OPERATING SYSTEMS SAAD FAROOQ

→ Every device has its own ready queue
→ Device controller is used to read data from file in a disk.
→ Proccesses read data from RAM
→ Transfer data directly to RAM (DMA) other CPU (Advantage)
→ need to move data into RAM
→ Every device hos its own buffer and controller
→ PCB (Process Control Block)
→ CPU Scheduler is used for deciding ~~which process~~ which process takes CPU (short term Scheduler)
→ Job Scheduler (Long term Scheduler)
→ Context switching is doone by dispatcher.
→ Dispatcher is used to load envirament of the process
→ Antivirus, window update → non interactive process.
→ Game playing → interactive process.
→ Interactive processes are given more priority because these processes interact with user more frequently
→

→ Process Management system:

```
int main(){
    cout << "good morning";   // 1 time
    int pid = fork();          // → create copy of the process
    cout << "Hello world";
    return 0;
}
```

execution start from after fork()

- Parent
- pid > 0 (child's id)
- child
- pid == 0

→ Concurrent execution in fork()

→

good morning	good morning
I am child	I am parent
I am parent	I am child

→ Prèvileged operations

→ kernal only able to execute previleged instructions.

→

F~~I~~X~~t~~

FAST

1 credit hr | 50 Rs | 1 credit

OPERATING SYSTEM

16 Rs

→ Kernal mode has direct access to the memory and hardware.
→ I/O operation always in Kernal Mode.
→ Kernal code always run in Kernal mode
→ Switching from user to Kernal Mode.
→ Memory protection

→ If user allowed to access kernal memory then system generate trap
→ System calls → kernal mode.
→ Command line arguments
./a.out hello world
arr[0] = ./a.out
arr[1] = hello
arr[2] = world

→ mkdir is used u to make directory

→ atoi( ) → used to convert charters into numbers

→

OS LECTURE 3

→ execlp

→ execlp(".\sum.out","\",",","2","10",NULL);

index 0

Path that we want to execute

Arguments end

→ ~~mon cp~~ mon execlp

→ g++ exclp.cpp

→ a.out memory override with sum.out

→

→ int main()

int id=fork();

if(id==0)

int val = exclp(".\sum.out",".\sum.out","1","2",
".3",NULL);

}

else if(id>0) → wait(NULL);

{

cout<<"exclp called"<<endl;

}

Am 9 a.m a per

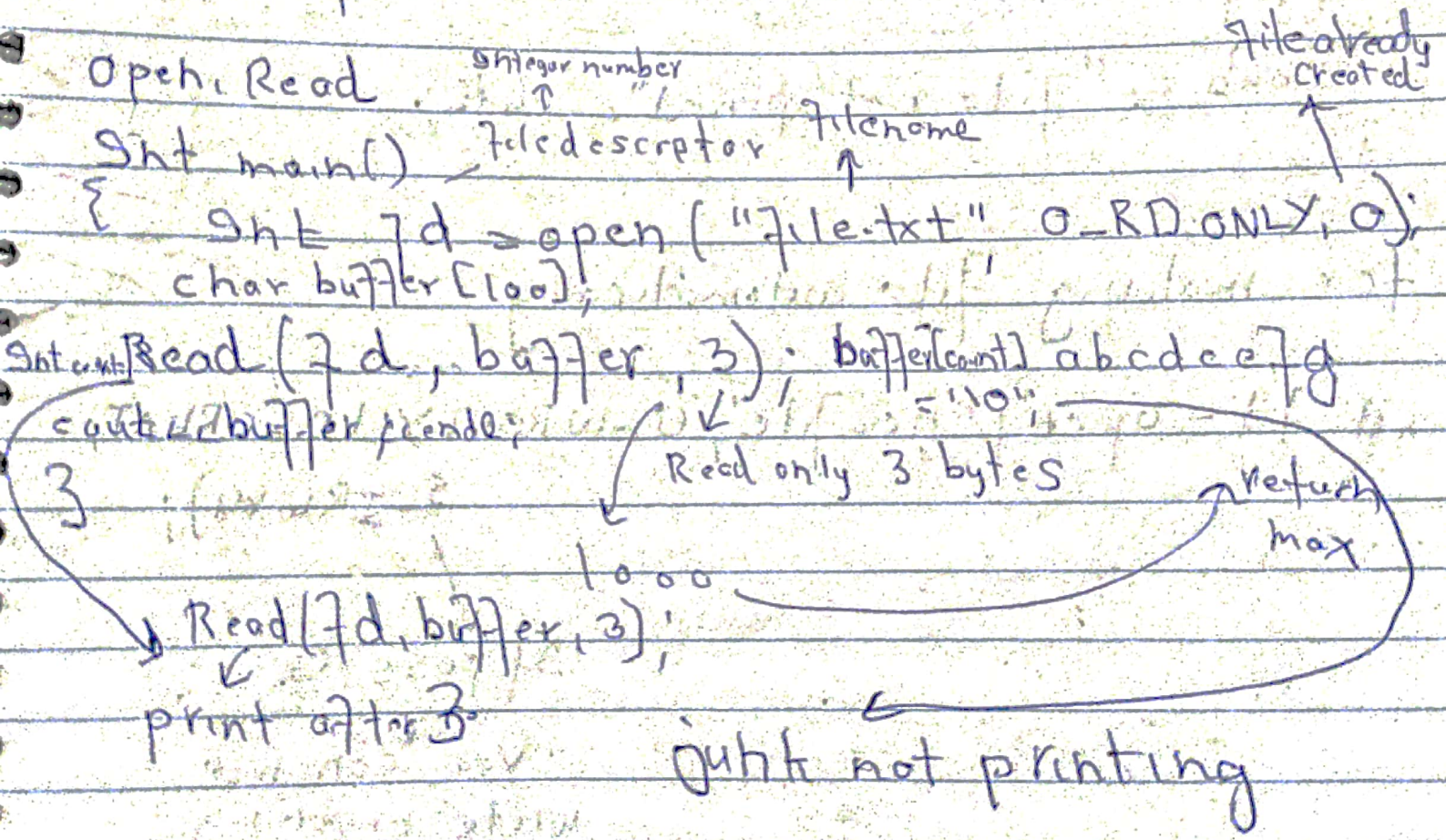
Output

execlp called

The value of sum is 30

→ If we used WAIT(NULL) in the parent process, then output will be

The value of sum is 30
execlp called.



SPA

NC Quiz III Shella

→ write(fd, "Hello WORLD", 11); ✓

→ Read(fd, "Helloworld", 11); ✗

→ int main()
{
int fd = open("new_file.txt", O_RDONLY, O ← not created file automatically

if(fd == -1)
{
cout << "file not opened" << endl;
}

→ for making file automatically

int fd = open("newfile", O_WRONLY | O_CREAT
S_IRWXU);

if poss ✓

You can not write anything

ACBDC

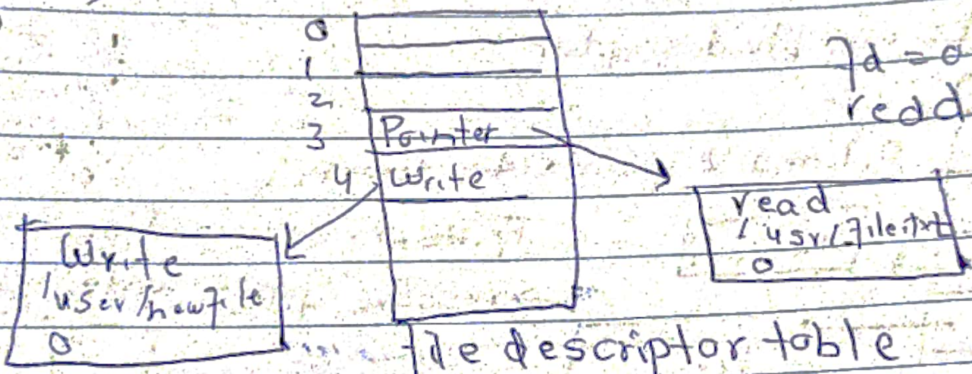
→ For terminal running

getid newfile.txt

→ Every process has its own file descriptor Table

→

fd = open(file.txt)
redd (fd, buffer, 10)



file descriptor table

→

fseek(fd, 10, 0);

↓ offset

↓ initial position

SEEK_END
SEEK_SET
SEEK_CUR

ftell(fd) → print position of pointer

PIPES

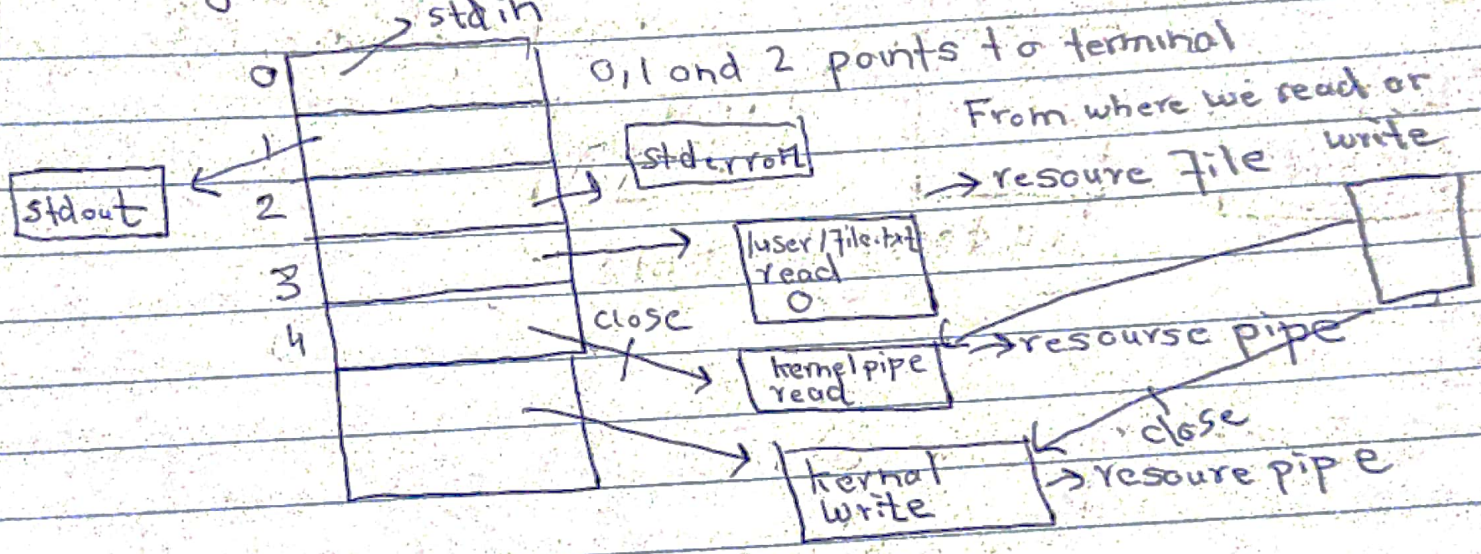
→ named pipe (2 or more independent processes)

→ unnamed pipe (parent and child)

Parent P
child C

read
write

→ Every process has its own file descriptor



→ if pipe create successfully and memory of pipe then return 0 otherwise return -1

→ int pipe
→ pipe(fd);
→ index = 0 → read end
→ index = 1 → write end

Code

```
int main()
{
    int fd[2];
    int status = pipe(fd);
    if(status == -1)
    {
        cout << "pipe not opened!" << endl;
    } return; -1;
int status = fork();
```

If Next code on Next page

```
9] if (status == 0)
{
    close(fd[1]);
    char buffer[100];
int count = read(fd[0], buffer, 11);
}   buffer[count] = '\0';
    close(fd[0]);
else if (status > 0)
{   close(fd[0]);
    write(fd[1], "Helloworld", 11);
    close(fd[1]);
    WAIT(NULL);
}   e
else
{
    cout << "child process not created!" << endl;
    return 1;
}
```

* this we want child do reading and parent writing do

read is an blocking system call

→ Pipe is an unidirectional process.
→ terminal has also buffer.

→ write(fd, const void * out, size_t)

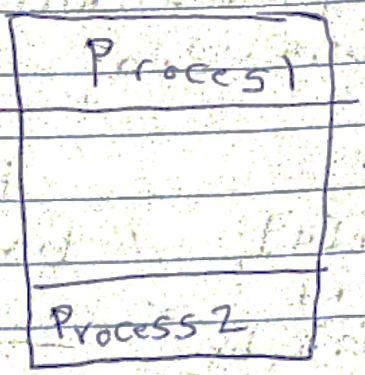
not char pointer

→ Read System call return count

Shared Memory

Process 1
int a=0; 0XFF00A
a=10;
Process 2;
~~int a=10;~~
~~cout<<a;~~
int b 0xFF00A
cout<<b;

* point in the same place



process1, process2

int a=0;
Process 1
a=10;
Process 2;
cout<<"process1 send"<<a;

we want to execute process2 first

int a=0;
flag=false; // lock

Process 1:
a=10
flag=true

Process 2:
while(flag==false) // CPU WAIT (BUSY WAIT)
cout << "process1 sent:" << a;

every process has its own register value.

PCB
Program Counter block

a=0
Process 1
a++; mov bx,[a]; // bx=0
// Race condition add bx,1
mov e [a],bx

Process 2
a++;
// Race condition → Different processes want to write at same location

~~add bx,1~~ move bx,[a] bx=0
~~move [a],bx~~ add bx,1 bx=1
~~move [a],bx~~ mov [a],bx a=1

Synchronization issue

→ This is not atomic instructions.

Critical section → set of instructions where we access shared memory.

Real Life example

int a=10000

transaction 1: deposit 2000 remove from ready queue

mov bx,[a] // bx,1000
add bx,2000 // bx=12000
mov [a],bx // ax=12000

content switch

transaction 2: withdraw 5000

mov bx,[a] // bx,1000
Sub bx,5000 // bx=5000
mov [a],bx // a=5000

→ Copy-on-write

int a=10;
fork();

→ If we change variable after calling fork(); new memory assign to a.

shmid= Always unique identifier

→ int shmget(key_t key, int size, int flags)
↳ used for create shared memory

flags
↳ 0666 → All process have read and write option
↳ IPC_CREAT create shared memory region if not ex
↳ IPC_EXCL
↳ 0 → already existed shared region

→ shared memory has two ideas

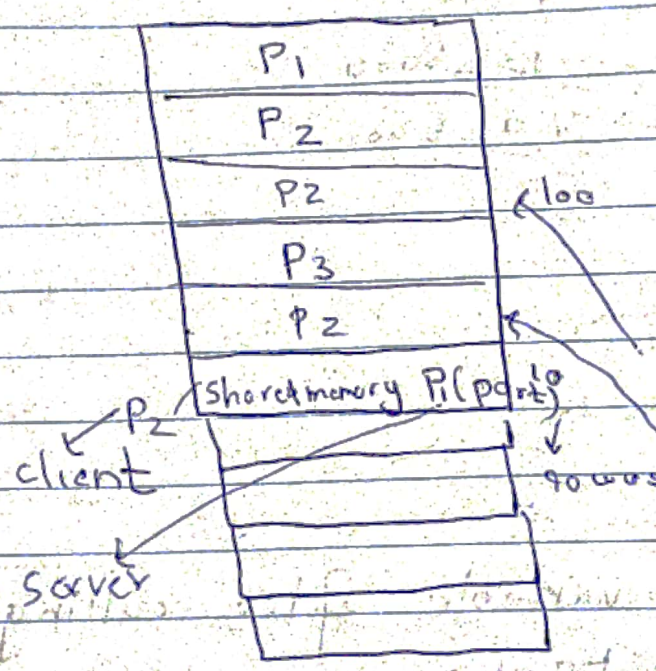
↙ one assign user ↘ other assign kernal

→ Private shared memory assigned only between child and parent

→ 0-id assign to Private shared memory

→ Non-zero assign to Public shared memory

→



→ void *shmat(int shmid, const void *shmaddr, int shmflg);

CHAPTER 5
THREADS

→ 1,000,000 entries

```
bool exists(int arr[], int size, int search)
{
}
```

→ Parent (First half) search
child (2nd half) search
→ 1 CPU more time → more content switching
→ 2 CPU less time → do multi processing
→ modern os do thread scheduling

→ memory shared is easy
→ Parent Thread

FAST

2019 [35] example1

```
void * myFunc(void* param)
{   int a = *(int*)param;
    int * new_ptr = new int;
    *new_ptr = a
    (*new_ptr)++;
    pthread_exit((void*)new_ptr);
}
int main()
{
    pthread_t id;
    int var=100;
    if(pthread_create(&id,NULL,&myFunc,&var))
    {
    cout << "creation failed" <<endl;
    }
    int* ptr;
    pthread_join(id,(void**)&ptr);
    cout << *ptr;
}
```

environment

pointer points to pointer

How to run

example1.cpp -lpthread

Multithreading advantages.

Adv
Accept client request

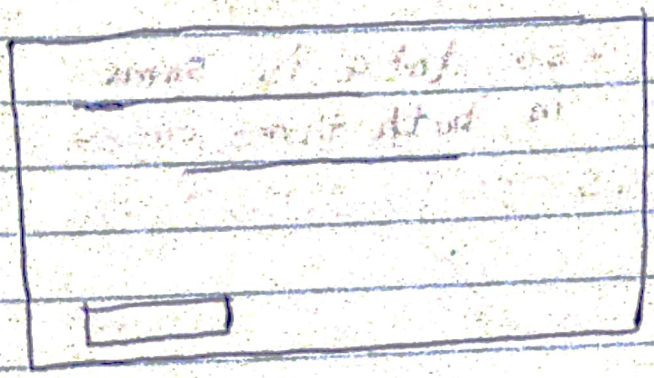
→ Server code

```
while(true)
{
   client=waitForClientRequest(); //blocking call
Thread(serviceRequest(client)); //10seconds.
}
```

→ processing speed fast

→ server accept more than 1 clients accept

→



Gui Adv
Responsiveness

```
GUI Thread
{
waitForEvents();
```

Thread (onSubmitButtonClick()); // 5seconds

Threadpool(100)

→ of threadpool full it means server down.

• Identifying task

→ Not every task become multithreaded → Bubble sort

→ subarrays (2 threads) / Merge (1 thread)

→ • Balance.

Divide data equally to every thread

→ • Data dependency :-

Subarrays [2 threads] Data dependency

↓ Depends between two threads

Merge [1 thread] ↘

Pthread_join for this

TYPES OF PARALISM

→ Data Parallism

Sum * so data is same

count in both three cases

Avg

GPU best to do this

→ Task Parallism

→ 1000, 000

Thread 1 → harmful so we will cancel the remaining threads

(For example if we have Thred 2 so we will cancel it by using pthread_cancel.

1. Asynchronous cancellation
   thread immediately terminates the target thread.
2. Deferred cancellation

SEMOPHORES AND MUTEX (SYNCHRONIZATION)

Flag = false
Transaction1 : (withdraw)
while(flag == false);
Mov ax, [account]
Sub ax, 500
Mov [account], ax

Transaction2 :
Mov ax, [account]
Add ax, 1000
Mov [account], ax
Flag = true

Account = 10 000

issues → Busy wait
issues → order issue.

Solution

Account = 10000
Flag = false

Transaction1 : (withdraw)
1 while(flag == true); // test
2 flag = true; // set
Mov ax, [account];
Sub ax, 5000
Mov [account], ax
Flag = false

Transaction2 :
1 while(flag == false); // test
2 flag = true; // set
Mov ax, [account]
Add ax, 1000
Mov [account], ax
Flag = false;

But there is issue in this model because when line 1 execute of both processes then race condition occurs.

Solution

Mutex

Mutual extrol (one transaction at a time in critical section).

Mutex m
Account = 1000
Transaction 1 : withdraw
Aquire(m) → wait
Mov ax, [account];
Sub ax, 5000
Mov [account], ax
Release(m)

Used queue for waiting (first in first out condition used)

Atomicity

Transaction 2
Aquire(m) → wait
Mov ax, [account]
Add ax, 10000
Mov [account], ax
Release(m)

2 3 4 5 6 7, 8 9

Mutex m
thread 1
aequire(m) → CPU1
critical section
release(m)
thread 2
aquire(m) 10ms
critical section → CPU 2
release(m) 10ms

↓

Mutex m
aquire(m) → fails
accessprinter
release(m)

↓

access one thread at a time but printer have ablility to handle ten threads at a time

Solution ← * Server example.

Semaphore=10
~~acqu~~

wait(m) // decrement For 11th thread → waiting queue
access Printer
Post(m) // increment
signal

```
Post(m)
{
  counter++;
  m.dequeue();
}
```

```
WAIT (m)
{
  if(mcounter > 0)
    m.counter--;
  else m.enqueue(process)
```

another way ↓

```
WAIT(m)
  m.counter--;
if(m.counter < 0)
m.enqueue(process)
```

→ If semaphore.m=1 // binary semaphore (behave like mutex)
→ counting semaphore if semaphore>1
→ Book (The little book of semaphore).

Problems

rendevous problem:

thread 1:
statement a
statement b

thread 2:
statement c
statement d

↓ Solution2

mutex m1=true
mutex m2=true

thread 1:
statement a
release(m2);
aquire(m1);
statement b

thred 2:
statement c
release(m1);
aequire(m2);
statement d

Solution

flag1=true
flag2=true

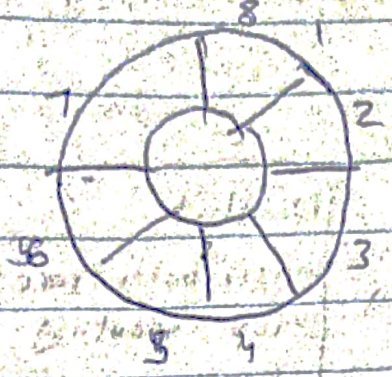
tread 1:
statement a;
~~while~~
flag2=false
while(flag1);
section b

thread 2:
statement c
flag1=false;
while(flag2);
section d

Deadlock
when both process stop executing progress

Start from Semaphore Barriers problem

Producer - consumer Problem:



* Producer have maximum ability to produce 8 elements

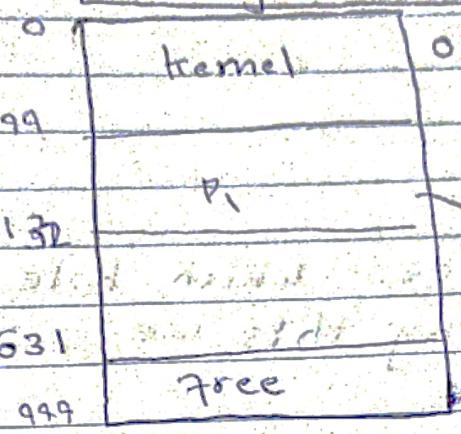
buffer[N]
int counter=0; Semaphore p=N;
Producer: Semaphore c=0;
in=0;
while(true)
{ wait(p); while(counter == N); X ↓ Busy wait finished
buffer[in]=produceItem();
in=(in+1)%N;
counter++;
} post(c);

From 15:17

Consumer:
out=0;
while(true)
{
while(counter == 0); X wait(c);
a=buffer[out];
out=(out+1)%N;
counter--;
} signal(p);

CHAPTER 9
MEMORY MANAGEMENT

Continuous Memory location



OS kernal size = 100 bytes

P1 = 32 bytes

P2 = 500 bytes

After execution P1 portion also free

368

P3 = 400

Lab 5 Practice

Parent → child

Lab 5 ✓
Lab 0 ✓
Lab 1 ✓
Lab 2 ✓
Lab 3 ✓
Lab 7
Lab 6
Lab 8
Lab 5
Lab 9

Fixed Partitioning → All blocks have same size.

Variable partitioning

→ Fragmentation occurs when we have more space in junks then the what the coming process needed.

↓ solution

defragmentation (compaction → merge holes by moving processes)

→ If we have two spaces which hole we fill first? For doing this we have three method

→ First Fit:-
First Free whole

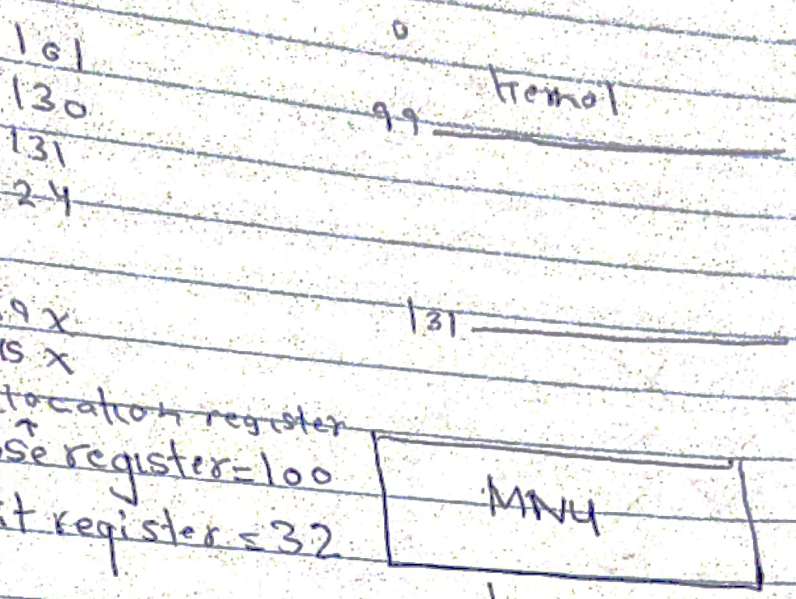
→ Best Fit:-
smallest possible whole

→ Worst Fit:-
largest possible whole

First fit and best fit ↓ utilization better

→ variable partioning → no internal fragmentation
↳ space left due to fixed partioning

→ Accurate address



if (addr < base register
trap
else (addr > base register + limit)
trap
else
allow memory space.

→ Address binding

a = b + c

```
mov ax, [101]
mov bx, [102]
add ax, bx
mov [100], ax
```

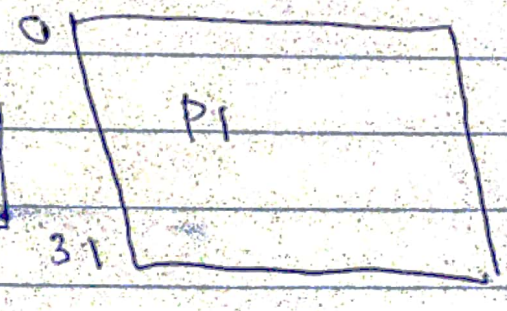
compile time binding is not possible in moder systems.

↓ solution

offset

```
mov ax, [1]   151
mov bx, [2]   152
iadd ax, bx
mov [0], ax
   150
```

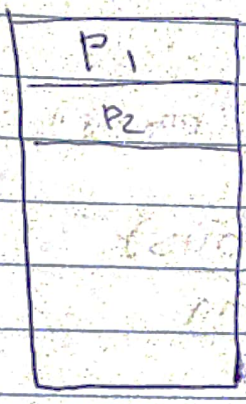
load time binding



Contiguous Memory Allocation

- fixed partioning
  - same size
  - different size portions
- variable partioning

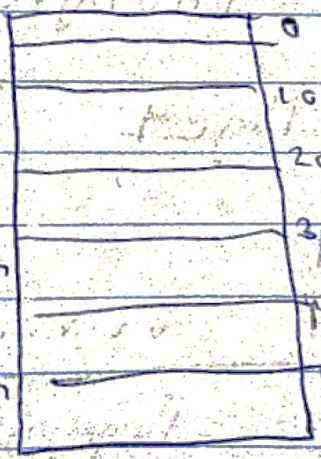
→ variable partioning



100 bytes
50 bytes

no internal fragmentation but external fragmentation occurs.

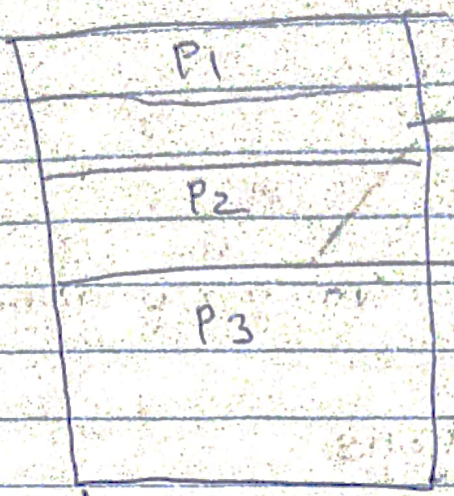
→ fixed partioning



70 bytes free

internal fragmentation and external fragmentation

P1 = 31



→ shared memory

P3 + P1 → shared memory

↓

not possible

→ MMU convert logical address to physical address

PAGING (NON-CONTEGIOUS)

Process → Paging

Page size = 50 bytes

RAM → frame size

P1 = (page 0)
P3 (page 0)
P1 (page 1)

P2 = 1 page

P3 = 1 page

1000 bytes

0

49

50

99

Page 0

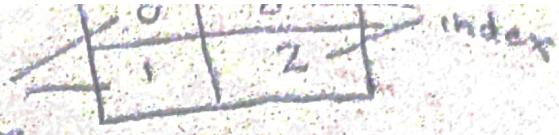
100 bytes

Page 1

v.v.r Imp

Frame size = Page size

Every process has its own page table



Page number

address register size = 32 bits

range = 0 - [(2^32) - 1) = 4 GB

Address register = 2 bits

~~2^2 = 4 (0-3)~~

0, 1, 2, 3

2^2 = 4 (0-3)

Vertual memory from Gate smashers.

The process whose size more than the main memory can be executed with the help of vertual memory.

→ logical address is called vertual address.

Contigious memory allocation

0

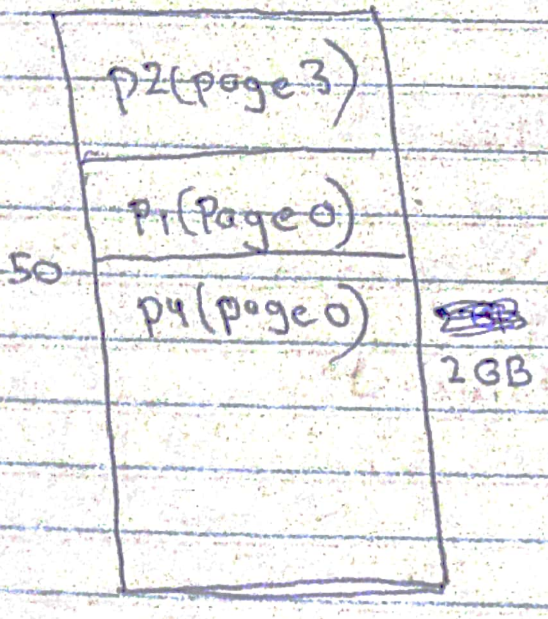
2^32 - 1

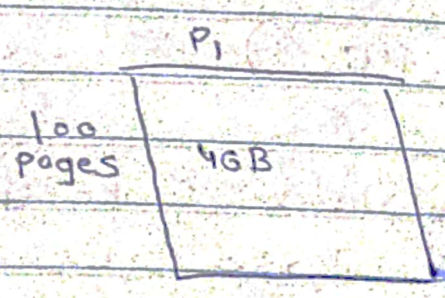
سیاری کی ساری میموری کو allocate کرنی ہے

→ In Contiguous its impossible to store 2GB Memory into 4GB

→ But we can do this by paging

Paging





Hardisk —execution slow→ Memory

locality of reference
temporal locality = memory same location memory accessed in near future
special locality

context switch

page0 = while loop
page1 =
a = true

آس پاس

→ If memory is full then we use page replacement policy.

→ Page load repeadly from Harddisk to memory → trashing.

→

IBRAHIM IVADIR

MEMORY MANAGEMENT

1) Protection
2) Add trans
   - log
   - Phy

→ Memory Allocation Method

- Contongcous
  - fixed
    - equal
    - ~~contiguous~~ unequal
  - variable
- Non-contagious.
  - Paging

C

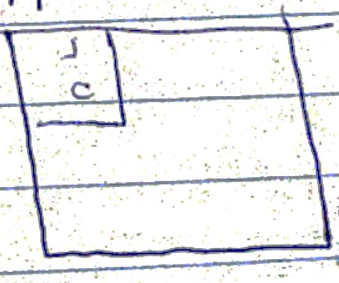
FILE SYSTEM

→ collection of information
→ normally placed in secondary storage
→ video form, audio form
→ filename.type.
→ identifier → so we can access file
→ Normally file is in Harddrive,
Non-volatile memory

→ Protection
→ images files, video files, audio files.
→ Difference between delete and truncate
↳ remove content on
→ we can divide file into layered approach
so that we can modify it easily.
→ Every file has its own FCB/inodes
↓
Data structure

→ FFS → FAST File System.

→ File system implementation.
Boot control block
→ Boostrap, BIOS



→ Page table is in the RAM

EAT = Effective address time =

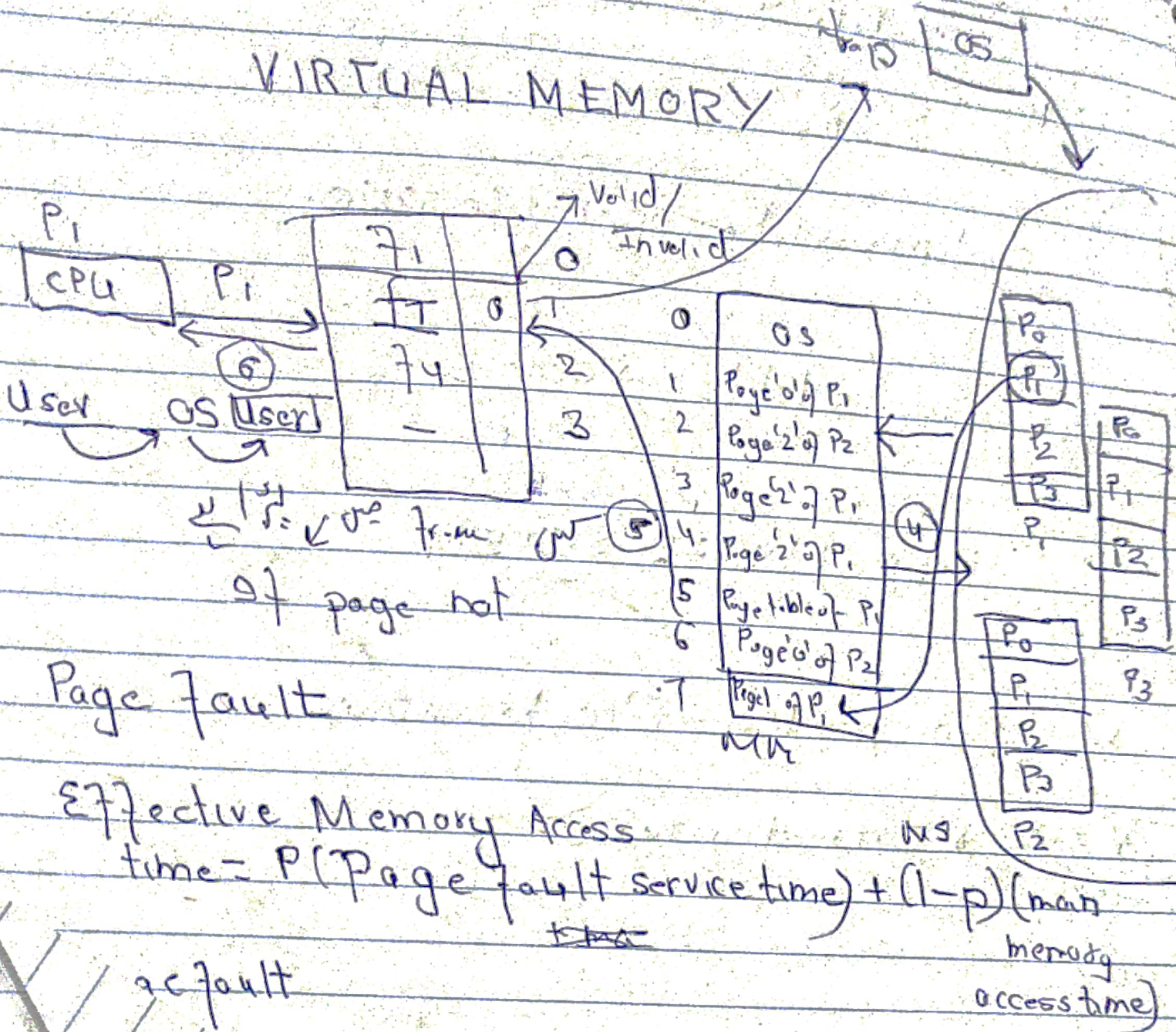
(tlb hit ratio x memory_access time) +
(tlb_miss_ratio x 2* memory_access_time)

= (0.80 x 100) + (0.20 x 200)

= 120 nanoseconds

PTBR
PTLR

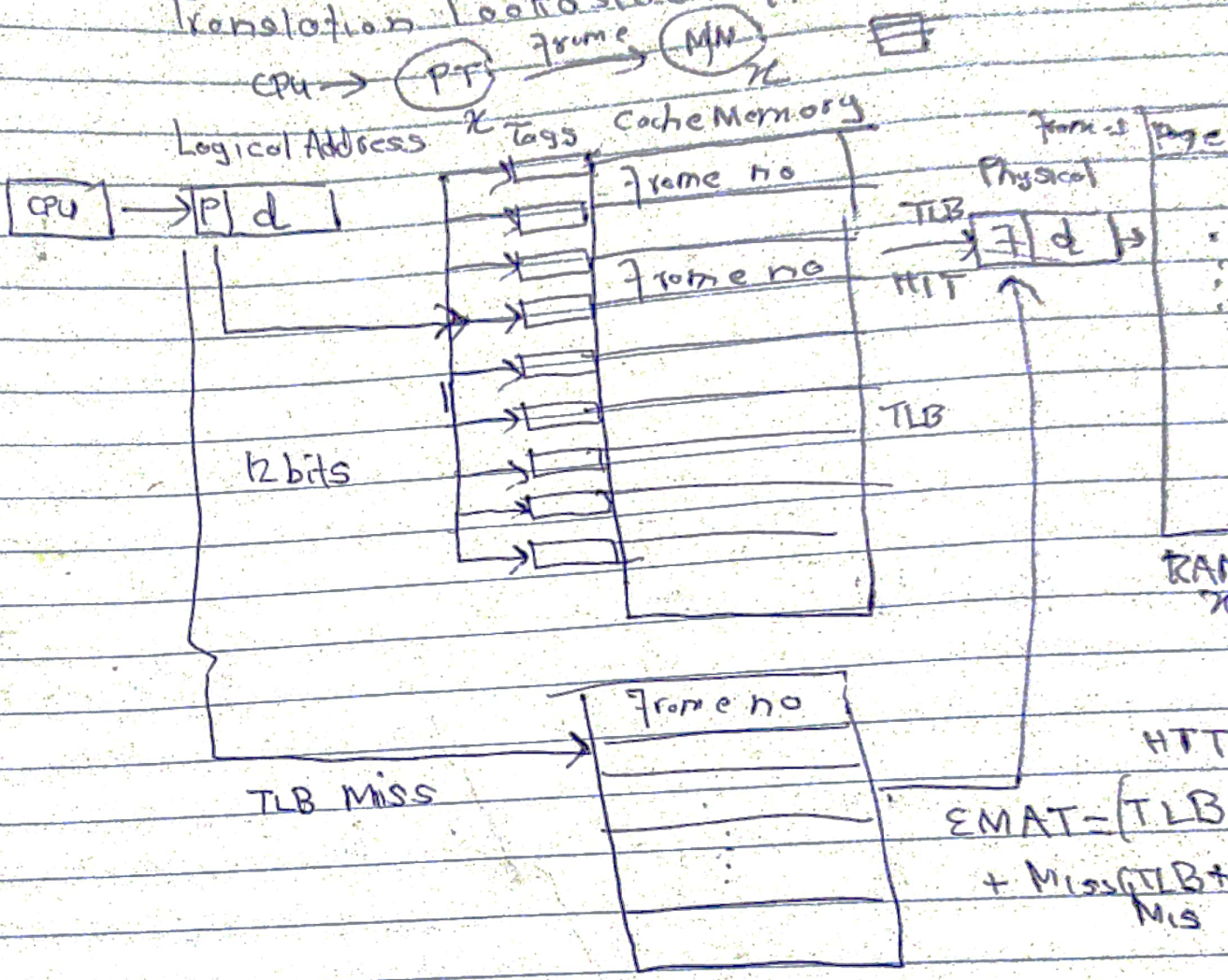
VIRTUAL MEMORY



Convert logical address into Physical address.

MMU

Translation Lookaside Buffer



EMAT = (TLB + Miss(TLB + Mis