KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
Information and Computer Science Department
ICS 431 Operating Systems
Lab # 5

Process Creation and Execution – Part II

Objective:

This lab describes how a program can replace its code with that of another executable file. Actually, there are three distinct operations involved: creating a new child process, causing the new process to execute a program, and coordinating the completion of the child process with the original program.

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.

This section describes the exec family of functions, for executing a file as a process image. You can use these functions to make a child process execute a new program after it has been forked.

The functions in this family differ in how you specify the arguments, but otherwise they all do the same thing. They are declared in the header file **"unistd.h"**.

Function: int execv (const char *filename, char *const argv[ ])

The **`execv()`** function executes the file named by filename as a new process image. The **`argv`** argument is an *array of null-terminated strings* that is used to provide a value for the **`argv`** argument to the main function of the program to be executed. The last element of this array must be a **null pointer**. By convention, the first element of this array is the file name of the program sans directory names.

The environment for the new process image is taken from the environ variable of the current process image.

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

This is similar to **`execv`**, but the **`argv`** strings are specified individually instead of as an array. A null pointer must be passed as the last such argument.

Function: int execvp (const char *filename, char *const argv[ ])

The **`execvp`** function is similar to **`execv`**, except that it searches the directories listed in the **PATH** environment variable to find the full file name of a file from filename if filename does not contain a slash.

This function is useful for executing system utility programs, because it looks for them in the places that the user has chosen. Shells use it to run the commands that user’s type.

Function: int execlp (const char *filename, const char *arg0, ...)

This function is like **`execl`**, except that it performs the same file name searching as the **`execvp`** function.

These functions normally don't return, since execution of a new program causes the currently executing program to go away completely. A value of **-1** is returned in the event of a failure.

If execution of the new file succeeds, it updates the access time field of the file as if the file had been read.

Executing a new process image completely changes the contents of memory, copying only the argument and environment strings to new locations. But many other attributes of the process are unchanged:

- The process ID and the parent process ID.
- Session and process group membership.
- Real user ID and group ID, and supplementary group IDs.
- Current working directory and root directory. In the GNU system, the root directory is not copied when executing a setuid program; instead the system default root directory is used for the new program.
- File mode creation mask.
- Process signal mask.
- Pending signals.
- Elapsed processor time associated with the process; see section Processor Time.

Examples

The following programs execs the commands "ls -l -a" and "echo hello there" using the 4 most-used forms of exec. Enter each, compile, and run.

Lab1.c (Using execl)

```
#include <stdio.h>
#include <unistd.h>

main ( )
{
        execl ("/bin/ls",         /* program to run - give full path */
                    "ls",                  /* name of program sent to argv[0] */
                    "-l",                  /* first parameter (argv[1])*/
                    "-a",                  /* second parameter (argv[2]) */
                    NULL) ;              /* terminate arg list */

    printf ("EXEC Failed\n") ;
        /* This above line will be printed only on error and not otherwise */
}
```

Lab2.c (Using execlp)

```
#include <stdio.h>
#include <unistd.h>

main ( )
{
        execlp ("ls",         /* program to run - PATH Searched */
                      "ls",                  /* name of program sent to argv[0] */
                      "-l",                  /* first parameter (argv[1])*/
                      "-a",                  /* second parameter (argv[2]) */
                      NULL) ;              /* terminate arg list */

    printf ("EXEC Failed\n") ;
         /* This above line will be printed only on error and not otherwise */
}
```

Lab3.c (Using execv)

```
#include <stdio.h>
#include <unistd.h>

main (argc, argv )
int argc ;
char *argv[ ] ;
{
        execv ("/bin/echo",        /* program to load - full path only */
                        &argv[0] ) ;

    printf ("EXEC Failed\n") ;
        /* This above line will be printed only on error and not otherwise */
}

Sample Output:

vlsi> gcc lab3.c -o myecho
vlsi> myecho ICS431 in term-032
ICS431 in term-032
vlsi>
```

Lab4.c (Using execvp)

```
#include <stdio.h>
#include <unistd.h>
main (int argc, char *argv[] )
 {
        execvp ("echo",        /* program to load - PATH searched */
                        &argv[0] ) ;

    printf ("EXEC Failed\n") ;
        /* This above line will be printed only on error and not otherwise */
}

Sample Output:

vlsi> gcc lab4.c -o myecho
vlsi> myecho ICS431 in term-032
ICS431 in term-032
vlsi>
```

Lab5.c : Write a program where a child is created to execute a command.

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

main ( )
{
   int  forkresult ;

   printf ("%d:  I am the parent. Remember my number!\n", getpid( ) ) ;
   printf ("%d:  I am now going to fork ... \n", getpid( ) ) ;

   forkresult = fork ( ) ;

   if (forkresult != 0)
    {    /* the parent will execute this code */
         printf ("%d:  My child's pid is %d\n", getpid ( ), forkresult ) ;
     }
      else    /* forkresult == 0 */
       {        /* the child will execute this code */
          printf ("%d:  Hi !  I am the child.\n", getpid ( ) ) ;
          printf ("%d:  I'm now going to exec ls!\n\n\n", getpid ( ) ) ;
          execlp ("ls", "ls", NULL) ;
          printf ("%d:  AAAAH ! ! My EXEC failed ! ! ! !\n", getpid ( ) ) ;
          exit (1) ;
       }

      printf ("%d:  like father like son. \n", getpid ( ) ) ;
}
```

Sample Output:

```
vlsi> gcc lab5.c
vlsi> a.out
24639:  I am the parent. Remember my number!
24639:  I am now going to fork ...
24640:  Hi !  I am the child.
24640:  I'm now going to exec ls!
24639:  My child's pid is 24640
24639:  like father like son.
vlsi> a.out      lab1.c   lab11.c  lab2.c   lab8.c   myecho
lab3.c   lab9.c
vlsi>
```

Run this program several times. You should be able to get different ordering of the output lines (sometimes the parent finished before the child, or vice versa). This means that after the fork, the two processes are no longer synchronized.

Process Completion Status

If the exit status value of the child process is zero, then the status value reported by wait is also zero. You can test for other kinds of information encoded in the returned status value using the following macros. These macros are defined in the header file **"sys/wait.h".**

Macro: int WIFEXITED (int status)
This macro returns a nonzero value if the child process terminated normally with `exit()` or `_exit()`.

Macro: int WEXITSTATUS (int status)
If WIFEXITED is true of status, this macro returns the low-order 8 bits of the exit status value from the child process.

Macro: int WIFSIGNALED (int status)
This macro returns a nonzero value if the child process terminated because it received a signal that was not handled.

Macro: int WTERMSIG (int status)
If WIFSIGNALED is true of status, this macro returns the signal number of the signal that terminated the child process.

Macro: int WCOREDUMP (int status)
This macro returns a nonzero value if the child process terminated and produced a core dump.

Macro: int WIFSTOPPED (int status)
This macro returns a nonzero value if the child process is stopped.

Macro: int WSTOPSIG (int status)
If WIFSTOPPED is true of status, this macro returns the signal number of the signal that caused the child process to stop.

Here's a program, which forks. The parent waits for the child. The child asks the user to type in a number from 0 to 255 then exits, returning that number as status.

Examples

Lab6.c

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

main ( )
{
   int    number=0, statval ;
   printf ("%d:  I'm the parent !\n", getpid ( ) ) ;
   printf ("%d:  number = %d\n", getpid ( ), number ) ;
   printf ("%d:  forking ! \n", getpid ( ) ) ;

   if ( fork ( ) == 0 )
   {
         printf ("%d: I'm the child !\n", getpid ( ) ) ;
         printf ("%d: number = %d\n", getpid ( ), number ) ;
         printf ("%d: Enter a number :  ", getpid ( ) ) ;
         scanf ("%d", &number) ;
         printf ("%d: number = %d\n", getpid ( ), number ) ;
         printf ("%d: exiting with value %d\n", getpid ( ), number ) ;
         exit (number) ;
   }

   printf ("%d: number = %d\n", getpid ( ), number ) ;
   printf ("%d: waiting for my kid !\n", getpid ( ) ) ;
   wait (&statval) ;

   if ( WIFEXITED (statval) )
   {
        printf ("%d: my kid exited with status %d\n",
                   getpid ( ), WEXITSTATUS (statval) ) ;
   }
   else
   {
         printf ("%d: My kid was killed off ! ! \n", getpid ( ) ) ;
   }
}
```

Lab7.c

Here's an example call to `wait()`: a program spawns two children, then waits for their completion and behaves differently according to which one is finished. Try to compile and execute it.

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

main ( )
{
    pid_t  whichone, first, second ;
    int  howmany, status ;

    if ( (first = fork ( ) ) == 0 )          /* Parent spawns 1st child */
    {
        printf ("Hi, I am the first child, and my ID is %d\n", getpid ( ) ) ;
        sleep (10) ;
        exit (0) ;
    }
    else if (first == -1)
    {
        perror ("1st fork: something went wrong\n") ;
        exit (1) ;
    }
    else if ( ( second = fork ( ) ) == 0 )    /* Parent spawns 2nd child */
    {

        printf ("Hi, I am the second child, and my ID is %d\n", getpid ( ) ) ;
        sleep (15) ;
        exit (0) ;
     }
    else if (second == -1)
    {
        perror ("2nd fork: something went wrong\n") ;
        exit (1) ;
    }

    printf ("This is parent\n") ;

    howmany = 0 ;
    while (howmany < 2)                /* Wait Twice */
    {
         whichone = wait(&status) ;
         howmany++ ;

    if (whichone == first)
        printf ("First child exited\n") ;
    else
        printf ("Second child exited\n") ;

    if ( (status & 0xffff) == 0 )
        printf ("correctly\n") ;
    else
        printf ("uncorrectly\n") ;
  }
}
```

The first part of this example, up to the **howmany=0** statement, contains nothing new: just make sure you understand what the instruction flow is in the parent and in the children. The parent then enters a loop waiting for the children's completion. The **wait()** system call blocks the caller process until one of its immediate children (not children's children, or other siblings) terminates, and then returns the pid of the terminated process. The argument to **wait()** is the address on an integer variable or the NULL pointer. If it's not NULL, the system writes 16 bits of status information about the terminated child in the low-order 16 bits of that variable. Among these 16 bits, the ***higher*** 8 bits contain the ***lower*** 8 bits of the argument the child passed to **exit()**, while the ***lower*** 8 bits are all zero if the process exited correctly, and contain error information if not. Hence, if a child exits with 0 all those 16 bits are zero. To reveal if this is actually the case we test the bitwise AND expression (**status & 0xffff**), which evaluates as an integer whose lower 16 bits are those of `status`, and the others are zero. If it evaluates to zero, everything went fine, otherwise some trouble occurred. Try changing the argument passed to **exit()** in one of the children.

Process Groups

The **setpgrp ( )** System Call :

The **setpgrp()** system call creates a new process group. The **setpgid()** system call adds a process to a process group.

The synopsis for **setpgrp()** follows:

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

    pid_t    setpgrp (void);
    int   setpgid (pid_t pid, pid_t pgid);
```

If the process calling **setpgrp()** is not already a session leader, the process becomes one by setting its GID to the value of its PID. **setpgid()** sets the process group ID of the process with PID **pid** to **pgid**. If **pgid** is equal to **pid** then the process becomes the group leader. If **pgid** is not equal to **pid** , the process becomes a member of an existing process group.
Compile and Run the following program to understand the process groups creation.

Example

Lab8.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
main ( )
{
  pid_t    parent_pid, child_pid, fork_pid, wait_pid ;
  pid_t    parent_grp, child_grp, grpid ;
  int         child_stat, exit_val ;

  exit_val = 10 ;
  parent_pid = getpid ( ) ;
  parent_grp = getpgrp ( ) ;
  printf ("\nParent process:   process ID: %ld     group ID: %ld\n",
              (long) parent_pid, (long) parent_grp) ;

  fork_pid = fork ( ) ;

  switch (fork_pid)
  {
     case -1:
        perror ("FORK FAILED\n") ;
        errno = 0 ;
        break ;
     case 0:
        child_pid = getpid ( ) ;
        child_grp = getpgrp ( ) ;
        printf ("Child process:   process ID: %ld    group ID: %ld   "
                    "parent process ID: %ld\n", (long) child_pid,
                    (long) child_grp, (long) getppid ( ) ) ;
        grpid = setpgrp ( ) ;                        /* Change the group of child */
        setpgid (child_pid, grpid) ;
        child_grp = getpgrp ( ) ;
        printf ("Child process again: process ID: %ld group ID: %ld "
                   "parent process ID: %ld\n", (long) child_pid,
                   (long) child_grp, (long) getppid ( ) ) ;
        printf ("Child process:  terminate with \"exit\" - value:  %d\n",
                    exit_val) ;
        exit (exit_val) ;
        break ;
    default:
        printf ("Parent process: child process with ID %ld created.\n",
                    (long) fork_pid) ;
        wait_pid = wait (&child_stat) ;
        if (wait_pid == -1)
        {
           perror ("wait") ;
           errno = 0 ;
        }
        else
        {
            printf ("Parent process: child process %ld has terminated.\n",
                      (long) wait_pid) ;
         }
    }
}
```

Important NOTE:

1. The Shell acts as the parent process. All the processes started by the user are treated as the children of shell.

2. The status of a UNIX process is shown as the second column of the process table when viewed by the execution of the ps command. Some of the states are R: *running*, O: *orphan*, S: *sleeping*, Z: *zombie*.

3. The child process is given the time slice before the parent process. This is quite logical. For example, we do not want the process started by us to wait until its parent, which is the UNIX shell finishes. This will explain the order in which the print statement is executed by the parent and the children.

4. The call to the wait ( ) function results in a number of actions. A check is first made to see if the parent process has any children. If it does not, a -1 is returned by wait ( ). If the parent process has a child that has terminated (a zombie), that child's PID is returned and it is removed from the process table. However if the parent process has a child that is not terminated, it (the parent) is suspended till it receives a signal. The signal is received as soon as a child dies.

Exercises

Note:

Lab exercise will be given at the lab time.