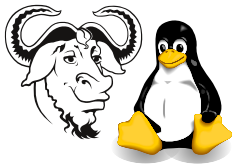

Processes: Writing a Simple Shell

1 Aim

The successful student will write a very simple shell, i.e., a program that can interactively start other programs.

2 Background

A *shell* is a program that can start other programs. A real shell can do lots of other things (it supports a programming language, for example), but here we keep it very simple, and restrict it to starting other programs.

2.1 How do you Run an External Program as a New Process?

- Replace the instructions in a running process with a new set of instructions, using the `exec` function
- First make an exact copy of your process using `fork()`
- Then replace the contents of this new process with another program, using `exec()`.

2.2 The `exec*()` functions

- There are six kinds of `exec*()` function; see `man 3 exec` and `man 2 execve`
- We will use `execl()`:

  ```
  int execl(const char *path, const char *arg, ...);
  ```

  Parameter number:

  1. gives full path of the program file you want to execute
  2. gives name of the new process
  3. specifies the command line arguments you pass to the program
  4. (in this example) is a `NULL` pointer to end the paramter list. We *must* always put a `NULL` pointer at the end of this list.

- Program 1 on the following page is a simple example, without error checking.

As we saw in the lecture, Linux and Unix provide simple system calls to manage processes:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Returns *twice*; returns 0 if child, returns child's PID if parent, returns -1 if error.

Program 1 A simple program using `fork()` and `execl()`. It does no error checking.

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

int main()
{
        printf( "hello world\n" );
        int pid = fork();
        printf( "fork returned %d\n", pid );
        if ( pid == 0 )
                execl( "/bin/ls", "ls", NULL );
        else
                printf( "I'm the parent\n" );
}
```

2.3 What Happens Between fork() and exec() and After?

- Before calling `fork()`:
  - There is one process, the parent process.
- After calling `fork()`:
  - Two process are running, both still have the original code
- After calling `exec()`:
  - The child process, which called `exec()`, now has completely different code.

Program 2 is called `print.c` and prints a number n times. Program 3 on the following

Program 2 A simple program `print.c` that takes two numbers as parameters, and repeats the first number the number of times given by the second number.

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

int main( int argc, char *argv[] )
{
        // argv[0] is the program name
        int num = atoi( argv[1] );
        int loops = atoi( argv[2] );
        int i;
        for ( i = 0; i < loops; ++i )
                printf( "%d ", num );
}
```

page is called `call-print.c`, and is written to call the program `print.c`, using `execl()`.

Program 3 A program `call-print.c` that uses `execl()` to call program 2 on the previous page, `print.c`, in two separate processes.

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

int main()
{
        printf( "hello world\n" );
        int pid = fork();
        printf( "fork returned %d\n", pid );
        if  ( pid == 0 )
                execl( "./print", "print", "1", "100", NULL );
        else
                execl( "./print", "print", "2", "100", NULL );
}
```

2.4 Exercise Set 1

1. Copy the programs from the network filesystem at `/home/nfs/processes-and-threads` to a new directory in your `$HOME`.

2. Compile and run program 1 on the preceding page.

   ```
   $ gcc -o fork-1 fork-1.c
   $ ./fork-1
   ```

3. Modify the program 1 on the previous page so that it runs the program `ls` with the option `-l`.

4. Compile the program `print.c` in program 2 on the preceding page and run it:

   ```
   $ gcc -o print print.c
   $ ./print 10 5
   ```

   Try running it with a few different numbers.

5. Compile the main program `call-print.c` in program 3 and run it.

6. Try printing each number: 100 times, 1000 times, 10,000 times, 100,000 times.

2.5 Implementing a Shell

```
Prompt user
Get command
If not time-to-exit
     Fork new process
     Replace new process with either who, ls or uptime
Read next command
```

Program 4 on the following page is a simple example shell.

Program 4 A simple shell program, `shell-1.c`.

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

void print_menu( void )
{
      printf( "Enter 1=who, 2=ls, 3=uptime -> " );
}

int main()
{
        int cmd;
        print_menu();
        scanf( "%d", &cmd );
        while ( cmd != 0 ) {
                int pid = fork();
                if ( pid == 0 ) {
                        if ( cmd == 1 )
                                execl( "/usr/bin/who", "who", NULL );
                        if ( cmd == 2 )
                                execl( "/bin/ls", "ls", NULL );
                        if ( cmd == 3 )
                                execl( "/usr/bin/uptime", "uptime", NULL );
                        exit( 1 );
                }
                /* add: wait( NULL ); here */
                print_menu();
                scanf( "%d", &cmd );
        }
}
```

2.6 Exercise Set 2

1. Implement program 4 and run it. What if you give it a wrong number?
2. Open a second shell window, and monitor the creation of zombie processes by executing the command `watch -n1 "ps aux | grep ' [Z] '"`
3. Modify the program to print an error message if a command is not supported.
4. Add the the call to `wait()` in the loop before printing the menu. What is the difference in the behaviour of your program?
5. Modify the program and add two more commands to your shell, such as `date` and `hostame`.
6. Modify the program so that it will exit cleanly if it reads end of file. You can manually provide end of file to a process reading standard input by pressing Control-d. Hint: see `man scanf`.
7. Examine the program `simplesh.c`. Modify it to implement background or foreground processes.