NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
ISLAMABAD

OPERATING SYSTEMS LAB SPRING 2021

Lab Manual 02

Process Creation

fork, wait, exit, getpid and getppid Unix System Calls

shehr bano
shehr.bano@nu.edu.pk

1 COMPILE AND EXECUTE A C PROGRAM

Write down the C code in a text editor of your choice and save the file using **.c** extension i.e. *filename.c*. Now through your terminal move to the directory where *filename.c* resides.

```
gcc firstprogram.c
```

It will compile the code and by default an executable named *a.out* is created.

```
gcc -o firstprogram firstprogram.c
```

If your file is named firstprogram.c then type *-o firstprogram* as the parameter to gcc. It would create an executable by the name *firstprogram* for the source code named *firstprogram.c* .
To execute the program simply write the following:

```
./a.out   OR   ./firstprogram
```

In case you have written the code in C++ saved using .cpp extension, compile the code using g++ as:

```
g++ filename.cpp   OR   g++ -o exec_name filename.cpp
```

And execute as ./a.out or ./exec_name

2 Process Creation

When is a new process created?

1. System startup
2. Submit a new batch job/Start program
3. Execution of a system call by process
4. A user request to create a process

On the creation of a process following actions are performed:

1. Assign a unique process identifier. Every process has a unique process ID, a non-negative integer. Because the process ID is the only well-known identifier of a process that is always unique, it is used to guarantee uniqueness.
2. Allocate space for the process.
3. Initialize process control block.
4. Set up appropriate linkage to the scheduling queue.

3 Fork System Call

An existing process can create a new process by calling the fork function.

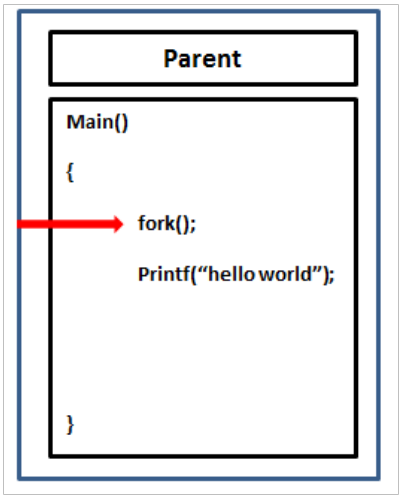
```
#include <unistd.h>
pid_t fork(void);
// Returns: 0 in child, process ID of child in parent, -1 on error
```

The definition of the pid_t is given in <sys/types> include file and <unistd.h> contain the declaration of *fork* system call.

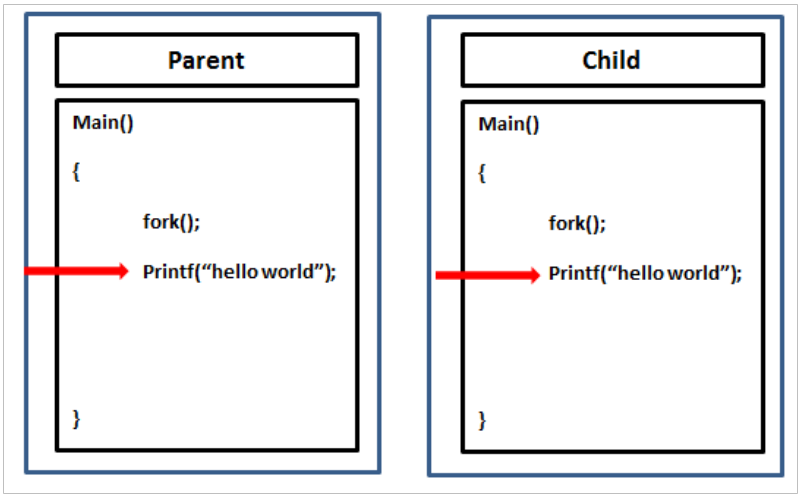
Important Points

1. The new process created by fork is called the child process.
2. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
3. Both the child and the parent continue executing with the instruction that follows the call to fork.
4. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child; the parent and the child do not share these portions of memory.

5. In general, we never know whether the child starts executing before the parent, or vice versa. The order depends on the scheduling algorithm used by the kernel.

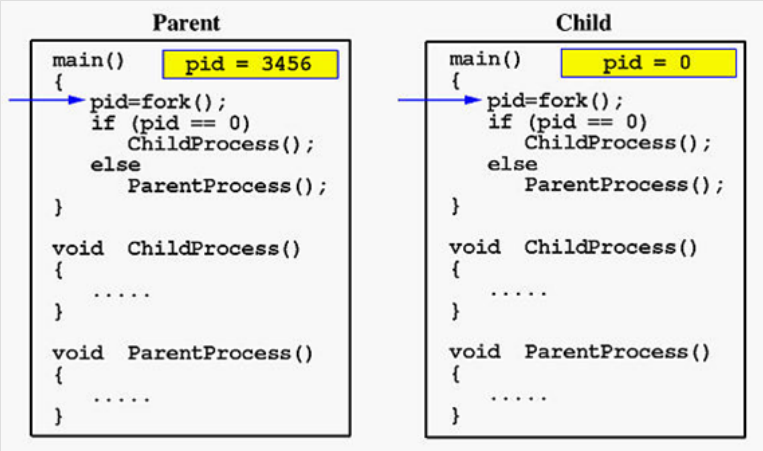


(a) The process will execute sequentially until it arrives at fork system call. This process will be named as parent process and the one created using fork() is termed as child process.

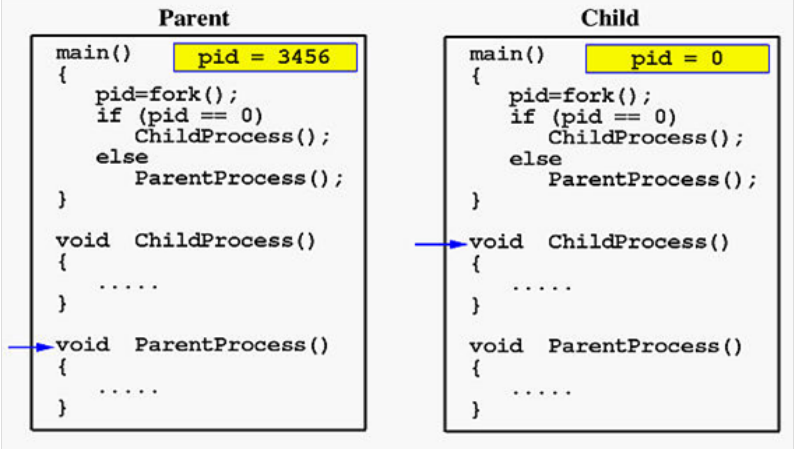


(b) After fork system call both the child and the parent process continue executing with the instruction that follows the call to fork.

Figure 1: Parent and Child Process



(a) Fork system call returns child process ID (created using fork) in parent process and zero in child process



(b) We can execute different portions of code in parent and child process based on the return value of fork that is either zero or greater than zero.

Figure 2: Return Value of fork system call

4 Examples Of Fork()

1. Fork()'s Return Value

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

void main(void){
        pid_t  pid=fork();
        if(pid == 0){ //This check will pass only for child process
                printf("I am Child process \n");
```

```
        }

        else if (pid > 0){ // THIS Check will pass only for parent
                printf("I am Parent process \n");
        }

        else if (pid < 0){ // if fork() fails
                printf("Error in Fork");
        }
}
```

2. Manipulating Local and Global Variables

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

int global=0;

int main()
{
        int status;
        pid_t child_pid;
        int local = 0;
        /* now create new process */
        child_pid = fork();

        if (child_pid >= 0){ /* fork succeeded */
                if (child_pid == 0){
                        /* fork() returns 0 for the child process */
                        printf("child process!\n");
                        // Increment the local and global variables
                        local++;
                        global++;
                        printf("child PID =  %d, parent pid = %d\n",
                                        getpid(), getppid());
                        printf("\n child's local = %d, child's
                                        global = %d\n",local,global);
        }

                else{ /* parent process */
                        printf("parent process!\n");
                        printf("parent PID =  %d, child pid = %d\n",
                                        getpid(), child_pid);
                        int w=wait(&status);
                        //The change in local and global variable
                        //in child process should not reflect
                        //here in parent process.
                        printf("\n Parent'z local = %d, parent's  global
                                        = %d\n",local,global);
                        printf("Parent says bye!\n");
```

```
                        exit(0);  /* parent exits */
        }
}

        else{ /* failure */
                perror("fork");
                exit(0);
        }
}
```

5 WAIT SYSTEM CALL

This function blocks the calling process until one of its child processes exits. wait() takes the address of an integer variable and returns the process ID of the completed process.

```
pid_t wait(int *status)
```

Include <sys/wait.h> and <stdlib.h> for definition of wait().
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.

EXAMPLE

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

void  main(void){
        int status=0;
        pid_t  pid=fork();
        if(pid == 0){ //This check will pass only for child process
                printf("I am Child process with pid %d and i am not
                        waiting\n",getpid());
                exit(status);
        }

        else if (pid > 0){ // THIS Check will pass only for parent
                printf("I am Parent process with pid %d and
                        i am waiting\n",getpid());
                pid_t exitedChildId=wait(&status);
                printf("I am Parent process and the child with pid %d
                        is exited\n",exitedChildId);
        }

        else if (pid < 0){ // if fork() fails
```

```
                printf("Error in Fork");
        }


 }
```

6 EXIT SYSTEM CALL

A computer process terminates its execution by making an exit system call.

```
#include <stdlib.h>
int main(){
        exit(0);
}
```

7 COMMAND LINE ARGUMENT TO C PROGRAM

```
void main(int argc, char *argv[]){
        /* argc -- number of arguments */
        int i;
        for(i=0;i<argc;i++){
                printf("The argument at %d index is %s\n",i,argv[i]);
        }
}
```

To run:
./commandlineargument.out abc def 1 2