Paper Name : Operating System
Prepared by : Debanjali Jana

Process Synchronization

Race Condition

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes. Let us consider two simple examples.

As a first example, suppose that two processes, P1 and P2, share the global variable a . At some point in its execution, P1 updates a to the value 1, and at some point in its execution, P2 updates a to the value 2. Thus, the two tasks are in a race to write variable a . In this example, the "loser" of the race (the process that updates last) determines the final value of a .

For our second example, consider two process, P3 and P4, that share global variables b and c, with initial values b = 1 and c = 2. At some point in its execution, P3 executes the assignment b = b + c, and at some point in its execution,

P4 executes the assignment c = b + c. Note that the two processes update different variables. However, the final values of the two variables depend on the order in which the two processes execute these two assignments. If P3 executes its assignment statement first, then the final values are b = 3 and c = 5. If P4 executes its assignment statement first, then the final values are b = 4 and c = 3.

Peterson's solution:

Peterson's solution is a classic software based solution to the critical section problem. This is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered Po and P1. For convenience, when presenting Pi, we use Pj to denote the other process. Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi 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 flag[i] is true, this value indicates that Pi is ready to enter its critical section.

The algorithm for Peterson's solution is seen in Fig. below.

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

To enter the CS, process P_i 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 CS, 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. Only one of these assignments will last; the other will occur but will be overwritten immediately.
The eventual value of turn decides which of the two processes is allowed to enter its CS first.

Mutual exclusion is preserved.

Each P_i enters its CS only if either *flag*[*j*]==*false* or *turn*==*i*.
Also note that, if both processes can be executing in their CSs at the same time, then *flag*[0]==*flag*[1]==*true*. These two observations imply that $P_0$and $P_i$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 P_j-must have successfully executed the while statement, whereas P_i had to execute at least one additional statement ("*turn == j*").
However, since, at that time, *flag*[*j*] == *true*, and *turn == j*, and this condition will persist as long as $P_j$is in its CS, the result follows: Mutual exclusion is preserved.

The progress requirement is satisfied & The bounded-waiting requirement is met.

A process $P_i$can be prevented from entering the CS 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 $P_j$isnotreadytoentertheCS,then*flag*[*j*]==*false*, and P_i can enter its CS.
If P_j 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 $P_i$will enter the CS. If *turn == j*,then $P_j$will enter the CS.
However, once P_i exits its CS,it will reset *flag*[*j*] to false, allowing
P_i to enter its CS.
If $P_j$resets *flag*[*j*]to true, it must also set *turn* to *i*.
Thus, since P_i does not change the value of the variable turn while executing the while statement,
P_i will enter the CS(progress)after at most one entry by P_j(bounded wait)

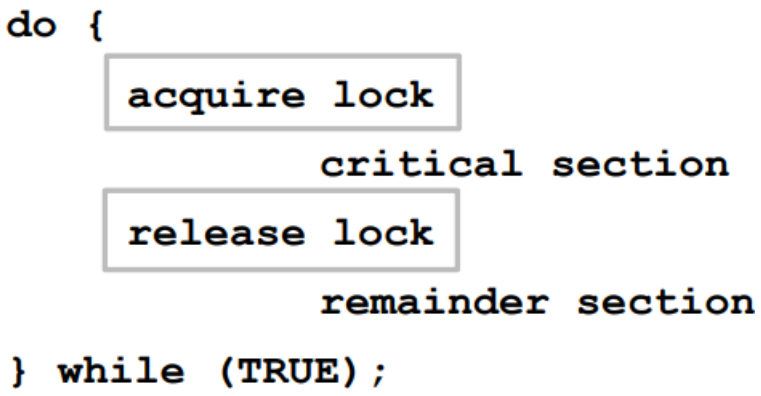
The characteristics of this synchronization method –

- It ensures mutual exclusion
- It follows the strict alteration approach where processes have to compulsorily enter the cs whether they want it or not otherwise the other process will never get the chance to execute again.
- It does not guarantee progress as it follows the strict alteration approach
- It ensures bounded waiting since processes are executed turn wise one by one and each process is guaranteed to get a chance

Synchronization hardware(lock) -

A lock variable provides the simplest synchronization mechanism for processes. Some noteworthy points regarding lock variables are-
It's a software mechanism implemented in user mode,i.e. no support required from the Operating System.
It's a busy waiting solution(keeps the CPU busy even when its technically waiting).

It can be used for more than two processes.
Protect a critical section by first acquire() a lock then release() the lock
Boolean variable indicating if lock is available or not
Calls to acquire()and release()must be atomic
Usually implemented via hardware atomic instructions



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

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

The definition of signal() is as follows:

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

All modifications to the integer value of 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.

Counting semaphore–integer value can range over an unrestricted domain
Binary semaphore – integer value can range only between 0 and 1

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2 . Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by inserting the statements

```
S1;
signal(synch) ;
```

in process P1 and the statements

wait(synch);
S2;
in process P2.

Because synch is initialized to0,P2 will execute S2 only after P1 has invoked signal (synch), which is after statement S1 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.

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

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 list.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()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.

Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared boolean variables S1 and S2 are randomly assigned.

Method used by PI	Method used by P2
While (S1 == S2); Critical Section S1 = S2;	While (S1 != S2); Critical Section S2 = not (S1);

P1	P2
while (S1 == S2); Critica1 Section S1 = S2;	while (S1 != S2); Critica1 Section S2 = Not (S1);

We initially take S1=1, and S2=1

When P1 execute S1 = S2 then it do nothing and process P1 go into critical section after executing it make the value of S1 = S2 and when P1 execute then P2 not execute because that checks different condition in while so both show mutual exclusion. But progress is not shown by these two because in this one process can be delayed infinitely.

Because again both P1 make its own condition true (S1 = S2 and S2 = not (S1);).

Dining-Philosophers problem–classical problem of synchronization

The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down their chopstick and begin thinking again.

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
- Need both to eat, then release both when done

Inthecaseof5philosophers

- Shared data
- Bowl of rice(dataset)
- Semaphore chopstick[5]initialized to1

The structure of Philosopher *i*:

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

Possible remedies to the deadlock problem here -

- Allow at most 4philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up the forks only if both are available.
- Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.