Operating System Lab

Exp 6: Processes

1. *Objectives:*

- Learn how to create a new child process.
- Becoming familiar with the processes-related system calls.
- Coordinate the completion of the child process with the original program parent process

2. *Process Identification*

The *pid_t* data type represents process IDs, which is a signed integer type (int). You can get the process ID of a process by calling getpid().

```
pid_tgetpid (void)
```

The getpid() returns the process ID of the calling process.

The function getppid() returns the process ID of the parent of the current process (this is also known as the parent process ID).

```
pid_tgetppid (void)
```

The getppid() function returns the process ID of the parent of the calling process. In order to use getpid() and getppid() Your program should include thefollowing libraries:

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

3. *Processes Creation*

The fork function is the primitive for creating a process. It is declared in the header file "unistd.h".

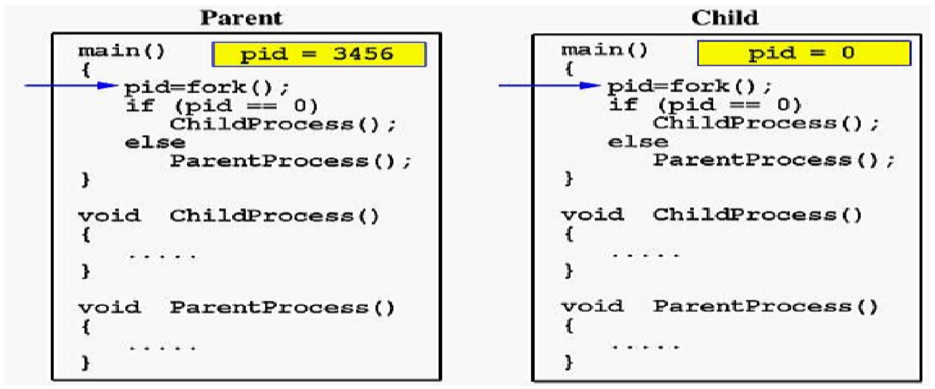
```
pid_t    fork (void)
```

The fork function creates a new process. If the operation is successful, there are, then both parent and child processes and both see fork return, but with different values: it returns a value of 0 in the child process and returns the child's process ID in the parent process. If process creation failed, fork returns a value of -1 in the parent process and no child is created.

```
/*************************************
Creating Processes
File Name: Test6.c
**************************************/
```

```
#include <stdio.h>
#include <unistd.h>
void main()
{
int pid;
printf("Hello World!\n");
printf("I am the parent process and pid is : %d .\n",getpid());
printf("Here I'm before use of forking\n");
pid = fork();
printf("Here I am just after forking\n");
if (pid == 0)
printf("I am the child process and pid is :%d.\n",getpid());
else
printf("I am the parent process and pid is: %d .\n",getpid());
}
```

When the main program executes fork(), an identical copy of its address space, including the program and all data, is created. System call fork() returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable pid. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.



```
main()
{
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess()
{
    .....
}

void  ParentProcess()
{
    .....
}
```

4. *Process Completion*

If the parent process wants to wait, its child to terminate the wait system call is used. The ***wait()*** system call suspends execution of the current process until one of its children terminates. The ***waitpid()*** system call suspends execution of the current process until a child specified by pid argument has changed state.

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);

The ***exit()*** system call terminates the process which calls this function and returns the exit status value. Both UNIX and C (forked) programs can read the status value. By convention, a status of 0 means normal termination. Any other value indicates an error or unusual occurrence. Many standard library calls have errors defined in the sys/stat.h header file.

void exit (int status);

A process may suspend for a period using the sleep command.

unsigned int sleep (seconds);

```
/***************************************
Process Completion
File Name: Test7.c
***************************************/
#include <stdio.h>
#include <sys/wait.h> /* contains prototype for wait */
int main(void)
{
int pid;
int status;
printf("Welcome to OS Lab !\n");
pid = fork( );
if (pid == -1) /* check for error in fork */
{
perror("bad fork");
exit(1);
}
if (pid == 0)
printf("I am the child process.\n");
else
{
wait(&status); /* parent waits for child to finish */
printf("I am the parent process.\n");
}
}
```

5. *Executing a file*

A child process can execute another program using one of the ***exec*** functions. The program that the process is executing is called its process image. Starting execution of a new program causes the process to forget all about its previous process image; when the new program exits, the process exits too, instead of returning to the previous process image. ***exec*** functions differ in how you specify the arguments, but otherwise they all do the same thing. They are declared in the header file `"unistd.h"`.

int execl (const char *filename, const char *arg0,...)

e.g.
execlp("ls", "ls", "-l", NULL)