Topic _____ Date _____

CS-341.

Operating Systems

Mid-Semester Assignment

- Tanusi Muttal
- 190108565
- Tanusi Muttal

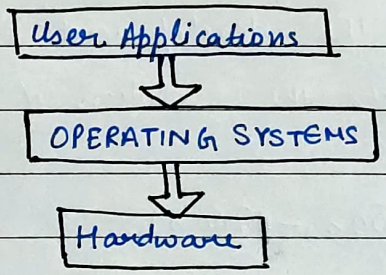
Topic _____ Date _____

Tanusi Mittal

Que 1:- Write a critical analysis report on the history and underlying concepts of operating system development. Use tables for comparison, focus on all Early Operating Systems, Growth of Operating System Concepts, Concurrency and Parallelism, Multi-programming. Elaborate with example.

Ans:- An operating system is a piece of software that is responsible for making it easy to run computer programs (along with running many number of them at the same time), allowing programs to share memory, enabling them to interact with various devices, and other related stuff.

Basically it is a system software that manages computer hardware, software resources, provides common services for computer programs and acts as a interface between the computing system and end user.



History of Operating System:-

Early computing systems were purely mechanical machines, they had specific functionality based on the hardware and could only perform a single computation at one time. There were no need of an user-interface as today.

After that developments in electronics and electrical industry led to electronic computers, which were based on machine languages. They were only able to do mathematical computations

Topic _____ Date _____

at very slow speed. Computing advances after this led to requirement of OS.

Growth of Operating System Concepts:

In early 1950's, Punch Cards were invented. At the same-time various computing machines supported punching and reading at card decks. The programmer would write instructions on the card deck, pass it to the operator who would then load the program into machine. After that computation was done. After invention of transistors, they were used in computers as they were smaller, more realiable and used less power compared to vacuum tubes, but they were costly. Majority of the Operating Systems evolved according to the corresponding development in requirements of a computer.

We can divide the development of operating systems into stages to describe the evolution of the features.

(A) STAGE - I (1950's - 1960's)

Computers used to be very costly, human labour was cheap. One user at a time, Operating System was a subroutine library.

Key Developments

1) **Console - User System:**

→ One user would access the system at a time.

→ Job entry required user to manually enter details when one job is finished.

→ As a result, large amounts of time was wasted in entering details.

Topic_____ Date_____

2) **Batch Monitor System:**

→ Can load next job immediately after previous one finishes.

→ There was no protection from program crashes which would result in time waste due to reboot.

→ Programs were first loaded and then run.

→ Computers were not used during I/O.

3) **Data channels, interrupts:**

→ Overlap I/O and computation

→ Direct memory access for I/O devices.

→ OS requests I/O, goes back to computing and gets interrupt when I/O device finishes.

(B) **STAGE - 2 (1960's - 1970's)**

Computers became a little cheaper and were now used for research purposes commonly and rarely for commercial purposes.

Key Developments:

1) **Multiprogramming:**

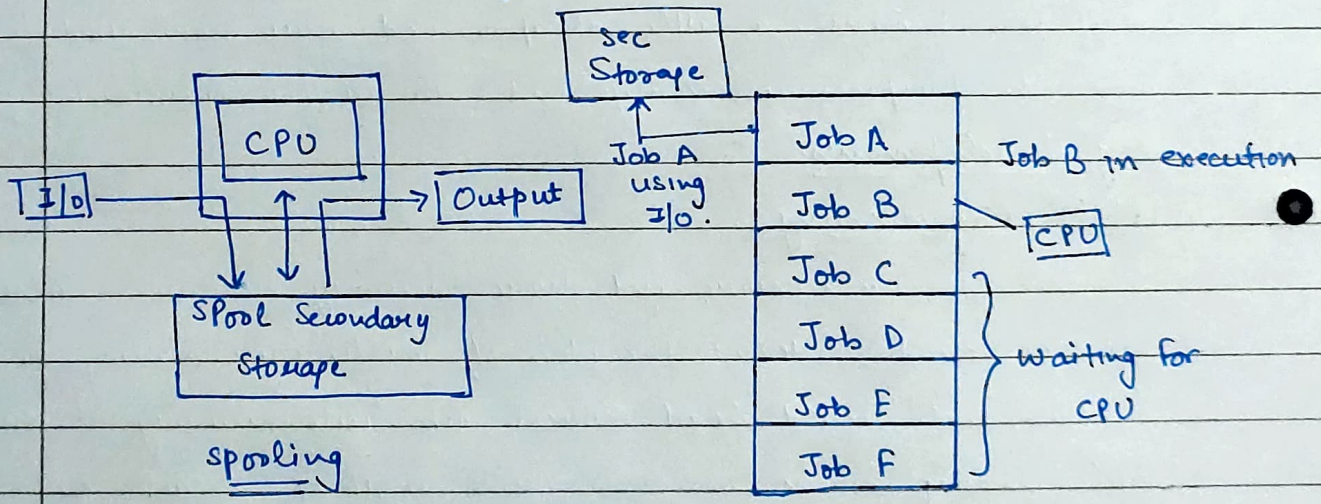
→ As machines with more and more memory became available, it was possible to extend the idea of multiprogramming to create systems that would load several jobs into memory at once and cycle through them in some order working on each one for a specific period of time.

→ When a program stopped to wait for I/O in these systems, the OS was now able to switch very rapidly from currently executing program to the next. The short interval was called context switching time.

Topic _____ Date _____

→ Execution of jobs at same instance of time is not referred to as multiprogramming but it is defined as the number of jobs available to the producer and a portion of another process is executed then a segment of another and so on.

eg: Consider a machine that can run two jobs at once. Further suppose that one job is I/o intensive and other is CPU limited. One way to allocate CPU time between these jobs would be to divide time equally among them. However CPU would be idle much of the time I/o bound process was executing. A good solution is to allow the CPU bound process to execute until I/o bound process needs CPU time. Presumably it will soon need to do some I/o and so the computer can return the CPU to the first job.



spooling

Multiprogramming

Topic _____ Date _____

(C) **STAGE-3 (1970's - 1980's)**

They were the next important generation of systems developed.

Key Developments:

1) **Interactive Time-sharing:**

→ Time sharing interactive system's most significant advantage was the capability to provide interactive computing to the users connected to the system.

→ The most basic technique employed on these systems was that the processor's time was significantly shared by the user programs.

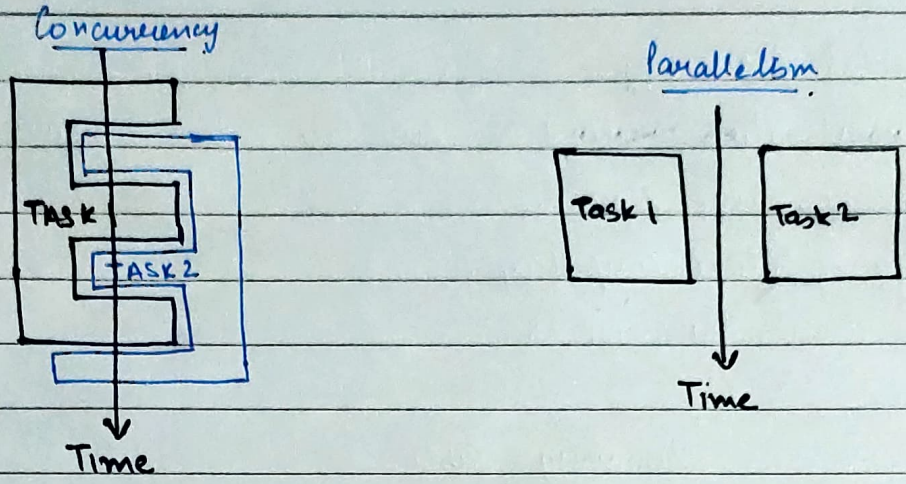
→ The OS provided CPU service during a small and fixed interval to a program.

→ Time sharing enabled many people, located at various terminals to use a particular computer system at the same time, hence this was the illusion that every user gets.

→ The concepts of concurrency and parallelism were developed.

CONCURRENCY	PARALLELISM
1. Concurrency means different processes were run adjacent to in time and interleaved such that the user thought that the computer is running all the programms at the same time	1. The technique of using multiple processors concurrently is called parallel processing.
2. CPU scheduling can help acheive concurrency	2. Threading and load balancing b/w processes can help achieve it
3. Makes system responsive	3. Speeds up the system also
4. A single CPU can support concurrency	4. More than 1 CPU's are needed in single system

Topic _____ Date _____



(D) STAGE-4 (1980's - 1990's)

→ Hardware systems become even cheaper and computers were commercially used.

→ OS became a subroutine library again (MsDos, MacOs).

→ Features like multiprogramming, memory protection were accomodated.

→ The Operating system was very user-friendly.

(E) STAGE-5 (1990's - Present)

→ PC's are very common and OS is highly user friendly.

Key Concepts

1) Distributed Systems:

→ As computers started to get cheaper, can have them in various places in high numbers.

→ Internet gained popularity.

→ Network fast allow machines to share resources & data easily

→ Network cheap allow distant machines to interact with each other.

Topic _____ Date _____

2) Virtual Machines:

→ Virtualisation helped to simulate entire computer system virtually on the same machine

→ Rise of Cloud Computing

→ eg Azure, AWS, GCP etc.

Summary - Table

OS Class and Time Period	Prime Concern	Technical Innovations	Key Concepts
Batch Processing (1950-1960's)	CPU Idle Time	Tape Batching FIFO Scheduling	Automatic transition between jobs
Multiprogramming (1960's - 1970's)	Resource Utilization	I/O Spooling, Priority scheduling, Remote job entry	Programmer writes, pre-emption.
Time Sharing (1970's - 1980's)	Good Response Time	Simultaneous User Interaction on-line file systems	Time slice, Round Robin Scheduling
Real Time (1980's - 1990's)	Meeting Time Concerns	Hierarchical Systems Extensible Kernels	Real Time Scheduling
Distributed (1990's - Present)	Resource Sharing	Remote Sources	Distributed control transparency.

Topic _____ Date _____

Que 2:- Discuss Semaphore-based Solution for classical problems of Synchronization (eg- 1. Bounded buffer (or Producer-Consumer) Problem, 2. Dining-Philosphers Problem, 3. Readers and Writers Problem, 4. Sleeping Barber Problem etc.)

Ans:- A semaphore is a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system. Semaphores are a type of synchronisation process.

Basically it makes sure that if in a code there is a critical section which can be used by multiple users but their simultaneous usage might lead to asynchronisation, the semaphore in that case provides a method that only one of them / multiple (based on the problem) codes can use the critical section.

Now, we will look at some standard problems of synchronization and their Semaphore-Based Solutions.

1. Bounded Buffer Or Producer-Consumer Problem

Statement:

We consider two processes:- consumer process and producer process. And the assumption is that they are parallel processes i.e they are coming at the same time and they share something between them that means they are cooperative processes.

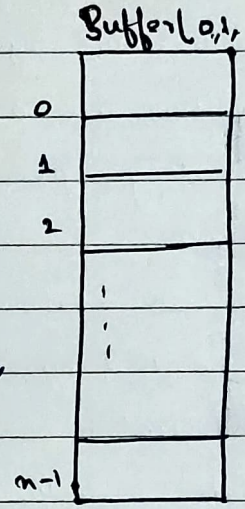
In our case there is a finite buffer pool on which both the processes operate.

Topic _____ Date _____

In this buffer, let us consider that there are n slots, so now the producer process when it runs fills one slot of the buffer by the item produced by it and the consumer process tries to consume one item from the buffer slot every time it runs.

Consumer Code

```
void consumer (void){
  int itemc;
  while(true){
    while (count==0);
    itemc = Buffer(out)
    out = (out+1) mod n;
    count = count - 1;
    Process_item(itemc);
  }
}
```



Producer Code

```
int count = 0
void producer (void){
  int itemp;
  while(true){
    produce_item(itemp);
    while (count==n);
    Buffer[in] = itemp;
    in = (in+1) mod n;
    count = count + 1
  }
}
```

As we can see from the codes their is a global variable count which is used by both the processes.

Now, our producer code is trying to increase the value of count whereas our consumer process is trying to reduce the value of count.

In simple words; count basically represents the no. of items currently in our buffer.

Here the asynchronization occurs on the step count = count+1 or count = count-1

If we dig deep into how these instructions are executed by our CPU

Topic _____ Date _____

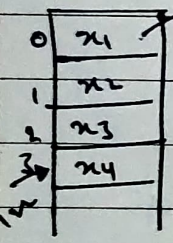
we know;

count = count - 1 count = count + 1

1. load R_c, m[count] → loads value in register 1. load R_p, m[count];
2. DECR R_c → decreases the value of register 2. INCR R_p;
3. Store m[count], R_c → stores the new value 3. Store m[count], R_p;

Lets try to simulate the code and see how problem is arising. Let us say we have 3 items already in our buffer that are produced by our producer process and the consumer process has not be run till now; so the values of different variables would be. → represents the position of buffer where new item would be added.

count = 3 ; in = 4 ; out = 0



Now lets say producer tries to produce one more item ⇒ at 3rd position x_4 is inserted; in = 5. , but now when we were incrementing count; the code will be run by instructions and first 2 instructions are completed but after that our process is pre-emptied.

Now Rp value is 4

And now our consumer code will run and ~~producer~~ consumer will take out 1 item. Now while doing count - 1 it will again run the instructions. Till now the count value is still 3. since the value was not updated by producer and it was pre-empted.

Now lets assume our consumer code also gets pre-empted ~~before~~ after it ~~could~~ executes the third instruction and now the 3rd instruction of producer will be run first. Here value of count was 3 but Rp = 4. so Rp will store 4 in count now.

Topic_____ Date_____

So at last our count value is 4; while there are only 3 items in our buffer array. Hence this leads to RACE Condition

Semaphore - based Solution

For the above problem we use both counting and binary semaphores to reach the solution.

Binary Semaphore : m : used to acquire and release lock.

Counting Semaphores:

1) empty = buffer size ; initialize empty slots
2) full = 0 ; initialize full items.

Producer Code

```
do{
  wait(empty); → waits until empty > 0 and decrements it
  wait(mutex); → acquires lock to prevent buffer or count access
                 while producer runs.
// Producer Code
  signal(mutex); → release the lock
  signal(full); → increments 'full' as the item is added to
                  the buffer by producer
} while(TRUE)
```

Consumer Code

```
do{
  wait(full); → waits until full > 0 and then decrements it.
  wait(mutex); → acquire lock to prevent buffer or count access
// Consumer Code   while consumer runs
  signal(mutex); → release the lock
  signal(empty); → increments 'empty' as the item is removed
} while(TRUE)      from the buffer by consumer.
```

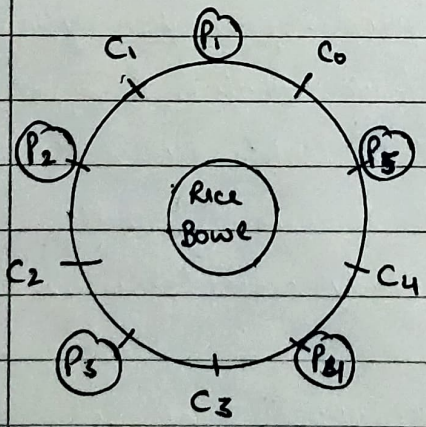
Topic _____ Date _____

Now if we try to see that whether our problem has been solved or not: Lets assume that the producer tries to produce an item. Lets us there are a total of n slots.
So producer first decrements the value of empty to n-1 and then it locks the mutex.
Now lets assume that before it could enter the ~~producer~~ producer code it is pre-empted and goes to consumer code. In consumer code the value of full is decreased but as it will go to the next line to lock the mutex it finds out that the mutex is already locked and consumer code will not execute further. and it will return to the producer code.
Hence we can say that our problem is solved here.

2. Dining – Philosopher's Problem

Statement:

In this problem, we have 5 philosophers who are sitting around a circular table. This table has 5 chopsticks, 1 b/w each philosopher and a rice bowl in the middle.



At any instant of time, any philosopher is either thinking or is eating. If he is thinking then there is no issue but if he wishes to eat, then he needs 2 chopsticks, the left and the right one. After he is done eating and wants to think again. He puts the chopsticks at their original place.

Topic _____ Date _____

This is a classical synchronization problem which is used when we need to allocate multiple resources to multiple processes. and it is an example of concurrency control problem.

The function code is like this:

```
Philosopher(i){
  while(TRUE){
    thinking();                    → while thinking
    take_fork(i);
    take_fork((i+1) mod N);        } Picks up chopsticks
    eat();                         → while eating
    put fork(i);
    put fork((i+1) mod N)          } Put down chopsticks.
    }
}
```

Now, here the problem arises when let us all the philosphosers want to eat. Lets assume that first P_1 will take its left fork ie C_0 and say the procer preempts after that and ends in P_2 taking his left fork which is C_1 and it also pre-empts finally continuing till P_5 takes his left fork C_4. Now when the first procer will resume P_1 wants his right fork which is C_1, but it is already been taken by P_2 and so on.

so in this a case of deadlock arises and we are stuck in the situation forever.

Topic _____ Date _____

Semaphore - based Solution

We represent each chopstick / fork with a semaphore (binary) so for that we take an array of binary semaphores s[5] lets say where s[0] = s[1] = s[2] = s[3] = s[4] = 1 initially that means all the forks are available.

```
Philosopher(i){
    while(TRUE){
        thinking();                          → while thinking
        wait(take-fork(a[i]));
        wait(take-fork(a[(i+1) mod N));      } wait until both the forks are available and then mark them unavailable
        eat();                               → while eating
        signal(put-fork[a[i+1) mod N)]
        signal(put-fork(a[i]);               } marks both forks as available.
        }
    }
```

The problem of deadlock in this problem can be removed by:

1) Allowing only n-1 philosophers to sit at the table
2) Allowing only to pick chopsticks when both of them are available
3) We can reverse the order of chopstick pickup for any one of the philosopher.

All the methods could prevent the situation of deadlock from happening.

Topic _____ Date _____

3. Readers and Writer's Problem

Statement:

In this problem we consider that there is a database which should be accessed by multiple processes. However there are only 2 type of processes

1) Read → It can only read from the database
2) Write → It can change the database ie. write into it.

Now the following things can happen

1) Many users are reading from the database
2) One user is reading and other is writing
3) More than one user is writing.

If we see in the first case there is no issue, but in the 2nd and 3rd case, there can be inconsistency. like it may happen that data is lost, or the reader reads the inconsistant data from the database.

So we need to develop a solution such that

1) Any no of readers can read from the database
2) If one user is writing there should be no other user writing or reading the data.
3) If first user is already reading, no body should write into the data in that feame of time

To solve the above conditions we came up with the semaphore - based solution.

Topic _____ Date _____

Semaphore - Based Solution:

Semaphores used:

mutex = 1

w = 1

Integer variable rc = read count = 0; no of readers reading.

Writer - Code

```
writer(){
  while(TRUE){
    wait(w); → write waits until it gets the opportunity to write.
    // writes.
    signal(w); → w incremented so that next writer may come.
  }
}
```

If a writer is in a critical section and n readers are waiting, then 1 reader is queued on w and other n-1 readers are queued on mutex.

When a writer executes signal, we may resume the execution of either the waiting readers or a single waiting writer.

Reader - Code

```
reader(){
  while(TRUE){
    wait(m); → decrease mutex 'm' to make it independent of other readers
    rc = rc+1; → read count increased
    if(rc==1) wait(w); → If rc=1, decrease w, and hence no one will be able to write.
    signal(m); → m incremented to allow other readers to come
    // reads
    wait(m); → decrease m to make it independent of other readers.
    rc--  → read count decreased
    If(rc==0) signal(w); → If rc=0, increment w, to allow writers to come
    signal(m); → increase m to allow other readers to come
  }
}
```

Topic _____ Date _____

4. **Sleepy Barber Problem :**

Statement:

In this problem, there is a barber shop, with one barber, one barber chair, n chairs for waiting customers (if any) to sit.

1. If there is no customer, barber sleeps in his own chair
2. When customer comes, he wakes up the barber.
3. When a customer arrives and the barber is busy, he waits if there is empty chair or else the customer leave.

In this problem, there arises the condition of a race condition, which we have to take care in our semaphore solution.

Semaphore - Based Solution.

In this solution we use 3 semaphores.

customers → to count the number of waiting customers.

barber → to check if barber is free or not.

mutex → for mutual exclusion.

Apart from this we use

variable seats → seats filled by customers.

In our solution, the customer entering the shop counts the no. of waiting customers. If they are less than the no of seats, the new customer stays or else, he leaves.

Topic _____ Date _____

Initially:

semaphore customer = 0 (no customer is waiting)

semaphore barber = 1 (he is free)

semaphore mutex = 1

variable seats = 0; (no seat is taken initially)

Barber Procedure

```
barber(){
  while(TRUE){
  wait(customers);
  wait(mutex);
  seats++; → increment seat
  signal(barbers) → barber now busy.
  signal(mutex)
  // cuts hair
  }
}
```

Customer Procedure

```
customer(){
  wait(mutex);
  if(seats > 0){
  seats--;          ← available seats decreased.
  signal(customers);
  signal(mutex);
  wait(barber);     ← waits for barber to be free.
  }
  else{
  signal(mutex);
  }
}
}
```

P.T.O. →

Topic _____ Date _____

Que 3:- Extend A2(Que) with Monitor based solution.

Ans:- **Monitors:**

Monitors are abstract data types and contain shared data variables and procedures. The shared data variables cannot be directly accessed by a process. Procedures are required to allow a single process to access the shared data variables at a time. Since only one process can be active in a monitor at a time, this allows monitor to enforce mutual exclusion.

1. **Monitor based Solution for Bounded Buffer (or Producer-Consumer) Problem:**

Here the critical sections of the producer and consumer are inside the monitor 'Producer Consumer'. Once inside, a process is blocked by the wait and signal primitives if it cannot continue.

Code:

```
monitor ProducerConsumer
        condition full, empty;
        int count;

procedure enter( ){
    if (count == N) wait(full);        → if buffer is full, block
    put-item(widget);                  → put item in buffer
    count ++;                          → increment count of full slots
    if (count == 1) signal(empty);     → if buffer empty, wake consumer
}
```

Topic _____ Date _____

```
procedure remove ( ){
    if (count == 0) wait (empty);      → if buffer is empty, block
    remove - item (widget);            → remove item from buffer
    count --;                          → decrement count of full slots
    if (count == N-1) signal (full);   → if buffer was full, wake
    }                                    producer
count = 0
end monitor;

Producer ( ){
   while(TRUE){
   make - item(widget);                → make a new item
   Producer Consumer. enter;           → call enter function in monitor
   }
}

Consumer {
   while(TRUE){
      Producer Consumer . remove ;     → call remove fnc. in monitor.
      consume - item(widget);          → consume an item
      }
}.
```

Topic _____ Date _____

2. Monitor - based Solution for Dining Philosophers Problem:

This monitor based solution imposes a restriction that a philosopher can pick up his chopsticks only if both are available. Now the philosopher at any instant of time can be in one of the 3 states:

1. Thinking: When philosopher does not want access to chopstick.
2. Hungry: When philosopher wants to enter critical section
3. Eating: when philosopher has both the chopsticks and is there in the critical section.

We need to know that philosopher i can set the variable state[i] = EATING; only if the 2 neighbours are not eating. i.e.

[(state[(i+4)%5] != EATING) and (state[(i+1)%5] != EATING]

Code:

```
monitor DP {
    status state[5];
    condition self[5]
    Pickup(int i){                    → pickup chopsticks
        state[i] = HUNGRY;            → indicate that philosopher i is
        test(i);                        hungry and check if he can eat
        if (state[i] != EATING)       → if not able to eat, then wait.
            self[i].wait();
    }
```

Topic _____ Date _____

```
Putdown (int i) {                    → putdown chopsticks.
    state[i] = THINKING;             → indicate philosopher i is thinking.
    test((i+1)%5);                   → if right neighbour is hungry
    test((i+4)%5);                   → and both neighbours are not
    }                                  eating update accordingly.
test (int i) {                       → to check if philosopher i can eat.
  if (state[i] = HUNGRY &&
      state[(i+1)%5] != EATING &&
      state[(i+4)%5] != EATING) {
      state[i] = EATING;             → indicate philosopher i is eating.
      self[i].signal();              → wake up other hungry
      }                                waiting philosophers.
  }

init () {
   for (int i=0; i<5; i++) {         → initialises all the
      state[i] = THINKING;             states to thinking.
      }
   }
}

end Monitor DP.
```

Topic _____ Date _____

3. **Monitor Based Solution for Readers and Writers Problem:**

In this approach, we need to ensure that the methods should be executed with mutual exclusion. But only that is not enough. Threads attempting an operation may need to wait until some assertion P holds true. While a thread is waiting upon a condition variable, that thread is not considered to occupy monitor, and so other threads may enter to change monitor's state.

Code:

```
monitor ReaderWriter {
    condition canRead = NULL, canWrite = NULL;
    int activeReader = 0, activeWriter = 0;
    int waitingReader = 0, waitingWriter = 0;

    BeginWrite(){
      if (activeReader > 0 || activeWriter == 1){
          waitingWriter++;
          wait(canWrite);
          waitingWriter--;
          }
      activeWriter = 1;
    }
```

} state variables.

// A writer can enter begin if there are no other active writers and no readers are waiting to access the database

Topic _____ Date _____

```
EndWrite(){                          // checks if any readers
  activeWriter = 0                      are waiting.
  if (waitingReader)
     signal(canRead);
  else
     signal(canWrite);
  }

BeginRead(){
  if(activeWriter == 1 || waitingWriter > 0){ // A reader
     waitingReader++;                  → can enter if there
     wait(canRead);                    are no writers active
     waitingReader--;                  or waiting, so many
     }                                 readers can be active
  activeReader++;                      at once.
  signal(canRead);
  }

EndRead(){                           // When a reader finishes
  if(--activeReader == 0)               it lets a writer in (if
     signal(canWrite);                  there is any), if it was
  }  }                                  the last reader.
}

end Monitor;
```

Topic _____ Date _____

4. Monitor Based Solution for Sleeping Barber Problem:

```
monitor barber {
    int waiting = 0 ;              → customers in chair = 0
    bool sleeping = TRUE;          → barber is sleeping
    condition chair;               → customer waiting in chair
    enter(){
        if( waiting == numChairs) → if no empty chairs
            exit();                  → customer leaves.
        else if (sleeping == TRUE) → if barber is sleeping
            sleeping = FALSE;        wake him up
        else {
            waiting++;             → if barber is cutting
            chair.wait();            someon's hair already,
            waiting--;               then wait.
            sleeping = FALSE
            }
        }

    leave(){                       → customer leaves
        sleeping = TRUE
        if(chair.queue())          → if someone is waiting
            chair.signal();        → indicate the first customer
        }                            who was waiting that barber
                                     is now free.
    customer(){
    barber.enter();
    // gets haircut
    barber.leave();
    }
```

Topic _____ Date _____

Que 4:- Write a critical analysis report on deadlocks and methods for their detection, prevention and recovery in modern operating systems.

Ans:- Deadlock can be formally defined as:
'A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.'
Now, because all the processes are waiting, none of them will ever cause any event that could wake up any of the other members of the set, and all the processes continue to end up waiting forever.
Deadlocks can also occur among machines. eg many offices have a LAN network with many computers connected to it. Often devices such as scanners, printers etc are connected to the network as shared resources, available to any user. If these devices can be reserved remotely, deadlocks can happen.

Conditions for deadlock to happen:

As we know, in a deadlock, processes never finish executing and the system resources are tied up, preventing other jobs from starting.
A deadlock situation can arise if the following 4 conditions hold simultaneously in a system.

1. **Mutual Exclusion:** At least one resource ~~should~~ must be held in a non-sharable mode means that only one process at a time can use the resource.

Topic _____ Date _____

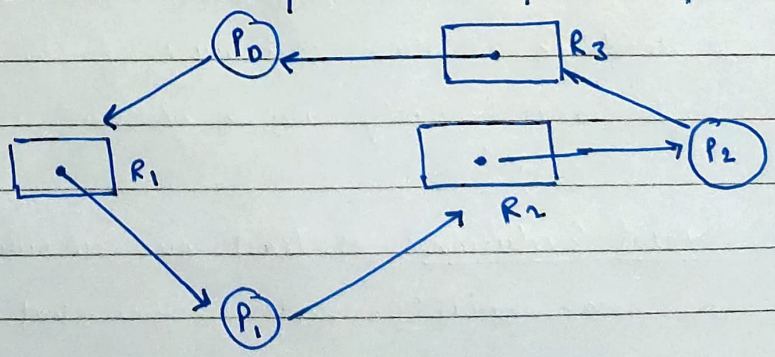
If another process requests that resource; the requesting process must be delayed until the resource has been released.

2. **Hold and Wait**: A process must be simultaneously holding at least one resource and waiting to acquire at least one more resource that is currently held by some other process.

3. **No-Preemption**: Once a process is holding a resource (i.e. once the request has been granted), then that resource cannot be taken from the process until the process voluntarily releases it, after it has ~~been~~ completed its task. In short resources cannot be preempted.

4. **Circular Wait**: A set of processes $\{P_0, P_1, \ldots, P_n\}$ must exists such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ---, P_{n-1} is waiting for a resource held by P_n and P_n for a resource held by P_0. i.e every $P[i]$ is wating for $P[(i+1)\%(N+1)]$.

If we see through the resource - allocation graph the condition should be something like this for 3 processes.



Deadlock situation of 3 processes.

Topic _____ Date _____

Methods of Handling Deadlocks:

There are four general methods of handling deadlocks:

1. Ignoring the problem → If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalities associated with deadlocks handing other methods.

2. Deadlock Detection and Recovery → Abort a process or preempt some resources whenever deadlocks are detected in a process

3. Deadlock Avoidance → dynamically avoid the deadlock situation by careful resource allocation.

4. Deadlock Prevention → We try to prevent deadlocks by structurally negating one of the four necessary conditions required for deadlock to occur.

Now, we will see all the methods in detail:

1. The first method of ignoring the problem is also known as the Ostrich Algorithm. Just like the ostrich sticks it head in the sand and pretend their is no problem, we also try to do the same, since the chances of occurring of deadlock in that condition maybe are very less.

Topic _____ Date _____

2. DEADLOCK PREVENTION

As we know, that for a deadlock to occur, each of the 4 necessary conditions must hold. So, by ensuring that at least one of them cannot hold; we can prevent the occurance of a deadlock.

a) Attacking the Mutual Exclusion Condition:

If no resources were ever assigned exclusively to a single process, we would never have deadlocks. The mutual exclusion condition must hold for non-sharable resources. for eg:

1) Read only files → if several processes attempt to open a read-only file, they all can be granted access
2) Printers → Now printers cannot be shared simultaneously by several processes.

So, we can conclude that

1. Shared resources do not lead to deadlocks
2. Some resources such as printers, tape drivers etc, require exclusive access by a single process and hence deadlocks prevention by avoiding mutual exclusion cannot be possible here.

b) Attacking the Hold and Wait Condition:

If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. This can be done by the following methods:-

Topic_____ Date_____

1. To require all processes to request all their resources before start executing. If everything is available, the process will be allocated whatever it needs to have and can run to completion.
   However this can be a wasteful of system resources if a process needs one resource early in its execution and doesn't need other resources until much later.
2. To require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request.
   However this can also be a problem if a process has partially completed an operation and then fails to get it re-allocated after releasing it. Also both the approaches can lead to starvation.

c) Attacking the No Preemption Condition

Preemption of process resource allocation can prevent the condition of deadlocks, when it is possible:
This can be done by the following approaches:

1. Process is forced to wait when requesting a new resource, then all the other ~~processes~~ resources previously held are implicitly released (preempted), forcing the process to re-acquire old resources along with new one in a single ~~pro~~ request.
2. When a resource is requested and it is not available, then the system looks to see if they is any other process who is in the wait condition and has those resources. If yes then resources are taken from that process as they get preempted.

These approaches are however not applicable to printer, drivers etc.

Topic _____ Date _____

d) **Attacking the Circular-Wait Condition :**

The Circular-Wait Condition can be avoided by the following method :

We number all the resources and to require that each process requests resources in an increasing order of enumeration.

i.e. in order to request a resource R_j, a process must first release all R_i such that $i >= j$.

However a challenge in this scheme is determining the relative ordering of the different resources.

3. DEADLOCK AVOIDANCE

→ The idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the four necessary conditions.

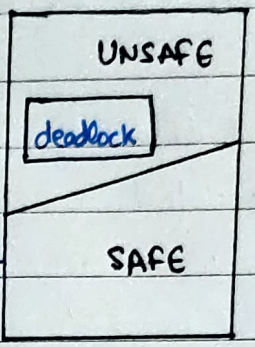
→ It is a conservative approach ~~and~~ as it requires more information about each process and tends to low the device utilization.

→ A resource allocation state is defined by the number of available and allocated resources and the maximum requirements of all the processes.

To see the deadlock avoidance algorithms we need to see what a safe and unsafe state means

Topic _____ Date _____

Safe State :



→ A state is a safe state if the system can allocate all processes requested by the process to it, without entering a deadlock state.

→ If a safe state does not exist, then the system is in an unsafe state, which may lead to deadlock.

Banker's Algorithm:

→ With the help of this algorithm, the deadlock avoidance can be acheived.

→ A banker's algorithm gets its name, because it is a method that banker's could use to ensure that when they lend out resources they will still be able to satisfy all the clients.

→ For resources categories that contain more than one instance of the resource allocation graph method does not work.

→ The banker's algorithm rely on the several key data structures:

1) Available [m] → how many resources are currently available of each type
2) Max [n][m] — indicates the maximum no of demand of each process
3) Allocation [n][m] — no. of each resource category allocated to each process
4) Need [n] [m] — remaining resources needed.

In order to apply banker's algorithm we check the Safety algorithm first which checks whether or not a particular state is safe and then we apply the banker's algorithm. This algorithm basically determines if new request is safe & grants it only if it is safe. It is also called Resource-Request Algorithm.

Topic _____ Date _____

4. DEADLOCK DETECTION

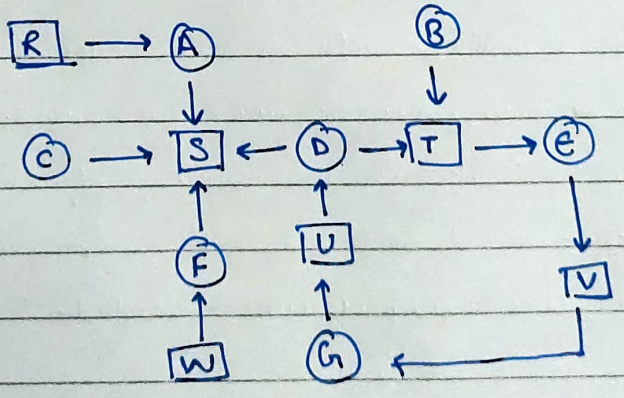
When deadlocks cannot be avoided, then we let them happen, then detect them and see how they occur and finally try to recover somehow.

Also in addition to the performance hit of constantly checking for deadlocks, a policy/ algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.

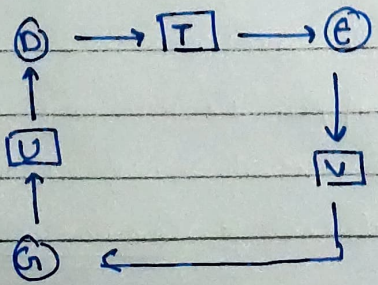
1) Deadlock Detection With One Resource of Each Type.

For such a system, we can construct a resource graph. If this graph contains one or more cycles, a deadlock exsists. Any process that is part of a cycle is deadlocked. If no cycle exsists, the system is not deadlocked.

eg.



A Resource Allocation graph



A cycle extracted from the graph.

Topic _____ Date _____

Although we can simply pick out deadlocked process by visual inspection from a simple graph, but there are also algorithms to detect that.

In our algorithm what we do is that we take each node, in turn as the root of what it hopes to be a tree and do a depth first search on it. If it ever comes back to a node it has already encountered, then we have found a cycle. If it backtracks to the root and cannot go further, the subgraph reachable from the current node does not contain any cycles.

2) Deadlock Detection with Multiple Resources of Each Type

For such a system, we have a matrix-based algorithm for detecting deadlock among n processes P_1 through P_n. Let the number of resources classes be 'm'. with E_1 resources of class 1, E_2 → class 2 and so on.

At any instant some of the resources are assigned and are not available. Lets take A to be the Available Resource Vector. where A_i gives the current availability.

E is called the Exisiting Resource Vector.

C → Current Allocation Vector

R → Request Matrix.

C_{ij} → no of instances of resource j that are held by process i.

R_{ij} → no of instances of resource j that P_i wants.

So; $$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Topic _____ Date _____

Our deadlock detection algorithm can be given as :-

1. First all processes are initially unmarked.
2. Look for unmarked process, P_i, for which the i^{th} row of R is less than or equal to A.
3. If such process is found, add the i^{th} row of C to A, mark the process and go back to ~~process~~ step 2
4. If no such process exists, the algorithm terminates

When the algorithm finishes, all the unmarked processes if any are deadlocked.

5. DEADLOCK RECOVERY

There are 3 basic approaches to recovery from deadlock.

1. **Process Termination:**
   - → The simplest way to break a deadlock is to kill one or more processes
   - → One possibility is to kill the cycle.
   - → However, a process not in the cycle can be chosen as the victim, in order to release the resources. Killing this process will release the resources and hence will break the deadlock of our main process
   - → Where possible it is best to kill a process that can be rerun from the beginning with no ill effects.

Topic _____ Date _____

2. Preemption of Resources:

In some cases it may be possible to temporarily take a resource from its current owner and give it to another process. However the following issues should be addressed before doing so:-

1. Selecting a victim: Deciding which resources to preempt
2. Rollback: Ideally we would like to roll back a preempted process but sometimes it can be difficult to determine considering the safe states
3. Starvation: There is a possibility that process getting preempted ends up in starvation. To prevent this we should increase the priority of a process every time its resources get preempted.

3. Recovery through Rollback:

If a system designers and machines operators know that deadlock are likely, they can arrange to have processes checkpointed periodically. Checkpointing a process means that its state is written to a file so that it can be restarted later. The checkpoint also contain the resource state along with the memory image. So when deadlock is detected, it is easy to see which resources are needed.

—x—x—x—x—END OF —x—x—x—x—

ASSIGNMENT