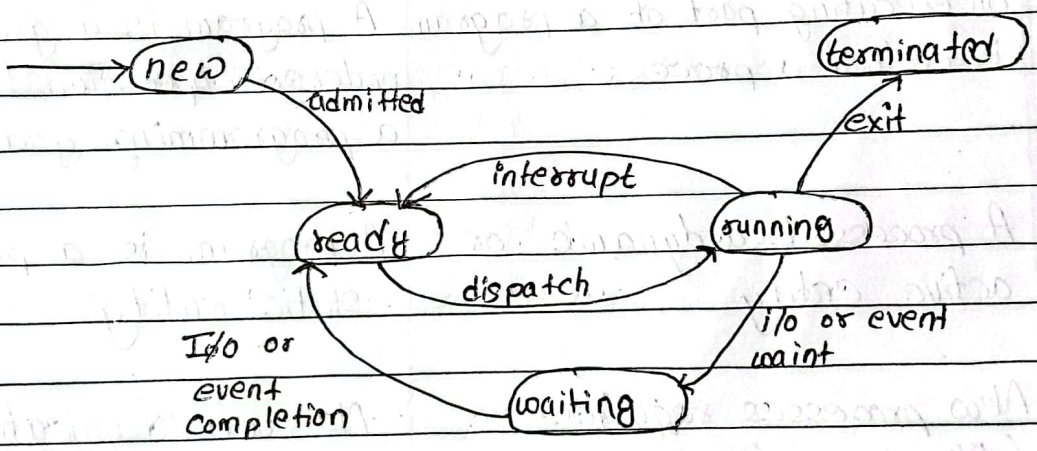
Do you think a process can exist without any state? Justify your view with the help of process state transition diagram and PCB

Resource Management	Requirement is quite high in case of process
	The program only needs memory for storage

Imp

Process states and Transition diagram

When a process executes, it passes through different states. These stage may differ in different operating system and name of these state are also not standardized. In general, a process can have one of the following five state at a time



New: A process that has just been created but has not yet admitted to the pool of executable process by OS

Ready: Process that is prepared to execute when given the opportunity. That is, they are not waiting on anything except the CPU availability.

Running : Once the process has been assigned to a process by OS scheduler, the process state is set to running and the processor executes its instructions.

Whiting : Process move into the waiting state if it needs to wait for a resource such as waiting for user input, or waiting for a file to become available

Terminated : Once the process finishes its execution, or it is terminated by the OS, it is moved to terminated state where it waits to be removed from main memory.

IMP

Process Control block (PCB)

Process control block is a data structure used in OS to store all data related information to the process. PCB is identified by an integer process ID (PID). The PCB is maintained for a process throughout its life time and deleted once process terminates. It keep all the information needed to keep track of processes.

What is PCB? Describe the field in PCB?

2022 Fall

Process ID
State
Pointer
Priority
Program counter
CPU register
I/O information
Accounting information
etc .....

fig: Process control Block

Process ID: Unique identification for each of the process in the operating system

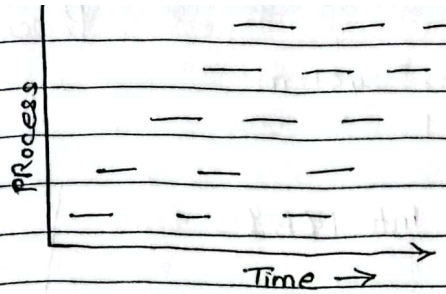
Process state: The current state of the process i.e. whether it is ready, running, waiting or whatever

Pointer: A pointer to parent process

Priority: Contain the priority number of each process

Program Counter: It is a pointer to the address of the next instruction to be executed for this process

CPU Register: Various CPU register where process need to be stored for execution for running state.



Only one program is active at once

Context Switching

This is a technique or method used by the OS to switch a process from one state to another to execute its function using CPUs in the system. When switching perform in the system, it store the old running process's status in the form of register and assign the CPU to an new process to execute its task. While a new process is running in the system, the previous process must wait in a ready queue. The execution of old process starts at that point where another process stopped it. It defines the characteristics of a multitasking OS in which multiple process shared the same CPU to perform multiple task without the need for additional processor in system. It helps to share a single CPU accross all process to complete its execution and store the system's task status. When process the reload in system, execution of process starts at the same point where there is conflicting.

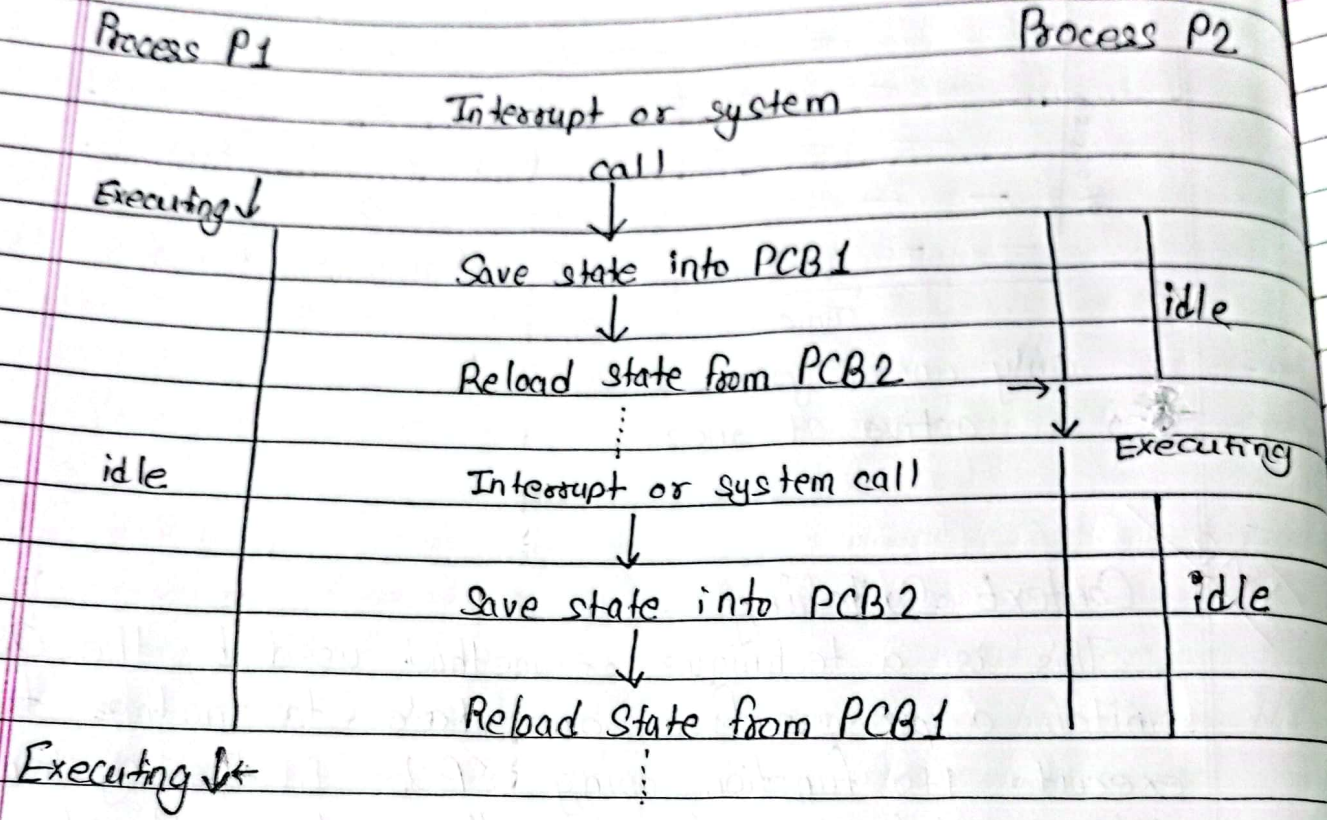


fig: Context switching from P1 to P2

2021, 2020, 15,

Imp 2019 15,

Parameter	Process	Thread
Define	It means a program is in execution	It means a segment of process
Lightweight	Not Lightweight	Lightweight
Termination Time	It takes more time to terminate	It takes less time for terminate
Creation Time	It takes more time for creation	It takes less time for creation
Communication	Communication between process need more time	Communication between thread require less time
Context Switching	It takes more time for context switching	It takes less time for context switching
Resource	Process conume more resource	Thread consume fewer resource
Treatment by OS	Different process are tread separately by OS	All the level peer threads ar treated as a single task by OS.
Memory	Process is mostly isolated	Thread share memory
aring	It does not share data	Threads share data with each others.

2.2 Thr

Kno
of
th
u
th
b
s

Benefit of using threads

↳ Threads are easy and inexpensive to create.
Thread are 10 times as faster to create than process
↳ It takes less time to terminate the thread than process
↳ Uses of thread provide concurrency within process
↳ Threads minimizing context switching time
↳ It enhances efficiency in communicate between different executing program.
↳ Utilization of multiprocessor architecture to a greater scale and efficiency.

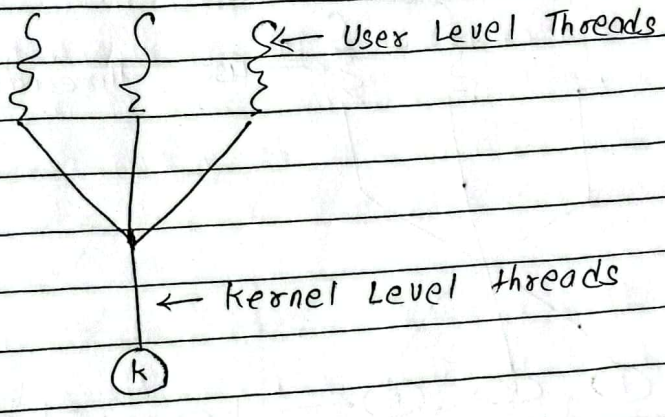
- **Threads Models** (Imp)
  1. Many to one model
  2. One to One model
  3. Many to many model

2019 Fall

2 a) What is multi-threading? Explain the different multi-threading model. What is the biggest advantage of implementing thread in a user space

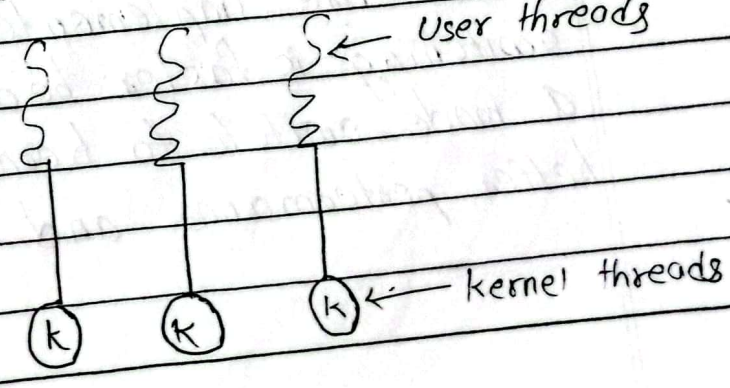
(1) Many to one model

It maps many users threads to single kernel threads. Threads management is done by thread library in user space so it is efficient but the entire process will block if a thread makes a blocking system call. Because only one thread can access the kernal at a time, multiple threads are unable to run in parallel on multiprocessor



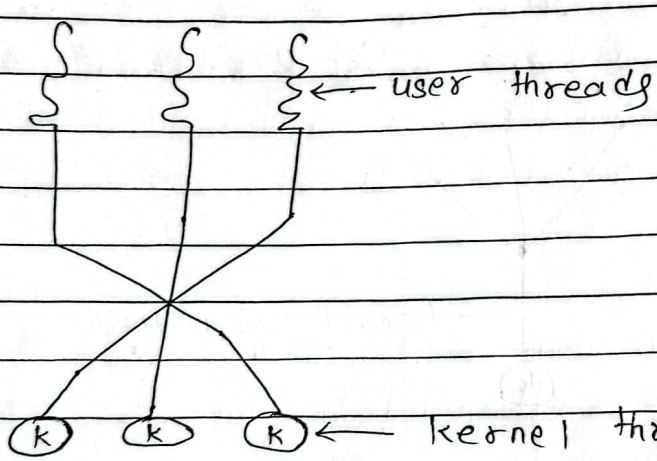
(2) One to One model

It mach each user thread to a corresponding kernel thread. It provide more concurrency than many to one model by allowing another thread to run when a thread makes a blocking system call i.e. it also allows multiple threads to run in a parallel on multiprocessor.



③ Many to Many model

It multiplex many user level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine. This model allow the developer to create as many user threads as the user wishes an the corresponding kernel threads can run in parallel on a multiprocessor.



The biggest advantage of implementing threads in a user space is that it provide faster context switching. Context switching is the process of storing and restoring the state of a thread so that execution can be resumed from the same poin later. When threads are implemented in user space, context switching is faster because it does not require a mode switch to kernel mode. This result in better performance and scalability.

2020 Fall 2b)
What is the IPC? Describe its implementation using its type

3 Interprocess Communication and Synchronization (IPC)

Inter-process communication (IPC) is a set of techniques for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computer connected by a network. IPC techniques are divided into methods for message passing, synchronization shared memory and remote procedure calls (RPC)

IPC enables one application to control another application and for several application to share the same data without interfering with one another.

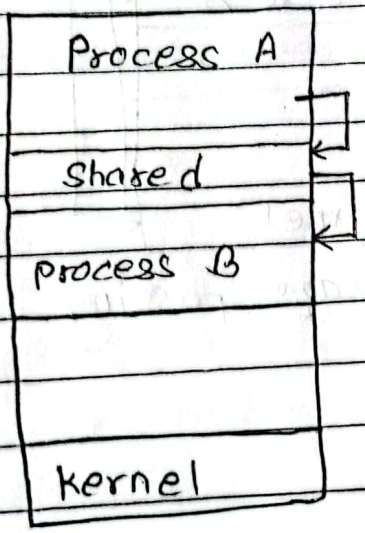
Co-operating Process: A process is independent if it can't affect or be affected by another process. A process is co-operating if it can affects others or be affected by other process. Any process that shares data with other process is called cooperating process. There are many reason for providing an environment for process cooperation.

* Information sharing
* Computation speedup
* Modularity
* Convenience

Communication Model

There are two fundamental ways of IPC

(a) Shared Memory
- An area of memory shared among the process that wish to communicate.
- The communication is under the control of the user's process not the operating system
- A major issue to provide mechanism that will allow the user processes to synchronize their action when they access shared memory.
- System calls are required only to establish shared memory regions. Once shared memory is established no assistance from the kernel is required, all access are treated as routine memory access.



(b) Message Passing

- Mechanism for process to communicate and to synchronize their actions
- Message passing is useful for exchanging the small amount of data since no conflict need to be avoided
- Easier to implement than shared memory
- Slower than that of shared memory as message passing system are typically implemented using system call which requires more time consuming task of kernel intervention.

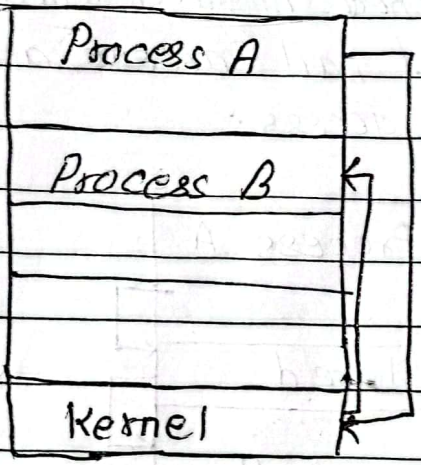


Fig. Message passing

2018 Spring. 2017 Fall 2 a)

2 a) What is a race condition and mutual exclusion
2 a) What is the causes of race condition
~~Who~~

Race condition is the situation where two or more processes are reading and writing some shared data and final results depends on who runs precisely when.

To avoid race condition we need Mutual Exclusion Mutual exclusion is someway of making sure that if one process is using a shared variable or file, the other processes will be ~~exec~~ excluded from doing the same things.

Critical Rigeon or Section is the part of program in which shared memory is updated.

The causes of race condition are as follows:

- Lack of proper thread synchronization
- Incorrect assumption of a sequence for process execution
- Multi threading overheads

What are the rule for avoiding Race condition?

1. No process running outside its critical region may block other processes
2. No two process may be simultaneously inside their critical regions
3. No assumption may be made about speeds or the number of CPU
4. No process should have to wait forever to enter its critical region

Process Synchronization

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency require mechanism to ensure the ordely execution of cooperating process

Process Synchronization is the task of coordinating the execution of process in a way that no two processes can have access to the same shared data and resources.

1. What are the techinque for avoiding Race Condition
2. Describe critical region? List out how it can be solved through mutual Exclusion.
3. What is mutual Exclusion? What are the different techinques to achieve mutual exclusion?

(1) Disabling Interrupts
(2) Lock Variable
(3) Strict Alteration
(4) Peterson's Solution
(5) TSL instruction
(6) Sleep and Wakeup
(7) Semaphores

(1) Disabling Interrupts

The simplest solution is to have each process disable all interrupt just after entering its critical section and ~~reables~~ reenables ~~tem~~ them before leaving it. The CPU is only switched from process to process due to clock or other interrupts and with interrupt turned off the CPU will not be switched to another process. So, once a process has disabled interrupts it can examine and update the share memory without fear that any other process will intervene (disturb).

This approach is generally unattractive because it gives user process to turn off interrupts. If the system is multiprocessor, disabling interrupts affect only the CPU that executed the disabled instruction and other ones will continue running and can access the shared memory.

(2) Lock variable

Considered having a simple single shared variable called lock which is initially 0. When a process wants to enter its critical regions, it first tests the value of lock variable. If lock is 0, the process sets it to 1 and enters the critical region.

If lock is already 1, the process just ~~want~~ wait until it becomes 0. Thus, 0 means that no process is in its critical region and 1 means that some process is in its critical region

Drawback:

Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs and sets the locks to 1. When the first process run again, it will also set the lock to 1, and two process will be in their critical region at a same time

```
while (lock != 0)
    Lock = 1;
        Critical section
    Lock=0;
```

```
do {
    acquire lock
        Critical section
    release lock
        Remainder Section
} while (true);
```

③ Strict Alteration

Process 1

```
while (TRUE){
    while (turn !=0);
    critical_region();
    turn = 1;
    noncritical_region();
}
```

Process 2

```
while (True){
    while (turn != 1);
    critical_region();
    turn = 0;
    noncritical_region();
}
```

Initially the integer variable "turn" is 0. The "turn" variable keeps the track of whose turn it is to enter the critical section and examine or update the shared memory. Initially process 1 inspects turn find it to be 0 and enters its critical section. Process 2 also find it to be 0 and therefore sits in a tight loop continually testing turn to see when it become 1.

Continally testing a variable until some value appear is called busy waiting and it should be avoided since it waste CPU time. When process 1 leaves the critical section, it set "turn" to 1 to allow process 2 to enter its critical section. This way no two process can enter critical section simultaneously.

Drawback:

Taking turn is not a good idea when one of the process is much slower than other. This situation require that two processes strictly alternate in entering their critical region.

2017 Explain about Peterson's algorithm

(4) Peterson's Algorith / Solution

```
# define FALSE 0
# define TRUE 1
# define N 2                    /* number of process
int turn;                       // whose turn is it?
int interested[N];              // all values initially 0(FALSE)
void enter_region (int process)  // process is 0 or 1
{
    int other;                  // no. of other process
    other = 1 - process;        // opposite of process
    interested[process] = TRUE; // show that you are interested
    turn = process;             // set flag
    while (turn == process && interested[other] == TRUE)
}

void leave_region (int process) //process : who is leaving
{
    interested[process] = False; //indicate departure
                                 from critical region
}
```

fig. Peterson's solution for achieving mutual exclusion

```
#define FALSE 0
#define TRUE 1
#define N 2
int turn;
int interested [N];
void enter_region (int process)
{
  int other;
  other = 1- process;
  interested [process] = TURE;
    turn = process;
  while (turn == process && intrested [other] == TRUE);
}

void leave_region (int process)
{
  interested [Process] = FALSE;
}
```

Initally neither process is in critical region. Now process 0 call enter_region. It indicate its interest by setting its array element and set turn to 0. Since process 1 is not interested, enter_region returns immediately. If process 1 now calls enter_region, it will hang there until interested [0] goes to FALSE, i.e when process 0 calls leave_region to exit the critical region.

Now consider the case that both processes call enter_region almost simultaneously. Both will store

their process number in turn. Whichever store is done last is the one that counts and first one is lost. Suppose that process 1 store last so turn is 1. When both processes comes to the while statement, process 0 execute it zero time and enter its critical region. Process 1 loops and does not enter its critical region.

2018 (5) Show how mutual exclusion can be achieved using TSL (Test and set Lock)

It is special machine instruction used to avoid mutual exclusion. It works as follows:

It reads the contents of the ~~word~~ memory word LOCK into register RX and then store a nonzero value at the memory address LOCK. The operations of reading the word and storing into it are guaranteed to be indivisible no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

```
enter_region
  TSL REGISTER, LOCK  ||copy Lock to register and set Lock to 1
  CMP REGISTER, #0    || was LOCK zero?
  JNE enter_region    ||if it was non zero, LOCK was set so loop
  RET                 ||return to caller; critical region entered
```

```
leave_region:
    MOVE LOCK, #0        //store a 0 in LOCK
    RET                  //return to caller
```

One solution to the critical region problem is now straightforward. Before entering its critical region, a process calls enter_region which does busy waiting until the lock is free; then it acquire the lock and returns. After the critical region the process call leave_region which store a 0 in LOCK. As with all solution based on critical regions, the process cheats, the mutual exclusion will fail.

The above techniques achieves the mutual exclusion using busy waiting. Here while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.

Mutual Exclusion with busy waiting just check to see if the entry is allowed when a process wants to enter its critical region, if the entry is not allowed the process just sits in a tight loop waiting until it is

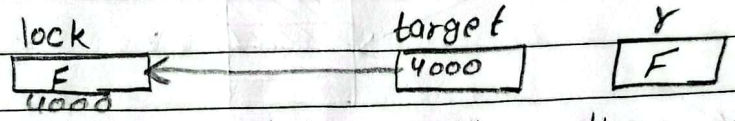
(i) This approach waste CPU time
(ii) There can be an unexpected problem called priority inversion problem.

TSL Algorithm

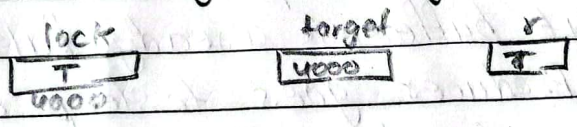
```
while (lock != 0);
    Lock = 1;
        CriticalSection
    Lock = 0;
```

```
while (TestAndSet (& lock));
        Critical section
    lock = false;
```

```
boolean TestAndSet (boolean *target)
{
    boolean r = *target;
    *target = ~~turn~~ true;
    return r;
}
```



initally lock = False has an address 4000 and the value is transfer to target i.e. target is a pointer variable



⑥ Sleep and Wake up

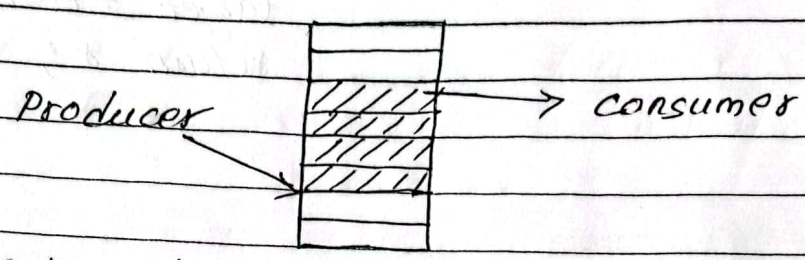
Sleep and wakeup are system calls that block process instead of wasting CPU time when they are not allowed to enter their ciritical region. Sleep is a system call that causes the calles to block that is be suspended until another process wakes it up. The wakeup call has no parameter the process to be awakened

Syntax: sleep();
wakeup (process to be allowed)

Explain the producer consumer problem or (Buffer Bounded)

Producer consumer problem is the example of Sleep and wakeup.

Two processes share a common, fixed size buffer. One of them the produces put information to the buffer and other cousmer take it out



Trouble aries when

1. The producer want to put a new data in the buffer but buffer is already full.
   Solution: Producer goes to sleep and to be awakened when the consumer has removed data
2. The consumer coant to remove data the buffer but buffer is already empty
   Solution: Consumer goes to sleep until the producer put some data in buffer and wakes consumer up.

```
#define N 100
int count = 0;
void producer (void)
{
  int item;
    while (True) {
      item = produce_item();
      if (count == N) sleep();
      insert_item (item);
      count = count + 1;
      if (count == 1) wakeup (consumer);
    }
}

void consumer (void)
{
  int item;
  while (TRUE)
    if (count == 0) sleep();
    item = remove_item();
    count = count - 1;
  if (count == N-1) wake up (produces);
    consume_item (item);
  }
}
```

N → size of buffer
Count → a variable to keep track of no. of items in the buffer

fig. Producer-Consumer Problem with a fatal race condition

Producers Code

It is first test to see if count is N. If it the producer will go to sleep ; if it is not the producer will add an item and increment count

Consumer Code

It is similar as of producer. First test count to see if it is 0. If it is, go to sleep ; if it non zero remove an item and decrement the counter.

Each of the process also tests to see if the other should be awakened and if so wake it up

2022) Explain the Producer Consumer Problems in process synchronization and give solution to it using semaphores

The Producer Consumer Problem is process synchronization is saw below

① The buffer is empty and consumer has just read count to see if it is 0

② At instant, the schedular decides to stop running the consumer temporarily and start running the producer. so Consumer is interrupt and producer resumed

③ The producer create an item, puts it into the buffer and increase count

④ Because the buffer was empty prior to the last addition (count was just 0), the producer tries to wake up the consumer

⑤ Unfortunately, the consumer is not yet logically asleep so wakeup signal is lost

⑥ When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep

⑦ Sooner or later producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. For temporary solution we use wakeup signal from getting lost, but it can't work for more process.

So for this problem we use semaphores

Semaphore

A semaphore is a special kind of integer variable which can be initialized and can be accessed only through two atomic operation. P and V. If S is the semaphore variable then

P operation: wait for semaphore become positive and then decrement	V Operation: Increasement semaphore by 1
P(s):	V(s):
while (s<=0)	s=s+1;
do no-op;	
s=s-1	

Write the algorithm for producer consumer problem using binary semaphores.

Producer - Consumer Proble using semaphore

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer (void)
{
    int item;
    while (TRUE) {
        item = produce_item();
        down(&empty);
        down (&mutex);
        insert_item (item);
        up(&mutex);
        up (&full);
    }
}

void consumer(void)
{
 int item;
  while (TRUE) {
      down (&full);
      down (&mutex);
      item = remove_item ();
      up (&mutex);
      up (&empty);
```

consume_item(item);
}
}

This solution uses three semaphor

① Full - for counting the number of slots that are full, initially 0
② Empty - for counting numbers of slots that are empty, initially equal to the no. of slots in buffer
③ Mutex - To make sure that producer and consumer do not access the buffer at the same time initially 1.

~~For Synchronization:~~

2020 2a) What is semaphore? Explain how you solve producer consumer problem using semaphore

2019) 3a) Write and explain solution for producer consumer problem using semaphore variable

2019) State Producer Consumer problem. Explain how to solve it.

It is a ~~class~~ synchronization proble that involve a fixed size buffer and producer process that create an item and adds it to the shard buffer and consumer process that remove the items from the buffer and consume them

Classic IPC Problem

2019 What are classic IPC Problem? Explain reader writer problem.

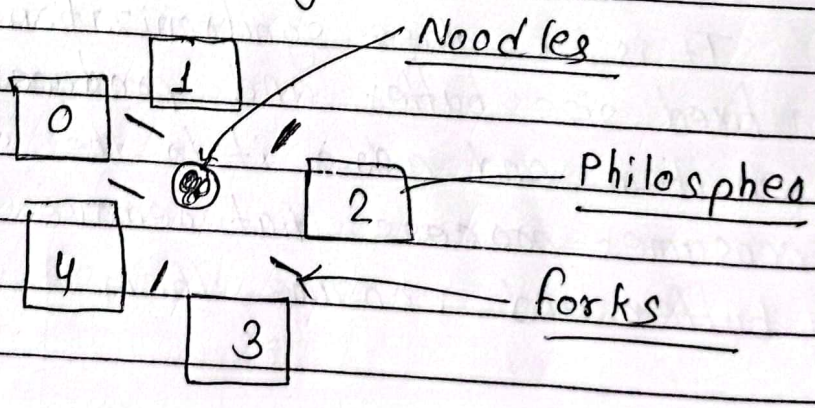
The classical IPC Problem are :-

1. Dining Philosophers Problem
2. The Reader and Writers Problem
3. The Sleeping Barber Problem

2022 Describe Dinning Philospher problem and show how deadlock occurs in it. Provide the solution to deadlock in Dinning Philospher problem

① Dinning philospher problem is a classical synchronization problem which is used to evaluate situation where there is need of allocating multiple resource to multiple process

Consider there are five philospher sitting arround a circular dinning table. The dinning table have five forks and a bowl of noodles in the middle as shown in figure



At any instant, philospher is either eating or thinking. When a philospher want to eat, he uses two forks. When a philospher want to think, he keep down both forks at their original place. However if a fork is already beign used by another philospher, the philospher must wait until the fork is available.

One of the main challenge of this problem is to avoid deadlock. Dead lock occurs when each philospher picks up the forks on their left and then waits indefinitely for the fork on their right to become available. Noone will be able to take their right forks and there will be a deadlock.

The code for each philospher look like:

Let an array of five semaphore stick[5] for each of the five chopsticks/forks

```
while (TRUE)
{
    wait(stick[i]);
        /* mod is used because if i=5, next
           fork is 1 */
    wait(stick[(i+1)%5]);
        /* eat */
    signal(stick[i]);
    signal(stick[(i+1)%5]);
        /* think */
}
```

The four way to deal with deadlock in the Dining Philosphere Problem:

① Resource Hierarchy solution
Assigne a unique numbes to each fork and require that a philosphere pick up the lowest numbered fork first. This ensure that no two philosphers will be waiting for the same fork at the same time, thus avoiding deadlock

② Chandy / Misra Solution
This solution involves passing message between philosphere to determine whether they are holding a fork. If a philosphere is holding only one fork, they will pass it to the philosphes who has requested it. This solution avoid deadlock by ensuring that no two philosphers are waiting for the same fork at the same time.

③ Arbitrator Solution
In this solution, an arbitrator solution is introduced to control access to the fork. A philosphere must require request permission from the arbitrator before picking up a fork This solution avoid deadlock by ensuring that only one philosphere can hold a fork at a time.

④ Deadlock Detection and Recovery

This solution involves periodically checking the state of the system to detect whether a deadlock has occured. If a deadlock is detected, the system can recover by releasing all resource held by the philosphers and restarting the algorithm. This solution avoids deadlock by ensuring that the system can recover from it if it occur.

Resource Hierachy Solution
Chandy solution
Arbitrator solution

The possible solution for the avading the deadlock are:

- Allow only four philosphere to sit to the table That way, if all the four philosphes pick up four forks, there will be one fork left on the table. So one philosphere can start eating and eventually, two forks will be available. In this way deadlock can be avoided.

Readers and Writer Problem

Another famous problem is reader and writer problem which models access to database. Imagine for example an airline reservation system with many competing process wishing to read and write it. It is acceptable to have multiple process reading the database at same time but if one process is writing the database, no other process may have acess to the database not even a reader.

Solution to Reader Writer problem

```
typedef int semaphore;
seamaphore mutex = 1;
seamaphore db = 1;
 int rc = 0

void reader (void)
 {
   while (True) {
      down (&mutex);
      rc = rc + 1;
      if (rc == 1) down (&db);
       up (&mutex);
      read_data_base ();
       down (&mutex);
      rc = rc - 1
      if (rc == 0) up (&db);
```

```
        up(&mutex);
        use_data_read();
    }
}

void writer(void)
{
  while (TRUE) {
        think_up_data();
        down(&db);
        writer_data_base();
        up(&db);
    }
}
```

In this solution, the first reader to get access to the database does a down on the semaphore db. Subsequent readers merely have to increment a counter, rc. As reader leave, they decrement the counter and the last one but does an up on the semaphore, allowing a blocked writer, if there is one to get in.

Sleeping Barber Problem

There is one barber, one barber chair and n chair for waiting for customer

- If there is no customer barber sleeps in his own chair
- When customer comes he has to wakeup the barber
- When many customer come and waiting chair are empty then they sit on the waiting chair else they leave if no chair is empty

Solution:

```
chairs = N;
semaphore customer = 0; // No custorm is waiting room
semaphore barber = 0; // barber is idle
semaphore mutex = 1; // mutual exclusion
int waiting = 0

void customer_process (void) {
    while (TRUE) {
        wait (mutex); // down
        if (waiting < chair) {
            waiting = waiting + 1;
            signal (customer); // up
            signal (mutex);
            wait (barber); // down
            get_hair_cut ()
        }
```

```
        else {
            signal (mutex);
        }
    }
}

void barber-process (void) {
    while (true) {
        wait (customes);
        wait (mutex);
        waiting = waiting - 1;
        signal (barbes);
        signal (mutex);
        cut-hair ()
    }
}
```