Assignment no -02
Title : Process control system calls

The demonstration of *FORK, EXECVE* and *WAIT* system calls along with zombie and orphan states.

Problem statement :

A] Implement the C program in which main program accepts the integers to be sorted. Main program uses the *FORK* system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using *WAIT* system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.

B] Implement the C program in which main program accepts an integer array. Main program uses the *FORK* system call to create a new process called a child process. Parent process sorts an integer array and passes the sorted array to child process through the command line arguments of *EXECVE* system call. The child process uses *EXECVE* system call to load new program that uses this sorted array for performing the binary search to search the particular item in the array.

OBJECTIVE:

1. Study how to create a process in UNIX using fork() system call.
2. Study how to use wait(), execve() system calls, zombie, daemon and orphan states.

Theory :

fork():

It is a system call that creates a new process under the UNIX operating system. It takes no arguments. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

- If fork() returns a negative value, the creation of a child process was unsuccessful.
- fork() returns a zero to the newly created child process.
- fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.

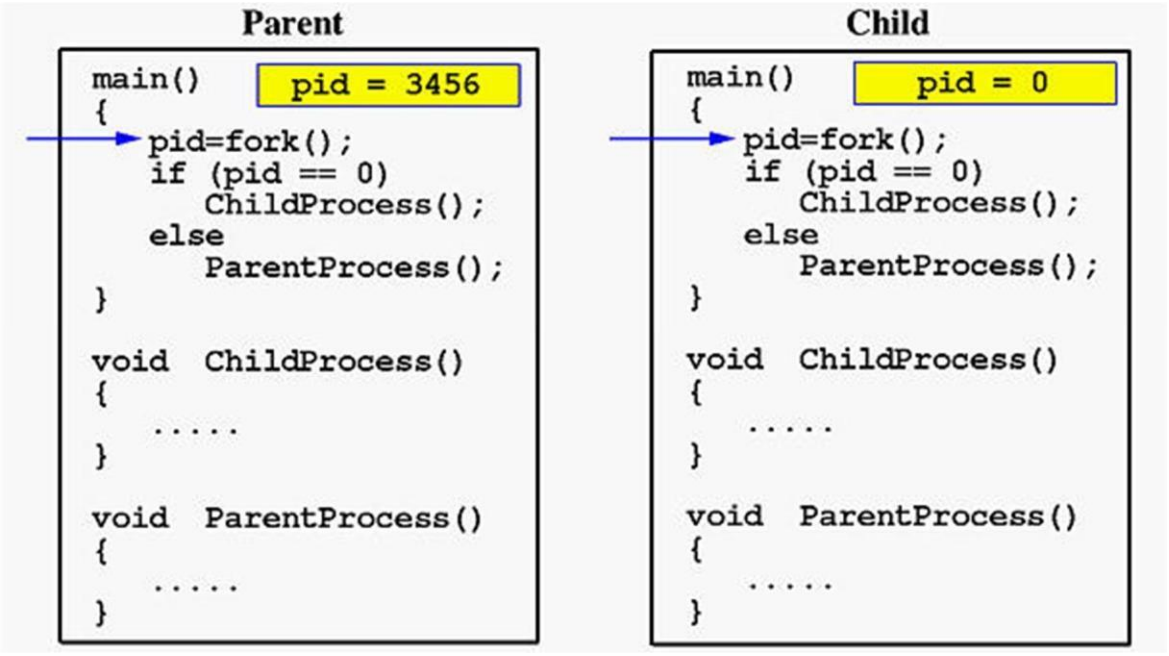
Therefore, after the system call to fork(), a simple test can tell which process is the child. Note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

Consider one simpler example, which distinguishes the parent from the child.

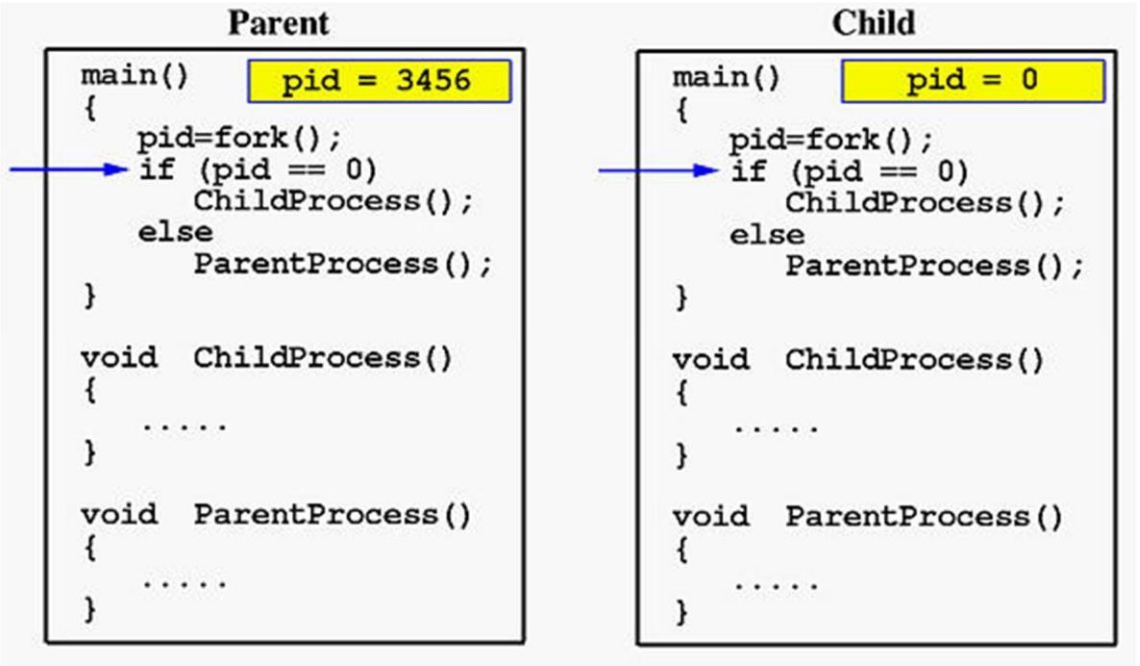
```
#include <stdio.h>
#include <sys/types.h>
void ChildProcess();        /* child process prototype */
void ParentProcess();       /* parent process prototype */
int main()
{
          pid_t  pid;
          pid = fork();
          if (pid == 0)
         ChildProcess();
        else
        ParentProcess();
        return 0;
}
void ChildProcess()
{
}
void ParentProcess()
{
}
```

In this program, both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and (2) the value of variable i.

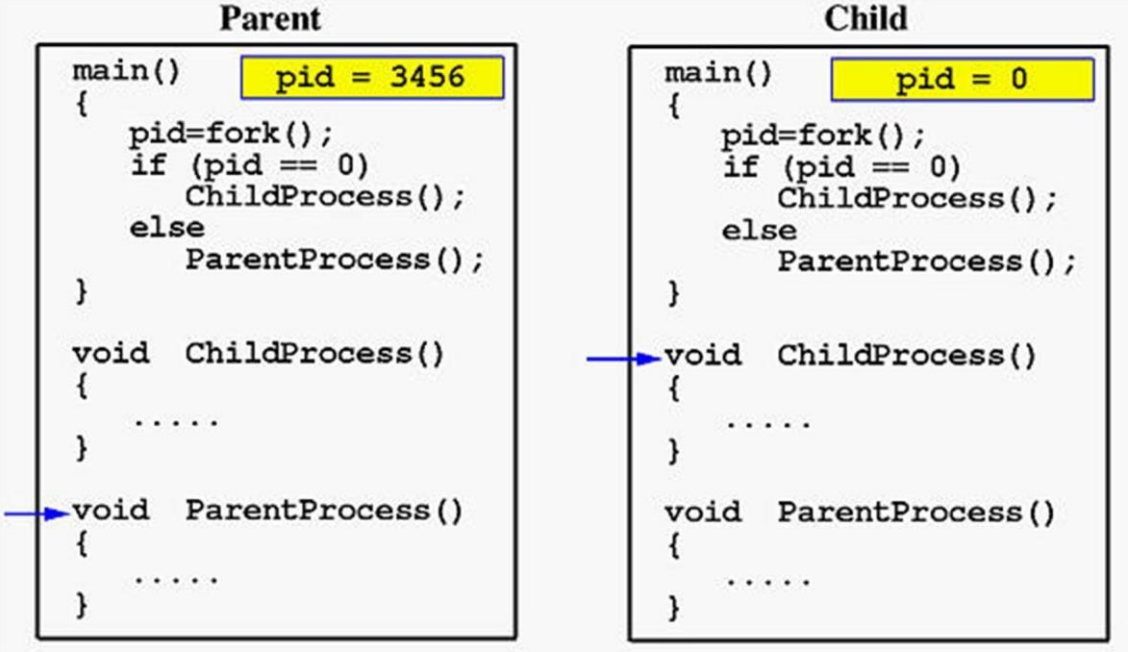
When the main program executes fork(), an identical copy of its address space, including the program and all data, is created. System call fork() returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable pid.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



Now both programs (i.e., the parent and child) will execute independent of each other starting at the next statement:



In the parent, since pid is non-zero, it calls function parentprocess(). On the other hand, the child has a zero pid and calls childprocess() as shown below:



Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process.

ps command:

The ps command shows the processes we're running, the process another user is running, or all the processes on the system. E.g.

$ ps –ef

By default, the ps program shows only processes that maintain a connection with a terminal, a console, a serial line, or a pseudo terminal. Other processes run without needing to communicate with a user on a terminal. These are typically system processes that Linux uses to manage shared resources. We can use ps to see all such processes using the -e option and to get "full" information with -f.

The wait() system call:

It blocks the calling process until one of its child processes exits or a signal is received. wait() takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer. One of the main purposes of wait() is to wait for completion of child processes.

The execution of wait() could have two possible situations.

1. If there are at least one child processes running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.

2. If there is no child process running when the call to wait() is made, then this wait() has no effect at all. That is, it is as if no wait() is there.

Zombie Process:

When achild process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait. The child process entry in the process table is therefore not freed up immediately. Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait. It becomes what is known as defunct, or a zombie process.

Orphan Process:

An orphan process is a computer process whose parent process has finished or terminated, though itself remains running. A process may also be intentionally orphaned so that it becomes detached from the user's session and left running in the background; usually to allow a long- running job to complete without further user attention, or to start an indefinitely running service. Under UNIX, the latter kinds of processes are typically called daemon processes. The UNIX nohup command is one means to accomplish this.

Daemon Process:

It is a process that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes.

exec() system call

The exec() system call is used after a fork() system call by one of the two processes to replace the memory space with a new program. The exec() system call loads a binary file into memory (destroying image of the program containing the exec() system call) and go their separate ways. Within the exec family there are functions that vary slightly in their capabilities.

1) execl() and execlp():

execl(): It permits us to pass a list of command line arguments to the program to be executed. The list of arguments is terminated by NULL. e.g.

execl("/bin/ls", "ls", "-l", NULL);

execlp(): It does same job except that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used. The function execlp() can also take the fully qualified name as it also resolves explicitly. e.g.

execlp("ls", "ls", "-l", NULL);

2) execv() and execvp():

execv(): It does same job as execl() except that command line arguments can be passed to it in the form of an array of pointers to string. e.g.

char *argv[] = ("ls", "-l", NULL);

execv("/bin/ls", argv);

execvp(): It does same job expect that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used.

execve() executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form "#! interpreter [arg]".

int execve(const char *****filename*, char *const** *argv***[],**
char *const *envp***[]);**

argv is an array of argument strings passed to the new program. *envp* is an array of strings, conventionally of the form **key=value**, which are passed as environment to the new program. Both *argv* and *envp* must be terminated by a null pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as **int main(int argc, char *argv[], char *envp[])**.

Conclusion – Thus we have scuccessfully studied process control system calls**.** fork() is to create a new process, which becomes the child process of the caller. Wait system call blocks the calling process until one of its child processes exits or a signal is received. execve() executes the program pointed to by filename.

References:

1. Beginning Linux Programming by Neil Mathew and Richard Stones, Wrox Publications.

2. Unix Concepts and Applications By Sumitabha Das, Tata McGraw Hill