Distributed Systems Group
Department Software Technology
Faculty EEMCS
DELFT UNIVERSITY OF TECHNOLOGY

Operating Systems (TI2726-C)
Lab Session 1: Introduction to C and Processes

D.E. Schut, M. van de Ruit, O. W. Visser & J.S. Rellermeyer

February 22, 2018

Deadline: 7 March 2019 12:45

In this session you will:

- Become familiar with the Raspberry Pi
- Write some simple C programs on the Pi
- Learn how to compile a C program on the Pi
- Learn what a process is and what it does
- Learn what a thread is and what the difference is between processes and threads

Processes in Linux

Early computers allowed only one program to be executed at a time, which had complete control of the system and had access to all of the system's resources. In contrast, contemporary computer systems allow multiple programs to be executed concurrently, that is, to have multiple simultaneous processes. A process is a program in execution, so each time you start a program a new process is created. Each process has a unique ID.
You can use the command `ps` in the console screen to list current processes. Normally `ps` only shows the processes associated with the current user and terminal session. To show all system processes you can use `ps -aux`, this often returns a long list so you might want to use `ps -aux | less` to be able to scroll through the text. Another option is to use the `top` command, which displays real time process information.
To run a process from the terminal in the background, end your call with an `&` for example `sleep 3 &`. Try also running `sleep 3` without the `&` to see the difference. To stop a running process you can use `kill <Process ID>` or `pkill <Process name>`. For example try calling `sleep 1000 &` and then try to stop this process using both methods.

Assignments

For the assignments all code has to be written in C, and has to able to run on the Raspberry Pi. Only the code has to be submitted to CPM. Please use a separate .c file for each assignment and name it with the convention X_Y.c, where X is the number of the lab session and Y is the number of the assignment (use X_b.c for the bonus assignment).

A makefile is provided to compile your programs. For example to compile assignment 1.1 type:

```
make 1_1
```

... and to run it:

```
./1_1
```

Please read the assignments carefully and make sure your code exactly implements the given descriptions. We will assess both the functionality and the clarity of your code and comments. Answers to questions are allowed in both Dutch and English.

1.1

For the first assignment you have to write a program that

1. Creates variables with your name and student number
2. Prints your name and student number in an organised way
3. Prints the ID of the process that runs your program

Hint: For point 3, use the appropriate Linux system call.
You can use the makefile for the compilation step which should make it easier to compile it again.
You should understand why the process id differs across different runs.

1.2

In this assignment you have to write a program that is able to execute commands that you will usually use in the command window. The objective of this task is to become familiar with two different ways of executing Linux commands within a C program.
The objectives are of your program are to:

1. Create a directory in the directory in which the program is executed
2. List all files and directories in the current directory
3. Accomplish the above by making use of:
    - The possibility to invoke shell functions from C programs
    - The functionality for interaction with files and folders from the C standard libraries that are available through `#include`

Please implement both methods for both objectives.

1.3

In the previous two assignments you wrote a program that was executed as a process on the Raspberry Pi. In this assignment you will extend assignment 1.2 with the functionality to create and run another process. This so called "child" process has to execute the same code as the main program. The objectives of your program are to:

1. Create a single program that creates a new process running the same code using the function fork.
2. Create an if/else statement to print which process is active (child or parent)
3. The child and parent processes should do different things:
    - Make the child process show all files and folders using the functionality you built in assignment 1.2.
    - Have the parent process wait for the child process to finish.

1.4

In the previous assignment you spawned a child process to execute functionality. In this assignment you will create a thread instead, which executes the functionality of assignment 1.2. For this you can use Pthreads. Please also briefly explain the differences between child processes and threads. The objectives of your program are to:

1. Create a simple program that creates a thread.
2. Have this thread show all files and folders using the functionality you built in assignment 1.2.
3. Have the main thread wait for the created thread to finish execution.
4. Add a short comment explaining the difference between a child process and a thread.

Bonus Exercise: UNIX Shell and History Feature

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt osh> and the user's next command, which displays the file prog.c on the terminal using the Linux cat command:

```
osh> cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, cat prog.c), and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. However, Linux shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (&) at the end of the command. Thus, if we rewrite the above command as

```
osh> cat prog.c &
```

the parent and child processes will run concurrently. The separate child process is created using the fork() system call, and the user's command is executed using one of the system calls in the exec() family of system calls. A C program that provides the general operations of a command-line shell

is supplied below. The main() function presents the prompt osh-> and outlines the steps to be taken after input from the user has been read. The main() function continually loops as long as the boolean should_run equals 1; when the user enters exit at the prompt, your program will set should_run to 0 and terminate. This exercise is organised into two parts:

1. creating the child process and executing the command in the child, and
2. modifying the shell to allow a history feature.

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

#define MAX_LINE 80 /* The maximum length command */
int main(void) {
char *args[MAX_LINE/2 +1]; /* command line arguments */
int should_run = 1; /* flag to determine when to exit program */
while (should_run) {
printf("osh>");
fflush(stdout);
/**
* After reading user input, the steps are:
* (1) fork a child process using fork()
* (2) the child process will invoke execvp()
* (3) if command did not include &, parent will invoke wait()
*/
}
return 0;
}
```

Part 1: Creating a Child Process

The first task is to modify the main() function as mentioned above so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings. For example, if the user enters the command ps -ael at the osh> prompt, the values stored in the args array are:

```
args[0] = "ps"
args[1] = "-ael"
args[2] = NULL
```

This args array will be passed to the execvp() function, which has the following prototype:

```
execvp(char *command, char *params[]);
```

Here, command represents the command to be performed and params stores the parameters to this command. For this project, the execvp() function should be invoked as execvp(args[0], args). Be sure to check whether the user included an & to determine whether or not the parent process is to wait for the child to exit.

Part 2: Creating a History Feature

The next task is to modify the shell interface program so that it provides a history feature that allows the user to access the most recently entered commands. The user will be able to access up to 10 commands by using the feature. The commands will be consecutively numbered starting at 1, and the numbering will continue past 10. For example, if the user has entered 35 commands, the 10 most recent commands will be numbered 26 to 35. The user will be able to list the command history by entering the command `history` at the `osh>` prompt. As an example, assume that the history consists of the commands (in order of execution):

```
ps; ls -l; top; cal; who; date
```

The command history will output:

```
1 ps
2 ls -l
3 top
4 cal
5 who
6 date
```

This the same as the `history` command on the normal terminal of the Pi, except that we want you to just print the last 10 commands.
Your program should support two techniques for retrieving commands from the command history:

1. When the user enters !!, the most recent command in the history is executed.
2. When the user enters a single ! followed by an integer N, the Nth command in the history is executed.
3. (In the normal terminal on the Pi there is also the option !-X which would execute the X to last command, so !-1 would repeat the last command. You don't have to implement this feature, but it might be useful to know.)

Continuing our example from above, if the user enters !!, the ps command will be performed; if the user enters !3, the command `top` will be executed. Any command executed in this fashion should be echoed on the user's screen. The command should also be placed in the history buffer as the next command. The program should also manage basic error handling. If there are no commands in the history, entering !! should result in a message *No commands in history*. If there is no command corresponding to the number entered with the single !, the program should output *No such command in history*.