csc2404 Practical Week

Processes and Inter-Process Communication

Week 4

Fork and Exec

Code: os-sh.c

Exercise 1: The code `os-sh.c` is a simple shell. It provides a command line prompt and will start executing the command entered at the prompt.

Compile the shell with the command

```
gcc -o os-sh os-sh.c -lreadline
```

the shell uses the library function `readline()`[1] which allows command line editing (see `man 3 readline`).

At the prompt enter any command as you would any shell.

Study the code especially from the `fork()` command to the end of the main loop—this is the critical part.

Exercise 2: The simple shell can recognise when an ampersand is the last word on the line—the variable `background` is set to true.

Modify the code so that if `background` is true the shell will not wait for the command to terminate—the command will effectively run in the background.

Test your code with the command:

```
os-sh> gnome-terminal &
```

Exercise 3: The `statcmd()` function searches for the file that is to be executed using a predefined list of binary directories. When it finds it it uses the system call `stat` (see `man 2 stat`) to find the properties of the file—with the following code:

```
if( S_ISREG(sb.st_mode) && (
        (sb.st_mode  & S_IXUSR) ||
```

[1] To be able to link against this library the developer package must be installed on your machine. Under Debian-like OSes the package is called `libreadline-dev`. Under RedHat-like OSes the package is `readline`

```
(sb.st_mode  & S_IXGRP) ||
(sb.st_mode  & S_IXOTH) ) ) return 0;
```

(see `man 2 stat` for an explanation of the macros above) First it checks if it is a regular file then if it is 'usr' or 'group' or 'other' executable. What is wrong with this? What are the circumstance in which this can fail? What other information in the 'stat buffer' structure needs to be used to ensure the executable will run?

Interprocess Communication

Pipes

The Unix standard specifies that a pipe only allows data to flow in one direction! When a pipe is created two file descriptors are returned—one for each end of the pipe—one end is for writing the other for reading. A pipe is created by the following system call:

```
int fd[2];
pipe(fd);
```

The array `fd` contains the file descripters of the open ends of the pipe—`fd[1]` is the write end and `fd[0]` is the read end.

Code: pipe.c

Exercise 4:

Compile and run the program `pipe.c`. Study the code.

Exercise 5: The term "file descriptor" is used above and in the man page for 'pipes' (and any system routine associated with files). What is a "file descriptor"? (See `man 3 stdout`) Note: this topic will be covered in detail later in the course, see Chapter 9 of the Study Book.

Exercise 6: For two-way communication instead of using a pipe a "socket-pair" are used. A socket-pair are created using the "socketpair" system call.

Modify the pipe code above to use a socket pair and demonstrate that each end of the data stream can both read and write.

Message Passing

Code: Message.tgz

Exercise 7:

The following shell session demonstrates how to use the code found in `Message.tgz`

```
prompt> create /mq
prompt> send /mq "Message one"
prompt> send /mq "Message two" 3
```

```
prompt> send /mq "Message three" 2

prompt> receive /mq
Read 11 bytes; priority = 3
Message two
prompt> receive /mq
Read 13 bytes; priority = 2
Message three
prompt> receive /mq
Read 11 bytes; priority = 0
Message one
```

run the code and experiment with message parsing.

Exercise 8: What happens when you try to read a message from an empty queue? This behaviour is probably not the behaviour you would want—how can you fix it? Hint: see the man page for `mq_open` and the flag `O_NONBLOCK`

Exercise 9: What happens when you try to store six messages in the queue? Again this is probably not the behaviour you would want—without increasing the size of the queue how can this be fixed? Hint: see the man page for `mq_open` and the flag `O_NONBLOCK`

Shared Memory

Code: shmem.c

Exercise 10:

Compile and run the program `shmem.c`. Study the code—in particular the sequence in which the shared memory is updated.

Exercise 11: Comment out the two lines that contain the call to the `sleep()` system call. Compile and run the code. What is the output? Run it a number of times. What was the purpose of those two lines of code? Why the different sleep times?

Sockets

Sockets are used to establish two-way communication between two processes not necessarily on the same machine. To be able to connect to the remote process a way is required to identify the process and the machine it is running on which is independent of the any operating system. The standard Internet way to identify a machine is via its IP (Internet Protocol) number and to identify the process you wish to talk to on that machine is via the Port number it is listening on.

Code: ServerClient.tgz

Exercise 12: Unpack the file `ServerClient.tgz` and compile

and run the server-client software it contains.

Exercise 13: Both the server and client use a single call to read the data they expect from the other. Explain why this is probably not a good idea and how should it be improved.

strace

A useful tool to see what system calls a process is executing is `strace` It will list all system calls and the parameters passed to the system routine. Remember with every system call the mode switches from User to Kernel.

Exercise 14: Try the following command:

```
prompt> strace cat /dev/null
```

Exercise 15: Use `strace` to examine one of the executables you have run above. Compare the `strace` output with the known system calls from the source code. How different are they? Where have the extra calls come from? What are they for?

Note : If you do not have `strace` installed on your system you will need to install it. Under Debian/Mint/Ubuntu the package name is `strace`. If you have never installed a package then please contact the examiner for help.

Exercises

Ex. 16: Answer the exercises at the end of Chapter 3 of the Study Book.