Process Synchronization

⇒ **Independent process:** are those process that can neither affect other process or be affected by other process when executing in the system.

→ do not share data / resource

⇒ **Cooperating process:** are those process that can affect or are affected by other process when executing in the system, result → inconsistency

Note: This process may share resources, data, variable, code with each other.

counter = 5

↓

P_1 ()
read (counter)
count++;
write (counter);

Preemption

→ PCB (save)

P_2 ()
read (counter) → 5
counter++; → 6
write (counter);

$P_1 = 6$
$P_2 = 7$
} synchronized way,

$P_1 = 6$

$P_2 = 6$

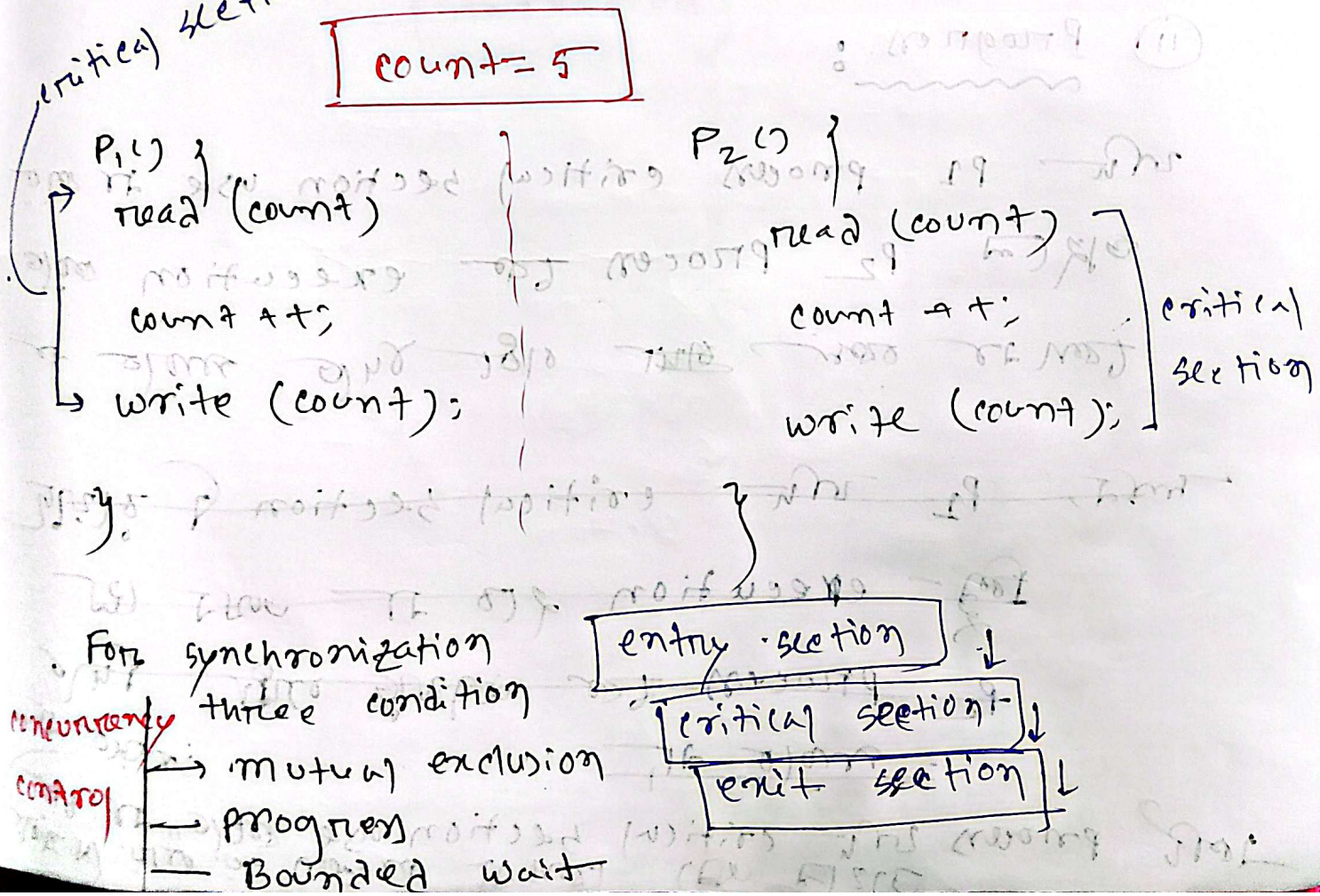
race condition.

here, second phase we can see, $P_1 = 6$, $P_2 = 6$ which is inconsistency data. which is occur race around condition.

Race Around condition:

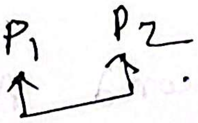
A race condition occur two or more process can access shared data and they try to change it at the same time.

Critical section Problem



- For synchronization three condition
  - mutual exclusion
  - progress
  - Bounded wait

(Concurrency control)



Entry section

critical section

Exit section

avoid race condition

(i) **Mutual exclusion:**

let, P1 and P2 ~~to~~ are two process.

যখন P1 critical section এ entry করে

তখন P2 entry section এ lock থাকবে

তাকে থাকবে যতক্ষণ না P1 execution হয়।

(ii) **Progress:**

যদি P1 process critical section use না করে

তাহলে P2 process কে execution করতে

দেওয়া হবে অথবা ~~আটকে~~ লাইনে ধরে রাখা যাবে না,

~~যদি, P1 যদি critical section এ ঢুকেছে~~

~~কিন্তু execution হচ্ছে না তখন সে~~

~~P2 process কে প্রবেশ করতে দিবে~~

~~= করছে না,~~ করছে না.

একটি process যদি critical section use করছে, না তখন অন্য
তাহলে একটা কোন না process এর জন্য অপেক্ষা করে থাকবে

(iii) **Bounded wait:** কোনো process যখন কোনো critical section এ ঢোকার জন্য সময় এর wait করে তখন সে অবশ্যই একটি নির্দিষ্ট সীমার (bounded) মধ্যে সুযোগ। অসীম হবে না।

⇒ Every process waiting to enter its critical section will get a chance within a limited no. of a turn prevent indefinite waiting (starvation).

Semphore

Semaphore is an integer variable that is used to solve the critical problem by using two atomic operations wait and signal that are used to process synchronization.

wait → down → value 1 কমে

Entry section

critical section

Exit section

signal → up → value ++ হয়ে

$S = 1$ / 1_0

```
wait (s)
{
  wait (s<=0);
  s = s-1;
}
```

```
signal (s)
{
  s = s+1;
}
```

0+1 = 1

P_1

```
do { wait (s)
```

P_1 P_2 (P_1, P_2, P_3)

wait → P_1 → critical section

```
signal (s)
} while (true)
```

counting semphore:

A semphore is a synchronization mechanism used to control access to shared resource by multiple processes or thread in a concurrent system.

→ It help prevent concurrent ~~system~~ condition and ensure proper coordination when several processes need to read/write or use the same resource at the same time.

wait() / P / down

- Decrement the value
- If the value become negative, the process blocks (waits)

signal() / V / up:

→ Increment the value

→ If there are waiting ~~process~~ processes one of them is woken up.

Type of semaphore:

① counting semaphore

→ Value can be any ~~none~~ non-negative integer (0, 1, 2 ...)

→ useful when multiple instances of a resources are available.

ex: A printer pool with 3 printers can allow 3 simultaneous users.

② Binary semaphore (also called mutex)

→ value is only 0 to 1

→ Used to enforce mutual exclusion allowing only one process into a critical section at a time.

Practical use:

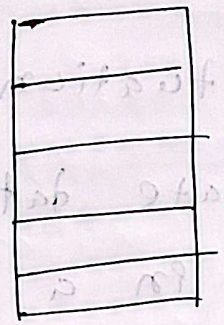
→ Managing access to shared memory, files, Printer or database

→ classic problem: producer-consumer, Reader-writer, Dining Philosophers.

Producer - Problem

P_1 (Producer)

P_2 (Consumer)



shared resource

buffer

→ critical section

P_1 যখন buffer এ কাজ করবে তখন P_2 কে অপেক্ষা করতে হবে যতক্ষণ P_1 কাজ শেষ করে।

→ same for P_2

Producer:

(i) check buffer overflow

(ii) mutual exclusion / synchronization (process)

(iii) update buffer

Consumer:

(i) check underflow

(ii) synchronization

(iii) update buffer

Producer - consumer problem :-

The producer - consumer problem (also called the bounded buffer problem) is a classic example of a synchronization problem in OS and concurrent programming.

It models a situation where:

→ producer create data (item) ~~over~~ and place them in a shared ~~resource~~ buffer

→ consumer take data from the buffer and process it.

→ the challenge is to make sure producer dont add data when the buffer is full, and consumer don't remove data when the buffer is empty — without causing race condition

Key

① mutual exclusion:

Only one process (producer & consumer) can access the buffer at a time.

② Synchronization:

→ producer wait if the buffer is full

→ consumer " " " null/empty

To solve this problem we use semaphore:

→ Three semaphore are used.

① Mutex: Initial value: 1

purpose: Ensure mutual exclusion for buffer access.

② Empty: value: N (buffer size)

→ count how many slot are available.

③ Full: → value: 0

→ how many items are available to consume

Algorithm:

semaphore → full
↓ empty
→ mutex

producer:

```
do{
wait(E);
wait(S);
add();
signal(S);
signal(F);
}while(true);
```

wait() { while(E<=0); E=E-1; }

```
do{
wait(empty);//check for empty slot
wait(mutex);//lock buffer
//added item to buffer
signal(empty)   buffer[in] = item;
                in = (in+1)%N;
signal(mutex);//unlock buffer
signal(full);//increase count of
              filled slots.
}while(true);
```

consumer:

```
do{
wait(F);
wait(S);
consume();
signal(S);
signal(E);
}while(true);
```

item=buffer[out]; out=(out+1)%N;

wait(S) { while(S<=0); S=S-1; }

signal(S) { S=S+1; }

signal(E) { E=E+1; }

```
do{
wait(full);//check is buffer has an item.
wait(mutex);//lock buffer
//remove item from buffer
signal(mutex);//unlock buffer
signal(empty);//increase count of
//consume the item.          empty slots.
}while(true);
```

Explain:

→ producer waits if empty == 0 (buffer is full)

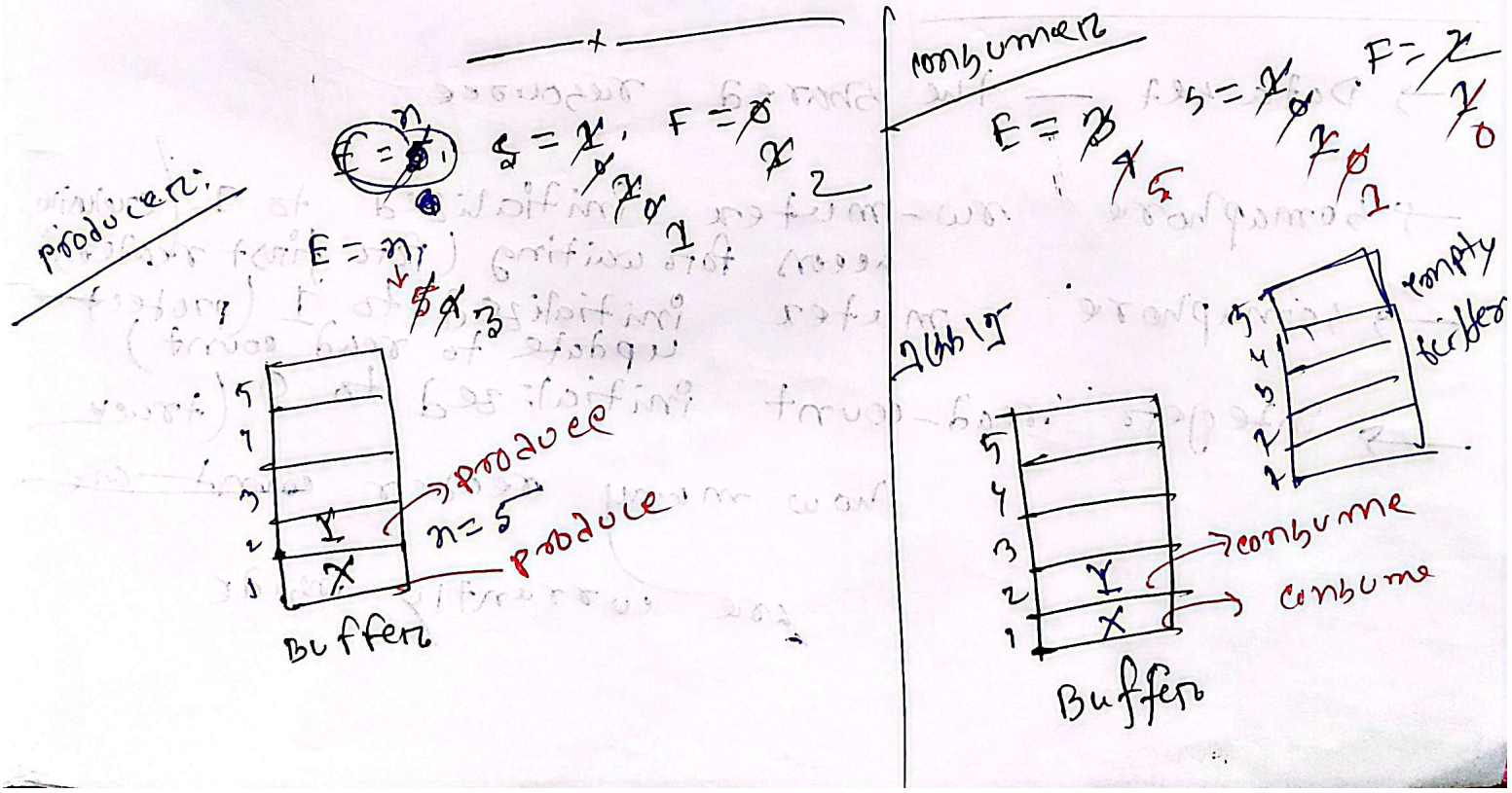
→ consumer waits if full == 0 (buffer is empty)

→ mutex ensures only one process changes the buffer at a time.

purpose:

→ proper use of semphores for synchronization

→ prevention of race condition, dead locks, and busy waiting in multi-threaded system.



Reader writter problem using semaphore

A shared dataset accessed by many concurrent processes.

Reader – Only read the data set, they do not perform any updates

writer – can both read and update data.

Problem

→ allow multiple readers to read at the same time (simultaneously)

→ ~~only~~ Ensure that only one writer accesses the shared data at a time.

Shared Variable

→ Data set — the shared resource.

→ semaphore rw-mutex initialized to 1 (exclusive access for writing (for first reader))

→ semaphore mutex initialized to 1 (protect update to read count)

→ integer read-count initialized to 0 (track how many reader ~~count~~ are are currently inside.

writer

```
do {
    wait(wrt);
    write operation;
    signal(wrt);
} while (true);
```

```
do {
    wait(rw_mutex); //lock for exclusive access
    //writing is performed
    signal(rw_mutex); //release lock
} while (true);
```

→ A writer must acquire rw_mutex before writing

→ while writing, no reader or other writer can enter.

rw_mutex = 1;

Reader:

```
wait (mutex)
read_count ++
if (read_count == 1)
    wait (wrt);
signal (mutex);
```

Entry section

read operation ← critical section

```
wait (mutex);
read_count --;
if (read_count == 0)
    signal (wrt)
signal (mutex)
```

Exit section

e.g.

wrt = 1, mutex = 1, read_count = 0

→ The first reader locks wrt to keep writer out

→ multiple readers can read together because only read-count is protected by mutex.

→ The last reader wrt. allowing writer to proceed

Variation & starvation:

① Reader-priority:

→ reader are never delayed unless a writer is already writing.

Risk: writer starvation (writer must wait indefinately)

② writer-priority:

→ once a writer is ready, it's write as soon as possible

→ Risk: reader starvation (reader must wait indefinately)

producer - consumer problem / bounded buffer

producer: A producer is the process that creates data/items & places them into the shared buffer.

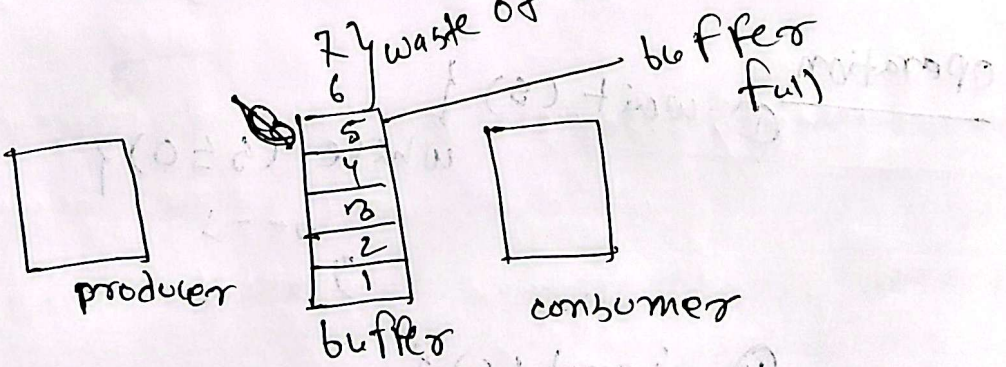
Ex: web server → web pages

consumer: A consumer is the process that takes the data/items from the shared buffer & uses or processes them.

Ex: web page → clients.

→ It is a classic synchronization problem in OS & concurrent programming.

→ Also called bounded buffer.



Goal

→ A producer must wait when the buffer is full.

→ A consumer must wait when the buffer is empty

No two share threads corrupt shared data
(No race condition)

→ The access share buffer should be mutually exclusive

Semaphore soln to the problem

three semphore (variable) used

Full = 0 → no. of filled slots } counting
Empty = N → no " empty " } semaphore
mutex = 1 → binary semaphore for mutual exclusion

two operation

① wait(s) {
while (s ≤ 0);
s--;
}

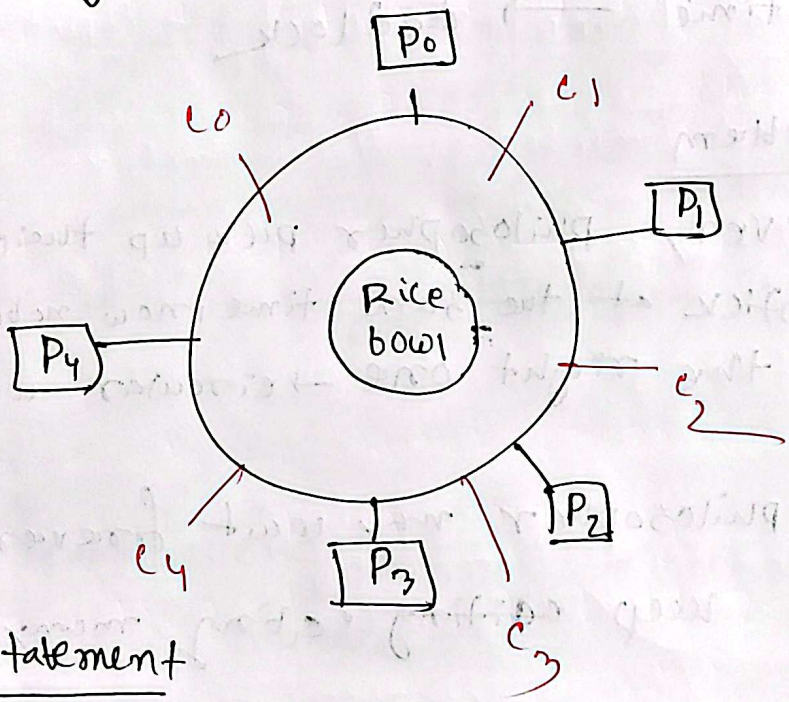
② signal(s) {
s++;
}

producer code:

~~Dinning~~

Dining philosopher problem

The dining philosopher problem is a classic synchronization problem introduced by Dijkstra to illustrate deadlock, starvation, concurrency, proper resource sharing.



Problem statement

There are 5 philosopher sitting around a table.

Between each pair of philosophers, there is one chopsticks

--Each philosopher repeat forever:

- Think
- pick up left chopstick
- " a right "
- Eat
- put down both chopsticks
- Think again.

Problem is: Each philosopher need two chopstick to eat. If all philosopher pick one chopstick at the same time ⟶ deadlock

Issue in the problem

DeadLock: if every philosopher pick up their left chopstick at the same time, now nobody can pick the right one → circular wait

starvation: One philosopher may wait forever while others keep ~~editing~~ eating many times

Concurrency Issue: No adjacent philosopher cannot eat at the same time, because they share chopsticks

We use semaphore to manage chopstick & avoid deadlock.

semaphore: ① binary semaphore

chop[5] = {1,1,1,1,1}

② counting semaphore

~~semaphore~~ count = 4 ; allow only 4 philos.. to pick chop at once

semaphore mutex

Algorithm

→ Each chopstick represented as a binary semaphore (mutex)

→ Phil.. must acquire both left & ~~wait~~ right semaphores before eating.

→ After eating philosopher releases both semaphore

code:

semaphore chop[5] = {1,1,1,1,1}

" mutex = 4

do {

~~wait (mutex)~~ wait (chop[i]);

wait (chop[(i+1)%5]);

//eat

code

```
semaphore chop[5] = {1,1,1,1,1}; //each are free
u          mutex = 4 //allow at most N-1 phils try
do { thinking();
     wait(mutex) //ensure at most N-1 are trying
     wait(chop[i]); //pick left chop
     wait(chop[(i+1)%5]); //pick right chop
          // eat ---- //critical section
     signal(chop[i]); //put down left
     signal(chop[(i+1)%5]); //put down right
     signal(mutex); //allow another to try
          // think ----
} while(true);
```

Explanation: Dead lock handling

→ allow at most 4 philosopher to be sitting simultaneously at the table.

→ allow a philo -- to pick up the chop only if both are available (in critical section)

→ use an asymmetric soln — an odd-num philo:- pick up 1st the left chop & then the

right chop. Even no- philo-- pick first the right chop & then the left.

sleeping barber problem

The sleeping berber problem is a classic process synchro-- problem introduced by dijkstra.

It models a barber shop with:

→ 1 barber.
→ 1 " chair
→ N waiting chair for customers.

Berber behaviour:

→ If no customer is present → barber sleep

→ when a customer arrives: three condition

① → If barber is sleeping → wake him

② → " " is busy & seats are available
→ customer waits

③ → If no seats available → customer leaves.

Goal: synchronized barber and customer using semaphores.

Sol^{n} → challenge

→ Only one customer is in the barber's chair at a time.

→ customer either wait or leave when seats are full.

→ The barber works when customer are available & sleeps otherwise.

Sol^{n}

Semaphores: use 4 semaphore

① barberReady → signals that barber is ready

② customerReady → " a customer is waiting

③ mutex → protect share variable

④ waiting → count of customer waiting

code

semaphore barberReady = 0

" customerReady = 0

" mutex = 1

int waiting = 0

int N = number of chairs.

```
customer ( ) {
        wait (mutex);
          if (waiting < N) // free waiting chair
          {
            waiting++; //increment the no of
                         waiting customers
          signal (customerReady); //customer arrives
          signal (mutex);
          wait (barberReady); //wait until the
                               barber call you
        }
      else {
          signal (mutex);
      }
    }

barber ( ) {
    while (true) {
      wait (customerReady) > //sleep if no customer
        wait (mutex); //critical section
        waiting --; //one waiting customer move to
                      barber chair.
      signal (barberReady); //call's customer to haircut
      signal (mutex); //exit critical section
    }
}
```

we set,

→ barber sleep when no customer ~~coming~~. came.

→ customer wake barber when arrives.

→ customer only enter if waiting seats < N

→ mutual exclusion

  ← mutex prevents race condition on waiting variable.

→ Barber & customer always release each other with semaphore.

  — No deadlock

ex: Barbar shop has

  → 1 barber

  → 3 waiting chairs

situation

(1) customer A arrives → wakes barber

(2) " B " → sit and waits

(3) " C " → sits

(4) " D " → No seat limit → leaves.

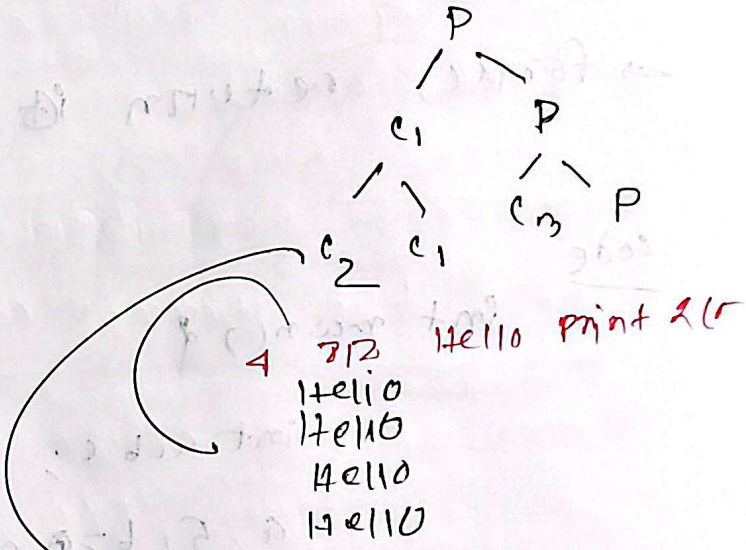
Barber :

→ cuts hair of A
→ signal B
→ cut hair of B
→ signals C
→ After finishing → sleeps until next customer.

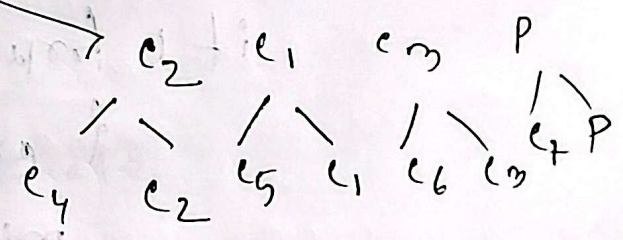
time sharing / multitasking

LINUX

```
int main() {
    fork();
    fork();
    print("Hello");
}
```




```
fork();
printf("ABC");
```



output: 2^n
n = no. of fork() calls.
child = n-1

output:

8 O/P.
ABC
ABC
ABC
:
ABC

Hello + ABC

Hello
Hello
} 8 time.

fork(): is a system call in unix/linux used to create a new process.

The process that calls fork() is the parent process, & the newly created process is the child process.

When fork() called:

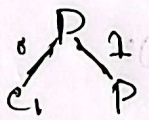
The entire parent process is duplicated.

A new child process is created.

⟹ fork() return int value.

code

```
int main() {
    int a, b, c;
    a = 5, b = 0;
    if (fork() == 0)
        { a = a - b;
          printf("%d", c);
        }
    else
        { c = a + b;
          print (c);
        }
}
```



Output:

result = 0
result = 2

Ex:

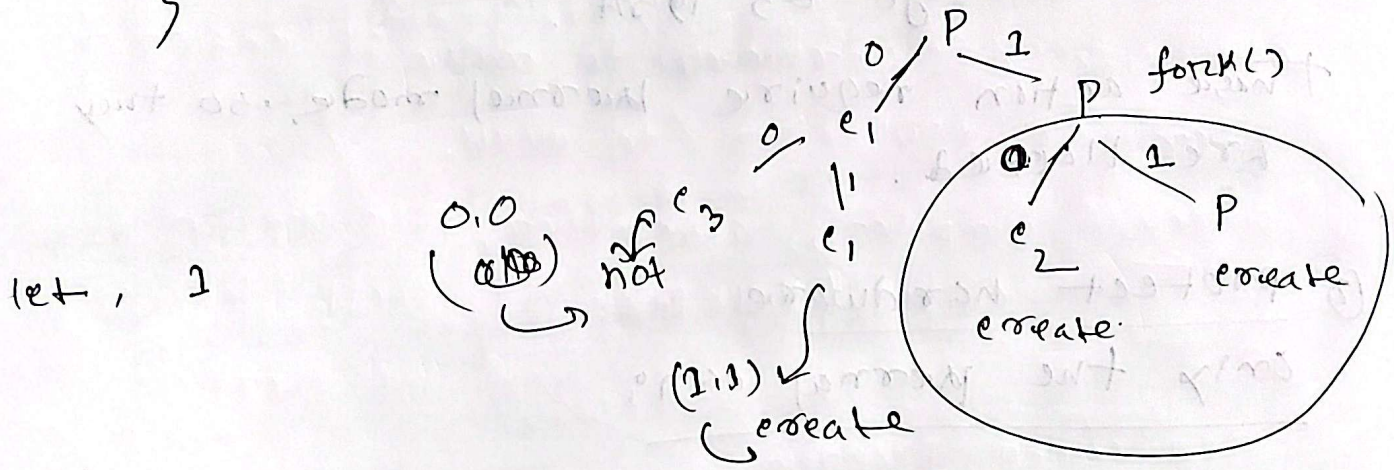
```
int main() {                    1,0        1,0
    if ( fork() || fork() ) {
        print ("create");
    }
    else {
        print ("NOT");
    }
}
```

OR

0	0	0
0	1	1
1	0	1
1	1	1

OR - Operation

B



let, 1

Output:

create.

create

Not

21-22

1 (c)

How does provide security

(1) prevents from damaging the system

used-mode program cannot:

→ Shut down the system
→ Format the disk
→ access another program's memory
→ change OS DATA.

These action require kernel mode, so they are blocked.

(2) protect hardware

only the kernel can:

→ change page table
→ Access I/O devices
→ manage ~~Interrupts~~ Interrupts.

if user programs could do this, the system would crash immediately.

(3) prevent unauthorized access :—

user-mode request services using system calls.

like — read(), write(), open(), fork().

when a system call is made:

switch user mode → kernal mode,
→ OS check permission,

→ executes the request safely
→ return to user-mode

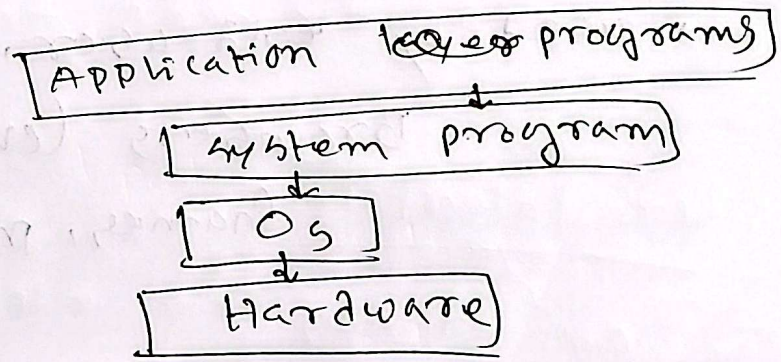
this ensure — no program can bypass security
no direct hardwire access.

④ Handle error safely

If a user program tries something illegal.
→ CPU raises a trap (exception)
→ OS kills the program
→ system remain safe.

ex: Trying to access restricted memory
show → segmentation Fault!

2(a) A computer system has main 4 layer



① Hardware (bottom layer)

CPU, RAM, Hard disk/SSD, keyboard, mouse, monitor
network devices.

Hardware can understand machine-level instruction

2(b)

New

→ The process is being created

→ Allocates PCB & other resources.

Trigger: process moves to Ready state when it is fully loaded into memory.

Ready

→ The process is ready to run but waiting for cpu allocation.

→ Place in the ready queue

trigger: short term scheduler select the process & dispatcher assign cpu control

Running

→ The process executing on the cpu

trigger:

To waiting/blocked: If process request I/O or waits for a resource.

To ready: If time slice expires in time-sharing system

To Terminated: If process complete execution or is explicitly killed

Waiting (the process cannot continue until an external event occurs)

Trigger:

To ready: Event occurs (I/O completed, resource available)

Terminated: finished/killed

Trigger: Reclaims all resources, remove all process from all queues.