Process Management And Synchronization

In a single processor multiprogramming system the processor switches between the various jobs until to finish the execution of all jobs. These jobs will share the processor time to get the simultaneous execution. Here the overhead is involved in switching back and forth between processes.

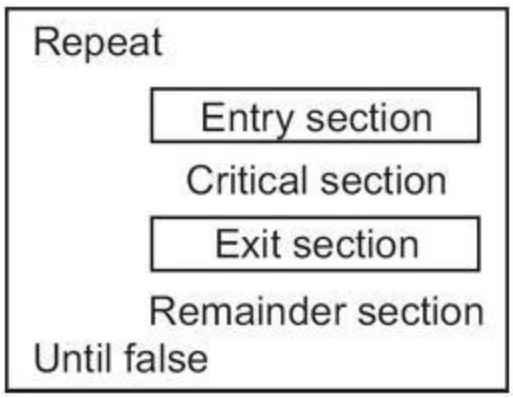
Critical Section Problem

Consider a system consisting of n processes {P0, P1, …, Pn - 1}. Each process has a segment of code called critical section. In which the process may be changing common variables. Updating a table, writing a file, etc., when one process is executing in its critical section, no other process is allowed into its critical section. Design a protocol in such a way that the processes can cooperate each other.

Requirements:-

- Mutual exclusion - Only one process can execute their critical sections at any time.
- Progress - If no process is executing in its critical section, any other process can enter based on the selection.
- Bounded waiting - Bounded waiting bounds on the number of times that the other processes are allowed to enter their critical sections after a process has made a request to enter into its critical section and before that the request is granted.

General structure of a process:



Synchronization With Hardware

Certain features of the hardware can make programming task easier and it will also improve system efficiency in solving critical-section problem. For achieving this in uniprocessor environment we need to disable the interrupts when a shared variable is being modified. Whereas in multiprocessor environment disabling the interrupts is a time consuming process and system efficiency also decreases.

Syntax for interrupt disabling process

Code:

```
repeat
disable interrupts;
critical section;
enable interrupts;
remainder section&gt;
forever.
```

Special Hardware Instructions

For eliminating the problems in interrupt disabling techniques particularly in multiprocessor environment we need to use special hardware instructions. In a multiprocessor environment, several processors share access to a common main memory. Processors behave independently. There is no interrupt mechanism between processors. At a hardware level access to a memory location, excludes any other access to that same location. With this foundation designers have proposed several machine instructions that carry out two actions automatically such as reading and writing or reading and testing of a single memory location.

Most widely implemented instructions are:

- Test-and-set instruction
- Exchange instruction

Test-and-set Instruction

Test-and-set instruction executes automatically. If two test-and-set instructions are executed simultaneously, each on a different CPU, then they will execute sequentially. That is access to a memory location excludes any other access to that same location which is shared one.

Implementation

Code:

```
function Test-and-set (var i : integer) : boolean;
begin
 if i = 0 then
begin
 i : = 1;
Test-and-set := true
end
else
 Test-and-set: = false
 end.
```

Exchange Instruction

A global boolean variable lock is declared and is initialized to false.

Code:

```
Var waiting : array [0 ... n - 1] of boolean
lock : boolean
 Procedure Exchange (Var a, b : boolean);
var temp:boolean;
begin  temp: = a;
 a : = b;
 b: = temp;
```

```
end;
repeat
 key:= true;
repeat
Exchange (lock, key);
until key = false;
 critical section
 lock: = false;
 remainder section
until false;
```

Properties of the Machine-instruction Approach

Advantages

- It is applicable to any number of processes on either a single processor or multiple processors which are sharing main memory.
- It is simple and easy to verify.
- Support multiple critical sections.

Disadvantages

- Busy-waiting is employed.
- Starvation is possible, because selection of a waiting process is arbitrary.
- Deadlock situation may rise.

Semaphores

The above solution to critical section problem is not easy to generalize to more complex problems. For overcoming this problem a synchronization tool called a ‘semaphore’ is used. A semaphore ‘S’ is an integer variable. It is accessed only through two standard atomic operations ‘wait’ and ‘signal’. These operations can be termed as P and V.

Principle

Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific

signal. For signaling, special variables called semaphores are used. To transmit a signal by semaphores, a process is to execute the primitive signal (s). To receive a signal by semaphores, a process executes the primitive wait (s). If the corresponding signal has not yet been transmitted; the process is suspended until the transmission takes place.

Operations

- A semaphore may be initialized to a non-negative value.
- The 'wait' operation decrements the semaphore value. If the value becomes negative, then the process executing the wait is blocked.
- The signal operation increments the semaphore value. If the value is not positive, then a process blocked by a wait operation is unblocked.

No-op stands for no operation.

Code:

```
wait (s) : while S = 0 do no-op
 S: = S - 1;
Signal (s) : S : = S + 1;
```

Usage

To deal with the n-process critical-section problem 'n' processes share a semaphore by initializing mutex to 1.

Code:

```
repeat
 wait (mutex);
 critical section
 signal (mutex);
remainder section
until false
```

Another usage of semaphores is to solve various synchronization problems. For example, concurrently running processes

Code:

```
 P1 with a statement S1.
S1;
Signal (synch);
 and P2 with a statement S2
wait (synch);
 S2;
```

By initializing synch to zero (0), execute S2 only after P1 has invoked signal (synch), which is after S1.

Implementation

In the above semaphore definition the waiting process trying to enter its critical section must loop continuously in the entry code.This continuous looping in the entry code is called busy waiting.Busy waiting wastes CPU cycles, this type of semaphore is called spinlock. These spinlocks are useful in multiprocessor systems. No context switch is required when a process waits on a lock. Context switch may take considerable time. Thus when locks are expected to be held for short times, spinlocks are useful.
When a process executes wait operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can block itself, and restart by wakeup when some other process executes a signal operation. That is wakeup operation changes the process from waiting to ready.

Binary Semaphore

Earlier semaphore is known as counting semaphore, since its integer value can range over an unrestricted domain, whereas a Binary semaphore value can range between 0 and 1. A binary semaphore is simpler to implement than counting semaphore. We implement counting semaphore (s) in terms of binary semaphores with the following data structures.

Code:

```
Var S1: binary-semaphore;
 S2: binary-semaphore;
S3: binary-semaphore;
 C: integer;
```

initially S1 = S3 = 1, S2 = 0
value of C is the initial value of the counting semaphore ‘S’.

Wait operation

Code:

```
 wait (S3);
wait (S1);
 C: = C - 1;
 if C < 0 then
begin
 signal (S1);
wait (S2);
end
else
signal (S1);
 signal (S3);
```

Signal operation

Code:

```
wait (S1);
 C: = C + 1;
 if C = 0 then
signal (S2);
 signal (S1);
```

Classical Problems Of Synchronization

For solving these problems use Semaphores concepts. In these problems communication is to takes place between processes and is called ‘Inter Process Communication (IPC)’.

1) **Producer-Consumer Problem**
Producer-consumer problem is also called as Bounded-Buffer problem. Pool consists of n buffers, each capable of holding one item. Mutex semaphore is initialized to ‘1’. Empty and full semaphores count the number of empty and full buffers, respectively. Empty is initialized to ‘n’ and full is initialized to ‘0’ (zero). Here one or more producers are generating some type of data and placing these in a buffer. A single consumer is taking items out of the buffer one at a time. It prevents the overlap of buffer operations.

Producer Process

Code:

```
repeat
   ...
   produce an item in next P
   ...
   wait (empty);
   wait (mutex);
   ...
   add next P to buffer
   ...
   signal (mutex);
   signal (full);
until false;
```

Consumer Process

Code:

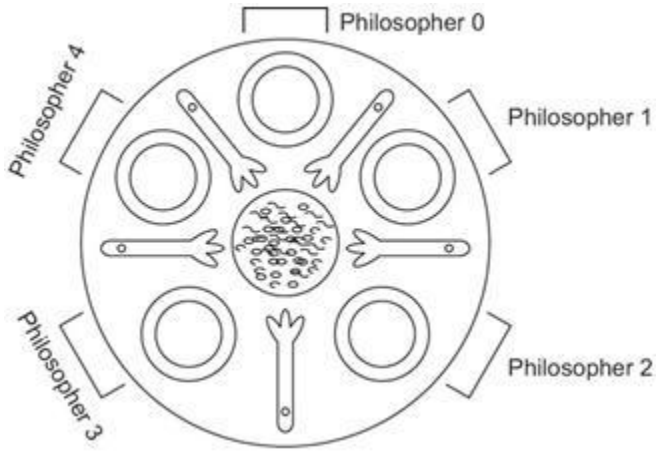
```
repeat
```

```
        wait (full);
        wait (mutex);
        ...
        remove an item from buffer to next C
        ...
        signal (mutex);
        signal (empty);
        ...
        Consume the item in next C
        ...
until false;
```

With these codes producer producing full buffers for the consumer. Consumer producing empty buffers for the producer.

2) **Dining-Philosophers Problem**:-

Dining-pilosophers problem is posed by Disjkstra in 1965. This problem is very simple. Five philosophers are seated around a circular table. The life of each philosopher consists of thinking and eating. For eating five plates are there, one for each philosopher. There is a big serving bowl present in the middle of the table with enough food in it. Only five forks are available on the whole. Each philosopher needs to use two forks on either side of the plate to eat food.



Now the problem is algorithm must satisfy mutual exclusion (i.e., no two philosohers can use the same fork at the same time) deadlock and starvation.

For this problem various solutions are available.

1. Each philosopher picks up first the fork on the left and then the fork on the right. After eating two forks are replaced on the table. In this case if all the philosophers are hungry all will sit down and pick up the fork on their left and all reach out for the other fork, which is not there. In this undegnified position, all philosophers will starve. It is the deadlock situation.
2. Buy five additional forks
3. Eat with only one fork
4. Allow only four philosophers at a time, due to this restriction at least one philosopher will have two forks. He will eat and then replace, then there replaced forks are utilized by the other philosophers (washing of the forks is implicit).
5. Allow philosopher to pick up the two forks if both are available.
6. An asymmetric solution is, an odd philosopher picks up first their left fork and then their right side fork whereas an even philosopher picks up their right side fork first and then their left side fork.

Code for the fourth form of solution is as follows:

Program dining philosophers;

Code:

```
Var fork: array [0 ... 4] of semaphore (: = 1);
 room : semaphore (: = 4);
        i : integer;
procedure philosopher (i : integer);
begin
  repeat
         think;
         wait (room);
         wait (fork [i]);
         wait (fork [(i + 1) mod 5]);
         eat;
         signal (fork [(i + 1) mod 5]);
         signal (fork [i])
         signal (room)
  forever
end;
```

```
begin
  philosopher (0);
  philosopher (1);
  philosopher (2);
  philosopher (3);
  philosopher (4);
end.
```

3) **Readers and Writers Problem**:-

A data object (i.e., file or record) is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, whereas others may want to update (i.e., read and write) the shared object.

In this context if two readers access the shared data object simultaneously then no problem at all. If a writer and some other process (either reader or writer) access the shared object simultaneously then conflict will raise.

For solving these problems various solutions are there:-

1. Assign higher priorities to the reader processes, as compared to the writer processes.
2. The writers have exclusive access to the shared object.
3. No reader should wait for other readers to finish. Here the problem is writers may starve.
4. If a writer is waiting to access the object, no new readers may start reading. Here readers may starve.

General structure of a writer process

Code:

```
 wait (wrt);
  ...
  writing is performed
  ...
 signal (wrt);
```

General structure of a reader process

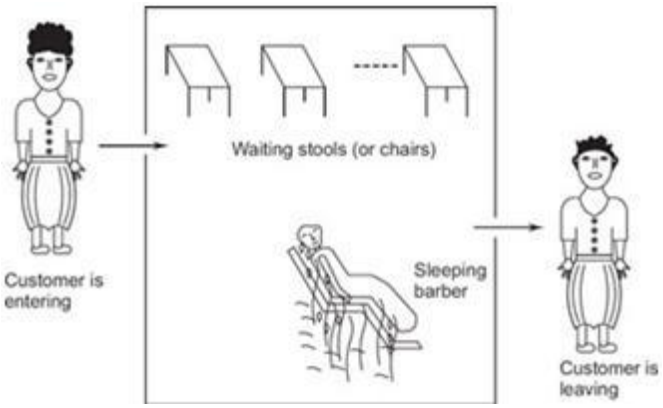
Code:

```
wait (mutex);
  readcount: = readcount + 1;
  if readcount = 1 then wait (wrt);
signal (mutex);
  ...
  reading is performed
  ...
wait (mutex);
  read count: = readcount - 1;
  if readcount = 0 then signal (wrt);
signal (mutex);
```

3) **Sleeping Barber Problem**:-

A barber shop has one barber chair for the customer being served currently and few chairs for the waiting customers (if any). The barber manages his time efficiently.

1. When there are no customers, the barber goes to sleep on the barber chair.
2. As soon as a customer arrives, he has to wake up the sleeping barber, and ask for a hair cut.
3. If more customers arrive whilst the barber is serving a customer, they either sit down in the waiting chairs or can simply leave the shop.



For solving this problem define three semaphores

1. Customers — specifies the number of waiting customers only.
2. Barber — 0 means the barber is free, 1 means he is busy.
3. Mutex — mutual exclusion variable.

Code:

```
# Define Chairs 4
typedef int semaphore;
semaphore customers = 0;
semaphore barber = 0;
semaphore mutex = 1;
int waiting = 0;
Void barber (Void)
{
        while (TRUE)
        {
                waiting = waiting - 1;
                signal (barber);
                cut-hair();
        }
}
Void customer (void)
{
        if (waiting < chairs)
        {
                waiting = waiting + 1;
                signal (customers);
                get-hair cut();
        }
        else
        {
                wait (mutex);
        }
```

```
}
```

Critical Regions

Critical regions are high-level synchronization constructs. Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect use can still result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place, and these sequences do not always occur.

The critical-region construct guards against certain simple errors associated with the semaphore solution to the critical-section problem made by a programmer. Critical-regions does not necessarily eliminate all synchronization errors; rather, it reduces them.

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

Difficulties

- Suppose that a process interchanges the order wait and signal then several processes may be executing in their critical section simultaneously, violating the mutual exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. The code is as follows:
  Signal (mutex);
  ...
  critical section
  ...
  wait (mutex);
- Suppose a process replaces signal (mutex) with wait (mutex) i.e.,
  wait (mutex);
  ...
  critical section
  ...

wait (mutex);
Here a deadlock will occur.

- If a process omits the wait (mutex) or the signal (mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.
  Solution is use high-level synchronization constructs called critical region and monitor. In these two constructs, assume a process consists of some local data, and a sequential program that can operate on the data. The local data can be accessed by only the sequential program that is encapsulated within the same process. Processes can however share global data.

Monitors

A monitor is an another high-level synchronization construct. It is an abstraction over semaphores. Coding monitor is similar to programming in high-level programming languages. Monitors are easy to program. The compiler of a programming language usually implements them, thus reducing the scope of programmatic errors.

A monitor is a group or collection of data items, a set of procedures to operate on the data items and a set of special variables known as ‘condition variables’. The condition variables are similar to semaphores. The operations possible on a condition variable are signal and wait just like a semaphore. The main difference between a condition variable and a semaphore is the way in which the signal operation is implemented. Syntax of a monitor is as follows:

Code:

```
monitor monitor name
{ // shared variable declarations
        procedure P1 (...)
        {
                ...
        }
        procedure P2 (...)
        {
                ...
        }
```

```
        Procedure Pn (...)
        {
                ...
        }
        Initialization code (...)
        {
                ...
        }
}
```

The client process cannot access the data items inside a monitor directly. The monitor guards them closely. The processes, which utilize the services of the monitor, need not know about the internal details of the monitors.

At any given time, only one process can be a part of a monitor. For example consider the situation, there is a monitor M having a shared variable V, a procedure R to operate on V and a condition variable C. There are two processes P1 and P2 that want to execute the procedure P. In such a situation the following events will take place:

- Process P1 is scheduled by the operating system
- P1 invokes procedure R to manipulate V.
- While P1 is inside the monitor, the time slice gets exhausted and process P2 is scheduled.
- P2 attempt to invoke R, but it gets blocked as P1 is not yet out of the monitor. As a result, P2 performs wait operation on C.
- P1 is scheduled again and it exists the monitor and performs signal operation on C.
- Next time P2 is scheduled and it can enter the monitor, invoke the procedure R and manipulate V.
  Monitor concept cannot guarantee that the preceding access sequence will be observed.

Problems

- A process may access a resource without first gaining access permission to the resource.
- A process might never release a resource when once it has been granted access to the resource.

- A process might attempt to release a resource that it never requested.
- A process may request, the same resource twice without first releasing the resource.
- The same difficulties are encountered with the use of semaphores. The possible solution to the current problem is to include the resource access operations within the resource allocator monitor.
  To ensure that the processes observe the appropriate sequences, we must inspect all the programs. This inspection is possible for a small, static system, it is not reasonable for a large or dynamic system.

Message Passing

The need for message passing came into the picture because the techniques such as semaphores and monitors work fine in the case of local scope. In other words, as long as the processes are local (i.e., on the same CPU), these techniques will work fine. But, they are not intended to serve the needs of processes, which are not located on the same machine. For processes which communicate over a network, needs some mechanism to perform the communication with each other, and yet they are able to ensure concurrency. For eliminating this problem message passing is one solution.

By using the message passing technique, one process (i.e., sender) can safely send a message to another process (i.e., destination). Message passing is similar to remote procedure calls (RPC), the difference is message passing is an operating system concept, whereas RPC is a data communications concept.

Two primitives in message passing are:

- Send (destination, & message); i.e., send call
- receive (source & message); i.e., receive call

Source: http://www.go4expert.com/articles/process-management-synchronization-t22193/