Operating Systems (TI2726-C)

Lab Manual

2014-2015

Facluty of Electrical Engineering, Mathematics and Computer Science

A.T. van Rijs B.Sc., R.L. van der Plaats B.Sc.

TU Delft Delft University of Technology

Challenge the future

Lab Manual V1.0 (Wednesday 18th February, 2015)

Contents

Part I

Information

Session 1

Introduction

Operating Systems play an important part of any modern computer system. The field of operating systems has gone through a growing-spurt in the last few years and will continue to grow effortlessly as computers have become prevalent in a lot of areas around us. You will find operating systems in cars, house-appliances, printers, and many other embedded devices.

In front of you lies the course manual in which you will learn the basics of operating systems on an intuitive level. This guide consists of two parts. The manual consists of 5 sessions. Each session will cover an important subject of operating systems. The first session will explain what processes are and how you should use them. The second session will cover Threads and the third session will cover multithreading. Multithreading is an important concept when you want to increase the efficiency of the program you desire to run. The fourth section will show how you can effectively schedule threads and the last section will cover the synchronisation of threads.

Each session consists of several exercises. Once you have finished an exercise you have to let an assistent sign-off on this exercise in your manual before continuing to the next question. Additionally, you will have to upload your solution to CPM[1]. Here your solutions will be subjected to a plagiarism check and officially checked-off. At the end of each session you will find a programming exercise which will be more complex than the exercises you have done before in that session. However, if you manage to successfully implement/answer that question you will be rewarded with 0.2 bonus point. Therefore, if you are able to successfully answer all bonus questions, you will have 1 bonuspoint for your final exam! The bonuspoint only counts if you have all exercises checked and signed by the assistants.

[1] http://cpm.ewi.tudelft.nl

1.1 Overview

Microcontrollers process instructions in a binary matter. Considering the fact that there are a lot of instructions to be called to perform a simple task, it would be time-consuming if users would execute functions by means of binary instructions. Furthermore, in order to increase the amount of users, the system must be easy to work with as not everyone has the knowledge to perform difficult instructions. An operating system acts as an intermediary between the user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient yet easy manner. An operating system disconnects the user from the hardware as the operating system is the one that translates the users' instructions to hardware instructions. Therefore, it can be seen that operating systems manage the computer hardware.

During this lab you will be using the Raspberry Pi which is explained in further detail in section 1.2 and depicted in figure 1.1.

1.2 Raspberry Pi

The Raspberry Pi is a credit card-seized single-board computer created with the intention of promoting the teaching of basic computer science in schools. The Raspberry Pi is based on the Broadcomp BCM2835 system on a chip, which includes an ARM1176JZF-S 700 MHz processor, VideoCore IV GPU, and is shipped with 512 megabytes of RAM. The processor used in the Raspberry Pi is equivalent to a chip used in the early generation smartphones. The Raspberry Pi primarily uses Linux kernel-based operating systems. As mentioned in the previous section, you will be using the Raspberry Pi during this laboratory.

In the next chapter you will be guided through the installation and explained how to upload your code to the Raspberry Pi in order to test your implementation.

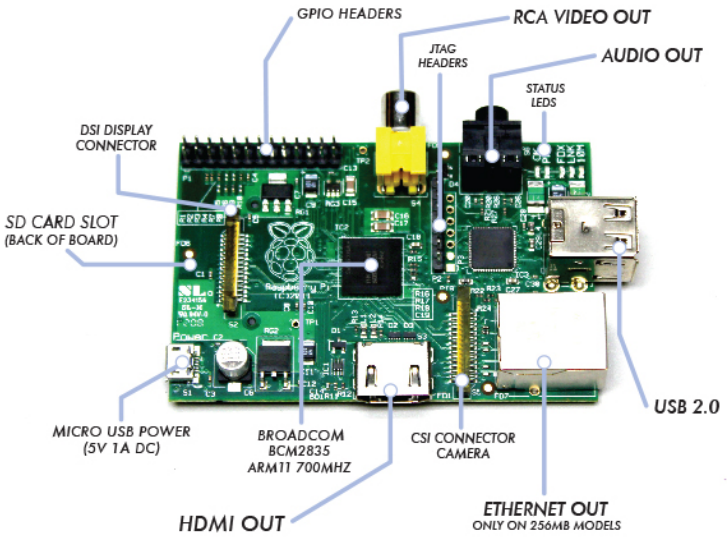

Figure 1.1: Raspberry Pi

Session 2

Laboratory Environment

This chapter discusses the setup you will be using during this lab. Section 2.1 will explain what external hardware you will be going to use during this lab. Furthermore, this section will depict the way the hardware should be configured and which additional packages are required to guarantee correct functionality.Section 2.2 will list the various operating systems that can be used in order to guarantee a correct and stable operation and communication with the hardware.

2.1 External Hardware

Raspberry Pi

During this lab you will be using the Raspberry Pi to run your implementations on. The Raspberry Pi is a small, powerful and lightweight ARM based computer which can do many of the things a desktop PC can do. The immense functionality of the Raspberry Pi together with the huge range of Raspberry Pi accessories offers an almost limitless choice of uses. So whether you want to learn to programme, hack around with some hardware, or to simply learn how operating systems work; the Raspberry Pi can cover these issues.

Console Cable

There are several methods on how to connect the Raspberry Pi to your computer. However, due to restrictions of the TU Delft network, a direct connection to your computer is the easiest and most reliable method. You have been supplied a Raspberry Pi, an SD card, an extension board and lastly as a FTDI USB cable.

Connecting the Raspberry Pi with a console cable has some advantages. For instance, you don't need an external power supply. In the case that your laptop is not able to supply the Raspberry Pi with enough power, you can use an external power device. Additionally, a keyboard, mouse, display or Ethernet cable become redundant when using a console cable. The only thing you need to do is to install

the USB drivers for the Console Lead. The Raspberry Pi has its own built-in serial port which allows users to connect to its console and issue commands just as if you were logged in through ssh.

We have taken te liberty to pre-install these drivers on the lab computers you will be using. The Raspberry Pi must be connected by following the steps listed below.

- Insert the micro SD card in the Raspberry Pi
- Connect the FTDI cable as mentioned on the extension print (corresponding colors)
- Connect the power supply

Make sure you connect the FTDI cable in the right way as mentioned below. Connection the Raspberry Pi differently may cause the Raspberry Pi to blow up.

Other methods of connecting the Raspberry Pi are known however aren't supported by the assistants of this laboratory.

For this lab we used the Raspbian version of Linux which is build for the Raspberry Pi. Besides the version of Linux, a package of Wiring Pi included for I/O support. The package consist of various libraries which provide functions which you can use for I/O operations.

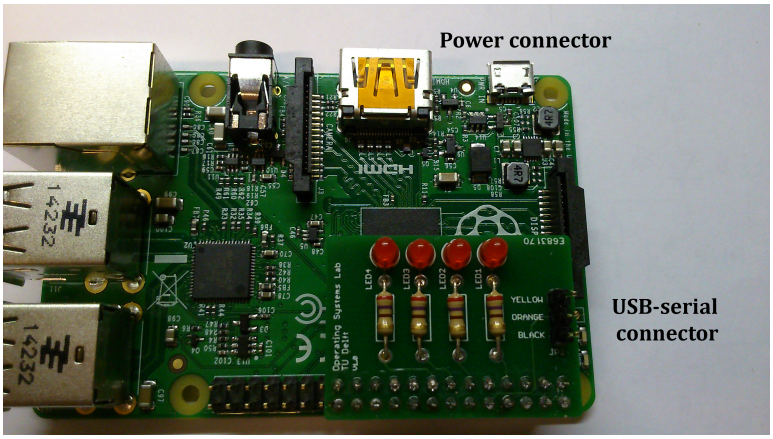

Figure 2.1: Raspberry Pi with extension board

2.2 Programming Environment

In order to successfully program the supplied hardware, you need an operating system with a compiler. Dozens operating systems exists, however to guarantee

functionality and service we will only support the operating systems mentioned below. Please feel free to chose either of these operating systems.

Windows

As mentioned before you will be using a serial cable as a connecting gateway between the computer and raspberry pi. In the case you are using your own windows powered laptop, it is necessary to install the corresponding drivers. In that case - if you have not already done so - install the PL2303 drivers. These can be downloaded from: `http://www.prolific.com.tw/US/ShowProduct.aspx?p_id=225&pcid=41`

Follow the steps mentioned on that page to install the driver. The driver is installed in such a way that when you later plug in the USB console lead, it will still launch the "Found New Hardware" wizard. If you allow the Wizard to search the Internet and install the drivers, it should work. When you have successfully installed the drivers you will be able to connect to your Raspberry Pi through the Console Lead.

Before you are able to connect to the device, you need to know to which communication port the Raspberry Pi is connected. You can find this by looking in the **Ports** section of the **Windows Device Manager** which is accessible through **Control Panel** under **System**.

The program you need to use for connecting with the Raspberry Pi is called Putty. You can download putty from the website `http://www.putty.org`. If you are using an x86 machine you can directly download putty via `http://goo.gl/BgbXpy`. It should be noted that Putty is not an installer but the actual program itself. Therefore, no installation is required to run Putty.

When you open Putty you have to select the connection type *Serial* from the radio buttons and set the speed to *115200* and the serial line to the com port number you found in the previous paragraph. Additionally, please see figure 2.2. If the setup is correct you can open the connection and a console screen will be opened (see figure 2.3). Here you have to login with username **pi** and password **raspberry**.

Linux

If you use Linux it is not needed to install additional software as the *screen* package is included. However in some Linux distributions such as Ubuntu 12.10, the *screen* package is not included. In this case you will have to install the package. You can do this by typing the following command.

```
sudo apt-get install screen
```

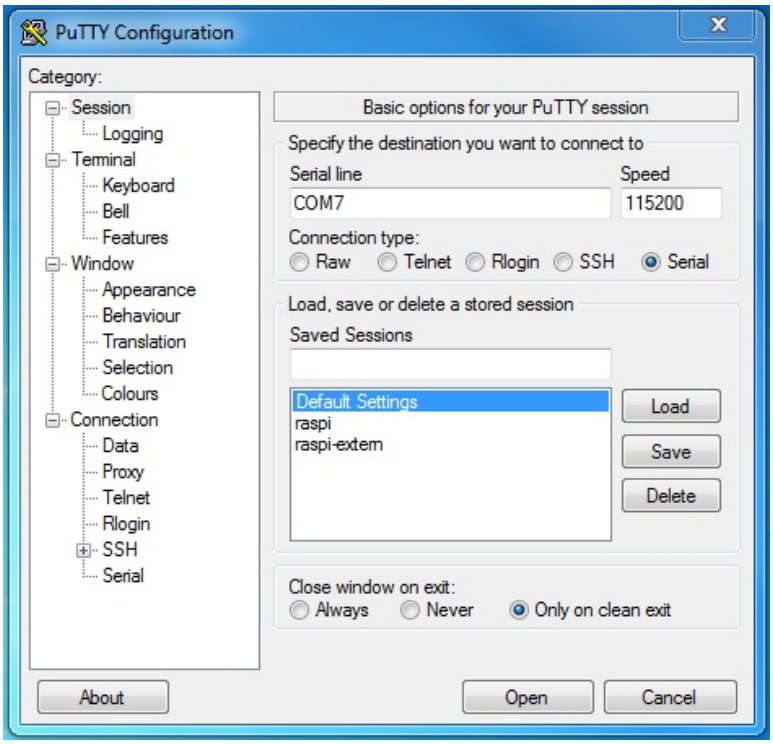

Figure 2.2: Example configuration for Putty

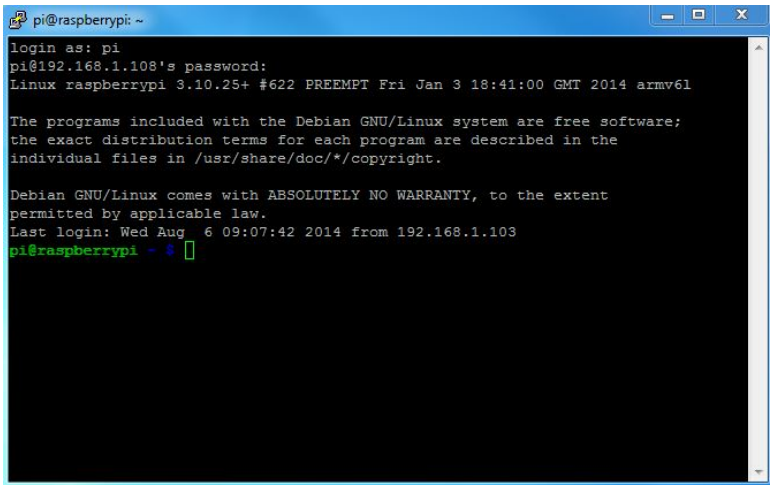

Figure 2.3: Console screen on Windows PC

In order to find to which TTY port the Raspberry Pi is connected, execute the following command:

```
dmesg | grep tty
```

Now if you want to connect to the Raspberry Pi (in this case connected to /dev/ttyUSBO as found in the previous step) you type the following command.

```
screen /dev/ttyUSB0 115200
```

When your connection is successfully initialised you will see the screen of figure 2.4.

Figure 2.4: Console screen Linux

Part II

Assignments

Session 1

Introduction to C and Processes

In previous classes you all have had classes that utilised C as a programming language. This first lab session is used to get to know the environment you will be working with as well as compiling the code you will be going to write.

The objectives of this session are listed below.

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

1.1 Assignments

This first lab session consist of 3 assignments. In these assignments you have to write different programs which will be executed on the Raspberry Pi.

Early computers allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, contemporary computer systems allow multiple programs to be loaded into memory and executed concurrently. The later case requires you to write a program, compile it and finally run it on a computer as a process. A process it a program that is in execution.Your program becomes a process of the Linux operating system. You could use the command `ps` in the console screen to list all the current processes of the system.

To make it easier to assess your assignments you have to write them in different files and you have to write enough comments in your code. We will assess you accordingly!

Assignment 1.1

For the first assignment you have to write a small program. The file with the code of the program must uploaded to the Raspberry Pi followed by a compilation of your code to get the actual program. Once you have performed the compiling step, you could run the program. The program you have to write has some simple objectives which are listed below. Keep in mind that when this program runs on the Raspberry Pi it has become a process of the Linux Operating System.

Objectives

Objectives for a correct implementation of Assignment 1.1

- Create variables with your name and student number
- Make sure your program is able to print your name and student number in an organised way!
- Print the ID of the process that runs your program!

Once your implementation complies with these objectives, you have to upload your solution to the Raspberry Pi and compile it with the GCC compiler of the Raspberry Pi. You could use a makefile for the compilation step which should make it easier to compile it again. When you run this program multiple times you will see that it gets a different process ID each time you run the program. Explain why this is the case.

Signature Assistant:

Assignment 1.2

In this assignment you have to write a program that is able to execute commands that you will usually use in your command window. The objective of this task is to become familiar with two different ways of executing Linux commands within a C program. For this assignment you have to write a program that creates a folder in the folder where your program is executed and you have to execute the function to list all the files and directories in the current folder.

Objectives

Objectives for a correct implementation of Assignment 1.2

- Create a folder
- List all files and directories in the current directory
- Use 2 different functions to realise the above objectives

Signature Assistant:

Assignment 1.3

In the previous two assignments you wrote a program which was executed as a process on the Raspberry Pi. In assignment 1.3 you will extend assignment 1.2 with the functionality to run another process. This so called "child" process has to execute the same code as the main program. The objectives of this assignment are also listed below.

Objectives

Objectives for a correct implementation of Assignment 1.3

- Create a "child" process which runs the same code as the main program
- Create a if/else statement to print which process is active (child or parent)
- The parent process has to wait for the child process before it is finished

Signature Assistant:

1.2 Bonus Question: 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. This project can be completed on any Linux, UNIX, or Mac OS X system. 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: `cat prog.c`. (This command displays the file prog.c on the terminal using the UNIX 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, UNIX 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. 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 should run equals 1; when the user enters exit at the prompt, your program will set should run to 0 and terminate. This project 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.

```c
#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 included &, 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 (from most to least recent):

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

The command history will output:

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

Your program should support two techniques for retrieving commands from the command history:

Objectives

- When the user enters !!, the most recent command in the history is executed.
- When the user enters a single ! followed by an integer N, the Nth command in the history is executed.

Continuing our example from above, if the user enters !!, the ps command will be performed; if the user enters !3, the command cal 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."

Signature Assistant:

Session 2

Threads

The second session of this lab will contain some exercises about threads. In session 1 you learned about process and how they work. In this assignment you will use threads and these threads will be used in all other session of this lab. The objectives of this session are listed below.

- Learn what a thread is and how you could create one
- Learn how you could use the I/O interface of the Raspberry Pi

2.1 Assignments

In the previous assignment you already learned something about processes on a Linux system. In this second assignment you will learn something about threads.

A thread is a basic unit of CPU utilisation; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Since each unique thread has its own memory, it is a lot faster to create and destroy threads.

In this session you will learn how to create threads. The concept of threads will be further utilised by all the other assignments of this lab. The next session is about multithreading and the advantage of threads above processes.

Using I/O on Raspberry Pi

This tutorial will state information with regards to the input/output (I/O) functionalities of the Raspberry Pi. To make it easy, we will use a library which provides a set of functions to communicate with the I/O pins of the Raspberry Pi. This lab will use the Wiring Pi library. This tutorial will mention the basic functions you have to use during this lab. If you want to use more advanced

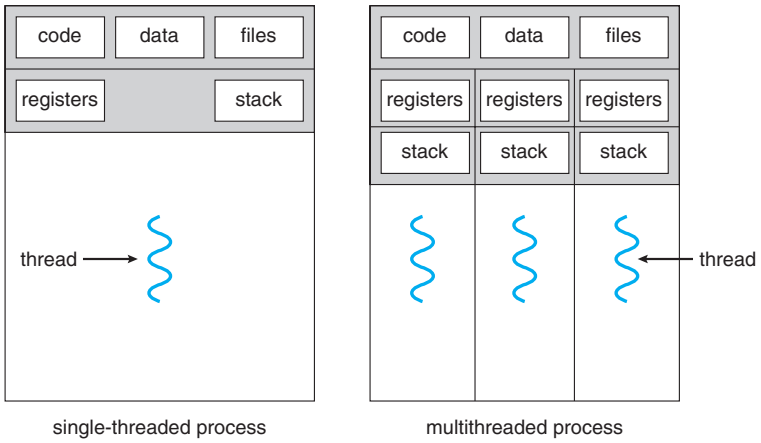

Figure 2.1: Single-Threaded and Multi-Threaded Processes

functions you could find a detailed reference on the website of WiringPi `http://www.wiringpi.com/reference`

Before using the WiringPi I/O library, you need to include its header file in your programs:

```
1 #include <wiringPi.h>
```

Due to the fact that you will be using a non-standard library, do not forget to link the library when compiling your file. Therefore you could use the following line:

```
-I/usr/local/include -L/usr/local/lib -lwiringPi
```

When you have successfully included the library you are able use the WiringPi functions. The first thing you have to do is to call the function `wiringPiSetup()`:

```
1 // Setup WiringPi
2 wiringPiSetup();
```

Once you have called the function `wiringPiSetup()` you are able to set the mode of the pins. Each pin can be set to several modes and therefor you have to select the right one in order to operate the pin correctly. To select the different pin modes you could use the function `pinMode(int pin, int mode);`:

```
1 // Set LED1 pin to output
2 pinMode(LED1, OUTPUT);
```

Another function you will be going to use is the function to change the output value of the LED pins in order to turn the LED on or off. You could change the output value low or high by the function `digitalWrite (int pin, int value);`

```
1 // Turn LED on
2 digitalWrite(LED1, HIGH);
```

```
3
4 // Turn LED off
5 digitalWrite(LED1, LOW);
```

The previously stated functions will allow you to turn the LED's on or off, more specifically the first function turns LED1 on, and the second one turns it off. The functions mentioned below can be used to control the LED's with a Pulse Width Modulated signal (PWM). PWM can be used to control the brightness of the LED's. Additionally PWM can be used to fade-in and fade-out the LED's. The LED's which you will be going to control with PWM signals must be initialised different than just turning LED's on and off. The function `softPwmCreate (int pin, int initialValue, int pwmRange);` is used to initialise the LED. In this function you have to enter a initial value (0 for off and 100 for fully on) and the pwmRange which should be 100.

The function `softPwmWrite(int pin, int value);` is used to update the PWM value of the LED. We use for PWM the soft PWM library because there is only one pin on the Raspberry Pi which allows hardware PWM. Soft PWM means a thread is created which controls the LED to the corresponding PWM value. Before you can use the functions for PWM you have to include the soft PWM library. `#include <softPwm.h>`

```
1 // Include headers
2 #include <wiringPi.h>
3 #include <softPwm.h>
4
5 // Initialise pin for PWM output
6 softPwmCreate(LED1, 0, 100);
7
8 // Update the PWM value of a LED
9 softPwmWrite(LED1, 50);
```

Assignment 2.1

The objective of this assignment is creating a new thread. You have to include the POSIX Thread library in order to create a thread in your main function. The only thing you have to implement in your main function is the creation of a thread, waiting for the thread to finish and killing the program. The complete list of objectives of this assignment are listed below.

Objectives

Objectives for a correct implementation of Assignment 2.1
The main function must create a thread, wait till the thread is finished and finally close the program. The objectives of the thread are listed below.

- Print the thread ID and function name of the thread
- Create a counter which counts from 1 to 10 and wait 1 second between each increment

Signature Assistant:

Assignment 2.2

In this assignment you will - just as in assignment 2.1 - create a thread. Basically the same main functions can be used. However, instead of using a timer you will use the input/output port of the Raspberry Pi. In this assignment you have to use the output function of this port to blink multiple LED's. The objectives of this assignment are listed in the grey box below.

Objectives

Objectives for a correct implementation of Assignment 2.2
The main function must create a thread, wait till the thread is finished and finally close the program. The objectives of the thread are listed below.

- Initialise the LED pins as output
- Create a certain pattern with the LED's
- Close the thread after 20 seconds

The pattern named above may be every pattern you want. It could be some difficult pattern or just simple blinking of LED's

Signature Assistant:

2.2 Bonus Question: Sudoku Solution Validator

A Sudoku puzzle uses a 9×9 grid in which each column and row, as well as each of the nine 3×3 subgrids, must contain all of the digits $1 \cdots 9$. Figure 2.2

presents an example of a valid Sudoku puzzle. This project consists of designing a multithreaded application that determines whether the solution to a Sudoku puzzle is valid. There are several different ways of multithreading this application. One suggested strategy is to create threads that check the following criteria:

- A thread to check that each column contains the digits 1 through 9
- A thread to check that each row contains the digits 1 through 9
- Nine threads to check that each of the 3×3 subgrids contains the digits 1 through 9

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

Figure 2.2: Solution to a 9 x 9 Sudoku puzzle

This would result in a total of eleven separate threads for validating a Sudoku puzzle. However, you are welcome to create even more threads for this project. For example, rather than creating one thread that checks all nine columns, you could create nine separate threads and have each of them check one column.

Passing Parameters to Each Thread

The parent thread will create the worker threads, passing each worker the location that it must check in the Sudoku grid. This step will require passing several parameters to each thread. The easiest approach is to create a data structure using a struct. For example, a structure to pass the row and column where a thread must begin validating would appear as follows:

```
1 /* structure for passing data to threads */
2 typedef struct {
3    int row;
```

```
4     int column;
5 } parameters;
```

Both Pthreads and Windows programs will create worker threads using a strategy similar to that shown below:

```
1 parameters *data = (parameters *) malloc(sizeof(
    parameters));
2 data->row = 1;
3 data->column = 1;
4 /* Now create the thread passing it data as a
    parameter */
```

The data pointer will be passed to the `pthread create()` (Pthreads) function, which in turn will pass it as a parameter to the function that is to run as a separate thread.

Returning Results to the Parent Thread

Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent. One good way to handle this is to create an array of integer values that is visible to each thread. The ith index in this array corresponds to the ith worker thread. If a worker sets its corresponding value to 1, it is indicating that its region of the Sudoku puzzle is valid. A value of 0 would indicate otherwise. When all worker threads have completed, the parent thread checks each entry in the result array to determine if the Sudoku puzzle is valid.

Signature Assistant:

Session 3

Multithreading

This lab session will dive into multithreading. In the previous assignments you learned the basics with regards to threads on a Linux system. In this assignment you will create a program that contains multiple threads. The objectives of this session are listed below:

- Learn the concept of multithreading
- Get to know the advantages and disadvantages of multitasking

3.1 Assignments

A multithreaded process contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing within the process, economy, and scalability factors, such as more efficient use of multiple processing cores. In the following assignments you have to write a program that consist of multiple threads. From these assignments it should become clear that threads are more easily to use than the regular processes.

Assignment 3.1

In the previous session you have learned how to create a single process. Processes can come in extremly handy but they suffer from a single point of failure. In this assignment you will get to know processes in a more advanced way, hoping to eludicate the advantages and disadvantages of multithreading processeses. You will use the LED extention board to visualy see what is going on. The objectives of this assignment are listed in the grey box below.

Objectives

Objectives for a correct implementation of Assignment 3.1
Create a program (main function) that creates 3 threads. The program is only aloud to finish after the execution of all 3 threads! Lastly, you have to alter the output such that it outputs the currently active thread, it should correspond to the following structure: `[Thread?]:`.

Thread 1 must implement:

- Blink the LED's with a certain self choosen pattern. This pattern could be whatever you want

Thread 2 must implement:

- Count from 1 to 20 with 1 second in between each increment and print the updated counter value

Thread 3 must implement:

- Create a 10x10 array with random numbers between 0 and 100 and print this array

Signature Assistant:

Assignment 3.2

In the previous assignment you created three processes that operated in a multi-threaded regime. This assignment has the same method of approach. The difference lies in the difficulty of the assignment as this assignment will introduce binary counting! The objectives of this assignment are listed below.

Objectives

Objectives for a correct implementation of Assignment 3.2
Use the source code of the previous assignment and alter it such that the output meets the following requirements. Keep in mind that you still have to output to currently active thread according to the same convention as mentioned in assignment 3.1

Thread 1 must implement:

- Use the LED's on the extention board to make a binary counter which binary counts from zero to 10 and back to zero with 1 second in between each incremation. So for the value 5_{dec} the binary value is 0101_{bin} outputed and only LED's 1 and 3 should be active!

Thread 2 must implement:

- Make a counter from 10 to 0 and back to 10.

Thread 3 must implement:

- Rearrange the array created in the previous assignment in such a way the first column has ascending numbers

Signature Assistant:

Assignment 3.3

Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers:

```
90 81 78 95 79 72 85
```

The program will report: The average value is 82 The minimum value is 72 The maximum value is 95

The objectives of this assignment are listed below

Objectives

Objectives for a correct implementation of Assignment 3.3

- The program must be able to listen to input values (sanitize the input to check if it is valid!)
- Compute the average value
- Compute the minimum value
- Compute the maximum value
- Compute the median value
- Compute the standard deviation
- Determine how many threads you should be using to assure a fast execution of the program.

Signature Assistant:

3.2 Bonus Question: Multithreaded Sorting Application

Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads (which we will term **sorting threads**) sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third thread — a **merging thread** — which merges the two sublists into a single sorted list.

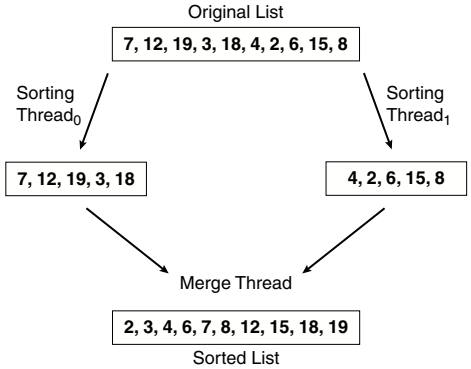

Figure 3.1: Multithreaded Sorting

Because global data are shared cross all threads, perhaps the easiest way to set up the data is to create a global array. Each sorting thread will work on one half of this array. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sublists into this second array. Graphically, this program is structured according to Figure **??**. This programming project will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the starting index from which each thread is to begin sorting. Refer to the instructions of section 2.2 for details on passing parameters to a thread. The parent thread will output the sorted array once all sorting threads have exited.

Signature Assistant:

Session 4

Scheduling of threads

In the previous sessions you have learnt what a thread is, how to create one and how to create multiple threads. But can the total running time of this execution be optimised? This chapter is going to introduce scheduling as a possible optimisation. The objectives of this session are listed below.

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems.
- To understand the difference between process scheduling and thread scheduling.

4.1 Assignments

In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximise CPU utilisation. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively.

Several scheduling algorithms exists.

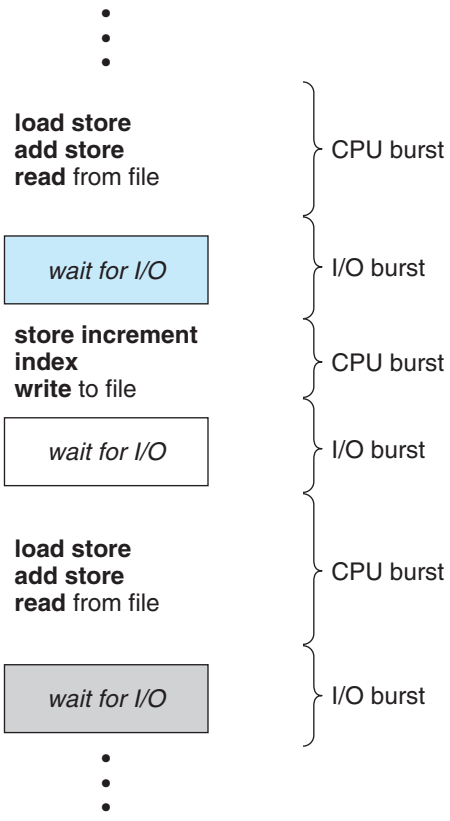

Figure 4.1: Alteration of IO and CPU bursts

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. You can find some scheduling algorithms below.

- First-Come, First-Served Scheduling (FCFS)
- Shortest-Job-First Scheduling (SJF)
- Priority Scheduling
- Round-Robin Scheduling (RR)
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

Assignment 4.1

In order to schedule processes fairly, a round-robin scheduler generally employs time-sharing, giving each job a time slot or quantum (its allowance of CPU time), and interrupting the job if it is not completed by then. The job is resumed next time a time slot is assigned to that process. In the absence of time-sharing, or if the quanta were large relative to the sizes of the jobs, a process that produced large jobs would be favoured over other processes. Round Robin algorithm is a pre-emptive algorithm as the scheduler forces the process out of the CPU once the time quota expires. You will use Round-Robin as a scheduling algorithm in this assignment.

Use your your solution to assignment 3.2 and change the scheduling method. The threads must have different priorities and use Round-Robin as the desired scheduling method. Implement this assignment such that you can show that the priorities are changed and have indeed influenced the way of executing the code.

Objectives

Objectives for a correct implementation of Assignment 4.1

- Implement Round-Robin as scheduling algorithm.
- Prove that priorities change when implementing RR.

Signature Assistant:

Assignment 4.2

By far the simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. This assignment has the same objectives as assignment 4.1. The only difference is that you will be utilising the FCFS scheduling algorithm as apposed to the Round-Robin scheduling algorithm.

Objectives

Objectives for a correct implementation of Assignment 4.2

- Implement FCFS as scheduling algorithm.
- Beeing able to explain in detail the difference between FCFS and other scheduling algorithms!

Signature Assistant:

4.2 Bonus Questions

Below you will find three questions. If you answer all these questions correctly, you will be rewarded a bonus!

Question 1

Which of the following scheduling algorithms could result in starvation? You will have to explain to the assistants which one(s) could result in starvation so be prepared!

- first-come, first-served
- Shortest job first
- Round Robin
- Priority

Signature Assistant:

Question 2

Consider a variant of the RR scheduling algorithm in which the entries in the ready queue are pointers to the PCBs.

- What would be the effect of putting two pointers to the same process in the ready queue?
- What would be two major advantages and two disadvantages of this scheme?
- How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

Signature Assistant:

Question 3

Consider the scheduling algorithm in the Solaris operating system for time-sharing threads.

- What is the time quantum (in milliseconds) for a thread with priority 15? With priority 40?
- Assume that a thread with priority 50 has used its entire time quantum without blocking. What new priority will the scheduler assign this thread?
- Assume that a thread with priority 20 blocks for I/O before its time quantum has expired. What new priority will the scheduler assign this thread?

Signature Assistant:

Session 5

Synchronisation of Threads

In the previous chapter you have learned what scheduling is and some algorithms that can be used to schedule threads. This chapter will introduce the scheduling of threads.

5.1 Assignments

In concurrent programming, a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task, or process will have to wait for a fixed time to enter it, this is called bounded waiting. Some synchronisation mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore.

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.
- To examine several classical process-synchronisation problems.
- Understand what mutexes are.
- Understand what semaphores are.

Assignment 5.1

Objectives

Create a program that consists of 3 threads. Prioritise the threads such that one thread has a higher priority than the other thread. You program should employ the Round-Robin timing schedule. Furthermore, create a variable that is accessible throughout all threads. This should be your counter variable. Initialise this variable to zero.

Thread 1 must implement:

- Increment the counter 15 times with 1. Keep at least 1 second in between each incrementation.

Thread 2 must implement:

- Decrease the counter of the first thread 10 times with steps of 1. Wait 1 second in between each step.

Thread 3 must implement:

- Display the current value of the counter on the LED extension board in a binary representation.

Ask yourself: Why does this work, or why doesn't this work?

Signature Assistant:

Assignment 5.2

In Assignment 5.1 you have created a counter that ends at 5. In this assignment you will alter/extend the previous assignment.

Objectives

Use the previous assignment and improve it such that the counter increases to 15 and then counts back to 5. To realise this behaviour you have to use mutexes (Chapter 5.5 in your book). You must be able to explain the working principle of mutexes.

Signature Assistant:

Assignment 5.3

Another method of implementing Assignment 5.2 such that the counter increments to 15 before it is decremented to 5 is by using semaphores.

Objectives

Use the previous assignment and alter it such that the counter increases to 15 and then counts back to 5. To realise this behaviour you have to use semaphores (Chapter 5.6 in your book). You must be able to explain the working principle of semaphores.

Signature Assistant:

Assignment 5.4

Could you think of other solutions solving the problem of assignment 5.3?

Signature Assistant:

5.2 Bonus Question: The Sleeping Teaching Assistant

A university computer science department has a teaching assistant (TA) who helps undergraduate students with their programming assignments during regular office hours. The TA's office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the TA to ask for help. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time. Using PThreads, mutex locks, and semaphores, implement a solution that coordinates the activities of the TA and the students. Details for this assignment are provided below.

The Students and the TA's

Using Pthreads (Section 4.4.1 in your book), begin by creating n students. **Each will run as a separate thread**. The TA will run as a separate thread as well. Student threads will alternate between programming for a period of time and seeking

help from the TA. If the TA is available, they will obtain help. Otherwise, they will either sit in a chair in the hallway or, if no chairs are available, will resume programming and will seek help at a later time. If a student arrives and notices that the TA is sleeping, the student must notify the TA using a semaphore. When the TA finishes helping a student, the TA must check to see if there are students waiting for help in the hallway. If so, the TA must help each of these students in turn. If no students are present, the TA may return to napping.

Perhaps the best option for simulating students programming — as well as the TA providing help to a student — is to have the appropriate threads sleep for a **random** period of time.

PThread Synchronization

Coverage of PThread mutex locks and semaphores is provided in Section 5.9.4 of your book. Consult that section for details.

Signature Assistant: