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

Operating Systems (TI2726-C)
Lab Session 2: Multithreading and LEDs

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

March 01, 2018

Deadline: 7 March 2017 23:50

Session 2 of this lab contains several exercises about threads. In Session 1 you learned about processes and how they work. In this assignment you will use threads, and these threads will be used in all later sessions of this lab. The objectives of this session are to:

- Learn what a thread is and how you can create one
- Learn how to use the I/O interface of the Raspberry Pi to turn on LEDs
- Learn the difference between sleeping and busy waiting
- Learn how to change the priority and scheduling policy of a thread
- Get some experience with multithreading and what that entails

Using I/O on the Raspberry Pi

This tutorial will provide information with regard to the input/output (I/O) functionalities of the Raspberry Pi. To make it easy, we will use a library that provides a set of functions to communicate with the I/O pins of the Raspberry Pi. This lab will use the WiringPi library. This tutorial will explain the basic functions you have to use during this lab. If you want to use more advanced functions, you can find a detailed reference on the website of wiringPi at `http://www.wiringpi.com/reference`.
Before using the WiringPi I/O library, you need to include its header file in your programs:

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

We have to link against the library in the compilation process. The library is installed in the C library on the Linux image used in the lab. To link against it when compiling using gcc on the Pi, we use the following gcc flags:

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

This line is part of the makefile provided last week.
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()`.

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

You also have to define the constants for the LED output pins (these can be found in the processor manual):

```
// define led output pins
#define LED1 7
#define LED2 0
#define LED3 2
#define LED4 3
```

Now you are ready to set the mode of the pins. Each pin can be set to several modes, and therefore you have to select the right one in order to operate the pin correctly. To select the different pin modes you can use the function `pinMode(int pin, int mode)`, for example:

```
// Set LED1 pin to output
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 can change the output value to low or high using the function `digitalWrite(int pin, int value)`.

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

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

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). You can use the same makefile as last week.

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.

2.1

In this assignment you will build an application that accepts a number between 0 and 15 as input, and then displays this number (in binary!) on the LEDs on the Pi. This input must be given as a command line parameter. So if your program is named `2_1`, it can be executed by calling `./2_1 11` and the LEDs will then display `1 0 1 1`. The objectives of your program are to:

1. Check if a command line argument is given and if it is in the valid range [0, 15]. Display an error message if this is not the case.
2. Convert the command line argument to binary.
3. Display this binary number on the Pi's LEDs.

2.2

In this assignment you will build an application that accepts numbers between 0 and 15 as input while the application is running, and then has the Pi's LEDs count towards this number over time. The number is only allowed to go up or down one step every 200ms. When you enter another number as input while the application is running, the LEDs must count towards that number instead. You will create separate threads for reading input, and for counting in binary with the LEDs. The objectives of this assignment are to:

1. Create a thread that continually listens to command line input, and checks if it is in the valid range [0, 15]. Display an error message if this is not the case. Use this number as the goal of your counter.

2. Create a thread that incrementally counts towards the goal number (provided by your other thread), allowing only a change of -1 or +1 every 200 milliseconds (use usleep!). On the same thread display the current number in binary using the Pi's LEDs.

3. When a new goal is entered before the current one is reached, count to the new goal instead.

4. The program should keep listening to new goal values and counting to them (even after the first goal is reached) until the goal -1 is entered. When -1 is entered both threads should stop.

2.3

In this assignment you will extend your code from assignment 2.2 with a third thread running at the same time as the other two threads. This thread reads a file (provided on Brightspace) filled with doubles, and performs a computation on these numbers.
Your objectives are to:

1. Create a thread that reads a file with a double on each line. This file is provided on Brightspace. Your code should be able to read files of different sizes.

2. On this same thread, for every number `n` in the file, calculate `n = atan(tan(n))`.

3. On this same thread, sum all these numbers, and print the result, as well as the time it took to complete the entire calculation. **(this is important for the next assignment!)**

4. Ensure the other two threads still function correctly while processing the file.

2.4

In this assignment you will extend your code from assignment 2.3 to investigate the differences between busy waiting and sleeping. Furthermore, you will look at thread priorities.

You will need the following snippet for this assignment. This is how inexperienced programmers might implement waiting instead of using usleep (please note this is a bad example):

```
void busyWait(int s) {
    clock_t now = clock();
    while (clock() < now + s) {};
}
```

You will also need the following snippet for this assignment. This is how you can set a priority for your thread before you start it. Don't change the order of the function calls, or the code will no longer work.

```
pthread_t t1;
pthread_attr_t attr;
pthread_attr_init(&attr); //Initialize your thread attributes
//Makes sure the thread does not inherit the attributes of its parent:
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
//Sets the scheduling policy to Round Robin:
pthread_attr_setschedpolicy(&attr, SCHED_RR);
struct sched_param param;
param.sched_priority = 10; //Sets the priority of the thread
//Includes the priority in the attribute:
pthread_attr_setschedparam(&attr, &param);
//Creates the thread with the defined policy and priority:
pthread_create(&t1, &attr, threadFunction, NULL);
```

Your objectives are to:

1. Replace usleep() with busyWait() in your code. Look at the difference in execution time of the work thread. Please ensure that you can still control the LEDs.
2. In a short comment explain the difference between sleeping and busy waiting.
3. While still using busyWait you can speed up the execution of the thread that reads the file by giving it a higher priority than the other two threads. Give the other two threads the same lower priority and use Round Robin scheduling. You can give each thread a priority and policy using the snippet above. Again, look at the execution time.
4. In a short comment explain why the LEDs do not function correctly as long as the thread that reads the file (with a higher priority) is not yet finished.

Bonus Exercise: 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 1,2,. . .,9. The next figure presents an example of a valid solution to a Sudoku puzzle. This project consists of designing a multithreaded application that determines whether the solution to a Sudoku puzzle is valid.
There are several 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

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 can create nine separate threads that each check one column.

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 1: Sudoku

Loading a sudoku puzzle Your program must load the sudoku grid from a file (example file `sudoku.txt` provided on Brightspace). Please ensure that it works with other files of the same format as well.
Hint: Be aware that line endings differ across different operating systems.

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:

```
/* structure for passing data to threads */
typedef struct {
   int row;
   int column;
} parameters;
```

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

```
parameters *data = (parameters *) malloc(sizeof(parameters));
data->row = 1;
data->column = 1;

/* Now create the thread passing it data as a parameter */

free(data); //free the memory after the thread is done with it
```

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 i-th index in this array corresponds to the i-th worker thread. If a worker sets its corresponding value to 1, it indicates 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 all entries in the result array to determine if the solution to the Sudoku puzzle is valid.