Assignment 1 Internet Programming

Marc Went (2507013) and Ferry Avis (1945653)

September 19, 2015

1 Programming the shell

1.1 mysh1

For this part of the assignment, the implementation of the `exit` command is worth mentioning. Before executing each command by `execlp`, the `strcmp` function is used to check whether the exit command is typed to terminate the shell if needed. It appeared that the new line character, that is typed in by the user to send the request of the program, is included in the input. A possible solution is to include the new line character in the `strcmp` function, but another solution was chosen instead. All white space characters, i.e. spaces, tabs and new lines, are are stripped from the command. Those characters are never part of the command and can safely be removed if they are entered by accident. Stripping white space characters also appeared to be convenient at implementing pipes.

The typed in command should not be larger than 100 characters, as a buffer of fixed size is used.

The use of the function `execlp` from the `exec` family was sufficient, as no support for arguments is needed.

1.2 mysh2

To implement this program, the `execvp` system call is chosen because it allows multiple parameters to be passed as a single array while taking the PATH variable into account. The user input is split into an array by splitting on a space. The `strtok` system call is used repeatedly to obtain each chunk of the split input and added to the array used for `execvp`. An array of fixed length is passed to `execvp`, as an upper bound on the length can be computed. The input of the program has a given length. If each parameter is separated by one space and each parameter consists of one character, the minimum amount, the maximum length is half of the original input buffer. Add two for rounding and the closing of the array by `NULL`.

1.3 mysh3

The structure of the program is as follows. First, the input is split on the — character and the pipe is created. Then, the custom function `execute` is called that trims the first command that needs to be executed, exits the program if requested and creates a process to execute the requested program. If it is not the last command in a chain, the standard output is changed to the write end of the pipe. After the program is finished and the command contains a pipe, the function `execute` is called again from the parent. The standard input is changed to the read end of the pipe. By using a recursive call, duplicate code for checking whether `exit` is entered and forking is avoided. This structure is also better suited if in future chained pipes would be implemented.

As an extra, the `cd` command is implemented in the shell. The cd command is implemented at the same level as the exit command. It can thus be the last parameter in the chain, but output from previous commands is not read. The cd command supports both relative and absolute paths, as well as the `" "` character to go to the home directory. As arrays of fixed sizes are used, the maximum path length is currently 200.

1.4 Questions from section 1.2

1.4.1 A

The shell needs to create a process for each command in the input, i.e. for `ls /tmp | wc -l` two processes need to be created. Only one pipe is needed per pair of commands in the input. The shell needs to wait for accepting the next command after the last command in the input is finished.

If the shell would need to support chained pipes, we think another pipe must be created in the code. A command that is not the first or the last in line, must read from a pipe and write to it. It is either not possible to do this with one pipe or causes mangled input and output. We think the problem is different from communicating between processes that run simultaneously, which is not possible to do with one pipe, as here pipes are only read after the process of the previous requested program is terminated. The parent should only close the pipes at the last requested program.

1.4.2 B

Could the shell program be implemented using threads instead of processes? No, not completely. The functions from the `exec` family replace the complete contents of the program. No code from the shell program after this call will be executed. After executing one requested program, no additional piped commands can be executed, nor will be asked for new input.

1.5 C

The `cd` command is a shell command that is built in, like the implementation of `exit`. Without the added support, it could not be used.

2 Programming with Synchronization Primitives

2.1 syn1.c

The way we came up with was to use semaphores to decide which process has access to the `display` function. Either the ‘`Hello world`’ is printer or ‘`Bonjour monde\n`’. Only after the display function is finished the semaphore is unlocked.

2.2 syn2.c

To implement `syn2.c` we used semaphores again, but now in a pipe fashion. Two semaphores are used, where ‘`ab`’ would enable ‘`cd\n`’ to be printed, and after that ‘`cd\n`’ allows ‘`ab`’ to be printed again. This is looped 10 times.

2.3 synthread1.c

Here two threads are created: one for the parent, and one for the child. Both share a mutex, which either can lock or unlock. By using two threads instead of one thread and one main code, the code is a lot cleaner.

2.4 synthread2.c

This code is a little more complicated, but in essence it is ‘`syn2.c`’ and ‘`synthread1.c`’ combined. It uses two mutexes and two predicated in a similar fashion as the double semaphore. The difference between `synthread1.c` is that here only one thread function, where the string is passed and the mutex for the parent and the mutex for the child. The child receives them in the opposite way, but same format.

The biggest issue with this solution was proper pointer allocation and value settings. By creating two structs ‘`threadArguments`’ it was possible to pass multiple arguments to the threads. The struct ‘`mutexCond`’ allows to define two variables, one for the parent and one for the child, containing the necessary mutex locks as well as the predicate necessary for the loop.

2.5 syn1.java

By copying the C code to java and synchronizing the display function, the printing of the proper strings was quite easy.

2.6 syn2.java

This is a copy of `synthread2.c`, but in a more organized fashion where the ‘`mutexCond`’ struct is replaced by an AtomicInteger serving both as a mutex and as a predicate.

The thread received two AtomicIntegers. One is for the parent mutex and one for the child mutex. The first argument is, depending on the kind of thread (parent or child), the parent or child mutex and the second argument is either child or parent. In fact, the order does not matter as long as the threads get switched arguments, so they can block and enable each other.