NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES ISLAMABAD

OPERATING SYSTEMS LAB FALL 2023

Lab Manual 02

Process Creation

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

---

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 nonnegative 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

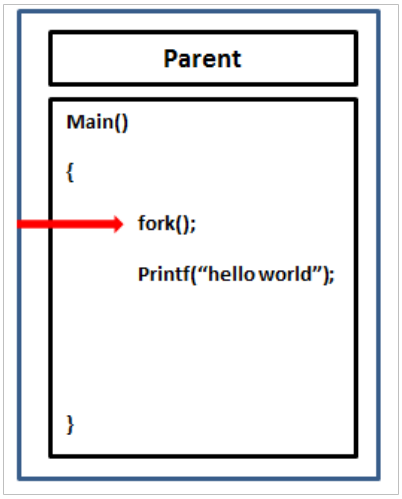
An existing process can create a new process by calling the fork function.

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

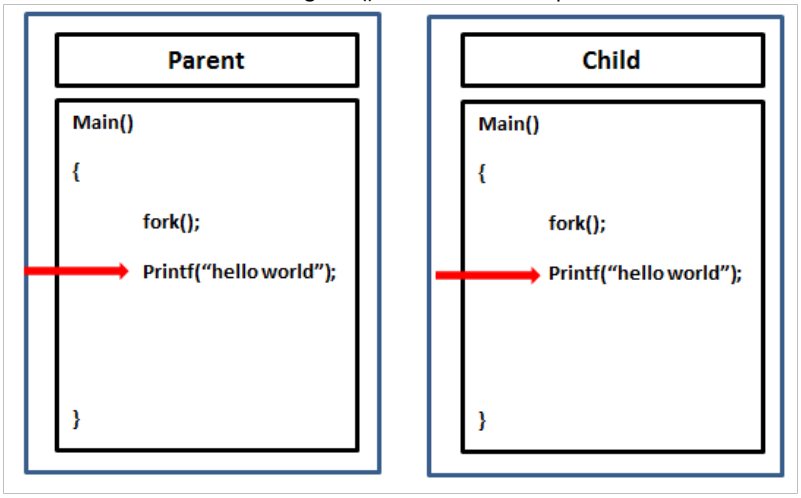
The definition of the pid_t is given in <sys/types> include file and <unistd.h> contain the declaration of *fork* system call.

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.

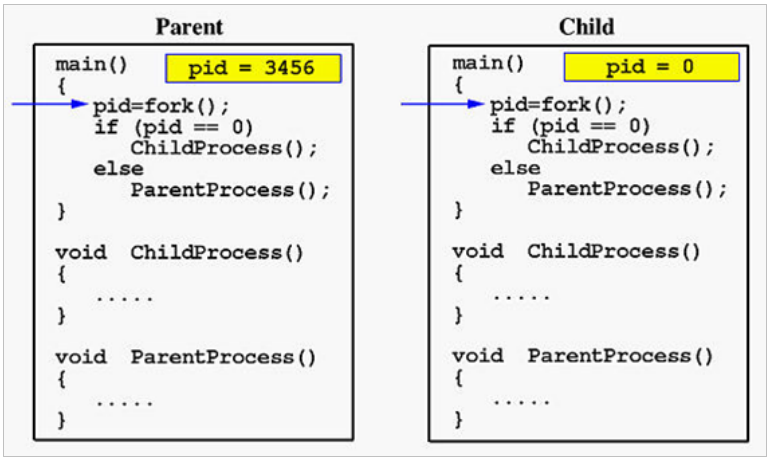


(a) The process will execute sequentially until it arrives at fork system call. This process will be named as parent process and the one created using fork() is termed as child process.

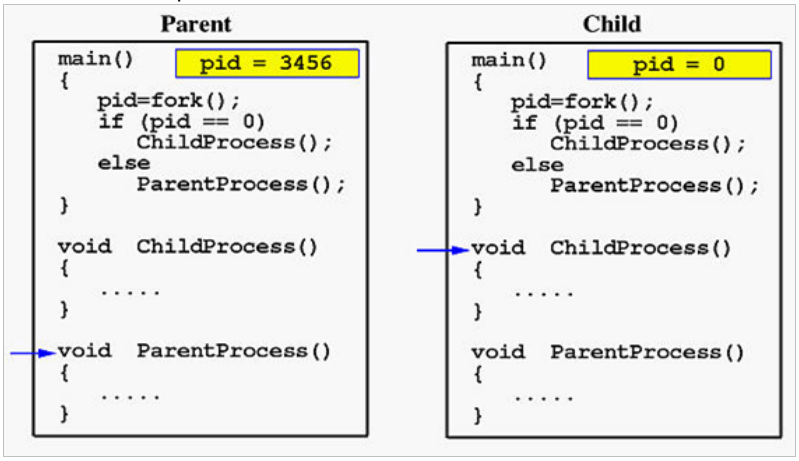


(b) After fork system call both the child and the parent process continueexecuting with the instruction that follows the call to fork.

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 processbased 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 printf ( " I am
                     Parent process \n" ) ;                          only   for   parent
        }

            else if  ( pid < 0){ / / i f fork ( ) f a i l s 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 c r e a t e 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 l o c a l and global v a r i a b l e s 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 l o c a l and global v a r i a b l e / / in child
                                    process should not r e f l e c t / / 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      e x i t s    */

            } }

               else {


            } }         /* f a i l u r e */ 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 WAITPID

Suspends the calling process until a child process ends or is stopped. More precisely, waitpid() suspends the calling process until the system gets status information on the child. If the system already has status information on an appropriate child when waitpid() is called, waitpid() returns immediately. waitpid() is also ended if the calling process receives a signal whose action is either to execute a signal handler or to end the process.

waitpid(pid_t pid, int *status_ptr, int options)

pid_t *pid*

Specifies the child processes the caller wants to wait for:

- If *pid* is greater than 0, waitpid() waits for termination of the specific child whose process ID is equal to *pid*.
- If *pid* is equal to zero, waitpid() waits for termination of any child whose process group ID is equal to that of the caller.
- If *pid* is -1, waitpid() waits for any child process to end.
- If *pid* is less than -1, waitpid() waits for the termination of any child whose process group ID is equal to the absolute value of *pid*.

int **status_ptr*

Points to a location where waitpid() can store a status value. This status value is zero if the child process explicitly returns zero status. Otherwise, it is a value that can be analyzed with the status analysis macros described in “Status Analysis Macros”, below.

The *status_ptr* pointer may also be NULL, in which case waitpid() ignores the child's return status.

int *options*

Specifies additional information for waitpid(). The *options* value is constructed from the bitwise inclusive-OR of zero or more of the following flags defined in the sys/wait.h header file:

WCONTINUED

Special behavior for XPG4.2: Reports the status of any continued child processes as well as terminated ones. The WIFCONTINUED macro lets a process distinguish between a continued process and a terminated one.

WNOHANG

Demands status information immediately. If status information is immediately available on an appropriate child process, waitpid() returns this information. Otherwise, waitpid() returns immediately with an error code indicating that the information was not available. In other words, WNOHANG checks child processes without causing the caller to be suspended.

WUNTRACED

Reports on stopped child processes as well as terminated ones. The WIFSTOPPED macro lets a process distinguish between a stopped process and a terminated one.

Special behavior for XPG4.2: If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of the children terminate, and waitpid() will fail and set errno to ECHILD.

Status analysis macros: If the *status_ptr* argument is not NULL, waitpid() places the child's return status in **status_ptr*. You can analyze this return status with the following macros, defined in the sys/wait.h header file:

WEXITSTATUS(status_ptr*)**

When WIFEXITED() is nonzero, WEXITSTATUS() evaluates to the low-order 8 bits of the status argument that the child passed to the exit() or _exit() function, or the value the child process returned from main().

WIFCONTINUED(status_ptr*)**

Special behavior for XPG4.2: This macro evaluates to a nonzero (true) value if the child process has continued from a job control stop. This should only be used after a waitpid() with the WCONTINUED option.

WIFEXITED(status_ptr*)**

This macro evaluates to a nonzero (true) value if the child process ended normally (that is, if it returned from main(), or else called the exit() or _exit() function).

WIFSIGNALED(status_ptr*)**

This macro evaluates to a nonzero (true) value if the child process ended because of a signal that was not caught.

WIFSTOPPED(status_ptr*)**

This macro evaluates to a nonzero (true) value if the child process is currently stopped. This should only be used after a waitpid() with the WUNTRACED option.

WSTOPSIG(status_ptr*)**

When WIFSTOPPED() is nonzero, WSTOPSIG() evaluates to the number of the signal that stopped the child.

WTERMSIG(status_ptr*)**

When WIFSIGNALED() is nonzero, WTERMSIG() evaluates to the number of the signal that ended the child process.

Return Value

If successful, waitpid() returns a value of the process (usually a child) whose status information has been obtained.

If WNOHANG was given, and if there is at least one process (usually a child) whose status information is not available, waitpid() returns 0.

If unsuccessful, waitpid() returns -1 and sets errno to one of the following values:

Error Code

Description

ECHILD

The process specified by *pid* does not exist or is not a child of the calling process, or the process group specified by *pid*does not exist or does not have any member process that is a child of the calling process.

EINTR

waitpid() was interrupted by a signal. The value of **status_ptr* is undefined.

EINVAL

The value of *options* is incorrect