Operating Systems

OS - Resource Manager - Collection of Program to manage system resources.
- System Software
- Does scheduling, Uses System calls - fork, pthread, etc.
  - ↳ Kernel Interfaces.

First Academic OS:
BSD - Berkley Standard Distribution
UNIX.

Now: OS exists for Usability and Efficiency.
Is the solution starvation-free, deadlock-free, etc.

Sub Topics: Process Management; Scheduling; PThreading;
[(Synchronize) Semaphore (DL) ← (Parallel Processing Setup)]
Memory Management

Scheduling Algorithms: SRT, Priority, Round Robin.
Shortest Remaining Time (Next) - Preemptive sjf.

	AT	BT	CT	TAT (CT-AT)	WT (TAT-BT)	Pid.	AT	BT	CT	TAT	WT
	0	20	20	20	0		1	0	7		
sjf	1	2	22	21	19		2	1	5		
	2	2	24	22	20 → Avg 13		3	2	3		
	0	2	2	2	0		4	3	1		
sjf	1	2	4	3	1		5	4	2		
	2	20	24	22	2 → Avg 1		6	5	1		

Scheduling Queue: $P_1^6\ P_2^4\ P_3^2\ P_4^*\ P_3^{*1}\ P_3^*\ P_6^*\ P_5^*\ P_2^*\ P_1^*$

Kernel Support for Scheduling Process: Sched-RR, Sched-FIFO, Sched-Other.

SRT ~~Check~~. Runs current process only upto Next Arrival

Round Robin: Stops multiple pre-emption. Let process run for atleast some time slice / time quanta.

Aged Process: Probability of a process scheduling increases with the amt. of time spent in queue.

	AT	BT	CT	TAT	WT
	6	4	24	18	14
	2	5	7	5	30
fcfs	3	3	10	7	4
	1	1	2	1	0
	4	2	12	8	6
	5	6	18	13	7

	AT	BT	CT	TAT	WT
	6	1			
	3	3			
sjf	4	6			
	1	5			
	2	2			
	5	1			

	AT	BT	CT	TAT	WT
	3	4			
	4	2			
SRT	5	1			
	2	6			
	1	8			
	2	4			

Priority < PE / NPE

- static : Priority not changing after scheduling.
- Dynamic : Priority changes as the process waits in Queue.

Best : PE Dynamic Priority.

Round Robin (PE) :

PCB - Process control Block (characteristics of Process)

Frequent switches as in SRT are costly.

Process context switching. [Ready state - Process can start]

Round Robin (TQ = 2)

Pid	AT	BT	CT	TAT	WT
1	0	4	8	8	4
2	1	5	18	17	12
3	2	2	6	4	2
4	3	1	9	6	5
5	4	6	21	17	11
6	5	3	19	14	11

GANTT Chart.

0 - 2	P_1 - 2
2 - 4	P_2 - 3
4 - 6	P_3 - 0
6 - 8	P_1 - 0
8 - 9	P_4 - 0
9 - 11	P_5 - 4
11 - 13	P_2 - 1
13 - 15	P_6 - 1
15 - 17	P_5 - 2
17 - 18	P_2 - 0
18 - 19	P_6 - 0
19 - 21	P_5 - 0

Ready Q : $\cancel{P_1}, \cancel{P_2}, P_3^*, P_1^*, P_4^*, \cancel{P_5}, \cancel{P_2}, \cancel{P_6}, \cancel{P_5}, P_2^*, P_6^*, P_5^*$.

<< indirection (append)
>> redirection

< overwrite
>

RR → Only one to use TQ.

Pid	AT	BT	CT	TAT	WT
1	5	5	32	27	22
2	4	6	27	23	17
3	3	7	33	30	23
4	1	9	30	29	20
5	2	2	6	4	2
6	6	3	21	15	12

Gantt Chart (TQ = 3)

0-1 Idle
1-4 P_4 - 6
4-6 P_5 - 0
6-9 P_3 - 4
9-12 P_2 - 3
12-15 P_4 - 3
15-18 P_1 - 2
18-21 P_6 - 0
21-24 P_3 - 1
24-27 P_2 - 0
27-30 P_4 - 0
30-32 P_1 - 0
32-33 P_3 - 0

RQ - $P_4, P_5^*, P_3, P_2, P_4, P_1, P_6^*, P_3, P_2^*, P_4^*, P_1^*, P_3^*$

NPE Pr.

P	A	B	C	TA	W.	Pr
1	0	4	4	4	0	4
2	1	5	16	15	10	5
3	2	1	5	3	2	7
4	3	2	18	15	13	2
5	4	3	21	17	14	1
6	5	6	11	6	0	6

Gantt chart
0-4 P_1
4-5 P_3
5-11 P_6
11-16 P_2
16-18 P_4
18-21 P_5

~~RQ: P_1^*~~

PE Pr.

P	A	B	C	TA	W	Pr
1	1	4	18	17	13	4 ✓
2	2	2	14	12	10	5 ✓
3	2	3	10	8	5	7 ✓
4	3	5	8	5	0	8 ✓
5	3	1	15	12	11	5 ✓
6	4	2	12	8	6	6 ✓

Gantt chart
0-1 Idle
1-2 P_1 - 3
2-3 P_3 - 2
3-8 P_4 - 0
8-10 P_3 - 0
10-12 P_6 - 0
12-14 P_2 - 0
14-15 P_5 - 0
15-18 P_1 - 0

RQ: $P_1, P_3, P_4,$

Left to discuss:

① HRRN (Highest Response Ratio Next) - NPE

② Priority P.E.

③. Overlapped Scheduling.

[People] OR - Operations Research (Typically use Gantt chart).

Overlapped Scheduling : Scheduling required for other resources : Eg I/O Operations.

Eg: P_1 requires file input, P_2 requires file output
P_3 requires printer, P_4, P_5, P_6 require CPU.

So when P_3 is printing, let P_4 use CPU.
i.e. don't leave CPU Idle - Increase throughput.

- Types of OS:
  1. Batch OS. - $P_1, P_2, \ldots, P_n$.
     CPU I/O
     - Until both CPU & I/O is complete, the process runs and CPU remains idle.
     Same disadvantage as FCFS.

  2. Multi-programming, Multi-Tasking, Multi-Processing Operating System.
     - Uses overlapped scheduling (MP).

* Process - Program in Execution

  - Multi-Tasking - Refined MP
    Uses Time Quanta / Time Slice.

  - Multi-Processing - Refined MT by using more cores in CPU.
    No. of Process (m) >>> No. of cores (n).

* Law: n Processor setup won't be n types faster than the single Processor
  Since 1 process breaking into n process will take some CPU time and hence it won't be n times

Important Objectives of OS.
  1. Resource Management
  2. User Interface

* Priority PE

Priority is Static.

Hence a process with high priority may starve

Thus Dynamic Priority is better.

HRRN (Highest Response Ratio Next) - NPE

Response Ratio = $1 + (W/S)$ W: Wait S: Service.

Overcomes (minimizes) Starvation which affect SJN and SRT.

HRRN

Pid	A	B	CT
1	1	1	2
2	4	2	6
3	5	6	12
4	6	2	14
5	7	4	18

calculate RR for 3 & 4.

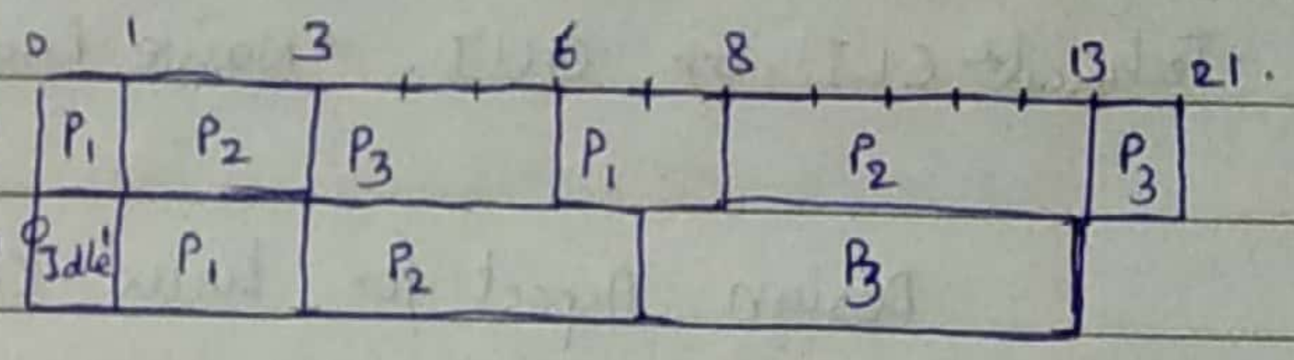
Favour longer waiting ones over new shorter ones.

$RR_3 = 1 + 1/6$; $RR_4 = 1 + 0/2$.

$RR_4 = 1 + 6/2 = 4$ $RR_5 = 1 + 5/4 = 2.25$.

overlap

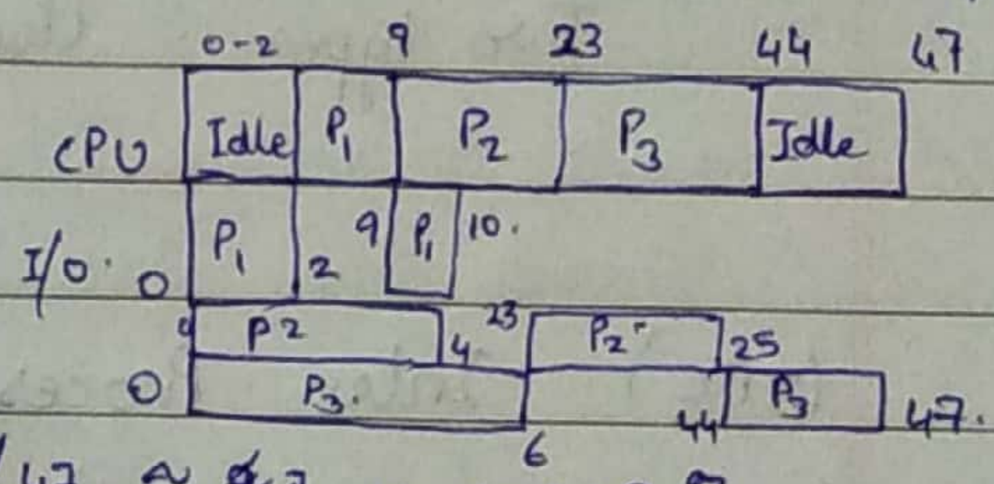
Pid	A	CPU	I/O	CPU	CT
0	~~1~~	~~2~~	~~2~~	8	
1	~~2~~	~~4~~	~~5~~	13	
2	~~3~~	~~6~~	8	21.	
3					



i.e. all parallel.

Presume I/O can be overlapped

	A	I/O	CPU	I/O	CT
P_1	0	~~2~~	~~7~~	~~1~~	
P_2	0	~~4~~	~~14~~	~~2~~	
P_3	0	~~6~~	~~21~~	~~3~~	



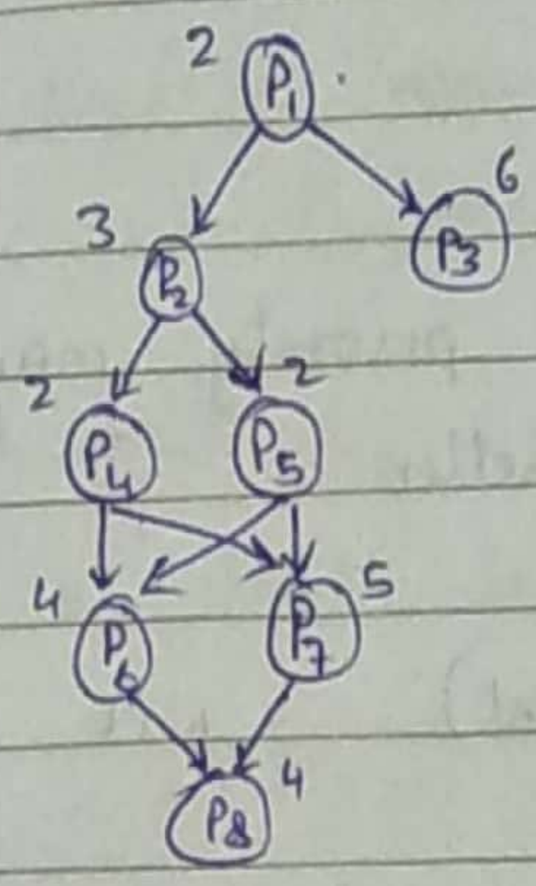
CPU Idle : $5/47 \approx$ 10.63829 %.

	A	I/O	CPU	I/O	CT
P_1	0	~~4~~	~~14~~	~~2~~	20.
P_2	0	~~8~~	~~28~~	~~4~~	50.
P_3	0	~~12~~	~~42~~	~~6~~	94

0-4 P_1 I/O
0-8 P_2 I/O
0-12 P_3 I/O
18-20 P_1 I/O.
46-50 P_2 I/O
88-94 P_3 I/O.

0-4 - Idle
4-18 - P_1.
18-46 - P_2
46-88 - P_3.
88-94 - Idle.

CPU Idle : $10/94 \approx$ 10.63829.



CPU ①	2-5 - P_2 5-7 P_4	7-9 - P_5 9-13 - P_6	14-18 - P_8
CPU ②	0-2 - P_1 2-8 P_3	8-9 - Idle 9-14 - P_7	

* <u>Process Concept in OS</u>:

Has : Threads, Sync, Malloc / Protection, Security.

2 Types : CPU Bound & I/O Bound.

2 Major Responsibilities of OS : ⓘ Resource Management (Efficient)

ⓘⓘ User Interface.

Usable & User Friendly. Increase Throughput, Decrease WT

WIMP - Windows Icon Mouse Pointer

Technical ← CLI or GUI. → Naive End User

Design Aspect for Linux Commands : 7 ± 2.

User can only remember upto range 5 to 9

~~Break down of Efficient Resource Management.~~

OS is required to support ⓘ "Program Execution".

Two types : User Program and System (kernel) Program

ⓘⓘ I/O Operations

ⓘⓘⓘ File Manipulations

(IPC) Inter Process (pipe) - ⓘⓥ. Communication

ⓥ Protection.

Program is passive (binary residing on HDD)

Process is active (program scheduled).

Each process has Address space. Multiple related process will have different memory space. Thus data sharing is done via pipe (IPC).

Scanned by CamScanner

Uniprocessor : $||^{n}$ processing = Time sliced / Interleaved Processing.
shm - shared memory.
System Boots : Pid = 01 : INIT – Parent of all process.
os.iiitdm@gmail.com

Protection : chmod rwerwerwe; chown - Change permissions of a file.

File Manipulation : Each device is a file.

System Calls :

Based on Process Control : fork(), wait(), etc.
File manipulation : read(), write().
Device manipulation :
Communication : pipe(), shm-get(), shm-set
Information / Maintainence : getpid(), getppid()
Protection :

Parent - Child Process : 2 Setups
① wait until child process dies.
② Let Parent process die - creates orphan process.

7/8/18

Fork - System Call - Process Creation / Management.
* Passive v/s Active.
Not entire OS is in the ram, only the core portion is. If some portion is required, it is called onto the RAM thereby becoming active

Process - Program in Execution scheduled on CPU.
Developmental user interacts with kernel using system calls.
Eg: Cramer's Rule - Eq^{n} Solver - 4 binaries (1 App).
Bring - $||^{n}$ processing or multi processing.

Process Creation - Memory intensive. ∴ Thread.
Process State : ① New - When a process is created, it is in New state and its details are stored in PCB (Process control Block).

sort is a child of bash process
Anything executing on bash is child of bash.

Failed fork returns
fork fails if critical resources are not available

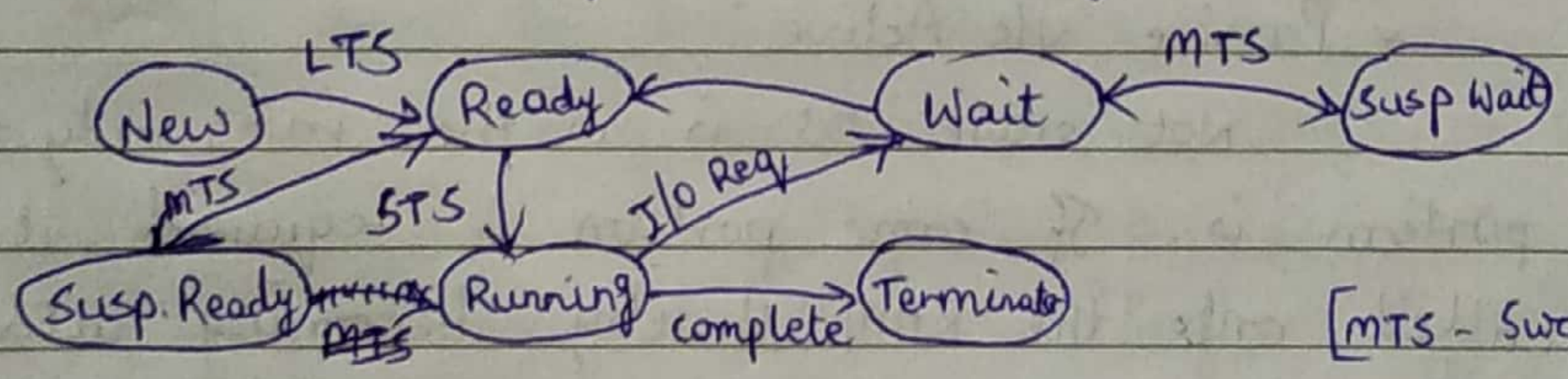
② Ready State : When basic amt of resources are available

3 Schedules : LTS, MTS, STS
All Sys Software: long term scheduler, Middle, Short.
LTS - schedules process to move into Ready Queue.

③ Running State : STS does scheduling of process from Ready Queue to Running State

④ Terminated State : If process completes, then it goes to terminated state

⑤ Waiting State : If CPU job is over and I/O is required then it goes to waiting state



[MTS - Swapper]

⑥ Suspended Ready : Used when Ready Queue is full or high priority process comes in. swap.

* Fork : vfork() - older version of fork.
used to create new processes. (developer user processes)

fork related system calls : wait, exit, getpid, getppid, exec[l, p, v]

→ Header files required : "unistd.h" ; "sys/types.h"
for datatype management.

Program → Process → Thread. (subdivision)

Kernel mode or User Mode. are 2 modes process run in.

clog and CEH

```
int main () {
    printf ("os course");
    Pid_t pid = fork ();
    if (pid < 0)
        fprintf (stderr, "Err Msg");
    else if (pid == 0) printf ("child");
    else printf ("Parent");
} return (0); }.
```

Process Characteristics (PCB):

① Pid ② State (Info).
③ Priority ④ PC (Program Counter)
⑤ Open files / devices
⑥ GPR Details
⑦ Protection (Permissions).

Process Sections:

① Program / Code / Text section.
② Data Section (stores global variables) ③ Process Stack (stores local variables).
④ Current Activity.

The ~~duplication~~ execution of parent in child happens only after forking point. but child is complete duplicate of parent. (or duplication of code happens after fork but data is ✓ fork has shared global variables only whereas fork has two completely different sets of memory. so different copies of all variables.

To share data, pipes will come in.

Variables characteristics: ① Name ② Datatype (from internal mem POV).
③ Address ④ Scope (from src code POV) ⑤ Storage Class. - Auto, Reg, Extern, Static.

Fresh allocation of memory by alloc inside every block. Static variables only get deallocated at the end of main. Extern variables are available across files. Reg is used for faster access. Generally Reg is used for looping variables (i, j, k).

Fork n times creates 2^n process: 1 parent, $2^n - 1$ child.

ls | grep "a" Here | is pipe.

Data sharing can be done via pipes or shared memory.

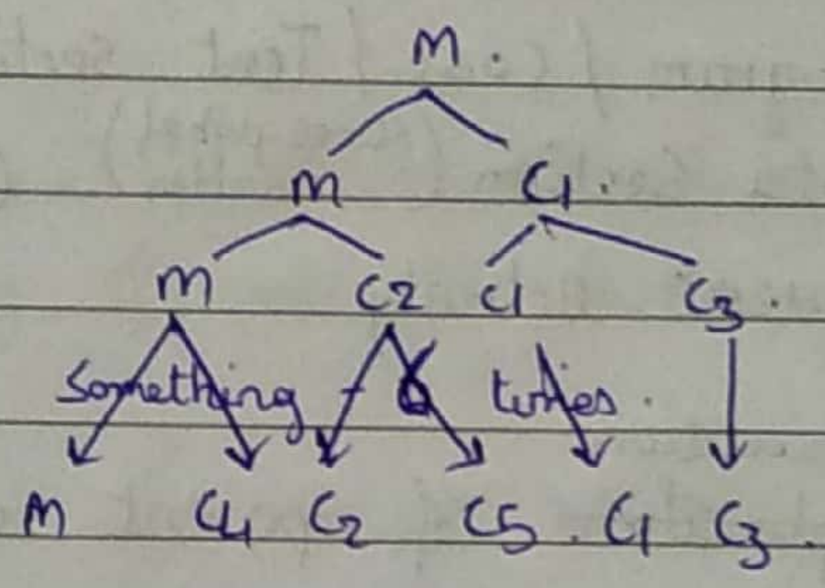
Lab:

* Parent gives n; child generates primes or fibonacci series of n terms.

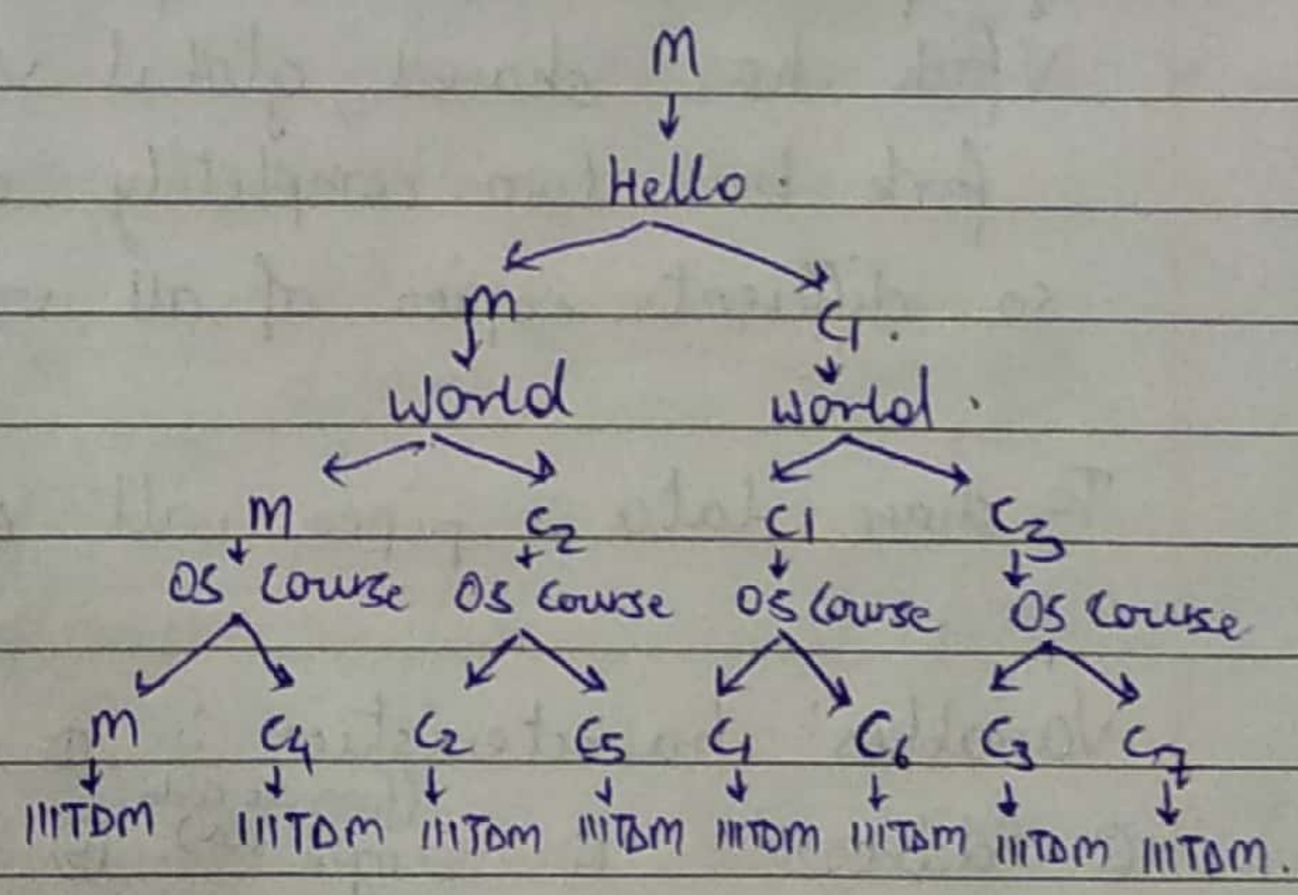
Here, if n is taken input from keyboard, then child won't have the value of its copy of n.

Examples:

① M()
{ pid=fork();
  if (pid != 0)
    fork();
  fork();
  p(something);
}.



② M()
{ p("Hello");
  fork();
  p("World");
  fork();
  p("OS Course");
  fork();
  p("IIITDM");
}.



Default: Most kernels prefer child over parent.

* Wait & Sleep:
Parent should wait() rather than sleep().
wait(NULL) - wait for all children to finish. to avoid orphan process.
Fork - Used to simulate multi-processing.
multi-processing (single processor); parallel processing (multiple processor)

execv : better dynamic call by giving vector.

* Exec :

Variants : execl, execp, execv, exece.
list, path, vector, environment variables.

exec cannot be supplied with functions (source code).
only with binaries.

```
if (pid == 0)
   execl("/bin/ls", "ls", "-al", NULL);
else if (pid > 0)
   wait(NULL);
```

execp is generally used as execlp
exec generally executes the path given to it

```
execlp("ls", "ls", "-al", NULL).
```

For execv, vector/array is used for passing cmds.

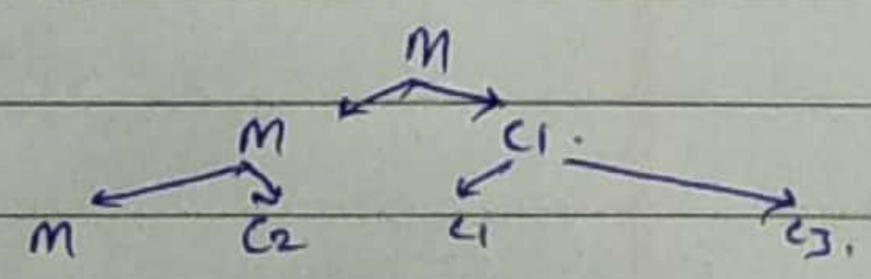
```
execv("/bin/ls", cmd);
* cmd[] = {"ls", "-l", NULL}.
```

or execvp("ls", cmd); or execvp(cmd[0], cmd);

Note "\n" in printf calls normal fork. If not given, output buffer is not flushed which causes problems with fork.

Find No. of "Hello\n";

```
A { f();
B   f() && f() || f();
E   f();
    p("Hello\n");
}.
```

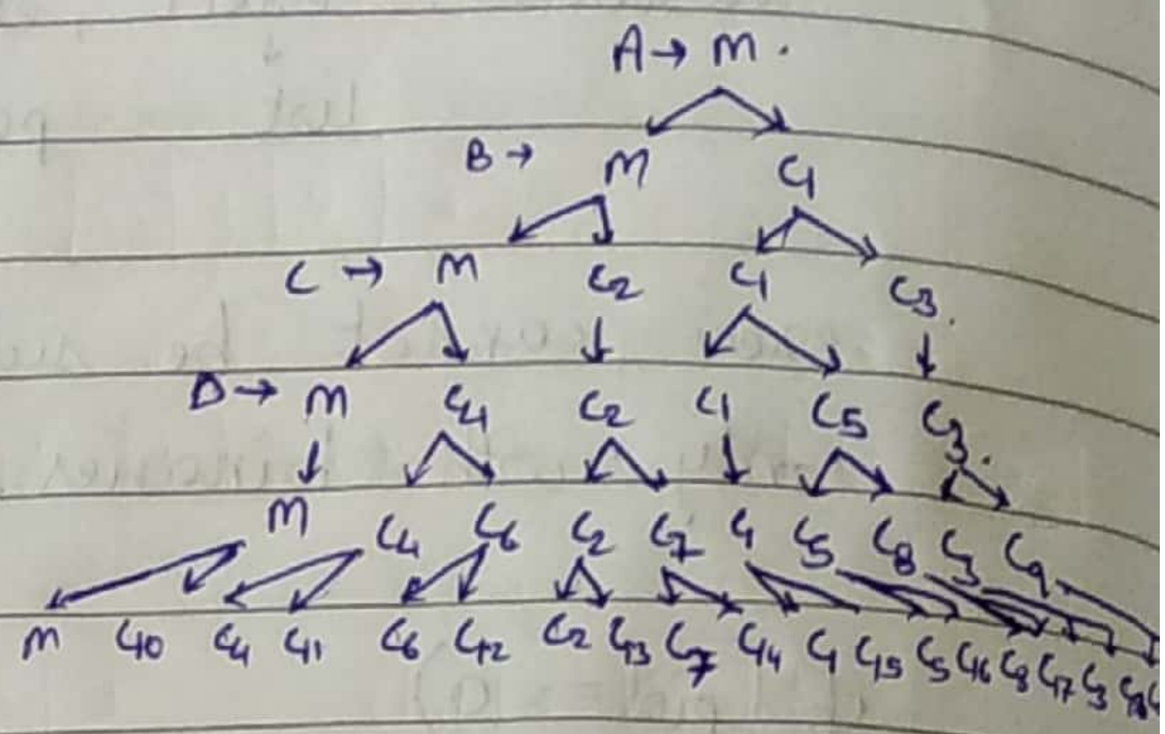


Pipes [used for ipc]. - Share Data.
&& and || are short circuited operators.
if (x=5&&1) Here && checks if 5 is non zero.

* Fork Buffering Issue:

Previous Eg: Solved:

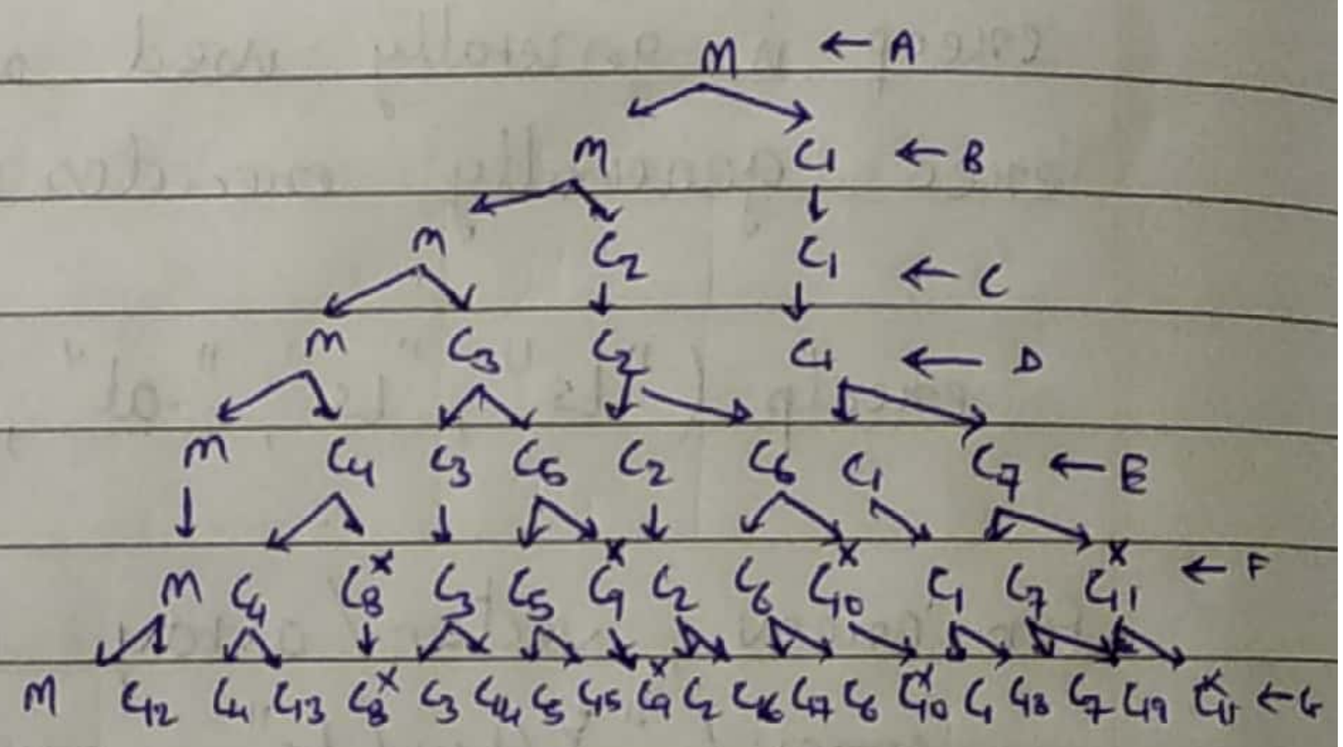
```
A f();
B f() && C f() || D f();
E f();
p("Hello");
```



C only executes where B is non zero i.e. B parents.
D ~~only~~ does not execute where the expression's LHS evaluates to 1 since the output will be 1.

Once more:

```
if (f() && f())
{ f();
if (f() || f())
{ f();
  f();
} p(Hello);
```



36 total children.

Once more:

```
p("Hello");
f();
f();
f();
p("World");
```

If Hello has "\n" then 1 print
else print 8 times.

"\n" - O/P Buffer. (Goes here first)
If overflows then sent to terminal
or if flush() is called to o/p buffer.
\n acts as a force flush.

Buffer is copied for each process.

Find Trace for: for (i=0; i≤3; i++)
{fork();}. printf("Hello\n");

Tutorial I

① Given all Process arrive @ 0 ; TQ = 1 in a RR setup complete $F_T(P_1)$ with Burst times as P_1 4, P_2 1, P_3 8, P_4 1.

② Overlapped Execution.
Compute $F_T(P_1, P_2, P_3)$.

AT	IO	CPU	IO
0	4	14	2
0	8	28	4
0	12	42	6

③ Assume context switch is 's' seconds and a process runs for Quanta 'q' sec; Compute time for P_1 to regain access in a RR setup.

④
```
if ( fork() && fork())
    fork();
if (fork() || fork())
    fork();
printf (" * \n");
```

⑤
```
for (i=1; i≤3; i++)
    fork();
printf (" * \n");
```

⑥
```
for (i=1; i≤3; i++)
{ fork();
  printf(" * \n");
}
```

⑦
```
for (i=1; i≤3; i++)
{ fork();
  printf(" * ");
}.
```

⑧
```
for (int i=0; i≤2; i++)
{ if (fork() == 0)
    printf (" * \n");
}.
```

Answers:

①.

Pid	AT	BT	CT
P_1	0	4	9
P_2	0	1	2
P_3	0	8	14
P_4	0	1	4.

$\boxed{F_T(P_1) - 9}$

Gantt Chart: TQ = 1.

P_* -	0-1 P_1-3.	6-7 P_1-1
	1-2 P_2-Done.	7-8 P_3-5
	2-3 P_3-7	8-9 P_1-Done
	3-4 P_4-Done	9-14 - P_3-Done
	4-5 P_1-2	
	5-6 P_3-6	

Ready Q: $\cancel{P_1}, P_2^*, \cancel{P_3}, P_4^*, P_1, P_3$

Scanned by CamScanner

ls | more } piping @ CL
ls | grep 'a'

a.c → Compiler (Producer) $\xrightarrow{a.o}$ Assembler (Consumer, Producer) $\xrightarrow{a.obj}$ Linker (Consumer, Prod) $\xrightarrow[a.out]{a.exe}$ Loader (Consumer).

linker links printf with its input.

* Inter-Process Communication (IPC's).

Vfork should not be used to bypass IPC.

P-C relationship - Need for exchange data

Vfork only shares parent's data to child process.

IPC - Implemented by (i). Pipes (ii) Shared Memory.

Why share data : (i) Information Sharing
(ii) Modularity (iii) Convenience for Parallel Processing
(iv) Communication Speedup.

Process Type : (i) Independent (ii) Co-operative.

'Producer - Consumer' Paradigm : client-server.

Eg: (i) Capitalise output of ls ; (2) Transliterate output of ls.
Linux Command : tr.

vfork: Memory leaking ; If no exec, then recursive forking.

Pipes : Simplex ; Half Duplex ; Full Duplex
Issues in Full Duplex : Synchronization.

Example : Unidirection :

```
(1) → #define BS 50 | REND 0 | WEND 1. //3 separate #define.
      (Buffer Size)  (fd[0])   (fd[1])
      int main(){
      char wm[BS] = "IIIT", rm[BS];
      int fd[2];
      if (pipe(fd) == -1) {printf("Error: Failed Pipe\n");
        return(-1); }.
      pid = fork();
      if (pid > 0)
      { close(fd[REND]);
        write(fd[WEND], wm, strlen(wm));
```

1 - stands for output device. (stdout).
0 - stdin.

```
    close(fd[WEND]);   //When pipe is successfully
}else if (pid == 0)    //created, both REND & WEND are
  {  close(fd[WEND]);  //open.
     read(fd[REND], rm, BS);
     close(fd[REND]);
     printf("%s", rm);
}}
```

Bidirectional : (simultaneous).

```
int main(){
 int ~~fd[2]~~, fd2[2], fd1[2]; char rm1[25], rm2[25];
 pipe(fd1); pipe(fd2); char wm1[25], wm2[25];
 pid = fork();
 if (pid > 0)
  { close(fd1[REND]);
    close(fd2[WEND]);
    write(fd1[WEND], wm1, strlen(wm1));
    read(fd2[REND], rm1, 25);
    close(fd1[WEND]); close(fd2[REND]);
  }
 else if (pid == 0)
  { close(fd1[WEND]); close(fd2[REND]);
    read(fd1[REND], rm2, 25);
    write(fd2[WEND], wm2, strlen(wm2));
    close(fd1[REND]); close(fd2[WEND]);
}}
```

After an execvp of ls, output goes to TTY.
To capitalise the o/p, let it go to a file, specifically pipe. Hence, we use dup2.

In child block:

```
∴ dup2(fd[WEND], 1);
  execvp("ls", argv);
```

In Parent block

```
while(read(fd[0], buf, BS) > 0)
  toupper...;
```

Read - Generally called as Blocking Read.
since it is blocked until write is complete

^D , ^Z - Eof Characters.

Simulate grep & finger.
Program to capitalise ls output.

```
if (pid == 0)
{ dup2 (fd[1], 1);
  execvp ("ls", argv);  ~~close (fd[1]);~~
}
else if (pid > 0)  { close (fd[1]);
nb=
while(read (fd[0], buffer, BS) > 0)
  { for (int i=0; i<nb; i++)
buffer[i] = toupper (buffer[i]);
  write (1, buffer, nb);
```

For transliterate : Parent : { close(fd[1]);
dup2 (fd[0], 0);
execlp ("tr", "tr", 'a-z', 'A-z'); }.

Write - Send — Blocking & Non Blocking.
Default.
Block till read what is written
Read - Receive - Blocking & Non Blocking.

* MultiThreading :
Application = Process ← Program in Execution.
Atomic Unit of CPU Utilization → Thread.

Process → Collection of Threads.
OpenMP : Parallel Programming Platform.
pragma - Block for parallel execution.

Threads in OS ≅ Functions in C (But to increase Performance).

Task — Series
Parallel.

Moores' law favors HW. Amdahl's law & Wirth's law

WL: S/W gets slower faster than H/W gets faster.
Ubiquitous Computing - Pranav Mistry. (6th Gen)

SW Wirth's Law - Instruction level Parallelism (ILP)
Amdahl's Law - Limit to Parallelism.

* Advantages of Multi-Threading:

(i) Responsiveness increases.
Comparing to browser, it cannot display only when all content is ~~displ~~ loaded; the text should be displayed when it is loaded, etc. The text, video & image should be independent. It should not be one 'main' function.

The loading bars do not actually display the task completed; Ribbed Loading Bars are generally used to retain the user and not lose him. The loading bars moving forward gives user the feeling that task is executing quickly.
HCI - Feedback!
Good feedback → Multithreaded appⁿ.
Pulsating Loading Bar: IE Download Bars.

(ii) Scalable. (If developed multithread appⁿ for uniprocessor, it cannot work on multiprocessor down the line).
OS needs support of MultiThreading.

(iii) Sharing:
DLL - Dynamic Link Libraries.
One process called 5 times, doesn't open 5 process files but 1 process file is shared.
Threading shares data, fork doesn't. (TnC Applied).
In general, Threading helps resource sharing (Eg: Global variables).

Sort [Internal (In-Place) - Bubble, merge, etc. - Entire data in RAM.
External - Bucket, Counting, Radix.]

Exchange sort: atomic in In-Place sort.

Development Phases: SRS - Analysis - Design (Architecture / Interfaces) - Implementation - Testing

(iv) Economy: Cost of forking > cost of threading.

* Linux Interfaces: POSIX (Portable OS Interface).
  Makes <pthread.h> available
  Thread definitions: runner functions

* Challenges of Multi-Threading:
  (i) Identify parallelizable tasks.
  (ii) Load balancing.
  (iii) Data Splitting [Data Parallelism]
  (iv) Data Dependency. [Tasks]

  [Non Thread s/w v/s Threaded s/w ⇒ Complex Testing.]

Design creates interfaces so that Implementation/Coding team uses it - Cohesion & Coupling.

Testing: Test for Boundary Cases - Prove s/w works.
Invalidate input (Eg: Marks > 100 should not be input)
Integrated testing

Testing is easy if the s/w goes from one stable state to another. If multi-threaded, then this flow is difficult to follow. Test cases are lot more in multithreaded. [Empirical Approach].

Eg: $T_1 \rightarrow T_2 \rightarrow T_3$ or $T_2 \rightarrow T_1 \rightarrow T_3$, etc.

Safety critical Programmer reduces failure probability (system should never go in unstable state).

* Formal methods in CS - Used to prove

Empirical: Instantiated for specific Input. Can't conclude will work for all inputs if works for 5 input.

* Amdahl's law:
  Limit to Parallelism extent.
  N core solution is not N times faster than 1 core sol^n

Statement : Speedup $= \frac{1}{S + \frac{(1-S)}{N}} \leq N$.

Speed = Increase in Efficiency.
S - Probability of Serializable Component.

Speed $\leq \frac{1}{S + (1-S)/N}$ N = No. of cores.

Eg: P = 0.75 ; S = 0.25
N = 2 Speedup = 1.6
N = 4 Speedup = 2.3. (can speedup be 100%)
Ideal Case : N = ∞ ; Speedup = (1/S)
So, even if infinite cores are there, the serial component of the code will dictate the speedup.
Can't achieve the speedup equivalent to not having S.

ie. Speedup can't reach N.

Let T(1) be time to solve on uniprocessor setup and T(n) be time to solve on n processor setup

$\therefore$ Speedup $= \frac{T(1)}{T(n)}$ $T(n) = S(T(1)) + \frac{1-S}{n} \times T(1)$

$\therefore$ Speedup $= \frac{S+1}{S + (1-S)/N}$

* PThread : POSIX Thread.
# include "pthread.h".

Aspects of threading : Thread models and Thread Pooling.
Fork is costly on memory front.
Fork till fail : ulimit → Users limit of process creation.

2 Types of threads : users thread & kernel thread.
Models: ① Earlier : m user thread : 1 kernel thread.
If one thread makes blocking sys call then none of other threads can proceed.

② Then : 1 user thread : 1 kernel thread.
User can create big no. of threads, creating so many threads (kernel). Best for exploiting multi-threading. costly solution.

③ Now : m user thread : n kernel thread
(m >> n)
Thread Pools : 10 Threads. (Pool.)
when done, it returns back to pool
Since cost of initialization and de allocation of thread memory is high.
Collection of Busy Thread and Free Thread.
Pool of Kernel Threads. If done, the thread returns to pool and not killed, but the user thread is killed.
STS controls kernel thread. LTS controls user threads

Generally threads have cmdline arguments.

code continued : main ()
{ pthread_t {tid}; // → maps to runner (routines) functions.
pthread_attr_t attr;
pthread_attr_init (&attr);
pthread_create (&tid}, &attr, runner, argv[1]);
/* Dynamic Binding : Only pointers. (Eg: OOPS Polymorphism)
Even C can do that using function pointers. C++ does this by the compiler. Device driver companies rely on.
(Java supports good multi threading for users POV). */
pthread_join (tid, NULL); // Equivalent to wait call.
}

Runner Routines.
wait is process level support.
pthread_exit(0); signals main → Include in runner.

Validation: if (argc != 2)
{ ~~cout~~ printf(stderr, "Error");
return -1;
}.
if (atoi(argv[1]) < 0).
{ fprintf (stderr, "Error");
return -1;
}.

// void *1 → generic pointer (Not NULL pointer)
// int *a= NULL → NULL pointer.

```
void *runner (void *parameter)
{int ul = atoi(parameter); ~~sum~~;
    for (int i=1; i ≤ ul; i++)
        sum + = i;      // Global Sum.
}
```

* Thread level Scheduling: Interfaces.
  - 2 Types → User Threads (UT)
    - → Kernel Threads (KT)
  - → Light Weight Process

T_i → UT → KT → CPU.
UT scheduled for getting KT which is scheduled on CPU.

2 Scope Settings:
① Process Contention Scope (New) → (Ready) (T_i → UT)
② System Contention Scope. (Ready) ← (Run) (UT → KT).

Set Scheduling Policy: ① RR ⓘ FCFS ⓘⓘ Others.

To execute in multi threaded fashion: using specific scheduling policy:
① check Process Contention Scope setting

Diff: malloc and calloc.
↓
Need to typecast

(i) Set scopes (iii) Read policy. (iv) Set policy.

pthread_create → (UT → KT).

Eg: Matrix Multiplication:

```
# define M 3 // (Matrix Sizes: m×k ; k×N ; Result m×N)
 —//— k 2 // Make as cmdline arguments.
 —//— N 3

int A[m][K] = { {1,2}, {3,4}, {5,6} }, B[K][N] = {
  {1,2,1}, {4,5,4} }. , C[M][N] ;
/* No data dependency issues. One thread to
  calculate Result (1,1), (0,0), (1,0), etc.
C(0,0) → T1 , ... , C(2,2) → T9 : (i,j) → Data ↓ */.
//                                  for runner function
  struct v { int i,j } ; //Since only 1 data argument can be sent.

main()
{ pthread_t tid[9]; int count = 0;
  pthread_attr_t attr[9]; //Just one instance is enough.
  ~~pthrea~~ //Since attr will be used again for 9 thread loop.
  for (i=0; i<M; i++)
    for (j=0; j<N; j++)
    { struct v *data = (struct v*) malloc (sizeof (struct v));
      data→i = i ; data→j = j;
      pthread_attr_init (&tid[count], &attr)
      pthread_create (&tid[count], &attr, runner, data);
      count ++;
    }. for (int i=0; i<M; ~~i++~~ +N; i++)
      pthread_join (tid[i],NULL);
  for (int i=0; i<M ; i++)
    for (int j=0; j<N ; j++)
      printf ("%d", &c[i][j]);
}.
```

Read Monte Carlo - Calculating Pi

ISR - Interrupt Service Routines.

```
void * runner (void * parameter)
{ int sum=0;
  struct v* data = parameter;
    for (z=0; z<N; z++)
        sum += A[data->i][z] * B[z][data->j];
  C[data->i][data->j] = sum;
  pthread_exit(0);
}.
```

* Scope, Policy Settings:
  SIGNAL Handling in C or Linux. (IPC Mechanism)

  Thread Scheduling:
  Scopes : PES, SCS.
  Sched_Policy : SCHED_RR ; SCHED_FIFO ; SCHED_OTHER.

  Scopes Keywords : PTHREAD_SCOPE_PROCESS
  PTHREAD_SCOPE_SYSTEM.

  Scheduling Interfaces:
  pthread_attr_get_scope , pthread_attr_set_scope
  pthread_attr_getschedpolicy
  pthread_attr_setschedpolicy

* Signal Handling:
  2 Basic Signals ~~Types~~ : Kill and (other) Signals.
  (Internally) Notification of an event to other process (thread)
  Eg: Ctrl-C - SIGINT Ctrl-Z (Suspend)
  ↓ Terminate ↓ Stop (SIGCONT)

  # include "signal.h"
  Form of IPC.
  Signal Handling is used so that user can decide what will happen when a signal is sent or an event occurs.
  2 Basic Signal Types - Sync and Async.
  Errors - Exception Handling.

MPI - Message Passing Interface
Scopes - Contention Scopes.

a[-8] - Out of Bounds Exception.
Divide by zero -
try, throw, catch - Simulate Exception Handling.

Hardware event - Error. Software - Exception.

Steps signals are handled: (Phases of Sig Hand)
① Signal Generation
② Signal Delivery (MPI)
③ Signal Handling.
} can Happen in different scopes.

SIGSTOP & SIGFPE - Floating Point Exception.

Synchronous: Point of Occurrence & Handling is the same.
Asynchronous: Different scopes.
Eg: Ctrl-C - Occurs in the input device kbd and sent to software.

<exception.h> For c++.

* Scope code:

```
int main()
{ int scope, int policy;
  pthread_t tid[NT]; pthread_attr_t attr;
  pthread_attr_init(&attr);
  if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Failed at Scope");
  else if (scope == PTHREAD_SCOPE_PROCESS)
      printf("Process Contention Scope");
  else if (scope == PTHREAD_SCOPE_SYSTEM)
      printf("System Contention Scope");
  else fprintf(stderr, "Ouch");
```

2 Types of Functions:
Inline && Non-Inline (Scope Resolution Operator),

g++(-c) m.cpp f1.cpp f2.cpp → m.obj + f1.obj + f2.obj → a.out

/usr/src → Implementation : /usr/bin → Interface

```
if ((pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM)
        fprintf(stderr, "Setting Problem");        != 0)
for (i=0; i< NT; i++)
pthread_create(&tid[i], &attr, runner, NULL);
    pthread_join(tid[i], NULL);

//Setting Policy
    int policy
if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Policy Get");
else if (policy == SCHED_FIFO)
        printf("FIFO");
else if (policy == SCHED_RR)
        printf("RR");
else if (policy == SCHED_OTHER)
        printf("Other");
else fprintf(stderr, "Oops");
if (pthread_attr_setschedpolicy(&attr, SCHED_RR);
    for (i=0; i< NT; i++)
    pthread_create(&tid[i], &attr, runner, NULL);
pthread_join(&tid[i], NULL);
```

* Try HRRN using SCHED_OTHER.

* MakeFiles:
  n functions = n source files
  ⇒ Multiple Source.
  Compile without main.
  Main is generally defined by Lead Programmer
  Eg: Rational Number Arithmetic.

Forking in Threading = pthread_atfork()
  Fork in Thread = Duplicate the calling thread.

3 ways to Handle

SIGILL - Illegal Instruction.
: SIG~~_I~~GN - Ignore Signal (_ = Response).
SIG_DFL - Default Signal Handle
let User Handler do its job

* Handling of Signals:

3 Phases - Generate | Deliver | Handle
C++/Java (Exceptions:) try throw catch.

System Calls: kill( ), raise( ), signal( ).
↓
kill command - terminates a process.
↓
(MISNOMER - Used to Generate & Deliver)

kill & raise - generate & deliver
signal - handler

Four Possible (Approaches) Actions:
Why handle signals : IPC, [Asynchronous Events]
↓
Interrupts.

Error - Hardware -(^-C) ; Exception - Software (a[-8])

① Terminate ② core [log file] for mem Access].
③ Ignore ④ Stop and ⑤ Continue.
(Same as continue but has Kernel Handler support) (User Defined H

"signal.h" → 31 signal Names:

SIGHUP (Hang-up) SIGINT SIGILL
SIGABRT (Abort) SIGFPE SIGSEGV.
SIGKILL SIGCONT SIGSTOP

Syntax: kill(pid, signo); pid = 0 All User Process
pid > 0 : sig given to pid.
raise(getpid(), signo); Deliver to the calling process
All pids are live PID
kill(getpid(), signo) ≡ raise = Quit() in C.

SIGCHLD - child sends when exiting.
'^\' - SIGQUIT
SIGKILL cannot be handled.

Hang-up = Indefinite Hold: SIGHUP, SIGINT, SIGQUIT.

Code:
```
pid = fork()
  if (pid == 0)
  { // Register Signal Handler
    signal (SIGHUP , UserHandler1) ; // UserHandler = Pointer function
    signal (SIGINT , UDH2) ;
    signal (SIGQUIT , UDH3);
    for (;;) ;
  }
```
Eg:
```
signal ( SIGINT , SIG_IGN);
for (i=0 ; i < 10000 ; i++)
  { Important Mem Access Instruction
    ...
  }
signal (SIGINT , SIG_DFL);          // Reset
```

If assert( ) [macro] fails : SIGABRT.
↳ similar to if else
assert(newptr != 0) → string replacement.
Fails = call C function abort() ≡ SIGABRT.

SIGHUP Eg: Parent terminal — startx, Netscape Navigator, etc.
↓ Dies → SIGHUP to all child.
Init takes over those process who received SIGHUP.

code continued.
```
else if (pid > 0)
  { print("Parent Throws Hangup");
    kill (pid, SIGHUP);
    sleep (5);
    // same for sig:INT & SIGQUIT
```

Default is not Quit for every signal.

* Tutorial.

① What happens when SIGINT hits the following process.

```
void response (int sig_no)
{ printf("Operating Systems Course \n"); }
int main ( )
{ struct sigaction act ;    // Inbuilt structure
  act.sa_handler = response ;
  act.sa_flags = 0;
  sigEmptyset (& act.sa_mask);
  sigaction (SIGINT , &act, 0);
  while (1)
   { printf ("IIITDM \n");
       sleep(1);
   } return(0) ; }.
```

ⓐ Process Terminates ⓑ String 'OS' printed.
ⓒ Both b and a ⓓ None of the above.

②
```
void response (int sig_no)
{ printf (" %s initiated \n" ,sys_siglist [sig_no]); }.
 int main( )
{ alarm (2) ;   sleep (10) ;   printf ("IIITDM\n");
  signal (SIGALARM , response) ;
  return 0 ; }.
```

ⓐ IIITDM ⓑ ALARM Clock ⓒ Nothing ⓓ None of the above.

③ void response (int sig-no)
{ printf (" %s ", sys_siglist [sig-no]);
printf ("Handler called "); }
int main ()
{ child = fork()
switch (child)
{ case -1 : perror ("A"); exit(1); break;
case 0 : kill (getppid (), SIGKILL);
printf ("Orphan ..\n"); break;
default : signal (SIGKILL, response);
wait (& status);
printf ("Parent alive"); break;
} return 0; }.

ⓐ child kills parent ⓑ Parent kills child.
ⓒ Handler executes as sig arr to Parent ⓓ None.

④ child = fork()
switch (child)
{ case -1 : // same as before
case 0 : while (1)
{ printf ("child \n"); sleep(1); break;
default : sleep (5);
kill (~~sig~~ child, SIGINT);
printf ("child killed by Parent\n"); break;
} return 0; }.

ⓐ child kills Parent ⓑ Parent kills child.
ⓒ Both are killed by each other ⓓ None.

⑤ child = fork();
switch (child)
{ case -1 : //Same as before,

```
default : signal (SIGCHLD , response);
          wait (& status) ; break ; }
void response ( int sig-no)
{ printf ("%s", sys-siglist [signo]); }.
```

ⓐ Prints Nothing ⓑ Child exited. ⓒ Seg fault
ⓓ None.

⑥. What is not true of these options?

```
void response (int sig-no)
{ printf ("%s", sys-siglist [sig-no]);
  signal ( SIGSEGV , SIGIGN); }
int main () { signal (SIGSEGV ; response);
  char *str ; *str = 10 ; return 0; }.
```

ⓐ Kernel sends SIGSEGV to Process
ⓑ Handler will execute after signal receipt
ⓒ Both a , b ⓓ None.

⑦. What happens if ^C is pressed 2 times?

```
void response (int sig-no)
{ printf ("OS Course \n"); signal (SIGINT , SIG_DFL); }
int main ()
{ signal (SIGINT , response);
  while (1) { printf ("IIITDM\n") ; sleep(1) ; }.
  return 0; }.
```

ⓐ Process Terminates on 1st ^C press
ⓑ Process Terminates on 2nd ^C press.
ⓒ No termination ⓓ None.

⑧.
```
void fn (void * arg)
{ pthread_exit ("Bye"); }
int main ()
{ pthread_t pt; void *result;
  ret = pthread_join (pt, &result);
  printf ("%d", ret);
}
```

ⓐ -1 ⓑ 2 ⓒ 3 ⓓ 0.

⑨
```
main () { printf ("Quiz 2");
  pthread_exit ("Bye");
  printf ("OS Course"); return 0; }
```

ⓐ OS Course ⓑ Quiz 2 ⓒ Bye ⓓ None

⑩.
```
void response (int sig_no)
{ printf ("%s", sys_siglist [sig_no]); }.
main () { signal (SIGINT, response);
while (1) { printf ("OS ..."); sleep (1); } return 0;
```

What happens on occurrence of SIGINT

ⓐ Interrupt ⓑ stop ⓒ. Terminate. ⓓ None.

①.
H14. → H02 → H01. → H11.

End of Tutorial.

P/C - Producer Consumer.

Revision.

Sharing Data - Advantages

(i) Info Sharing (ii) Modularity

(iii) Convenience for ||" processing (iv) Comm. Speedup

Multi-Threading - Advantages.

(i) Responsiveness (ii) Scalable

(iii) Sharing (iv) Economy.

Multi-Threading - Challenges

(i) Identify ||" task (ii) Load Balancing

(iii) Data Splitting (iv) Data Dependency.

(v) Testing.

Resume.

*. Synchronization - Semaphores, MUTEX locks.
For Threaded Setup

Simultaneous Access of Data (write)
Map to Lost update in DB (Interleaved Execut
for Non serial access which causes conflict
Check for cycles in the Graph (DAG)
We handled these issues in Concurrency
Management. ≅ Synchronization.

Problems
~~Solution~~ : (i) Dining Philosophy (ii) R/W (iii) P/C.
Round Table : One needs 2 forks (1 fork given
per person) - How to share resources.
↓
Data sharing in a Consistent fashion -
If not done then Race Around State.

Interface Support for Mutual Exclusion : MUTEX
(pthread_mutex) PTHREAD_MUTEX.

Binary locking Protocol in DB ≅ PTHREAD_MUTEX in OS.

↓ Problem : Many read will have to

∴ Many read is parallel & write has loc
≅ Semaphores.

Producer / Consumer Simulation :
1 Thread generates , another consumes
For Implementation - Semaphores is req^d

Interfaces : SEM_WAIT

Can simulation be done w/o system calls o
interfaces? - Yes.

```
int buffer_array , in , out   //Buffer Size = BS
  //Generate only upto BS ; In = Next Free place = F
  // Out = First Full ≅ Pop (Read)
//More equivalent to P = Increment & C = Decremen

Producer :  while (true) {
    while ((in +1) % BS == out)  ;  //Keep waiting
    buf [in] = item();
    in = (1 + in) % BS.;
  } // counter ++ ;
//Assuming kernel is Non-PreEmptive.
Consumer : while (true) {
        while (in == out) ;   //Keep waiting;
        cons = buf [out];
        out = (out +1) % BS ;
  } //counter - - ;
```

If kernel is Pre-Emptive - Problems arises.

programming language C++ ≅ C with new OO features added
Not ++C ≅ Old C gone

Bring in counter = No. of items.

```
Producer: { while ( ctr == BS) ;//keep waiting
            buf [in] = item();
            in = (in+1) % BS
            counter ++;
Consumer : while ( ctr == 0) ;
           cons = buf[out];
           out = (out + 1) % BS ;
           counter --;
```

Now only counter has conflicting access
Checking the register Implementation:

Producer (counter ++)	Consumer (counter --)
(A) $r_1 = ctr$	$r_2 = ctr$ (C)
(C) $r_1 = r_1 + 1$	$r_2 = r_2 - 1$ (C)
(B) $ctr = r_1$	$ctr = r_2$ (D).

Assume Order = ACBD [Eg: Counter = 5].
In the end counter = 4.
Inconsistent state ≅ Race Around.
Producer races to push 6 & consumer racer to push 4.

Peterson's Algorithm: - Followed by semaphores.
Characteristics:
① Mutual Exclusiveness.
② Progress (X Deadlock)
③ Wait - Bounded (X Starvation)

If the algo satisfies above characteristics then it is a solution for synchronization ~~solution~~ problem
Progress should happen after Bounded wait.

Date
Page

Multiple gedit - which to write.
Everything in OS is file manipulation
→ OS does this in sync block (semaphore)

Sleep - Trial & Error for Synchronization.
CSP - Critical ~~Synchronization~~ Section Problem
~~Proof of Correctness~~ :

CSP of a Code or P_i - (Eg: counter++)
Generally shared variables.

```
do {
    "CS"
} while (true);
```

4 Structures : Entry Section, Critical Section, Exit Section and Remainder Section.

Critical Section : Manipulating shared resources.

(i) Mutual Exclusion : If P_i is in CS, then no other P_j can enter it's CS.

If $\exists P_i \rightarrow (CS)$ then $\nexists P_j (CS)$.

(ii) Progress : P_i should not acquire lock indefinitely.
No P in CS, n process waiting for CS
Don't allow same P to access CS
Favour those who haven't moved in the Remainder section.

(iii). Bounded Wait : Limit to P_i to move into CS after exceeding the bound. Remove limit if no process waiting for it's CS ~~which~~ or ~~haven't moved~~ wait for other P_j who

haven't moved into CS for the bound.

$P_{i=0}$ $P_{j=1}$

//*flag (= desire) is an array to indicate its interest to move into CS
and turn = i indicates i^{th} process can move in CS
*/.

```
do { {flag[i] = true;
     turn = j ; } //Entry Cond^n
while (flag[j] && turn==j);
  counter++ ; // Critical Sec^n.
flag[i] = false; // Exit Sec^n
```

```
do{ {flag[j] = true;
     turn = i ; } //Entry Sec^n
while (flag[i] && turn==i);
  counter -- ;
flag[j] = false ;
```

Proof of Correctness :

① Prove MUTEX : Contradictory Proof
i.e There are 2 process in CS.
i.e Both P_i & P_j ~~are~~ have flag = true
flag[i] = true & flag[j] = true

P_i can move in CS when while is false
i.e. flag[j] = false or turn = i is followed.

But flag[j] is true initially but P_i is in CS that means flag[j] is false. Contradiction.

Since turn cannot be both i & j, it is set by either P_i & or P_j. P_i in CS means turn = j & ~~tu~~ flag[j] is true hence while is true So while cannot be false. Hence P_i cannot be in CS.

Date
Page

(iii) Prove Progress : contradictory Proof ;

P_i is not progressing $\cong$ P_i not going to CS
$\cong$ P_i in while $\cong$ $f[j]$ == true && turn == j

For P_i to be stuck in while : cases are as follows :

(i) P_j has flag[j] = false
Then P_i can progress

(ii) P_j has flag[j] = true & Nobody progresses.
P_i is not progressing & P_j is not progressing $\cong$ P_j is in while
$\cong$ flag[i] = true & turn = i

Thus P_i can move forward since it's while fails and it moves into CS. P_i progresses.

Also for (iii) Bounded Wait.

(iii). P_j has flag[j] = true & P_j progresses

P_i does not enter since turn = j ~~so~~ as P_j progresses. When P_j reaches exit section flag[j] = false and hence P_i while fails
Thus P_i progresses after P_j reaches Exit section.

Thus every Process accesses CS and wait is of amount : 1 turn. of do while.

* N Process Setup :

Atomic { test_set - lock setting function.
↓
compare_swap - Upgraded version of test_set

Helper Function - Predicate Functions - Utility Function
Eg: isStackFull(), isStackEmpty(), etc.

```
bool test_set (bool *target)
{ bool temp = * target;
  ~~set~~ *target = TRUE;
  return temp;
}.

Pi : do {
    while (test_set(&lock));
        "CS"                          // lv = lock value
        lock = false                  // ev = Expected lock value
    }. while (True)                   // nv = new lock value.

int comp-swap (int *lv, int ev, int nv)
{ int temp = * lv;
  if (* lv == ev)
      *lv = nv;
  return temp;
}

Pi : do { while (comp-swap(&lock, 0, 1) != 0);
        "CS"
        lock = ~~True~~ 0;
    } while (True);
```

↳ solutions to CSP ; Much better than Peterson's Algo
↳ Dekker's, Dijikstra's, Eugene's, Baker's

* Semaphores :
  Required for greater than one Threads
  To avoid Race Around and thereby inconsistent state.

All threads Reading : No conflict.
Any thread Writing : conflict → Block read till write.

a.k.a Reader - Writer's Problem.
Reader → Shared, Writer's → Exclusive

Block readers till writer releases write lock.
read_lock and write_lock.

Binary lock can't allow shared read.
↳ Resource available or not available

↑ Justification for Semaphores
Resources > Threads - Simultaneous Reads
Resources < Threads - Exclusive (RR) Access.

→ linux Support for Semaphores.
sem_init = 10
sem_wait ; sem_post (signal - No more lock required).
sem_wait → Access, sem_post → Release (Communicate)

Semaphore = 1 then Binary lock.
Multiple copies more powerful than Binlock.

Example : Not the exact syntax.
semaphore S = 10. // worst case 10 threads can ac
// 11th thread should wait.

wait (s)	signal (s)
while (s <= 0) ;	
s--;	s++;

Signal should always be followed by a wait.
Otherwise signal will increment s > 10

Semaphore variable is an atomic variable.
Inside wait block, an instruction is atomic.

ctrl - Alt - Backspace - Kill GUI.

Algorithmic Usage : Semaphores = S, Q = 1.

t_1	t_2
wait(S)	wait(Q)
wait(Q)	wait(S)
signal(S)	signal(S)
signal(Q)	signal(Q)

→ Deadlock : Order of release should be order of wait calls.

Time slice = 1 instruction.

For Time slice = 2, Normal Progress.

If Semphore = 2 & Threads = 3 & Deadlock Order need not be present, then deadlock will happen.

Best case to break deadlock = Reinitialize.

If deadlock on Q : t_2 last instruction pushed in deadlock whereas $t_3, \dots, t_{10}$ are just waiting for lock on Q in the start of code, then don't kill t_2, but kill others.

Linux kernel - Doesn't break deadlock. Waits for user. Computationally expensive to identify which process to kill

Better to Prevent / Avoid Deadlock.

Not Practical Deadlock Detection - Get into deadlock & understand system is in deadlock & do recovery - Difficult.

Deadlock Prevention - From current state, if new requests are handled then should kernel allow to handle based on next state is safe state or not

Safe state = No deadlock

If handled, will system be in safe state.

Deadlock Avoidance - For all possible combinations of execution of request, find if there is

safe state or not and if safe state is there - proceed.

Readers - writers Problem :

shared - exclusive :- shared read, exclusive write

n Readers / Writer - Simultaneous read, exclusive write

Writelock - Implement using semaphore

Similar to diner's philosophy but there multiple resources can be accessed.

// mutex will be used to synchronize betn readers and writers.

```
<semaphore.h>
sem_t writelock, mutex;
int data = 0, rcount = 0; // writer increments data.
```

/* What should semaphores be initialized with?
There is pthread_mutex_lock is a binary lock and thus no ~~synchronization~~ initialization required.
Since only 1 writer allowed; Writelock = 1.
Here, we take mutex = 1 since sync only 2 process (read or write) */.

```
main()
{ sem_init(&writelock, 0, 1);
  sem_init(&mutex, 0, 1);
```

variable — always 0 or NULL — init value

// Data of interest to pass to thread = Thread number.

```
pthread_t rtid[5], wtid[5];
pthread_attr_t attr; // Not used. Skip it.
```

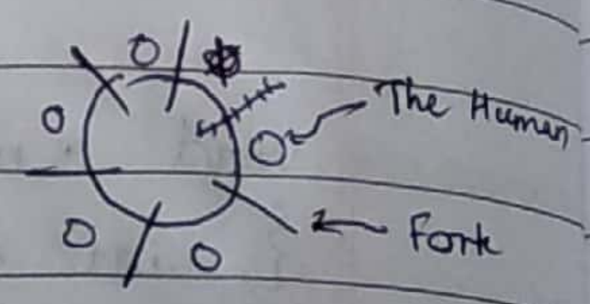
```
for (int i=0; i ≤ 4; i++)
{ pthread_create (&wtid[i], NULL, writer (void*)&i);
  pthread_create (&rtid[i], NULL, reader (void*)&i);
}
for (int i=0; i < 5, i++)
{ pthread_join (&wtid[i]);
  pthread_join (&rtid[i]);
}
}.

void *reader (void *arg)
{ int f = (int) arg;
  sem_wait (&mutex);
  rcount ++;
  if (rcount == 1)  // Active Reader
   sem_wait (&writelock)  // Intervening write avoided
   printf ("Thread : %d ; Data : %d ", f, data);
   sleep(1)  // Simulate Context switching.
   sem_post (&mutex);
   sem_wait (&mutex);
   rcount = rcount - 1
  if(rcount == 0).
   sem_post (&writeblock);
   sem_post (&mutex);
}

void *writer (void *arg)
{ int f = (int) arg;
  sem_wait (&writeblock);
   data ++;
  printf ("Thread : %d , Data : %d", f, data);
  sem_post (&writeblock);
}.
```

Writer block code is easy in this problem but generally it is way more complex.
In Dining-Philosopher, there are n forks and n diners, & each diner requires 2 forks. Here there are conditions that the forks should be near diner, not busy and others should not starve.

* Dining Philosophy: Semaphores.
(can be done via MUTEX as well)




```
// Generally N is 5 for this problem
# define N 5
# define Thinking 0
# define Hungry 1          #define left (Ph_num + 4) % N
# define Eating 2          #define Right (ph_num + 1) % N

// Hungry to Eating ≅ Acquire lock on fork
// Eating to Thinking ≅ Release lock on fork.
// Left & Right defined by (PhNo + 4) % N & (Ph No + 1) % N
sem_t muter, s[5]; // S = Spoon.
int phil_num [N] = {0, 1, 2, 3, 4}.
int state [N]; // = {0, 0, 0, 0, 0}.

void *phil (void *num)
void takefork (int)
void putfork (int)
void test (int)

int main() {
  pthread_t tid[N];
  sem_init (&muter, 0, 1);
  for (int i=0; i<N; i++)
      sem_init (&s[i], 0, 0);
```

```
for (int i=0; i<N; i++)
{ pthread-create (&tid[0], NULL, phil, &phil-num[i]);
   printf ("Philosopher : %d, is Thinking", i+1);
}
for (int i=0; i<N; i++)
   pthread-join (tid[i], NULL);
}.
```

```
void * phil (void * num)
{ while(1) {
   int *i = num;
   takefork (*i);
   sleep (2);
   putfork (*i);
   sleep (2);
}}
```

```
void takefork (int ph-num)
{ sem-wait (&mutex);
   state[ph-num] = Hungry;
   printf ("Philosopher : %d is
                Hungry", ph-num+1);
   //sleep(1);
   test (ph-num);
   sem-post (& mutex);
}sem-wait (&s[ph-num]); }.
```

```
void test (int ph-num)
{if (state[ph-num] == Hungry
      && state[Left] != EATING
      && state [Right] != EATING)
{ state[ph-num] = EATING;
   printf ("Philosopher : %d is
      Eating", ph-num+1);
   printf ("Spoons : %d %d",
      left+1, ph-num+1);
   sleep (2);
   sem-post (&s[ph-num]);
}
```

```
void putfork (int ph-num){
   sem-wait (& mutex);
   state [ph-num] = Thinking;
   printf ("%d %d Fork kept
      back on Table", left+1,
      ph-num+1);
   printf ("By Philosopher : %d",
      ph-num+1);
   test (Left); test (Right); //Progress
   sem-post (& mutex);
}.
```

* MUTEX based Sync : pthread-mutex_t (Binary lock)
  ↳ Just like mutex var in Semaphores
  Inversion - Inheritance Issue
  ↓
  MARS Pathfinder.

First thread to lock - sets muten appropriately
No need to initialise with a value

DP w/o Sem :

```
{   pthread_muten_t   spoons[5];
    pthread_t phil~~tid~~[5]
    ~~pthread_muten~~
    for (int i=0; i<5; i++)
      pthread_muten_init ( &spoons[i], NULL); //sem_init.
    for (int i=0; i<5; i++)
        pthread_create ( phil[i], NULL, func, ~~int~~ (void*) i);
    for (int i=0; i<5; i++)
        pthread_join (phil[i], NULL);
}


void * func ( ~~int~~ void * para)
{   int n = (int*) para;
    printf ("%d Philosopher is Thinking ", n);
    pthread_muten_lock (& spoons[n]);
    pthread_muten_lock (& spoons[(n+1)%5]); // n+1 %5.
    printf ("%d Philosopher is Eating ", n);
    pthread_muten_unlock ( & spoons[n]);
    pthread_muten_unlock (& spoons[(n+1) %5]);
    sleep (3);
}.
```

Go through n process Bounded ~~to~~ wait code ~~in~~ given by Dekkers, or other algorithms. - Extension of Peterson's Algo for 2 process setup.

* Prod - Cons :-

```
pthread_muten_t   muten;
pthread_cond_t   NotFull, NotEmpty;   // muten variable supporting wait & signal
/* condition_t requires an overall muten var to
    operate on */.
```

```
char items[BS];
int index = 0; // similar to (counter) data.

pthread_mutex_init (&mutex, NULL); //
        OR
mutex = PTHREAD_MUTEX_I..R; // Either function
NotFull = PTHREAD_COND_INITIALIZER; // or initializer.
NotEmpty = PTHREAD_COND_INITIALIZER;

void *consumer (void *para)
{ while (1)
  { pthread_mutex_lock (&mutex);
    if (index == -1);
      pthread_cond_wait (&NotEmpty, &mutex);
    printf ("Consumer No: %d\n", index) //Decrement index post
    pthread_cond_signal (&NotFull);
    pthread_mutex_unlock (&mutex);
}}

void *producer (void *para)
{ while (1)
  { pthread_mutex_lock (&mutex);
    if (index == BS)
    pthread_cond_wait (&NotFull, &mutex);
    items[index++] = '@';
    printf ("Producer No: %d\n", index);
    pthread_cond_signal (&NotEmpty);
    pthread_mutex_unlock (&mutex);
}}.
```

* Priority Inversion - Solution: Priority Inheritance

Eg: Rover - Mars Pathfinder.

Sync has H priority

T_1 T_2 T_3 : Threads
low(L) (m)Medium High(H) : Priority
Meterological Comm. Info Bus : System

L is executing & H comes in. All threads using mutex-locking. Since L has to release the lock, H has to wait till then. So locking by H will fail till T_1 unlocks. Now, as H is waiting, M enters. L's lock release and M entering is simultaneous. Since M is higher priority than L, it can gain the lock and H has to wait again.

Priority keeps changing as Processes arrive. Only Binary comparison is done for Priority checking

As M > L, M gets the control.

Priority Inheritance: Make L work as H priority (T_1 working with H priority) then even if M(T_2) comes in during T_3's wait, it has lower priority than T_1 and thus after releasing the lock, T_3 becomes H and runs.

If all higher priority pre-empts lower ones, synchronization goes for a toss.

T_1 inherits priority of T_3. Now T_2 won't get scheduled. Thus T_3 runs.
Low priority Threads inherits priority of a higher priority thread - Priority Inheritance
Problems faced by Mars Pathfinder.
T_1 - Gather weather data ; T_2 - Send data back to Earth

Wind River Systems - VX Works - Check History
(WRS)

T_3 - Enable communication between different systems.

Pathfinder OS: company - Wind River Systems.
OS - VX Works (EOS - Embedded OS)
EDF - Earliest Deadline First
In SJF, Burst time should be calculated but too computationaly heavy for EOS, ∴ Use EDF : Schedule the one with closest to deadline
Good OS - Written in C.

The solution was Reset by the OS provider (WRS) and the system thus went through lot of results.
Read Schilberts for more problems like this.
Priority Inversion was published before OS development. But still the company didn't mention the authors in the conference while they were present.

Semaphore Examples:

① P_1 : ⓘ $C = B - 1$; ⓘⓘ $B = 2 * C$
P_2 : ⓘⓘⓘ $D = 2 * B$; ⓘⓥ $D = 1$.

How many different values of B for any order of execution of P_1 & P_2.

② Using 2 semaphores give the code (snippet) to ensure the following order of execution
$P_2 \rightarrow P_1 \rightarrow P_3$.

Wind River Systems (WRS) - VX Works - Check History

T_3 - Enable communication between different systems.

Pathfinder OS: company - Wind River Systems.

OS - VX Works (EOS - Embedded OS)

EDF - Earliest Deadline First

In SJF, Burst time should be calculated but too computationaly heavy for EOS, ∴ Use EDF : Schedule the one with closest to deadline.

Good OS - Written in c.

The solution was Reset by the OS provider (WRS) and the system thus went through lot of results.

Read Schilberts for more problems like this.

Priority Inversion was published before OS development. But still the company didn't mention the authors in the conference while they were present.

Semaphore Examples:

① P_1 : (I) $C = B - 1$; (II) $B = 2 * C$

P_2 : (III) $D = 2 * B$; (IV) $D = 1$.

How many different values of B for any order of execution of P_1 & P_2.

② Using 2 semaphores give the code (snippet) to ensure the following order of execution

$P_2 \rightarrow P_1 \rightarrow P_3$.

③

```
P                 Q
while(1)          while(1)
{ w               { y
  printf("0");      printf("1");
  printf("0");      printf("1");
  x                 z
}                 }.
```

If v = Signal and w = Wait what would be w, x, y, z to achieve "00110011"

	w	x	y	z	S	T
a)	P(S)	V(S)	P(T)	V(T)	1	1
b)	P(S)	V(T)	P(T)	V(S)	1	0
c)	P(S)	V(S)	P(S)	V(S)	1	X
d)	P(S)	V(S)	P(T)	V(T)	1	0

④ Above setup in ③ in which of the above option is the ~~op~~ output not like 01^n0 or 10^n1 where n = Odd.

	w	x	y	z	S	T
a)	P(S)	V(S)	P(T)	V(T)	1	1
b)	P(S)	V(T)	P(T)	V(S)	1	1
c)	P(S)	V(S)	P(S)	V(S)	1	X
d)	V(S)	V(T)	P(S)	V(T).	1	0

* Deadlocks:

2 Runner functions:

```
T1              T2.
lock(&m1)       lock(&m2)
lock(&m2)       lock(&m1)
unlock(&m2)     unlock(&m1)
unlock(&m1)     unlock(&m2)
```

Deadlock doesn't happen always. Eg: Interleaving is in steps of 2 → No deadlock.
Deadlock → Interleaving is steps of 1.

Properties of Deadlock - Deadlock will happen given the following properties.

Solution - (i) Detection (ii) Prevention (iii) Avoidance

Easy - Preferred for OS Implementation Difficult.

Properties (Characteristics) for Deadlock) :

(i) Mutual Exclusion : Atleast 1 non-shareable resource

Shared Resource ≠ Copies of Semaphore

Shareable Resource ⇒ Read only files.

Detection - Given a state, if resource request granted will it be in unsafe state.

Avoidance - All possible allocation of resources.

Prevention - Given Current allocation requirements, calculate all possible futuristic requirements of n copies of the resource (Banker's Algo).

(ii) Hold - wait : The process is holding some resources (R_i) and requested for (R_i+1) and is waiting for the $(R_i+1)^{th}$ resource.

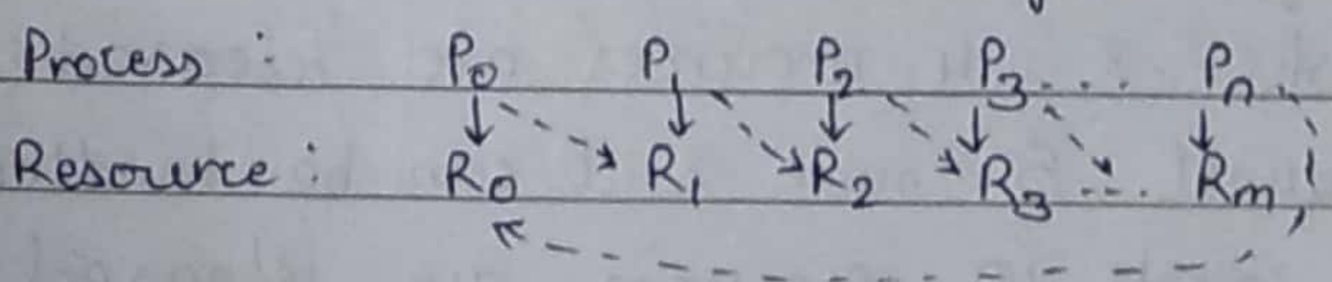
(iii) No PreEmption (In PreEmption - Sync falls apart i.e. No need to even lock)

Priority : Data Consistency > Efficiency by PreEmption.

Atleast 1 such resource should exist whose preemption can't be allowed.

Some process can't make other process lose its lock on a resource (preempt a resource).

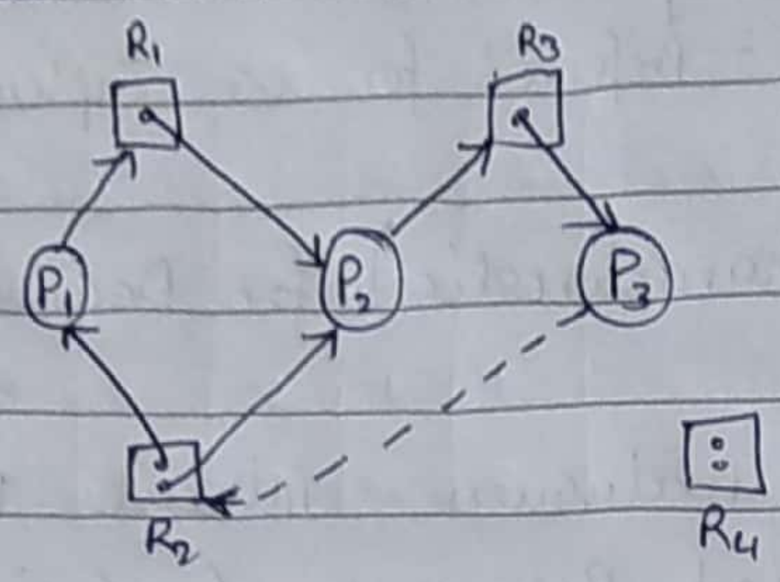
(iv) Circular Wait : A cyclic wait has to exist.



Date
Page

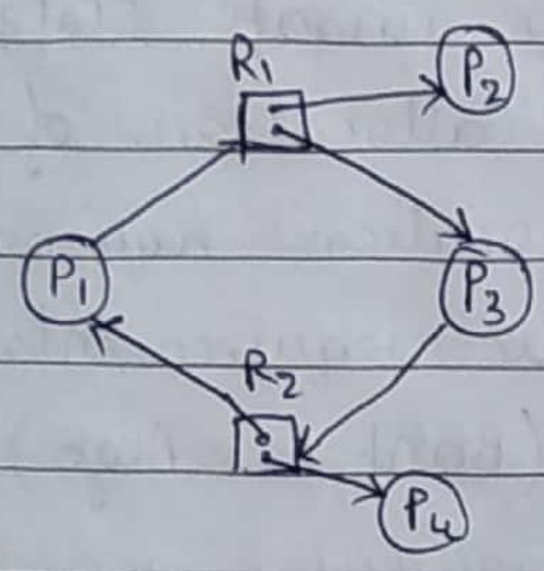
Q Deadlock Detection - RAG = Resource Allocation Graph.

Proved: System Not in Deadlock



• - Single instance
•• - >2 instances of Resource
O→ - Request
□→ Assignment of Resource

---→ Adding a request from P_3 for R_2 : System in Deadlock - 2 cycles - $P_1, R_1, P_2, R_3, P_3, R_2, P_1$ and P_2, R_3, P_3, R_2, P_2.



- Not a Deadlock, Since after end of P_4's use of R_2, P_3 can use R_2.

i.e. Cycles alone are not necessary for Deadlock in case of multiple resources.

If P_4 (and P_2) was a part of cycle then it will be Deadlock

Every process should satisfy hold & wait, only then cycle is necessary condition for Deadlock.

To:

Process	MaxAlloc	CurrentAlloc.
P_0	10	5
P_1	4	2
P_2	9	2.

R = 12. (Resource).
F = 3 (Free Resource).
Order of Execution : $P_1 \rightarrow P_0 \rightarrow P_2$.

P_1 can request 2 more, can it be handled - Yes. F=1.
Thus P_1 finishes & 4 resources are released : F = 5.
P_0 can request 5 more & it can be handled : F=0
P_1 finishes and 10 resources are released : F = 10
P_2 can request 7 more & it can be handled : F = 3
P_2 releases ~~and~~ 9 resources : F = 12.

POSet - Partially Ordered Set.
Req^m - Requirement

Thus Order of $P_1 - P_0 - P_2$ is pushes system in a <u>safe state</u>.

Example of Unsafe Allocation:
At T_0, the given state is followed
T_1 : $P_2 + = 1$. ; $F = 2$.

$(P_2 + 1) \longrightarrow P_1 \longrightarrow \boxed{P_0}$? → Needs 5 but have 4, Hence cannot be scheduled.
F: 2 0 4
$\hookrightarrow \boxed{P_2}$? → Needs 6 more but have 4. Hence cannot be scheduled.

This is because of the $P_2 + = 1$. (Unsafe state Trigger).

(ii) Avoidance:

Claim	R_1	R_2	R_3.
P_1	3	2	2
P_2	6	1	3
P_3	3	1	4
P_4	4	2	2

Allocation		
1	0	0
6	1	2
2	1	1
0	0	2.
Total: 9	2	5

Resource Vector

R_1	R_2	R_3
9	3	6

Free Vector

R_1	R_2	R_3.
0	1	1

① Can P_1 be scheduled initially?

Future Req^m $\boxed{2 \mid 2 \mid 2.}$ > Free ⇒ ~~Unsafe~~ (Not Unsafe since other ordering) Not Proceed able.

* [As there are other ordering available] it is not Unsafe since the request itself cannot be proceed.

Unsafe is make a request proceed & now you're nowhere to go.

② P_2 - Future Req^m $\boxed{0 \mid 0 \mid 1}$ < Free ⇒ Allowed.

P_2 - Runs to Completion. Frees its resources.

Claim			Alloc			Free		
3	2	2	1	0	0	6	2	3
0	0	0	0	0	0			
3	1	4	2	1	1			
4	2	2	0	0	2			

Since Free was [0 | 1 | 0] when P_2 proceeded and released [6 | 1 | 3] after completion.

$P_2 \rightarrow P_1$? F. Req^m [2 | 02 | 2] $\leq$ Free.

P_1 runs till completion.

Free After completion P_1 : [78 | 2 | 34]
Free Before Completion P_1 : [4 | 0 | 1]

0	0	0	0	0	0
0	0	0	0	0	0
3	1	4	2	1	1
4	2	2	0	0	2

$P_2 \rightarrow P_1 \rightarrow P_3$? F Req^m [1 | 0 | 3] $\leq$ Free - Granted

Free when P_3 running - [6 | 2 | 0]
Free when P_3 completes - [9 | 3 | 4]

$P_2 \rightarrow P_1 \rightarrow P_3 \rightarrow P_4$? F Req^m [4 | 2 | 0] < Free - Granted

Free when P_4 running - [5 | 1 | 4]
Free after P_4 completes - [9 | 3 | 6] → All done.

Example of Unsafe:

To :

Claim		
3	2	2
6	1	3
3	1	4
4	2	2

Alloc		
1	0	0
5	1	1
2	1	1
0	0	2
8	3	4

Free = | 1 | 1 | 2 |

P_2 ? T_1 : FReq^m : | 1 | 0 | 1 | $\leq$ Free - Granted.

Free during P_2's run | 0 | 1 | 0 |

Free after P_2's completion : | 6 | 1 | 3 |

T_1 :

P_1 ? New Request : | 1 | 0 | 1 | (Interim)

(Not Running till completion) $\leq$ Free - Granted

Thus need not allocate all resources in the claim

claim is the max but anything less than max can also be allocated.

Free : | 0 | 1 | 1 |

$P_{1(101)} \rightarrow P_3$? FReq^m | 1 | 0 | 3 | > Free (cannot be Granted)

Since no R_1 left & 1 is the minimum request amount - No interim request can be made by P_3.

Thus P_3 cannot be granted locks.

Same with P_4 and P_2. So only P_7 can complete now. but not enough resource

Thus, we get into Unsafe state.

($P_{1(101)} \rightarrow$ Anything) cannot be a safe order.

Unsafe Order $\rightarrow$ Unsafe state

Date
Page

* Memory Management :
(Stdlib functions).

malloc : Dynamic memory allocation of given size.
Not initialized.

calloc : Initialized Dynamic mem alloc.

free : Deallocate allocated memory.

realloc : Allocate the given ptr with new size.
(realloc (a, 20))
Initially a had 10 bytes for ptr a
After reallocc, if 10 more bytes are available
a is size 20 or new ptr is returned
with size 20 (uninitialized memory).

—x—

brk : Break Point of Program Data Segment
If the data segment ends at 4000
then brk (4004) allocates 4 more bytes.

sbrk : Increment Break Point of PDS.
sbrk(0) gives current break point.
sbrk(2) increments by 2 bytes.
sbrk of -ve number deallocates.

```
void * b = sbrk(0);
int *p = (int *)b;
*p = 1; // fails.
```

```
brk(p+2); // Increase by 8 bytes
*p = 1; // Success.
// Increase 8 since p is (int)
```

Depreciated ↑ (Old) Implementation of malloc, calloc).

—x—

mmap & munmap

mmap - Map file into memory. (virtual).
(Used to implement malloc, calloc - Mem).
prot - protection (r, w, x, etc).
flags - MAP SHARED - Shared Memory
MAP PRIVATE - Only process which created the mem space can access.

munmap - Opposite of mmap.

If fd is not given then it might allocate variables

munmap doesn't happen on fclose but on termination automatically.

mremap - Similar to realloc : Expand or shrink given mmap (can move the entire mem space giving new address).
flag - MREMAP_FIXED. (cancels the effect).
(Generally moves mem to new address).
default - MREMAP_MAYMOVE.

If mmap protection is violated, serve them SIGSEGV.
mprotect - Change access protections for the calling processes memory pages.
(Shared mem changed for all process).

PROT_NONE, PROT_READ bitwise or PROT_WRITE.
PROT_EXEC.

Beware : A lot of jumps in topics for mem Mgmt.

madvise - Give advice about memory space to kernel.
Eg : MADV_NORMAL , MADV_SEQUENTIAL,
can be used for linear search.

mlock - Keep the given memory in RAM (Not in Virtual Memory)
Given address & length, that memory will be guaranteed to reside in RAM.

mlock2 - same as mlock but have flags.
State of pages depends on value of flag.

mlockall & munlockall
Same as mlock but all pages related to process are in RAM.
Child doesn't inherit the property on fork.

mincore - Given address, & length with a char vector, tells if the space of mem is in RAM or not (m - in core or not)

sbrk cannot allocate mem somewhere else.
Just increment / decrement the breakpoint.

Memory Management : Segmentation, Paging.

Memory : { array of bytes }
↳ Allocated for Programs & Data
Series of instructions
(Exists) → RAM (primary memory) → physical memory.
↳ There was a time when RAM was of 4MB.

μP - microProcessor.

i.e RAM <<< (Required for multiprocessing) Requirements

Worst case - Client, Server - Server runs users process i.e # of users ~~are~~ ~~no~~ is large. (no theoretical limit on # users)

OS only discusses about Virtual Mem Mgmt.
↓
DNE (Does not Exists).

Since RAM is limited, OS needs to flush out & bring in new data, thus disk mem is used. Virtual Memory consists of disk mem & software memory. (Swap files / Page files).

Random Access is used (like a book, refer to Table of Contents or Word used in page at the end to jump to that page).

→ Logical Memory (Virtual Memory).

With respect to Mem, only load & store is executed (eg: Fetch decode execute w.r.t CPU) In the end, everything is executed from RAM.
↓
Von Neumann architecture.

Addressing Modes in μP - Modes that help addresses the instructions.

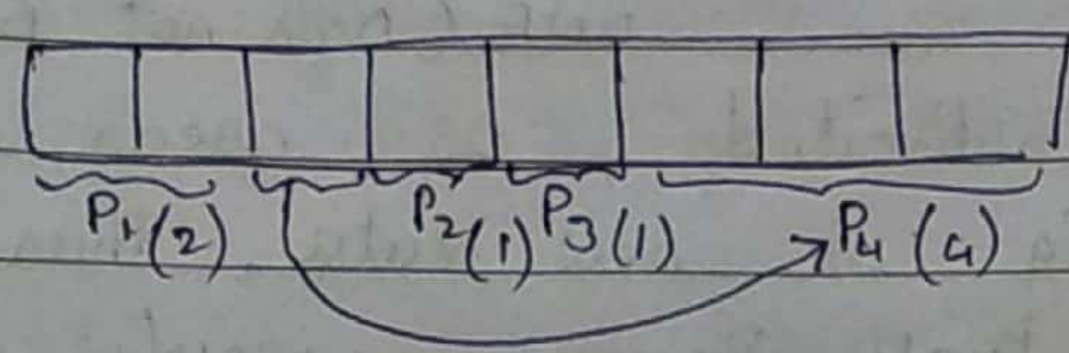
0 to 4L. ← Users View - large chunk of Contigous Data
continuous mem.
Programmers View - Segments (Stack, Data, Code, etc).
Code & Data has to be loaded (This is where Addressing Modes comes in).
Actual View - Limited Data Area (RAM). - Blocks
Not Contigous Segments.

Symbolic Address $\xrightarrow{\text{Compiler}}$ Relocatable Address

// Page Table - Stored in RAM

Linked list - Better than Array since if new node is to be added / removed, it is easy in linked list.

Contigous - Linear - Sequential -



This access of mem by P_4 is the main focus of mem mgmt

Page Table - What blocks allocated to data of a process is where.

Mem Mgmt - Mapping of logical mem onto the Physical Mem.
(logical mem is very very large).

The whole OS cannot load on RAM.
If loaded then no Application Process will have space (cobalt Blue OS)
Device Drivers not loaded - DLL (Dynamic link library)

Image of OS has gaps which is replaced by the loaded drivers.

*. Deadlock

(iii) Prevention:

Construct a graph on requirement & available resources. If cycle on single instance of resource then deadlock.

* The kernel is not NPE, Content switch happens but the resource's lock is NPE.

Refer Banker's (Avoidance) from Sterling's Book.

Characteristics for Deadlock: (i) Muter (Shareable resource)
(ii) Hold & wait (No PreEmption Possible) → *.
(iii) No PreEmption (iv). Circular Wait.

In Avoidance, there is an initial state & the max allocation matrix, etc.
Start from initial safe state, is there an ordering of process where all process move till completion. - Banker's is more futuristic in nature. Costly as there are 100's of process and 10's of resources. Deadlock Recovery (Handling) is Restart. or Kill the process which cause deadlock (Do Not Kill High Priority, Nearing Completion or Having allocated resources more than others).

Prevention - Violate 1 of the 4 characteristics. then deadlock always prevented.

Can't violate Muter (Resources can't be accessed simultaneously like printer).

No hold-wait i.e. Not the case of P_1 has R_2 & waits for R_1. (Not waiting at all or not holding at all) Get all resources before starting then hold-wait violated, thereby no deadlock.
2 Options — No hold or Not wait.
No hold: Having 6 & Need 3 more - Release 6 since no holding.
No wait - Get all resources even those that won't be used. Similar to FCFS.
P requires R_1 & R_2 but P has to acquire all resources R_1 to R_n.

Both are inefficient - No wait is definitely useless.
↓ Even this does.
No hold causes starvation.

∴ Not violating Hold-Wait. (Valid but inefficient)

No PreEmption - Enable PreEmption - A process can make other process release its lock.
PreEmption — 2 Options [Wait & Die.
Wound & Wait.

Valid but causes starvation.
P_1 has X & needs Y. If received before threshold - Wait otherwise die.

P_1 has X & Needs Y held by P_2. Kill P_2 & take Y from it and P_2 waits again for Y.
May End up in starvation.

Not violating NPE.

No circular-Wait:- Map every resource to a natural number
$F(R_1) = 1$; $F(R_2) = 5$; $F(R_3) = 12$.
Process P_i holding R_i & wants R_j possible only if $F(R_j) > F(R_i)$.

P_1 holding R_1 can ask for R_2 but P_1 holding R_2 cannot ask for R_1 but can ask for R_3.
If P gets R_2 & asks for R_1 then it is killed.

a.k.a. Ordering Solution.

t_1 : $l(R_1)$ $l(R_2)$ $u(R_2)$ $u(R_1)$. Deadlock.
t_2 : $l(R_2)$ $l(R_1)$ $u(R_1)$ $u(R_2)$.

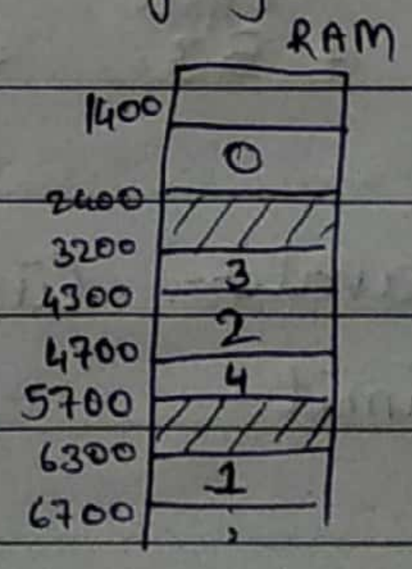
Kill t_2 by the given solution.
Thus cycle formation is avoided.
T_2 will proceed if it releases R_2.
(Assuming no depedency).
Here atleast T_1 will proceed rather than both waiting.

Mathematical Proof : $P_{i+1}(R_i) \rightarrow R_{i+1}$.
$F(R_i) < F(R_i+1)$
$F(R_0) < F(R_1) < F(R_2) < \dots < F(R_{n-1}) < F(R_0)$.
This causes deadlock since a process holding R_{n-1} can wait for R_0. i.e. There is cyclic wait. Hence it violates Ordering sol^n

Both cyclic wait & Ordering sol^n can't be true at the same time.

* Segmentation & Paging:

	limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700



User: Linear / Contigous allocation of mem.
Actual : Blocks of seperate length.

If linear allocation is done - End up fragmenting the RAM. (Unused - Holes in the memory) - Linear creates external fragments.

Solution : Segmented Allocation :

memory is byte addressable.

What were trying to do - Map logical to physic

Paging : 2D :- logical → Physical
↓
2D function

Every request is now of the form
< Seg No , Offset >

5 Segments - Stack, Heap, Global Variable,
Code ↓ and 1 more.
(Threads & Functions).

Privacy - User P can't access Kernel P data
Uses Base Register & Limit Register.

Add Offset to Base & if > limit
Segmentation Fault.

BR & LR - Segment Table (Part of PCB).

Request : (2, 53).
Base Offset
∴ 4300 + 53 = 4353 < 4300 + 400.
Base limit

Internal Comparator Circuit : Compare Offset
with Limit.

→ Complete Page Replacement (LRU, MFU) - comes
in GATE.

Fragmentation is still possible in segments
but in large ~~chun~~ chunks.
Logical Block - Page ; Physical Block - Frame.

Equisize Partitioning (EP) : 500 bytes - 5 x 100b blocks
5 Process & of 10 bytes : 90 bytes per block wasted (internal fragmentation).

6th Process of 120 bytes can't be allocated.

∴ EP causes large internal fragments.
- worst possible solution.

VariableSize Partitioning - Segmentation - No internal fragments but external fragmentation possible.

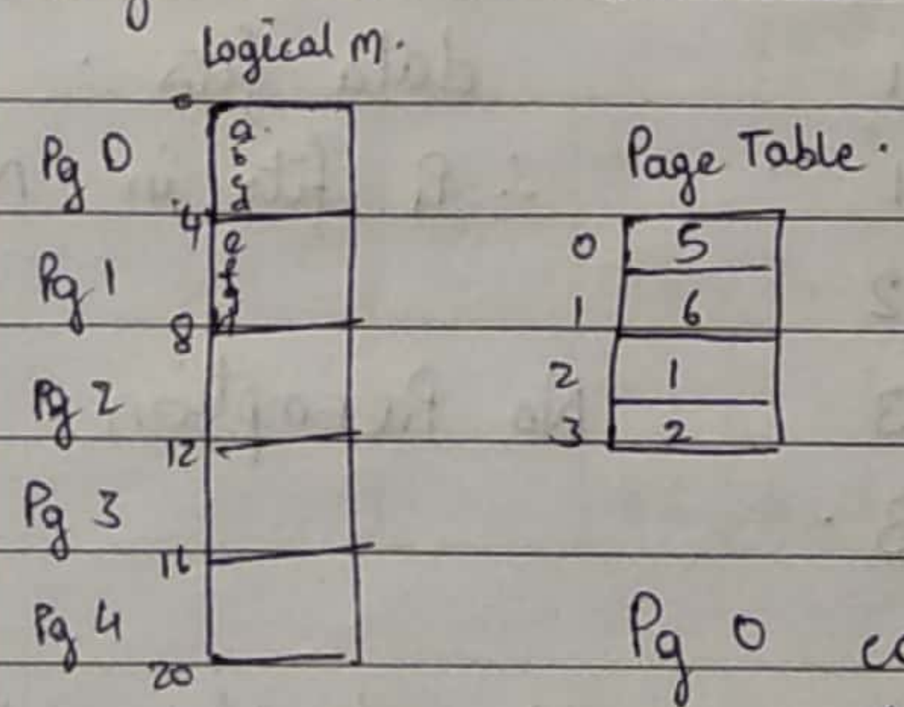
This will be avoided by Paging.

Mapping of LAS to PAS (2D function -(Seg#, offset)).

Linear < Segmentation < Paging (Improvement).

Paging : View as Pages (LM) ⟶ frame (PM).
2D function (Page#, Displacement).

Page Table - consists of Pages & its address.



In Page 0.
a has 0 displacement.
c has 2 displacement

Pg 0 can be accessed at mem Ad = 5x4 = 20 (20 to 24).

No. of bytes wasted is never more than or equal to page size.

Eg : LAS = 32 bit PS = 4 kB = 2^{12}

& PageTS = $\frac{2^{32}}{2^{12}}$ = 2^{20} = 1 MB. (Page Table Size)
(No of Pages)

Each Frame Size = 4 bytes.

No of Frames = 2^{20}.

∴ Total Physical RAM = $2^{20} \times 2^{2}$ = 2^{22} = 4 MB.

Always : No. of Frames = Page Table Size.

* Fitting Strategies :
First, Best, Worst : fits

Process No	Bytes Req^d	TAT
1	100	3
2	10	1
3	35	2
4	15	1
5	23	2
6	6	1
7	25	1
8	55	2
9	88	3
10	100	3.

MPart.

50	200	70	115	15	byte
m_1	m_2	m_3	m_4	m_5	

First fit : Get the block where the data fits.
∴ P_1 fits in M_2.

No Preemption.

Best Fit - "Somewhat" complement of 1'st Fit
Minimize the wasted mem.

		50	200	70	115	15.
1st Fit	T_0:	$P_2(0)$	$P_1(2)$	$P_3(1)$	$P_4(0)$	$P_6(0)$.
	T_1:	$P_5(1)$	$P_1(1)$	$P_3(0)$.	$P_7(0)$	
	T_2:	$P_5(0)$	$P_1(0)$	$P_8(1)$	$P_9(2)$	
	T_3:		$P_{10}(2)$	$P_8(0)$	$P_9(1)$.	
	T_4:		$P_{10}(1)$		$P_9(0)$.	
	T_5:		$P_{10}(0)$			

→ This is possible because all P arrive at same time.

Goal : Minimize holes

T_0:	$P_3(1)$	$P_5(1)$	$P_4(0)$	$P_1(2)$	$P_2(0)$.
T_1:	$P_3(0)$	$P_5(0)$	$P_7(0)$	$P_1(1)$	$P_6(0)$
T_2:		$P_9(2)$	$P_8(1)$	$P_1(0)$	
T_3:		$P_9(1)$	$P_8(0)$	$P_{10}(2)$	
T_4:		$P_9(0)$	P_8	$P_{10}(1)$	
T_5:				$P_{10}(0)$	

Worst fit is used for coalescing (merging holes)

Worst fit : Maximize hole size. Put P_8 where hole size is maximum in 200.

1st fit - N blocks - 50% wasted

* Replacement Algos:

(i) FIFO (ii) LRU (iii). OPT.

x - Page Fault.
✓ - Available

Page Fault.

Eg: Request : 0, 2, 1, 6, 4, 0, 1, 0, 1, 0, 3, 1, 2, 1.
(x x x x x x ✓ ✓ ✓ ✓ x ✓ x ✓)

Frames : 4.

	FIFO.	LRU.	OPT.
→ 0	4, 4, 4, 2	4, 4, 4, 2	0, 0
→ 2	2, 0, 0, 0	2, 0, 0, 0	2, 2
→ 1	1, 1, 1, 1	1, 1, 1, 1	1, 1
→ 6	6, 6, 3, 3	6, 6, 3, 3.	4, 3.

Page fault : Page not available in memory.
challenge - Replace which page.
LRU - Least Recently Used (Based on Past Data)
OPT - Optimal (Based on Preceeding & Succeeding Request).

Scanned by CamScanner

Optimal : Choose the one to replace which is the farthest from the current page.

Optimal is like Bankers (very difficult to implement).

Buddy Allocation, Paging System calls - could not be covered.

Scanned by CamScanner