Conditional variables

→ The condition variable is a synchronization primitive that lets the thread wait until a certain condition occurs.

→ Works with a lock.

→ Thread can enter a wait state only when it has acquired a lock. When a thread enters the wait state it will release the lock and wait until another thread notifies that the event has occurred.

→ Once the waiting thread enters the running state it again acquires the lock immediately and starts executing.

eg:- P_1

↓

lock critical section

↓

condition not true?

↓

wait on condition

↓

(P_1 is blocked here) [Goes to waiting state]

↓

condition becomes true

↓

access shared resources

↓

unlock critical section

P_2

↓

lock critical section

↓

update condition

↓

signal condition [notifying P_1]

↓

unlock critical section

→ In above example first P_1 enters and acquires lock of ~~critical~~ critical section.

→ Then it goes to waiting state where it waits for a condition to occur leaving the CPU and unlocking the critical section then P_2 executes and enters and acquires lock of critical section.

→ P_2 then signals or notify the process P_1 that condition has occurred and unlocks the critical section and is terminated.

→ P_1 then again occupies CPU and enters critical section and access shared resources and then unlocks critical section and is terminated.

{After P_2 calls signal, P_1 is moved to ready queue but it still needs to reacquire the mutex or lock before ~~continuing~~ continuing}

** → Conditional variable is used to avoid busy waiting. (~~Conte~~ Contention is not here)

Semaphores

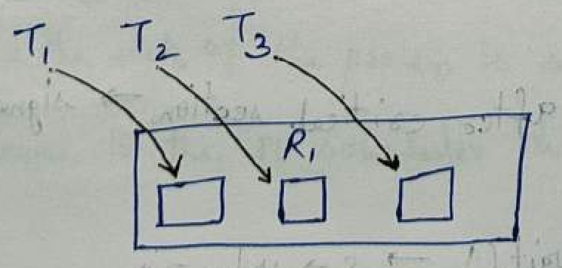
→ Synchronization method

→ An integer that is equal to number of resources.

→ Multiple threads can go and execute critical section concurrently.

→ Allows multiple program threads to access the finite instance of resources whereas mutex allows multiple threads to access a single shared resource. one at a time.

eg:- Resource R_1 has 3 instances so we take a semaphore sem which cannot be negative.



Sem = 3 (as R_1 has 3 instances)

- If a thread occupies an instance of R_1 then sem -= 1
- If a thread releases an instance of R_1 then sem += 1
- If sem = 0 then no more threads can access the resource.

(As T_1, T_2, T_3 are occupying the 3 instances of R_1)

```
wait (S)
{
  S -> value -- ;
  if (s -> value < 0)
  {
     add to s -> block list
     Block ( ) ;
  }
}
```

```
signal (s)
{
  S -> value ++ ;
  if (s -> value <= 0)
  {
     remove P from s -> block list
     wakeup (P)
  }
}
```

Semaphore S(2)
↳ 2 instances of Resource

T_1 → wait ( ) ⇒ S → value = 1

T_2 calls wait ( ) due to this s → value is decremented.

or

T_2 → wait( ) ⇒ S → value = 0

T_3 → wait( ) ⇒ s → value = -1 → Block( )

↳ This is why used condition s → value <= 0 to wakeup waiting processes

then when

T_1 → after critical section → signal( ) → s → value = 1

then

T_3 → wait( ) → s → value = 0

In code it's written like

{ wait( )
critical section
signal( ) }

If semaphore (1) ⇐ Binary semaphore

If semaphore (>1) ⇐ Counting semaphore

→ Binary semaphore : Value can be 0 or 1

(i) aka, mutex/locks

→ Counting semaphore

(i) Can range over an unrestricted domain.

(ii) Can be used to control access to a given resource consisting of a finite number of instances

→ To overcome the need for busy waiting we can modify the definition of the wait( ) and signal( ) semaphore operations.

→ When a process executes the wait ( ) operation and finds that the semaphore value is not positive it must wait.
However rather than engaging in busy waiting the process can block itself.

→ The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to waiting state. Then control is transferred to the CPU scheduler which selects another process to execute.

→ A process that is blocked, waiting on a semaphore S should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation which changes the process from the waiting state to ready state. The process is then placed in a ready queue.

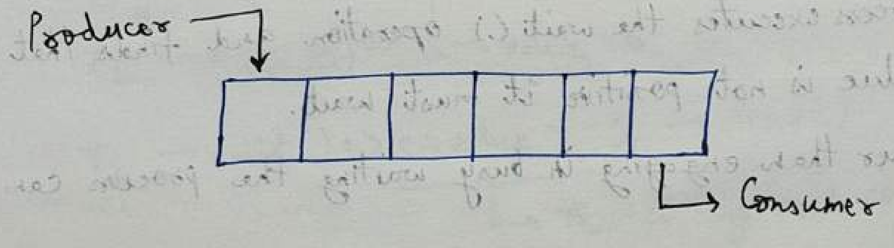
Producer – Consumer problem (Bounded Buffer problem)

→ There are 2 types of thread here

- Producer thread
- Consumer thread.

→ Producer produces any data and consumer consumes the data produced by producer.

→ There is a critical section (buffer) in which producer produces data and places it in buffer and consumer consumes data from buffer



→ So there are n slots in buffer in which producer produces and puts data in buffer while consumer consumes data from buffer

→ So we want that when producer produces data and puts it in buffer (critical section) we don't want consumer to access data or consume data at that time.

And when consumer is consuming data we don't want producer to produce data and put data in buffer.

• Problem

→ Buffer is a shared resource so we don't want any data inconsistency.

→ So we make access to buffer in a synchronize manner to avoid data inconsistency.

→ Synchronize between Producer thread and Consumer thread.

→ When buffer is full we don't want producer to produce more data as it will only lead to wastage of CPU cycles.

So producer must not insert data when the buffer is full.

→ Similarly consumer must not pick/remove/consume data when the buffer is empty.

{ As the buffer length or size is finite hence it is also called as bounded buffer problem. }

↳ Need to synchronize as buffer is bounded

• Solution

→ Solution is given by semaphores.

① m , mutex → Binary semaphore

m → mutex

Used to acquire lock on buffer.

② empty → Counting semaphore

Initial value is n (as at start all slots in buffer is empty)

Used to track empty slots

③ full → Counting semaphore

Used to track filled slots

Initial value is 0.

- Producer

```
do{
    wait(empty);  // wait until empty > 0 then, empty → value--

    wait(mutex);
    // Critical section, add data to buffer
    signal(mutex);

    signal(full);  // increment full → value

} while(1)
```

· Consumer

```
do {

    wait(full);   // wait until full>0, then full--

    wait(mutex);
    // remove data from buffer
    signal(mutex);

    signal(empty);   //increment empty

} while(1)
```

In general
wait() does semaphore = semaphore − 1
whereas signal does semaphore = semaphore + 1

Case-I : General flow

→ First producer is executed and wait(empty) is called.

Since value of empty in initially n which is greater than zero hence

$$empty = empty - 1$$

So producer didn't wait as empty > 0

→ Then wait(mutex) is executed and producer acquires the lock of critical section. Then it adds data to buffer.

→ After adding data the producer releases the lock by wait(mutex). Then signal(full) is executed.

$$full = full + 1$$

Hence value of full is 1 as initial value was zero.

→ We can see wait (mutex) and signal (mutex) as this also

mutex = 1 {unlock}

mutex = 0 {lock}

wait (mutex) => mutex = mutex − 1 = 0 {locked}

signal (mutex) => mutex = mutex + 1 = 0 + 1 = 1 {unlocked}

→ Then consumer is executed as it was on wait before due to

wait (full) {as full wasn't greater than zero due to which condition was false}

Now when wait (full) is executed.

full = full − 1

→ Then wait (mutex) is executed so that consumer can acquire lock for critical section.

mutex = mutex − 1 = 0

→ Consumer consumes data in buffer (only, data in 1 slot as value of full was decremented by 1).

→ Then consumer releases the lock of critical section by signal (mutex)

mutex = mutex + 1 = 1

→ Then as consumer has consumed data so 1 more slot will be empty so signal(empty).

$$empty = empty + 1.$$

Case - II : Consumer is executed first

→ Initially $empty = n$

$full = 0$

$mutex = 1$

→ Consumer then check if there's any data in buffer by wait(full)

$$full = full - 1$$

But this will not happen as condition for $full > 0$ is false hence the consumer will wait until producer produces data and places it in buffer.

Case - III : Producer is executed till buffer it full

→ When producer keeps on producing data and consumer isn't ~~cons~~ consuming data then after a particular period of time

$empty = 0$

$full = n$

→ Then when producer is executed again wait(empty) is executed but as $empty > 0$ condition is false hence the producer will wait until consumer consumes data.

Read - write problem

- Problem

→ There are 2 types of thread here

① Reader thread ← Read

② Writer thread ← write, update

→ In this we only want one writer to write at a time as if there are multiple writer writing at the same time then it would lead to data inconsistency.

→ As for readers ~~we~~ can have multiple readers reading at the same time as they are not writing or updating data so cannot lead to data inconsistency

→ Hence to sum it up

① if >1 Readers are reading ← No issue

② if >1 writers OR 1 writer & some other thread (read/write), parallelly ← Race condition & data inconsistent

→ In case of 1 writer and 1 reader or >1 reader if a reader reads something then goes and writer writes new data or more data then reader might not have read ~~correct~~ correct data.

Thus leading to data ~~inconsistent~~ inconsistency.

eg :- reader read abcd data in buffer and goes

then writer updates the data in buffer to abzf

Hence readers read incorrect data.

→ So here our critical section is database (or a page)

→ So we need to satisfy these 3 conditions to solve this problem.

1. Only one writer at a time can write or update (mutual exclusion)

2. When one writer is writing no reader can read (mutual exclusion)

3. Many readers can read at a time.

• Solution

→ Solution is given by semaphores

1. mutex → Binary semaphore

   ↳ To ensure mutual exclusion when read count (rc) is updated.
   No two threads can modify rc at same time
   { Similar to previous aadhar center problem }

2. wrt → Binary semaphore

   ↳ Common for both reader and writer

③ readcount (rc) → Integer

↳ Initial value is zero.

Used to track how many readers are reading in critical section.

$\boxed{\text{* Note : readcount (rc) is not a semaphore}}$

- Semaphores are used to block/unblock processes via wait () and signal ().

  rc just keeps a number or count of readers.

  It doesn't block or control process scheduling

- We only need to protect rc using a semaphore

- Writer

```
do {
    wait (wrt);
  //do write operation
    signal(wrt);
} while (true);
```

→ When a writer executes the wait (wrt), it acquires the lock for critical section.

$$wrt = wrt - 1 = 0 \qquad \left\{ \begin{array}{lll} wrt = & 0 & \text{Lock} \\ & 1 & \text{unlock} \end{array} \right\}$$

→ Then it performs write operation and then releases the lock.

$$signal(wrt) \Rightarrow wrt = wrt + 1 = 0 + 1 = 1 \quad \{\text{unlock}\}$$

• Reader

```
do {
    wait(mutex);  // to mutex readcount variable
    rc ++ ;
    if (rc == 1)
    {
        wait(wrt) ; // ensures no writer can enter if there is even one reader
    }
    signal(mutex);
  // Critical section : Reader is reading
    wait(mutex);
    rc -- ;   // a reader leaves
    if (rc == 0)   // no reader is left in critical section.
    {
        signal(wrt); // writer can enter
    }
    signal(mutex);    // reader leaves
} while(1)
```

→ First wait(mutex) is executed to ~~loo~~ acquire lock as rc is global variable so can cause data inconsistency if multiple threads perform operations together.

$$mutex = mutex - 1 = 0$$

→ rc is ~~by~~ incremented as reader thread code.

→ If rc == 1 or if it's the first reader then it will check that there's no writer in critical section by executing wait(wrt). If there's a writer the reader will wait and if not then it will acquire the lock.

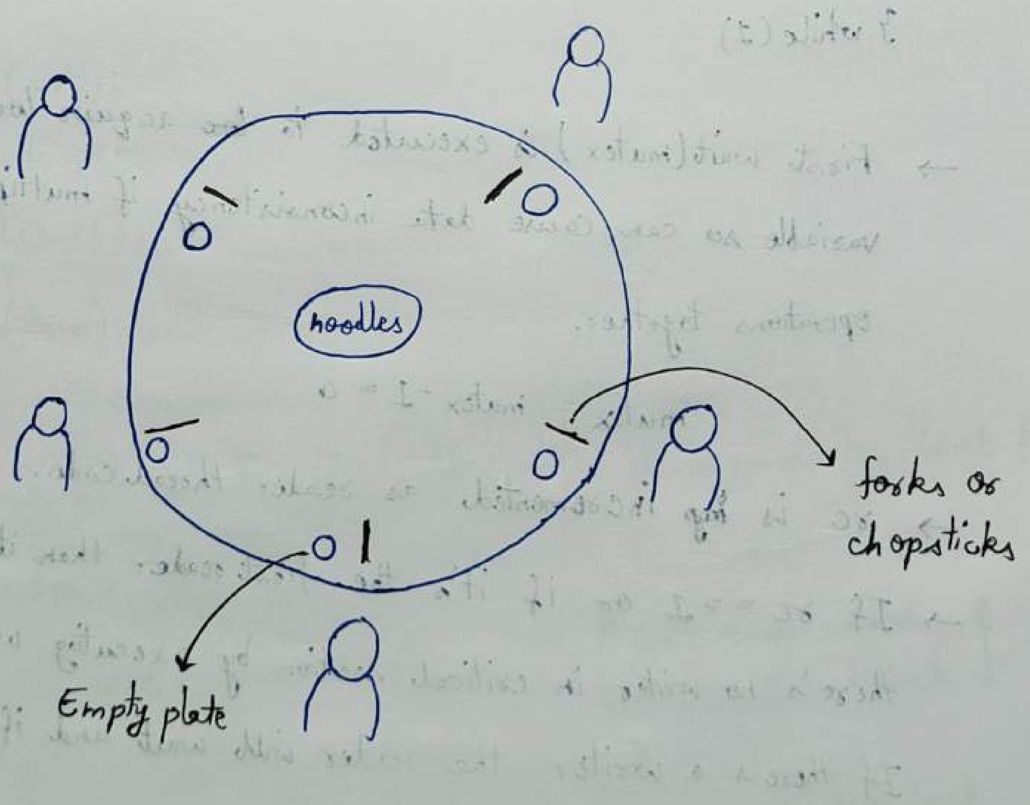
→ Then signal(mutex) is executed to release lock so that other readers can also access and increment it one by one to keep correct count of readers.

→ Then the readers read and then again acquires lock of global ~~variable~~ variable to decrement it as they are leaving (as they have performed their operation)

→ It checks condition that if rc == 0 or there are no more readers in critical section then it can release the lock of critical section such that if a writer is waiting it can enter.

→ Then it releases lock of global variable so that other readers can access it (either for incrementing or decrementing)

The Dining Philosophers problem



① 5 philosophers

② Noodles

③ 5 forks / chopsticks

→ The philosophers do not interact with each other.

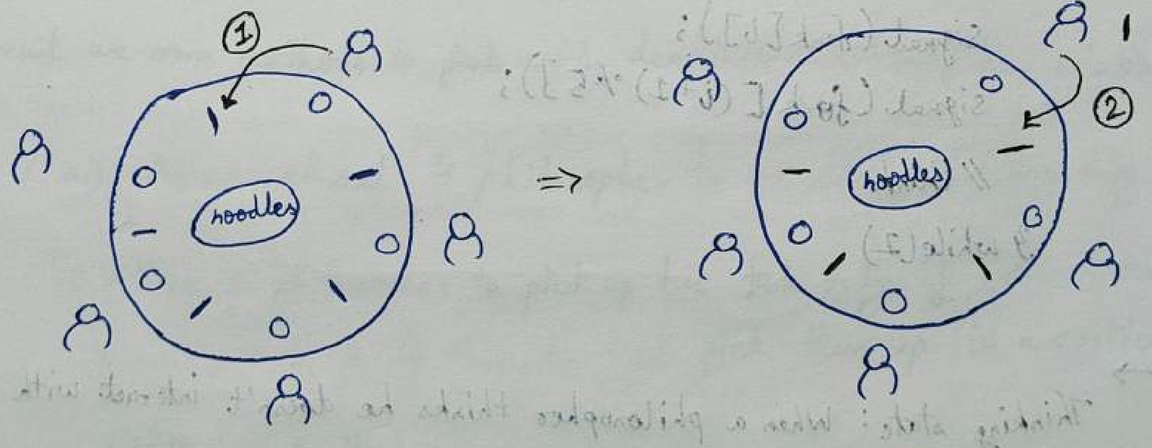
→ Philosophers spend their life just being in two states :

(a) Thinking

(b) Eating

→ They sit on a circular table surrounded by 5 chairs (1 each), in the center of table is a bowl of noodles and the table is laid with 5 single forks.

→ Each philosopher needs 2 forks to eat the noodles. Philosopher picks fork one by one.



→ We need to synchronize so that they all can eat noodles.

→ Solution is given by semaphore

(1) each fork → Binary semaphore

Semaphores fork [5] {Initialized with value 1}

wait () → fork[i] => ph[i] → acquires

release ( ) → fork[i] → fork → Release

↑ done by calling signal( )

→ One can't pick up a fork if it is already taken.

→ When philosopher has both forks at the same time, he eats without releasing forks.

```
do {
    wait(fork[i]);
    wait(fork[(i+1)%5]);

    //eat

    signal(fork[i]);
    signal(fork[(i+1)%5]);

    //think
} while(1)
```

→ Thinking state: When a philosopher thinks he doesn't interact with others.

Eating state: When a philosopher gets hungry he tries to pick up the 2 forks adjacent to him (Left and right). He can pick one fork at a time.

→ In above code the philosopher calls wait( ) to acquire lock of 2 adjacent forks.

$i = 1$ $\quad\quad$ $(i+1) \% 5 \Rightarrow 2^{nd}$

fork[1] $\quad\quad$ fork[2]

→ Then philosopher eats and then releases lock by signal( ) then enters thinking state.

→ Although the semaphore solution makes sure that no two neighbors are ~~eating~~ eating simultaneously but it could still create deadlock.

→ Suppose ~~all~~ that all 5 philosophers become hungry at the same time and each picks up their left fork then all fork semaphores would be 0.

→ When each philosopher tries to grab his right fork after acquiring left fork he will be waiting forever. (Deadlock)

→ We must use some methods to ~~pick~~ avoid deadlock and make the solution work

(a) Allow atmost 4 philosopher to be sitting simultaneously.

(b) Allow a philosopher to pickup his fork only if both forks are available and to do this, he must pick them up in a critical section (atomically)

$$\left\{ \begin{array}{l} \text{wait(fork[i]);} \\ \text{wait(fork[(i+1) \% 5]);} \end{array} \right\} \text{ critical section}$$

adding wait (mutex)

signal (mutex)

→ Then we would ~~be~~ be needed to make signal section also critical as philosopher 1 is releasing fork 1 and at the same time philosopher 2 is trying to pick fork 1.

→ Now without making it critical section it can lead to

- Inconsistent ~~fork~~ fork state

  eg :- fork marked available when it's not.

- One philosopher might think fork is available when it's still in use ~~or might think fork is busy~~ (due to race).

→ This leads to undefined or incorrect behaviour.

(C) Odd-even rule

An odd philosopher picks up his left fork first and then his right fork whereas an even philosopher picks up his right ~~fork~~ fork first then his left fork.

→ Hence only semaphores are not enough to solve this problem. We must add some enhancement rules to make deadlock free solution.

{ Producer Consumer problem, Read-write problem, Dining philosopher problem are classical synchronization problems. }