KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Website: www.ktunotes.in

Process Synchronization

$reg_1 = counter$
$reg_1 = reg_1 + 1$
$counter = reg_1$

Producer Process

```
while (1) {
/* produce an item in next Produced */
  while (counter == BUFFER_SIZE)
    ; /* do nothing */
  buffer[in] = nextProduced;
  in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

$reg_2 = counter$
$reg_2 = reg_2 - 1$
$counter = reg_2$

Consumer Process:

```
while (1) {
  while (counter == 0)
    ; /* do nothing */
  next consumed = buffer[out];
  out = (out+1) % BUFFER_SIZE;
    counter--;
  /* consume the item in next consumed */
}
```

If counter = 5,

P_0 : producer execute $reg_1 = counter$ $\{reg_1 = 5\}$

P_1 : producer execute $reg_1 = reg_1 + 1$ $\{reg_1 = 6\}$

P_2 : consumer execute $reg_2 = counter$ $\{reg_2 = 5\}$

P_3 : consumer execute $reg_2 = reg_2 - 1$ $\{reg_2 = 4\}$

P_4 : producer execute $counter = reg_1$ $\{counter = 6\}$

P_5 : consumer execute $counter = reg_2$ $\{counter = 4\}$.

- It is an incorrect states, as counter = 4 as there are 4 full buffer but in fact, there are 5 full buffers.
- An incorrect state arrive because, both process manipulate the variable counter concurrently.

- A situation where several processes access & manipulate same data concurrently & outcome depends on a particular order in which access takes place is called race condition.

To avoid race condition, it must be ensured that only one process at a time can be allowed to manipulate the variable counter.

Critical Section Problem.

- Consider a system consisting of n processes $\{P_0, P_1, \ldots P_{n-1}\}$
- Each process has a segment of code called critical section, in which the process may be:
  - ✓ changing common variable
  - ✓ updating a table
  - ✓ writing a file.
- When one process is executing in its critical section, no other process is to be allowed to execute in its CS.
- Each process must request permission to enter its CS. The section of code implementing this request is the entry section. The CS may be followed by an exit section. The remaining code is the remainder section.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (1);
```

A solution to CS problem must satisfy :

✓ Mutual Exclusion.

- If a process Pi is executing in CS, then no other process can be executing in their CS.

✓ Progress Waiting

- If no other process is executing in its CS & if some other process is waiting then the process that is not in the remainder section can enter into the critical section.

✓ Bounded waiting :

- There exists a bound on the no. of times that other processes are allowed to enter its CS after a process has made a request to enter its CS & before that request is granted.

Two process solution.

- Solution to CS problem is obtained — PETERSON PROBLEM (Solution)
- Restricted to two processes — Problem
- Two variables :
  - ✓ boolean flag[2];
  - ✓ int turn;

Initially flag[0] = false [1] = false & turn can either be 0 or 1.

do {

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

critical section.

```
flag[i] = false;
```

remainder section.

} while (1);

To enter the CS, Pi first set flag[i] to be true & then set turn to j, ie, if other process wishes to enter the cs it can do so. If both processes ~~wishes~~ try to enter at same time, turn will be set to both i & j at roughly the same time.

The value of turn decides which of the 2 processes is allowed to enter its CS first.

we need to show that :

- ✓ mutual exclusion is preserved.
- ✓ The progress req. is satisfied.
- ✓ Bounded waiting req. is met.

To prove prop-1 :

- Each Pi enter its CS only if either flag[j] == false or turn == i.
- If both the processes can be executing in their CS at same time, then flag[0] = flag[1] = = true.
- This indicates that P0 & P1 cannot execute while statement at same time, since value of turn can either be 0 or 1.

- Since at a time, flag[j] == true & turn == j & this condition exists as long as P_j is in CS.
Thus mutual exclusion preserved

To prove prop 2 & 3:

Process P_i can be prevented from entering CS only if it is stuck in while loop with flag[j] == true & turn == j. If P_j is not ready to enter CS, then flag[j] = false & P_i can enter CS. If P_j has set flag[j] = true & is also executing in while statement, then either turn == i or turn == j.

If turn == i, then P_i will enter CS.
If turn == j, then P_j will enter CS.

Once P_j exits its CS, it will reset flag[j] = false, allowing P_i to enter its CS. If P_j resets flag[j] = true, it must also set turn == i.

Since P_i does not change the value of turn while executing while statement, P_i will enter the CS (progress) after atmost one entry by P_j (bounded waiting).

PETERSON SOLUTION.

turn = 0, 1.

flag = TRUE, FALSE
(1) (0)

If turn = 0, P_0 turn to enter CS.

If " = 1, P_1 " "

	P_i	P_j
flag	F	F

turn [i/j] only one variable allowed..

Code

	P_i	P_j
flag	T	F

turn [j]

Line1 flag[i] = T; it needs to enter CS.

Line-2 //allowing other process to participate in CS.

Line 3
both cond - T, while - T it loops again bu
if one cond - F, While - F it enter CS.

F & T.
while (flag[j] && turn = j);
↳ F ; P_i enters CS.

Line-4
release lock ; set flag[i] = T.

	P_i	P_j
flag	F	F

III^ly, If P_j needs to enter CS.

	P_i	P_j
flag	F	T

turn [i]

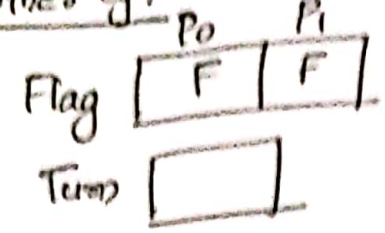
F

while (flag[i] && turn == i)
T

↳ F ; P_j enter CS

release lock, set flag[j] = T.

Another eg:



Po

Entry

① flag[0] = T

② Turn = 1

③ while(flag[1] && turn = 1);

↳ Po enter CS.

P1

① flag[1] = T

② Turn = 0.

③ while(flag[0] && turn = 0)

↳ do not enter CS as Po is in CS.

* Req(1) Mutual Exclusion – Satisfied.

* If Po release lock only P1 can enter CS.

ie, flag[0] = F.

→ Thus P1 can enter CS

* Req(2) – Progress waiting.

Po enter CS then RS then sleep.

P1 enter CS then RS

Po doesn't want to enter CS; so it is never gng to set flag = T. Thus P1 can enter CS.

* Req(3) – Bounded waiting.

↳ Process cannot repeatedly enter into CS if some other process is wanting to enter into CS.

Thus Req(3) failed.

Synchronization Hardware

```
boolean TestAndSet (boolean &target)
{
    boolean rv = target;
    target = true;
    return rv;
}
```

(Definition)

If the machine supports TestAndSet instn, then it can implement mutual exclusion by declaring a boolean variable lock, initialized to false.

```
do {
    while (TestAndSet (lock));

        critical section.

    lock = false;

        remainder section.

} while(1);
```

HARDWARE SYNCHRONIZATION

Limitation of Peterson → Only 2 process.
→ Not adaptable for modern computer arch.

Special HW instruction

↓ Test & Set()

↓ Swap()

Test & Set()

- It is a an atomic Instruction → uninterruptable unit

```
boolean TestAndSet(boolean & target)
{
    boolean rv = target;
    target = TRUE;
    return rv;
}
```

target [T/F]

rv [T/F]

initial rv	initial target	rv	target	return rv
F	F	F	T	F
T	F	F	T	F
F	T	T	T	T
T	T	T	T	T

Initially,

target [F]

rv [F]

Line-1: target [F], rv [F]

Line-2: target [T], rv [F]

Line-3: rv = F.

Line-1: target [F], rv [F]

Line-2: target [T], rv [F]

Line-3: rv = F

Line-1: target [T], rv [T]

Line-2: target [T], rv [T]

Line-3: rv = T

Line-1: target [T], rv [T]

Line-2: target [T], rv [T]

Line-3: rv = T

target	return rv
T	T
F	F

```
do{
①  while(TestAndSet(lock));
②  CS.
③  lock = False;
④  RS.
}while(1);
```

Set target = lock.

Lock	return RV
T	T
F	F

~~lock~~ ~~Lock~~

Lock. False

If P_1 needs to enter
Line ① : while(False);
Line ② : & P_1 enter CS
& lock = True

If P_2 need to enter :
Line ① : while (True);
Cannot enter CS.

* Thus Mutual Exclusion Satisfied .

. when P_1 finishes . lock = False.

P_2 enchecks
Line① : while (F); & enter CS
& set lock = T .

* Progress .

Initial Lock = F.
, then Lock = T.

P_1	P_2
Set Lock = T.	init Lock = F
CS.	Lock = T.
Lock = F	CS
RS (sleep).	Lock = F
	RS

Lock is f, P_1 doesn't want to enter so P_2 can enter as many times as it need.

Thus Progress waiting Satisfied .

* Bounded waiting :

P_1	P_2
L = T.	
CS	
L = F	
Rs	

If P_1 again want to enter CS, it can do as Lock = F. P_2 ~~will~~ should wait many times as if P_1 has higher priority .
NO Bounded Limit for P_1.
- Starvation .

* Bounded waiting Not Satisfied .

If the machine supports swap instn, then mutual exclusion can be implemented by declaring a global boolean variable lock initialize to false & a local boolean variable key.

```
void swap (boolean &a, boolean &b)
{
    boolean temp = a;
    a = b;
    b = temp;
}
```
(defenition)

```
do {
    key = true;
    while (key == true)
        swap (lock, key);
        Critical section
    lock = false;
        remainder section
} while (1);
```

Swap()

```
Void swap(boolean &a, boolean &b)
{
    boolean temp = a;
    a = b;
    b = temp;
}
```

} defenition of swap()

```
do {
    key = true;
    while (key == true);
    swap(lock, key);
        CS
    lock = false;
        R.S.
} while(1);
```

Defn of swap

a	b	a	b
T	F	F	T
F	T	T	F
F	F	F	F
T	T	T	T

~~Defn of swap()~~ Code

Global var. Lock = 

Local var. key = 

If key = T will not enter CS.

If key = F in Line (3) only it will enter CS.

Initially lock = F.

P_1, P_2 ~~Swap~~ Line (1) : key = T.

(2) : key == T.

(3) : swap (lock, key).

lock	key	lock	key
F	T	T	F

Initially lock = F,

P_1	P_2
1: key_{P_1} = T.	(1) key_{P_2} = T.
2: while(key == T) (F, T.)	(2) key == T. (T, T.)
3: swap(lock, key) → lock = T., key = F; CS	(3) swap(lock, key) → True. So it will not enter CS as key = T. swap(key = F, lock = T)
4 lock = F →	key = T, lock = F : Enter CS (as key = F)

Thus ME is satisfied ~~as~~ P_2 is blocked as key = T. P_1 is in CS.

If lock = F in P_1 then only P_2 can enter CS

Progress

P_1	P_2
CS	CS
Lock = F	Lock = F
RS	RS.
sleep	

If P_1 doesn't want to enter CS then it doesn't put lock = F. so P_2 can enter as many times as it need.

Bounded Waiting

High priority P_1	P_2
CS	X (1)
lock = F	
R.S	
CS	X (2)
CS	X (3)

P_2 is starving as P_1 is of high priority P_2 is blocked on entering CS.

So bounded waiting cond is not maintained.

Semaphore

- Semaphore is a synchronization tool
- Semaphore S is an integer variable that can be accessed only through two operations:
  - ✓ wait
  - ✓ signal


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

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

* Binary Semaphore & Counting Semaphore

S = (0 or 1)

Eg: M.E / C.S poblm soln

```
wait(S)
{
  while(S<=0); // T→repeat
               // F→dec. S.
    S--;
}
```

S = (n)

Eg: Resource Mgmt.

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

1) Semaphore usage.

- Semaphore can be used to deal with n-process cs problem.
- The n processes share a semaphore, mutex initialized to 1.
- Semaphore can solve synchronization problem.

```
do {
    wait (mutex);
        critical section
    signal (mutex);
        remainder section.
} while (1);
```

Consider two concurrently running process, P_1 with a statement S_1 & P_2 with a statement S_2. Suppose, S_2 should be executed only after S_1 has completed.

① S_1
② S_2

For this, let P_1 & P_2 share a common semaphore synch, initialized to 0.

The statement, S_1;
signal (synch); in process P_1.

The statement,

wait (synch);

S2; in process P_2.

∵ synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal (synch) which is after S1.

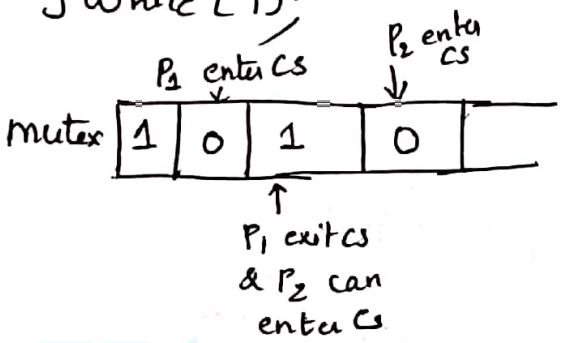
① SEMAPHORE USAGE

Global var (shared by all)
↳ mutex = 1.

```
do
{
    wait(mutex);
        C.S
    signal(mutex);
        R.S
} while(1);
```

```
wait(mutex)                 | signal(mutex)
{                           | {
  while(mutex<=0);          |   mutex++;
    mutex--;                | }
}
```

P_1	P_2
CS	while(0<=0); T (loop. Waits till mutex=1)



P_1 wait(mutex) (1)
{ while(1<=0); F.
mutex--;
}

P_1 enter CS; make mutex = 0
now no other process can enter CS.
P_2 cannot enter as mutex = 0.
when P_1 finishes, it calls signal() & inc. mutex & now mutex = 1 & P_2 can enter CS.

Thus Mutual Exclusion maintained.

Progress

P_1 | P_2

```
while(1<=0)
    C.S
```

① If P_1 doesn't want to enter CS. if wont set mutex = 1. So P_2 can enter CS as many times as it needs.

Bounded waiting.

P_1 enter CS → R.S (mutex = 0)
As P_1 (High priority) if repeats the process.
Thus P_2 is not allowed to enter CS. ie, its starving
Thus this cond. is not maintained.

SEMAPHORE APPLICATIONS.

```
signal(s)        wait(s)
{                {
 s++;             while(s<=0)
}                  s--;
                 }
```

1. CRITICAL SECTION (using mutex).
   - ↳ ORDERING OF STATEMENT
   - ↳ RESOURCE MANAGEMENT

* Ordering of Stmt.

P1
↳ executing stmt S1

P2
↳ S2

S_1 executing first. S_2 will never execute before S_1.

initially

S=0	1	0

```
S1;          wait(S);
signal(S);   S2;
```

P1

S1;
signal(S) → S++
wait(S); → F → (1 <= 0); F
s-- so s--; (0)
and P1 enter CS.

S_2 shouldn't execute before S1. So to check that cond;

~~S=0~~ if

```
wait(S);
```

S=0 initially

while (S<=0); T so it loops thus S_2 cannot enter CS

So S2 cannot execute before S1.

* Resource Management.

Resource – Printer [5]

Process – $P_1, P_2, P_3, P_4, P_5, P_6$ are competing for resource printer.

Any process that want printer must perform wait(S)
Once process release printer it must peform signal(S)

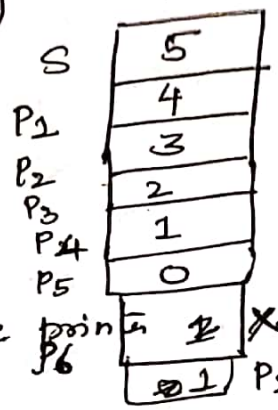
Initially,

S = 5 (5 resources) - Counting Semaphore

P1 request pointer,

```
wait (S)
{
while (S<=0);
    S--;
} use the pointer
```

```
signal(s)
{
  S++;
} release the pointer
```



Now P6 comes it does wait(S); while checking while (S<=0); it is T so it will be looping.

So whenever resource is not available, that process must wait.

If P2 release pointer, it will perform signal(S) so as S=0; after signal (S=1)

Now P6 perform wait(S); → (1<=0); F so P6 will use pointer.

2) Synchronization Implementation :

- The main disadvantage of semaphore is busy waiting
- while a process is in its CS, any other process that tries to enter CS must loop continously in entry code.
- This type of semaphore is called spinlock.
- To overcome the need for busy waiting, modify the defenition of wait & signal semaphore operations. When a process executes the wait operation & finds that the semaphore value is not positive, it must wait. Rather than busy waiting, the process can block itself. The block operation placess a process into a waiting queue. 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 waiting state to ready state. The process is then placed in ready queue.

```
typedef struct
{
    int value;
    struct process *L;
} semaphore;
```

```
void wait (semaphore s)
{
   s.value --;
    if (s.value < 0)
    {
     add this process to s.L;
          block();
    }
}
void signal (semaphore s)
    {
       s.value++;
       if (s.value <= 0)
       {
          remove a process P from s.L;
                wakeup(P);
       }
    }
```

- wait((1)) P_1 trying to enter cs

  s.value--; 1 → 0.

  if (s.value < 0) No. it enters cs

  { cs

  }

  // cs (P_1 enter cs)

S	1	0	-1	-2	-3

-2	-1	0

- P_2 trying s.value--; (0 → -1)

  if (s.value < 0) -1 < 0. ✓

  - add P_2 to process list & block.

- P_3 trying s.value--; (-1 → -2)

  if (s.value < 0) -2 < 0. ✓

  - add P_3 to process list & block.

- P_4 trying s.value--; (-2 → -3)

  if (s.value < 0) -3 < 0 ✓

  - add P_4 to process list & block

w.q.

P_2
P_3
P_4

- When P_1 finishes cs it performs signal()

  s.value++; -3 → -2.

  if (s.value <= 0)

  remove proces from Process list (FIFO).

  So P_2 removed from w.q & it is wakeup.

P_2 enter cs & after performing it call signal()

-2 → -1

-1 <= 0 & P_3 wake up.

P_3 enter cs & then signal()

-1 → 0.

0 <= 0. remove P_3 & P_4 wake up.

We avoided busy-waiting by adding in waiting queue.

3) Deadlocks and Starvation

- Two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting process. When such a state is reached, the process are in deadlocked.

Two process P_0 & P_1, each accessing two semaphores S & Q, set to value 1:

P_0	P_1
wait (S); ✓	wait (Q); ✓
wait (Q); ✓	wait (S); ✓
⋮	⋮
signal (S);	signal (Q);
signal (Q);	signal (S);

Suppose that P_0 executes wait(S) & then P_1 executes wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q). Similarly, when P_1 executes wait(S), it must wait until P_0 executes signal(S). Since these signal operations cannot be executed, P_0 & P_1 are deadlocked.

Another problem related to deadlock is indefenite blocking or starvation, a situation where processes wait indefenitely within the semaphore.

The semaphore is known as counting semaphore, its integer value can range over an unrestricted domain.

Classic Problems of Synchronization

1.) Bounded Buffer Problem.

Assume that it consists of n buffers, each capable of holding one item. The mutex semaphore is initialized to value 1. The empty & full semaphores count the no. of empty & full buffers.

The semaphore empty is initialized to value n.

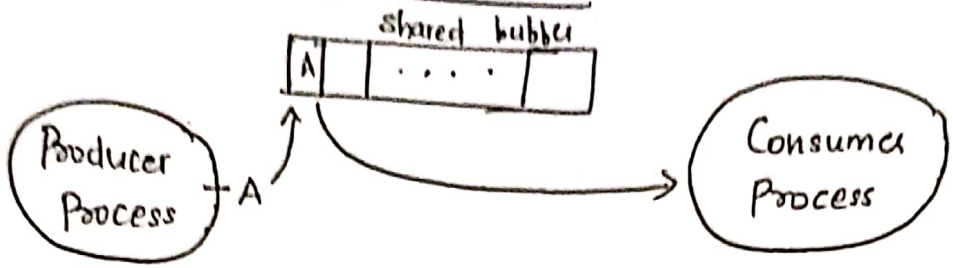
The semaphore full is initialized to value 0.

```
do {
    produce an item in nextp
     wait (empty);
     wait (mutex);
     add nextp to buffer
      signal (mutex);
       signal (full);
    } while (1);

do {
     wait (full);
      wait (mutex);
      remove an item from buffer to nextc.
        signal (mutex);
        signal (empty);
          consume the item in nextc
    } while (1);
```

CLASSICAL PROBLEMS OF SYNCHRONIZATION

① BOUNDED BUFFER PROBLEM (PRODUCER-CONSUMER PROBLEM)



3 issues :

* Mutual Exclusion.
* Overflow - Buffer is full - Producer waits
* Underflow - Buffer is empy - Consumer waits.

Solution using Semaphore

- Mutex
- Empty
- Full.

Structure of producer process

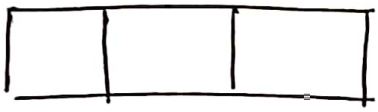
```
do{
    // produce an item
        .....
        wait (empty);
        wait (mutex);
        .....
    // add item to buffer
        .....
        signal (mutex);
        signal (full);
} while (TRUE);
```

```
wait (s)
{
  while (s<=0);
      s--;
}
   // CS.
signal (s)
   {
      s++;
   }
   // RS
```

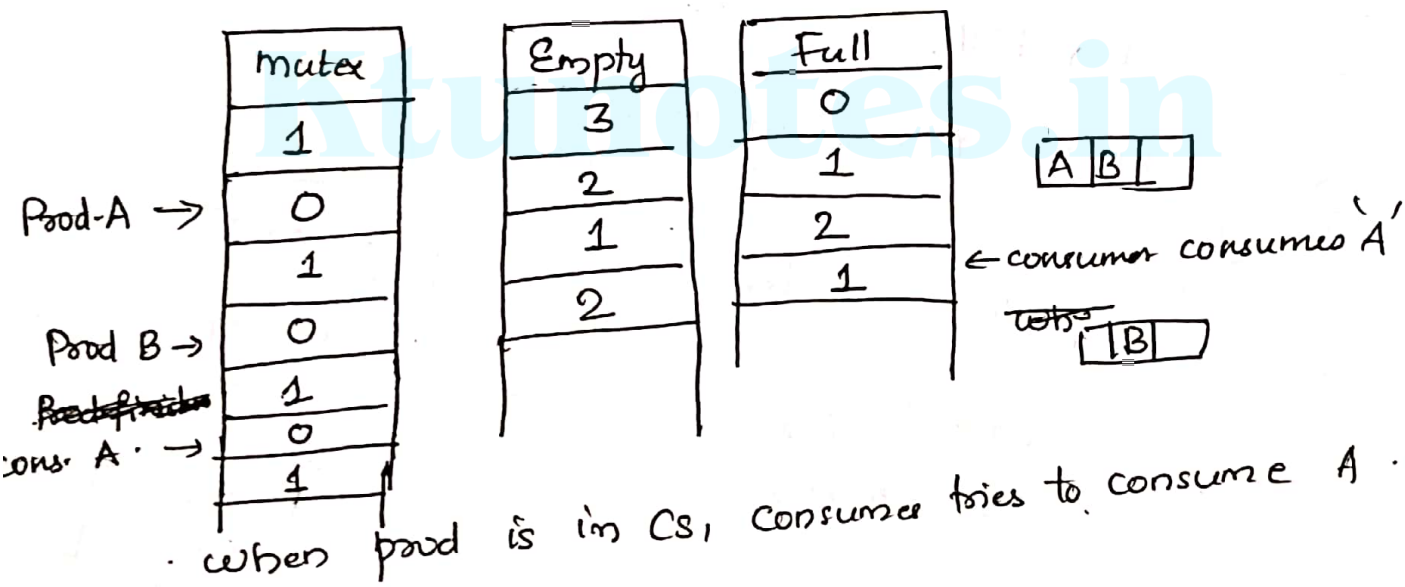
Structure of Consumer Process.

```
do {
    // consume an item.
        wait(full);
        wait(mutex);
        . . . . . .
    // remove an item from buffer
        . . . . . .
        signal(mutex);
        signal(full);
} while(TRUE);
```

① Mutex



n = 3



- when prod is in CS, consumer tries to consume A.

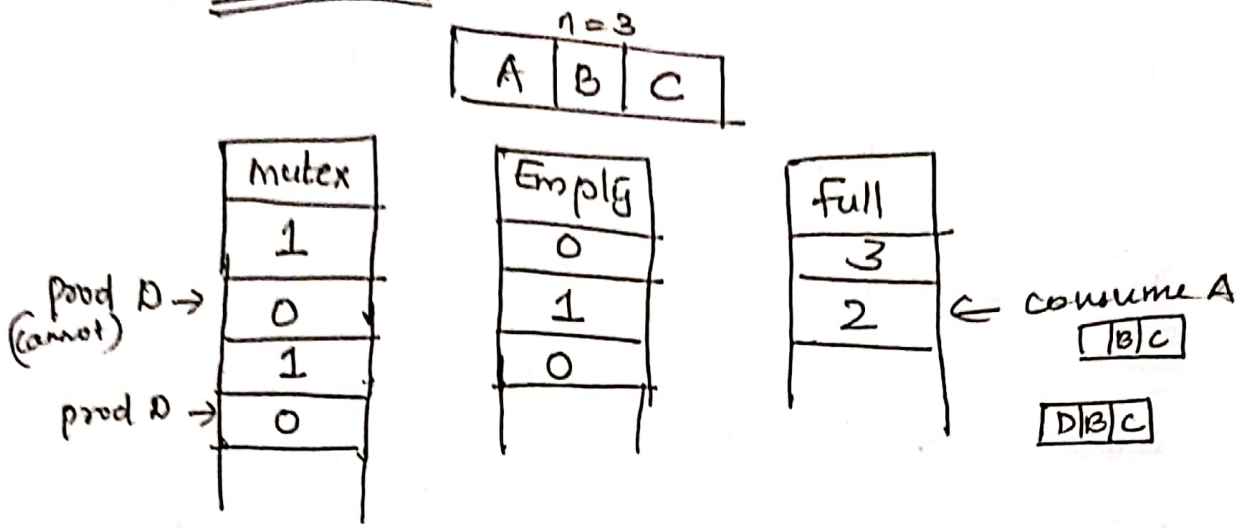
wait(full); 1→0.

wait(mutex); 0<=0 (T) · doesn't enter CS as producer is in CS.. Consumer will be in waiting list.

∴ Mutual Exclusion is maintained.

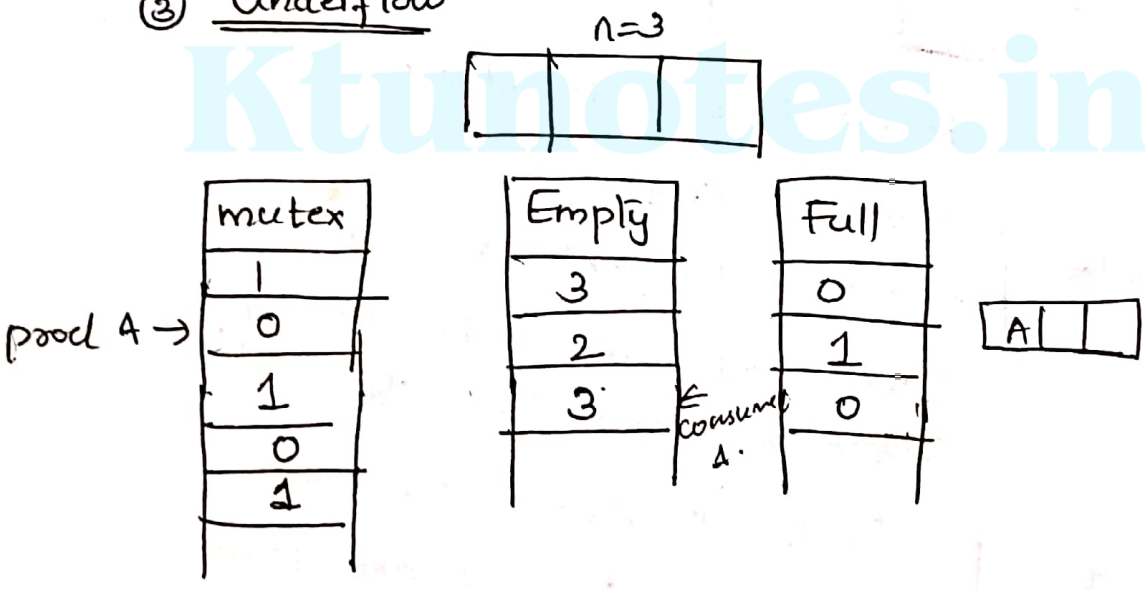
When producer finishes CS, consumer is moved to CS.

① Overflow



when prod wants to prod D. it is not possible wait(empty); 0<=0 (waiting list).
~~If~~ Thus overflow cond is checked & prod D is pushed into the waiting list when consumer consumes an item only D will be moved to buffer

② Underflow



when consumer wants to consume, wait(full); 0<=0 it cannot consume. It is moved to waiting list
Thus underflow cond is checked.
If ~~some~~ producer produces an item, then consumer can consume that item.

2.) Readers - Writers Problem:

If two readers access the shared data object simultaneously, no adverse effects will result. However, if a writer & some other process access the shared object simultaneously, effects may result. This synchronization problem is referred to as the reader - writers problem.

Writer process

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

Reader process

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

- The first reader - writer problem, requires that no reader should wait for other readers to finish simply because a writer is waiting.
- The second reader - writer problem requires that if a writing is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the 1st case, writers may starve. In 2nd case, readers may starve.

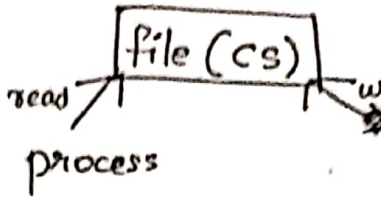
Data structures,

```
semaphore mutex, wrt;
int readcount;
```

The semaphore mutex & wrt is initialized to 1; readcount is set to 0. The semaphore wrt is common to both reader & writer processes. The mutex semaphore is use when the variable readcount is updated.

② READERS WRITERS PROBLEM.

* Reader processes & writer processes share a file.



Problems:

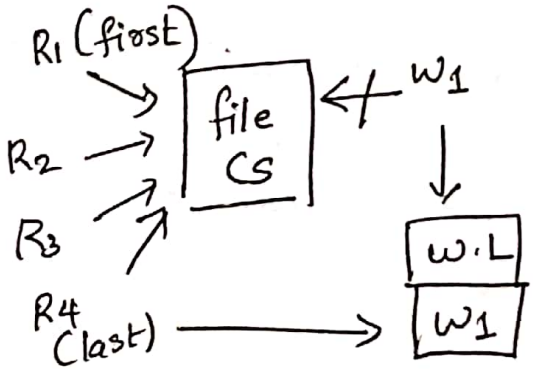
1. Allow multiple readers to access the file (R_1, R_2, R_3). (R-R)
2. When reader process is reading the file, no writer process are allowed to access (R-W)
3. When a writer process is writing the file, no reading process or writing process can access the file. (W-W, W-R)

Solution - ~~Semaphore~~.

1. Mutex = 1 } Semaphore.
2. wrt = 1. }
3. Integer variable readcount = 0.

* Reader
  - → Allow multiple readers.
  - → First reader block the writer processes.
  - → Last reader allow the waiting writer process to access the file.

* Writer
  - → Blocks all other writer process.
  - → Blocks all other reader process.

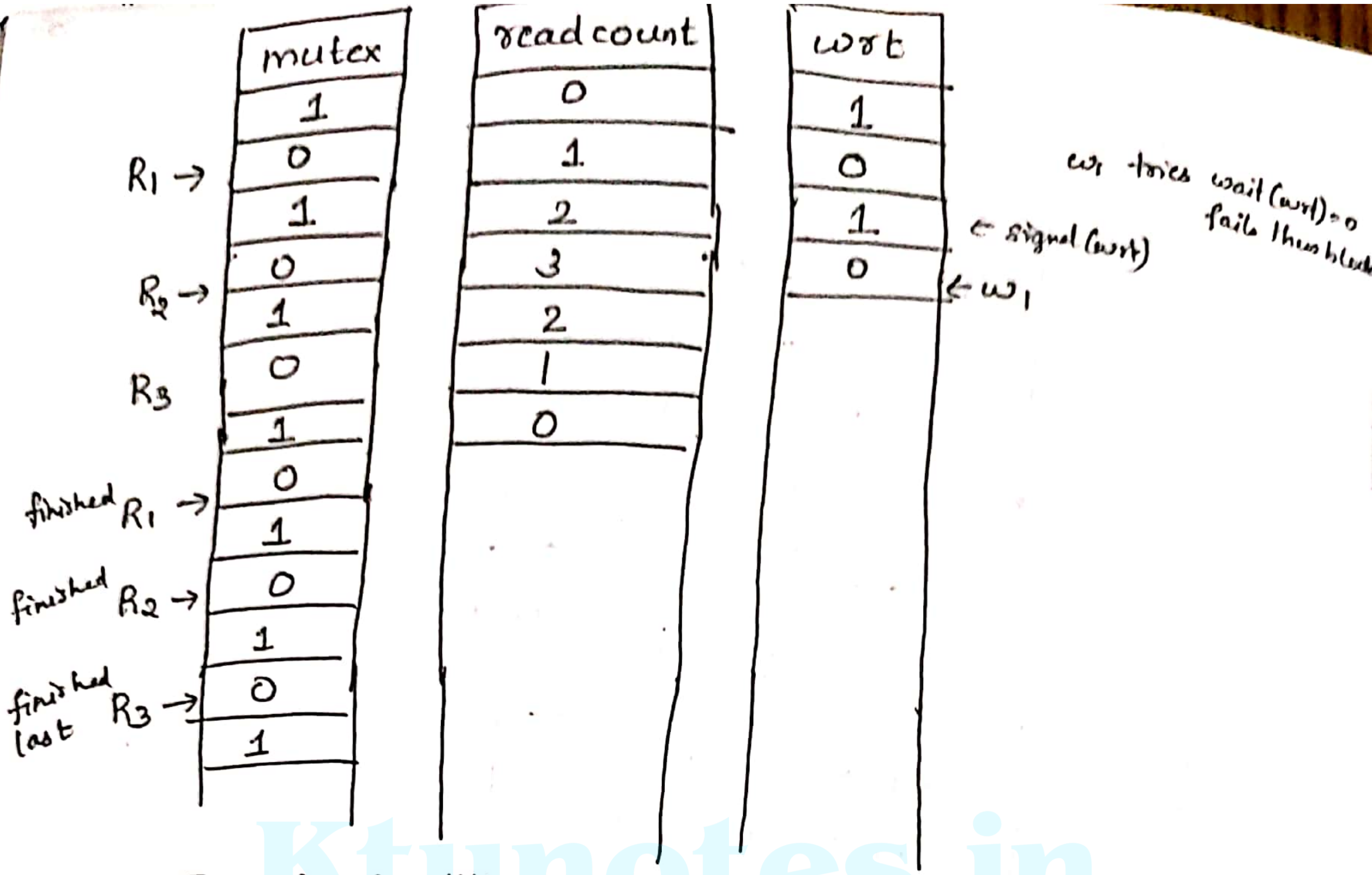


Structure of a Reader Process

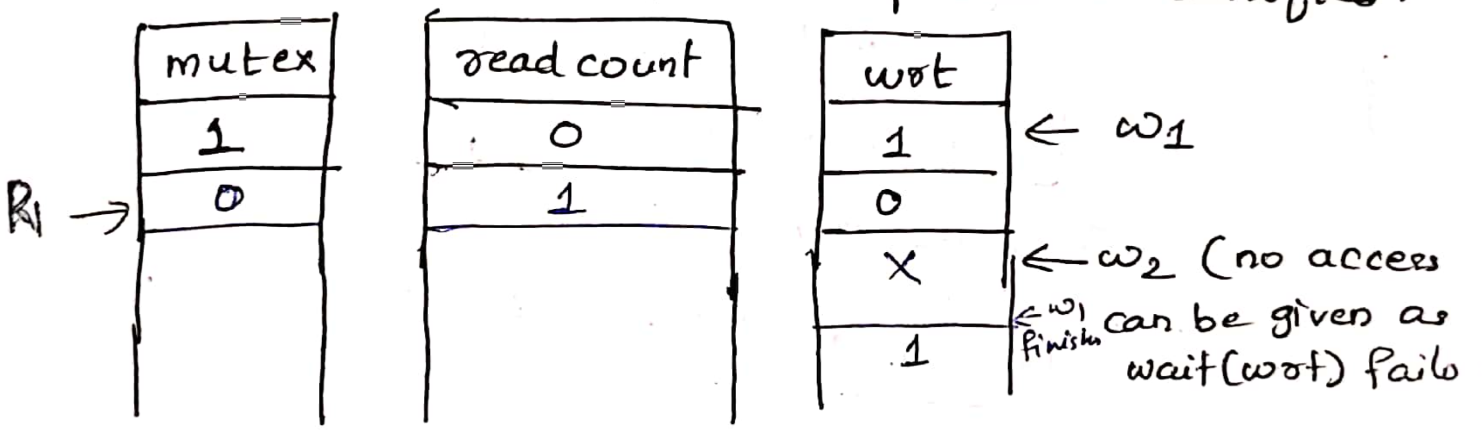
```
do {
Entry   wait(mutex);
        readcount++;
        if(readcount == 1)
            wait(wrt);
        signal(mutex);
        .....
        // Reading is performed
        CS
Exit    wait(mutex);
        readcount--;
        if(readcount == 0)
            signal(wrt);
        signal(mutex);
} while(TRUE).
```

Structure of a Writer Process

```
do {
Entry   wait(wrt);
        .....
        CS  // writing is performed
Exit    signal(wrt);
} while(TRUE);
```

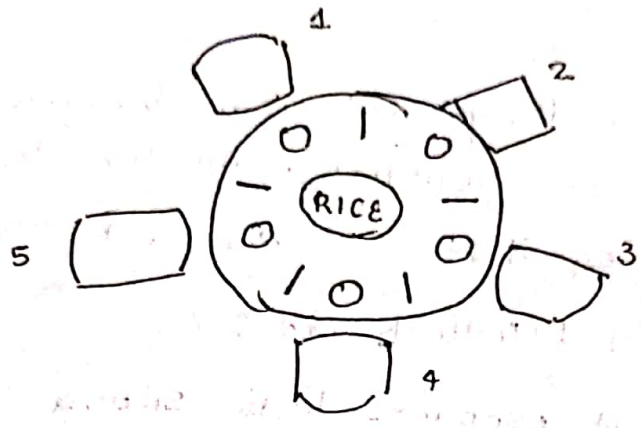


Thus 3 conditions of Reader process is satisfied.



R1 tries, when performing wait (wrt) it is blocked.
Thus the 2 conditions of writer process is satisfied.

Dining-Philosophers Problem.



Consider 5 philosophers who spend their lives thinking & eating. The philosophers share a common circular table surrounded by 5 chairs. In the center of the table is a bowl of rice & table is laid with 5 single chopsticks. A philosopher picks only one chopstick at a time. She cannot pick up a chopstick that is already in the hand of a neighbour. When a hungry philosopher has both her chopsticks at same time, she eats w/o releasing her chopsticks. When she finishes eating, she put down both of her chopsticks & start thinking again

One simple solution is to represent each chopstick by a semaphore.

A philosopher tries to grab the chopstick by executing a wait operation on that semaphore.

She releases her chopstick by executing signal operation on the semaphore.

Shared data are,

```
semaphore chopstick[5];
```

where all elements of chopstick are initialized to 1.

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

Suppose that all 5 philosophers become hungry simultaneously & each grabs her left chopstick. All the elements of chopstick now be equal to 0, a deadlock arises.

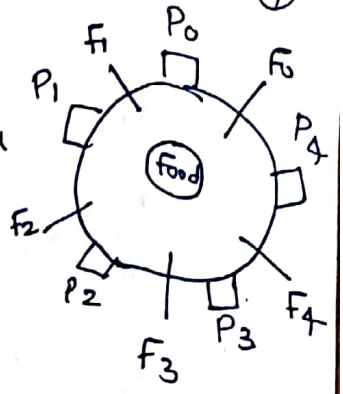
Solutions to Avoid Deadlock are;

1) Allow atmost 4 philosophers to be sitting simultaneously at the table.

2) Allow a philosopher to pick up her chopstick only if both chopsticks are available.

3) An odd philosopher picks up first her left chopstick & then her right chopstick, whereas an even philosopher pick up her right chopstick & then her left chopstick.

DINING PHILOSOPHERS PROBLEM



* Five philosophers are sitting around a circular table.
* Dinning table has 5 chopsticks & bowl of rice in the middle.
* Philosophers either eat or think.
* when philosopher wants to eat, he use 2 chopsticks.
* when he wants to think, he keeps down both chopsticks.
* Problem:

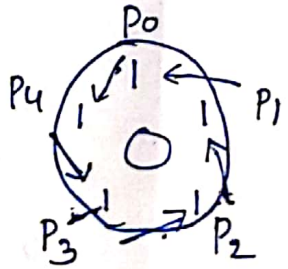
Develop an algorithm where no philosopher starves i.e, Every philosopher should eventually get a chance to eat.

⇒ chopstick[5] initialized to 1.

Structure of Philosopher i; P0

```
do {
    wait (chopstick[i]);          F0
    wait (chopstick[(i+1)%5]);    (0+1)%5 = 1, F1
        // Eat
    signal (chopstick(i));        F0
    signal (chopstick((i+1)%5));  F1
        // Think;
} while (true);
```

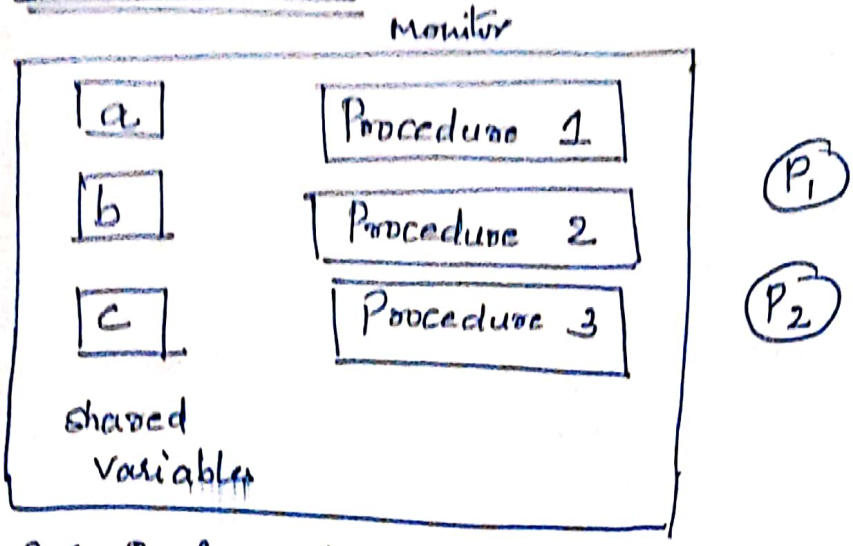
release locks.



P0 waits until P1 releases, P1 waits for P2 to release, P2 waits for P3 to release.

Everyone is in deadlock state.

MONITORS



P_1 & P_2 has shared variables a, b, c. We need to solve CS problem.

Only one process can enter monitor.

When P_1 enter monitor, P_2 must wait.

P_1 can access the shared variables only though procedures.

If proc-1 is get(a) & proc-2 is set(a).

Then P_1 must first use proc-1 to get a & P_1 must then use proc-2 to modify value of a.

when P_1 leaves only P_2 can enter monitor.

* Monitor is a module that contains :-
  - Shared data.
  - Procedures that operates on the shared data

Syntax of Monitors

```
monitor
{
    condition variables;        // vars operates using 2 opns:
                                   wait(), signal().
                                   ↓
                                blocked
      variables;
    Procedures 1
      {  =
      }
    Procedure 2
      {  =
      }
}
```

DEADLOCK

Process require resource, if not available then moves to waiting state. The resources that are requested are held by waiting process.

System Model

Set of resources → Logical resources [files, semaphore]
→ Physical resources [CPU, memory, printer etc]

A process must request a resource before using it must release the resource after using it.

The no. of resources requested may not exceed the total no. of resources available in the system.

Normal mode of operation :

1) Request : If the request cannot be granted immediately ie, the resource is being used by another process, then the requesting process must wait until it can acquire the resource.

2) Use : The process can operate on the resource ie, if the resource is a printer, the process can print on the printer.

3) Release : The process releases the resources.

Deadlock Characteristics

1) Necessary Conditions.

Deadlock condition hold if all the four conditions hold simultaneously :

(i) Mutual Exclusion.

- Atleast one resource must be held in a non-sharable mode. ie, if only one process at a time can use the resource.
- If another process requests that resource, the requesting process must be delayed until the resource has been released.

(ii) Hold and wait.

- A process must be holding atleast one resource & waiting to acquire additional resources that are currently being held by other processes

(iii) No Preemption.

- Resources cannot be preempted ie, a resource ~~can~~ can be released only voluntarily by the process holding it, after that process has completed its task.

(iv) Circular wait.

- A set $\{P_0, P_1, P_2, \ldots P_n\}$ of waiting process must exist such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2, \ldots P_{n-1}$ is waiting for a resource that is held by P_n and P_n is waiting for a resource that is held by P_0.

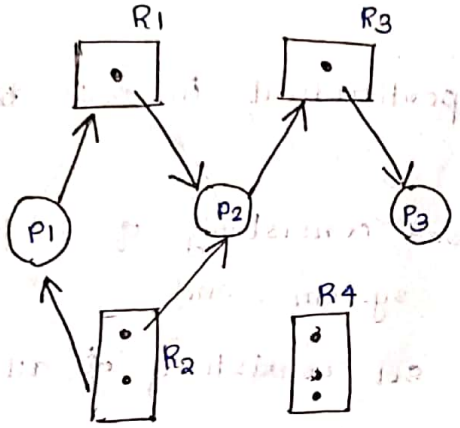
2) Resource Allocation Graph.

- Deadlocks can be described in terms of directed graph called a system resource-allocation graph.
- The graph consists of a set of vertices V & a set of edges E.
- The set of vertices V is partitioned into 2 different types of nodes:

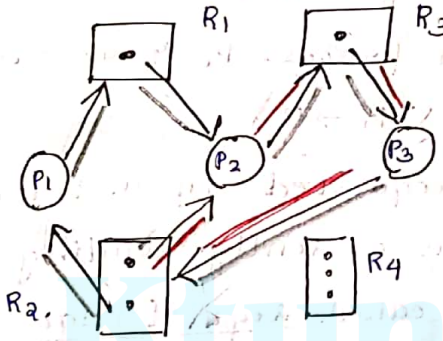
V
R

- $P = \{P_1, P_2, \ldots P_n\}$, the set consisting of all the active process in the system. and
- $R = \{R_1, R_2, \ldots R_m\}$, the set consisting of all resource types in the system.
- A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$.
  ie, The Process P_i requested an instance of resource type R_j & is currently waiting for that resource. It is called request edge.
- A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$.
  ie, An instance of resource type R_j has been allocated to process P_i. It is called Assignment edge.
- Process as circle, resource as square.
- Since request type R_j has more than one instance, we represent each such instance as a dot within square.
- When Process P_i requests an instance of resource type R_j, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled the request edge is transformed to assignment edge. When the process no longer needs access to the resource it releases the resources, & as a result assignment edge is deleted.

- If the graph contains no cycle, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies a deadlock has occured



[(a) Resource Allocation graph]



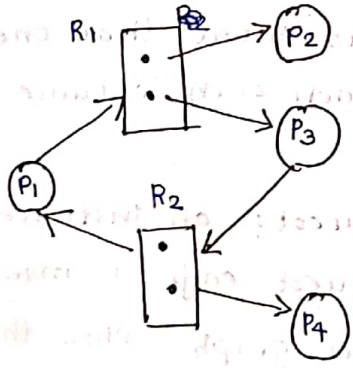
[(b) Resource Allocation graph with deadlock]

- Suppose that P_3 requests an instance of R_2. Since no resource instance is currently available, a req-edge $P_3 \rightarrow R_2$ is added to graph.
- Two minimal cycles exists,

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$.

- ✓ P_1, P_2, P_3 are deadlocked.
- ✓ P_2 is waiting for R_3 which is held by P_3.
- ✓ P_3 is waiting for either P_1 or P_2 to release R_2.
- ✓ P_1 is waiting for P_2 to release R_1.



[(c) Resource allocation graph with a cycle but no deadlock]

- A cycle has occured,

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$.

- There is no deadlock.
- P_4 may release its instance of R_2.
- That resource can then be allocated to P_3, breaking the cycle.

Methods of Handling Deadlocks

We can deal with the deadlock problem in one of three ways:

1. Can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
2. Can allow the system to enter a deadlock state, detect it & recover.
3. Can ignore the problem altogether, & pretend the deadlocks never occur in the system.

To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock avoidance scheme.

- Deadlock Prevention is a set of methods for ensuring that atleast one of the necessary conditions cannot hold.
- Deadlock Avoidance requires that the OS to give additional information about which resources a process will request & use.

Deadlock Prevention.

Ensure that atleast one condition among 4 doesn't hold:

1. Mutual Exclusion.

- Mutual exclusion condition must hold for non-sharable resources.
- Eg: A printer cannot be simultaneously shared by several processes.
- Sharable resources do not require mutually exclusive access & thus cannot be involved in a dead lock.

- If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- A process never needs to wait for a shasable resource.

2. Hold and wait.

- Whenever a process requests a resource, it does not hold any other resources.
- One protocol that can be used requires each process to request & be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resource only when the process has none. A process may request some resources & use them. Before it can request any additional resources, however it must release all the resources that it is currently allocated

we consider a process that copies data from tape drive to a disk files, sort the disk files & then prints the results to a printer. If all the resources must be requested at the beginning of the process, then the process must initially request the tape drive, disk file & printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the tape drive & disk file. It copies from the tape drive to disk, then releases both the tape drive & disk file. The process must then again request the disk file & printer. After copying the disk file to the printer, it releases these 2 resources & terminates.

3. No Preemption.

$P_i \rightarrow R_j \rightarrow P_j$

- P_i wait until P_j releases R_j

- If a process is holding some resources & requests another resources that cannot be immediately allocated it, ie, the process must wait, then all resources currently being held are preempted.
- The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that is requesting.
- Alternatively if a process requests some resources, we first check whether they are available. If they are, we allocate them. $P_i \rightarrow R_j \not\rightarrow P_j \rightarrow R_K$.
  $P_i \rightarrow R_j$ ✓ then $R_j \rightarrow P_j$.
- If they are not available, we check whether they are allocated to some other process that is waiting for additional resource. If so, we preempt the desired resources from the waiting process & allocate them to the requesting process.
  If the resources are not either available or held by a waiting process, the requesting process must wait
- while it is waiting, some of its resources may be preempted but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting & recovers any resource that were preempted while it was waiting.

4. Circular Wait.

One way to ensure that this condition never holds is to impose a total ordering of all resource types & to require that each process requests resources in an increasing order of enumeration.

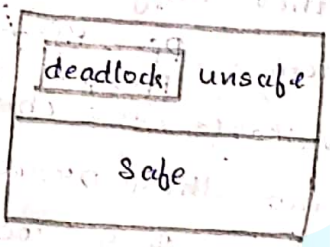
A Function $F: R \rightarrow N$, where N is the set of natural numbers.

$F(R_0) < F(R_1) < \ldots F(R_n) < F(R_0)$

Deadlock Avoidance.

- For Avoiding deadlock it is to require additional information about how resources are to be requested.
- The simplest and most useful model requires that each process declare the maximum no. of resources of each type that it may need.

1) Safe state.



- A state is safe, if the system can allocate resources to each process in some order & still avoid a deadlock.
- A system is in a safe state only if there is a safe sequence.
- A sequence of processes $< P_1, P_2, \ldots P_n >$ is a safe sequence, if for each P_i, the resources that P_i ~~can~~ ~~still~~ requests can be satisfied by the currently available resources plus the resources held by all P_j.
- If the resource that Process P_i needs are not immediately available, then P_i can wait until all P_j has finished. when they have finished, P_i can obtain all of its needed resources, complete its task, return its allocated resources & terminate. When P_i terminates, P_{i+1} can obtain its needed resource & so on. If no such sequence exists, then system is in unsafe.

- A safe state is not a dead lock state.
- A deadlock state is an unsafe state
- Not all unsafe states are dead locks.

Problem.

Total : 12 magnetic tape drives.

	Max. Need	Current Need (Allocated)	Needs.
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7.
		9	

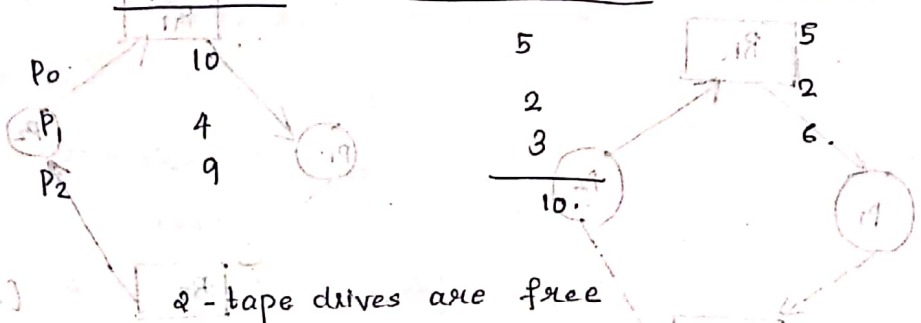
3-tape drives are free.

$\langle P_1, P_0, P_2 \rangle$ - safe state.

First P_1 is allocated, then when it releases is have 5 availdl. Then P_0 can get all its tape drives & return them, now it may have 10 available & finally P_2 could get all 10 & it may return as 12.

Suppose if P_2 requests & is allocate 1 tape drive.

	Max. Need	Current Need	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6.
		10.	



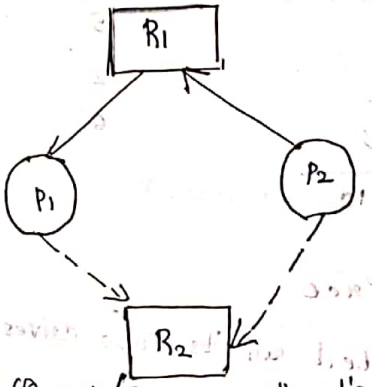
8 - tape drives are free

Here, P_1 can ~~only~~ be allocated all its tape drives. when it returns, it has 4. Since P_0 needs 5 it cannot satisfy the need of P_0 or P_2. Thus a dead lock arises.

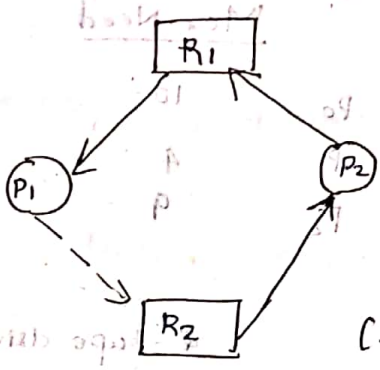
2) Resource Allocation Graph.

A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in future. This edge resembles a request edge in direction, but is represented by dashed lines. When process P_i requests resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to request edge. Similarly, when a resource R_j is released by P_j, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$. Suppose that process P_i requests resource R_j. The resource can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in resource allocation graph.

If no cycle exists, then the allocation of resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.



(Fig-1). (Resource allocation graph for deadlock avoidance)



(fig-2)

(Unsafe state in resource allocation graph)

[Fig-1] - Suppose that P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this action will create a cycle in the graph.

[Fig-2] - A cycle indicates that the system is unsafe. If P_1 request R_2 & P_2 requests R_1, then a deadlock will occur.

Banker's Algorithm.

↳ Available [j] = k

- k instance of resource type j available.

↳ Max [i,j] → Pi.

- Pi require atmost k instance of resource type R

↳ Allocation [i,j] = k.

- Process Pi is currently allocated k instance of resource type Rj.

↳ Need [i,j]

- Process Pi need k more instance of resource type Rj to complete its task.

Need [i,j] = Max [i,j] − Allocation [i,j].

Safety Algorithm.

1) Let work & finish be vectors of length m & n resp.

Initialize work := Available &

Finish [i] := false, for i = 1, 2, ... n.

2) Find an i such that both.

(a) Finish [i] = false

(b) Need i ≤ work.

If no such i exists then go to step 4.

3) work := work + Allocation i.

Finish [i] := true

Go to step 2.

4) If Finish [i] = true for all i, then the system is in a safe state.

Resource Request Algorithm.

1.) If Request ≤ Needi, go to step 2.

Otherwise raise an error condition, since the process has executed its maximum claim.

2.) If Requesti ≤ Available, go to step 3.

Otherwise Pi must wait, since the resource are not available.

3.) Have the system pretend to have allocated the requested resource to process Pi by modifying the state as follows:

- Available := Available − Request;
- Allocation := Allocation + Request;
- Needi := Needi − Request;

Problem

1)

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Soln:

Need [Max - Allocation]

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Safety Algorithm

Need ≤ Avail

① P_0 743 ≤ 332 ✗

② P_1 122 ≤ 332 ✓

Avail = 332 + 200 = 532

③ P_2 600 ≤ 532 ✗

④ P_3 011 ≤ 532 ✓

Avail = 532 + 211 = 743

⑤ P_4 431 ≤ 743 ✓

Avail = 743 + 002 = 745.

⑥ P_0 743 ≤ 745 ✓

Avail = 745 + 010 = 755.

⑦ P_2 600 ≤ 755 ✓

∴ Avail = 755 + 302 = 1057

∴

A	B	C
10	5	7

∴ < P_1, P_3, P_4, P_0, P_2 > is safe.

b) Request P_1 comes for 102.

Request Algorithm.

① 102 ≤ 122 [Req ≤ need]

② 102 ≤ 332 [Req ≤ Avail]

③ Avail := 332 − 102 = 230 [Avail := Avail − Req]

④ Allocat = 200 + 102 = 302 [Allo := Alloc + Req]

⑤ Need = 122 − 102 = 020 [Need := Need − Req]

	Alloc	Max	Avail	Need
P_1	302	322	230	020.

Past [1]

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	2	3	0	7	4	3
P_1	3	0	2	8	2	2				0	2	0
P_2	3	0	2	9	0	2				6	0	0
P_3	2	1	1	2	2	2				0	1	1
P_4	0	0	2	4	3	3				4	3	1
	8	2	7									

① P_0 . 743 ≤ 230 ✗ .

② P_1 020 ≤ 230 ✓

230 + 302 = 532 .

③ P_2 ~~020~~ 600 ≤ 532 ✗ .

④ P_3 011 ≤ 532 ✓

532 + 211 = 743 .

⑤ P_4 431 ≤ 743 ✓

743 + 002 = 745 .

⑥ P_0 . 743 ≤ 745 ✓

745 + 010 = 755 .

⑦ P_2 600 ≤ 755 ✓

A B C

755 + 302 = 1057

∴ < P_1, P_3, P_4, P_0, P_2 > is safe

Ob Req. P_4 (3 3 0) & Req . P_0 (0 2 0) comes .

Req. Algorithm .

① 330 ≤ 431 ✓ 330 ≤ 230 ✗ .

② 020 ≤ 743 ✓ 020 ≤ 230 ✓ .

Avail = 230 − 020 = 210

All = 010 + 020 = 030 .

Need = 743 − 020 = 723 .

	Allocation			Need	Avail
	A	B	C	A B C	A B C
P_0	0	3	0	7 2 3	2 1 0
P_1	3	0	2	0 2 0	
P_2	3	0	2	6 0 0	
P_3	2	1	1	0 1 1	
P_4	0	0	2	4 3 1	

① P_0 723 ≤ 210 ✗

② P_1 020 ≤ 210 ✗

③ P_2 600 ≤ 210 ✗

④ P_3 011 ≤ 210 ✗

⑤ P_4 431 ≤ 210 ✗

Undo the changes, since it is unsafe

So change as in Past [1]

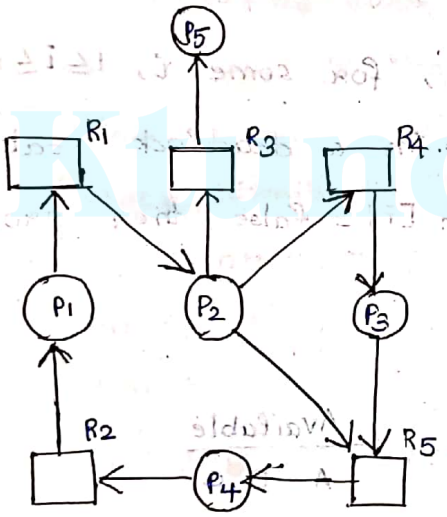
Deadlock Detection.

Determine whether deadlock has occured.

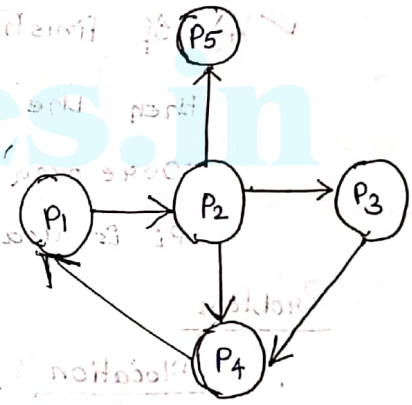
1.) Single Instance of Each Resource Type.

If all the resources have only a single instance, then we can define a deadlock detection algorithm. called a wait-for graph.

An Edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if & only if the corresponding resource-allocation graph contains two edges $P_i \rightarrow R_q$ & $R_q \rightarrow P_j$ for some resoura R_q.



(a) Resource-allocation graph



(b) Corresponding wait-for graph.

2.) Several Instance of a Resource Type.

The wait-for graph scheme is not applicable to a resource allocation system with multiple instance of each resource type.

Deadlock detection Algorithm

1.) Let work & finish be vectors of length m & n respectively.

Initialize work := Available, for i = 0, 1, 2, ...,

If Allocation ≠ 0, then finish [i] = false

otherwise

finish [i] = true.

2.) Find an index i such that both:

(a) finish [i] = false.

(b) Request$_i$ ≤ work.

If no such i exists, go to step 4

3.) work := work + Allocation.

finish [i] := true, go to step 2.

4.) If finish [i] = false, for some i, $1 \le i \le n$, then the system is in a deadlock state. Moreover, if finish [i] = false, then process P_i is deadlocked.

Problem.

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

req ≤ avail

P_0. 000 ≤ 000 ✓

000 + 010 = 010.

P_1 202 ≤ 010 ×.

P_2 000 ≤ 010 ✓

010 + 303 = 313.

P_3 100 ≤ 313 ✓

313 + 211 = 524

P_4. 002 ≤ 524 ✓

524 + 002 = 526

P_1 202 ≤ 526

526 + 200 = 726

<P_0, P_2, P_3, P_4, P_1> ∴ finish [i] = true

Change P_2 <001>.

P_0 000 ≤ 000 ✓

000 + 010 = 010.

P_1 202 ≤ 010 ×.

P_2 001 ≤ 010 ×.

P_3 100 ≤ 010 ×.

P_4 002 ≤ 010 ×

Recovery from Deadlock

1) Process Termination

✓ Abort all deadlocked processes.

- This method clearly will break the deadlock cycle, but at a great expense.

✓ Abort one process at a time until deadlock cycle is eliminated.

- This method incurs considerable overhead, since after each process is aborted, a deadlock detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on the printer, the system must reset the printer to a correct state before printing the next job.

Criteria for process selection :

✓ what the priority of the process is.

✓ How long process was computed.

✓ How many & what type of resource used.

✓ How many more resources needed.

✓ How many process will need to be terminated.

2) Resource Preemption

Issues regarding resource preemption:

✓ Selecting a victim.

- Which resources & which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost

✓ Roll back.

- If we preempt a resource from a process, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state & restart it from that state.

✓ Starvation.

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation arises.