☞ Chain of Processes : A chain of processes is a sequence where each process creates a single child process, forming a linear hierarchy.

.....

Example 1:

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

int main()
{
      int pid;
      int n=4;

      for (int i = 1; i < n; i++) {
            pid = fork();

            if (pid > 0) {
                  // Print the parent process
                  printf("Parent process ID is %d\n", getpid());
                  break;
            }
            else {
                  // Print the child process
                  printf("Child process ID is %d\n", getpid());
            }
      }

      return 0;
}
```

Output :

```
Parent process ID is 8678
Child process ID is 8679
Parent process ID is 8679
Child process ID is 8680
Parent process ID is 8680
Child process ID is 8681
```

Example 2:

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

int main() {
    pid_t pid;
    for (int i = 0; i < 3; i++) { // 3 child processes in a chain
        pid = fork();
        if (pid == 0) {
            printf("Process ID: %d, Parent ID: %d\n", getpid(), getppid());
        } else {
            break; // Only the child continues to fork
```

```
        }
    }
    return 0;
}

Example 3:
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    int n = 4;

    for (int i = 1; i < n; i++) {
        pid_t child_pid = fork();

        if (child_pid == -1) {
            // Fork failed
            perror("fork");
        }

        if (child_pid == 0) {
            // Child process
            printf("Child %d with PID %d is created. Parent PID %d\n", i ,
getpid(), getppid());
        }
       else {
            // Parent process
            printf("Parent PID: %d\n", getpid());
            // Wait for the child process to finish before creating the next one
            wait(NULL);
            break; // Prevent the parent from creating additional children
        }
    }

    return 0;
}


Output:
Parent PID: 15385
Child 1 with PID 15386 is created. Parent PID 15385
Parent PID: 15386
Child 2 with PID 15388 is created. Parent PID 15386
Parent PID: 15388
Child 3 with PID 15389 is created. Parent PID 15388


Example 4:
Creates a chain of n processes, where n is a command line argument.

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
pid_t childpid = 0;
int i, n;
if (argc != 2)
```

```
{/* check for valid number of
command-line arguments */
fprintf(stderr, "Usage: %s processes\n", argv[0]);
return 1;
}
n = atoi(argv[1]);
for (i = 1; i < n; i++)
if (childpid=fork())
break;
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n", i,
(long)getpid(), (long)getppid(), (long)childpid);
return 0;
}
```

☞ Fan of Processes : A fan of processes is a structure where a single parent process creates multiple child processes directly, creating a star-like hierarchy.

Example 1:

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

int main()
{
      for (int i = 1; i < 4; i++) {
            if (fork() == 0) {
                  printf("child pid %d from the parent pid %d\n", getpid(),
getppid());

                  exit(0);
            }
      }
}
```

Output:

```
child pid 2833 from the parent pid 2832
child pid 2834 from the parent pid 2832
child pid 2835 from the parent pid 2832
```

Example 2:

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

int main() {
    int n = 4; // Number of child processes to create

    printf("Parent process ID: %d\n", getpid());

    for (int i = 1; i < n; i++) {
        pid_t child_pid = fork();
```

```
        if (child_pid == -1) {
            // Fork failed
            perror("Fork failed");
            return 1;
        } else if (child_pid == 0) {
            // Child process
            printf("Child process %d with ID: %d and Parent ID: %d\n", i, getpid(),
getppid());


            // Exit the child process
            return 0;
        } else {
            // Parent process Wait for the child process to finish before creating
the next one
            wait(NULL);
        }
    }

    // Code executed only by the parent process
    printf("All child processes have completed.\n");

    return 0;
}
```

Output:

```
Parent process ID: 5800
Child process 1 with ID: 5801 and Parent ID: 5800
Child process 2 with ID: 5802 and Parent ID: 5800
Child process 3 with ID: 5803 and Parent ID: 5800
All child processes have completed.
```

Example 3:

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

int main (int argc, char *argv[]) {
pid_t childpid = 0;
int i, n;
if (argc != 2){
/* check for valid number of command-line
arguments */
fprintf(stderr, "Usage: %s processes\n", argv
[0]);
return 1;
}
n = atoi(argv[1]);
for (i = 1; i < n; i++)
if ((childpid = fork()) <= 0)
break;
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",i,
(long)getpid(), (long)getppid(), (long)childpid);
return 0;
}
```

Explain what happens when you replace the test (childpid = fork()) <= 0 of the previous program with (childpid = fork()) == -1 ?
Draw a suitable diagram labelling the circles with the actual process.

pstree : shows the hierarchical view of processes in a tree-like structure.
visualize parent-child relationships between processes.

☞ wait & waitpid

.....

wait system call

Example 1 :

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

int main()
{
    pid_t id = fork();

    if(id==0)
    {
       printf("I am child and my pid is :%d\n", getpid());
       sleep(5);
       exit(0);
     }
     printf("Before wait call \n");
     pid_t waitr = wait(NULL);    //return childid
     printf("Parent executed wait and wait return :%d\n",waitr);
}
```

Example 2 :

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

int main()
{
    int status;
    pid_t id = fork();

    if(id==0)
    {
```

```
       printf("I am child and my pid is :%d\n", getpid());
       pid_t waitr = wait(NULL);
       if(waitr==-1){
         perror("Error is :");
        }
       exit(0);
     }
     printf("I am Parent\n");

     //printf("Status=%d\n",status);
}
```

Example 3:

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

int main()
{
    int status;
    pid_t id = fork();

    if(id==0)
    {
       printf("I am child and my pid is :%d\n", getpid());
       sleep(5);
       exit(0);  // Normally Terminated
     }
     printf("Before wait call \n");
     pid_t waitr = wait(&status);
     printf("Parent executed wait and wait return\n",waitr);

     printf("Status=%d\n",status);   //status return 0 because exit(0)
}
```

status = 32 bit for representation 16 bit is used.
Process termination status (int) 16 bit number

1. status value if child process has normal exit/termination (exit(0))

```
   15......8    |     7......0
   XXXXXXXX     |     XXXXXXXX
   exitStatus   |      unused
```

  exit(0) status=0

```
   15......8    |     7......0
   00000000     |     00000000
   exitStatus   |      unused
```

  exit(1) status=256

```
   15......8    |     7......0
   00000001     |     00000000
   exitStatus   |      unused
```

2. killed by signal // stopped by signal

```
    15......8    |     7   ......0
    unused       |     X   termination signal
                         |
                         |-----> core dump flag
```

```
  kill -9 childpid   status = 9 (signal Number)

   00000000    |     00001001
    unused     |     termination signal
```

3. Core dump flag (Segmentation Fault, Floating point exception)

```
15......8    |     7   ......0
    unused   |     1   termination signal
             |
                   |-----> core dump flag
```

Example : Core dump

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

int main()
{
    int status;
    pid_t id = fork();

    if(id==0)
    {
       printf("I am child and my pid is :%d\n", getpid());

      int *p=NULL;
      *p=7;

       exit(1);
     }
     printf("Before wait call \n");
     pid_t waitr = wait(&status);
     printf("Parent executed wait and wait return\n",waitr);

     printf("Status=%d\n",status);
}
```

status = 128+11(segmentation fault signal) = 139

What will be the output of this code ?

Example 1 :

```
int main()
{ /* PID of child 3425 */
pid_t childpid,waitreturn; /* PID of parent 3424 */
childpid = fork();
if(childpid == 0)
{
printf("Process ID=%ld\n",(long)getpid());
}
waitreturn=wait(NULL);
if (childpid != waitreturn) // 0 != -1 True for child
{
printf("Return value of fork=%ld\n",(long)childpid);
printf("Process ID=%ld\n",(long)getpid());
printf("Return value of wait=%d\n",waitreturn);
}
return 0;
}
```

Example 2 :

```
int main()
{ /* PID of child 3425 */
pid_t childpid,waitreturn; /* PID of parent 3424 */
childpid = fork();
if(childpid == 0)
{
printf("Process ID=%ld\n",(long)getpid());
}
waitreturn=wait(NULL);
if (childpid == waitreturn) // True for Parent process
{
printf("Return value of fork=%ld\n",(long)childpid);
printf("Process ID=%ld\n",(long)getpid());
printf("Return value of wait=%d\n",waitreturn);
}
return 0;
}
```

Example 3 :

```
int main()
{ /* PID of child 3425 */
pid_t childpid; /* PID of parent 3424 */
childpid = fork();

// Code executed by both parent and child processes

if(childpid == 0)
```

```
{
printf("Process ID=%ld\n",(long)getpid());   // Code executed by the child process
}
if (childpid == wait(NULL))    // Code executed by the parent process
printf("Return value of fork=%ld\n",(long)childpid);
printf("Process ID=%ld\n",(long)getpid());
}
return 0;
}
```

In the child process (if (childpid == 0)), it prints the Child Process ID using getpid().
In the parent process (else part), it waits for the child process to terminate using wait(NULL).
The return value of wait() is the process ID of the terminated child process, which is printed.

Output :
Child Process ID=3425
Return value of fork=3425
Parent Process ID=3424

Example 4 :

```
int main()
{ /* PID of child 3425 */
pid_t childpid; /* PID of parent 3424 */
childpid = fork();
if(childpid == 0)
{
printf("Process ID=%ld\n",(long)getpid());
}
if (childpid != wait(NULL))
printf("Return value of fork=%ld\n",(long)childpid);
printf("Process ID=%ld\n",(long)getpid());
}
return 0;
}
```

Limitation of wait() System Call :

☞ It is not possible for the parent to retrieve the signal number during which the child process has stopped the execution (SIGSTOP(19), SIGTSTP(20)).

☞ Parent won’t be able to get the notification when a stopped child was resumed by the delivery of a signal (SIGCONT(18), SIGCHLD(17)).

☞ Parent can only wait for first child that terminates. It is not possible to wait for a particular child.

☞ It is not possible for a non blocking wait so that if no child has yet terminated, parent get an indication of this fact.

.....
.....

☞ waitpid

☞ The waitpid function lets us wait for one particular process, whereas the wait function returns the status of any terminated child.

☞ The waitpid function provides a nonblocking version of wait. There are times when we want to fetch a child’s status, but we don’t want to block.

☞ The waitpid function provides support for job control with the WUNTRACED and WCONTINUED options.

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

a pid
a pointer to a location for returning the status
a flag specifying options.

waitpid(): The parent process uses this function to wait for the termination of a specific child process.

Parameters:

pid: The PID of the child to wait for.
&status: Pointer to an integer where the child's termination status is stored.
0: Options flag (0 means wait until the specified child terminates).

First Parameter :

pid=-1 waits for any child
pid>0 waits for the specific child
pid=0 waits for any child in the same process group
pid<-1 waits for any child in the process group specified by the absolute value of pid |-2|=2

Note :

```
wait(NULL) or waitpid(-1, NULL, 0);

int status;
wait(&status) or waitpid(-1, &status, 0);
```

Third parameter :

can be zero , options = 0 waitpid waits until the child change state.
or
can be the bitwise inclusive OR of the constants: WNOHANG, WUNTRACED,and WCONTINUED

Example 1 :

```
/* Program to demonstrate waitpid and exit*/

#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
    pid_t cpid;
    pid_t cpid2;
    pid_t ret_pid;
    int status = 0;
    cpid = fork();
    if(cpid == -1)
        exit(-1);           /* terminate child  depending on use case*/

    if(cpid == 0){
        printf("\nchild-1 executing  its pid = (%d)\n",getpid());
        sleep(10);

        printf("Child1 exited\n");
        exit(0);
    }
    else{

        cpid2 = fork();
        if(cpid2 == -1){
            exit(-1);
        }
        if(cpid2 == 0){
            printf("\nchild-2 executing  its pid = (%d)\n",getpid());
            sleep(5);
            printf("Child2 exited\n");
            exit(1);
        }

        printf("\n Parent executing before wait() Parent pid is (%d)\n",getpid());
        ret_pid = waitpid(cpid2, &status, 0);
        printf("\n cpid returned is (%d)\n",ret_pid);
        printf("\n status is (%d)\n",status);


        ret_pid = waitpid(cpid, &status, 0);
        printf("\n cpid returned is (%d)\n", ret_pid);
        printf("\n status is (%d)\n",status);

        printf("\n Parent exited\n");
    }

    return 0;
}


child-1 executing  its pid = (<PID1>)
```

```
child-2 executing  its pid = (<PID2>)

Parent executing before wait() Parent pid is (<Parent_PID>)

Child2 exited

cpid returned is (<PID2>)
status is (256)    # status = 256.

Child1 exited

cpid returned is (<PID1>)
status is (0)      # status = 0.

Parent exited
```

Question 1:

Describe a situation where you have multiple child processes, and you need to wait for a specific child to finish. How would you achieve this using waitpid? Provide a code example in c to illustrate the implementation.

☞ ZOMBIE

If the child dies first, the kernel empties the process address space but retains the process table entry. The child is said to be in a zombie state.

The zombie is actually not a process at all, so canot be killed.
The only reason for a child to memain in the zombie state is the hope that the parent may eventually call wait or waitpid to pickup the exit status and clear the process table slot.

Example 1:

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

int main()
{

    pid_t id;
    printf("Parent Process : Executed by parent process before fork() - PID = (%d)
\n",getpid());
    id = fork();  // from this point of code, the child and parent process both
execute
    if (id < 0 ){
        printf("\nfork failed\n");
```

```
        exit(-1);
    }
    if(id > 0){ /* Parent process*/
        sleep(30);

    }else
    { /* Child process*/
        exit(0);
    }

    return 0;
}
```

Example 2:

```
/*
Demonstrate Zombie process
run command on terminal - ps aux | grep Z

Note: If a process is in Zombie state, this means a entry of that process is still
present in process table
*/

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

int main()
{

    pid_t id;
    printf("Parent Process : Executed by parent process before fork() - PID = (%d)
\n",getpid());
    id = fork();  // from this point of code, the child and parent process both
execute
    if (id < 0 ){
        printf("\nfork failed\n");
        exit(-1);
    }
    if(id > 0){ /* Parent process*/
        sleep(15);
        printf("\nParent Process: I have created child process withID = (%d)
\n",id);
        printf("\n Paren process exited\n");

    }else
    { /* Child process*/
        sleep(5);
        printf("\nchild process id is (%d)\n",getpid());
        printf("\nThe creator of child process is (%d)\n",getppid());
    }

    return 0;
}
```

How to identify Zombie ?

Open two terminal windows. Run the previous slide code in one terminal. On the other terminal run the command

(ps -la | grep CMD) ; (ps -la | grep a.out)

Look into the line that contains defaunct. It is the Zombie with process state code Z

```
 gcc pp.c //compile
 ./a.out & // Make the process run in the background //5864 (pid)
 pstree -p 5864    // a.out(5864) ------- a.out(5865)
 ps aux | grep 5865
```

After the program terminate, again type the command ps -la.
Is the ZOMBIE exit or not? If not process terminate.

man ps
to read process state information under the heading PROCESS STATE CODES

☞ ORPHAN

When the parent dies first, the child becomes an orphan since the parent is not just there to pickup the child’s exit status.

The kernel clears the process table slot of the parent, but before doing so, it checks whether there are any process spawned by the parent that are still alive. When it finds one , it makes init/ systemd its parent by changing PPID field of the child.

Example 1:

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

int main()
{

    pid_t id;
    printf("Parent Process : Executed by parent process before fork() - PID = (%d)
```

```
\n",getpid());
    id = fork();
    if (id < 0 ){
        printf("\nfork failed\n");
        exit(-1);
    }
    if(id > 0){ /* Parent process*/
        exit(0);

    }else
    { /* Child process*/
        sleep(60);
    }

    return 0;
}
```

How to identify Orphan ?

Open two terminal windows. Run the previous slide code in one terminal. On the other terminal run the command

```
(ps -le | grep CMD)
(ps -le | grep <ChildPID>)
```

Observe the process id (PID) and parent id (PPID) of the child and parent.

```
./a.out &    5889
pstree -p 5889   // done  ./a.out
pstree -p 5889
ps aux | grep a.out
ps -o ppid=5890  // 1
pstree -p 1 // childs of init process
```

Example 2:

```
/*
Demonstrate Orphan process (A Process that does not have a parent)
(Reparenting : Finding another parent for the process. Init process becomes parent)
*/

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

int main()
{

    pid_t id;
    printf("Parent Process : Executed by parent process before fork() - PID = (%d)
\n",getpid());
    id = fork();  // from this point of code, the child and parent process both
```

```
execute
    if (id < 0 ){
        printf("\nfork failed\n");
        exit(-1);
    }
    if(id > 0){ // Parent process, killing before child executes
        printf("\nParent process exited\n");
        return (0);

    }else
    { // Child process
        printf("\nChild process executing\n");
        sleep(10);
        printf("\nI am child process,  id value is (%d)\n",id) ;
        printf("\nchild process id is (%d)\n",getpid());
    }

    return 0;
}
```

QUESTION :

Example 1 :

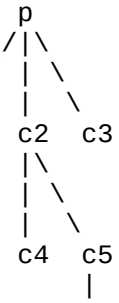
write a c code to generate the following correlated processes where arrow indicate parent child relation. Make sure no zombie process will not be created.

The wait(NULL) function is used in each parent process to wait for its child to terminate, ensuring no zombie processes are left.

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

void create_child(char name[]) {
    pid_t id = fork();

    if (id == 0) {

        printf("%s process: PID = %d, Parent PID = %d\n", name, getpid(),
getppid());
        exit(0); // Child terminates
```

```
    } else {

        wait(NULL); // Wait for child to terminate
    }
}

int main() {
    printf("P1 process: PID = %d\n", getpid());

    pid_t p2 = fork();   P1 creates P2

    if (p2 == 0) {

        printf("P2 process: PID = %d, Parent PID = %d\n", getpid(), getppid());

        pid_t p3 = fork();  P2 creates P3

        if (p3 == 0) {

            printf("P3 process: PID = %d, Parent PID = %d\n", getpid(), getppid());

            // P3 creates P4, P5, P6

            create_child("P4");
            create_child("P5");
            create_child("P6");

            exit(0);  P3 terminates after creating its children
        }
        else {
            wait(NULL); P2 waits for P3 to terminate
        }

        exit(0);  P2 terminates after creating P3
    }
    else {
        wait(NULL);  P1 waits for P2 to terminate
    }

    return 0;
}
```

Example 2:



```
              |
              c6
              |
              |
              c7
```

Develop a C code to create the following process tree. Display the process ID, parent ID and return value of fork() for each process.

Use ps utility to verify the is-a-parent relationship?

- Are ypu getting any orphan process case?
- Are you getting any ZOMBIE case?

Code :

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    // Process p creates c1, c2, c3
    pid_t c1, c2, c3;

    printf("p: PID=%d, PPID=%d, Fork()=%d\n", getpid(), getppid(), 0);

    c1 = fork();
    if (c1 == 0) {
        // Process c1
        printf("c1: PID=%d, PPID=%d, Fork()=%d\n", getpid(), getppid(), 0);
        exit(0);
    }

    c2 = fork();
    if (c2 == 0)
      {
        // Process c2 creates c4, c5
        printf("c2: PID=%d, PPID=%d, Fork()=%d\n", getpid(), getppid(), 0);

        pid_t c4, c5;

        c4 = fork();
        if (c4 == 0) {
            // Process c4
            printf("c4: PID=%d, PPID=%d, Fork()=%d\n", getpid(), getppid(), 0);
            exit(0);
        }

        c5 = fork();
        if (c5 == 0) {
            // Process c5 creates c6
            printf("c5: PID=%d, PPID=%d, Fork()=%d\n", getpid(), getppid(), 0);

            pid_t c6 = fork();
            if (c6 == 0) {
                // Process c6 creates c7
                printf("c6: PID=%d, PPID=%d, Fork()=%d\n", getpid(), getppid(), 0);

                pid_t c7 = fork();
                if (c7 == 0) {
```

```
                    // Process c7
                    printf("c7: PID=%d, PPID=%d, Fork()=%d\n", getpid(), getppid(),
0);
                    _exit(0);
                }
                wait(NULL); // Wait for c7 to finish
                exit(0);
            }

            wait(NULL); // Wait for c6 to finish
            exit(0);
        }

        wait(NULL); // Wait for c4 and c5 to finish
        exit(0);
    }

    c3 = fork();
    if (c3 == 0) {
        // Process c3
        printf("c3: PID=%d, PPID=%d, Fork()=%d\n", getpid(), getppid(), 0);
        exit(0);
    }

    // Wait for all child processes to finish
    wait(NULL);
    wait(NULL);
    wait(NULL);
    wait(NULL);
    wait(NULL);

    return 0;
}
```

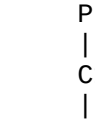
Analysis :

If the parent process (p) terminates before its child processes complete, those child processes may become orphaned. In the given code, the parent process (p) waits for all its immediate child processes to finish using the wait(NULL) calls.

However, if any of the child processes create additional processes and terminate without waiting for them, those processes may become orphaned. In the specific code, this scenario doesn't occur, as each child process (c1, c2, c3, c4, c5, c6, c7) immediately exits after printing its information.

Example 2 :

Create two different user-defined functions to generate the following process hierarchy shown in
Figure-(a) and Figure-(b). Finally all the processes display their process ID and parent ID.



```
    C2
  / |  \
C3  C4  C5
    |
    C6
    |
    C7
```

Fig(a)

Code :

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

// Function to create a child process and display process information
void createProcess(int childNumber) {
    pid_t pid = fork();

    if (pid == -1) {

        perror("fork");
        exit(1);
    }
   else if (pid == 0)
    {
        // Child process
        printf("Child %d: PID = %d, PPID = %d\n", childNumber, getpid(),
getppid());

        // Additional child processes creation
        if (childNumber == 1) {
            createProcess(2);
        } else if (childNumber == 2) {
            createProcess(3);
        } else if (childNumber == 4) {
            createProcess(6);
        }

        // Exit the child process
        exit(0);
    }
    else
   {
        // Parent process
        wait(NULL);  // Wait for the child process to complete
    }
}

int main()
{
    printf("Parent P: PID = %d\n", getpid());

    // Create the first child process (c1)
    createProcess(1);

    return 0;
```

```
}
```

☞ exec Family of System Calls

The exec family of functions replaces the current process image with a new process image.
The fork function creates a copy of the calling process, but many applications require the child process to execute code that is different from that of the parent.
The exec operation replaces the entire address space(text, data, and stack) with that of the new process.
Since the stack is also replaced, the call to exec family of functions donot return unless it results in an error.
The child executes (with an exec function) the new program while the parent continues to execute the original code.
All exec functions return -1 and set errno if unsuccessful.

exec Family Series

The exec: l series: l - a fixed list of arguments
The execl (execl, execlp and execle) functions pass the commandline arguments in an explicit list and are useful if the number of commandline arguments are known at compile time.

```
execl(argument list)
execle(argument list)
execlp(argument list
```

exec Family System Call Prototype

```
#include <unistd.h>
extern char **environ;


int execl(const char *path, const char *arg0, ... /*, char *(0) */);
int execle (const char *path, const char *arg0, ... /*, char *(0), char *const
envp[] */);
int execlp (const char *file, const char *arg0, ... /*, char *(0) */);
```

Example 1 : execl

execl: It takes a variable number of arguments where the arguments are explicitly listed in the function call.

Main Program : mp.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
```

```
int main()
{
        printf("\n I am main process pid = (%d)\n",getpid());

        execl("./np","np","arg1","arg2","arg3",NULL);  //Replaces the current
process image with a new process image specified by the given file path ("./np")

        printf("This line will not be printed... "); //execl replaces the current
process with a new one.

    return 0;
}
```

New Program : np.c

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

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

    printf("I am new program called by execl(), my pid is (%d) ",getpid());
    printf("\n");
    for(i = 0; i < argc; i++){
        printf("\n argv[%d] = (%s)\n",i,argv[i]);
    }
    return 0;
}
```

Compile the code :

```
gcc np.c -o np
gcc mp.c -o mp
```

Run the code :

```
./mp
```

Example 2 : execl

---

execv: It takes an array of pointers to null-terminated strings as its argument list.

Main Program : mp1.c

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

int main()
{
    pid_t cpid;
```

```
    int status = 0;
    int num  = 5;
    cpid = fork();
    if(cpid == -1)
        exit(0);           /* terminate child */

    if(cpid == 0){
        printf("\nBefore exec\n");
        execl("./np1","arg1","arg2",NULL);
        printf("\n line is not printed\n");
    }
    else{
        printf("\n Parent executing before wait(), child process created by parent
is = (%d)\n",cpid);
        cpid = wait(&status); /* waiting for child process to exit*/
        printf("\n wait() in parent done\nParent pid = %d\n", getpid());
        printf("\n cpid returned is (%d)\n",cpid);
        printf("\n status is (%d)\n",status);
    }

    return 0;
}
```

New Program : np1.c

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

int main(int argc, char *argv[])
{
    int i = 0;

    printf("I am new process called by execl() ");
    printf("new program pid  = (%d)\n",getpid());
    for(i = 0; i < argc; i++){
        printf("\n argv[%d] = (%s)\n",i,argv[i]);
    }
    sleep(20);
    return 0;
}
```

Example 3 : execl does not use the path environment variable so full path is needed.This program display all the current directory all its size.

```
------------------
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
  char *bin_path = "/bin/ls";
  char *arg1 = "-a";
  char *arg1 = "-s";
  execl(bin_path,p3,arg1,arg2,NULL);
```

```
  return 0;
}

gcc p3.c -o p3
./p3


Example 4 : execlp uses the PATH environment variable to search for the
executable..
------------------
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
  char *bin_path = "ls";
  char *arg1 = "-a";
  char *arg1 = "-s";
  execlp(bin_path,p4,arg1,arg2,NULL);

  return 0;
}

gcc p4.c -o p4
./p4

Example 5 : execlp : A program that creates a child process to run echo.

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(void)
{
char *temp1,*temp2;
temp1="Funny"; temp2="world";
pid_t pid;
pid=fork();
if(pid==0){
execlp("echo","echo",temp1,temp2,NULL);
printf("Error");
return 1;
}
else{
/* Parent code*/
}
return 0;
}


-------------------------------------------------------------------------------
---------
Questions:
-------------------------------------------------------------------------------
--------

Example 1:

Write a MulThree.c program to multiply three numbers and display the output. Now
```

write another C program PracticeExecl.c, which will fork a child process and the child process will execute the file MulThree.c and generate the output. The parent process will wait till the termination of the child and the parent process will print the process ID and exit status of the child.

Code :

MulThree.c

```
#include <stdio.h>

int main()
{
    int num1, num2, num3, result;
    printf("Enter three numbers: ");
    scanf("%d %d %d", &num1, &num2, &num3);
    result = num1 * num2 * num3;
    printf("Multiplication result: %d\n", result);
    return 0;
}
```

PracticeExecl.c

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

int main() {

    pid_t pid = fork();

    if (pid == -1) {
        // Forking failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }

    if (pid == 0)
   {

        printf("Child process executing MulThree.c:\n");

        // Execute MulThree.c using execl

        execl("./MulThree", "MulThree", NULL);

        // If execl fails
        perror("execl failed");
        exit(EXIT_FAILURE);
    }
   else
   {
        // Parent process
        int status;
```

```
        waitpid(pid, &status, 0);  // Wait for the child process to terminate
        printf("Parent process:\n");
        printf("Child process ID: %d\n", pid);

        if (WIFEXITED(status)) {
            // Child process exited normally
            printf("Child exit status: %d\n", WEXITSTATUS(status));
        }
        else {
            // Child process did not exit normally
            printf("Child process did not exit normally\n");
        }
    }

    return 0;
}
```

Compile the code :

```
gcc MulThree.c -o MulThree
gcc PracticeExecl.c -o PracticeExecl
```

Run the code :

```
./PracticeExecl
```

Example 2 :

The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8. Write a C program using the fork() system call that generates the Fibonacci sequence in the child process. The number of the sequence will be provided in the command line.

For example, if 5 is provided, the first five numbers in the Fibonacci sequence will be output by the child.

Code :

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

// Function to generate the Fibonacci sequence
void generateFibonacci(int n) {
    int first = 0, second = 1, next;

    printf("Fibonacci sequence for %d terms: ", n);

    for (int i = 0; i < n; ++i)
   {
        printf("%d ", first);
        next = first + second;
        first = second;
        second = next;
```

```
    }

    printf("\n");
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <number_of_terms>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);

    if (n <= 0) {
        fprintf(stderr, "Number of terms should be greater than 0\n");
        return 1;
    }

    pid_t pid = fork();

    if (pid == -1)
    {
        perror("Fork failed");
        return 1;
    }

    if (pid == 0)
   {          // Child process

        generateFibonacci(n);
    }

   else
   {           // Parent process
        wait(NULL);  // Wait for the child process to finish
        printf("Parent process is done.\n");
    }

    return 0;
}
```

INTER-PROCESS COMMUNICATION (PIPE AND FIFO) :

State the number of FDs will be opened for each of the process.

```
int main(void){
fork();
fork();
fprintf(stderr,"hello\n");
return 0;
}
```

Initial Process (Parent):
File descriptors: stdin (0), stdout (1), and stderr (2) by default, which are usually connected to the terminal.
4 processes × 3 file descriptors = 12 file descriptors.

Q1: Write the number of file descriptors will be opened for the following code snippet. Verify the descriptor numbers by exploring the fd folder for the process in the directory /proc/PID.

```
#include<stdio.h>
#include<unistd.h>
#include<errno.h>
int main(void)
{
int fd[2],fs[2],fds[2];
pipe(fd);
pipe(fs);
pipe(fds);
return 0;
}
```

The program creates multiple file descriptors (FDs) using pipe() system calls.

pipe(fd) creates a pair of file descriptors: one for reading (fd[0]) and one for writing (fd[1]).

Each pipe() call adds two new file descriptors.

Total FDs opened by the pipes:
2+2+2 =6
The program starts with the standard FDs:
0: stdin, 1: stdout, 2: stderr

Total FDs:3 (standard FDs) + 6 (from pipe()) = 9 FDs.

Each process has a directory in /proc named after its PID (process ID). Within this directory, the fd subdirectory contains symbolic links to all open file descriptors of the process.

Steps to Verify:
Run the program. Note the PID of the process. For instance, if the process ID is 12345:
./a.out &
echo $! Get the PID of the running process
cd /proc/<PID>/fd
ls -l List the file descriptors.This will display symbolic links for all open file descriptors

Q2 : Write the descriptor numbers attached to both parent and child process file descriptor table(PFDT).
Verify the descriptor numbers by exploring the fd folder for the process in the directory proc.

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

```
#include<unistd.h>
#include<sys/wait.h>
int main(void)
{
int fd[2],fs[2],fds[2];
pid_t pid;
pipe(fd);

pid=fork();

if(pid==0){
pipe(fs);
pipe(fds);
}
else{
wait(NULL);
printf("Parent waits\n");
}
return 0;
}
```

Analysis :

File Descriptor Table Analysis

Before fork():

The program starts with the three standard descriptors:0 → stdin, 1 → stdout, 2 → stderr
The pipe(fd) call creates two new descriptors:
3 → Read end of the first pipe (fd[0])
4 → Write end of the first pipe (fd[1])
At this point, parent and child processes share these descriptors.

After fork() :

Both the parent and child processes have identical file descriptor tables initially.
Changes made to the file descriptor table in one process do not affect the other.

In the child process:

Two new pipes are created (pipe(fs) and pipe(fds)):
5 → Read end of the second pipe (fs[0])
6 → Write end of the second pipe (fs[1])
7 → Read end of the third pipe (fds[0])
8 → Write end of the third pipe (fds[1])

In the parent process:

No new pipes are created. The parent process retains the original descriptors:
0, 1, 2 → Standard descriptors
3, 4 → Descriptors from the first pipe

Parent Process File Descriptor Table (PFDT): 0: stdin, 1: stdout, 2: stderr, 3: Read end of the first pipe (fd[0]), 4: Write end of the first pipe (fd[1])
Child Process File Descriptor Table (PFDT): 0 to 8

Verify :
./a.out &
pstree -p Shows the parent and child PIDs

cd /proc/<parent_PID>/fd
ls -l

cd /proc/<child_PID>/fd
ls -l

Q3 : Write the descriptor numbers attached to both parent and child process file descriptor table(PFDT).
Verify the descriptor numbers by exploring the fd folder for the process in the directory proc.

```
int main(void)
{
int fd[2],fs[2],fds[2];
pipe(fd);
pid_t pid=fork();

if(pid!=0){
pipe(fs);
pipe(fds);
}
else
{
wait(NULL);
printf("Parent waits\n");
}
return 0;}
```

Parent Process File Descriptor Table (PFDT):
0: stdin
1: stdout
2: stderr
3: fd[0] (read end of the first pipe)
4: fd[1] (write end of the first pipe)
5: fs[0] (read end of the second pipe)
6: fs[1] (write end of the second pipe)
7: fds[0] (read end of the third pipe)
8: fds[1] (write end of the third pipe)

Child Process File Descriptor Table (PFDT):
0: stdin
1: stdout
2: stderr
3: fd[0] (read end of the first pipe)
4: fd[1] (write end of the first pipe)

ps aux | grep <program_name> Identify the parent and child process PIDs.

For the parent process: ls -l /proc/<parent_PID>/fd

Q4: Develop a c program to write and read a message using pipe() that demonstrates inter-process communication (IPC). This program involves both the parent and child processes: the parent writes to the pipe, and the child reads from the pipe.

OR

Write a parent child code where the child receives a string from the user and shares it to the parent using pipe.

Parent process sent message: Hello from parent process!
Child process received message: Hello from parent process!

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
    int fd[2]; // File descriptors for the pipe
    pid_t pid; // Process ID
    char write_msg[] = "Hello from parent process!";
    char read_msg[100]; // Buffer to hold the message read by the child process

    pipe(fd);

    // Create a child process using fork
    pid = fork();

     if (pid == 0) {
        Child process

        close(fd[1]); // Close the write end of the pipe (child only reads)

        // Read the message from the pipe

        read(fd[0], read_msg, sizeof(read_msg));

        printf("Child process received message: %s\n", read_msg);

        close(fd[0]); // Close the read end of the pipe
    }
    else {
        Parent proces

        close(fd[0]); // Close the read end of the pipe (parent only writes)

        // Write a message to the pipe

        write(fd[1], write_msg, strlen(write_msg) + 1); // +1 to include null
terminator

        printf("Parent process sent message: %s\n", write_msg);

        close(fd[1]); // Close the write end of the pipe

        // Wait for the child process to finish
```

```
        wait(NULL);
    }

    return 0;
}
```

Q5. Develop a c program to write and read a message using pipe() for a single process.
(Hint: No need to use fork() and the main process will create and implement the pipe for both writing and reading.)

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
    int fd[2];
    char write_msg[] = "Hello from the pipe!";
    char read_msg[100];

    write(fd[1], write_msg, strlen(write_msg) + 1);

    read(fd[0], read_msg, sizeof(read_msg));

    printf("Message read from pipe: %s\n", read_msg);
    close(fd[0]);
    close(fd[1]);

    return 0;
}
```

Q6. Here, use the fork() system call to create a child process. The child process will write a message
into the pipe and the parent process will read the message from the pipe. The parent process will
display the message on stdout. Design a program to establish the communication using pipe between
the processes.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main() {
    int fd[2];
    pid_t pid;
    char write_msg[] = "Hello from child process!";
    char read_msg[100];

    pid = fork();

    if (pid == 0) {

        close(fd[0]);
```

```
        write(fd[1], write_msg, strlen(write_msg) + 1);

        printf("Child process sent message: %s\n", write_msg);

        close(fd[1]);
        return 0;
    }
    else {

        close(fd[1]);


        read(fd[0], read_msg, sizeof(read_msg));

        printf("Parent process received message: %s\n", read_msg);

        close(fd[0]);

        wait(NULL);
    }

    return 0;
}
```

FIFO

FIFO (Named Pipe): A named pipe allows two independent processes to communicate by writing and reading from the same file.

FIFO Creation: We use mkfifo() to create a named pipe (FIFO). The name of the FIFO is a special file in the file system that allows two processes to communicate.
Writing to FIFO: One process writes data to the FIFO using open() and write().
Reading from FIFO: The other process reads the data from the FIFO using open() and read().

Q1. Write a C code to create a FIFO and put a string on the FIFO. Now create another C code to read the content present in the FIFO and then unlink the FIFO file.

OR

Develop a program to communicate between two processes using a named pipe (FIFO). The program will demonstrate how data is written into a FIFO and how data is read from a FIFO. Implement FIFO in two aspects;

CASE-I: Between two different process (i.e. two independent processes).
CASE-II: Between parent process and child process (co-operative processes)

Case I: Communication Between 2 unrelated processes

Program 1 : fifowriter.c

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

#define FIFO_NAME "/tmp/myfifo"

int main()
{
int fd;
mkfifo(FIFO_NAME,0600);
fd=open(FIFO_NAME,O_WRONLY);
write(fd,"Writer\n",7);
close(fd);
return 0;
}

mkfifo /tmp/myfifo
gcc fifowriter.c -o fifowriter
./fifowriter
```

Program 2 : fiforeader.c

```
#include<stdio.h>
#include<unistd.h>
#include<sys/stat.h>
#include<fcntl.h>
#define FIFO_NAME "/tmp/myfifo"

int main()
{
int fd;
char buf[20];
fd=open(FIFO_NAME,O_RDONLY);
read(fd,buf,7);
write(1,buf,7);
close(fd);
unlink(FIFO_NAME);
return 0;
}

gcc fiforeader.c -o fiforeader
./fiforeader
```

Case II: Communication Between 2 related processes

The parent reads what its child has written to a named pipe.

```
int main (int argc, char *argv[])
{
pid_t childpid;
int fd,fd1;
char buf[20];
if (argc != 2) { /* command line has pipe name */
fprintf(stderr, "Usage: %s pipename\n", argv[0]);
return 1;
```

```
}
mkfifo(argv[1], 0600); /* create a named pipe */

childpid = fork();
if (childpid == 0){ /* The child writes */
fd=open(argv[1],O_WRONLY);
write(fd,"I am child\n",11);
}
else
{
fd1=open(argv[1],O_RDONLY);
read(fd1,buf,11);
write(1,buf,11);wait(NULL);
unlink(argv[1]);
}
return 0;
}
```

Q2 . Write a C code using pipe to simulate the shell pipe command “ls | sort -r”. First run on the command on the terminal and observe the output, and then create your code to meet the desired output.

This involves creating a pipe between two processes, where the output of the ls command in one process is used as the input to the sort -r command in the other process.

☞ Semaphore

Programs that manage shared resources must execute portions of code called critical sections in a mutually exclusive manner.

Semaphore : A semaphore is an integer whose value is never allowed to fall below zero.

Two operations can be performed on semaphores:

- Increment the semaphore value by one (sem_post)
- Decrement the semaphore value by one (sem_wait)

If the value of a semaphore is currently zero, then a sem_wait() operation will block until the value becomes greater than zero.

Other names for wait are down, P and lock.

Other names for signal are up, V, unlock and post.

POSIX:SEM Named Semaphores

☞ POSIX:SEM named semaphores can synchronize processes that do not share memory.

☞ Named semaphores have a name, a user ID, a group ID and permissions just as files do.

☞ A named semaphore is identified by a name of the form /somename. that is, a null-terminated string of characters (up to 251) consisting of an initial slash, followed by one ormore characters, none of which are slashes.

Creating and Opening Named Semaphores

```
#include <fcntl.h> /* For O_* constants */

#include <sys/stat.h> /* For mode constants */

#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag);

sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

Returns:
(1) If successful, the sem_open function returns the address of the semaphore.
(2) If unsuccessful,sem_open returns SEM_FAILED and sets errno.

Parameters of sem_open

☞ The name parameter is a string that identifies the semaphore by name.

☞ The oflag parameter determines whether the semaphore is created or just accessed by the function.

☞ The oflag parameter is either 0, O_CREAT, or O_CREAT | O_EXECL.

Named Semaphore getvalue

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);
```

Returns:
(1) If successful, sem_getvalue returns 0.
(2) If unsuccessful, sem_getvalue returns -1 and sets errno

Named Semaphore Wait Operation

```
#include <semaphore.h>

int sem_wait(sem_t *sem);

int sem_trywait(sem_t *sem);
```

Returns:
(1) If successful, these functions return 0.
(2) If unsuccessful, these functions return -1 and set errno.

Named Semaphore Signal Operation

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Returns:
(1) If successful, sem_post returns 0.
(2) If unsuccessful, sem_post returns -1 and sets errno.

Closing Named Semaphore

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Returns:
(1) If successful, sem_close returns 0.
(2) If unsuccessful, sem_close returns -1 and sets errno

Solved Some Problems :

Example 1 :

Two concurrent processes P and Q are accessing their critical sections by using boolean variables S and T as follows;

process P

```
while(true)
{
//Entry section ?
print(1);
print(1);
// Exit section ?
```

```
}

process Q

while(true)
{
//Entry section ?
print(0);
print(0);
// Exit section ?
}
```

Complete the entry section and exit section of process P and Q with suitable semaphore operations using the two boolean semaphores S and T. Also suggest the initial values of S and T, such that the execution of the processes will print the sequence 00110011....

Solution :

```
Semaphore S = 0;  // Initial value of semaphore S
Semaphore T = 1;  // Initial value of semaphore T

 while (true)
    {
        // Entry section
        wait(S);
        print(1);
        print(1);
        signal(T);
        // Exit section
    }

    while (true)
    {
        // Entry section
        wait(T);
        print(0);
        print(0);
        signal(S);
        // Exit section
    }
```

Analysis :

Here, wait(S) and signal(S) represent the wait and signal operations on semaphore S, and similarly for semaphore T. The wait operation (wait) decrements the semaphore value, and the signal operation (signal) increments the semaphore value.

By initializing S with 1 and T with 0, and using the semaphores as described, the processes will print the sequence 00110011... in a synchronized manner. The semaphore S ensures that process P prints '1' twice before allowing process Q to print '0' twice, and vice versa.

Example 2 :

Let 4 concurrent processes P1, P2, P3, P4 are accessing their critical sections by using Boolean semaphores S1, S2, S3 and S4. Write the entry section and exit section of all processing the semaphores with suitable initialization, such that P1 will complete its critical section before P2 and P3, P2 and P3 will complete their critical section in any order before P4.

solution :

```
# Initialization
S1 = Semaphore(1)  # Binary semaphore for P1
S2 = Semaphore(0)  # Initial value set to 0
S3 = Semaphore(0)  # Initial value set to 0
S4 = Semaphore(0)  # Initial value set to 0

# Process P1

# Entry Section
S1.wait()  # Decrement S1, blocking if necessary
# Critical Section for P1
# Exit Section
S2.signal()  # Increment S2 to allow P2 or P3 to proceed

# Process P2

# Entry Section
S2.wait()  # Decrement S2, blocking if necessary
# Critical Section for P2
# Exit Section
S3.signal()  # Increment S3 to allow P3 or P4 to proceed

# Process P3

# Entry Section
S2.wait()  # Decrement S2, blocking if necessary
# Critical Section for P3
# Exit Section
S3.signal()  # Increment S3 to allow P2 or P4 to proceed

# Process P4

# Entry Section
S3.wait()  # Decrement S3, blocking if necessary
# Critical Section for P4
# Exit Section
S4.signal()  # Increment S4 to release any waiting processes
```

Example 3 :

Implement the following pseudocode over the semaphores S and Q. State your answer on different initializations of S and Q.

(a) S=1, Q=1
(b) S=1, Q=0

(c) S=0, Q=1
(d) S=0, Q=0
(e) S=8, Q=0

```
Process 1 executes:                    Process 2 executes:
for( ; ; ) {                             for( ; ; ) {
    wait(&S);                                wait(&Q);
    a;                                       b;
    signal(&Q);                              signal(&S);
}                                        }
```

Solution :

(a) S=1, Q=1
In this case, both processes can execute their critical sections without any issue because initially, both S and Q are set to 1.

(b) S=1, Q=0
Here, Process 2 will be blocked initially because it needs to wait for Q to become 1. Process 1 can execute its critical section without any issue.

(c) S=0, Q=1
Similarly, Process 1 will be blocked initially because it needs to wait for S to become 1. Process 2 can execute its critical section without any issue.

(d) S=0, Q=0
In this scenario, both processes will be blocked initially as they both need the corresponding semaphore to be 1 to proceed.

(e) S=8, Q=0
Here, Process 2 will be blocked initially because it needs to wait for Q to become 1. Process 1 can execute its critical section up to 8 times (due to the initial value of S) before it gets blocked.

Example 4 :

Implement and test what happens in the following pseudocode if semaphores S and Q are both initialized to 1?

process 1 executes:

```
for( ; ; )
{
wait(&Q);
wait(&S);
a;
signal(&S);
signal(&Q);
}
```

process 2 executes:

```
for( ; ; )
{
```

```
wait(&S);
wait(&Q);
b;
signal(&Q);
signal(&S);
}
```

Solution :

Example 5 :

Process ordering using semaphore. Create 6 number of processes (1 parent + 5 children) using fork() in if-elseif-else ladder. The parent process will be waiting for the termination of all it’s children and each process will display a line of text on the standard error as;

P1: Coronavirus
P2: WHO:
P3: COVID-19
P4: disease
P5: pandemic

Your program must display the message in the given order :

WHO: Coronavirus disease COVID-19 pandemic.

Example :

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


int main(){

    int ret, sval;
    sem_t *sem;

    sem = sem_open("/sem1",  O_RDWR);

    if (sem == SEM_FAILED){
        perror("sem_open fail;");
        return -1;
    }else{
        printf("\n sem_open success\n");
    }

   sem_getvalue(sem , &sval );
   printf("\n semaphore val = (%d)\n",sval);

    ret = sem_wait(sem); //wait state
    printf("\nProcess 2 executing critical section\n");
```

```
    sleep(10);
    printf("\n ret is (%d)\n",ret);
    printf("\nprocess 2:\n");
    sem_post(sem);

    printf("\nProcess 2 executed critical section\n");
    //sem_unlink("/sem1");

}


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


int main(){

    int ret, sval;
    unsigned int value;
    sem_t *sem;

    sem = sem_open("/sem1",  O_RDWR);

    if (sem == SEM_FAILED){
        perror("sem_open fail;");
        return -1;
    }else{
        printf("\n sem_open success\n");
    }

   sem_getvalue(sem , &sval );
   printf("\n sval = (%d)\n",sval);

    ret = sem_wait(sem); //wait state
    printf("\n ret is (%d)\n",ret);
    printf("\nprocess 3:\n");
    sem_post(sem);

    //sem_unlink("/sem1");

}
```

Question :

Create a C program using named semaphores to display messages in a specific sequence with two processes (parent and child). The parent prints ITER and SUM, while the child prints SOA and CSE independently. Set the initial values of semaphore1 and semaphore2 to 0. Output sequence: ITER SOA SUM CSE

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

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

int main() {
    pid_t pid;
    sem_t *semaphore1, *semaphore2;

    // Create and initialize named semaphores

    semaphore1 = sem_open("/semaphore1", O_CREAT, 0644, 0);
    semaphore2 = sem_open("/semaphore2", O_CREAT, 0644, 0);

    pid = fork();

    if (pid == 0) {

        sem_wait(semaphore1); // Wait until semaphore1 is signaled
        printf("SOA\n");
        sem_post(semaphore2); // Signal semaphore2 for parent to print SUM

        sem_wait(semaphore1); // Wait again for semaphore1 to be signaled
        printf("CSE\n");
        sem_post(semaphore2); // Signal semaphore2 to indicate completion
    } else {

        printf("ITER\n");
        sem_post(semaphore1); // Signal semaphore1 for child to print SOA

        sem_wait(semaphore2); // Wait until semaphore2 is signaled by the child
        printf("SUM\n");
        sem_post(semaphore1); // Signal semaphore1 for child to print CSE

        sem_wait(semaphore2); // Wait for the child to finish CSE
    }

    // Cleanup named semaphores
    sem_close(semaphore1);
    sem_close(semaphore2);
    sem_unlink("/semaphore1");
    sem_unlink("/semaphore2");

    return 0;
}
```

Sample 2:

```
/* FOR SEMAPHORE */
#include <semaphore.h>
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */

#include <stdlib.h>
#include <stdio.h>

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

```
#include <sys/wait.h>

#define SEM_NAME_1 "/sem_1"
#define SEM_NAME_2 "/sem_2"

void main() {
    sem_t *sem1 = NULL, *sem2 = NULL;

    sem1 = sem_open(SEM_NAME_1, O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 0);
    sem2 = sem_open(SEM_NAME_2, O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 0);

    if (fork() == 0)
    {
        printf("1\n");
        sem_post(sem1);

        sem_wait(sem2);
        printf("3\n");
        sem_post(sem1);

        sem_close(sem1);
        sem_close(sem2);

    } else
       {
        sem_wait(sem1);
        printf("2\n");
        sem_post(sem2);

        sem_wait(sem1);
        printf("4\n");

        wait(NULL);

        sem_close(sem1);
        sem_close(sem2);

        sem_unlink(SEM_NAME_1);
        sem_unlink(SEM_NAME_2);
    }

}
```