WEEK – II Processes (I)

1. Useful system commands to deal with processes

a) ❯ `ps`

ps (Process status) can be used to see/list all the running processes.

b) ❯ `ps -f`

For more information -f (full) can be used along with ps

c) ❯ `pidof <proc_name>`

For a running program named ***`<proc_name>`*** lists the process ids

Fields described by ps are described as:

UID: User ID that this process belongs to (the person running it)
PID: Process ID
PPID: Parent process ID (the ID of the process that started it)
C: CPU utilization of process
STIME: Process start time
TTY: Terminal type associated with the process
TIME: CPU time is taken by the process
CMD: The command that started this process

There are other options which can be used along with ps command :

-a: Shows information about all users
-x: Shows information about processes without terminals
-u: Shows additional information like -f option
-e: Displays extended information

d) ❯ `top`

This command is used to show all the running processes within the working environment of Linux.

e) ❯ `kill pid`

For processes running in background kill command can be used if it's pid is known.

f) ❯ `bg`

A job control command that resumes suspended jobs while keeping them running in the background

g) ❯ `fg`

It continues a stopped job by running it in the foreground.

2. Creating a process

Creating a process using **`fork()`** system call

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

int main() {
    fork();
 //   fork();
//    fork();
    printf("hello\n");
    return 0;
}
```

a) Understanding process replication

Question: How many times will ***hello*** be printed on uncommenting the lines?

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

int main(){
   pid_t pid;
   int vble = 101;
   printf("Before fork \n");

   pid = fork();

   printf("pid = %d, process_id = %d, parent_process_id = %d, variable value = %d
\n", pid, getpid(), getppid(), ++vble);//exit(0);
   return 0;
}
```

b) Understanding Memory Space requirements

Question: What value is printed by **`++vble`** and how many times?

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

void forkexample() {
    pid_t pid;
    printf("Before fork \n");
    printf("pid = %d, process_id = %d, parent_process_id = %d, \n\n", pid, getpid(),
getppid());

    pid = fork(); // replicates current process
    printf("After fork \n");

    if ( pid == 0) {// child process because return value zero
        printf("Hello from Child!\n");
        printf("fork = %d, process_id = %d, parent_process_id = %d, \n", pid,
getpid(), getppid());
    }

    else if (pid > 0) {// parent process because return value non-zero and not -1.
        printf("Hello from Parent!\n");
        printf("fork = %d, process_id = %d, parent_process_id = %d, \n", pid,
getpid(), getppid());
    }

    else // return value negative
    printf("fork failed");
}

int main()
{
    forkexample();
    return 0;
}
```

c) Understanding Parallel Execution of parent and child

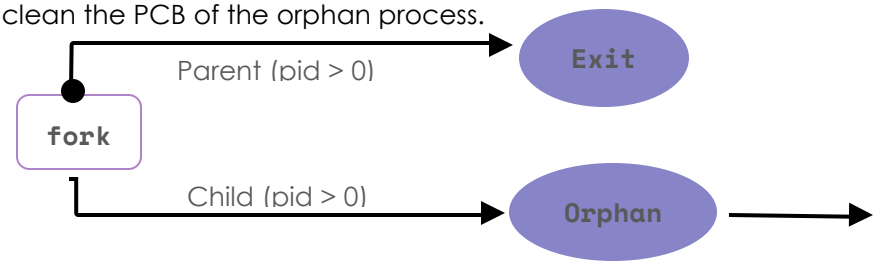
`fork()` On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and **`errno`** is set appropriately.

`getpid()` returns the process id of the current process

`getppid()` returns the process id of the parent of the current process

3. Orphan Process

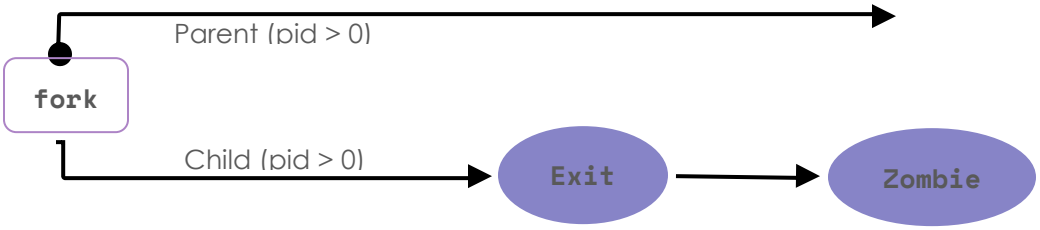
Parent does not invoke **wait()** but executes along with child process. In such a scenario when parent process terminates before the child process, the child process becomes **orphan**. Orphan process is a process that doesn't have a parent process. This can happen when the parent process terminates due to some reasons before the completion of the child.
An orphan process gets a new parent. The kernel detects that a process has become orphan and tries to provide a new parent to the orphan process. In most cases, the new parent is the INIT process, one with the PID 1. The new parent waits for the completion of the child (orphan process) and then asks the kernel to clean the PCB of the orphan process.




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

int main(){
    pid_t pid;
    if ((pid = fork()) > 0) { // parent process
      printf("I am Parent, my ID %d, I have child with ID: %d, My ParentID = %d\n\n",
getpid(), pid, getppid());
      exit(10);
   }

   else if (pid == 0) { // child process
     printf("I am Child, my ID %d, MY PARENT ID = %d\n\n",getpid(),getppid());
//getppid() reports different
     sleep(4); // child is waiting for some time. by this time parent process
finishes
     printf("I am Child, my ID %d, MY PARENT ID =
%d\n\n",getpid(),getppid());//getppid() reports different
     exit(0);
   }

   else printf("Problem in child creation....\n");
}
```

4. Zombie Process

Parent does not invoke **wait()** but executes along with child process. In such a scenario when child process terminates before the parent process, it creates a **zombie**. Zombie process is that process which has terminated but whose process control block has not been cleaned up from main memory because the parent process was not waiting for the child. If there are a large number of zombie processes, their PCBs IN WORST CASE can occupy the whole RAM.




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

int main(){
  pid_t pid;
  printf("Parent Process Id: %d\n",getpid());
  pid=fork();

  if (pid==0) { /* child process */
   printf("Child Process Id: %d and my Parent Process Id: %d \n",getpid(),
getppid());
   exit(0); // child made to exit
  }

  //parent process
  while(1) { //infinite excution
    sleep(1);
    printf("\nParent Not invoked Wait()!");
  }

  /* parent will keep waiting without knowledge of child being made to exit.
  as a result it still has an entry in the process table even though it has finished
execution.
  This creates a resource leak.*/

  return 0;
}
```

5. Shared memory space using `vfork()`

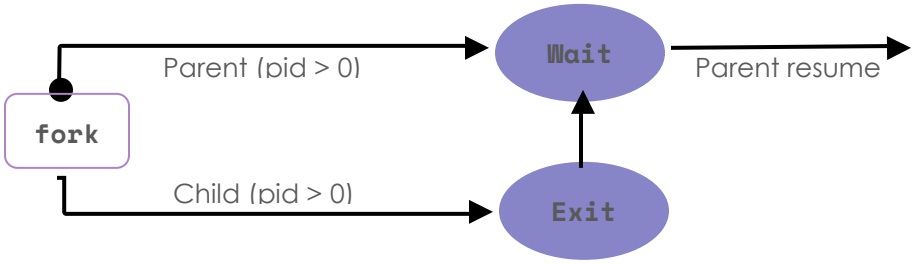
```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int globalvble = 10;

int main() {
    pid_t returnval;
    int localvble = 20;
    int childretval; // from child process: user provided return value
    int exitstatus, i=0;  // parent process : child's exit status

    if ((returnval=vfork()) >= 0) {    // including actions to be performed by both
parent and child processes
    if (returnval == 0) { // inside child
        printf("CHILD:: fork() returnValue = %d, My PID = %d, my Parent's ID =
%d\n",returnval,getpid(), getppid());        //globalvble++; localvble++;
        while (i < 10) {
            printf("CHILD:: i = %d, local vble = %d, global vble =
%d\n\n",i,localvble,globalvble);
            ++localvble; ++globalvble;
            ++i;
        }
//      system("ps afxj");
        printf("CHILD:: Enter an exit value (0 to 255) \n");
        scanf("%d", &childretval);
        printf("CHILD:: Good Bye\n\n");
        //exit(0);
        exit(childretval);
    }
```

```
else if (returnval > 0) { // inside parent
        sleep(2);
        printf("PARENT:: My PID = %d, my Child's PID = %d, my parent's PID = %d
\n",getpid(), returnval, getppid());
        printf("PARENT:: i = %d, local vble = %d, global vble =
%d\n\n",i,localvble,globalvble);
        while (i < 10) {
            printf("PARENT:: local vble = %d, global vble =
%d\n\n",localvble,globalvble);
            ++localvble; ++globalvble;
            ++i;
        }
        wait(&exitstatus);
        printf("PARENT:: Child's exit code is -> %d \n",WEXITSTATUS(exitstatus));
        printf("PARENT:: Good Bye\n\n");
        exit(0); // parent process is terminating -- normally
    }
    } // outer-if complete....
    else {
        printf ("Child creation error....");
        exit(0);
    }
}
```

Predict the output from the print statements.

6. wait() ing by the parent

Can avoid unwanted situations like orphan and/or zombie




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

int globalvble = 10;

int main() {
    pid_t returnval;
    int localvble = 20;
    int childretval; // from child process: user provided return value
    int exitstatus, i=0;  // parent process : child's exit status

    if ((returnval=fork()) >= 0) {    // including actions to be performed by both parent and child processes
        if (returnval == 0) { // inside child
            printf("CHILD:: fork() returnValue = %d, My PID = %d, my Parent's ID = %d\n",returnval,getpid(), getppid());        //globalvble++; localvble++;
            while (i < 10) {
                printf("CHILD:: i = %d, local vble = %d, global vble = %d\n\n",i,localvble,globalvble);
                ++localvble; ++globalvble;
                ++i;
            }
    //      system("ps afxj");
            printf("CHILD:: Enter an exit value (0 to 255) \n");
            scanf("%d", &childretval);
            printf("CHILD:: Good Bye\n\n");
            //exit(0);
            exit(childretval);
        }
```

```
else if (returnval > 0) { // inside parent
            sleep(2);
            printf("PARENT:: My PID = %d, my Child's PID = %d, my parent's PID = %d
\n",getpid(), returnval, getppid());
            printf("PARENT:: i = %d, local vble = %d, global vble =
%d\n\n",i,localvble,globalvble);
            while (i < 10)        {
                printf("PARENT:: local vble = %d, global vble =
%d\n\n",localvble,globalvble);
                ++localvble; ++globalvble;
                ++i;
            }
            sleep(2);
            wait(&exitstatus);
            printf("PARENT:: Child's exit code is -> %d \n",WEXITSTATUS(exitstatus));
            printf("PARENT:: Good Bye\n\n");
            exit(0); // parent process is terminating -- normally
        }
    } // outer-if complete....
    else {
        printf ("Child creation error....");
        exit(0);
    }
}
```

Compare the results of the last two programs and state the reason for their difference in behaviour.

7. Classwork

a) Write a program that accepts an integer **num** as command line argument. In your program create two processes. One process should report the **num!** (Factorial of **num**). The other process should report the summation of all factorials till **num**. [eg: if the argument is 5, one process calculates 5!=120, another process calculates 1!+2!+3!+4!+5!=153]. *Each process should report its pid, ppid, and child id.*

b) Write a program to create **exactly 3 processes**. Each process should then display its own pid along with the pid of the other two processes and tis relation with the other two processes [grandparent/parent/child, parent/child-parent/child, parent/child/grandchild]. Take care so that you don't have any unwanted orphan/zombie process(es). *Each process should report its pid, ppid, and child id.*

8. Home Assignment

a) Write a program that accepts two integers (**`low, high`**) as command line argument. Create two processes in your program. The first process should calculate the summation of all integers between (**`low, high`**) as **`sum_res`**. The second process should evaluate whether **`sum_res`** is prime or not. *Each process should report its pid, ppid, and child id.*

b) Write a program to create **exactly 3 processes**. The program should accept two integers (min and max) as command line arguments. The first process should report the even and odd numbers within the range of min-max. The second process should report the prime and non-prime numbers within the range min-max. The third process should report the summation of even numbers and prime numbers within the range min-max.