Unit 3 - Concurrency Control.

Concurrency → refers to execution of multiple instructions in sequence at same time.

Concurrency in OS → occurs when multiple process threads executing concurrently.

Principles of concurrency -

The ability to run multiple tasks at the same time.

① Process isolation -
→ process should have its own memory space & resources.
→ Maintains stability.

② Synchronization - process like locks, semaphores & mutexes

③ Deadlock avoidance

④ fairness OS should allocate CPU time fairly.

Problems in Concurrency -

① Sharing global resources -
if 2 processes make use of global variables & both perform read & write on that variable, then the order in which various read & write are executed is critical.

② Optimal allocation of resources -
difficult to mange the allocation of resources.

③ Locating Programming errors -
difficult to locate errors because reports are usually not reproducible.

④ Locking the channel -
Locking is difficult.

Advantages

→ Running of multiple applications
→ better resource utilization
→ better average response time
→ better performance

Drawbacks

→ required to protect multiple applications from one another
→ additional performance overheads
→ Sometimes leads degraded performance.

Critical section Problem.

Critical section → part of program which tries to access shared resources. (like memory location, OS, CPU / IO devices).

→ critical section cannot be executed by more than one process at the same point.

→ each process must request permission to enter its critical section

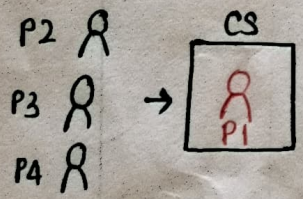
```
do {
    entry section   <-- section of code: code request for entering CS
        critical section
    exit section    <-- code exiting the CS
    remainder section   <-- remaining code is in this section.
} while (true);
```

SYNCHRONIZE ACCESS TO SHARED RESOURCES

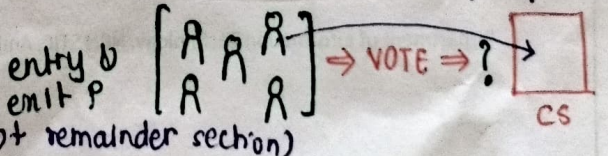
problem arises when multiple threads or processes need to access a shared resource / data, and if one process is executing inside this critical section, no other process / thread should be allowed to execute in CS simultaneously.

→ A solution to CS must obey :-

① Mutual exclusion :- if one process is executing inside CS then other process must not enter in CS.

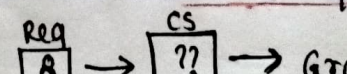


② Progress :- if no process is executing in CS & some wish to enter their CS, then the processes not in remainder section can decide which will enter its CS next & this selection cannot be postponed indefinitely.



③ Bounded waiting :-

there is bound / limit on number of times that processes are allowed to enter their CS after process has made a request to enter CS & before request is Granted.



Race condition -

→ The situation that occurs inside CS.
→ If two/more processes don't wait their turn to enter the cs & try to change things at the same time, the final results can be wrong & unexpected.
ie, if there is delay/overlap in execution, & processes don't have a way to control who gets access first, they can interfere with each other, leading to incorrect results
→ To prevent this, synchronization methods (like locks/semaphores) ensure that only one process can be in CS at any given moment, preventing RC.

Mutual Exclusion - to avoid Race condition

4 Requirements for mutual Exclusion :—
→ No 2 processes can reside inside their critical section.
→ No process outside its cs should block other processes (ie process not in cs cannot interfere with other process)
→ No process should wait arbitrary long to enter its CS (ie, not leading to deadlock/starvation).
→ No assumptions are made about relative speeds of processes or number of CPU's.

Semaphores

→ Synchronization tools used to control access to shared resources and prevent race conditions in concurrent programming.

→ manages processes that are trying to access the same resources simultaneously (shared mem., files, etc.)

→ (i) Binary semaphore (like muteres)

holds 2 values (0 & 1) ← wait, signal.

mainly used to ensure mutual exclusion. (ie 1 process in cs at a time)

(ii) counting Semaphore

holds non-negative integer value

allows fixed no. of processes to access resource at same time.

Q How does this work??

wait() / P() - decrements the semaphore value.
If value = -ve, the process is blocked.

signal() / V() - increment the semaphore.
If value = non-ve, process is awakened

Muten

locking → accessing → unlocking → waiting

→ Mutual Exclusion Object (ie muten) specifically designed for locking & unlocking cs. lock() unlock()

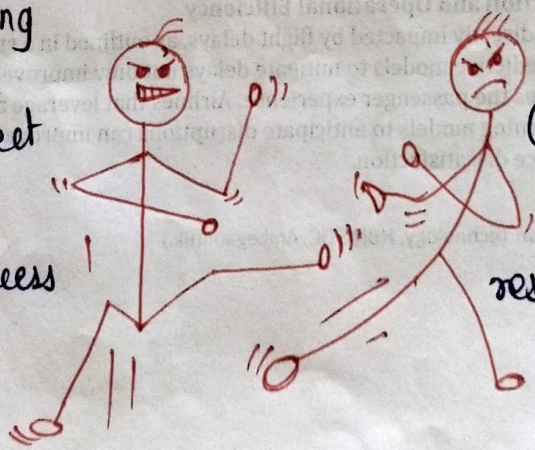
MUTEX VS SEMAPHORE

① Muten is a locking mechanism

② Muten is an object

③ Does not allow simultaneous cs access

④ No types



① Semaphore is a signaling mechanism

② It is an integer value

③ Allows simultaneous resource access but for finite processes

④ Binary & counting

Reader - Writers Problem

Readers → can access the database simultaneously without issues.
Writers → requires exclusive access
if writer is writing, no other process should access the database (reader / writer)

Synchronization issues

Simultaneous access by writer and any other process can lead to data inconsistency.
Thus, writers have exclusive access while performing write operations.

Variations of problem

① first readers-writers problem (active reader)

→ Readers can read simultaneously, unless there is already writer
→ If a writer is waiting to write, it does not block new readers do not to read (ie they can read) provided there are old readers

② Second readers-writers problem

→ once a writer is ready, it should perform its write ASAP.
→ New readers are not allowed to start reading if a writer is waiting (no old readers available then no new readers)

Issues

Starvation

→ In First RWP, writers may starve if readers keep accessing the database
→ In second RWP, readers may starve if writers continuously gain access.

Solutions

Semaphores - used to control access to shared resources.

Muter locks - Ensure mutual exclusion for writers

Pseudocode

```
semaphore muter = 1;
semaphore wrt = 1;
int readcount = 0;
```

```
function reader(){
    while (true){
        wait(mutex);
    readcount
        readcount ++;
        if (readcount==1){
            wait(wrt);
        }
        signal(mutex);

        wait(mutex);
    readcount
        readcount --;
        if (readcount==0){
          signal(wrt);
        }
        signal (mutex);
    }
}
```

```
function writer(){
    while (true){
        wait(wrt);
    signal(wrt);
    }
}
```

Producers-Consumers Problem.

Producer - creates items & adds them to shared buffer.

Consumer - removes items from shared buffer & consumes them.

Shared Buffer - it has fixed size ie limited no. of items
- if buffer is full, producer must wait & if buffer is empty, consumer must wait.

Conditions

Mutual Exclusion - only 1 process can access shared buffer at any time.

Bounded Buffer (Limit) - producer should not produce if buffer is full & consumer should not consume if buffer is empty.

Synchronization - sync. mechanism needed to manage access to shared buffer without conflicts.

Producer Process :-

```
while (true) {
    produce_item ();
    wait (Empty);
    wait (Mutex);
    add_item_to_buffer ();
    signal (Mutex);
    signal (full);
}
```

Consumer Process :-

```
while (true) {
    wait (full);
    wait (Mutex);
    remove_item_from_buffer ();
    signal (Mutex);
    signal (Empty);
}
```

Index

Solution using Semaphore

Empty - counts no. of empty slots in the buffer. Initialised to the size of buffer.

full - counts no. of filled slots in the buffer. Initialised it to zero.

Mutex - ensures mutual exclusion while accessing buffer

Inter Process Communication (Pipes, Shared Memory)

2 methods of IPC - Pipes & shared Memory

↳ methods used by OS to exchange data & signals. for allowing processes

Pipe :-

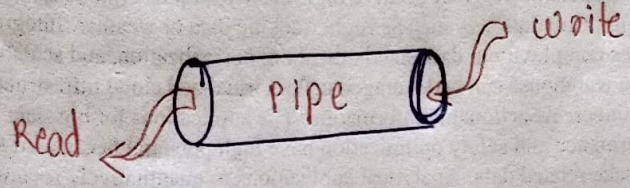
Communication medium that connects the output of one process to the input of other.

Communication between parent - child process.

Pipes are unidirectional channels ie flow in 1 direction (from writing to reading)

Implementation -

① Create a pipe - using pipe() system call
② fork the process - create child process by fork()
③ Close unused ends of pipe
④ Write Data - parent process writes the data to pipe.
⑤ Read Data - child process reads the data from pipe.



	Named Pipe (FIFO)	Unnamed Pipe (Anonymous Pipe)
Defⁿ ①	allows communication betⁿ 2 processes using a name in the filesystem.	temporary communication channel created in memory for data transfer betⁿ related processes.
②	Exists as long as it is not deleted	Exists only when processes using it are running.
③	creation sys calls mkfifo() & mknod()	Creation sys calls pipe()
④	Bidirectional	unidirectional
⑤	Appears as an entry in the filesystem	Exists only in memory

IPC through Shared Memory.

Shared memory is ~~process~~ portion of memory that can be accessed by multiple processes.

↳ ie specific memory segment can be accessed by multiple processes.

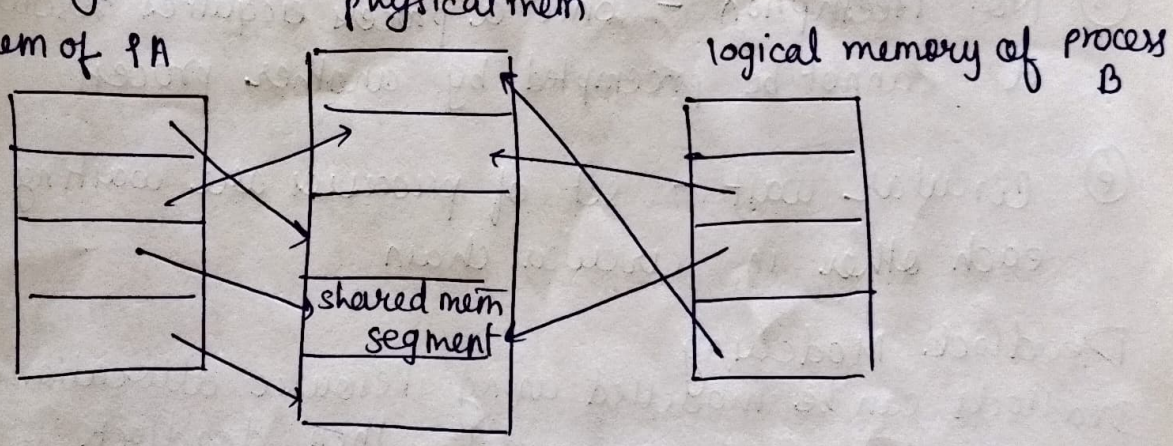
↳ Allowing faster data transfer between processes with any kernel

How does it work?

↳ one process (server process) creates the shared memory segment.

↳ client process attaches to this segment and can read data from it.

↳ Changes done by one process to this memory segment are instantly visible to other processes attached to the same segment



Deadlock

Principles of deadlock

↳ 2 or more processes are unable to proceed because each is waiting for other to release resources

Cond^n of Deadlock :-

(1) Mutual Exclusion - only one process can use resource at given time.

(2) Hold & wait - A process holding at least 1 resource is waiting to acquire additional resources that are

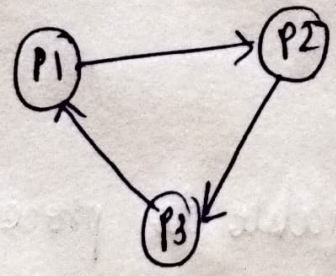
currently held by other processes.

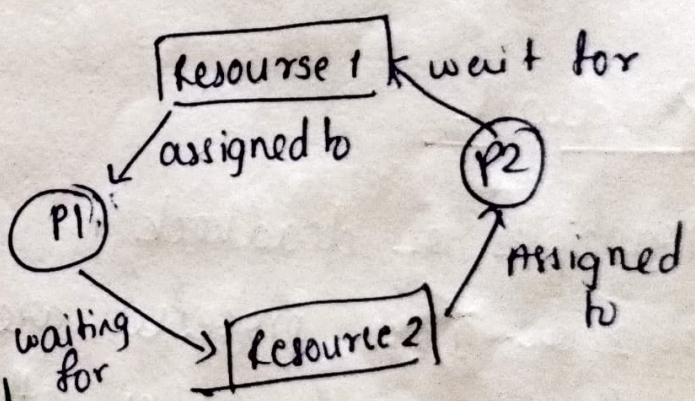
③ No Preemption - once a process acquires resource, it cannot be preempted by another process

④ Circular wait - set of processes are waiting for each other in circular chain

Deadlock Modelling

Deadlocks can be modelled using Resource allocation graphs. If there is cycle in model graph, then deadlock is possible





Strategies to deal with deadlock

① Prevention

Eliminate one / more condition (mutual exclusion, hold & wait, no preemption, circular wait), systems can prevent deadlocks from occuring.

② Deadlock Avoidance

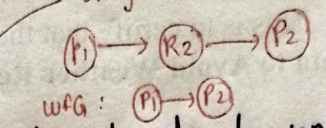
examine the state of resources allocation to ensure system remains in safe state.

But how??

↳ Banker's Algo

↳ RAG (Resource Allocation Graph)

single instance of each resource type



use of WFG

P_1 req R_2 held by P_2

WFG :

③ Deadlock Detection

Here system allows deadlock to happen/occur but includes mechanism to detect after they happen.

↳ WFG (Wait for Graph)

↳ RAG

④ Deadlock Recovery

deadlock occur ✓ now recover the system state

↳ Process termination - process in deadlock are terminated (kill)

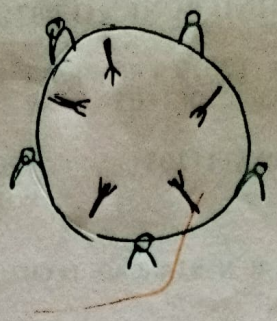
↳ Resource Preemption - forcefully take resources of one process & give it to another for completing execution

↳ Rollback - undo

↳ Priority Inversion - raise priority of processes that are stuck in deadlock.

Dining Philosopher Problem

Philosopers - Process
Chopsticks - Resource
Thinking & eating - do nothing / use resource



2 forks for 1
left right

Challenges?
↳ Deadlock ↳ Starvation ↳ Concurrency control (no stability)

Solution
↳ semaphore

Bankers Algorithm

↳ resource avoidance & allocation algo by Edsger Dijkstra.

Safe State
Unsafe state

Synchronization –
coordination in concurrent processes to prevent issues such as race cond^n, inconsistency, etc.
How ??
Data Consistency, Mutual Exclusion, Deadlock prevention
By What ??
Mutex, semaphore, CS