43 down vote

In Program 1 `Hello world` gets printed just once, but when I remove \n and run it (Program 2), the output gets printed 8 times. Can someone please explain me the significance of \n here and how it affects the `fork()`?

Program 1

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
  printf("hello world...\n");
  fork();
  fork();
  fork();
}
```

Output 1:

```
hello world...
```

Program 2

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
  printf("hello world...");
  fork();
  fork();
  fork();
}
```

Output 2:

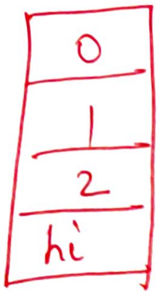
```
hello world... hello world...hello world...hello world...hello world...hello world...hello world...hello world...
```

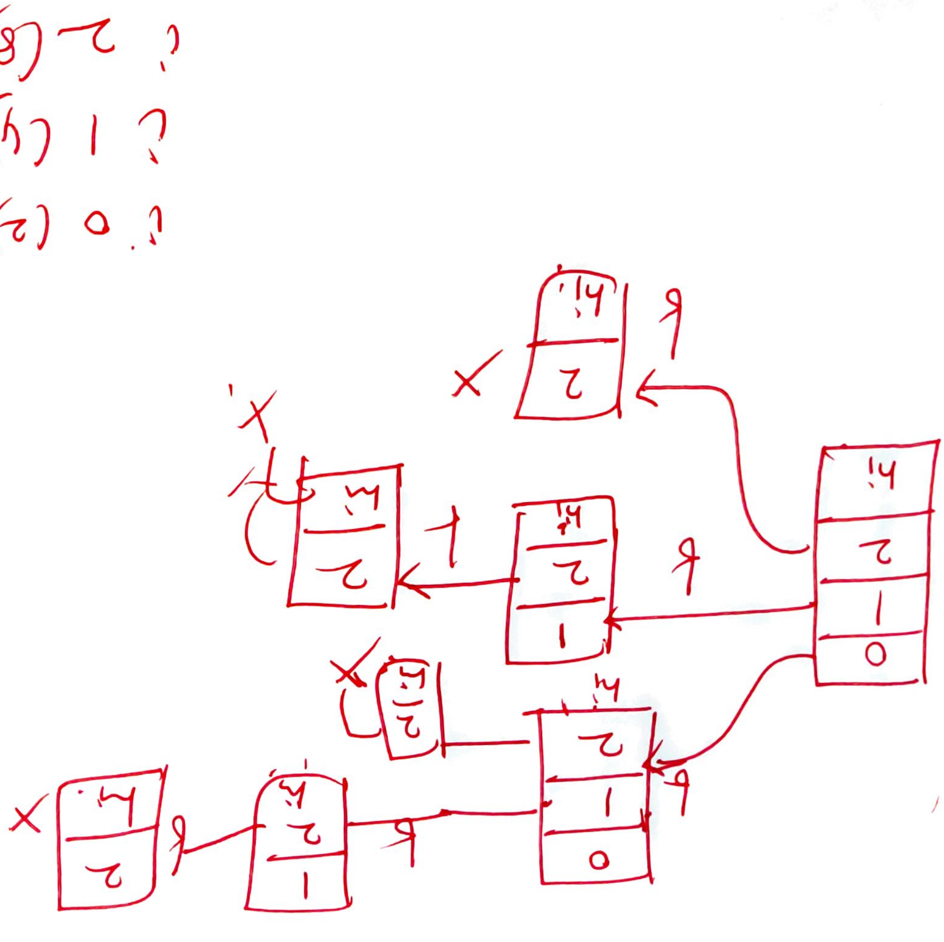
c | fork

When outputting to standard output using the C library's `printf()` function, the output is usually buffered. The buffer is not flushed until you output a newline, call `fflush(stdout)` or exit the program (not through calling `_exit()` though). The standard output stream is by default line-buffered in this way when it's connected to a TTY.

When you fork the process, the child processes inherits every part of the parent process, including the unflushed output buffer.

```
for (i=0; i<3; i++)
{
    fork(); p( %d %d %d, i, ppid, pid)
}
printf( i, ppid, pid)
```



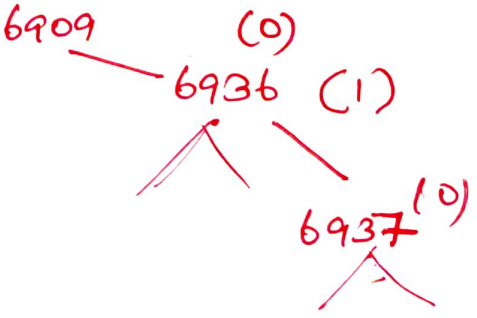


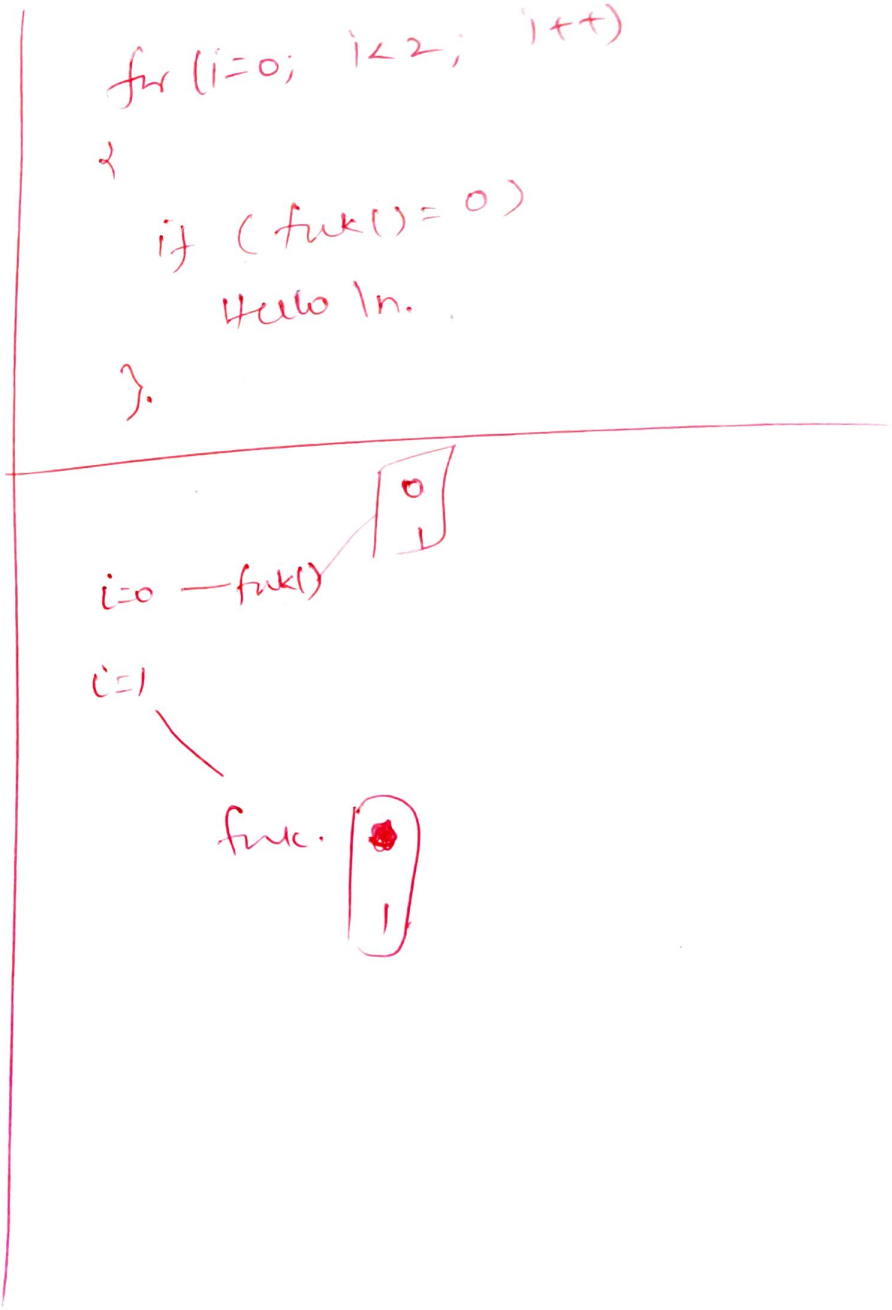
```
include <stdio.h>

void main()
{
  int i;

  for (i=0;i<3;i++)
  {
    fork();

    printf("[%d] [%d] i=%d\n", getppid(), getpid(), i);
  }
  printf("[%d] [%d] hi\n", getppid(), getpid());
}
```

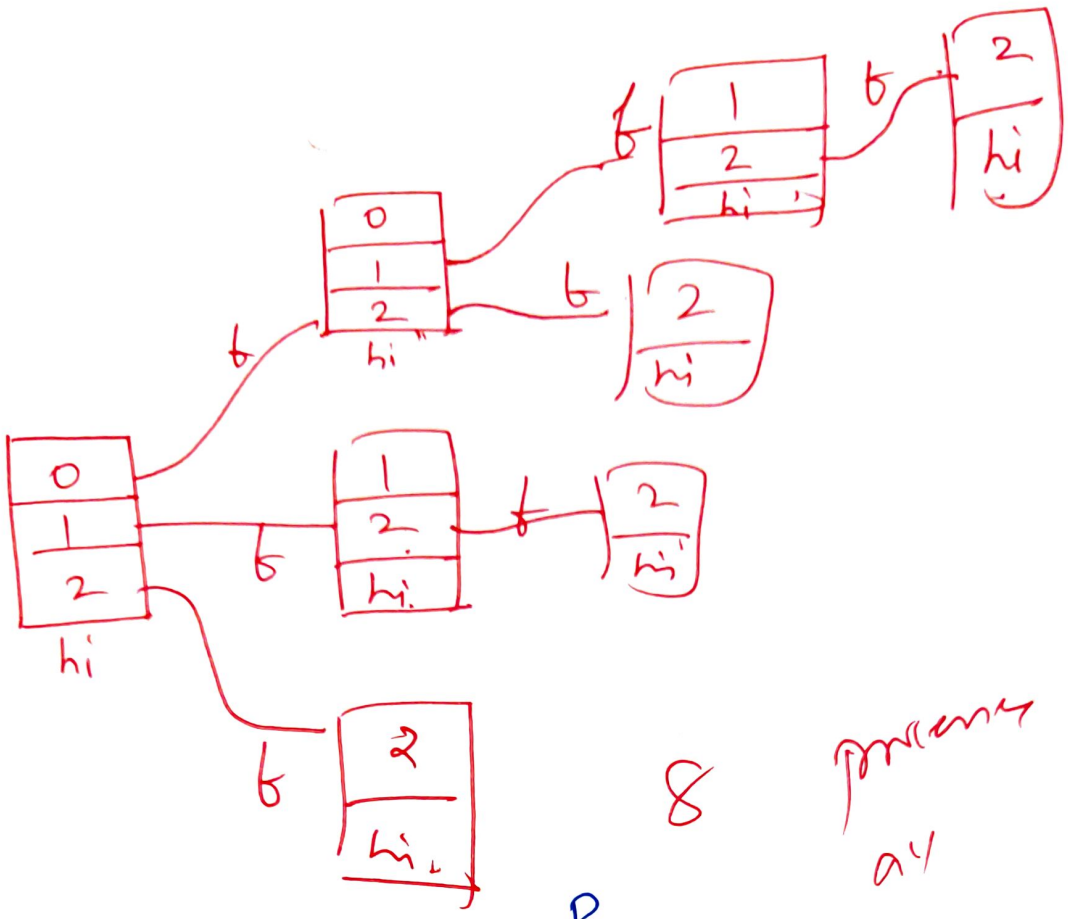
Here is the output:

```
[6909][6936] i=0
[6909][6936] i=1
[6936][6938] i=1
[6909][6936] i=2
[6909][6936] hi
[6936][6938] i=2
[6936][6938] hi
[6938][6940] i=2
[6938][6940] hi
[1][6937] i=0
[1][6939] i=2
[1][6939] hi
[1][6937] i=1
[6937][6941] i=1
[1][6937] i=2
[1][6937] hi
[6937][6941] i=2
[6937][6941] hi
[6937][6942] i=2
[6937][6942] hi
[1][6943] i=2
[1][6943] hi
```

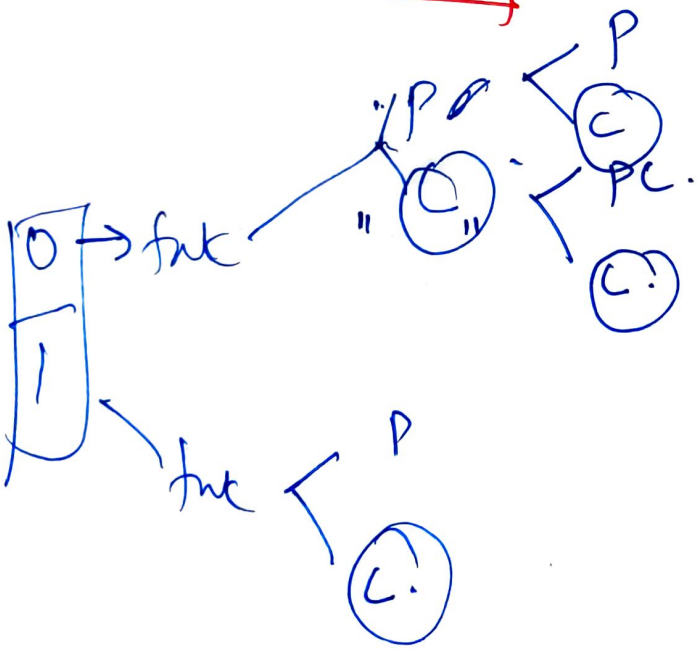
6909 6936 0

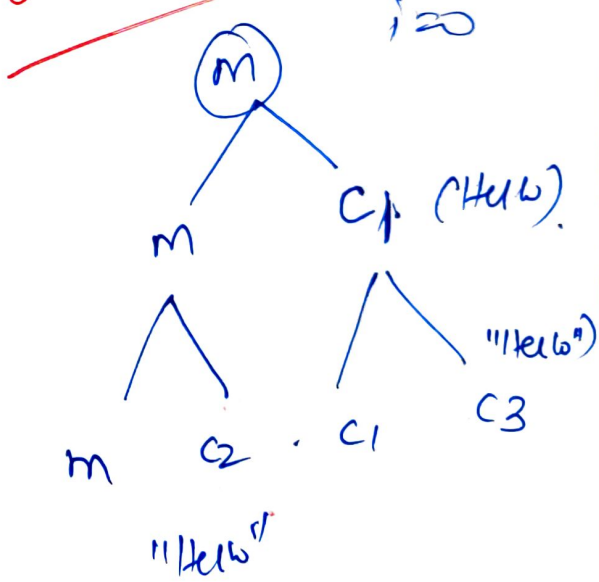


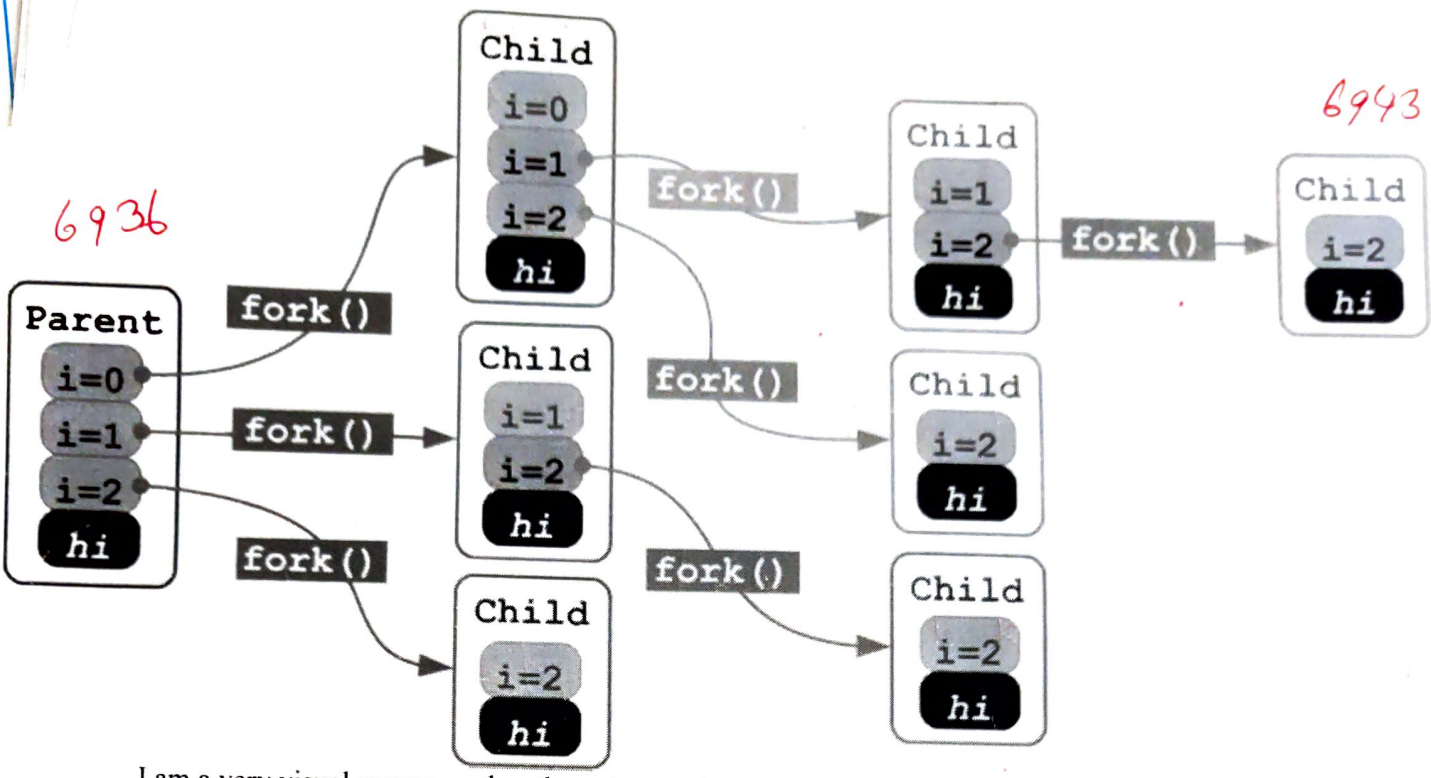



0
1
2
hi
b
1
2
hi
2
hi
0
1
2
hi
b
1
2
hi
2
hi
0
1
2
hi
b
2
hi
b
2
hi

&


P
P
C
C
C
0
1
fork
fork
P
C


m
C1 ("Hello")
m
"Hello"
C3
m
C2
C1
"Hello"



I am a very visual person, and so the only way for me to truly understand things is by diagramming. My instructor said there would be 8 *hi* statements. I wrote and ran the code, and indeed there were 8 *hi* statements. But I really didn't understand it. So I drew the following diagram:*Diagram updated to reflect comments :)*

Observations:

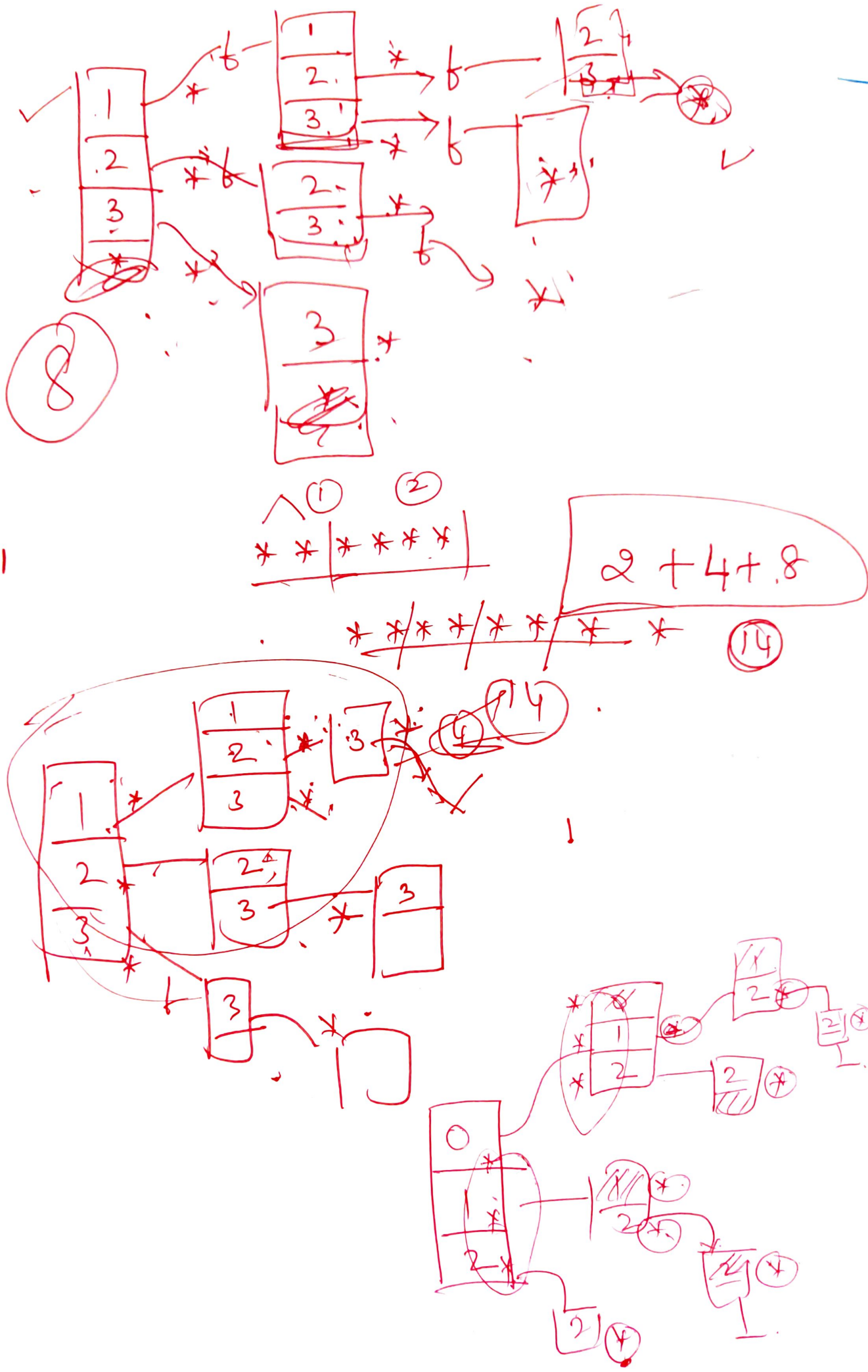
1. Parent process (main) must iterate the loop 3 times. Then printf is called
2. On each iteration of parent for-loop a fork() is called
3. After each fork() call, i is incremented, and so every child starts a for-loop from i before it is incremented
4. At the end of each for-loop, "hi" is printed

Here are my questions:

- *Is my diagram correct?*
- **Why are there *two* instances of `i=0` in the output?**
- What value of `i` is carried over to each child after the fork()? If the same value of `i` is carried over, then when does the "forking" stop?
- Is it always the case that `2^n - 1` would be a way to count the number of children that are forked? So, here `n=3`, which means `2^3 - 1 = 8 - 1 = 7`children, which is correct?

- Why not run it and print out `i`, the PID and parent PID after the `fork()`. It should be easy to track what's happening – Basic Nov 7 '14 at 3:14
- 2

@Basic, that's the first thing I did. I even used getpid() and getppid() and that's how/why I believe my diagram is correct. But I really want someone to verify this. – lucidgoldNov 7 '14 at 3:16


8
2 + 4 + 8
14

- That's a really nice diagram. Did you make it with dot/graphviz? – Steven Lu Nov 10 '14 at 17:08
- I used Microsoft Visio. But I now use LibreOffice Draw, very similar to Open Office Draw and both are open source project I believe, thus free! – lucidgold Nov 10 '14 at 17:15
- How did you disable the buffering by default ? On the gfg ide, the outputs are different without fflush - ide.geeksforgeeks.org/0TWiEZ, and with fflush - ide.geeksforgeeks.org/0JKaH5 – Udayraj Deshmukh Sep 18 '17 at 5:21

add a comment

3 Answers

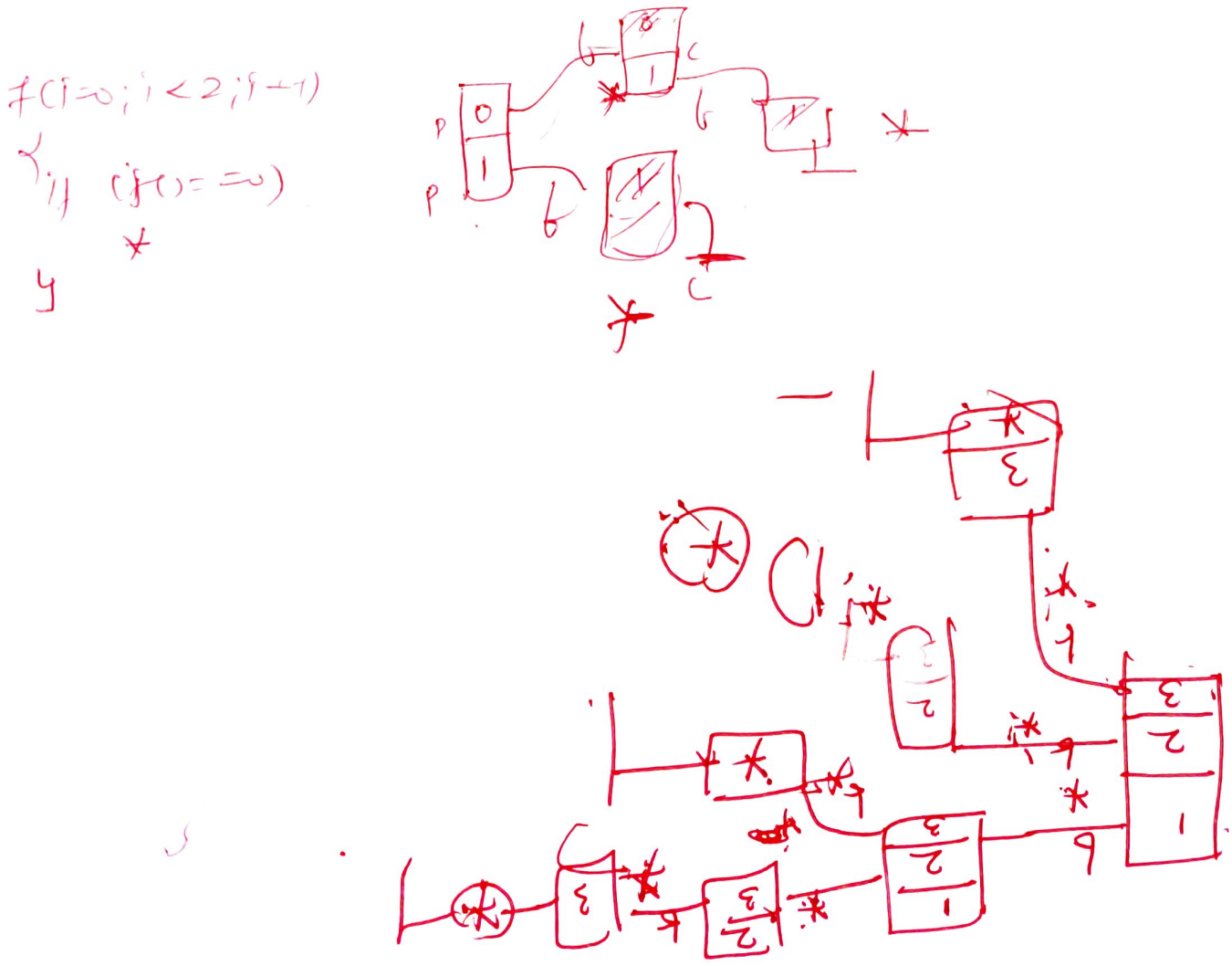
activeoldest votes

up vote30down vote

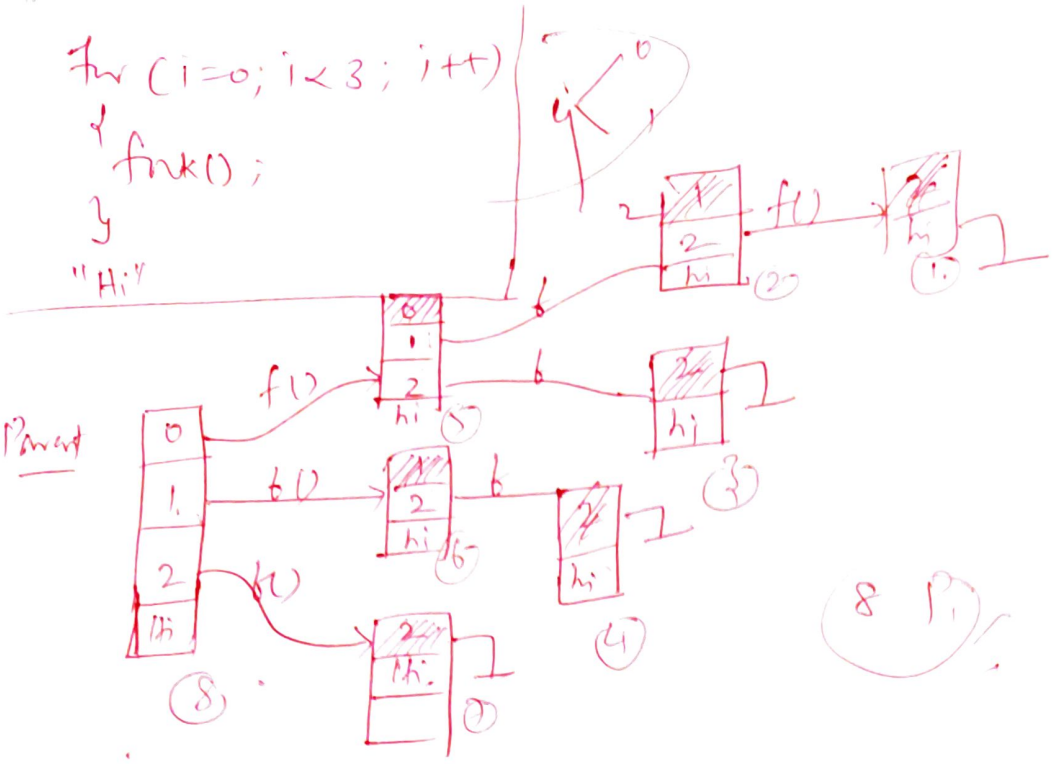
Here's how to understand it, starting at the `for` loop.

1. Loop starts in parent, `i == 0`
2. Parent `fork()`s, creating child 1.
3. You now have two processes. Both print `i=0`.
4. Loop restarts in both processes, now `i == 1`.
5. Parent and child 1 `fork()`, creating children 2 and 3.
6. You now have four processes. All four print `i=1`.
7. Loop restarts in all four processes, now `i == 2`.
8. Parent and children 1 through 3 all `fork()`, creating children 4 through 7.
9. You now have eight processes. All eight print `i=2`.
10. Loop restarts in all eight processes, now `i == 3`.
11. Loop terminates in all eight processes, as `i < 3` is no longer true.
12. All eight processes print `hi`.
13. All eight processes terminate.

So you get `0` printed two times, `1` printed four times, `2` printed 8 times, and `hi` printed 8 times.



```
for (i=0; i<3; i++)
{
  fork();
}
"Hi"
```



8 processes; $2^n - 1$

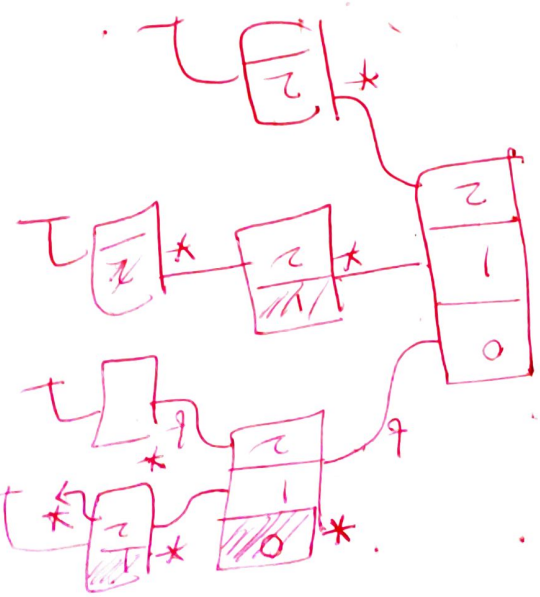
Assume printf (i) within for loop;

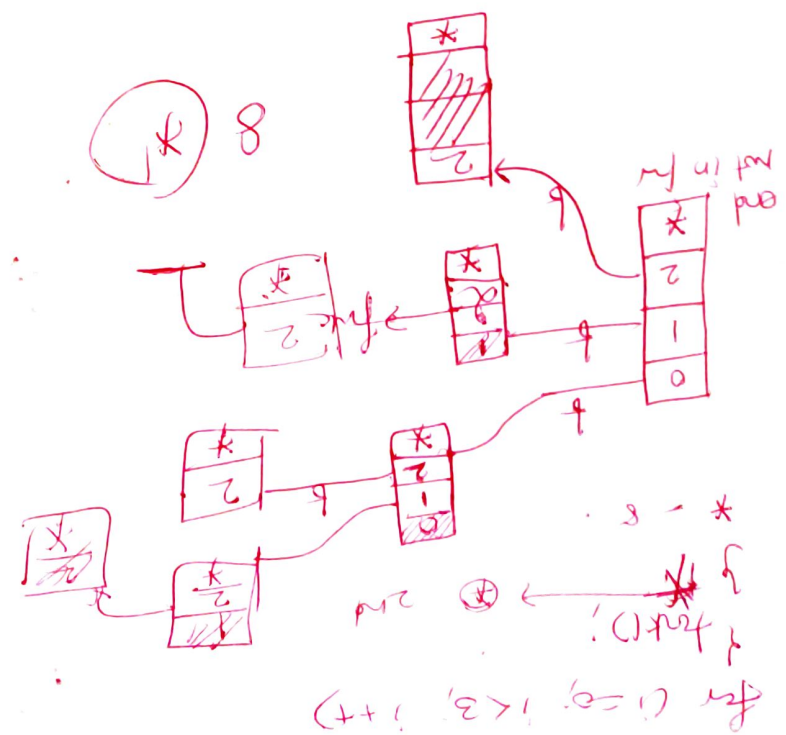
```
for(i=0; i<3; i++)
{
  fork();
  printf("%d", i);
}
```

Parent i=0 2 processes

i=1 4 processes

i=2





I just made a simple demo program in C to see how `fork()` works and came across something that I don't understand. In this example:

```
void fork2()
{
   printf("A");
   fork();
   printf("B");
}
```

The output is `ABAB`, but in this example:

```
void fork2()
{
   printf("A\n");
   fork();
   printf("B\n");
}
```

The output is `ABB` (separate lines of course). The second one makes sense, because if I understand `fork()` correctly it spawns a new child that starts right after where the `fork()` took place (so `printf("B\n");` in this case). What I don't understand, is why the output is different when I include the new line character.

c fork

`printf()` buffers output until a newline is encountered. So your `\n`-less version stuffs `A` into the output buffer, forks.
Since both processes are identical (minus PIDs and whatnot), they now both have an output buffer that contains `A`.
The execution continues and prints `B` into the buffers in each process. Both processes then exit, causing a flush of the output buffer, which prints `AB` twice, once for each process.
You other version, having `\n`, causes that output buffer to flush after the first printf() call, and when the fork hits, both processes have an empty buffer.

1 down vote

First of all, *do not* define a `fork()` function in your code. `fork()` is a *system call* already defined for you.
You call `fork()` and once it (successfully) returns, you have two identical processes executing the next line of the code past the `fork()` invocation. In the parent process `fork()` will return the PID of the child; in the child process the return code is 0. based on this return code, your two processes can do whatever they intended to based on their roles.
Now, there is *no guarantee* on relative order of execution of these two processes. For all you know, child might not execute for the next 2 minutes and it would still be the correct behavior. So, this means you cannot expect the order of the output from *both* of those processes to be predictable.
Finally, change `printf()` to `fprintf()` to avoid buffering (which confuses things further in this case).

A
~~fork();~~

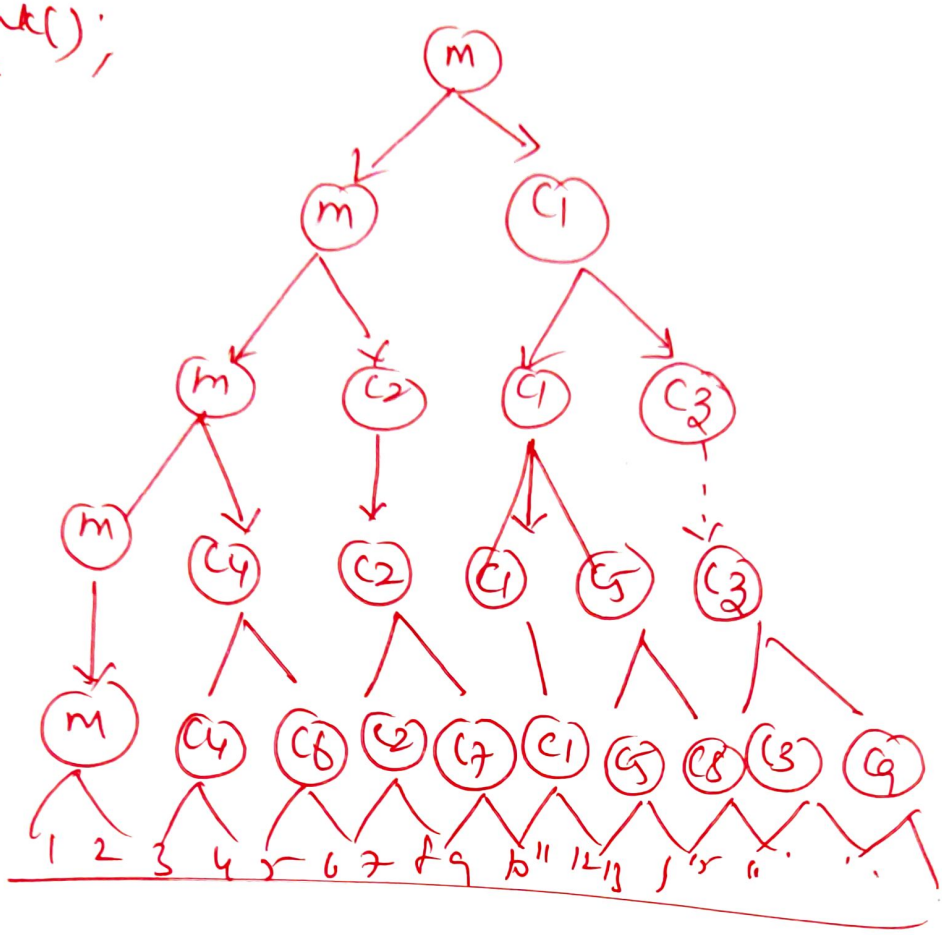
B | ~~fork() && fork()~~ | fork()

C D

fork();

=

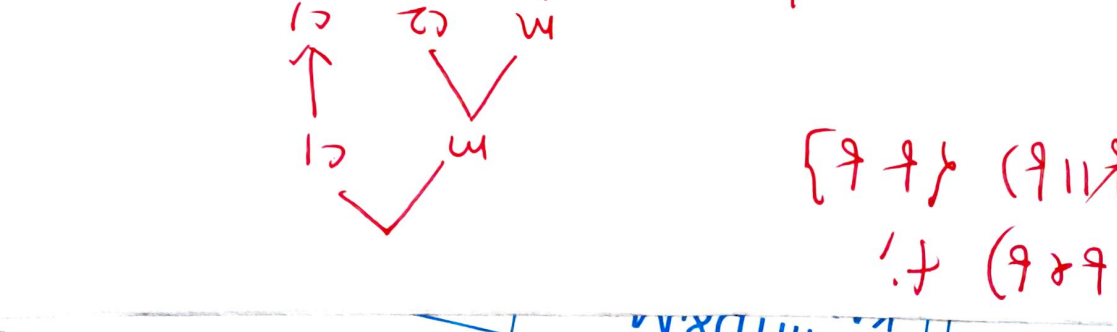
a & b



16

(a)

(b)



Interprocess Communication.

(a)

(x) Independent (or) (*) Cooperating processes

↳ cannot affect / be affected by other processes executing in the system

↳ does not share with any other process.

↳ reverse of (a)

Need for Cooperation:

(a) Information sharing (b) Computation Speedup (Chess — Subtasks)

(c) Modularity (d) Convenience

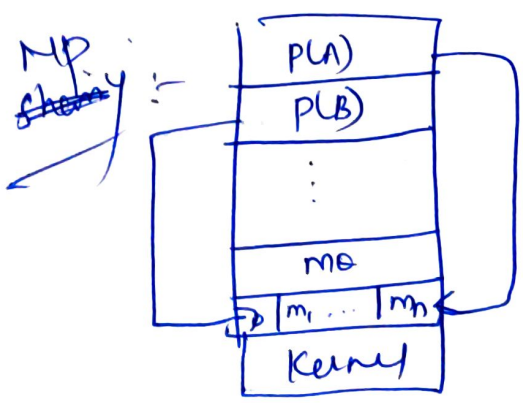
(b) Two models of ipc to exchge data / info

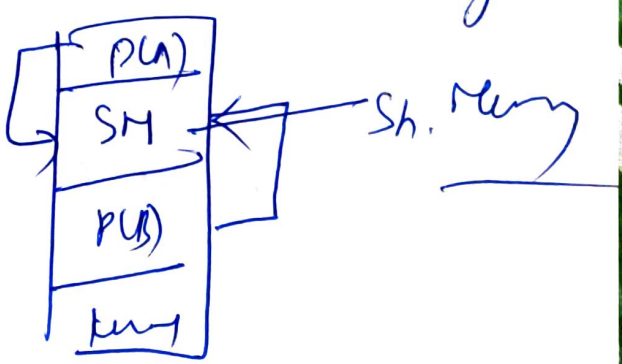
↳ Shared Memory / Message passing

↓

Memory region shared by cooperating process

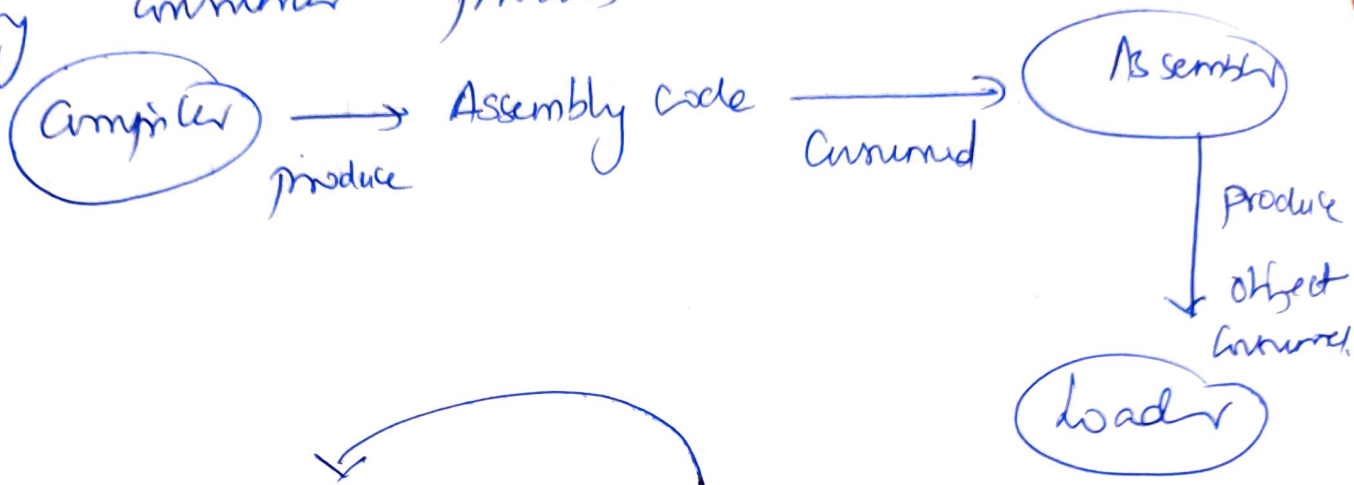
↳ process rd/write to shared region.



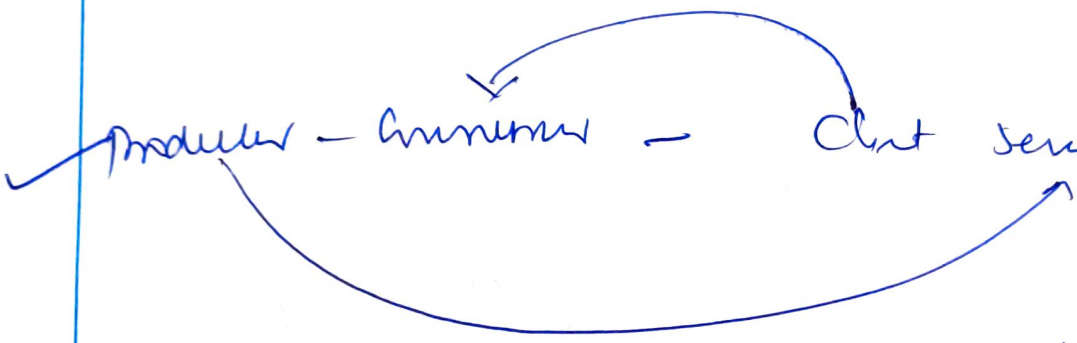


* Message passing better than sharing
↓
cache coherency issue

× Producer produces information that is consumed by consumer process



Producer – Consumer – Client server paradigm



shared memory:- Buffer of items that is filled by the producer & emptied by Consumer.

↓ (in memory) shared by P/C.

* Pr. produces 1 item while consu is cons. by Cr.

* consumer not consuming any unproduced item.

Sync.

* Unbounded / Bounded buffer

Unbounded :— No limit on Buffer

Producer → Can always produce items
Consumer → may have to wait for new items

Bounded : producer wait — full
Consumer wait — empty

```
while (true) {
        // item nextProduced
    while (((in+1) % BS == out)
        ; // full Buffer
    buffer[in] = nextProduced
    in = (in+1) % BS;
}
```

Buffer → circular array with 2 pointers
in (next free posn in Buffer)
out (first full ,, ,, ,,)
in == out / Empty
(in+1) % BS = out // full

while (free) { while (in == out)
;
next_consumed = buffer [out];
out = (out+1) % BS

]

MPI

send(message) / receive(message)

Messages ← fixed
variable

- Direct / Indirect Communication
- Sync / Async. Comm.
- Automatic / Explicit Buffering.

(1) Naming → Direct Comm.

Send (P, msg) :- send msg to proc P.
recv (Q, ") :- recv " from " Q.

- link — b/w 2 processes
- only link " " "
- auto link b/w pro. that want to Communicate

Pipes

conduit to allow two process to communicate

simplest mech. for IPC.

Issues to be addressed:-

(1) Bidir / Unidir Communication

(2) Half-duplex (data can travel only 1 way @ time)
full-duplex (" " " in both dirns)

(3) ? relationship (parent-child) exist b/w comm. procs

(4) Can pipes communicate over a n/w (or) must communicating processes reside

- child inherits pipe from parent
- parent or - pipe.

Ordinary pipes :- 2 processes communicate in standard-producer consumer fashion.

- producer writes @ one end {write-end}
- Consume reads from other end {read-end}
- Ordinary pipes — Unidirectional — One way communn.
- Two pipes — Two way — with each pipe sending data in different directions
- Procs writes message, other process reads using pipes

Pipes → pipe(int fd[2]);

- assumed there fd
- fd[0] → read end of pipe | fd[1] → write end
- unix → special type of file → read / write sys. calls

2017 → 2/9/17.

IIITD&M KANCHEEPURAM

Roll No : .....

Course No. : .....

* Simplex → one dir data transfer only.
Half duplex → both dir, not @ same time
full duplex → both dir, simultaneously.

* ord. pipe cannot be ~~created~~ accessed from outside the process that created it.

* Parent creates pipe and uses to comm. with child process.

* Child Inherits pipe from parent proc.

→ unistd.h
sys/types.h

```
#define BS 25
#define REND 0
#define WEND 1.
int main (void)
{
  char wm[BS] = "HITHAM K";
  char rm [BS];
  int fd[2];
  pid_t pid.
```

```
if (pipe(fd) == -1)
 fpf(stderr, "failed");
 return -1;
pid = fork();
if (pid < 0)
if (pid > 0)
{
   close(fd[READ]);
   write(fd[WRITE], wm, strlen(wm)+1);
   close(fd[WRITE]);
}
else
{
   close(fd[WRITE]);
   read(fd[READ], RM, BS);
   close(fd[READ]);
   Printf("%.s", RM);
return 0;
}
```

Shared memory

```
while (true){
    while ((in+1) % BS == out)
        ;
    buffer[in] = nextproduced;
    in = (in+1) % BS;
}
```

```
in = 0
out = 0
int buffer[BS];
```

```
while (true) {
    while (in == out)
        ;
    nc = buffer[out];
    out = (out+1) % BS;
}
```

in → Next free pos.

out → first full posn.

Shared Mem.

fcntl.h, (sys/stat.h, shm.h → Gj/).

```
const int size = 4096;
const char *m1 = "IIITT";
"    "  *m2 = "A";
"    "  *name = "OS";
int shm_fd;
void *ptr;
shm_fd =
shm_open(name, O_CREAT | O_RDRW, 0666);
ftruncate(shm_fd, size);
ptr = mmap(0, size, PROT_WRITE, MAP_SHARED, shm_fd, 0);
sprintf(ptr, "%s", m1);
ptr += strlen(m1);
sprintf(ptr, "%s", m2);
ptr += strlen(m2);
```

O_RDONLY.

shm_open(name, fr, 0666);

PROT_READ

printf("%s", ptr);

Inter process Comms → achieved in linux using Pipes and Shared Memory (SHM)

- Pipes are preferred as SHM is limited by buffer size and recv procs must wait till buffer is full.
- Pipes are supported by Message passing syscalls.

- classical example of producer - consumer relationship (p-c)
  - (a) Direct (b) Indirect
  - (b) Sync / Async ——— BS | BR | NBS | NBR
  - (c) Buffer < Bounded / unbounded
    - Blocking; Send (B); Send (S)
    - Receive (R)

- struct item [BufferSize] ; // array

Producer (in → ip slides, out → op hole)

```
while (true)
{
    // full Buffer
while ((in+1) % BS == out)
   ; // do nothing
buffer [in] = np; //new produce
}.
```

Consumer

```
while (true)
{
    //Empty Buffer
while ( in == out)
  ; // do nothing.
nc = buffer (out)
//new consume.
} out = (out+1) % BS;
```

//out → first full position
in → next posn available in array.

pipes :- pipe(int fd[2]);

Read 0 Write 1

BS 25,

char *com = " greetings ";

char rm[BS] = " ";

char rm[BS];

int fd[2];

pid_t pid;

if (pipe(fd) == -1)

{ err ; ret +1; }

Pid = fork();

if (pid < 0) { });

if pid > 0

{ close(fd[Read]);

write(fd[Write], com, Strlen(com)+1);

close(fd[Write]);

} else { close(fd[Write]);

read(fd[Read], rm, BS);

printf("%s", rm);

close(fd[Read]);

Two-way Communication Using Pipes

Pipe communication is viewed as only one-way communication i.e., either the parent process writes and the child process reads or vice-versa but not both. However, what if both the parent and the child needs to write and read from the pipes simultaneously, the solution is a two-way communication using pipes.

Two pipes are required to establish two-way communication.

Following are the steps to achieve two-way communication –

Step 1 – Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.

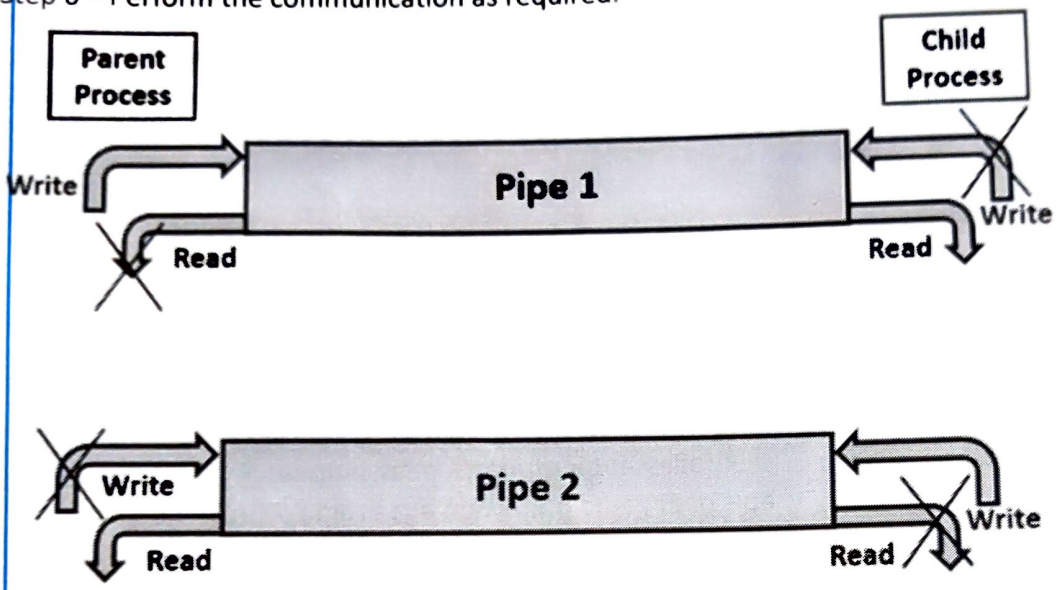
Step 2 – Create a child process.

Step 3 – Close unwanted ends as only one end is needed for each communication.

Step 4 – Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.

Step 5 – Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.

Step 6 – Perform the communication as required.



Sample Programs

Sample program 1 – Achieving two-way communication using pipes.

Algorithm

Step 1 – Create pipe1 for the parent process to write and the child process to read.

Step 2 – Create pipe2 for the child process to write and the parent process to read.

Step 3 – Close the unwanted ends of the pipe from the parent and child side.

Step 4 – Parent process to write a message and child process to read and display on the screen.

Step 5 – Child process to write a message and parent process to read and display on the screen.

Source Code: twowayspipe.c

```
#include<stdio.h>
#include<unistd.h>
int main() {
  int pipefds1[2], pipefds2[2];
  int returnstatus1, returnstatus2;
  int pid;
  char pipe1writemessage[20] = "Hi";
  char pipe2writemessage[20] = "Hello";
  char readmessage[20];
  returnstatus1 = pipe(pipefds1);
```

```
if (returnstatus1 == -1) {
  printf("Unable to create pipe 1 \n");
  return 1;
}
returnstatus2 = pipe(pipefds2);

if (returnstatus2 == -1) {
  printf("Unable to create pipe 2 \n");
  return 1;
}
pid = fork();

if (pid != 0) // Parent process {
  close(pipefds1[0]); // Close the unwanted pipe1 read side
  close(pipefds2[1]); // Close the unwanted pipe2 write side
  printf("In Parent: Writing to pipe 1 – Message is %s\n", pipe1writemessage);
  write(pipefds1[1], pipe1writemessage, sizeof(pipe1writemessage));
  read(pipefds2[0], readmessage, sizeof(readmessage));
  printf("In Parent: Reading from pipe 2 – Message is %s\n", readmessage);
} else { //child process
  close(pipefds1[1]); // Close the unwanted pipe1 write side
  close(pipefds2[0]); // Close the unwanted pipe2 read side
  read(pipefds1[0], readmessage, sizeof(readmessage));
  printf("In Child: Reading from pipe 1 – Message is %s\n", readmessage);
  printf("In Child: Writing to pipe 2 – Message is %s\n", pipe2writemessage);
  write(pipefds2[1], pipe2writemessage, sizeof(pipe2writemessage));
}
return 0;
}
```

Execution Steps

Compilation

gcc twowayspipe.c –o twowayspipe

Execution

In Parent: Writing to pipe 1 – Message is Hi
In Child: Reading from pipe 1 – Message is Hi
In Child: Writing to pipe 2 – Message is Hello
In Parent: Reading from pipe 2 – Message is Hello

$S \cdot T(1) + (1-S)\frac{T(1)}{N}$

IIITD&M KANCHEEPURAM

Roll No : .....

Course No. : Topic on Processor

Overall Speedup = $\frac{\text{Old ex time}}{\text{New ex time}} = \frac{T(1)}{T(j)}$

Time on j proc.

N processors for 1 task ⇒ Task Comp. time

$= 1/N$

Serial portion of Task = $S \cdot T(1)$ time

" " = $\frac{(1-S)\,T(1)}{N}$ time.

$$\frac{T(1)}{S \cdot T(1) + \frac{(1-S)\,T(1)}{N}} = \frac{T(1)}{T(1)\left\{S + \frac{1-S}{N}\right\}}$$

$$= \frac{1}{S + \frac{1-S}{N}}$$

$\frac{1-S}{N} \to 0$.

2 cores, 25% S.

Sp = 1.6 times.

↓ 2.28

4 cores

$N = \infty$. S = 40%

$Sp(up) = \frac{1}{.25} = \frac{1}{.4} = \frac{10}{4}$

$= 2.5$

child to finish and

Pipes are a simple, synchronised way of passing information between processe

Differences between files and pipes:

- **pipes have the bounded size** - the maximum capacity of a pipe is referenced by the constant `PIPE_BUF` (usually limited to 5120 bytes.)
- **sequential access** – one can only read or write from and to the pipe, the pointer of the current position can not be moved (`lseek` is unacceptable)
- `write` appends data to the input of a pipe while `read` reads any data from output of a pipe but:
- data that is read are **removed** from the pipe
- if the pipe is empty and at least one descriptor for reading is open, then the read is blocked until some data are written to the pipe or until the pipe descriptor is closed
- the process is blocked if it wants to write and the pipe is full

Pipes can be divided into two categories:

- unnamed pipes
- named pipes

Named pipes exist as directory entries and they can be used by unrelated processes provided that the processes know the name and location of the entry.

Next classes.
✓ modes of comm
✓ sitty
✓ pipes – unnamed

(1)

Unnamed pipes

Unnamed pipes may be only used with related processes (parent/child or child/child having the same parent). They exist as long as their descriptors are open.

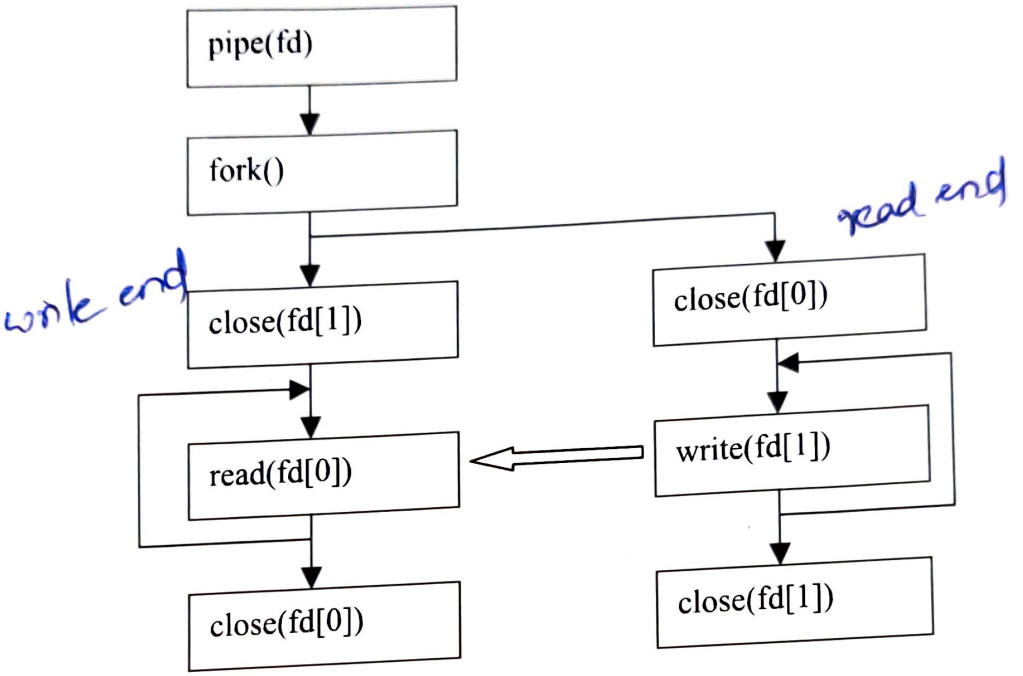
```
#include <unistd.h>

int pipe(int fd[2])
```

```
RETURNS: success : 0
         error: -1
```

1. `fd [0]` – descriptor used for reading
2. `fd [1]` – descriptor used for writing

Function creates the pipe and opens it for reading and writing.

Scheme of communication through pipe:



① Unidirectional pipe

Example 1 Communication via an unnamed pipe

```
main() {
int fd[2];

if (pipe(fd) == -1){
     perror("Create pipe error");
     exit(1);
}

switch(fork()){
case -1:
     perror("Failure on creating a process ");
     exit(1);
case 0:
     close (fd[0]);
     if (write(fd [1], "Hallo!", 7) == -1){
          perror("Writting to pipe error");
          exit(1);
     }
     exit(0);
default: {
     char buf[10];
     close (fd[1]);
     if (read(fd[0], buf, 10) == -1){
          perror("Reading from pipe error");
          exit(1);
     }
     printf("Data from the pipe: %s\n", buf);
}
}
}
```

① if (pid > 0) working. //child process else //parent process reads

②

```
pipe(fd1);  pipe(fd2);                    msg1, msg2
p = fork();        // Parent & child
if (p == 0) { close (pfd1[1]);
        read (pfd1[0], buf, BS);
    }
else { close (pfd1[0];
       write ( pfd1[1], msg, strlen(msg)+1);
}.
```

Example2 The problem of reading from the empty pipe

```
main() {
int fd[2];


pipe(fd);
if (fork() == 0){    // child process
  write(fd [1], "Hallo!", 7);
  exit(0);
}
else {                //parent process
  char buf[10];
  read(fd [0], buf, 10);
  read(fd[0], buf, 10);
  printf("Data from pipe: %s\n", buf)
}
}
```

3

read from empty pipe

lab 8mr:- pipe(fd1); 2, 3, 4.

child of fork (4 times)

ants c, w, l, s ant;

Parent of

for(i=0; i<5; i++) {

n= wait(&status)

if (n==pid1).

}.

Example 3 : Displaying result of `ls` in capital letters

```
#define MAX 512
main(int argc, char* argv[]) {

int fd[2];

if (pipe(fd) == -1){
     perror("Creating pipe");
     exit(1);
}

switch(fork()){
case -1:
     perror("Creating a process ");
     exit(1);
case 0:
     dup2(fd[1], 1);
     execvp("ls", argv);
     perror("program ls");
     exit(1);
default: {
     char buf[MAX];
     int nb, i;
     close(fd[1]);
     while ((nb=read(fd[0], buf, MAX)) > 0){
          for(i=0; i<nb; i++)
               buf[i] = toupper(buf[i]);
          if (write(1, buf, nb) == -1){
          perror ("Writting to stdout");

     exit(1);
}
}
if (nb == -1){
     perror("Reading from pipe");
     exit(1);
}
}
}
```

translikerate (replace)

Example 4 : `ls |tr 'a-z' 'A-Z'` with unnamed pipes

```
main(int argc, char* argv[]) {
int fd[2];

if (pipe(fd) == -1){
  perror("Creating pipe");
  exit(1);
}
switch(fork()){
 case -1:
     perror("Creating process");
     exit(1);
 case 0:                    // child process
     dup2(fd[1], 1);
     execvp("ls", argv);
     perror("command ls");
     exit(1);
 default: {                 // parent process
     close(fd[1]);
     dup2(fd[0], 0);
     execlp("tr", "tr", "a-z", "A-Z", 0);
     perror("command tr");
     exit(1);
 }
}
}
```

execution of ls | tr

Need to close pipes is avoided

rd end.

duplication of redirection on preferred descriptor.

Example 3 : Displaying result of ls in capital letters

```
#define MAX 512
main(int argc, char* argv[]) {

int fd[2];

if (pipe(fd) == -1){
     perror("Creating pipe");
     exit(1);
}

switch(fork()){
case -1:
     perror("Creating a process ");
     exit(1);
case 0:
     dup2(fd[1], 1);
     execvp("ls", argv);
     perror("program ls");
     exit(1);
default: {
     char buf[MAX];
     int nb, i;
     close(fd[1]);
     while ((nb=read(fd[0], buf, MAX)) > 0){
          for(i=0; i<nb; i++)
               buf[i] = toupper(buf[i]);
          if (write(1, buf, nb) == -1){
          perror ("Writting to stdout");

     exit(1);
}
}
if (nb == -1){
     perror("Reading from pipe");
     exit(1);
}
}
}
```

one pipe

fd 1 → stdout stream

1 for stdout

dup, dup2 → dup. fd's

↳ uses lowest # unused dsc.

↳ uses desc passed.

execvp( )

shm_id = shm_open(name, O_CREAT | O_RDRW, 0666);

ftruncate(shm_fd, size); // READ.

ptr = mmap(0, size, PROT_WRITE, MAP_SHARED);

sprintf(ptr, "%s", msg);

ptr += strlen(msg);

sprintf(ptr, "%s", msg);

}

Example 4 : ls |tr 'a-z' 'A-Z' with unnamed pipes

```
main(int argc, char* argv[]) {
int fd[2];

if (pipe(fd) == -1){
  perror("Creating pipe");
  exit(1);
}
switch(fork()){
 case -1:
     perror("Creating process");
     exit(1);
 case 0:                    // child process
     dup2(fd[1], 1);
     execvp("ls", argv);
     perror("command ls");
     exit(1);
 default: {                 // parent process
     close(fd[1]);
     dup2(fd[0], 0);
     execlp("tr", "tr", "a-z", "A-Z", 0);
     perror("command tr");
     exit(1);
 }
}
}
```

Case 0:

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

Case 1:

```
char buf[max], int nb;  → close(fd[1]);
while(nb= read(fd[0], buf, max) >0)
{
  for(i= 0; i<nb; i++)
     buf[i]= toupper(buf[i]);
}
write(1, buf, nb);
```

Named pipes – FIFO

- The named pipe has a directory entry.
- The directory entry allows using the FIFO for any process which knows its name, unlike unnamed pipes which are restricted only to related processes.

```
#include <sys/stat.h>
#include <sys/types.h>

int mkfifo(char * path, mode_t mode)
```

```
RETURNS: success : 0
         error: -1
```

path	points to the pathname of a file
mode	access rights

Function creates the pipe but **does not open it**

```
#include <fcntl.h>
main(){
     mkfifo("abc", 0600);
     open("abc", O_RDONLY);
}
```

open is blocked until another process opens FIFO in a complementary manner!

Example 5 : Creating and using named pipe

```
#include <fcntl.h>

main() {
 int fd;

 if (mkfifo("/tmp/fifo", 0600) == -1){
     perror("creating FIFO");
     exit(1);
 }
 switch(fork()){
   case -1:
     perror("Creating process");
     exit(1);
   case 0:
     fd= open("/tmp/fifo", O_WRONLY);
     if (fd== -1){
      perror("Opening FIFO for writing");
     exit(1);
     }
     if (write(fd, "Hallo!", 7) == -1){
          perror("writing to FIFO");
          exit(1);
      }
    exit(0);
 default: {
    char buf[10];
    fd = open("/tmp/fifo", O_RDONLY);
     if (fd== -1){
          perror("Opening FIFO for reading");
          exit(1);
     }
    if (read(fd, buf, 10) == -1){
          perror("Reading from FIFO");
          exit(1);
    }
    printf("Data read from FIFO: %s\n", buf);
     }
    }
}
```

Example 6 : ls |tr 'a-z' 'A-Z' with named pipes

```
#include <stdio.h>
#include <fcntl.h>

main(int argc, char* argv[]) {
 int fd;

 if (mkfifo("/tmp/fifo", 0600) == -1){
     perror("Tworzenie kolejki FIFO");
     exit(1);
 }
 switch(fork()){
     case -1:
          perror("Creating process");
          exit(1);
     case 0:
          close(1);
          fd= open("/tmp/fifo", O_WRONLY);
          if (fd == -1){
               perror("Opening FIFO for writing");
               exit(1);
          }
          else if (fd!= 1){
              fprintf(stderr, "Uncorrect write descriptor \n");
              exit(1);
          }
          execvp("ls", argv);
          perror("Running ls command");
          exit(1);
       default: {
          close(0);
          fd= open("/tmp/fifo", O_RDONLY);
          if (fd== -1){
               perror("Opening FIFO for reading");
               exit(1);
          }
          else if (fd!= 0){
              fprintf(stderr, "Uncorrect write descriptor \n");
              exit(1);
          }
          execlp("tr", "tr", "a-z", "A-Z", 0);
          perror("Running tr command");
          exit(1);
       }
     }
 }
```

Common failures

Example 7 : Failuers - unnamed pipes

```
#define MAX 512

main(int argc, char* argv[]) {
  int fd[2];
  if (pipe(fd) == -1){
          perror("Tworzenie potoku");
          exit(1);
  }
  if (fork() == 0){
     dup2(fd[1], 1);
     execvp("ls", argv);
     perror("Running ls");
     exit(1);
  }
  else {
     char buf[MAX];
     int nb, i;
     close(fd[1]);
     wait(0);
     while ((nb=read(fd[0], buf, MAX)) > 0){
          for(i=0; i<nb; i++)
          buf[i] = toupper(buf[i]);
          write(1, buf, nb);
 }
}
}
```

Example 8 : Failuers - named pipes

```
#include <fcntl.h>
#define MAX 512

main(int argc, char* argv[]) {
  int fd;

  if (mkfifo("/tmp/fifo", 0600) == -1){
     perror("Creating FIFO");
     exit(1);
  }
  if (fork() == 0){
     close(1);
     open("/tmp/fifo", O_WRONLY);
     execvp("ls", argv);
     perror("Running ls");
     exit(1);
   }
   else {
      char buf[MAX];
      int nb, i;

      wait(0);
      fd= open("/tmp/fifo", O_RDONLY);
      while ((nb=read(fd, buf, MAX)) > 0){
           for(i=0; i<nb; i++)
                buf[i] = toupper(buf[i]);
                write(1, buf, nb);
       }
   }
  }
```

Exercises:

Realize using unnamed and named pipes following commands:

1. `finger | cut -d' ' -f1`
2. `ls -l | grep ^d | more`
3. `ps -ef | tr -c \ \: | cut -d\: -f1 | sort | uniq -c | sort -n`

Case 0:

```
dup2(fd[1],1);
execvp("ls", argv); perror(" ");
exit(1);
```

Case 1:

```
int nb, char buf[max]; close(fd[1]);
while (nb = read(fd[0], buf, max) > 0)
{
    for (i=0; i<nb; i++)
        buf[i] = toupper(buf[i]);
    if (write(1, buf, nb) == -1)
        perror("----->");
}
```

What is a dup() system call

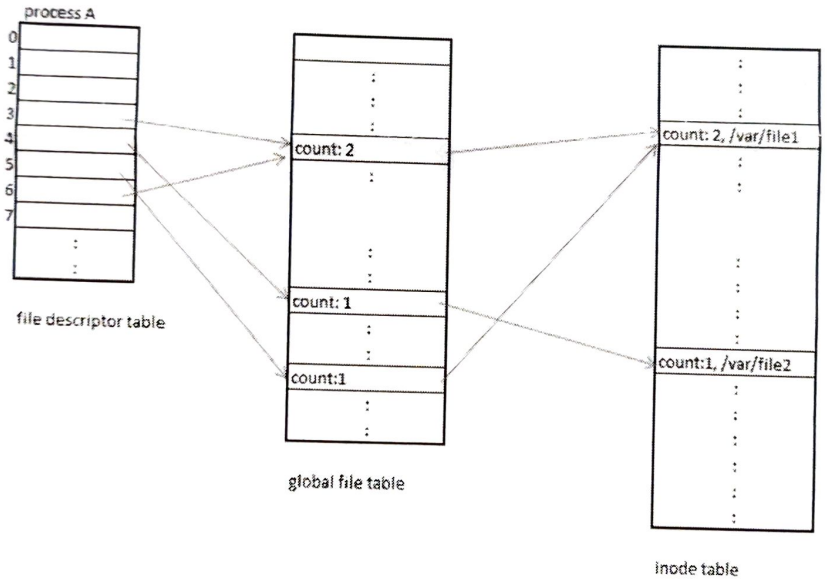
dup() system call in unix systems copies a file descriptor into the first free slot in the private file descriptor table and then returns the new file descriptor to the user. It works for all the file types. The syntax is :

```
newfd = dup(fd);
```

Here fd is the file descriptor being duped and newfd is returned to the user.

There are basically three different data structures that helps in manipulation of file system. These are - the ***inode table***, ***private user file descriptor table*** and the ***global file table***. Before moving forward to the description of *dup()* command, I urge you to please follow this article on Internal Data Structure for file handling in Unix kernel.

dup() system call doesnt create a separate entry in the global file table like the *open()*system call, instead it just increments the count field of the entry pointed to by the given input file descriptor in the global file table. Consider an example where fd 0, 1 and 2 are by default engaged to the standard input/output and error. Then if the user opens a file "**/var/file1**" (fd - 3), then he opens file "**/var/file2**" (fd - 4) and again he opened "**/var/file1**" (fd - 5). And now, if he does a ***dup(3)***, kernel would follow the pointer from the user file descriptive table for the fd entry '3', and increments the count value in the global file table. Then, it searches for the next avaialable free entry in file descriptor table and returns that value to the user (6 in this case).



Difference between open and dup system call

```
#include "fcntl.h"
int main()
{
 int i,j;
 char buf1[512],buf2[512];
```

```
i = open("/var/file1", O_RDONLY);
j = dup(i);

read(i, buf1, sizeof(buf1));
read(j, buf2, sizeof(buf2));
close(i);
read(j, buf2, sizeof(buf2));
return 0;
}
```

In the above program after doing the *dup(i)*, both i and j point to the same entry in the global file table and thus share the same byte offset in the file. Thus the next two read operation will read different data and buf1 and buf2 will have different data. This is where the difference comes into picture when compared to the *open()* system call. If instead of *dup()*, we use *open()* call to open the same file again, it will create a separate entry in the global file table and thus separate byte offsets for the files opened. The effect is same content in bufferes buf1 and buf2. The user can close either of the file descriptors and continue using the other without any issue.

Follow this article for the description of open() system call.

**** Input / output redirection using dup() system call***

dup() system call finds use in implementing input/output redirection or piping the output on unix shell. Suppose, we wish to redirect the output of 'ls' command to a file, we use the following command on shell to do our job:

root> ls /var/ > tempfile*

File descriptor 1 is bound to the standard output stream. The 'ls /var/*' command is supposed to output the data on this output stream i.e. 1. But, using '>' operator we are able to redirect this output to file 'tempfile'. What happens when the process that is executing the shell here is that it parses the command and when it finds '>' operator, it will first find the file descriptor of the rhs operand - 'tempfile' OR create the new fd if file doesnt exist already. Once, it finds this fd, it will close the stdout file descriptor and call a *dup()* on the given fd for this 'tempfile'.
Thats it, from this step onwards, the output will be redirected to the file 'tempfile'. We can also do an additional step of closing the file descriptor to preserve the number of descriptors.

```
/*redirection of I/O*/
 {
  fd = creat('tempfile', flags);
  close(stdout);  //stdout => 1
  dup(fd);
  close(fd);
  /* stdout is now redirected */
 }
```

The same logic is applied when we apply "pipe" operations on the shell. Thus, although *dup()* is not an elegant command but yet it is a powerful building block for several higher level commands.

redirect.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

/**
 * Executes the command "grep Villanova < scores > out".
 *
 * @author Jim Glenn
 * @version 0.1 9/23/2004
 */

int main(int argc, char **argv)
{
  int in, out;
  char *grep_args[] = {"grep", "Villanova", NULL};

  // open input and output files

  in = open("scores", O_RDONLY);
  out = open("out", O_WRONLY | O_TRUNC | O_CREAT, S_IRUSR | S_IRGRP | S_IWGRP
| S_IWUSR);

  // replace standard input with input file

  dup2(in, 0);

  // replace standard output with output file

  dup2(out, 1);

  // close unused file descriptors

  close(in);
  close(out);

  // execute grep

  execvp("grep", grep_args);
}
```

pipe.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

/**
 * Executes the command "cat scores | grep Villanova".  In this quick-and-
dirty
 * implementation the parent doesn't wait for the child to finish and
 * so the command prompt may reappear before the child terminates.
 *
 * @author Jim Glenn
 * @version 0.1 9/23/2004
```

IIITD&M KANCHEEPURAM

Invigilator's Signature :

Date :

Course No. :

Roll No :

MARKS

```
 */

int main(int argc, char **argv)
{
  int pipefd[2];
  int pid;

  char *cat_args[] = {"cat", "scores", NULL};
  char *grep_args[] = {"grep", "Villanova", NULL};

  // make a pipe (fds go in pipefd[0] and pipefd[1])

  pipe(pipefd);

  pid = fork();

  if (pid == 0)
    {
      // child gets here and handles "grep Villanova"

      // replace standard input with input part of pipe

      dup2(pipefd[0], 0);

      // close unused hald of pipe

      close(pipefd[1]);

      // execute grep

      execvp("grep", grep_args);
    }
  else
    {
      // parent gets here and handles "cat scores"

      // replace standard output with output part of pipe

      dup2(pipefd[1], 1);

      // close unused unput half of pipe

      close(pipefd[0]);

      // execute cat

      execvp("cat", cat_args);
    }
}
```

twopipes.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

/**
```

```
 * Executes the command "cat scores | grep Villanova | cut -b 1-10".
 * This quick-and-dirty version does no error checking.
 *
 * @author Jim Glenn
 * @version 0.1 10/4/2004
 */

int main(int argc, char **argv)
{
  int status;
  int i;

  // arguments for commands; your parser would be responsible for
  // setting up arrays like these

  char *cat_args[] = {"cat", "scores", NULL};
  char *grep_args[] = {"grep", "Villanova", NULL};
  char *cut_args[] = {"cut", "-b", "1-10", NULL};

  // make 2 pipes (cat to grep and grep to cut); each has 2 fds

  int pipes[4];
  pipe(pipes); // sets up 1st pipe
  pipe(pipes + 2); // sets up 2nd pipe

  // we now have 4 fds:
  // pipes[0] = read end of cat->grep pipe (read by grep)
  // pipes[1] = write end of cat->grep pipe (written by cat)
  // pipes[2] = read end of grep->cut pipe (read by cut)
  // pipes[3] = write end of grep->cut pipe (written by grep)

  // Note that the code in each if is basically identical, so you
  // could set up a loop to handle it.  The differences are in the
  // indicies into pipes used for the dup2 system call
  // and that the 1st and last only deal with the end of one pipe.

  // fork the first child (to execute cat)

  if (fork() == 0)
    {
      // replace cat's stdout with write part of 1st pipe

      dup2(pipes[1], 1);

      // close all pipes (very important!); end we're using was safely copied

      close(pipes[0]);
      close(pipes[1]);
      close(pipes[2]);
      close(pipes[3]);

      execvp(*cat_args, cat_args);
    }
  else
    {
      // fork second child (to execute grep)
```

```
    if (fork() == 0)
    {
      // replace grep's stdin with read end of 1st pipe

      dup2(pipes[0], 0);

      // replace grep's stdout with write end of 2nd pipe

      dup2(pipes[3], 1);

      // close all ends of pipes

      close(pipes[0]);
      close(pipes[1]);
      close(pipes[2]);
      close(pipes[3]);

      execvp(*grep_args, grep_args);
    }
    else
    {
      // fork third child (to execute cut)

      if (fork() == 0)
        {
          // replace cut's stdin with input read of 2nd pipe

          dup2(pipes[2], 0);

          // close all ends of pipes

          close(pipes[0]);
          close(pipes[1]);
          close(pipes[2]);
          close(pipes[3]);

          execvp(*cut_args, cut_args);
        }
    }
  }

// only the parent gets here and waits for 3 children to finish

close(pipes[0]);
close(pipes[1]);
close(pipes[2]);
close(pipes[3]);

for (i = 0; i < 3; i++)
  wait(&status);
}
```

This code can also be downloaded from the files redirect.c, pipe.c, and twopipes.c.

POSIX → Portable Operating System Interface

① Identifying Tasks ② Balance

③ Data Splitting ④ Data dependency

```
int sum;
<pthread.h>

main (int argc, char * argv[])
{
  pthread_t tid;
  pthread_attr_t attr;

  if (argc != 2)
  { fprintf(stderr, "usage: a.out <integer value>\n"); return 1; }

  if (atoi (argv[1]) < 0)
  { fprintf(stderr, "%d must be >= 0\n", atoi(argv[1])); return 1; }

  pthread_attr_init (&attr);
  pthread_create (&tid, &attr, runner, argv[1]);
  pthread_join (tid, NULL);
  printf ("sum = %d\n", sum);
}

void * runner (void * param)
{
  int i, upper = atoi (param);
  sum = 0;
  if (upper > 0)
    for (i = 1; i <= upper; i++)
      sum += i;
  pthread_exit (0);
}
```

Threading.

✓ Basic unit of cpu utilization

↳ thread id, pc, register set, stack.

↳ shares d/s. c/s & os signals / files with other threads in same process.

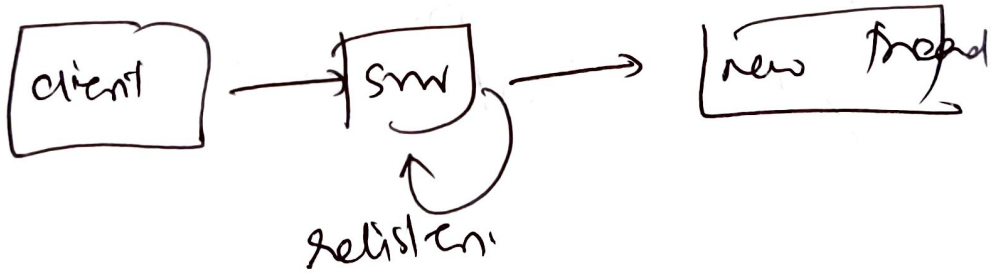
✓ client - server arch.

Web browser req — Image / Text / MM.

sep threads.

every new client → thread v/s process

✓ Word — Graphics, spell check, print } threads



Benefits

✓ Responsiveness ✓ Resource sharing ✓

Economy ✓ Scalability.

↳ multiprocessor arch.

↓

Context switching.

$n \rightarrow$ no. of threads of execution

$B \in (0,1) \rightarrow$ fraction of alg. that is strictly serial

$$T(n) = T(1)\left(B + \frac{1}{n}(1-B)\right)$$

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1-B)\right)}$$

$$= \frac{1}{B + \frac{1-B}{n}}$$

12% serial, 88% parallel

$$\text{Max speedup} = \frac{1}{1-0.12} = \frac{1}{0.88}$$

$S = .25$, $P = .75$

$N = 2$

$P_1 \ldots P_4$

11.1% | 18 | 23 | 48%

↳ 1.6

4 cores → 2.28

$N \longrightarrow \infty$.

$\frac{1}{40\%} = \frac{100}{40}$

Speed up is 2.5 times

= 2.5

Serial portion has effect on speed up

$S_p \frac{T(1)}{T(n)}$

$B\,T(1) + \left(\frac{1-B}{N}\right) T(1) = T(1)\{B + 1 - B/N\}$

Challenges :-

- Identifying tasks
  - ↳ areas that can be divided into separate, concurrent tasks (indep of one another)
- Balance – tasks must perform equal work of equal value
- Data splitting –
- data dependency – synchronization if one task is dep on data from another.
- Testing & debugging – many combn. paths possible

Testing Concurrent pgms >> diff in Single threaded cpgm.

* Threading models :-

User, kernel threads

↳ supported above kernel, no kernel support.

* Data ||ism → distn of data across multiple cores

Task ||ism → distn of tasks across multiple cores.

* Relship b/w user threads, kernel threads

@ m:1 model

m UT's ⟶ 1 KT.

↳ process will block if a thread makes a blocking sys call

↳ lacks to exploit multicore advantage

m:1
1:1
m:m
1:m

b) 1:1 Model

✓ 1 UT ⟶ 1 KT

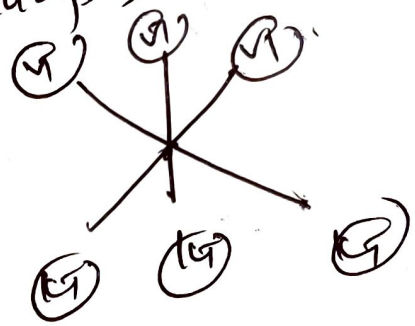
✓ more Concurrency, 1 thread blocks, another thread can run.

✓ Restn on #C threads) bcoz of overhead of KT creation.

M:M Model:

Many UT ⟶ Small / Equal KT's

multiplex



```
#include <pthread.h>
#  "      <stdio.h>
int sum;
void * runner (void * param);
int main (int argc, char *argv[])
{
  pthread_t tid;
  pthread_attr_t attrib;        // thread attributes
  if (argc != 2) {
     fprintf(stderr, "usage: a.out <integer value>\n");
     return -1; }
  if (atoi (argv[1]) < 0) {
     fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
     return -1; }
  pthread_attr_init (&attrib);  //set the attributes
  //  - create (&tid, &attributes, runner, argv[1]);
  //  - join (tid, NULL);
  printf("sum = %d\n", sum);
}
```

```
void * runner (void *param)
{ int i, upper = atoi (param);
  sum = 0;
  for (i = 0 ; i <= upper; i++)
      sum = sum + i;
  pthread_exit (0);
}
```

pthread_join — wait for child thread to terminate

① cc -pthread *.c

② gcc shmlc -o new -lrt

(2) List all primes before No;

```
Void * runner ( void * param)
{
  int i, j, upper = atoi (param);
for (i=2;  i<upper;  i++)
{  int trap=0;

  for (j=2; j < i;  j++)
    {
      int res =  i % j;

    if (res == 0)
      { trap=1;  break; }

  }
  if (trap == 0)  printf ("%d", i);
}
}
```

✓ Define thread reference variable (pthread_t tid);

✓ entry point for thread (runner)

Threading

(1) Prog. challenges →

(i) Task Identification → Program areas that can be parallelized / concurrent can. enabled.

(ii) Balance → tasks must perform equal work of equal value

(iii) Data splitting

(iv) Data Dependency by tasks — Synchronization to be ensured

$T_1 \rightarrow T_2$ (reads x written by T_1)

(v) Testing / Debugging → Multithreaded can. test cases | " is more difficult

(2) Parallelism types

(a) Data → Dist. ^ subsets of data across cores

A [0] ⟶ N

$T_1 \rightarrow \sum(0 \rightarrow N/2-1)$ and T_2 $\sum(N/2$ to $N-1)$

(b) Task → Distribute tasks and not data across threads.

✓ Mostly Hybrid of both followed

m: 1	1: 1	m: n.
UT KT	UT KT	KT UT

<pthread.h>

$ ./a.out 5 = 15.

int sum; void * run (void * param)

shared data across threads. ↳ thread calls this fn.

```
int main (int argc, char **argv)
{
  pthread_t tid;          // th. id
  pthread_attr_t attr;    // set of th. attribs
  if (argc != 2)
  {fprintf (stderr, "usage a.out int \n");
   return -1;}
  if (atoi (argv[1]) < 0)
  {
      "n must be >0";
  }return -1;
  pthread_attr_init (&attr); // default attributes.
  pthread_create (&tid, &attr, run, argv[1]);
                                  n.
  //Th. creation
  pthread_join (tid, NULL);
  P ("%d", sum); }.
```

Validation

m:1 1:1 m:n.

IIITD&M KANCHEEPURAM

Roll No : 2

Course No. :

```
Void * run (Void * param)
{
    int limit = atoi (param); sum = 0;
    for (i = 1; i <= limit; i++)
      sum = sum + i;
    Pthread_exit (0);
}
```

→

Thread pools:- Unlimited creation can exhaust memory / cpu time.

✓ create 'n' threads @ process start up in a pool

✓ threads allocated on receipt of requests from application

✓ on completion returns to pool & waits for more requests

✓ no thread free; requests wait for a thread

* No Inf. creation / Bound on memory.

* Initialization etc. only once.

↳ create threads after > than serving (costly) request with an existing thread.

Matrix Multiplication / using threads.

dummy

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define M 3
#define K 2
#define N 3
#define NUM_THREADS 10

int A [M][K] = { {2,2}, {2,2}, {3,6} };
int B [K][N] = { {8,7,6}, {5,4,3} };
int C [M][N];

struct v {
   int i; /* row */
   int j; /* column */
};

void *runner(void *param); /* the thread */

int main(int argc, char *argv[]) {

   int i,j, count = 0;
   for(i = 0; i < M; i++) {
      for(j = 0; j < N; j++) {
         //Assign a row and column for each thread
         struct v *data = (struct v *) malloc(sizeof(struct v));
         data->i = i;
         data->j = j;
         /* Now create the thread passing it data as a parameter
*/
         pthread_t tid;       //Thread ID
         pthread_attr_t attr; //Set of thread attributes
         //Get the default attributes
         pthread_attr_init(&attr);
         //Create the thread
         pthread_create(&tid,&attr,runner,data);
         //Make sure the parent waits for all thread to complete
         pthread_join(tid, NULL);
         //count++;
      }
   }

   //Print out the resulting matrix
   for(i = 0; i < M; i++) {
      for(j = 0; j < N; j++) {
         printf("%d ", C[i][j]);
      }
      printf("\n");
   }
}
```

No of rows in 2nd matrix
cols in 1st "

to hold row, column no.

```
for (i=0; i<N; i++)
  for (j=0; j<N; i++)
  {
    struct v * data = (struct v *) malloc ( sizeof (struct v) )

    data -> i = i;
    data -> j = j;
    P_c(&hd, &attr, run, data)

void * run( void * param)
{
    struct v * data = param;
    fun ~;
    for( z=0; z < k ; z++)
        sf += A[data->i][z] * B[z][data->j]

    c[di][dj] = s.
```

```
//The thread will begin control in this function
void *runner(void *param) {
   struct v *data = param; // the structure that holds our data
   int n, sum = 0; //the counter and sum

   //Row multiplied by column
   for(n = 0; n< K; n++){
      sum += A[data->i][n] * B[n][data->j];
   }
   //assign the sum to its coordinate
   C[data->i][data->j] = sum;

   //Exit the thread
   pthread_exit(0);
}
```

2 rows in 2nd Matrix.