Process synchronization

It is a mechanism to ensure a systematic sharing of resources amongst concurrent processes.

Racing Problem

Consider, two processes P0 and P1 accessing a common integer variable A as follows:

Initial Value of A: 1000

P_0	P_1
Read (A);	Read(A);
A:= A-100;	A:= A+200;
Write(A);	Write(A);

The execution of these two processes is expected to change the value of A to 1100, which is the logical end-result. This will be so, if P_0 and P_1 execute in one of the following two ways:

P_0 followed by P_1

P_1 follwoed by P_0

But suppose P_0 and P_1 are permitted to execute in any arbitrary fashion, then one the following could occur;

Possibility 1

$P_0 \rightarrow P_1 \rightarrow P_0 \rightarrow P_1$

Now the end value of A is 1200

Possibility 2

$P_0 \rightarrow P_1 \rightarrow P_0$

Now the end value of A is 1000

When the end value of execution of two or more concurrent processes is arbitrary and depends on the relative order of execution, is called a ***Racing condition***. This implies that the concurrent processes are racing with each other to access a shared resource in an arbitrary order and produce arbitrarily wrong results.

The racing condition must be avoided using some protocols amongst the concurrent processes, for the use of shared resources.

Critical Section

It refers to the code segment of a process, whereby it accesses a shared resource.

How to avoid racing problem?

The end result of the processes $P_0$and P_1will be correct or logical if their execution follows the sequence $P_0 \rightarrow P_1$or $P_1 \rightarrow P_0$. This implies that at a time only one of the processes should be executing in its critical section. This is known as **Mutual Exclusion**.

Co-operating processes

Two or more concurrent processes, sharing a common resource, have to follow some well-defined protocols to avoid racing problem. Such processes are known as co-operating processes.

<u>Critical section problem</u>

Consider a set of concurrent processes $\{P_0, P_1, P_3, \dots\dots\dots\dots, P_{n-1}, P_n\}$ sharing a common resource R, through the execution of their critical sections. These processes have co-operated with each other to provide a solution to the critical section problem.

Requirements of a critical section solution

An ideal critical-section solution should meet the following requirements –

- Mutual Exclusion
- Progress – Whenever none of the co-operating processes is executing in critical section and some of the co-operating processes are waiting to enter into their critical section, then one of the waiting processes must be immediately enabled to enter its critical section. This decision must not be delayed infinitely. This is termed as requirement of progress.
- Bounded waiting – There must exist a finite upper bound on the number of times that other co-operating processes can enter their critical section, after a process P_1has requested entry into its critical section and before the request is granted. Normally, this upper bound is 1

<u>Critical section solutions</u>

The general structure of a critical section solution will be as follows;

```
do
{
        *Entry section*
        Critical Section
        *Exit Section*
        Remainder Section
} while (TRUE);
```

- Entry Section
- Critical Section
- Exit Section
- Remainder Section

Critical sections algorithms

First we state some algorithms that provide solution to only for a set of two co-operating processes. Then, we discuss generic algorithms, providing solutions for an arbitrary number of co-operating processes.

Algorithm 1: I am finished with it. Now, you have it.

This is applicable to a set of two processes only, say, $\{P_0, P_1\}$ with the **assumption** that

Both are co-operating through a shared integer variable "turn" whose value can be either 0 or 1. If the value of turn is 0 then P_0 can enter into its critical section otherwise P_1 enters into its critical section.

```
int turn = 0;
P0:
do
{
     while(turn == 1); // keep looking, as long as turn equal to 1
     <Critical section>
     turn = 1;       // Enable P1 to enter critical section
     <Remainder section>
} while (TRUE);
P1:
do
{
     while(turn == 0); // keep looking, as long as turn equal to 0
     <Critical section>
     turn = 0;       // Enable P0 to enter critical section
     <Remainder section>
} while (TRUE);
```

Limitation:

It does not meet the requirement of Progress. This limitation arises from the fact that this algorithm considers only, 'Who is permitted to enter critical section' without caring for 'who wants to enter critical section' i.e. the turn is assigned strictly alternatively, without taking into consideration whether the need exists or not.

Algorithm 2: Let me have it, after you finish it.

This algorithm takes into consideration the need of a process to enter its critical section. It is a flag based approach that assumes the following

Boolean variables flag[0] and flag[1] are used to synchronise the two co-operating processes. The flags initially set to FALSE and are accessible to both the processes. Whenever a process P_0 intends to enter its critical section, it indicates its intention by setting the flag, flag[0] which is accessible to the other process P_1 also.

```
typedef enum boolean {FALSE, TRUE}
boolean flag[2];
```

P_0:

```
do
{
flag[0] = TRUE; // Intends to enter critical section
while(flag[1]); // keep looking, as long as flag[1] is true
<Critical section>
flag[0] = FALSE; // Exiting from critical section
<Remainder section>
}
while (true);
```

P_1:

```
do
{
flag[1] = TRUE; // Intends to enter critical section
while(flag[0]); // keep looking, as long as flag[0] is true
<Critical section>
flag[1] = FALSE; // Exiting from critical section
<Remainder section>
} while (true);
```

Limitation:

Whenever a process needs to enter its critical section it indicates its intention by setting its respective flag to TRUE. Before entering critical section, it takes into consideration the state of the other process. If the other process is executing in its critical section, then this process waits till the other process exits from critical section. But, unfortunately, a distinct possibility exists that both processes may set their flags to TRUE almost simultaneously, thus preventing each other from entering their respective critical section.

Algorithm 3: Peterson's Solution

I need it; but first you have, if you also need

```
int turn;
boolean flag[2]; //initially set to false
```

```
p0:
do
{
    flag[0] = true;
    turn = 1;
    while(flag[1] && turn == 1) ;
    <critical section>
    flag[0] = flase;
    <remainder section>
    }
while(true);
```

```
p1:
do
{
    turn = true;
    while(flag[0] && turn = = 0);
    <critical section>
    flag[1] = flase;
    turn = 0;
    <remainder section>
}
while(true);
```

Algorithm 4: Lamport's Bakery Algorithm (generic solution)

Consider n processes $P_0, P_1, P_2, \dots, P_{n-1}$. A process P_i has a PID i assigned by the system. The algorithm attempts to ensure the principle of FCFS and operates as follows –

1. When a process P_i wants to enter in its critical section, it chooses a token for itself.
   The token is an integer, commencing from 1 assigned to the first process. Therefore the token assigned to a process P_i will be an integer, whose value will be higher by one, than the largest held token at that time.
   When no process is executing in its critical section, then amongst the waiting processes, one with the lowest token at that time will be chosen to enter in its critical section.
2. In case of a tie with token, PID is used to break the tie, i.e., tokens of two processes being equal; the one with the lower PID will get preference over the other and enter critical section earlier.
3. Priority of a process P_i is determined by the tuple (t_i, i) where t_i is the token and i is the PID.
   (a) If $t_i = 0$, it indicates that P_i do not want to enter in its critical section and thus not a waiting process.
   (b) A process P_i will enter in its critical section earlier than P_j, if
   $t_i \neq t_j$ && $(t_i, i) < (t_i, j)$ where $(t_i, i) < (t_i, j)$ implies
   (i) $t_i < t_j$ or

(ii) $t_i = t_j$ but $i < j$

```
// Following are the variables used for synchronization
Boolean Choosing[N] = FALSE;
/*initially set to FALSE, when $P_i$ is choosing token, Choosing[i] = TRUE.
int t[N]; // All set to 0 initially. When $P_i$ is waiting to enter in its
critical section, t[i] will be nonzero.
// For a process $P_i$
int i, j;
do
{
     Choosing[i] = TRUE; // $P_i$ goes to choose token for entry into
its critical section
     t[i] = max(t[0],t[1],t[2],..,t[i-1],t[i+1],t[i+2],..,t[n-1])+1;
     Choosing[i] = FALSE;
     for(k=0; k< N; k++)
     {
          while(Choosing[k]) ;
// If Choosing[k] = TRUE, keep looking here, till $P_k$ has finished choosing
its token
          while($t_k \neq 0$ && $(t_k, k) < (t_i, i)$ );
     }
     <critical section>
     t[i] = 0; // $P_i$ has finished executing in its critical section
}while(1);
```

Semaphore

Semaphore is the tool used by OS for process synchronization. There are two variations

- Binary
- Counting

■ **Binary Semaphore**

A binary semaphore is an integer variable, initialized by the OS to 1 and can assume either 0 or 1, can be accesses by a co-operating process through the use of two primitives – “wait” and “signal”.

```
int s = 1;  // let, s be a binary semaphore
```

Wait →this primitive is invoked by a co-operating process, while requesting entry into its critical section. The first process invoking it will make the semaphore value 0 and proceed to enter its CS. If a subsequent process is requesting entry to its CS, when another process is already in its CS then the requesting process will be made to wait. A waiting process repeatedly checks the value of semaphore, till it is found to be 1. Then it will decrement semaphore value to 0 and proceed to its CS. It can be implemented as –

```
void wait(int *s)
{
     while(*s == 0) ; // keep looking for semaphore value
     *s - - ;
}
```

Signal →This primitive is invoked by a co-operating process, when it is exiting from CS. The operation comprises of incrementing the value of the semaphore to 1, to facilitate one of the waiting processes to enter its CS. It can be implemented as –

```
void signal(int *s)
{
     *s++;
}
```

A process pi can be synchronized to access its critical section as –

```
do
{
     wait(&s);
     <critical section>
     signal(&s);
     <remainder section>
}while(1);
```

Requirements of an Ideal solution:

1. Mutual Exclusion: The “wait” operation is to be executed automatically. When a process is in executing in its critical section then other processes will find the semaphore value to be 0 and continue to loop in the “while” loop. So at a time one of the co-operating processes can enter its critical section, subject to satisfaction of the condition that wait is executed automatically.
2. Progress: When no process is executing in critical section, the semaphore value will be 1. Then one of the waiting processes looping in the while loop of wait operation will find the semaphore value to be 1, exit from the while loop, decrement the semaphore to 0 and enter its critical section. So if no process is executing in critical section, and some are waiting to enter, then one of the waiting processes will enter its critical section and thus progress condition is met.
3. Bounded waiting: Arbitrarily, one of the waiting processes will get entry into its critical section, when a co-operating process, executing in its critical section exits. This selection of a process being arbitrary, a process waiting to enter in its critical section, is likely to face starvation. So, the requirement of bounded waiting is not met.

Limitation of Binary semaphore:

a) A solution using binary semaphore does not meet the requirement of bounded waiting.
b) A process willing to enter its critical section, will perform busy-waiting thus wastes CPU cycles. Due to this reason binary semaphore are also known as Spin-locks

■ **Counting Semaphore**

It consists of

a) An integer variable, initialized to a value $K\ (>= 0)$. During operation, it can assume any value $<= K$.
b) A pointer to a process queue. The queue will hold the PCS of all those processes, that are waiting to enter their critical section. This queue is implemented as a FCFS queue, o that the waiting processes are served in a FCFS manner.

A counting semaphore can be implemented as –

```
typedef struct process
{
     int process_id;
     ……
     process * next;
};
typedef struct semaphore
{
     int count;
     process * head;
     process *tail;
};
semaphore s;
```

```
// initialization of semaphore
void init(semaphore *s)
{
     s→count = 1;
     s→head = 0;
     s →tail = 0;
}
……………….
void wait(semaphore *s)
{
     s→count --;
     if(s→count <0)
     {
     <link this process as a tail of semaphore queue>
     block();
     }
}
……………….
void signal(semaphore *s)
{
     s→count ++;
     if(s→count <=0)
     {
     <de-link process p from the head of the semaphore queue>
     wakeup(p); // transfer this process from waiting state to ready
state
     }
}
```

Operation of counting semaphore

Let the initial value of semaphore count be equal to 1.

- When semaphore count equals 1, it implies that no process is executing in its critical section and no process is waiting in the semaphore queue.
- When semaphore count equals 0, it implies that one process is executing in critical section, but no process is waiting in the semaphore queue.
- When semaphore count equals N, it implies that one process is executing in its critical section and N processes are waiting in the semaphore queue.
- When a process is waiting in semaphore queue, it is not performing any busy waiting. It is rather in a “waiting” or “blocking” state.
- When a waiting process is selected for entry into its critical section, it is transferred from “blocked” state to “ready” state.

Advantage of counting semaphore over binary semaphore

1. Since the waiting processes will be permitted to enter their critical sections in a FCFS order, the *requirement of bounded waiting is fully met.*
2. A waiting process does not perform any busy waiting, thus saving some CPU cycles.

Disadvantage of counting semaphore over binary semaphore

1. A counting semaphore is relatively complex to implement, since it involves implementation of FCFS queue.
2. Additional context switches: When a process is not able to enter its critical section, it would involve two context switches –
   (a) from “running ” state to “waiting” state and
   (b) from “waiting” state to “ready” state.
   And then only, the process would enter its critical section. *These context switches will involve certain overheads.*

Implementation/ Emulation of a counting semaphore, using Binary semaphore

Since, the binary semaphores are easier to implement, sometimes it is preferred to emulate counting semaphores (to the extent possible) by using multiple Binary semaphores, as indicated below –

```
Semaphore S1, S2;
int count;
/* The Binary semaphores are initialized as follows */
S1 = 1;
S2 = 0; /* Note this semaphore is initialized to 0; thus first process
performing “wait” operation on this semaphore will go to busy-waiting */
/* The variable count is initialized to value N i.e., initial value of the
“count” of counting semaphore, being emulated */
count = N ;

/* “wait” operation on a binary semaphore */
void wait (Semaphore *S)
{
      while(*S <= 0);  // do nothing
      *S --; //decrement *S and return
}

/* “signal” operation on a binary operation */
void signal (Semaphore *S)
{
      *S++; // Increment *S and return
}
/* The counting semaphore wait is emulated as wait_c */
voidwait_c()
{
      Wait(&s1);
      Count --;
      If(count < 0)
      {
            Signal(&s1);
            Wait(&s2); /* This process will start looping in the ‘while’
loop of “wait” operation of the binary semaphore S1 */
      }
}
```

```
/* The counting semaphore signal is emulated as signal_c */
void signal_c()
{
     Wait(&s1);
     count++;
     if(count <= 0)
          signal(&s2); /* This will enable one of the waiting processes to
exit the “while” loop of “wait” operation of the Binary Semaphore S2 */
     else
          signal(&S1);
}
```

A co-operating process operates as follows –

```
do
{
     wait_c();
     <critical section>
     signal_c();
     <remainder section>
} while(1);
```

When a waiting process exits the busy-waiting “while” loop of “wait” operation of binary semaphore S2, it returns from the “wait” function and executes the “signal(&S1)” statement of the “wait_c” function, returns from the “wait_c” functions and enters critical section.
When a process exits its critical section, one of the waiting processes, at random, will get enabled to enter its critical section. Thus, this emulation fails to emulate the bounded-wait feature of a counting semaphore. So, the emulation does not emulate all the features of a counting semaphore.

Monitor

Monitor is a high level tool for process synchronization. The main limitation of semaphores is the rigidity of the wait and signal operations. In the case of monitors, *the functions to access the global variables are programmer defined*. The global variables, which need accessing by all co-operating processes, are encapsulated within the monitor, along with the functions needed to access them. The global variables, defined within a monitor can be accessed only through the functions, defined by the monitor. *The OS will ensure mutual exclusion and bounded waiting of the cooperating processes by ensuring that at a time only one of the cooperating processes can be active inside a monitor.*

- General structure of a monitor

  Monitor <Monitor name>
  {
  <global variable declarations>
  <functions to access the global variables>
  <initialization function of monitor>
  }
- Condition variables

The condition variables are used along with monitors for synchronization of cooperating processes, as follows –

Condition C; /* C is a variable of type condition */

Operations to access the condition variables:

(a) **C.wait();** /* A process executing this operation is suspended on the condition C, till another process executes C.signal(); */

(b) **C.signal();** /* This resumes exactly one process, if suspended on the condition C. if no process is suspended on C then this operation has no effect */