Internet Programming

Programming Assignment 1: Unix Multiprocessing

Deadline: Monday 17 September 2018 at 23:59pm

1 Writing a Micro-Shell

A shell is a program which reads commands from the keyboard and creates the appropriate processes to execute them. Whenever you type a command in a Unix system, the program which reads and starts your command is a shell.

Usually, shells have a multitude of additional functions, such as interpreting scripts, manipulating environment variables, and so on. The goal of this exercise is to write a minimalistic shell, which only reads commands from the keyboard and executes them.

1.1 Programming the shell

For the programming part of this assignment, you will have to implement four progressively more complex versions of a basic shell. These shell variants must be named `mysh1`, `mysh2`, `mysh3`, `mysh4`. Following are the descriptions of the shell variants.

1. `mysh1` reads a program name from the keyboard and executes it. The program binary may be in any directory of the `$PATH`. To execute it, the shell creates a new child process that uses `exec()` to run the requested program. The shell (parent process) waits for the child process to terminate before accepting another command. For example, entering "`ls`" to your shell should execute `/bin/ls` which lists the contents of the current directory.

   A special case has to be considered for the "`cd`" command, which does not correspond to an executable program. The command must support both absolute (e.g., `cd /home/user`) and relative paths (e.g., `cd ../Pictures`). For implementing the `cd` command, you should look at the `man` pages for `chdir(2)` and `getcwd(3)`.

   A second special case concerns the "`exit`" command. When entered, instead of looking for an executable with that name, your shell should just terminate (without printing any message).

   To indicate that the shell is currently idle and waiting for input, a "`$ `" prompt must be displayed, that is, precisely two characters: a dollar sign followed by precisely one space. If your prompt is not precisely that, our automated testing will detect all your output as wrong!

2. `mysh2` is an extension of `mysh1`. In addition to executing the specified program, it must also accept a number of parameters which will be supplied to the program. For example, your new shell should interpret correctly commands such as "`ls -l /tmp`".

3. `mysh3` is an extension of `mysh2`. It adds support for piped commands such as "`ls /tmp | wc -l`". You can assume that typed commands will contain at most one pipe. I.e., you do not need to support "chained pipes" like "`sort foo | uniq -c | wc -l`".

   ☞ In the implementation of `mysh3`, you will need to use the `dup(2)` or `dup2(2)` functions. Have a look at their `man` pages. For parsing the command line, function `strtok(3)` may come in handy.

4. `mysh4` is like `mysh3` but can also handle chained pipes of arbitrary length, e.g., “`cat a.txt | cut -f 3 | sort | uniq -c | wc -l`”.

The use of `system()` function in your shell program is forbidden. It is also forbidden to invoke another shell (e.g., `/bin/sh`) to do the work for you.

Make sure that any abnormal user input is handled gracefully. I.e., in the presence of abnormal user input, your shell must not abort.

Your shell should not output *any* output to stdout (except for the prompt, of course). That is, we expect that all output that is printed to stdout comes from the executed programs. In case you really want to output some messages, such as `Command not found`, make sure you print them to stderr, although you are advised to produce absolutely no output.

2 Synchronization

Study the source code of program `syn_process_1` (in Figure 1) and `syn_process_2` (in Figure 2). The goal of this exercise is to use synchronization primitives so that the two programs produce the desired output **without changing the `display()` function**.

You can compile these programs as follows:

```
gcc syn_process_1.c display.c -o syn_process_1
gcc syn_process_2.c display.c -o syn_process_2
```

- For `syn_process_1`, we want to prevent the two displayed messages from interpenetrating each other. E.g., this output is correct:

```
Hello world
Hello world
Bonjour monde
Hello world
Bonjour monde
Bonjour monde
```

  . . . but this output is not correct:

```
HelBonlo world!
jour monde
HBeonljloo ur mwonordeld
```

- For `syn_process_2`, we want the following output to be produced (i.e., always `ab` followed by `cd\n`:

```
abcd
abcd
abcd
abcd
...
```

. . . while something like this is considered incorrect output:

```
abcabcd
d
abcabd
abccd
cabd
abcdab
cabd
```

2.1 Programming with Synchronization Primitives

2.1.1 Process Synchronization

Modify the original source of `syn_process_1` and `syn_process_2` so that we get the desired behaviors. Before starting coding, think which is the proper synchronization primitive that will help you achieve the desired behavior from the forked processes. Also think for which part(s) of the programs you should enforce synchronization. You may need to use different synchronization primitives for the two programs.

2.1.2 Thread Synchronization

Transform the programs `syn_process_1` and `syn_process_2` to use pthreads instead of processes. Although process synchronization tools can also be used, this time you *must* use use the synchronization primitives provided by the pthreads library. Call the new programs `syn_thread_1` and `syn_thread_2`.

2.1.3 Java Threads

Write the same two programs in Java (using Java threads). Call their source files `Syn1.java` and `Syn2.java`.

Good luck!

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "display.h"

int main()
{
  int i;

  if (fork())
  {
    for (i=0;i<10;i++)
      display("Hello␣world\n");
    wait(NULL);
  }
  else
  {
    for (i=0;i<10;i++)
      display("Bonjour␣monde\n");
  }

  return 0;
}
```

Figure 1: `syn_process_1.c`

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "display.h"

int main()
{
  int i;

  if (fork())
  {
    for (i=0;i<10;i++)
      display("ab");
    wait(NULL);
  }
  else
  {
    for (i=0;i<10;i++)
      display("cd\n");
  }

  return 0;
}
```

Figure 2: `syn_process_2.c`

```
/* DO NOT EDIT THIS FILE!!!  */

#ifndef __HW_DISPLAY_H__
#define __HW_DISPLAY_H__
void display(char *);
#endif
```

Figure 3: `display.h`

```
/* DO NOT EDIT THIS FILE!!!  */

#include <stdio.h>
#include <unistd.h>
#include "display.h"

void display(char *str)
{
  char *p;
  for (p=str; *p; p++)
  {
    write(1, p, 1);
    usleep(100);
  }
}
```

Figure 4: `display.c`