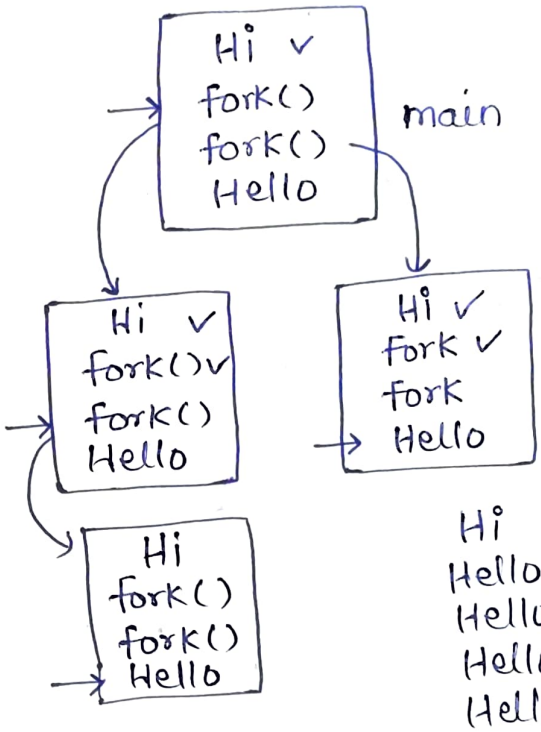
# Operating System

Date: 20/04/23

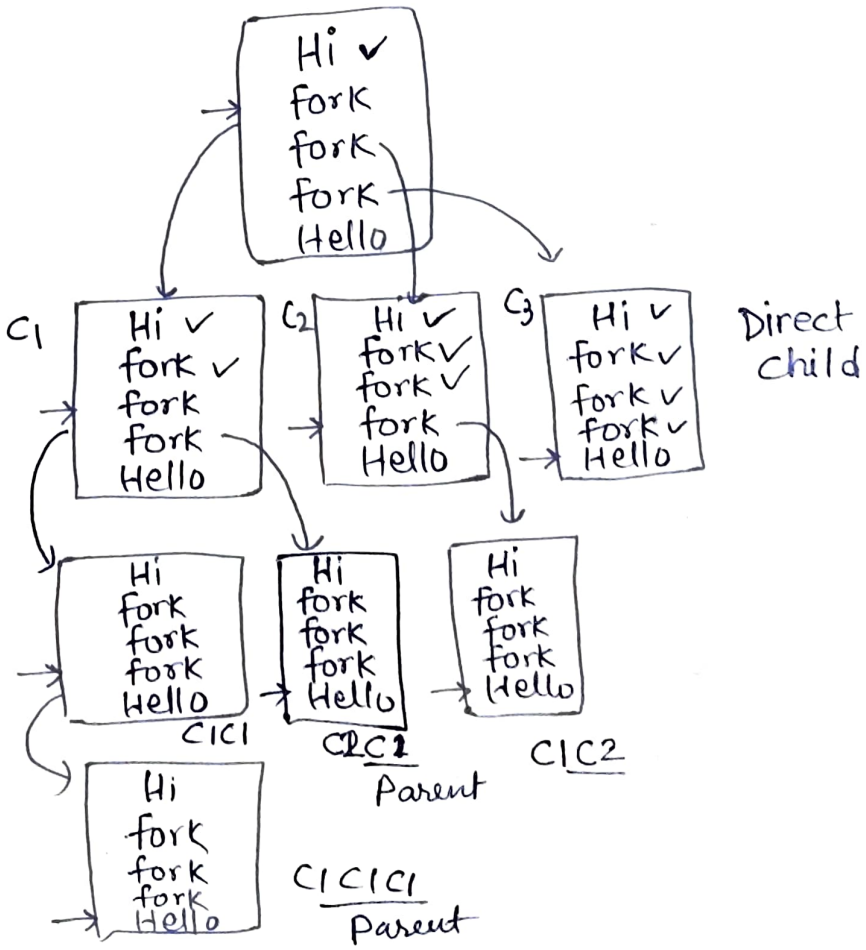
fork() → create a new process

```
# include <unistd.h>
  void main () {
       print ("Hi");
       fork ();
       fork ();
       print ("Hello");
  }
```




```
# include <unistd.h>
  Void main () {
      print ("Hi");
      fork ();
      fork ();
      fork ();
      print ("Hello");
  }
```

Hi
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello



| Fork | child (c) | Total Process |
|---|---|---|
| 3 | $2^3-1$ | $2^3$ |

## Generalised

| child Process | Total Process |
|---|---|
| $2^{\|Fork\|}-1$ | $2^{\|Fork\|}$ |

## Date : 21/04/23

fork() → unistd
↳ returns integer value / id of a child process to main

int Pid → predefined some variable of type int

id = 0
id > 0

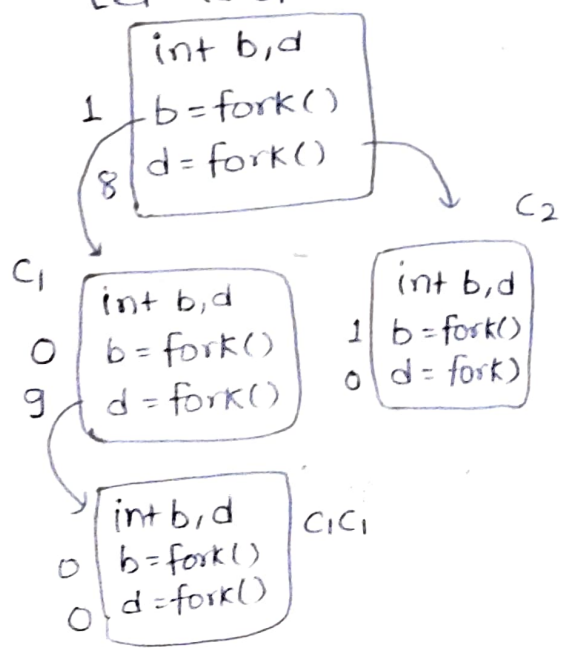
main → b = fork() → +ve
child → b = for() → 0

# Void main () {
int a = 10, b;
b = fork();

Main / child
id of child
0

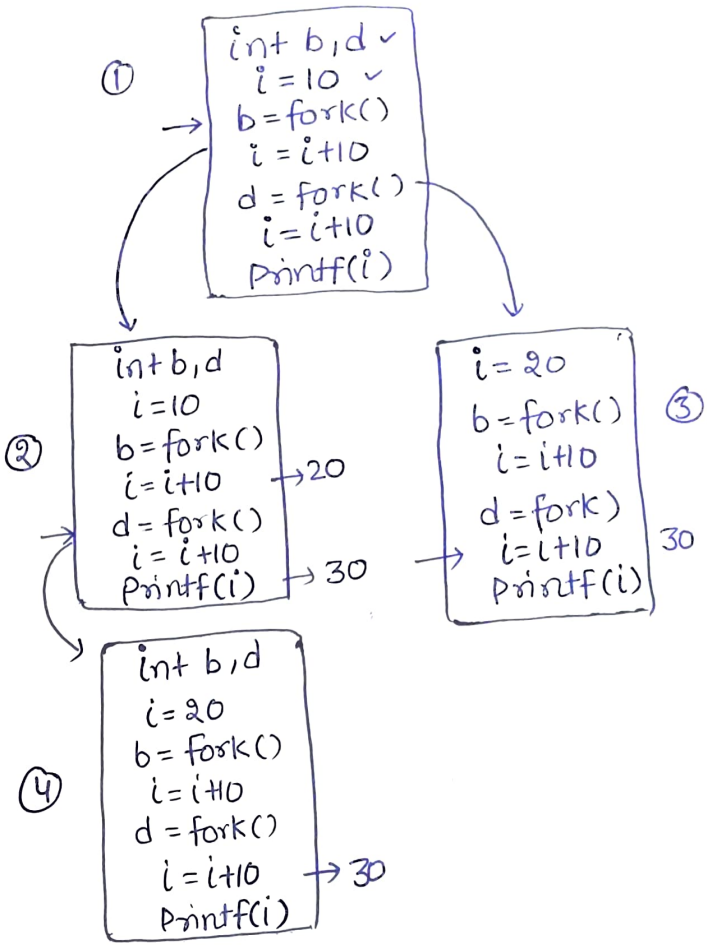
# Let id of main = 3

| id available |
|---|
| 1 |
| 8 |
| 9 |
| 10 |
| 15 |
| 20 |
| 30 |

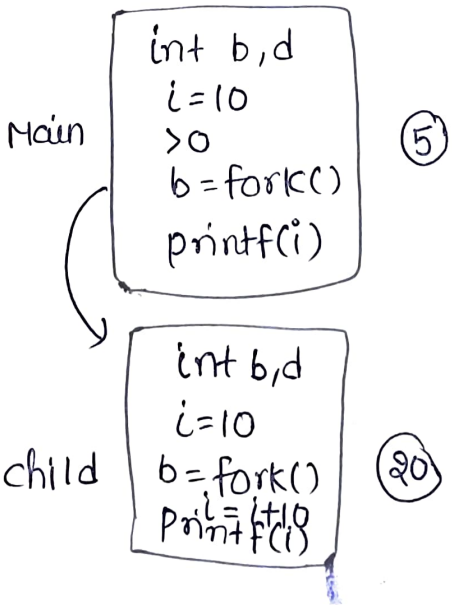


```
# int b,d ;
  int i = 10 ;
  b = fork() ;
  i = i+10 ;
  d = fork() ;
  i = i+10 ;
  Printf (i) ;
```

Output (i) ?




```
# int b,d ;
  int i =10 ;
  b = fork() ;
  if (b == 0)
     i = i+10
  else
     i = i-5
  printf (i) ;
}
```




X Hardware
↓
Usermode → print
Kernelmode → fork()
↓
interacting
with
H/w

```
P_id fork() ;
```

Signature / Complete Structure

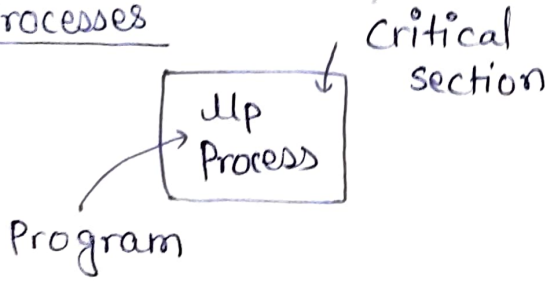
getpid() → return current process id

getppid() → get parent process id

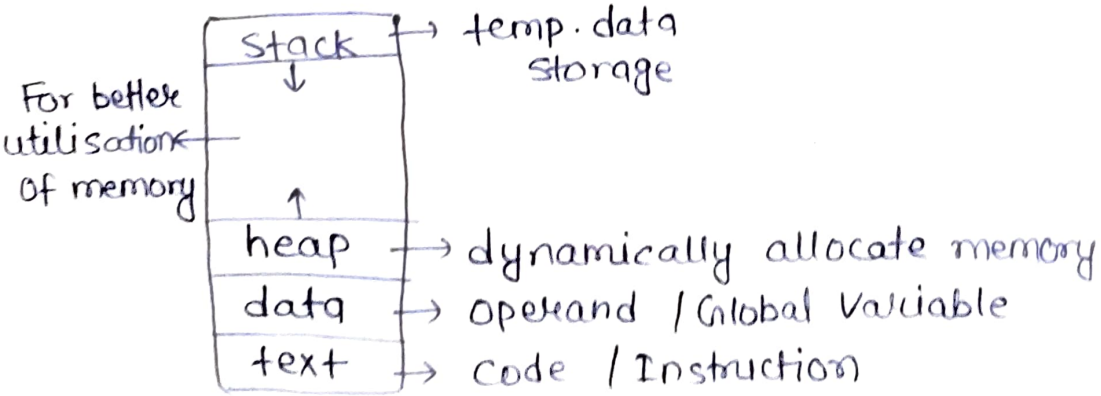
# 
```
int b, d;
int i = 10;
b = fork();
if (b == 0)
    getppid()
else
    getppid()
```

Date: 27/04/23

## Processes




When a program enters critical section i.e. called a process.




# 
malloc (size)
↓
Pointer is returned

Calloc (size, times)
↓ 4 5
Pointer is returned

| 4 | 4 | 4 | 4 | 4 |
|---|---|---|---|---|

***Malloc → no default initialization have some garbage value or ASCII value.

Calloc → have default initialization that is zero.

(int argc , char *argv[])

↓

no of passing arguments

↓

5

M.C

M 10 20 5 1

command line argument

argv[0] = 'M'

↳ name of file

argv[1] = '10'

argv[2] = '20'

argv[3] = '5'

argv[4] = '1'

CPU → running state

Date : 28/04/23

- Zombie Process (Parent is available child is terminated) However some entries of child process is available
- Orphan Process (Parents are terminated first child is still alive)
- Wait() → Parent is waiting for child process to terminate
- Sleep() → Parent is in sleep state while child complete work.

→ A process enters in waiting state by

i) I/O request
ii) Time slice
iii) Sleep() / wait()
iv) Interrupt

```
main()
{  int a;
   a = fork();
   if(a==0)
   {  ---
      ---
   }
   else
      wait()/sleep();
}
```

# <wait.h>

Store in 2 byte

Pid_t wait (*int P)
(*int status)

↳ id is returned of terminated child

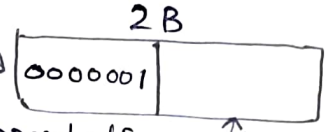
# How many ways to get id of a child ?

wait()
fork()

# <stdlib.h>

exit (int)
exit (0)
exit (1)
↳ 0,255 range

We assume integer
Store 2 byte




terminated by signals also
<signal.h>

# fork() → If no child created it return error -1
↳ due to space not available
↳ limits creation of child

# wait()

Two condition to fail

(i) Do not have any child to wait
(ii) status pointer is pointing to invalid address

→ return -1 in case of error

* waitpid ( Pid c , *int status , int options )
optional

↓ wait for Some fixed child

↓ to check status which is mentioned in status pointer

Pid > 0 → waiting for Some particular child
Process group ← Pid = 0 → waiting for any child
Pid = -1 → act like wait()
↳ waits for any child

Calling process only wait for calling

# The options in waitpid()

- WCONTINUED → to check the status of any Continued child process.
- WNOHANG → not enforcing parent to wait for a child
  ↳ No more wait
- WUNTRACED → child is untracable
  ↳ child is terminated but status is not updated in 2byte

# exit(int) → library function defined in <stdlib.h>
  _exit(int) → system call
  ↳ <chistd.h>

First perform flush or cleaning buffer then internally invoke _exit()
↳ tempfile() is called for cleaning

# abort() → library function <stdlib.h>

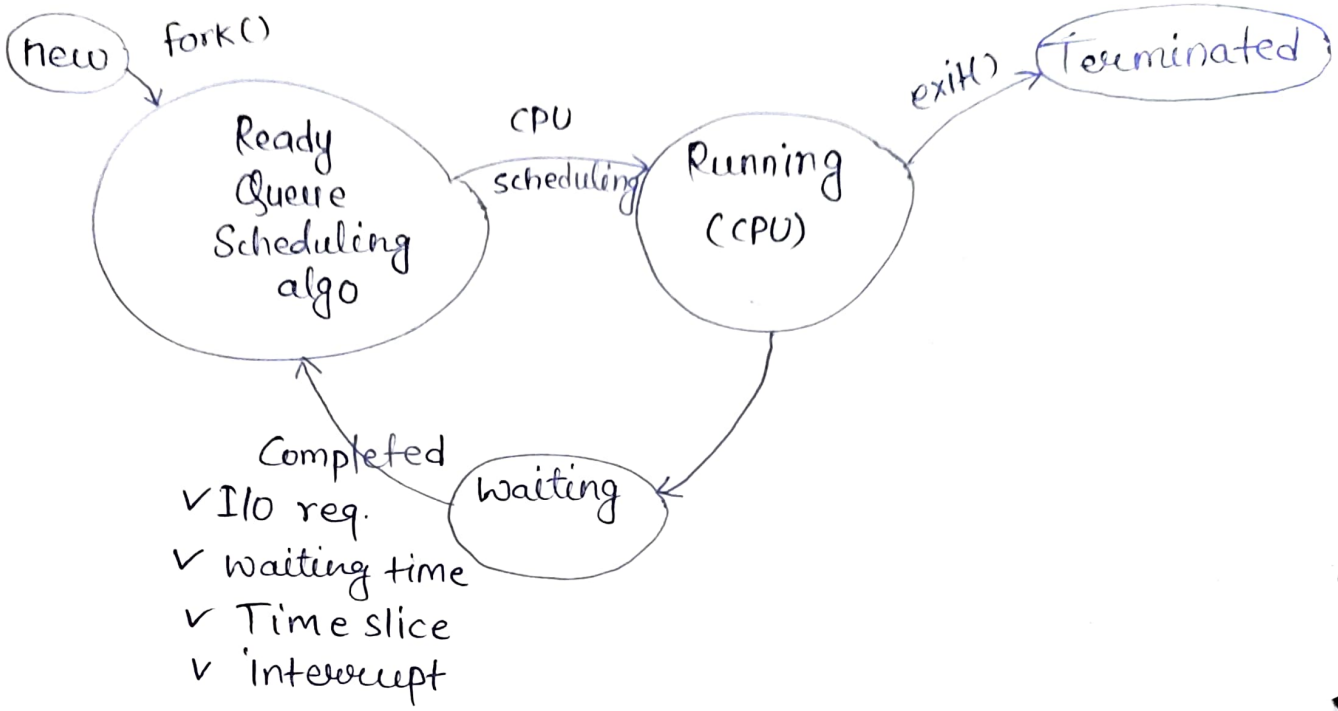
kill() → system call

For waiting

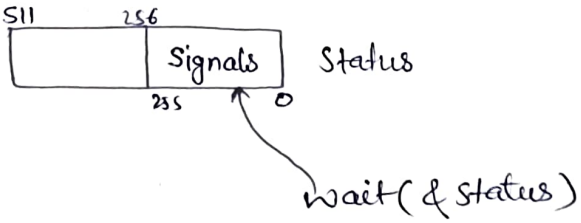
System calls:
- wait()
- waitpid()
- sleep(int) → int sleep(int sec)
  ↓ in unsigned form
  ↳ return most time 0 or remaining time
- Pause(void)

# int pause(void)
  ↳ return 0 / integer value
  ↳ we suspend the process until it awake by signal

→ Pause do not have any error




Date : 04/05/23



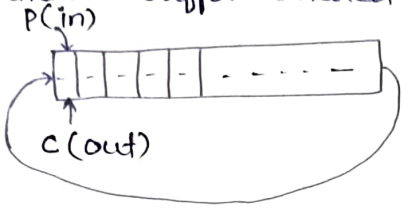

```
main() {
    int a, b, c;
a = fork()
printf("Hello")
if (a==0) {
    exec(filename) //child id
}
else
    :
    :
}
```

using exec child will not be same as parent

Date : 08/05/23

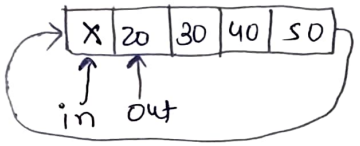
*** Shared memory

Bounded - buffer shared memory




empty
in == out

Full
(in+1)% BufferSize
== out




using only n-1 spaces

Date : 11/05/23

Disk Management (Unit-3)

Disk scheduling

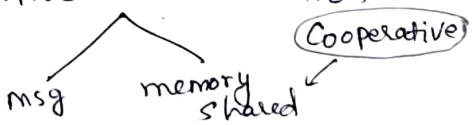
queue = 98, 183, 37, 122, 14, 124, 65, 67

FCFS , SSTF , SCAN , C-SCAN , LOOK , C-LOOK

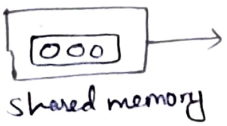
Date : 12/05/23

Race condition

Inter Process Communication




>1 process




→ inconsistent ← Race condition → Fixed order

If this is maintained inconsistency can be removed

main (){

Common variable, file table etc.

segment { = [ ] → shared → comes in critical section

segment { = [ ] → private

}

while(true){

entry section

Critical section - shared

exit section

Remainder section - Private

}

## Critical Section Problem

Algorithm *** * [ Mutual exclusion & Progress is mandatory
Bounded wait is optional.

Algorithm 3 Two process Solution

Date: 15/05/23

Deciding order of execution

$P_3 \rightarrow P_1 \rightarrow P_2$

Semaphores $S_1 = S_2 = 0$

| | | |
|---|---|---|
| wait($S_1$) | wait($S_2$) | $P_3$ |
| $P_1$ | $P_2$ | signal($S_1$) |
| signal($S_2$) | | |

# $P_1, P_2, P_3, P_4, P_5$

$P_4 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1 \rightarrow P_5$

No of semaphore used = $n-1 = 4$ (n: Process)

Let $S_1 = 0$
$S_2 = \cancel{0}\,1$
$S_3 = \cancel{0}\,1$
$S_4 = 0$

| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|
| wait($S_1$) | wait($S_2$) | wait($S_3$) | signal($S_2$) | wait($S_4$) |
| signal($S_4$) | signal($S_3$) | signal($S_1$) | | |

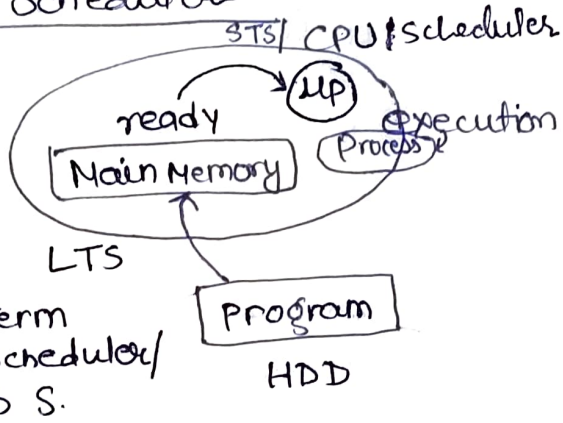
## Resource Management in Semaphore

R = 10

S = 10 , P = 15

| wait() | wait() | ..... | wait() |
|---|---|---|---|
| $P_1$ | $P_2$ | | $P_{15}$ |
| signal() | signal() | | signal() |

## Process Schedulers




Date: 22/05/23

5. Classical Synchronisation problem

### Mutex


Empty - shows how many slots are available or empty.

Full - Check whether anything is to consume or not.

### Semaphore

- Binary (0-1) - Mutex(1)
- Counting (0-N) < Empty(N) → check by producer
  Full (0) → check by consumer
  ↓
  No slots is filled

Full(1) → we have 1 data to access or consume or 1 slot is filled.

# Readers - Writers Prblm

Semaphore

| W | R |
|---|---|
| 1 | N |

wait(mutex) → use together
read_count

mutex = 1 (locking)

rw_mutex = 1 → Common to both R/W

read_count = 0 → keep track of how many processes are currently reading the data.

Reg ← rw-mutex
Reg ← Reg++
rw-mutex ← Reg
} run in one go all

wait(mutex)
↓
checking if there is any reader or not

mutex → used for only read-count
rw-mutex → for checking either reader or writer can enter

wait(rw-mutex)
↓
checking for writer
if (rw-mutex) value is 1 means there is no writer
if rw-mutex = 0 means there is a writer

## Dining Philosopheres problem

Two tasks either thinking or eating.

wait (chopstick[i])
↓
if value is 1 means it is available

## Remedies of dining philosopher problem

—

Date: 26/05/23

### 3. CPU Scheduling

Multithreading

$P_1$ $P_2$ ... $P_n$

Memory
↓
use same memory

Switching process
↓
mid-term scheduler

Preemptive & Nonpreemptive scheduling

Preemptive
↓
dep
$P_1$
After completion of $P_1$ it will release $P_1$
(Limited time)
If any interrupt occur like time slice / I/O req. it will release CPU

Nonpreemptive (Non sharing)
↓
without completion
releases
Forever till termination
↓
sharing of CPU

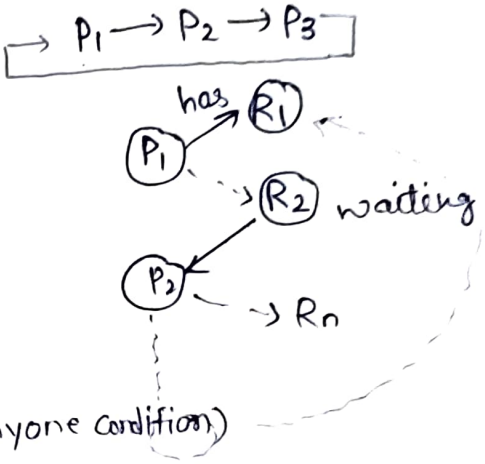
Burst time
↳ estimated time to complete ✓

Same Priority → Round-Robin

Date: 29/05/23

## Deadlock

Condition required for deadlock
i) No-Preemption
ii) Hold & wait
iii) Mutual exclusion (Nothing is shared)
iv) Circular waiting




- Deadlock detection (circle detect)
- Deadlock prevention (Safe) (Break anyone condition)
- Deadlock Avoidance (unsafe state)
- Deadlock recovery → costly

State

| Safe | Unsafe deadlock |
|---|---|

Date: 01/06/23

Banker's Algorithm

| | Allocated ABC | Need ABC | Available ABC |
|---|---|---|---|
| $P_0$ | 010 | 743 × | 230 ⑤ 1057 |
| $P_1$ | 302 | 020 ✓① | 532 |
| $P_2$ | 302 | 600 ✓⑪ | 834 |
| $P_3$ | 211 | 011 ✓⑪ | 1045 |
| $P_4$ | 002 | 431 ✓⑪ | 1047 |

Date : 02/06/23

Deadlock detection

Detection algorithm

Modified banker's algorithm

## 7. Threads

Exam #

<P_thread.h> → header files

PThread ( T1, T2, ... )

Create_thread ( T1 , NULL, &Sum, NULL )

reference passed ↓ address of function

```
main()
{
  voidsum()
  {
  }
  Void Sub()
  {
  }
}
```

Create_thread ( T2, NULL, &Sub , NULL)

→ we have two threads which perform Sum & Sub.

### Java thread

runnable interface → run() ← Java.lang Thread run()

→ No direct way to invoke run().

Java threads may be created by

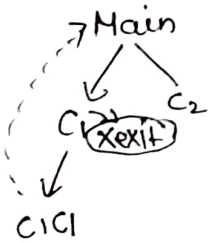
- Extending Thread class
- Implementing the runnable interface

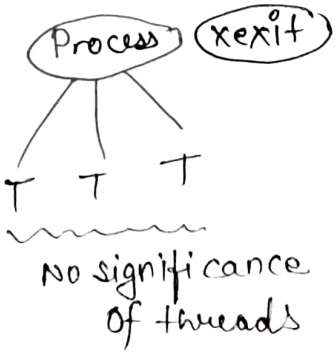
### Creating a thread

```
Thread worker = new Thread (new Task());
worker.start();
```

↳ It is in run()

("T1");



Date: 08/06/23

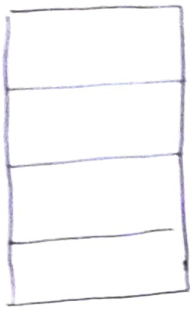
Memory Management

D fragmentation

1 to 6 + 13

#include<stdio.h> ↓ include at static

- Static loading (Compile time)
- Dynamic loading (Include when demanded)



Physical Memory (RAM)

External segmentation ↓ Performed on runtime a/c to demand

~~Paging~~ (Final exam)

Internal segmentation
- ↳ Fixed size partition
- ↳ Variable size partition

Date: 09/06/23

Exam

Disk scheduling Algorithm
fork(), layered architecture → benefits
Benefits of multiprogramming & multiprocessing system
Process Synchronisation problem, System calls
↑ State diagram

* -execl
exec ... except P

Threading (pthread.h)
create joint

return the id of currently executing process

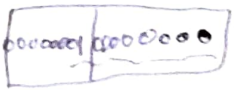
getpid, getppid } Both are used to get the information

init
main

Always successfully executed
Nothing to handle error

wait → return the id of just completed child to parent

wait (&status)
wait_pid (&status, Pid of child, 0)
↓ Status of just terminated child



256 / 256


signal(2)

exit(1)

last statement of any process

Sleep(10) → Suspend a process for some second

↓ return type is int (0)

execl("path", "filename", NULL)

list of a new program

execv("path", vector) except p

vector → [0] + filename

:

last statement should be NULL [10] → NULL

command line argument

PThread (ref, ref1, ...)

Create-thread( )

→ Minor (seniors) previous year paper

## File Management