because the process "spins" whilewaiting for the lock.

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

A process that is blocked, waiting on a semaphore S, should be restartedwhen some other process executes a signal() operation. The process isrestarted by a wakeup () operation, which changes the process from the waitingstate to the ready state. The process is then placed in the ready queue.

To implement semaphores under this definition, we define a semaphore asa "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 . Whena process must wait on a semaphore, it is added to the list of processes. Asignal () operation removes one process from the list of waiting processesand 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 () 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 operationsare provided by the operating system as basic system calls.

If the semaphore value is negative, its magnitudeis the number of processes waiting on that semaphore. The list of waiting processes can be easily implemented by a link field ineach process control block (PCB). Each semaphore contains an integer valueand a pointer to a list of PCBs. One way to add and remove processes fromthe list in a way that ensures bounded waiting is to use a FIFO queue, wherethe semaphore contains both head and tail pointers to the queue.

Deadlocks and Starvation

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

To illustrate this, we consider a system consisting of two processes, P1 and P2 , each accessing two semaphores, S and Q, set to the value 1:

```
P1                  P2
wait(S);            wait(Q);
wait(Q);            wait(S);
---------           -----------
---------           ------------
signal(S);          signal(Q);
signal(Q);          signal(S);
```

Suppose that *P1*executes wait (S) and then P2 executes wait (Q). When P1 executes wait(Q), it must wait until P2 executes signal(Q). Similarly, whenP2 executes wait(S), it must wait until P1 executes signal(S). Since thesesignal () operations cannot be executed, P1 and P2 are deadlocked.

We say that a set of processes is in a deadlock state when every process inthe set is waiting for an event that can be caused only by another process in theset.

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 listassociated with a semaphore in LIFO (last-in, first-out) order.

5. Classic Problems of Synchronization

In this section, we present a number of synchronization problems as examples. In our solutionsto the problems, we use semaphores for synchronization.

The Bounded-Buffer Problem

The *bounded-buffer problem* is commonlyused to illustrate the power of synchronization primitives.

We assume that the pool consists of *n* buffers, each capable of holdingone item.

The mutex semaphore provides mutual exclusion for accesses to thebuffer pool and is initialized to the value 1.

The empty and f u l l semaphorescount the number of empty and full buffers. The semaphore empty is initializedto the value n; the semaphore f u l l is initialized to the value 0.

The code for the producer process is shown in below Figure ; the code forthe consumer process is shown in below Figure. We can interpret this code as the producerproducing full buffers for the consumer or as the consumer producing emptybuffers for the producer.

```
do {
// produce an item in nextp
wait(empty);
wait(mutex);
// add nextp to buffer
signal(mutex);
signal(full);
}while (TRUE) ,-
```

The structure of the producer process.

```
do {
wait(full);
wait(mutex);
// remove an item from buffer to nextc
signal(mutex);
signal(empty);
// consume the item in nextc
}while (TRUE);
```

The structure of the consumer process.

The Readers-Writers Problem

A database is to be shared among several concurrent processes. Some of theseprocesses may want only to read the database, whereas others may want toupdate (that is, to read and write) the database. We distinguish between thesetwo types of processes by referring to the former as **readers** and to the latteras **writers**.

Obviously, if two readers access the shared data simultaneously, noadverse affects will result. However, if a writer and some other thread (eithera reader or a writer) access the database simultaneously, chaos may ensue.To ensure that these difficulties do not arise, we require that the writershave exclusive access to the shared database. This synchronization problem isreferred to as the *readers-writers problem.*

The readers-writers problem, requires that no readerwill be kept waiting unless a writer has already obtained permission to usethe shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.

In the solution to the readers-writers problem, the reader processesshare the following data structures:

semaphore mutex, wrt;
int readcount;

The semaphores mutex and wrt are initialized to 1; readcount is initializedto 0. The semaphore wrt is common to both reader and writer processes.

The mutex semaphore is used to ensure mutual exclusion when the variablereadcount is updated. The readcount variable keeps track of how manyprocesses are currently reading the object.

The semaphore wrt functions as amutual-exclusion semaphore for the writers. It is also used by the first reader that enters or exits the critical section. It is not used by readers whoenter or exit while other readers are in their critical sections or last.

```
do {
wait(wrt);
// writing is performed
signal (wrt) ,-
}while (TRUE);
```

The structure of a writer process.

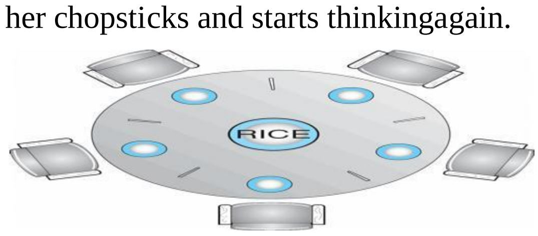
```
do {
wait(mutex);
readcount + + ;
if (readcount == 1)
wait(wrt);
signal(mutex);
// reading is performed
wait (mutex) ,-
readcount--;
if (readcount == 0)
signal(wrt);
signal(mutex);
Jwhile (TRUE);
```

The structure of a reader process.

The code for a writer process is shown in above Figure; the code for a readerprocess is shown in above Figure. Note that, if a writer is in the critical sectionand *n* readers are waiting, then one reader is queued on wrt, and n — 1 readersare 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 waitingwriter. The selection is made by the scheduler.

The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. Thephilosophers share a circular table surrounded by five chairs, each belongingto one philosopher. In the center of the table is a bowl of rice, and the table is laidwith five single chopsticks. When a philosopher thinks, she doesnot interact with her colleagues. From time to time, a philosopher gets hungryand tries to pick up the two chopsticks that are closest to her (the chopsticksthat are between her and her left and right neighbors). A philosopher may pickup only one chopstick at a time. Obviously, she cannot pick up a chopstick thatis already in the hand of a neighbor. When a hungry philosopher has both herchopsticks at the same time, she eats without releasing her chopsticks. Whenshe is finished eating, she puts down both of her chopsticks and starts thinkingagain.



The situation of the dining philosophers.

One simple solution is to represent each chopstick with a semaphore. Aphilosopher tries to grab a chopstick by executing a wait () operation on thatsemaphore; she releases her chopsticks by executing the signal() operationon the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1. The structure ofphilosopheri is shown in Figure

```
do {
wait (chopstick [i] ) ,-
wait(chopstick [ (i + 1) % 5] ) ;
// eat
signal(chopstick [i]);
signal(chopstick [(i + 1) % 5]);
/ / think
}while (TRUE);
```

The structure of philosopher *i*.

Although this solution guarantees that no two neighbors are eatingsimultaneously, it nevertheless must be rejected because it could create adeadlock. Suppose that all five philosophers become hungry simultaneouslyand each grabs her left chopstick. All the elements of chopstick will now beequal to 0. When each philosopher tries to grab her right chopstick, she will bedelayed forever.

we present a solution to the dining-philosophers problem thatensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks areavailable.
- Use an asymmetric solution; that is, an odd philosopher picks up first herleft chopstick and then her right chopstick, whereas an even philosopherpicks up her right chopstick and then her left chopstick.

Finally, any satisfactory solution to the dining-philosophers problem mustguard against the possibility that one of the philosophers will starve to death.A deadlock-free solution does not necessarily eliminate the possibility of starvation.

Sleeping barber Problem

The analogy is based upon a hypothetical barber shop with one barber. The barber has one barber chair and a waiting room with a number of chairs in it. When the barber finishes cutting a customer's hair, he ismisses the customer and then goes to the waiting room to see if there are other customers waiting. If there are, he rings one of them back to the chair and cuts his hair. If there are no other customers waiting, he returns to his chair and sleeps in it.

Each customer, when he arrives, looks to see what the barber is doing. If the barber is sleeping, then the customer wakes him up and sits in the chair. If the barber is cutting hair, then the customer goes to the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits his turn. If there is no free hair, then the customer leaves. Based on a naïve analysis, the above description should ensure that the shop unctions correctly, with the barber cutting the hair of anyone who arrives until there are no more customers, and then sleeping until the next customer arrives. In practice, there are a number of problems that can occur that are illustrative of general scheduling problems.

Many possible solutions are available. The key element of each is a mutex, which ensures that only one of the participants can change state at once. The barber must acquire this mutex exclusion before checking for customers and release it when he begins either to sleep or cut hair. A customer must acquire it before entering the shop and release it once he is sitting in either a waiting room chair or the barber chair. This eliminates both of the problems mentioned in the previous section. A number of semaphores are also required to indicate the state of the system. For example, one might store the number of people in the waiting room.

```
# The first two are mutexes (only 0 or 1 possible)
Semaphore barberReady =0
Semaphore accessWRSeats =1 # if 1, the # of seats in the waiting room can be incremented or decremented
Semaphore custReady =0# the number of customers currently in the waiting room, ready to be served
int numberOfFreeWRSeats = N  # total number of seats in the waiting room
def Barber():
do
{              # Run in an infinite loop.
   wait(custReady);        # Try to acquire a customer - if none is available, go to sleep.
```

```
wait(accessWRSeats);   # Awake - try to get access to modify # of available seats, otherwise sleep.
   numberOfFreeWRSeats +=1# One waiting room chair becomes free.
signal(barberReady)        # I am ready to cut.
signal(accessWRSeats)      # Don't need the lock on the chairs anymore.
# (Cut hair here.)
} while (true);
def Customer():
do# Run in an infinite loop to simulate multiple customers.
{wait(accessWRSeats)        # Try to get access to the waiting room chairs.
if numberOfFreeWRSeats >0: # If there are any free seats:
     numberOfFreeWRSeats -=1#   sit down in a chair
signal(custReady)        #   notify the barber, who's waiting until there is a customer
signal(accessWRSeats)    #   don't need to lock the chairs anymore
     wait(barberReady)         #   wait until the barber is ready
# (Have hair cut here.)
else:                    # otherwise, there are no free seats; tough luck --
signal(accessWRSeats)    #   but don't forget to release the lock on the seats!
# (Leave without a haircut.)
} while (true);
```

6. Monitors

Although semaphores provide a convenient and effective mechanism forprocess synchronization, using them incorrectly can result in timing errors that are difficult to detect.

To illustrate how, we review the semaphore solution to the criticalsectionproblem. All processes share a semaphore variable mutex, which is.initialized to 1. Each process must execute wait (mutex) before entering thecritical section and signal (mutex) afterward. If this sequence is not observed,two processes may be in their critical sections simultaneously.

Let us examinethe various difficulties that may result.

• Suppose that a process interchanges the order in which the wait(j andsignal () operations on the semaphore mutex are executed, resulting inthe following execution:

```
signal(mutex);
critical section
wait(mutex);
```

In this situation, several processes may be executing in their critical sectionssimultaneously, violating the rmitual-exclusion requirement.

• Suppose that a process replaces signal (mutex) with wait (mutex). Thatis, it executes

```
wait(mutex);
critical section
wait(mutex);
```

In this case, a deadlock will occur.

• Suppose that a process omits the wait (mutex), or the signal (mutex), orboth. In this case, either mutual exclusion is violated or a deadlock willoccur.

To deal with such errors, researchers have developed high-level languageconstructs. In this section, we describe one fundamental high-level synchronizationconstruct—the monitor type.

Usage

A type, or abstract data type, encapsulates private data with public methodsto operate on that data. A monitor type presents a set of programmer-definedoperations that are provided mutual exclusion within the monitor. The syntax of a monitor is shown in Figure.

```
monitor monitor name f
{
II shared variable declarations
procedure PI ( . . . ) {
}
p r o c e d u r e P 2 ( . . . ) {…..}
::::::::::::::::::::::::::::::::::::::::
p r o c e d u r e P n ( . . . ) {….}
```

```
i n i t i a l i z a t i o n c o d e ( . . . ) {
……..
}
          }
```

Syntax of a monitor.

Thus, a procedure defined within a monitor can access onlythose variables declared locally within the monitor and its formal parameters.Similarly, the local variables of a monitor can be accessed by only the localprocedures.

The monitor construct ensures that only one process at a time can beactive within the monitor. Consequently, the monitor construct, as defined so far, is not sufficiently powerful formodeling some synchronization schemes. For this purpose, we need to defineadditional synchronization mechanisms. These mechanisms are provided bythe condition construct. A programmer who needs to write a tailor-madesynchronization 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 anotherprocess invokes

x . s i g n a l ( ) ;

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

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

1. Signal and wait. *P* either waits until Q leaves the monitor or waits foranother condition.

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

Dining-Philosophers Solution Using Monitors

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

enum {thinking, hungry, eating} s t a t e [5] ;

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

We also need to declarecondition self [5];

where philosopheri can delay herself when she is hungry but is unable toobtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophersproblem. The distribution of the chopsticks is controlled by the monitor dp,whose definition is shown in Figure.

```
dp.pickup(i);
eat
dp.putdown(i);
```

Each philosopher, before starting toeat, must invoke the operation pi ckup (). This may result in the suspension ofthe philosopher process. After the successful completion of the operation, thephilosopher may eat. Following this, the philosopher invokes the putdownOoperation. Thus, philosopher *i* must invoke the operations pi ckup () andputdownO in the following sequence:

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

     void putdown (int i) {
              state[i] = THINKING;
              // test left and right neighbors
               test((i + 4) % 5);
               test((i + 1) % 5);
      }

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

A monitor solution to the dining-philosopher problem.

It is easy to show that this solution ensures that no two neighbors are eatingsimultaneously and that no deadlocks will occur. We note, however, that it ispossible for a philosopher to starve to death. We do not present a solution tothis problem but rather leave it as an exercise for you.

Implementing a Monitor Using Semaphores

We now consider a possible implementation of the monitor mechanism usingsemaphores. For each monitor, a semaphore mutex (initialized to 1) is provided.A process must execute wait (mutex) before entering the monitor and mustexecute signal (mutex) after leaving the monitor.

Since a signaling process must wait until the resumed process either leavesor waits, an additional semaphore, next, is introduced, initialized to 0, onwhich the signaling processes may suspend themselves. An integer variablenext-count is also provided to count the number of processes suspended onnext. Thus, each external procedure F is replaced by

```
wait(mutex);
body of F
if (next_count > 0)
signal(next);
else
signal(mutex);
```

Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented. For eachcondition x, we introduce a semaphore x_sem and an integer variable x_countboth initialized to 0.

The operation x. wait () can now be implemented as

```
x_count++;
if (next_count > 0)
signal(next);
else
signal(mutex);
wait(x_sem);
x_count—;
```

The operation x. signal () can be implemented as

```
if (x_count > 0) {
next_count++;
signal(x_sem);
wait(next) ;
next_count--;
```

7. Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- **Implements a variety of locks to support** multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
- An event acts much like a condition variable

Linux Synchronization

- Linux:lPrior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
- semaphores
- spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
- mutex locks
- condition variablesnNon-portable extensions include:
- read-write locks
- spin locks