MCA Part II

Paper-XIII

Topic: Semaphores

Prepared by: Dr. Kiran Pandey
(School of Computer Science)

Email-id: kiranpandey.nou@gmail.com

Semaphores

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called **semaphore**. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, wait and signal designated by P() and V() respectively.

The classical definition of wait and signal are :

- **Wait :** decrement the value of its argument S as soon as it would become non-negative.
- **Signal :** increment the value of its argument, S as an individual operation.

Types of Semaphores

Semaphores are mainly of two types:

1. **Binary Semaphore**

It is a special form of semaphore used for implementing mutual exclusion, hence it is often called *Mutex*. A binary semaphore is initialized to 1 and only takes the value 0 and 1 during execution of a program.

2. **Counting Semaphores**

   These are used to implement bounded concurrency.

Detail Description of Semaphore

A semaphore **S** is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal (). The waitO operation was originally termed P (from the Dutch *probercn,* "to test"); signal () was originally called V (from *verhogen,* "to increment"). The definition of wait 0 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 one process modifies the semaphore value, no other process can simultaneously modify that same semaphore

value. In addition, in the case of wait(S), the testing of the integer value of S (S < 0), and its possible modification (S—), must also be executed without interruption.

Consider two currently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose that we require that S2 be executed only after S1 has completed. This scheme can be implemented by letting P1 and P2 share a common semaphore synch, initialized to 0, and by inserting the statements:

S1; signal(synch);
in the process P1 and the statements:
wait(synch);

S2; in the process P2
Since synch is initialized to **0,** P2 will execute S2 only after P1 has involved signal (synch), which is after S1.

The disadvantage of the semaphore definition given above is that it requires busy-waiting, i.e., while a process is in its critical region, any either process it trying to enter its critical region must continuously loop in the entry code. It's clear that through busy-waiting, CPU cycles are wasted by which some other processes might use those productively. To overcome busy-waiting, we modify the definition of the wait and signal operations. When a process executes the wait operation and finds that the semaphore value is not positive, the process blocks itself. The block operation places the process into a waiting state. Using a scheduler the CPU then can be allocated to other processes which are ready to run. A process that is blocked, i.e., waiting on a semaphore S, should be restarted by the execution of a signal operation by some other processes, which changes its state from blocked to ready. To implement a semaphore under this condition, we define a semaphore as:

```
struct semaphore

{

 int value;

List *L; //a list of processes
```

```
 }
```

Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore, it is added to this list. A signal operation removes one process from the list of waiting processes, and awakens it. The semaphore operation can be now defined as follows:

```
wait(S)

{

S.value = S.value -1;

 if (S.value <0)

{

 add this process to S.L;

 block;

 }

}
signal(S)
{ S.value = S.value + 1;

if (S.value <= 0)

 {

 remove a process P from S.L;

wakeup(P);

} }
```

The block operation suspends the process. 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.

Properties of Semaphores

1. Simple
2. Works with many processes
3. Can have many different critical sections with different semaphores
4. Each critical section has unique access semaphores
5. Can permit multiple processes into the critical section at once, if desirable.

Limitations of Semaphores

1. Priority Inversion is a big limitation of semaphores.
2. Their use is not enforced, but is by convention only.
3. With improper use, a process may block indefinitely. Such a situation is called Deadlock. We will be studying deadlocks in details in coming lessons.

Classical Problems of Synchronization

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

```
do { *

// produce an item in nextp

wait(empty);

wait(mutex);

// add nextp to buffer

signal(mutex);
```

signal(full);

}while (TRUE)

Producers/Consumers Problem

Producer – Consumer processes are common in operating systems. The problem definition is that, a producer (process) produces the information that is consumed by a consumer (process). For example, a compiler may produce assembly code, which is consumed by an assembler. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized. These problems can be solved either through unbounded buffer or bounded buffer.

- **With an unbounded buffer** The unbounded-buffer producer- consumer problem places no practical limit on the size of the buffer .The consumer may have to wait for new items, but the producer can always produce new items; there are always empty positions in the buffer.

- **With a bounded buffer** The bounded buffer producer problem assumes that there is a fixed buffer size. In this case, the consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

Shared Data

char item; //could be any data type

char buffer[n]; semaphore full = 0; //counting semaphore semaphore

empty = n; //counting semaphore

semaphore mutex = 1; //binary semaphore

char nextp,nextc;

Producer Process

do

```
{
produce an item in nextp
 wait (empty);
 wait (mutex);
 add nextp to buffer
 signal (mutex);
 signal (full);
}
while (true)
```

Consumer Process

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

Readers and 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:

semaphore mutex, wrt;

int readcount;

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. The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last.

```
do {

wait(wrt);

// writing is performed

signal (wrt) ,-

}while (TRUE);

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);
```

reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections. The code for a writer process is shown in Figure 6.12; the code for a reader process is shown in Figure 6.13. Note

that, 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 read mode; 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.

Dining Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks .When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats

without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

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. We present a solution to the dining-philosophers problem that ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table

do {

wait (chopstick [i]) ,-

wait(chopstick [(i + 1) % 5]) ;

```
// eat

signal(chopstick [i]);

signal(chopstick [(i + 1) % 5]);

/ / think

}while (TRUE);
```

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. Finally, any satisfactory solution to the dining-philosophers problem must guard 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

A barber shop consists of a waiting room with chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barber shop and all chairs are occupied, then the customer leaves the shop. if the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.

The following is a sample solution for the sleeping barber problem.

```
 # define CHAIRS 5 // chairs for waiting customers

 typedef int semaphore; // use this for imagination semaphore

customers = 0; // number of customers waiting for service
```

```
semaphore barbers – 0; // number of barbers waiting for customers

semaphore mutex = 1; // for mutual exclusion

 int waiting = 0; //customers who are waiting for a haircut

 void barber(void) {

while (TRUE)

 {

 down(&customers); //go to sleep if no of customers are zero

 down(&mutex); //acquire access to waiting

waiting = waiting -1 ; //decrement count of waiting

 customers up(&barbers); //one barber is now ready for cut

 hair up(&mutex); //release waiting




 cut_hair(); //this is out of critical region for hair cut


}

}

 void customer(void)

{

 down(&mutex); //enter critical region
 if (waiting < CHAIRS) //if there are no free chairs, leave
```

```
{

waiting = waiting +1; //increment count of waiting customers

up(&customers); //wake up barber if necessary

up(&mutex); //release access to waiting

down(&barbers); //go to sleep if no of free barbers is zero

get_haircut(); //be seated and be serviced

}

Else

{

up (&mutex); // shop is full: do no wait

}

}
```

Explanation:

- This problem is similar to various queuing situations
- The problem is to program the barber and the customers without getting into race conditions
- Solution uses three semaphores:
  - **customers**; counts the waiting customers
  - **barbers;** the number of barbers (0 or 1)
  - **mutex;** used for mutual exclusion

• Also need a variable waiting; also counts the waiting customers; (reason; no way to read the current value of semaphore)
• the barber executes the procedure barber, causing him to block on the semaphore customers (initially 0);
• The barber then goes to sleep;

• When a customer arrives, he executes customer, starting by acquiring mutex to enter a critical region;

• If another customer enters, shortly thereafter, the second one will not be able to do anything until the first one has released mutex;

• The customer then checks to see if the number of waiting customers is less than the number of chairs;
• If not, he releases mutex and leaves without a haircut;
• If there is an available chair, the customer increments the integer variable,

Locks

Locks are another synchronization mechanism. A lock has got two atomic operations (similar to semaphore) to provide mutual exclusion. These two operations are Acquire and Release. A process will acquire a lock before accessing a shared variable, and later it will be released. A process locking a variable will run the following code:

```
Lock-Acquire();

critical section

Lock-Release();
```

The difference between a lock and a semaphore is that a lock is released only by the process that have acquired it earlier. As we discussed above, any process can increment the value of the semaphore. To implement locks, here are something you should keep in mind:

- To make Acquire () and Release () atomic

- Build a wait mechanism.

- Making sure that only the process that acquires the lock will release the lock.

Monitors

To overcome the timing errors that occurs while using semaphore for process synchronization, the researchers have introduced a high-level synchronization construct i.e. the **monitor type**. A monitor type is **an abstract data type** that is used for process synchronization.

Being an abstract data type monitor type contains the **shared data variables** that are to be shared by all the processes and some programmer defined **operations** that allow processes to execute in mutual exclusion within the monitor. A process can **not directly access** the shared data variable in the monitor. The the process has to access it **through procedures** defined in the monitor which allow only one process to access the shared variables in a monitor at a time.

The syntax of monitor is as follow:

```
1. monitor monitor_name
2. {
3. //shared variable declarations
4. procedure P1 ( . . . ) {
5. }
6. procedure P2 ( . . . ) {
7. }
8. procedure Pn ( . . . ) {
9. }
10. initialization code ( . . . ) {
11. }
12. }
```

A monitor is a construct such as only one process is active at a time within the monitor. If other process tries to access the shared variable in monitor, it gets blocked and is lined up in the queue to get the access to shared data when previously accessing process releases it.

Conditional variables were introduced for additional synchronization mechanism. The conditional variable **allows a process to wait inside the monitor** and allows a waiting process to resume immediately when the other process releases the resources.

The **conditional variable** can invoke only two operation **wait**() and **signal**(). Where if a process **P invokes a wait()** operation it gets suspended in the monitor till other process **Q invoke signal()** operation i.e. a signal() operation invoked by a process resumes the suspended process.

Key Differences Between Semaphore and Monitor

1. The basic difference between semaphore and monitor is that the **semaphore** is an **integer variable S** which indicate the number of resources available in the system whereas, the **monitor** is the **abstract data type** which allows only one process to execute in critical section at a time.
2. The value of semaphore can be modified by **wait()** and **signal()** operation only. On the other hand, a monitor has the shared variables and the procedures only through which shared variables can be accessed by the processes.
3. In Semaphore when a process wants to access shared resources the process performs **wait**() operation and block the resources and when it release the resources it performs **signal**() operation. In monitors when a process needs to access shared resources, it has to access them through procedures in monitor.
4. Monitor type has **condition variables** which semaphore does not have.

Monitors are easy to implement than semaphore, and there is little chance of mistake in monitor in comparison to semaphores.

Monitor Implementation
Monitors are implemented by using queues to keep track of the processes attempting to become active int he monitor.

To be active, a monitor must obtain a **lock** to allow it to execute the monitor code. Processes that are blocked are put in a queue of processes waiting for an unblocking event to occur.
These are the queues that might be used:

- **The entry queue** contains processes attempting to call a monitor procedure from outside the monitor.
  Each monitor has one entry queue.
- **The signaler queue** contains processes processes that have executed a notify operation.
  Each monitor has at most one signaler queue.
  In some implementations, a notify leaves the process active and no signaller queue is needed.
- **The waiting queue** contains processes that have been awakened by a notify operation.
  Each monitor has one waiting queue.

- **Condition variable queues** contain processes that have executed a condition variable wait operation.
  There is one such queue for each condition variable.

The relative priorities of these queues determines the operation of the monitor implementation.

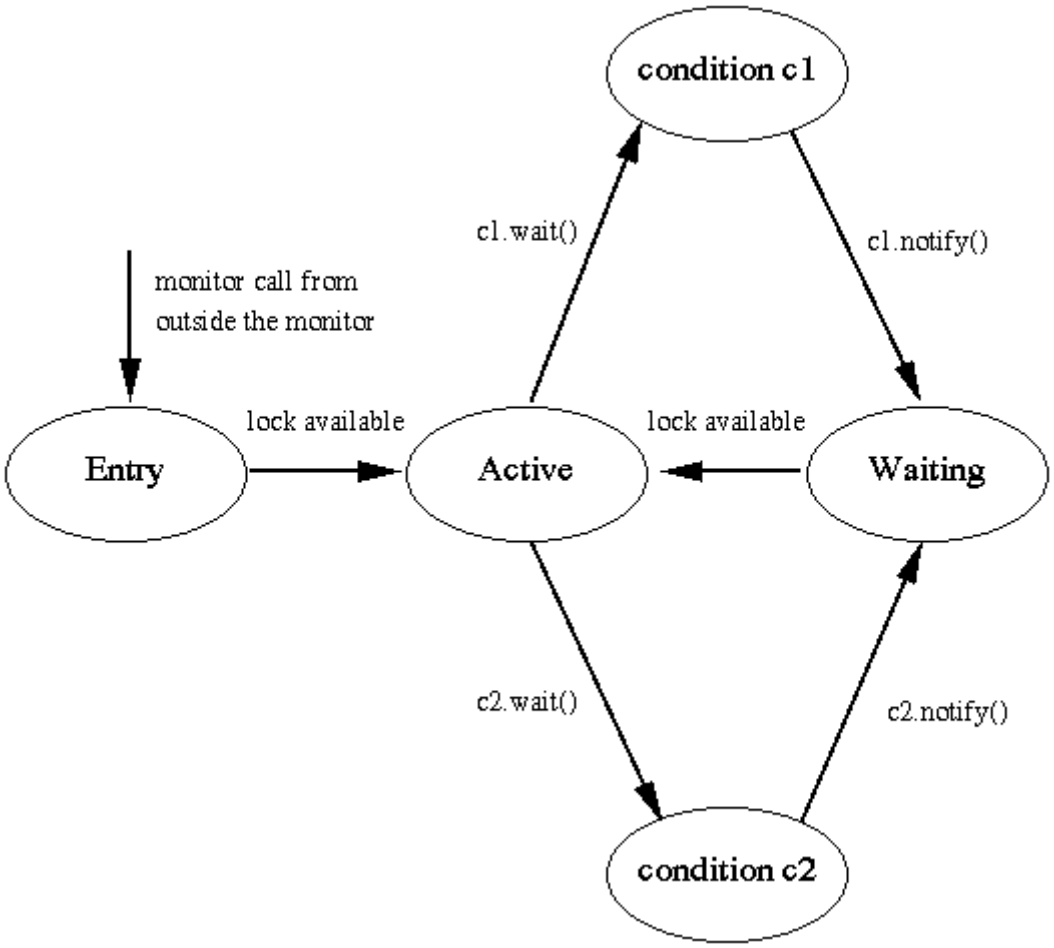

Figure 1: Monitors

The queues associated with a monitor that does not have a signaler queue. The lock becomes available when the active process executes a wait or leaves the monitor.

One modern language that uses monitors is Java.

- Each object has its own monitor.
- Methods are put in the monitor using the *synchronized* keyword.
- Each monitor has one condition variable.
- The methods on the condition variables are: wait(), notify(), and notifyAll().
- Since there is only one condition variable, the condition variable is not explicitly specified.

Questions

1. What are Semaphores? Discuss.
2. Explain Dining philosopher problem.
3. What are locks?
4. What are monitors? Why are they required?

5. Discuss sleeping Barber problem.