Process Synchronization

Date: __/__/__

Two process may co-operate or compete. In both case, Process will communicate using shared memory.

```
P1                          P2
{                           {
---                         ---
---                         ---
count++                     count--
---                         ---
}                           }
```

```
P1                          P2
count++                     count--
(1) mov count, R0           mov count, R1
(2) Inc R0                  Dec R1
(3) mov R0, count           mov R1, count
```

(Incase of P_1 & P_2)

If P_1 preempted at (2) then P_2 will read count as 5 not 6. If P_2 is not preempted then after P_2 executed, count will be 4.

~~In case of P_2~~ P_1 now, again P_1 came so, it will run its 3rd instruction. So, count will become 6. But actually count should 5.

If P_1 (1,2) → P_2 (1,2) → P_1 (3) → P_2 (3)

i.e., 1,2 instruction of P_1 then 1,2 of P_2 then 3 of P_1, then 3 of P_2 is executed.

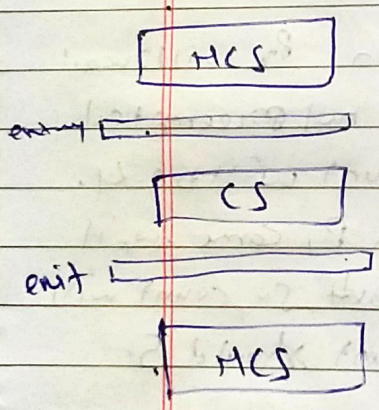
So, in this case final value of count will 4.

So, again inconsistent.

The part of the code of a process which will access the shared memory is called critical section. In other words Shared m/y is needed in case of IPC (Inter Process Communica^n).

i.e. Part of P/m where shared resources are accessed by processes.

The order of execu^n of instructions defines the result produced is called race cond^n. Race cond^n will still there in critical section. i.e. a process may be preempted even if it is in critical section. To solve this Problem, synchronization mechanisms are used.



If a process is in CS then synchronit^n mechanisms will prevent other processes to enter the CS even if the process present in CS is preempted.

Page No.:
Date: / /

However, the sync. mechanisms has ~~their~~ disadvantages such as wasting CPU times to run the code of sync mechanism, lead to deadlock

Lock variable

→ S/w mechanism implemented in usr mode.
→ Busy waiting soln
→ Can be used even for more than two Processes

Initially, Lock = 0

① while (Lock != 0); entry
② Lock = 1
③ CS
④ Lock = 0 exit

In this mechanism, let P_1 is in CS ~~is got preempted~~. Now, P_2 also wants to enter in CS but while loop will stop it and then P_2 will be infinitely waiting for CS. In this case, even if CPU is free (if P_1 preempted in the CS), P_2 can't execute itself.

another drawback: Let P_1 wants to enter CS and it ~~is in line~~ executed the line ① & got preempted. So, P_1 is eligible to CS but Lock is not updated, so, P_2 ~~can enter~~ will enter the CS. So, when P_1 will be again

Page No.:
Date: / /

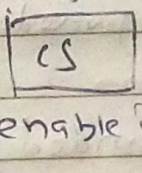
Come to CS wtho after resuming, it will directly enter the CS bcoz line ① has already been executed by P_1 before preemption. So, multiple processes can enter the CS at the same time.

① Load Lock, R0
② Cmp R0, #0
③ JNZ Step 1
④ Store #1, Lock
⑤ CS
⑥ Store #0, Lock

Page No.:
Date: / /

hardware soln

If a process is in CS, it should not be preempted. So, all interrupts should be disabled at the entry of the CS & enabled at the exit of the CS.



Such soln is not efficient in multiprocessor systems.

when a P_i is running its CS in processor 1 then other processes can't run their CS in other processors too which is inefficient use of multiprocessors.

If CS code of a process is too large then disabling the interrupts for such a long time is not good. e.g. if CS code has multiple Printf then holding the CPU is waste of time.

Any soln to critical sec problem must meet following criteria

1. mutual exclusion:- If process P_i is executing in its CS then no other processes can be executing in their CS.

(ii) Progress:- If no process is in its CS and there exist some processes that wish to enter their CS, then the selection of the processes that will enter the CS next cannot be postponed indefinitely.

(iii) Bounded waiting:- A bound must exist on no. of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

For example, the changing room in the a garment shop should be used by only one person at a time. which is mutual exclusion.
when the wing person come out of the ~~room~~ changing room, he/she should not stop other waiting person from entering into the changing room. This ensures the progress.

when the wing person comes out of the changing room, he/she may again ~~want to enter~~ immediately want to enter the ~~critical~~ room. So, there should be a limit that how many times a person can enter into the changing room. this is bounded waiting.

Page No.:
Date:

~~Turn variable (Strict alteration approach)~~ -

one more soft condition is architecture neutrality. i.e. the soln should not depend on a particular hardware or OS.

Bounded waiting is also a soft cond^n but mutual exclusion and progress are hard condi^n.

There are two types of synchroni^n soln are there. (i) busy waiting based soln's in which a process waiting for CS continue to wait untill not getting the CS.

(ii) without busy waiting based soln in which after requesting for the CS, the process does not wait for CS.

Strict alternation approch or turn variable

In the changing room example, in case of turn variable, if a person i.e. let there are 2 persons. If person 1 has the key of the changing room, he/she will unlock the room, use, and again lock the room & will give the key to other person 2. Illy, now person 2 will use the unlock, use, lock the room, and give the key to person 1. Now, if person 1 do not want to use the room, he/she can't give the key to person 2. So, even if person 2 wants to use the room, he/she can't. So, in other words, person 1 is not letting person 2 to enter the room by not ~~not~~ using it. So, it violates 'progress'. 'Bound waiting' is ensured because a person can't use the room two times consecutively.

→ It is a software mechanism implemented at user mode.
→ Busy waiting soln
→ 2-process soln

→ if turn = 0 then process P_0 can enter the CS. if turn = 1, then process P_1 can enter the CS.

P_0

non CS

while(turn != 0);

CS

turn = 1;

non-CS

P_1

non-CS

while(turn != 1);

CS

turn = 0;

non-CS

Here, mutual exclusion is guaranteed because at a time turn will either 0 or 1. So, either P_0 or P_1 will be in CS.

Let P_0 come out of CS and executing its non-CS. now, P_1 is executing its CS and came out and again need the CS but untill P_0 will not execute the CS again then P_1 can't ~~enter~~ use CS. So, ~~Page~~ if non-CS of P_0 is long then P_1 will have to suffer. So, it is not ensuring 'Progress'.

It ensure bounded waiting because the only possible squence to use the CS is $P_0P_1P_0P_1$ or $P_1P_0P_1P_0$ --- . So

In case of Lock, lock = 0, both P_0 & P_1 can enter
lock = 1 no one can enter

In case of turn, turn = 0 P_0 can enter
turn = 1 P_1 can enter

Both are based on one variable.

now, two variable :)))

Interested[0] = T if P_0 wants to enter CS critical sec
Interested[1] = T " P_1 " " " " "

P_0	Interest[0] = F Interest[1] = F	P_1
non-CS		non-CS
interested[0] = T		Interested[1] = T
while (interested[1] == T);		while (Interest[0] == T);
CS		CS
Interested[0] = F		Interest[1] = F

It ensures mutual exclusion

P_0	P_1	
T	T	
T	F	P_0 will enter
F	T	P_1 will enter
F	F	no need of CS

Page No.:
Date: / /

Progress is ensured bcoz if P_0 is not interested then it can't stop P_1 to enter CS. Vice versa, if P_1 is not interested, it can't stop P_0 to enter.

Bounded waiting: Let P_0 is in CS and Interested[1] = T but P_0 wants CS repeatedly. So, as soon as P_0 will come out of CS, it will again make Interested[0] = T. So, in this case, both P_0 and Interested[0] and [1] are true. So, can't enter CS. So, bounded waiting can't be ensured.

Architectural neutrality is guaranteed.

In turn variable Progress was not guaranteed whereas in Interested variable bounded waiting is not guaranteed. So, combination of both may give a soln where both can be guaranteed.

Peterson algorithm exploits the benefits of both turn and interested variables and proposes an algorithm which satisfies mutual exclusion, progress and bounded waiting.

Page No
Date

Peterson's method

→ S/w mechanism at usr mode
→ busy waiting soln
→ 2 process soln
→ It uses turn and interested variables.

Let the array 'interested' and the variable turn are global variable. So, entry and exit section code of a process Pi will be:

```
flag[i] = true;
turn = j;
while (flag[j] && turn == j);
```

```
entry { interested[i] = true;
        turn = j;
        while(interested[j] == true && turn = j);

        CS

exit  { interested[i] = false;
        non CS
```

here, 'j' is for other process.
So, if P0 and P1 are the processes, then
for P0, i=0, j=1
P1, i=1, j=0

Page No.:
Date: / /

P_0

① interested[0] = true
② turn = 1
③ while (interested[1] == True and turn == 1);
CS
④ interested[0] = false;
non-CS

P_1

① interested[1] = true
② turn = 0;
③ while (interested[0] == true and turn == 0);
CS
④ interested[1] = false
non-CS

This algo. ensures the mutual exclusion bcoz at a time only one process can enter the CS. Bcoz ~~both the conditions~~ in 'while' loop, ~~can not be 'true' for both the process~~ turn variable ensures only one process to enter CS.

Progress is also ensured bcoz if a process is not interested for CS then it can't block the other process to enter in CS.

Let the process P_0 is in the 'while' loop and finds that interest of P_1 as true then two possibilities are there. Either P_0 is in CS or P_1 has just executed its line no. ① of the its entry section. In case of earlier, P_0 will wait for P_1 to come out of CS and set interested[1] = false. But, in case of later, P_0 will wait for P_1 to execute line ② of its entry section. As soon as, line ② is executed by P_1, P_0 will be able to enter

Page No.:
Date:

the CS will block while loop of P0 will become false. So, we can see that the process which came to the while loop first will enter the CS first. So, bounded waiting is also satisfied.

Page No.:
Date: / /

Test and Set

(1) Load Lock, R0
(2) CMP R0, #0
(3) JNZ Step 1
(4) Store #1, Lock

CS

(5) Store #0, Lock

In the Lock variable example, we seen that if a process gets preempted after executing line (1) then it is possible that two process may be in the CS at a time. To minimize the possibility, we can re-write the above instruction as follows:

(1) Load Lock, R0
(2) Store #1, Lock
(3) CMP R0, #0
(4) JNZ Step 1

CS

(5) Store #0, Lock

In the above instructions, if the process gets preempted after line no. (1) or (2) or (3), possibility of entering 2 processes in CS is there but in this case, if a process gets preempted after line (1) then only 2 process will enter the CS but if the process enters after line 2 or (3) then possibility is not there bcoz 'Store' instruction has made the Lock as 1 after line (2) whereas in case of 1st, it makes Lock as 1 after line (4)

In the modified set of instructions if line ① and ② are clubbed together, i.e., the process cannot be preempted after line ① then problem of entering two process in the CS at the same time cannot be can be avoided. This soln is called Test and Set Lock.

So, Load Lock, R0 ⟹ TSL Lock, R0
Store #1, Lock

TSL Lock, R0 is atomic operation, Rest instructions are same.

So, this soln ensures mutual exclusion.
It also ensures progress because if a process is not interested in CS, it will not execute the line
TSL Lock, R0

Bounded waiting is not ensured bcoz a process may repeatedly enter the CS.

Page No.:
Date: / /

mutex locks

It is a s/w tool to solve CS problem.
It has a boolean variable "available" associated with it to indicate if the lock is available or not

```
acquire()
{   while(!available);
    available = false;
}
release()
{   available = true;
}
```

acquire() and release() are atomic. So, to disable and enable interrupts should be done before executing these funcs.

```
acquire lock
    CS
release lock
```

Page No.
Date: / /

Binary semaphore as to ensure mutual exclusion

```
wait(S)
{ while(S<=0);
    S=S-1;
}

signal(S)
{ S=S+1;
}
```

Let, initially, S=1

```
wait(S)
CS
signal(S)
```

wait() or P() or W() and signal() or V() or S() are atomic.

It ensures mutual exclusion.

It ensures progress bcoz only interested process will execute wait(S).

It does not ensure bounded waiting bcoz the same process coming out of CS can again claim for the CS.

Binary semaphore and mutex locks are almost same. So, they can be used interchangably.

mutex a, b , a=1, b=1

P_0	P_1
P(a)	P(a)
P(b)	P(b)
CS	CS
V(a)	V(a)
V(b)	V(b)

Here, if P_0 has executed P(a) first then P_0 P_1 will be order of execution even if Preemptions of process occur.

use of binary semaphore to decide the order of execution of process

Page No.:
Date: / /

Let we want to ensure $P_2 \rightarrow P_1 \rightarrow P_3$ as order of execution of the process P_1, P_2, P_3

Let S_1 and S_2 are two semaphores where initially, $S_1 = 0$, $S_2 = 0$

P_1	P_2	P_3
W(S_1)		W(S_2)
P_1	P_2	P_3
S(S_2)	S(S_1)	

⇒ Let Process P_0 prints '1' and P_1 prints '0'. Then apply binary semaphore or mutex to print 10101010

⇒

P_0	P_1
P(a)	P(b)
P(b)	P(a)
CS	CS
V(a)	V(a)
V(b)	V(b)

⇒

P_0	P_1
Print 0	Print 1
Print 0	Print 1

S, T are mutexes,

How to put P and V to ensure 00110011

Let a, b, c, d, e, f, g, h, i, j, k are mutexes. All have 0 initial values

$S_1, S_2, \ldots S_7$ are the processes

Page No.:
Date: / /

S_1 V(a) V(b)

P(a) S_2 V(c) V(d)

P(c) S_4 V(e)

P(d) S_5 V(f)

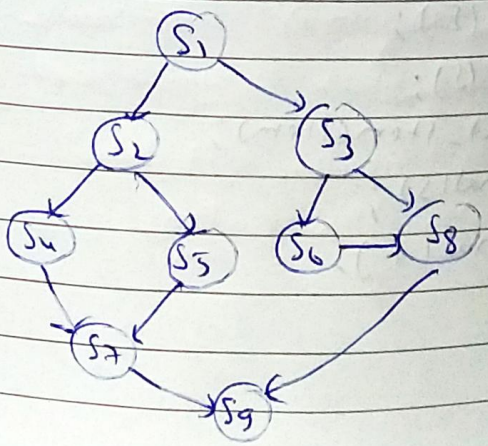
P(e) P(f) S_7 V(k)

P(b) S_3 V(g) V(h)

P(g) S_6 V(i)

P(h) P(i) S_8 V(j)

P(f) P(k) S_9



```
#define N 100   // buffer size
#define count = 0   // items in buffer
void producer(void)
{ int item;
  while(true)
  { int item = produce_item();
    if (count == N) // if buffer is full, no need to produce so producer will sleep.
      sleep();
    insert_item(item);
    count = count + 1;
    if (count == 1) // means earlier count was 0, so, buffer was empty. So,
      wakeup(consumer); // consumer might be sleeping. So, wake up the consumer.
  }
}
void consumer(void)
{ int item;
  while(true)
  { if (count == 0) // buffer is empty. So, nothing to consume. So, consumer will sleep.
      sleep();
    item = remove_item();
    count = count - 1;
    if (count == N-1) // means count was N earlier. So, Producer might be sleeping.
      wakeup(producer); // So, wake up the Producer.
    consume_item(item);
  }
}
```

```
if (count == 0)
    sleep();
```

Page No.:
Date: / /

Assume that after executing the if statement, the consumer process got preempted.

Now, when the producer process will run the following code: if (count == 1)
wakeup(consumer)

Here, the producer will release the wakeup signal to wakeup the consumer but due to the preemption, consumer is still in ready queue & not blocked. So, wakeup signal is wasted in this case. Apart from that, when the consumer will resume, it will execute the next line, i.e. sleep() So, now consumer will go to sleep even if buffer is not empty. As a soln, before sleeping, the consumer should check that whether the producer released a wakeup signal or not. The soln can be implemented using Semaphore.

As of now, we discussed the solns based on busy waiting. with the help of sleep and wakeup, without busy waiting solns can be developed. Semaphore is one such soln.

Page No.:
Date: / /

Semaphore

```
Struct semaphore
{ int value;
  struct process *list;
};
```

There are two types of semaphores
(i) Counting Semaphore (ii) binary Semaphore

In counting semaphore, multiple no of process will be allowed in CS. For example, let there are 4 printers are there, So, maximum 4 process can enter the CS.

Counting Semaphore :- In this case, 'value' in the semaphore structure will decide the no. of process to be entered in CS.
Let, value = 4 then if more than 4 process are entering the CS, then 5th process will be blocked and go to sleep. The 'list' variable will maintain a list of such blocked Processes.

```
wait (semaphore S)
{
  S.value = S.value - 1;
  if (S.value < 0)
  { add the process in 'list'
    sleep();
  }
```

Before entering to CS, a process will have to run wait() function.
if (S.value >= 0) then process is allowed in CS but if it is not then an entry related to the process will be made in the 'list' and process will go to sleep.

```
signal(Semaphore S)
{ S.value = S.value + 1;
    if (S.value <= 0)
    {
      select the oldest process from 'list'
      and wake up that process (i.e. put it in CS)
    }
}
```

Above code will be run by a process coming out of the CS.

It supports bounded waiting, progress, and ~~deadlock~~ mutual exclusion.

Page No.:
Date: / /

Binary Semaphore

```
struct binary_sem
  { int value // either 0 or 1
    struct process *list;
  };

wait( &binary_sem S )
{ if ( S.value == 1)
      S.value = 0
  else
      add the process in 'list'
}

Signal ( binary_sem S )
{
  if ( list is empty ) // i.e. no Process is waiting for CS.
      S.value = 1
  else
      Select a process from list and put in CS
}
```

Solving Producer-Consumer problem using Semaphores

Page No.:
Date: / /

In PC problem, an item will be produced by producer process and kept in buffer. The consumer process will consume the item from buffer. producing and consuming operation should be atomic. So, adding or deleting the buffer should be in CS. Let ~~mutex~~ S ~~is~~ Semaphore S is ensuring this. Initially, S = 1

If buffer is full, the producer will not produce anything. To take care of this, using counting semaphore 'E' whose initial value is n, i.e. size of the buffer. If n = 0, producer will sleep.

If buffer is empty, the consumer can't consume. Let S counting Semaphore 'F' take care of this. Initially, F = 0. when F = n, ~~Semaphore~~ Consumer will sleep.

```
void producer()
{ int item=produce_item();
  wait(E)
  while(true)
  { item = Produce_item();
    wait(E);
      wait(S);
      insert_item(item);
      signal(S);
      signal(F);
  }
}
```

```
void consumer()
{ while(True)
  {
    wait(F);
    wait(S);
    consume_item(item);
    Signal(S);
    Signal(E);

  }
}
```

Reader-writer problem solving using semaphores

Page N
Date:

Read and write operations can be done in a doc file.
more than one readers can read the file at a time.
If a write operation is going on then another process neither read nor write.

So, if a reader is reading, so, then no writer should be allowed to write.
if a writer is ~~so~~ writing, no reader or writer should be allowed.

To solve this problem using semaphore, assume that file is a CS. So, definitely write operation must be done under P() and V() func, but writer does not need to bother who is inside the CS.
But, in case of reader, if ~~reader is~~ another reader is inside then no need of P() and V() but if writer is inside then P() and V() will be required.

So, in case of reader, as soon as 1st ~~writer~~ reader is reading, it should block the writer first.
If the reader is last reader, it should unblock the writer after reading.

```
writer( )
{
   wait(wrt)
   write operation
   signal(wrt)
}

Reader( )
{
   wait(mutex)
   readcount ++;
   if (readcount == 1)
      wait(wrt)
   Signal(mutex)
   read operation
   wait(mutex)
   readcount --.
   if (readcount == 0)
      Signal(wrt)
   Signal(mutex)
}
```