Lecture 1:

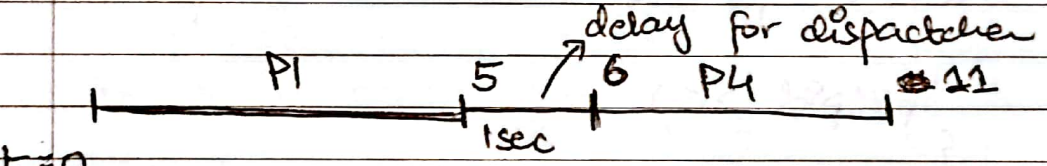
Parallelism → Two processes running at same time
Concrency → one process runs for a specific time
Then other process runs.
Kernel → core part of OS (stays in RAM)

Lecture 2:

Parallelism → multiprocessing (more than 1 CPU)
Scheduler → takes decision to give CPU to which process
Context → values in registers for a particular process
While giving CPU to other process, context is saved in memory
Dispatcher → loads the context of the process about to be executed.
(Small program)

PCB → process control block

time slash = 5 sec



Job scheduler → Degree of Multiprogramming

Lecture 3:

ek dafa (.) → current directory
(..) → current directory parent

root path: mkdir /home/temp/Desktop/quiz
relative: mkdir ./Desktop/quizzes (for current) (directory)
mkdir ../temp/Desktop/quizzes
(../ to go for parent)

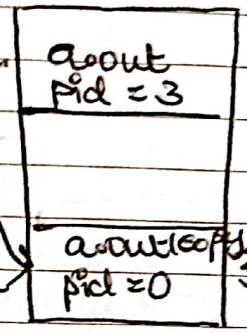
Lecture 4:

System calls: fork(), wait()

```
int main()
{  int pid = fork();
   cout << " ";
   cout << " ";
}
```

→ gives a numeric value [which is id of new copied process]

[creates a copy of file in which fork is called]



The process in which fork() is called is parent and it's copy is child. The pid of parent is id of child and pid of child is 0.

fork creates a copy of the whole program in which it is called and creates its copy and gives it a new id which is returned by fork().

→ In child, the process runs after the fork instruction.

```
if (pid > 0)
{ //parent code  cout<<"I am parent";
}
else if(pid==0)
 {//children code}{ cout<<"I am child";}
```

→ If we have one CPU, the time will be wasted in context switching (change of process to CPU to be executed)

→ The state of parent at the time fork is called is copied into child, i.e before fork call, if values of parent are changed then new values will be copied into child.

→ execution of code depends upon context switch.

→ Copy on write: If memory is not changed for processes, then that memory will be shared.

for immediate child

wait(); → ~~eg some~~ wait(NULL); ← variable is passed which tells the status of child that it is executed normally or abnormally.

e.g if parent has to do some calculations from the values child process will give, then parent will wait till child completes the process.

→ we can pass the id of the child in wait to check for which child parent is waiting.

wait(int *wstatus);

→ wait returns the id of the child process which is terminated.

waitpid(pid) → waits for the pid of the process passed in parameters.

Zombie process: The child process which is terminated in which there is no wait for it in parent. i.e child is completed and parent has not taken it's status.

Orphan process: Parent has terminated without calling wait on child. This process is adopted by init and init puts wait on it.

(without checking status of child)

Lecture 5:

```
int main()
{
    int status = execlp("./avg.out", "./avg.out", "23", "41", "53", NULL);
```

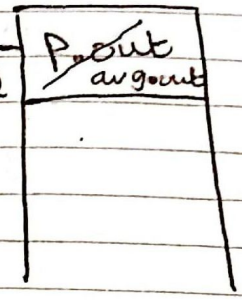
exec
execlp execvp

"./avg.out" → file name

"./avg.out" → process name at index 0

Operating Systems

avg.out will use the same memory as p.out i.e p.out will be overwritten by avg.out



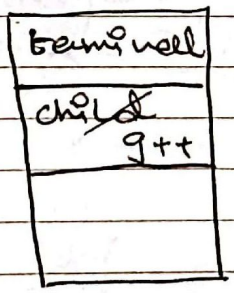

```
int main()
{
int status = execlp("./avg.out", "./avg.out", "23", "41", "20", NULL)
```

This line will not run because p.out is overwritten.

```
    cout << " process created";
    if ( status == -1)
        cout << " Execlp failed";
```

```
int status = execlp( "g++", "g++", "main.cpp", "-o", "a.out")
```

```
int main()
{ int id = fork();
  if (id == 0)
  { execlp("g++", "g++", "main.cpp", "-o", "a.out", NULL)
  }
else if (id > 0)
  { wait(NULL);
  }
```



execlp(command[0], command[0], command[1], command[2], NULL)

↳ This is not reliable in case of more arguments

execvp(command, argv) ← array of commands

```
int main()
{ cin >> command;
  int id = fork();
  if (id == 0)
  { execlp("g++", argv)
```

→ Khudi banani hai of **gargv

→ It should be of lenght n+2 ~~+2~~ because of command and NULL.

Lecture 6:

Pipes

- unnamed pipes
- named pipes

File discriptor table:

P1 →

0	occupied by default
1	
2	
3	qui2.txt, write, offset

Basic system calls:

→ open ( ) pass any of those indices

→ read ( )

→ write ( )

file discriptor (numeric value)

int fd = open("qui2.txt")

→ fd is discriptor of "qui2.txt".

write (3, "Hello world", 15)

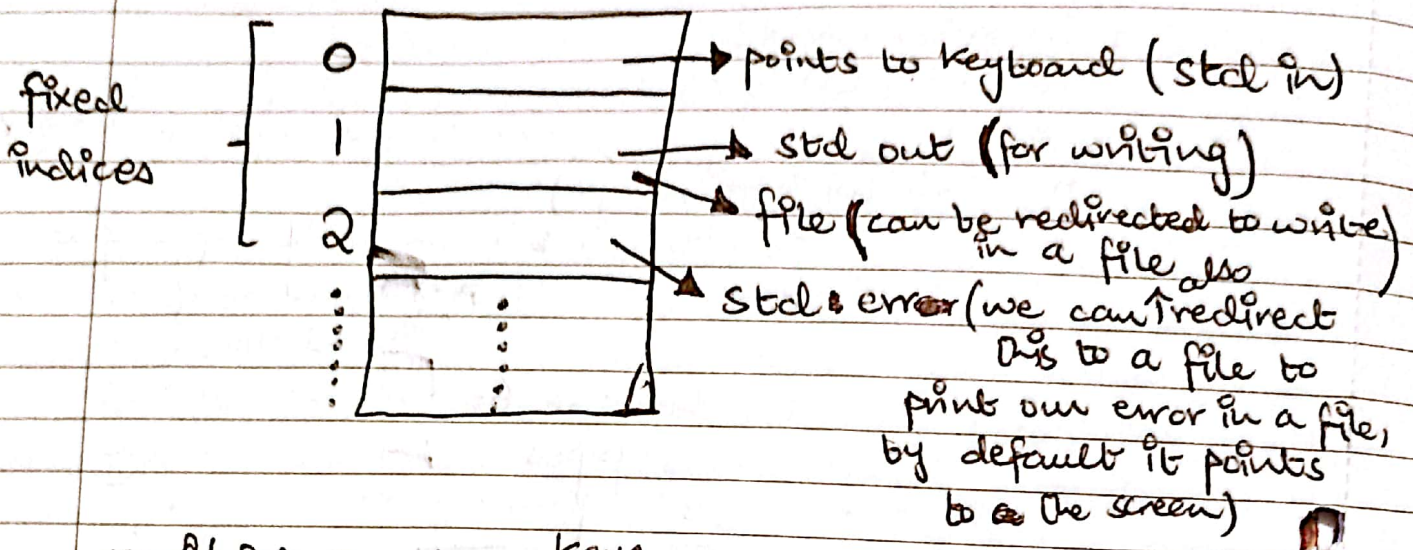
- 3 → file discriptor (fd)
- "Hello world" → message
- 15 → no of bytes

→ Now discriptor 3 can only be used to write because it is opened in write mode.

close(fd) → to free the index

we can point index 0, 1, 2 towards a file.
By default they point to the screen.

file descriptor table



~~read(fd, 0~~ char key;
read(0, &key, 1)
fd — address of variable — no of bytes to be read

→ read is a blocking system call.

returns no of bytes read. ← int read = read(0, &key, 5)

Interprocess communication [Between parent and child]: (memory)

Pipe: It is a buffer in which parent writes and child reads.

int fd[2];
int status = pipe(fd);

If status = 0 (pipe opened)
fd[0] → points to ~~read~~ read
fd[2] → point to write

```
int main()
{ int fd[2];
   int status = pipe(fd);

  if(status == -1)
      cout<< "pipe Error";
          return

 status = fork();
  { If(status == 0)
      { //child (only reads data)
          close(fd[1]); → because child only reads
        } char arr[12] = "HelloWorld"; parent doesn't write.
          read( int bytesRead = read(fd[0], arr, 12); }
   else if(status > 0)
      { //parent
          close(fd[0]); because parent doesn't read.
        { write(fd[1], "HelloWorld", 12)
          wait(NULL); close(fd[1]);
      }
     else
        { fork cout<< "Fork not opened";
            return 1;
        }
```

→ pipe is used before fork so that pipe contents are copied into child.

→ child inherits
→ data → instructions → file disciptor table (open file

size of( ) → tells us size of variable.

[Quiz next to Thursday till Lecture 7]

Lecture 7:

pipe → present in kernel memory

open() system call:

int open (const char* filename, int flags, mode_t mode);

- int: value → file descriptor, if failed returns -1.
- if file is already created
- flags: to tell write, read or both
- mode: if file is already made, we pass 0 here.

If we want to create file

flag for creation

int open("a.txt", O_WRONLY | O_CREAT, 0777);

mode value (gives permission to all users)

owner group permission → rest of users

0777

means it is an octal number

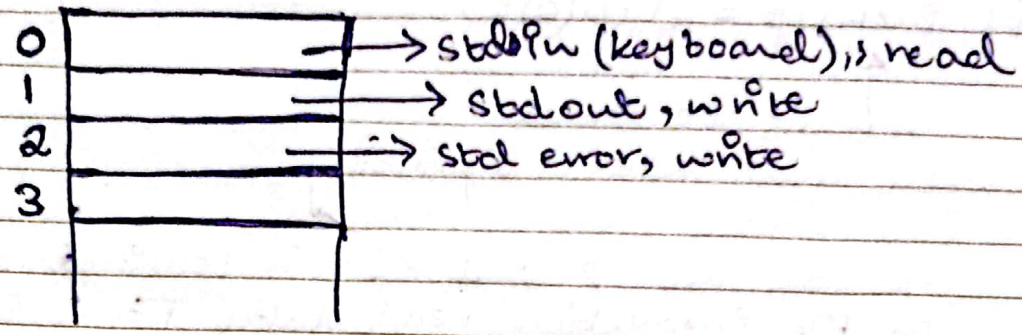
Number passed in mode:

0	7	7	7
↓	↓	↓	↓
octal number	permission of owner	permission of owner group	permission for rest of the users

O_EXCL → to write on a file which is not present and if file exists before then this system call is failed.

Lecture 8:

File discriptor table:




```
cin    read(0, arr, 10);
cout   write(1, arr, 10);
```

How to point index 0 to file

```
int main()
{  int readFileFd = ("./numbers.txt", O_RDONLY, 0777)
int writeFileFd  = (./result.txt", O_WRONLY, 0777)
int StdinBackup = dup(0);
int StdoutBackup = dup(1);

dup2(readFileFd, 0);
```

where to redirect it — index number

```
   dup2(writeFileFd, 1);

int main()
{     int FileFd = ("./numbers.txt", O_RDONLY, 0);
        dup2(FileFd, 0);
       int n;
       cin>>n
       int sum=0                    // now when we
      for(int i=0; i<n i++)           use cin, it takes
       {                              input from
       }                              "numbers.txt";
}
```

How to restore our index:

```
int Backup = dup(0);
```

↑ returns the index where our index 0 is pointing.

we can use > and < in terminal to specify the file instruction read data from. Ex ls < "file.txt"
> sign outputs into specified file.

< → input redirect > → output redirect

Lecture 9:

```
int main()
{ int status = fork();
  if (status == 0)
    {
      int fd = open("/sdsoutput.txt", O_CREAT, 0777)
      dup2(fd, 1) → only index 1 points to file
      execlp("ls", "ls", NULL);
    }
  else if (status > 0)
    { wait(NULL);
      cout << "Parent command executed";
    }
  return 0;
}
```

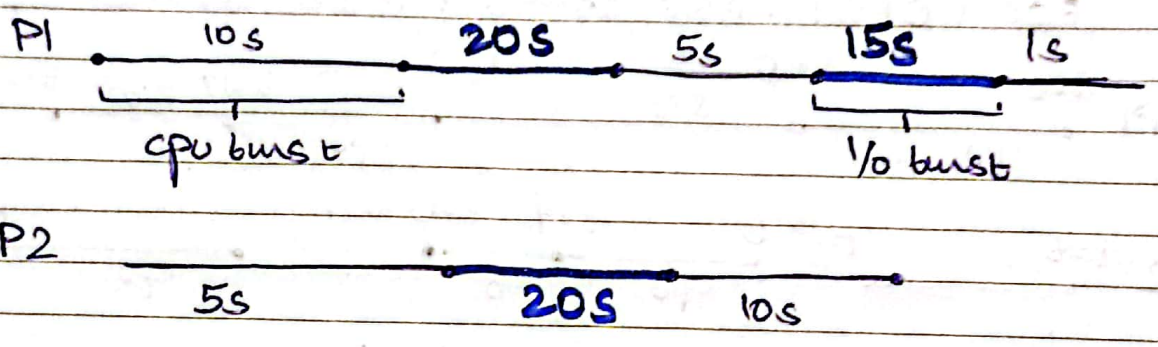
→ we will write in pipe by using 1 and not fd[1], so we close it.

grep is used to only select certain types of files for example grep.cpp only shows cpp files.

Lecture 9:

→ If there are n pipes, create n+1 childs.
→ Before calling wait(), close read and write ends of parent so if read end of child is open and write end of parent is open then the child thinks that parent would still write and doesn't close it's read end so the program gets stuck.

Lecture 10:



→ The scheduling algorithm in which the process gives up the CPU voluntarily (i.e until I/O burst comes) is known as non preemtive scheduling.

→ The scheduling algorithm in which we give cpu to a process for a certain time is known as preemtive scheduling.

Respone time → time process had to wait to get cpu first time

Turn around time → time taken by process to terminate

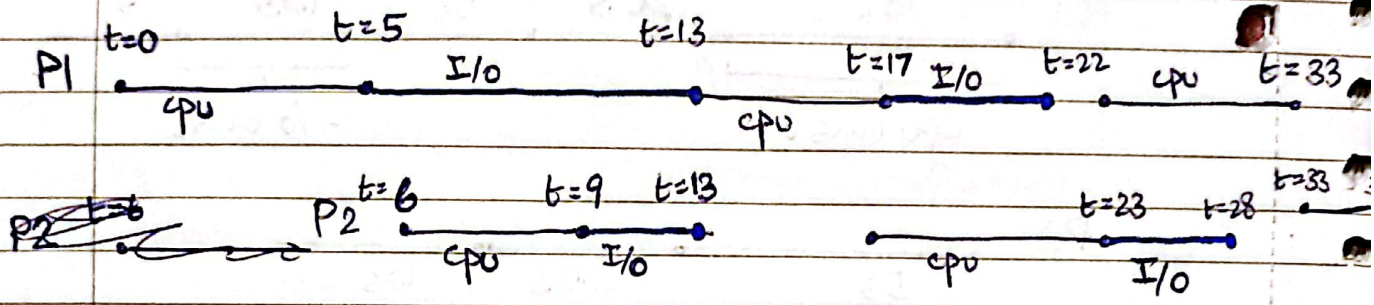
Waiting time → time spent by a process in ready que (I/O time is not included)

Scheduling Algorithms:

→ First come first serve (non-preamtive) [convey effect, I/O device wait
→ Shortest job first (non-preamtive)

Process	Arrival Time	CPU Burst	I/O burst	CPU burst	I/O burst	CPU burst
P1	0	5	8	4	5	10
P2	6	3	4	6	5	6

~~Ese~~ FCFS: First come First serve

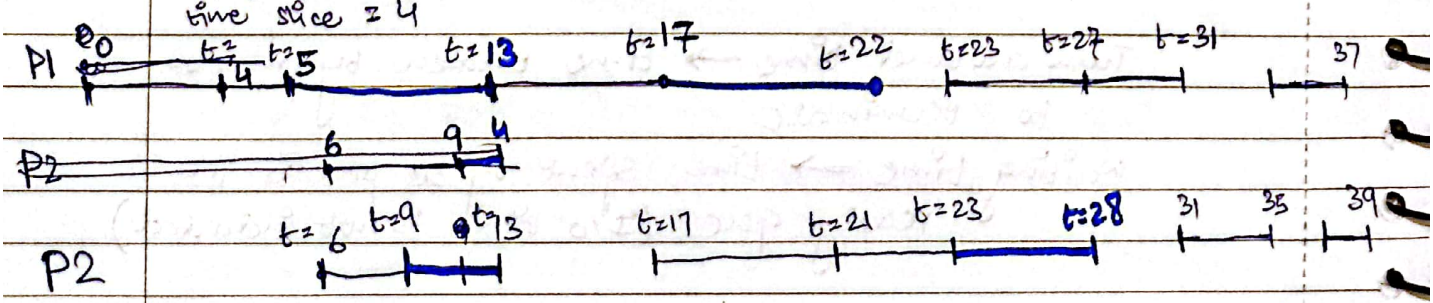


Response time: (P1) = 0s, P2 = 0s
Turn around time: P1 = 33s, P2 = 39-6 = 33s
waiting time: P1 = 1s, P2 = 9s

Lecture 10:

Process	Arrival Time	CPU Burst	I/O burst	CPU burst	I/O burst	CPU burst
P1	0	5	8	4	5	10
P2	6	3	4	6	5	6

Applying round robin: **Round Robin**
time slice = 4



→ gaps show that process is in ~~wait~~ ready queue.

Response time : $P_1 = 0$, $P_2 = 0$
waiting time : $P_1 = 5$, $P_2 = 9$
turn around time : $P_1 = 37$, $P_2 = 39 - 6 = 33$

→ If we are applying round robin and time slice ≥ average cpu burst then it acts like first come first serve scheduling.

→ time slice short → more context switching
→ time slice large → less context switching

Shortest Job First :

→ values of turnaround, waiting and response time are better than FCFS.
→ uses cpu and I/O better

Process	Time
P1	3
P2	8
P3	7
P4	3

⇒ There is no time slice in it, if P1 has 3 cpu burst and at t = 1sec, P5 comes with 1 cpu burst and it is less than remaining cpu burst of P1 i.e 2 sec is less than P5, so we will give cpu to P5 [because of high priority]

→ We predict that the next cpu burst of a specific process will be the same or close to the previous burst of that particular process.

Formula :

$$\tau_{n+1} = \alpha \cdot t_n + (1-\alpha)\tau_n \quad -(i)$$

- τ_{n+1} : next predicted burst
- α : wait
- t_n : actual cpu burst of previous step
- τ_n : predicted cpu burst of previous step.

Operating System

$\Rightarrow \tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$

putting in ①

$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \tau_{n-1}$

~~$\Rightarrow \tau_{n-1} = \alpha t_{n-1} +$~~ putting value of τ_{n-1}

$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + (1-\alpha)^3 \tau_{n-2}$

Final Formula

By equating, we get

$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 t_{n-2} + \dots + (1-\alpha)^k t_{n-k} + \dots + (1-\alpha)^{n+1} \tau_0$

$\Rightarrow \tau_2 = \alpha t_1 + (1-\alpha)\tau_1$

and $\tau_1 = \alpha t_0 + (1-\alpha)\tau_0$

$\Rightarrow \tau_2 = \alpha t_1 + (1-\alpha)\alpha\tau_0 + (1-\alpha)^2 \tau_0$

→ Jitna peeche jao utna wait kam hai
→ wait of recent will be more than the past. **[Exponential Moving Average]**
→ ~~This is a Exponential~~

Lecture 11:

Priority Scheduling:

setting priority to a process (1-10)

P1 = 1, P2 = 5, P3 = 7, P4 = 5, P5 = 1

It can be of two types

→ lower value means higher priority
→ higher value means higher priority

→ If algo is preambive and P1 is running and P5 comes, P1 will execute and then P5

→ If algo is preambive, and P1 is running with CPU burst (2) is running and P5 with cpu burst (1) comes, we will take cpu from P1 and give to P5.

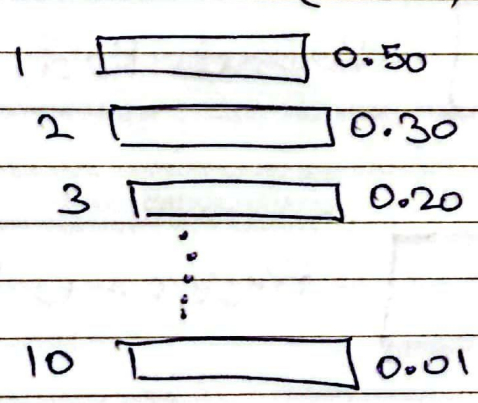
Starvation: when cpu is not a given to a particular process because of low priority.

Aging: we give the starved process high priority so that it may execute.

How to Implement
→ using max/min heap
→ using multiple queues

Multi level queue scheduling algorithm

Create no of queues equal to number of priorities



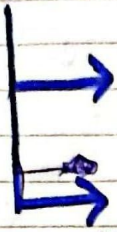
we will move to next queue when the previous one will be empty.

→ As we want to execute all processes, we will give certain time to each queue. And the one with higher priority will get more time.

Types of processes

→ Real-Time Process [highest priority] [output to be generated within a certain time i.e deadline]
→ System process [high priority] [Kernal process]
→ interactive process [Interaction with user]
→ batch process [runs in background]

Real time process

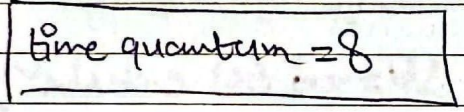


Hard real time: output to be generated within the deadline, if it doesn't happen then output is of no use

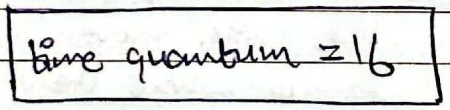
Soft real time: output to be generated within deadline but if output isn't given in deadline then it is fine.

→ so we can also create only 4 queues for these 4 types of processes.

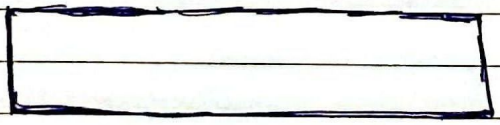
Multi-level Feedback queue scheduling



Round Robin



Round Robin



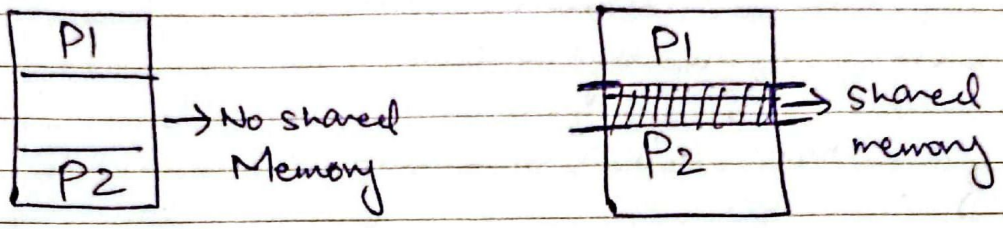
First come First come

→ process will not stay in a single queue. It can move to any queue.

→ In starting, every process is placed in quantum 8, if it has more cpu burst we will move it to 16 quantum queue and if it is still more then we will move it to FCFS queue and if cpu burst starts to become short we can move it to quantum 8 or 16 queue.

(chapter 5 pg 54)

Shared Memory:



P1	P2	if P1 writes 4 in a Den P2 will print 4
int a; cin>>a;	int a; cout<<a;	

→ If P2 runs before P1, garbage will be printed, same can also happen in case of context switch

C++ code
a=10;

Assembly code
mov bx, a
mov ax, 10
mov [bx], ax

→ To ensure no errors, we can shared a boolean variable flag

int a;
bool flag = false;

P1	P2
a = 10; flag = true;	while(!flag == false); ← busy wait cout << a;

P2 will stay in while loop till a is updated and flag is true.

→ busy wait is not very efficient as P2 will waste the cpu while waiting for a to be updated.

common set of instructions
↑ critical section → shared memory

Lecture 12:

```
withdraw( amount)
{ if (balance < amount) return
  else  // assembly code

      mov ax balance = - amount;
}
```

balance = 10,000

P1	P2 while(lock);
① → mov ax, [amount]	mov ax, [amount]
② → mov cx, [balance]	mov cx, [balance]
sub cx, ax	sub cx, ax
mov [balance], cx	mov [balance], cx

lock = false;

If ① and ② lines of P1 are executed then context is switched then we will not get the actual balance. In this case if P1 draws 1000 and P2 draws 3000 then actual balance will be 6000 whereas P1 will need 9000 which is wrong. It can be solved by using lock.

Possible Solution

P1	P2
while(lock == true);	while(lock == true);
lock = true	lock = true;
mov ax, [amount]	mov ax, [amount]
mov cx, [balance]	mov cx, [balance]
sub cx, ax	sub cx, ax
mov [balance], cx	mov [balance], cx
lock = false;	lock = false.

This solution still doesn't ensure mutual exclusion. [only one process in critical section]

means when it is running, there is no context switch

↑

```
bool test_and_set(bool * lock) ──→ atomic
{
    r = lock;
    * lock = true;
    return r;
}
```

sets locks == true and returns the previous value.

P1

while(test.and.set == true);	while(test.and.set == true);
// lines of code	// lines of code
lock = false	lock = false

→ Now using test and set instruction, we insure mutual exclusion i.e when P1 comes, it sets lock = true and if contact switch happens, & test and set in P2 returns true and stays in a loop until P1 changes the value of lock to false.

→ Mutual exclusion is acheived but busy wait is still not resolved.

→ There should be bounded wait.

→ we use **mutex** tool which contains two functions aquire() & and release() which are both atomic. aquire() makes the lock true but first checks that lock is already true or not. If lock is true, we maintain a mutex queue (waiting queue) so we can give aquire() function to the first process in the queue when the other process calls release(). ~~In this~~ So by using this, we aquire bounded wait.

mutex m1, m2

P1	P2	P3
while (true)	while (true) ~~while(true)~~	while (true)
{ aquire(m0);	aquire(m1);	{ aquire(m2);
~~cout~~		cout<< "c";
cout<<'a"	cout<< "b";	release(m0);
release(m1);	release(m2);	

Lecture 13:

Nameed Pipes:

mkfifo(fileName, 0777) → permission

int fd = open(fileName, O_WRONLY, 0)

~~fd~~ P1	P2
int fd = open(fileName, O_WRONLY, 0)	int fd = open(fileName, O_RDONLY)
write(fd, "Hello", 5);	read = (fd, buffer, 5);
close(fd);	

Mutex

```
aquire()
{  if (lock == false)
        lock = true
   else
     { remove p from ready queue;
       add p to waiting queue; }
}
release()
{  lock = false;
   if( waiting queue not empty)
   {    pop from waiting queue; }
}
```

→ both aquire and release work atomically

P1	P2
while(1)	while(1)
aquire(m)	aquire(m)
——	——
——	——
——	——
——	——
release(m)	release(m)

→ If P1 aquires and then releases mutex and there is no context switch, P1 will aquire mutex again so in this way bounded wait condition would not be satisfied.

Rendezvous Problem:

P1	P2
a	c
lock2 = false	lock1 = false
while(lock1 == true);	while(lock2 == true);
b	d

we want that a should be executed when c is executed and d is executed when a is executed

lock1 = ~~false~~ true, lock2 = ~~false~~ true

Synchronization ensures
→ mutual exclusion
→ bounded wait
→ progress (i.e no deadlock)

Barrier Problem [Generalized rendezvous problem]

→ Jab tak same process a problem nhi chala lete tab tk b ni chale ga

P1	P2
a	a
b	b

Mutex Rendezous Problem

m_1 = true , m_2 = true

P1	P2
a	a
release(m_2)	release(m_1)
aquire(m_1)	aquire(m_2)
b	b

→ First a of P1 will be executed then P1 will release m_2 and b of P2 will aquire it and execute b of P2. Same will be with a of P2 and b of P1.

Barrier Problem [Generalized rendezvous problem]

→ b statement of any process will not be executed until a statement of all processes is executed

```
int counter = 0;        mutex m = false;
    m2 = true;
P1
aquire(m);
counter++;
release(m);
if (counter == n) { release(m2); }
    aquire(m2);
    release(m2);
        b
```

Lecture 14: Semaphores

```
class semaphore
{ int counter;
  public:
      wait(); ⟶ aquire
      post()/signal(); ⟶ release
}

wait()
{ if(counter == 0)
    { ⟶remove P1 from ready queue;
      ⟶push p into waiting queue;
    }
  else
    { counter--; }
}

post()
{ counter++;
  if(waiting queue is not empty)
  { p = waiting_queue.pop();
    ready_queue.push(p);
  }
}

semaphore a = 10;
wait(a);        // 20 processes.
                // as semaphore is 10, so at a time
                   10 processes can use semaphore
post(a);
```

⟶ To provide mutual exclusion, we set semaphore value to 1.

Binary semaphore: Semaphore with value 1 (works as mutex)

counting semaphore: Semaphore having value greater than 1.

~~Producer Consumer Problem:~~
~~Bounded buffer:~~

Producer Consumer Problem:

```
int queue[10];
int total_elements=0
```

Producer

```
while(1);
{ int p-counter=0
while(total elements != size); //busywait
{ queue[p-counter] = produceitem();
p-counter++ =(p-counter+1)% size;
totalelements++; }
```

```
int c-counter = 0
```

Consumer

```
while(1)
{ while(totalElements == 0); wait
cout<< queue[c-counter];
c-counter =(c-counter+1)% size;
totalElements--;
}
```

→ We have to provide mutual exclusion here.
~~If producer produces,~~ Producer and consumer function should both be atomic.

Now Solving using Semaphore:

```
semaphore producer = SIZE
int p-counter =0
while(1)
{ wait(producer)
queue[p-counter] = produceitem();
p-counter =(p-counter+1)% SIZE;
post(consumer);
}
```

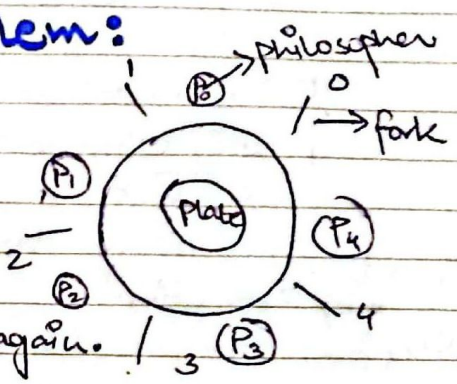
consumer will work when produce will produce something

```
semaphore consumer = 0;
int c-counter = 0;
while(1)
{ wait(consumer)
cout<< queue[c-counter];
c-counter =(c-counter+1) % SIZE;
post(producer);
```

producer will produce something when consumer will consume.

Lecture 15:

Dining Philosophous Problem:



Philosopher is thinking. When he gets tired he picks up his left and right fork and ~~then goes~~ eat food and then goes to thinking again.

Total n philosophers

```
philosophers(i)
{   while(1)
    {  Think();
       Pickfork(i); // pick left fork
       Pick fork((i+1)%.w); // pick right fork
        eat();
       put fork(i);      // put is left fork
       put fork((i+1)%.w); // put right fork
    }
}
```

Now we will synchronize by using semaphore

```
Semaphore forks[n] = {1}; // 1 value means fork is on the table

philosophers(i)
{ while(1);
  { Think(); (forks[i]);
     wait ;  // picks up left fork, if it not there then waits for it
     wait ; (forks[(i+1)%n]);
     eat();
post(forks[i]); ; // puts down left fork
          ; post(forks[(i+1)%n]);
  }
}
```

We may face a problem in which P0 picks fork 0 then context switch happens and P_1 picks fork 1 (left) and when wants to pick fork 2 (right) then again context switch happens and then this gets stuck in a deadlock.

We will solve this by making the both waits atomic if we apply mutex at ~~wa~~ when it's is ~~Philosopher(i)~~ picks up fork and mutex to when ~~{ while(1)}~~ it release it. But this also has a issue that no one can pick or put forks at the same time (no parallelism)

Final Solution → Our final solution is that each philosopher will pick first left fork and then right fork but the last philosopher will first pick right fork then left fork so there is no deadlock.

```
Philosophers(i)
{   while(i)
    {  think();
       if(i==n-1)  // last philosopher
       {  wait(forks[0]); // first picks right
          wait(forks[i]); // then picks left
       }
       else
          {  wait(forks[i]);  // first picks left
             wait(forks[i+1]);  // then picks right
          }
       eat();
       post(forks[i]);  // puts down left fork
       post(forks[(i+1)%n]); // then puts down right
    }
}
```

Now our problem is solved. There is no deadlock. Because we have changed the order of picking up forks of the last philosopher.

~~Methods~~ Methods to share memory:

→ pipes
→ shared memory [Fastest method]
→ Filing

Lecture 16:

Server

```
while(1)
{   client = waitforclientrequest();
    int id = fork();
    if(id == 0)  // child process
    {   serviceClient(client);
    }
}
```

accepts the request

data required by client e.g gmail, file etc

handles client requests and provides him the data that is required.

→ if we create a process multiple times then it uses its own CS, DS, SS, PCB and file descriptor table. so we use threads.

→ In threads, CS, DS, Filedescriptor, is same. But TCB(Thread control block) is different. Also SS will also be different because it controls function calls and local variables are also different because stack keeps a record of them.

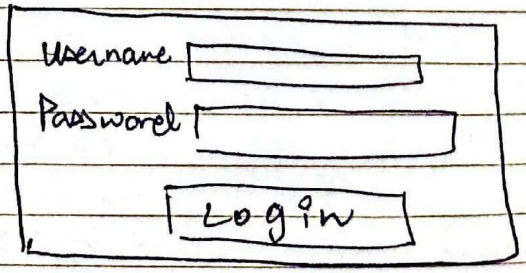
→ We will also communicate between threads. we can do it by changing global variables, because DS is same.

→ So now we synchronize threads instead of processes which is easy.

→ When task is logically similar, we use threads.

→ Threading is used for parallelism and concurrency.

→ Also used in GUI.



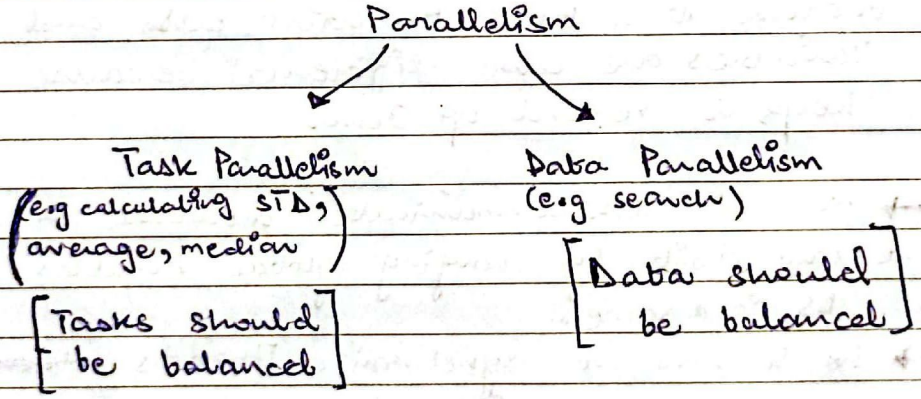
This function will run on GUI Thread.

```
onMouseClick()
{   userID = getbyID(userIDTextfield).value;   // gets everything written in username text box
    password = getbyID(passwordTextField).value;   // gets everything from password box.
    return DB.validateLogin(userID, password);   // checks password and username in DB.
}
```

→ we can make it more efficient by running DB.validateLogin in another thread.

How to do Threading:

→ Identify tasks which can be threaded
→ Balancing [giving equal instruction]
→ Handle dependencies through synchronization



Lecture 17:

Threads:

```
void * funct(void * param)
    int val = *( (int*) param);
}
int main()
{   pthread_t tid;
    status = pthread_create(&tid, NULL, &func, &num);
```

dereferencing of void pointer

type of integer we sent

parameter we will send in the function

[If we want to create 10 threads, we will create 10 thread variables]

tid → ID of thread

NULL → system will assign default values to attributes (i.e stack size etc)

&func → address of function we want to execute

`pthread_join(tid, NULL);` → This is wait call i.e waits for the thread

→ If we want to check status, we can pass a 2d pointer instead of NULL in pthread_join.

joinable thread : A thread which is joinable and can not become zombie process.

Detached thread : A thread which is not joinable and cannot become a zombie process. [We don't check it's execution status]

`pthread_detached(tid);`

Lecture 18:

User Thread And Kernel Thread:

User controls thread

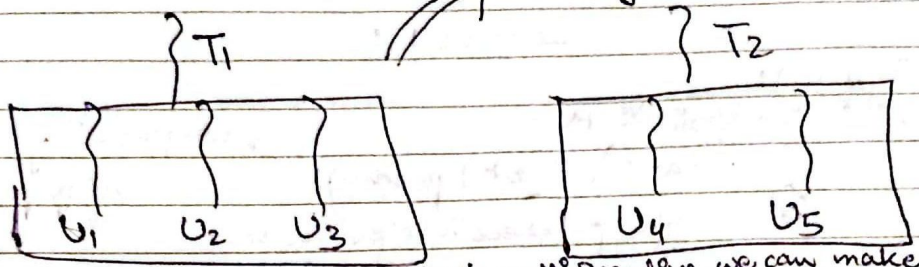
[Faster context switch because there is no mode shift]

Kernal controls threads

[Slow context switch because there is mode shifting]

User and Kernal Mode

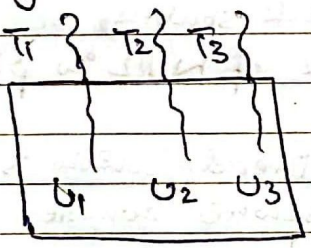
Many-to-one Model



→ If a ~~process~~CPU doesn't support multithreading, we can make user threads by user library.

Pros →
- → Kernal only schedules the kernal threads T1 and T2
- → we will schedule the user threads i.e U1...U5 in ~~with the~~ the time ~~CPU~~ Kernal will give CPU to thread T1 OR T2. [Faster context switch]

Cons →
- → There is no parallelism in user thread because as there is only one kernal thread, so it will ~~only~~ only get 1 CPU.
- → If a certain user thread will call a blocking system then the kernal will block the kernal thread i.e T1 OR T2 instead of blocking that single user thread e.g U1, U2.

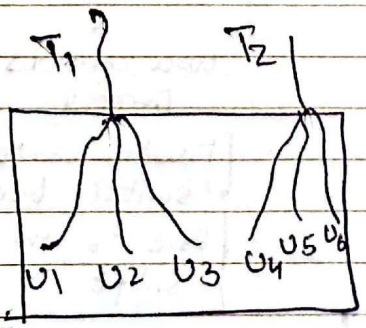


⟹ One to one model

Pros →
- → We can achieve parallelism, each thread can run on different CPU.
- → If a user thread calls a system block, only that ~~user~~ kernal thread will be block.

Cons → More resources required on the kernal side.

Many to Many Model ⟸



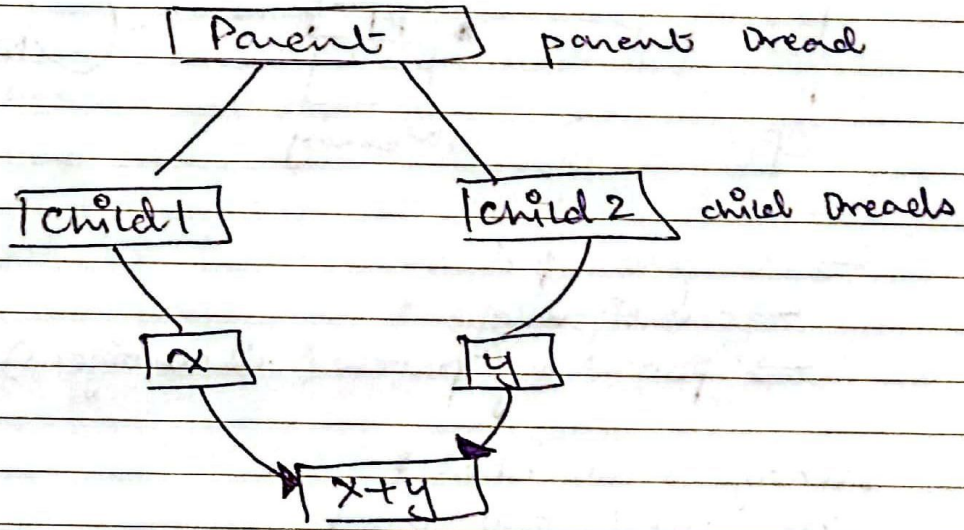
Pros → parallelism and faster context switching

Thread Pool:

→ already initialized n number of threads. We just have to request the thread pool to run any function

Pros → no need to initialize and destroy threads

→ If there are no free threads in thread pool and request for a thread to run a function, thread pool will save this request and run it when a thread will be free.



→ We will make parent wait by using pthread_join(tid, NULL);

→ Parent has to wait because he has to combine the results of child1 and child2.

→ This model is known as fork join model.

→ It is also a synchronous setting model as parent has to wait for childs to execute.

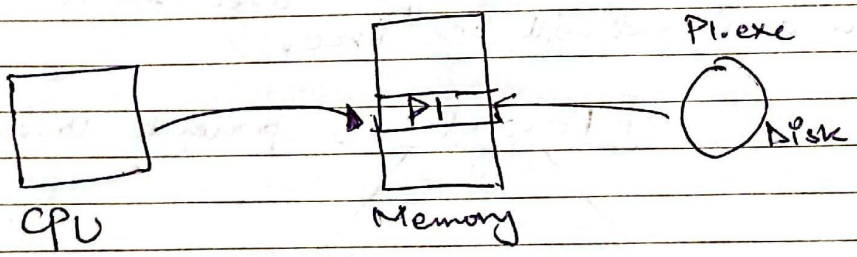
→ If parent doesn't wait for childs to execute then it is asynchronous model.

4.1, 4.2, 4.3, 4.5.1, 4.5.2, 4.6.1

→ If a thread calls a fork(), there are two options
① All threads in parent will be in child
② The thread which only called fork will be in child process.
→ Linux uses the ② option.

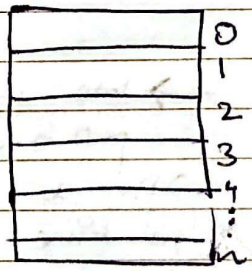
→ If a thread calls execlp then the threads will be destroyed.

Main Memory:



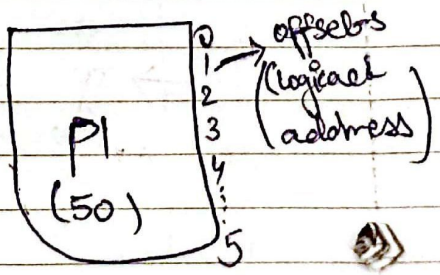
→ we put data
→ contiguously
→ patch by patch (discontiguous) [used in modern computer]

Contigous allocation:
We need two things
→ P1 physical addresses's first byte [starting byte location]
→ Total bytes in P1

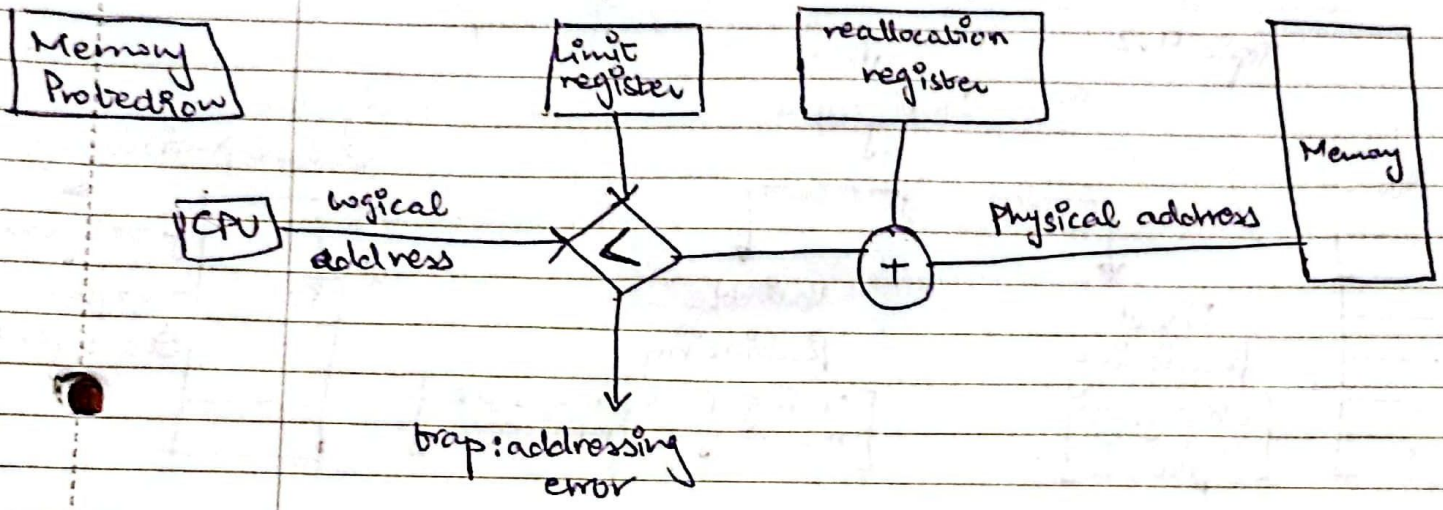


physical address

physical address = base address + offset
→ process issues only logical address
Every process has
① base register [starting physical address]
② Limit register [Total size of process (bytes)]



→ If a process tries to access illegal memory then CPU will generate an exception.



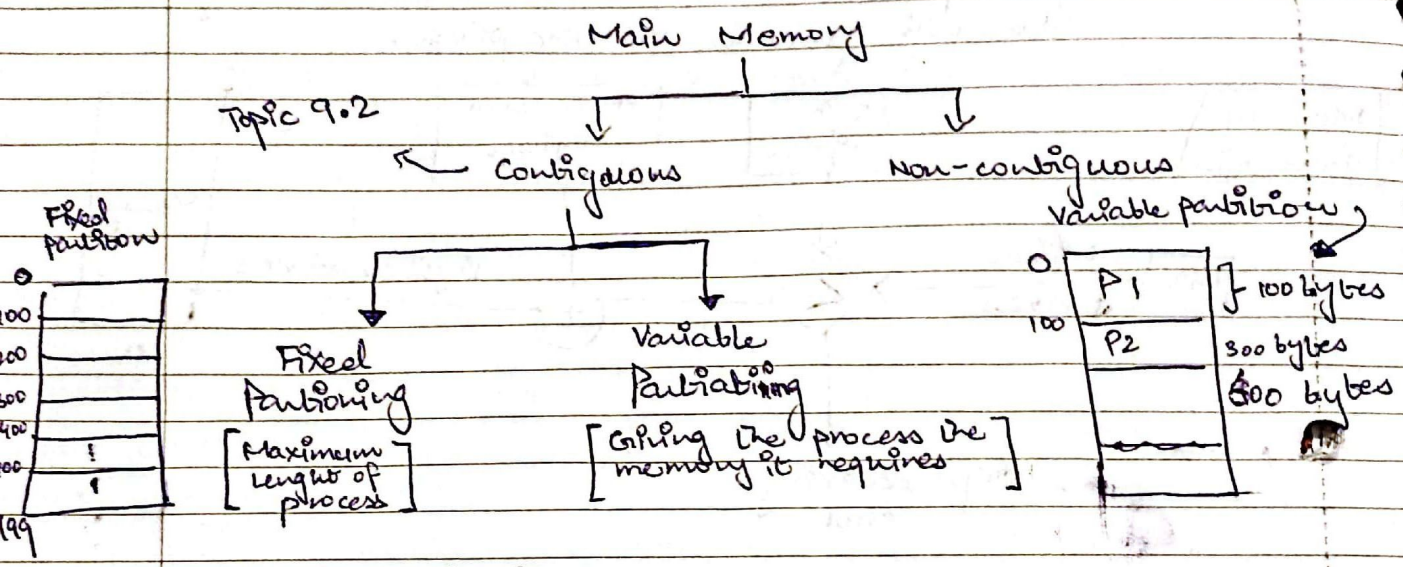
→ If logical address is greater than limit register, then it goes to trap.

→ Base register is called reallocating register because the starting physical address changes when process is moved in and out of memory

→ Modern operating systems issue logical address.

Lecture 19:

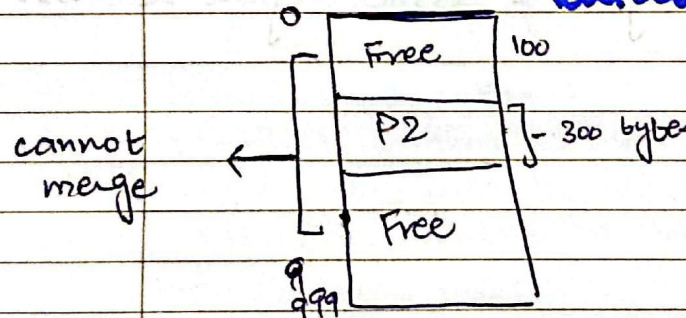
Topic 9.2



Pros of fixed partioning:
→ Easy Management

Cons:
→ internal fragmentation: Memory wasted inside a fragmentation:
→ We don't know the maximum size of process

Variable Partioning:



~~If we have 3 partitions of 100, 90, 300 and we have allo~~

⇒ So Now we have 3 ~~way~~ ways to deal with this

① Best Fit option
② Worst Fit
③ First Fit

If we have 3 partitions of 100, 90, 300 and we want to allocate 80 bytes

90 (best fit) 300 (worst fit) 100 (first fit)

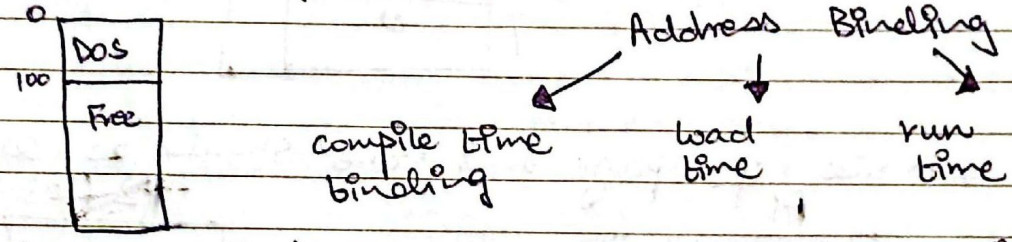
Cons of Variable Partioning:

→ External Fragmentation: When we have free to allocate to a process but we cannot because they are not together.

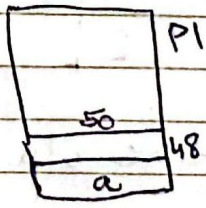
Solution → Compaction: we combine the free ~~free~~ partitions of memory together to avoid fragmentation.

Topic 9.1.2 Address Binding:

→ If CPU only runs one process at a time, then



→ So now we know every time our process will start from 100 so now we can generate physical address at compile time.



we can now use

direct [148]

↳ physical address

① compile time binding: In which we use the physical address of variable in executable.
[same starting physical address] [CPU runs only 1 process]

Process will stay at the same memory until it is executed.

② Load time binding: when we want to load the process into memory we convert the logical address of executable into physical addresses.

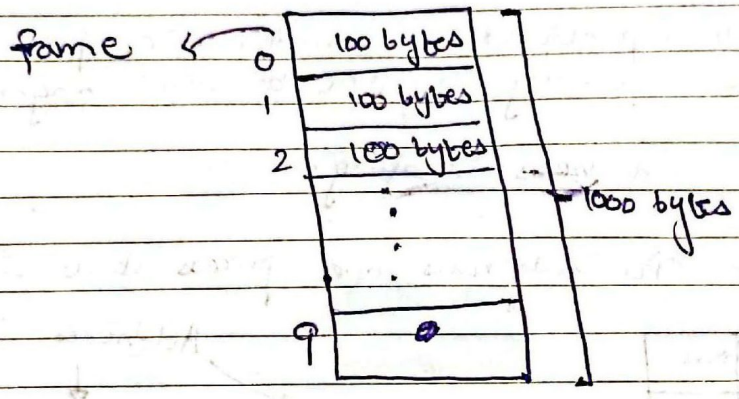
③ Run time binding: When we convert the logical address into physical address at run time
[So here are no problems regarding physical address if we give the memory of a process to another process for some time]

→ Modren operating systems use run time binding.

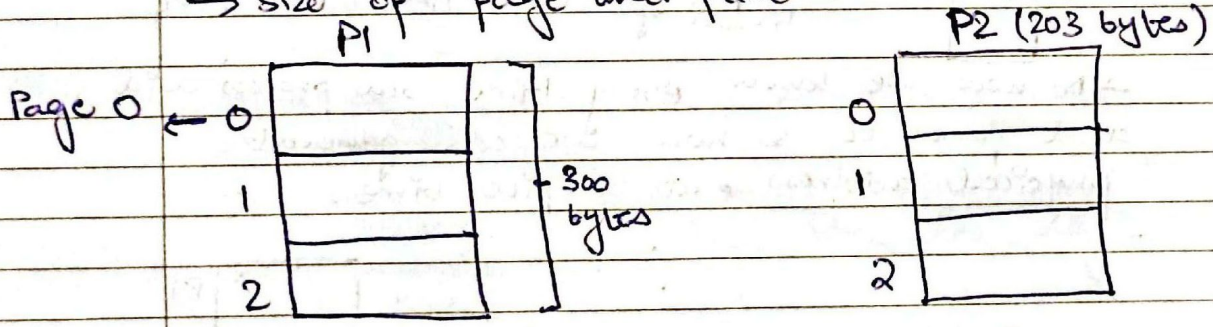
Lecture 20:

Non contiguous Memory Allocation:

Virtual Memory: [some portion of data is in harddisk]

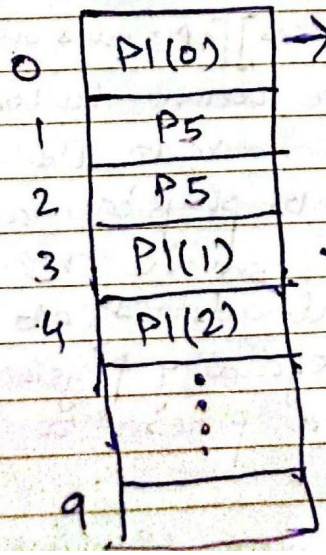


→ size of page and frame should be same.



→ We will load page in a frame

We can load P1 as.



→ Page 0 of P1 is loaded at frame 0, page 1 at frame 3, Page 2 at frame 4.

→ Every process has a page table which keeps record of which frames the pages are loaded.

→ When calculating physical address, we first check the page no, then the corresponding frame number, then the offset in that frame.

→ We will calculate page no as

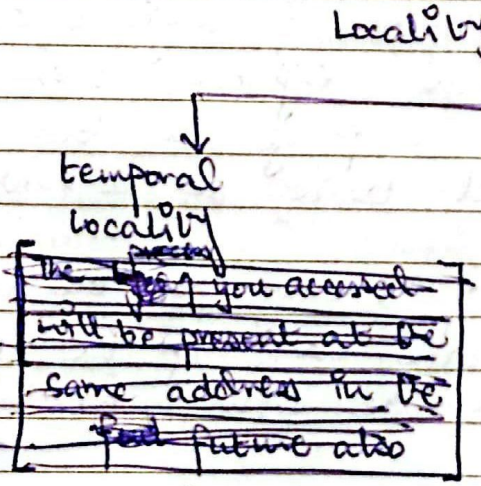
$$\text{Page no} = \left\lfloor \frac{\text{address}}{\text{size of page}} \right\rfloor$$

offset = address % size of page

* **Physical address = Frame No × Frame size + offset**

→ If the process is large, we can load only its main() function in a frame and keep the rest of in a hard drive. The other parts will be loaded in some frame when required.

→ Page fault exception comes when process will generate an address which is not present in RAM but in hard disk. Then interrupt will be generated and it will load that corresponding page into a frame and continue its execution.



Temporal locality: The process will access the same byte it accessed in the future. [No page fault exception]

Spacial locality: The process will access near byte near to the byte it accessed in the future. [No page fault exception]

process → paging frame size = page size
RAM → frames

Demand ~~Paging~~: Paging: Load page ~~whenever~~ only when it is required.
Thrashing: When page fault comes after every instruction [makes the system slow]

Lecture 21:

Main Memory:
Non-contiguous → paging

→ If we have address register of 32-bit, we can generate an address in the range $0 - 2^{32} - 1$

→ How can we ensure to run 4GB system on 2 GB?

2GB
100 pages

4GB
200 pages

→ we don't have to load all pages immediately, we will load only the page we need.

→ If a process generates a logical address greater than limit register, then this will create a page number which is not allocated to it so it will crash.

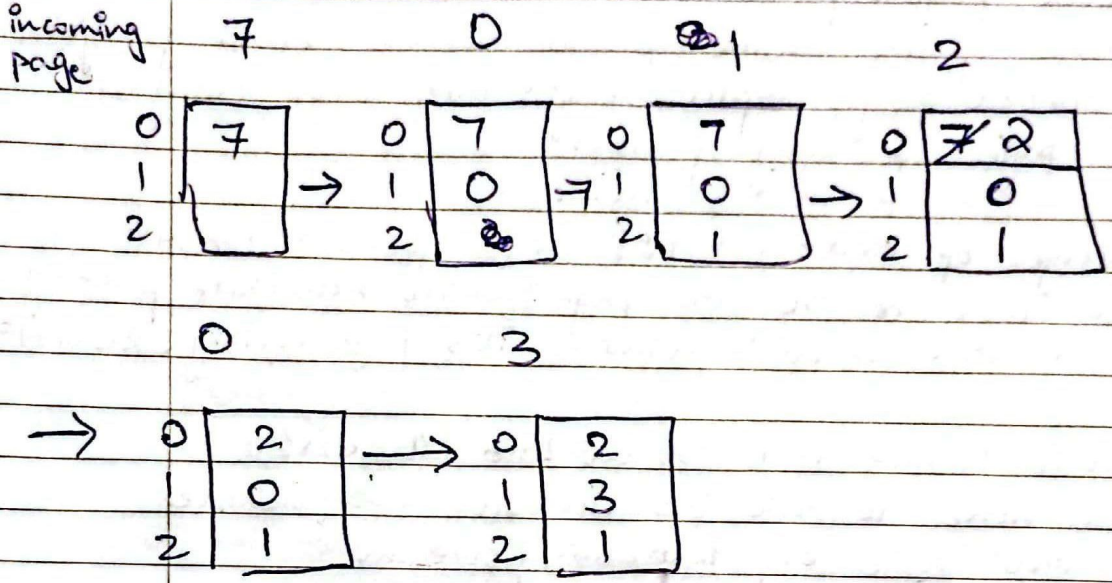
→ Memory sharing is only possible when atleast one page of the processes are loaded on the same frame.

→ There is no need to check for memory protection because process will not generate address greater than range ($0 \rightarrow 2^{32} - 1$) [32 bit register]

Page replacement Algorithm

① FIFO - page replacement:

→ The page loaded in the memory first will be removed.



→ The string of page i.e 7, 0, 1, 2, 0, 3 is known as reference string.

Disadvantages:

→ Many page faults occur [due to temporal and spacial locality]

Lecture 22:

② Optimal Page Replacement:

→ we will remove the page which has the least priority. [Not to used for longest period of time]

→ This cannot be implement because we don't know about the pages we will load in the future.

→ The best algorithm. It is used as a bench mark to check performance of an algorithm.

→ Very less page fault exception occur.

③ LRU [Least recently used] page replacement:

→ Remove the page which is least recently used by looking at the history.

→ Less page fault exceptions than FIFO algorithm. but more than optimal.

Page replacement Algorithm:

① local page replacement: we will remove the page of the same process we are executing right now.

② Global page replacement: We can remove the page of any process.

Advantages of ~~local~~ global: of the process least → we can remove the page which has priority and then we will remove its page using LRU or FIFO.

Dynamic Linking And Shared ~~Lab~~ Libraries:

→ we have to link the libraries we include in our code e.g iostream, fstream

① Static Linking: of library

→ the function we called in our executable will be present in our executable too. We will either write the code of whole library in our code ~~and~~ or the functions that we need in our code.

→ External library is embedded in our code.

Pros → This file can execute even if the system doesn't have that library because we have written.

Cons → ~~Size~~ File size is too much. code can duplicate in our memory in different processes.

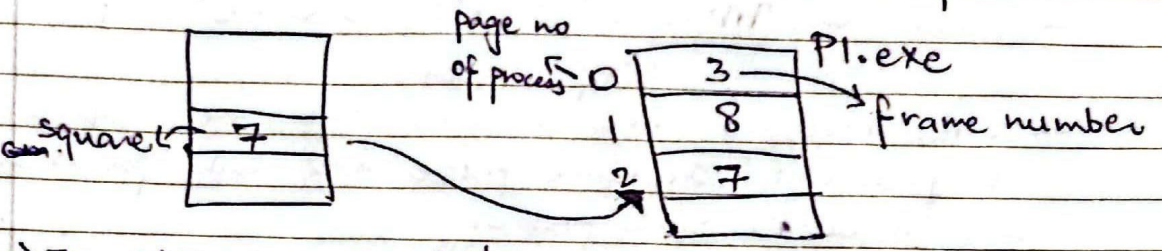
② Dynamic Linking:

→ It will not embed the library we need in our executable. It will not be present in our code.

→ While running the code, when we will execute .square() or strlen() functions, it will create a stub to tell where that function is present.

→ It will load the libraries we need in memory.

→ Library is present at a frame. When the process needs to execute it, it will make that frame a part of their address space.

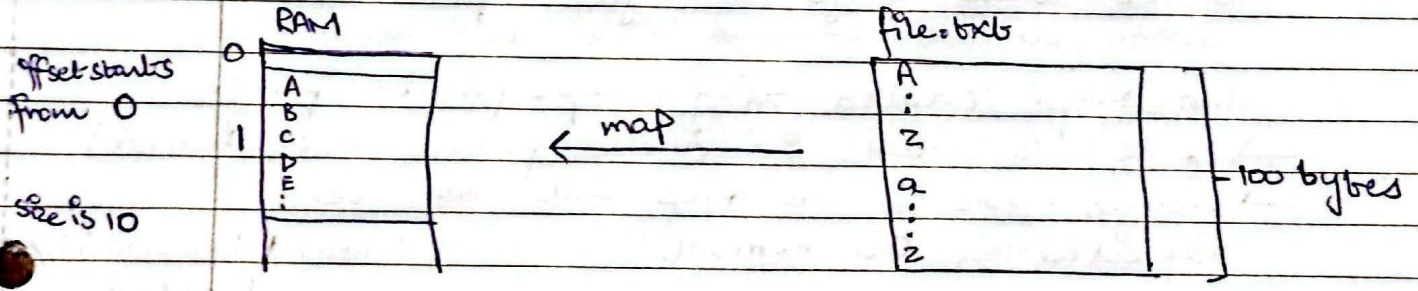


Pros → It uses less memory.

Lecture 23:

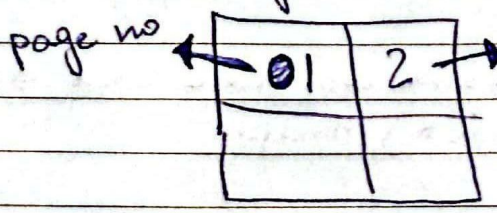
Memory Mapped Files:

→ previously, we used read() and write system calls.
→ a block of memory is read or written.
→ we open the map and we can read or write like we do in array.



→ we need a char pointer now to handle data because we have char data.
stores starting address of map ← * map = mmap( ) → system call to create map
be shared
→ changes in map will also be done in the file.
→ If A1.exe opens the file

Page Table



→ If a process P2 wants to make the same map P1, then map can be either shared or process has to make a new map.

→ Process use ~~Map~~ Map_Shared flag to share map

~~Map Shared~~

→ If two processes want to share map then map will be present in the same frame and both processes will access the same frame.

Pros → filing is fast and easy [because we deal it as a array]

→ data can be shared

→ Minimum memory that can be assigned to a map is size of one frame or page.

→ If ~~our~~ the size of our map is 10 bytes but a whole page of 100 bytes is loaded because frame is assigned then we can also access the bytes of the whole page through map.

Private Map: created using map_private flag.

→ ~~If~~ It is made in another frame (not shared)

→ changes made in the map will not be reflected in the memory.

PROT_READ | PROT_WRITE

NULL | size | read and write access

void * mmap (void * addr, size_t lenght, int prot, int flag (MAP_SHARED OR MAP_PRIVATE), int fd, offset offset)

file descriptor of opened file

→ same permissions must be for prot and fd regarding read and write else map won't be created

c convert into char* because we have char data

char * maps = (char *) mmap

munmap(map, 10) → This system call will remove the map from address space of the process.
(10 ↑ size of map)

Lecture 24:

→ If page is in memory, we access memory two times (one for frame no other for physical address) to bring actual data)
→ cache (frequently useed data)

→ Translation look-aside buffer: Cache for page Table
→ we will first look for frame no of a logical address in cache (translation look-aside buffer) and this takes very very less time or no time.
→ If page no is not in cache, we go to the memory and also put page no and frame no in cache.
→ If page no is found in cache, then it is cache hit and if not then cache miss.
→ If page table is in cache then we access the memory only one to find data at that physical address and if it is a cache miss then we have to access memory two times. One time to find frame no and bring data from physical address.

If 80% cache hit and 20% cache miss and memory access time is 10ns. Then

Effective access time = (0.80)(10) + (0.20)(20) = 12ns
(Faster access time)

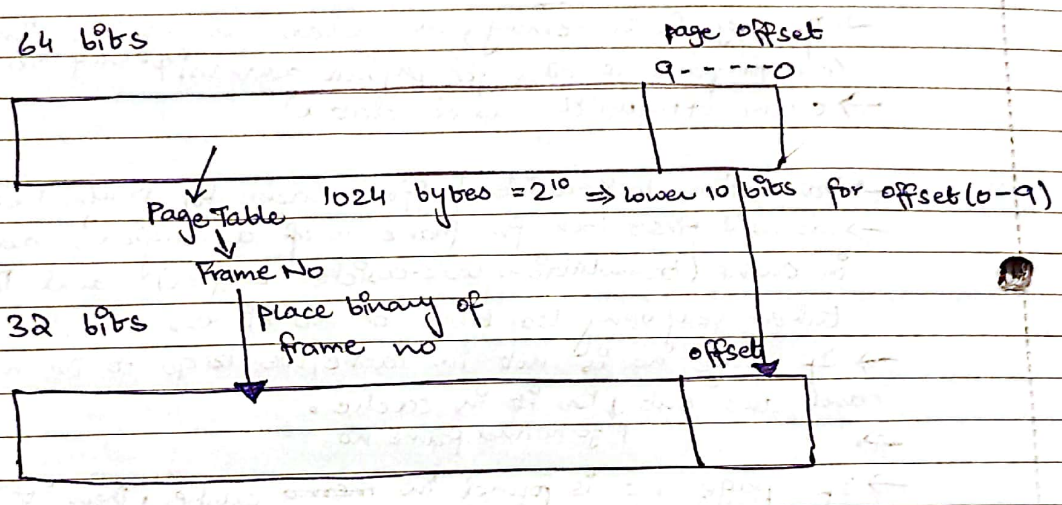
→ We can also find page No and offset as If we have our logical address as 2^m bytes and our page size can be expressed in 2^n i.e $64 = 2^6$ and $128 = 2^7$. In 2^m, m is no of bits to store logical address. So,

page offset = n
page size = m − n

~~If we have found~~

32 bits physical address space

64 bits logical address space



Physical address = binary of frame no + offset