Operating Processes (1)

Concurrent processes executing in the operating system may be either independent processes or cooperating processes

Independent Processes - if a process does not affect another processes executing in the system then these are independent process

→ Any process that do not share any data with another processes

Co-operating Processes - if process affect another processes then those are cooperating processes

Reasons for providing process cooperation

(1) Information Sharing - several users may require to use common files, so, information sharing is provided by concurrent access

(2) Computation Speed up - On multiprocessor system (multiple processing elements) CPUs a particular task can be broken into subtasks, each of these subtasks will be executing in parallel.

(3) Modularity - Dividing the system in to several processes

(4) Convenience - Single user may [illegible] several [illegible]

Concurrent execution that requires cooperation among the Processes requires Mechanisms to allow Processes communicate with each other and to Synchronize their actions.

Producer - consumer Problem (To understand cooperating Processes)

↳ A Producer Produces information that is then consumed by a consumer Process.

→ Eg A Print Program Produces characters that are consumed by printer driver.

→ A compiler may Produce assembly code which is consumed by by an assembler

A Buffer of items is maintained in the system to run Producer and consumer Processes concurrently.

→ Producer can fill the Buffer
→ consumer will make that empty
→ A Producer can Produce one item while the consumer is consuming another item.

→ Process Synchronization. (means Producer and consumer must be Synchronized)

So, that consumer does not try to consume an item that has not yet been Produced

In this situation, the consumer must wait until an item is Produced

types
1. unbounded Buffer Producer-Consumer Problem.
2. Bounded Buffer Producer - Consumer Problem

The un-Bounded Buffer — No Practical

There is no Practical limit on the size of the buffer.
Producer can always Produce new items
But Consumer may have to wait for new items

Bounded Buffer - There is a fixed size Buffer. In this Consumer must wait if the buffer is empty and the Producer must wait if the buffer is full.
The Buffer is Provided by the O.S using Inter process communication or can also be coded by programmer with the use of shared memory

Share Memory Solution to Bounded Buffer Producer-Consumer Problem
Shared memory (Share variables) B/w Producer and Consumer.

```
Var n;
type item = ;
var buffer : array[0....n-1] of item;
    in, out : 0 ... n-1;
```

In and out initialize to value 0. The shared buffer is implemented as circular array with two logical pointers: in and out. The variable in points to next free position in the buffer. out points to the first full position in the buffer.

Buffer is empty when [in = out] (empty) the buffer is full when [in + 1 mod n = out] (full).

Producer code

```
Repeat
    Produce an item in nextp
    while in+1 mod n = out do no-op;
    buffer[in] := nextp;
    in := in+1 mod n;
until false;
```

(local variable in which new item to be deposited. Produced.)

(Simply test the condition until it becomes false.)

Consumer code

```
repeat
while in = out do no-op;
nextc = buffer[out];
out = out+1 mod n;
    Consume the item in nextc
until false
```

This scheme allows n-1 items in the buffer at the same time.

Bounded-Buffer Problem, at most n-1 items in the buffer are allowed at the same time.

Modified Solution (using counter variable)

Integer variable

A counter initialized 0 is used. counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

Code for Producer Process

```
Repeat
  ...
  produce an item in nextp
  ...
  while counter = n do no-op;
     buffer[in] := nextp;
     in := in+1 mod n;
     counter = counter+1;
  until false
```

Code for Consumer Process

```
Repeat
  while counter = 0 do no-op;
  nextc := buffer[out]
  out = out+1 mod n;
     counter := counter-1;
     ...
     consume the item in nextc.
  until false
```

There may not function correctly when executed concurrently.

counter = 5

Counter is implemented in machine language

counter = counter + 1

register$_1$:= counter

register$_1$:= register$_1$ + 1

counter := register$_1$

register$_1$ is local C.P.U register

counter := counter − 1

⇒ Implement in register

register$_2$:= counter

register$_2$:= register$_2$ − 1

counter := register$_2$

T_0: Producer execute register$_1$:= counter {register$_1$ = 5}

T_1: Producer execute register$_1$:= register$_1$ + 1

{register$_1$ = 6}

T_2: Consumer execute register$_2$:= counter {register$_2$ = 5}

T_3 Consumer execute register$_2$:= register$_2$ − 1 {register$_2$ = 4}

T_4: Producer execute counter := register$_1$ {counter = 6}

T_5: Consumer execute counter := register$_2$ {counter = 4}

If interchange the order then counter = 5

counter = 4. Incorrect State because expected counter = 5

When several processes access and manipulate the same data concurrently and the outcome of execution depends on particular order in which the access takes place then that situation is called Race condition

To save the system from race condition we need to ensure that only one process at a time can manipulate the variable counter. So, we need some form of Process Synchronization.

the critical-Section Problem -

Consider a system consisting of n processes $\{P_0, P_1, \cdots P_{n-1}\}$.

Each Process has a segment of code, called a critical Section, - in which Process may change common variables, updating a table, writing a file and so on-

→ The important feature of the system is that, when one process is executing in its critical Section, no other process is allowed to execute in its critical Section.

→ The execution of critical Sections by the processes is mutually exclusive in time.

A solution to the critical - section problem must satisfy the following three requirements -

(1) Mutual exclusion - If Process P_i is executing in its critical section then no other processes can be executing in their critical sections.

(2) Progress - when no process is executing in its critical section and some process want to enter in critical section then only those processes that are not executing in their remainder section can be allowed to enter in its critical section. (No Process running outside its critical section may block other processes from entering the critical section)

(3) Bounded waiting - There is a bound on number of times that process are allowed to enter in their critical sections. OR (No process should have to wait forever to enter its critical section)

Repeat

entry section

critical section

exit section

remainder section

until false;

Structure of a typical process P_i

entry section - each process firstly request to enter in its critical section. Code for request is entry section.

exit section - critical section is followed by an exit section.

Remainder section - The remaining code is remainder section.

Semaphores

↳ solution to critical section Problem.

A semaphore S is an integer variable. We can apply following two types of operation on semaphore

1. wait() or down() or P()
2. Signal() or up() or V()

wait() operation

```
wait(s)
{
while (S<=0)
        {// do-nothing
        }
S--;
}
```

Signal operation

```
Signal(s)
{ S++;
}
```

Semaphore Types — ① Counting Semaphore
② Binary Semaphore

Counting Semaphore - if the value of Semaphore is integer. then it is counting Semaphore it can take values $-\infty$ to ∞.

Binary Semaphore - if the Semaphore variable takes value 0 or 1. then it is called Binary Semaphore. It can be used to provide mutual exclusion b/w two processes when they goes to their critical section.

eg.

```
Repeat
    wait(mutex);
        Critical section
    Signal(mutex);
Until false.
```

(Mutual-exclusion implemented with Semaphores)

Mutex Semaphore

Let mutex = 1 (Initialized to 1)

① → When process P_1 enters in its critical section it calls wait(mutex) i.e wait(1).

means now mutex = 0 is false. So, it will decrement mutex by 1.

Now, mutex = 0

② If Now any Process P_2 tries to enter in its critical section then it would not be able to enter in its critical section because for mutex = 0 wait(mutex) will in loop it

Usage (Application)

Example

eg if we have two processes P_1, and P_2 in concurrent execution.

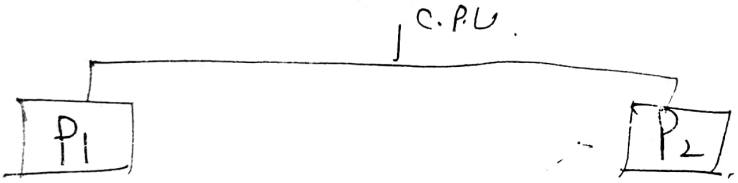
P_1 has statement S_1: printf("1");

P_2 has statement S_2: printf("2");

As C.P.U can execute any for the statement P_1, or P_2 first in case of concurrent execution we want o/P as 1 always.

$\Rightarrow$ 2

So, we can do it by applying wait() and signal() operations on semaphore S. Let initial S=0



S_1

Signal(S)

wait(S)

S_2

So, by using signal after S_1 in Process P_1 and using wait(S) before S_2 in process P_2 we can achieve the desired o/P $\Rightarrow$ 1 2

Stepwise description.

1. eg if P_1 executes first.
   $\Rightarrow$ then. Printf("1") i.e. 1 is printed.
   $\Rightarrow$ Signal(0) is called. and it will set S=1
   $\Rightarrow$ Now if P_2 executes
   then wait(S) i.e wait(1) is called

called and it will set S -- i.e S=0
So, now printf("2"); is called and it will do nothing loop. Thus will not be able to print("2"); first.

So we get desired o/p 1 2

② eg if P2 executes first
Initial S = 0
⇒ then wait(S) i.e wait(0) is called and it will be in do nothing loop. Thus will not be able to print("2"); first.
⇒ Now after some time if P1 executes then S1 is printed i.e 1 is printed it will call signal(S). hence S=1
⇒ Now if again P2 is executed then wait(S) is called. Now, S is set, S=0 and S2 i.e Print 2. thus again we get 1 2 as desired o/p

Example 2

P0	P1
wait(S)	wait(Q)
wait(Q)	wait(S)
:	:
signal(S)	signal(Q)
signal(Q)	signal(S)

Consider a system of two processes P_0 and P_1 each accessing two semaphores S and Q set value 1.

Initially S=1 Q=1

Stepwise description

1) If P_0 starts first it will call wait(S). i.e set the [illegible] Now it will execute wait(Q)

- Now, if wait (S) executes again i.e wait (0). it will enter in do nothing loop. Hence system is in deadlock.

(2) If P_0 execute first initial S=1 Q=1
it will call wait (Q) i.e wait (1) thus set Q=0, Now it will call wait(S) i.e wait (1) i.e. set S=0 Now. if it again prepare wait(0) i.e wait(0) it will enters again in deadlock state.

Thus, we semaphores are not implemented carefully then it may result in a situation of deadlock.

Classical Problems of Synchronization

(1) The Bounded Buffer (~~Reader~~ Producer / Consumer Problem)

(2) The Readers and writer Problem

(3) Dining-Philosopher Problem

The Bounded Buffer Problem (Producer / Consumer) –
If the size of the buffer is limited then it is Bounded Buffer Problem. Here we will take a buffer of size n. and we would have to provide the Synchronization between the two processes Producer and Consumer in such a way that if there is no item in the buffer then Consumer ~~will not~~ will not be

able to access the Buffer. and the producer must wait if the Buffer is full and must access the Buffer in mutual exclusion i.e either Producer or Consumer can access the Buffer at a time (or can enter in its critical section).

We will solve the Problem with the help of Semaphores :-

mutex → Semaphore provide mutual exclusion for access to the buffer. Initialized to 1.

empty → Count number of empty buffers Initialized to n. means In start we allBuffer is empty.

full → Count number of full buffers Initialize to 0.

Code for Producer Process

```
Repeat
    Produce an item in nextp
    ...
    wait(empty);
    wait(mutex);
    add nextp to buffer
    ...
    Signal(mutex);
    Signal(full);
until false
```

Code for producer consumer

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

Step wise description

Initial case

mutex = 1 | empty = n | full = 0

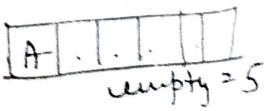
(i) If Producer arrives

(i) Produce an item

(ii) wait(empty) ⇒ wait (6) ⇒ empty-- i.e. Now empty = 5

(iii) wait (mutex) ⇒ wait (1) ⇒ mutex -- i.e Now mutex = 0

(iv) So, Now, it will go into critical section and add to buffer the item (as mutex is 0 No other Process will go to C.S)



(v) signal (mutex) ⇒ signal (0) ⇒ mutex ++ ⇒ mutex = 1

(vi) signal (full) ⇒ signal (0) ⇒ full ++ ⇒ full = 1

Correct as, one item is produced

So, empty is 5. Now.

and full = 1

mutex = 0 Now.

(2) for consumer Process

(3) Now [empty = 5] [full = 1] [mutex = 1]

(i) wait (full) ⇒ wait (1) ⇒ full-- ⇒ [full = 0]

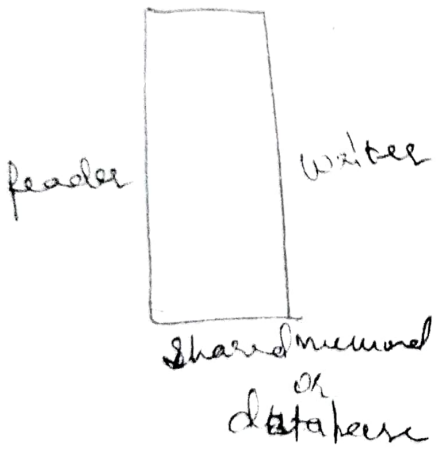
(ii) wait (mutex) ⇒ wait (1) ⇒ mutex -- ⇒ [mutex = 0]
Means now consumer will go in C.S and No other process can get it [mutex = 0]

(iii) Consumer is in C.S Here. Consuming item from buffer

(iv) Signal (mutex) ⇒ mutex ++ ⇒ [mutex = 1]

(v) Signal (empty) ⇒ empty ++ ⇒ [empty = 6]

Reader/writer Problem



A database (such as a file) is shared among many processes. Some processes may want only to read the content of shared object whereas others may want to update shared object.

Readers — are the processes which want to read the database.

Writers — writers want to update the database.

Four conditions that needs to be processed to provide synchronization in reader/writer problem.

① when Reader is reading the database, then writer is not allowed to write data.

$R \rightarrow W^X$

(If P_1 reads, write by any process can't be done)

② Multiple readers are allowed to access database simultaneously

$R \rightarrow R$ ✓

(If P_1 reads, then any other process can also read the database)

③ when writer is accessing database, reader should not be allowed to read

$W \rightarrow R^X$

(If P_1 write, P_2 can't Read)

④ when one writer is writing in the database other writer should not allow to write.

$W \rightarrow W^X$

Explaination :-)

- Semaphore mutex & wrt initialise to 1.
- Read count = 0 (Not semaphore count)
- mutex ensure mutual exclusion when read count is updated.
- Read_count variable keeps tracks of how many processes are currently reading the object.
- wrt act as semaphore for mutual exclusion in writer.

Structure of a writer process :-)

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

Structure of reader process:-

```
do
{  wait (mutex);        // Readcount is updated
   readcount++;
   if (readcount == 1)
      wait (wrt);       // writer waits if at least one reader is reading
   signal (mutex);      // another can update Readcount
   ...
   reading is performed
   ...
   wait (mutex);        // mutual
   readcount--;         // Readcount needs to be updated
   if (readcount == 0)
      signal (wrt);     // signal writer when no reader
   signal (mutex);      // signal mutex
```

Ques.

P1

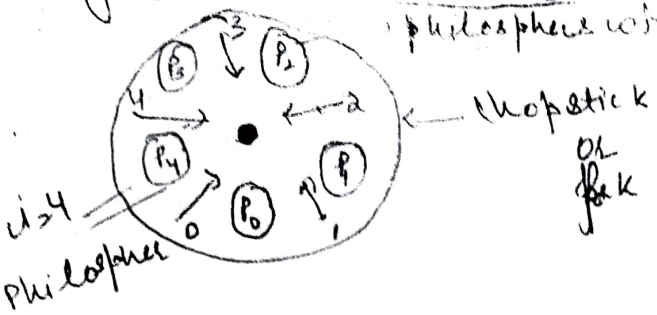
```
while (S1 == S2);
Critical Section
S1 = S2
```

P2

```
while (S1 != S2);
Critical Section
S2 = not (S1);
```

a) Mutual exclusion But not Progress
b) Progress but not mutual exclusion
c) Neither mutual exclusion nor progress
d) Both mutual exclusion and progress

Dining Philosophers problem




```
Void philosopher (int i)
{                                  ↑ philosopher
    while (true)
    { thinking ();
      take_fork(i);   // take left fork
      take_fork ((i+1) % N) ;  // take Right fork
        eat ();
      put_fork (i);  // put left fork back on table
      put_fork ((i+1) % N);  // put right fork back on table
    }
}
```

→ Every philosophers need two chopsticks to eat

→ Here we can imagine, philosophers as processes and chopsticks as resources.

→ If all the philosophers are hungry at the same time then everyone will take the left fork first and when they try to attempt to take the right fork. The right fork is not available.
Then all the philosophers will wait on each other which will leads to deadlock

Deadlock state

Solution with Semaphores

represent each chopstick by a semaphore.
A philosopher tries to grab the chopstick by executing a wait operation on semaphore.
The releases the chopstick by executing the signal operation on appropriate semaphore.

```
#define N 5          ← five philosophers
#define left (i+N-1)%N      → left philosopher
#define right (i+1)%N
#define thinking 0
#define Hungry 1
#define eating 2
Semaphore mutex = 1 ; // Binary Semaphore
Semaphore S[N]  // all S[i] is initialized to zero
int state[N]; // an array to keep track of every one's state
void philosopher(int i)   ← philosopher number
{ while(true)
  { Thinking();
    take_forks(i);
    eat();  // Philosophers are in eating state
    put_forks(i);
  }
}
```

take_forks(i)
{ down (mutex).
state[i] = Hungry ;
test(i);
up(mutex);
down(S[i]);
}

put_forks(int i)

{ down(mutex).
state[i] = Thinking.
test(Left) // allowing Left philosopher to continue.
test(Right); // " Right " "
up(mutex); if they were waiting to complete currently eating philosopher

}

Void test(int i)

{ if(state[i] == Hungry && state[Left] != eating && state[Right] != eating)

{ state[i] = eating

up(S[i]);

}

}

→ S[N] is a binary semaphore initially all are assign to zero.

→ state[N] is a integer array used to keep track of every philosopher state. initially all the philosopher's will be in the thinking state

→ philosophers initially will be in thinking state then they will go to hungry state they will go to eating state and again they will come back to thinking state

(1) Software type (a) Algorithm
(b) Dekker's algorithm
(c) Peterson algorithm

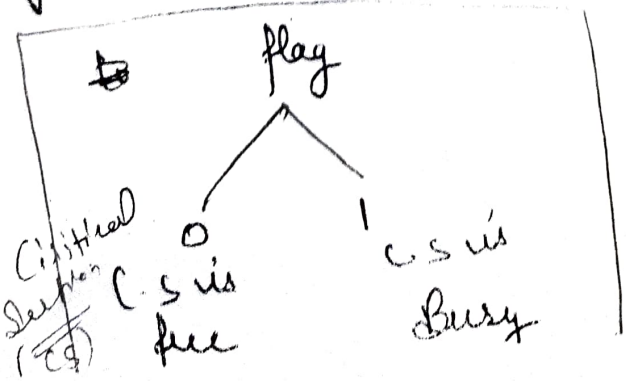
(2) Hardware type
(a) TSL (Test and Set lock) Instruction Set

(3) OS type (a) Counting Semaphore
(b) Binary Semaphore

Programming language (compiler Support type -
(a) Monitors

→ Hardware (TSL) Test and Set lock
(Synchronization Hardware)

TSL Register flag. Copies the current value of flag into register and store the value of "1" into the flag in a single atomic cycle without any pre-emption



(I) C Ri, m[flag]
II CMP Ri, #0
III JMP to step (I)
IV C.S
V m[flag] #0

(code is same for all process)

Analysis

Let initially flag = 0.

assume we have two process in Ready queue [P1 | P2]

execute Process P1

P1 → I // P1 executing Ist Instruction

P1 → II

P1 → III   R1 [0]   CS [P1]

P1 → IV

// Preemption occur & P1 & P2 execute

P2 → I, II, III

After 3rd instruction execution of P2, P2 will jump back to Ist instruction and hold in loop. So, P2 will not enter until P1 will from Critical section.

So TSL is mutually exclusive

Analysis for Progress

↳ Progress is satisfied.

[Analysis for Bounded waiting]

↳ Number of processes are not fix or not countable so this is not bounded waiting.

Note:

Here mutual exclusive satisfied due to TSL register flag which is performing store and load both in single instruction.