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

Operating Systems (TI2726-C)
Lab Session 3: Thread Safety and Thread Synchronization

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

March 07, 2019

Deadline: March 14 2019

Session 3 of this lab contains several exercises about thread-safety and thread synchronization. In session 2 of this lab, you learned about multithreading and controling the LEDs on your Pi. In this assignment, you will look at some of the issues that can occur when writing multithreaded applications. Finally, you will deal with the concept of pipes.

The objectives of this session are to:

- Learn what thread-safety is, and how to implement it
- Learn what thread-synchronization entails
- Learn how to PWM-control the LEDs on your Pi
- Learn what pipes are, and how to use them

Using PWM for your LEDs

This tutorial is an extension of the tutorial given in assignment 2, where we discussed how to control the LEDs on the Raspberry Pi. Previously, to use your LEDs, you wrote a value that was either high or low, and this would turn the led on or off completely.

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

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

Now, instead, we will control the LEDs using a Pulse Width-Modulated signal (PWM), which can be used to control the brightness of the LEDs. Additionally, it can be used to fade-in and fade-out the LEDs. The LEDs you are going to use with PWM must be initialized differently than when just turning them on and off. First off, you must include the softPwm library. The function `softPwmCreate(int pin, int initialValue, int range)` is then used to initialize the LED, and the function `softPwmWrite(int pin, int value)` is then used to update the LED. The following code snippet demonstrates this.

```
// Include headers
#include <wiringPi.h>
#include <softPwm.h> // Notice the extra included library

// Initialise pin for PWM output
softPwmCreate(LED1, 0, 100);

// Update the PWM value of a LED
softPwmWrite(LED1, 50);
```

Pipes in Linux

Piping in a Linux system connects the standard output of one application to the standard input of another application. When you use `printf()`, you would normally print to the terminal, but when using pipes you can send the data to another application. This application can read that data using `scanf()`. For example, let `numberGen` be a program that outputs some number every second, and let `num2str` be a program that receives numbers as input and prints them as a string:

```
[me@myComputer me]$ ./numberGen
3
5
1
[me@myComputer me]$ ./numberGen | ./num2str
 > three
 > five
 > one
```

Pipes are often used when a program outputs a lot of data. For example, `ps -aux` prints information on all currently running processes, which does not fit on a single screen. You can use `less` to make it possible to scroll through the output with the arrow keys. Another option would be to use `grep` to select only the processes you are interested in: `ps -aux | grep root` would only display lines containing "root". When that is still too much, it is also possible to use `ps -aux | grep root | less`.
If you want to use a text file as the input of your program, you can use the application `cat`, which prints the contents of a file to the standard output. For example:

```
[me@myComputer me]$ cat numbers.txt
7
11
0
[me@myComputer me]$ cat numbers.txt | ./num2str
 > seven
 > eleven
 > zero
```

If you want to write the standard output to a file you can use the `>` operator. For example, `cat file1.txt > file2.txt` copies the contents of `file1.txt` to `file2.txt`.
Apart from the standard output stream, there is also a standard error stream. For example, `make` prints its errors to the error stream. To write the output from an error stream to a file, use `2>`. You can also use `&>` to write both the standard and the error streams to the same file. For example, `make all &> makeOutput.txt` would save all output from make to the file `makeOutput.txt`.

Pointers in C

Here a quick recap of pointers will be given. Pointers store memory locations. They can be used to change the value of a variable indirectly. To do this the reference(&) and dereference(*) operators are used.

```
int x;
int* p = &x; // create a pointer that points to x
             // p now stores the memory location of x
*p = 5;      // x now is equal to 5
int* p2 = x; // p2 now stores the value of x
             // This is something you generally don't want to do
             // The compiler will throw a warning saying that
             // an integer is converted to a pointer without a cast
*p2 = 6;     // BAD: the memory at location 5 is now set to 6
             // this will probably cause a segfault
```

Pointers can point to a variable that goes out of scope. When this happens the behavior of the pointer is undefined: the memory might still be at its last used value, the memory might be used by a different part of your program, or the memory might not be assigned to your program anymore and a segfault will occur.

To avoid this you can dynamically allocate memory using `malloc(<size in bytes>)`. To obtain the right size you can use the `sizeof` keyword. This way, you can allocate memory that remains allocated until it is deallocated using `free`. Unlike Java, there is no garbage collector in C, so you have to deallocate memory yourself. However, when a program is finished executing, all memory that is still allocated will be deallocated automatically. Nevertheless it is good practice to always deallocate all dynamically allocated memory, because it makes your code more reusable and it makes clear what the lifetime of the memory is.

```
int *p, *p2; // creates two pointers (each one should have a separate *)

if(1) {      // always execute block below
    int x = 5;
    p = &x;   // p points to local variable x
    p2 = malloc(sizeof(int)); // allocates memory for one int
    *p2 = 6;
}
printf("%d\n", *p);  // BAD: output not defined, because x went out of scope
printf("%d\n", *p2); // prints 6
free(p2);            // deallocates memory
```

It is also possible to allocate blocks of memory. The syntax for pointers to blocks of memory is the same as the syntax of arrays. Actually an array in C is nothing more than a pointer to a block of memory that will be deallocated when the array goes out of scope. Arrays can be implicitly cast to pointers or be send to a function expecting a pointer.

```
int arr[] = {1, 2, 3};
int *p = arr;
int *p2 = malloc(sizeof(int)*3); // dynamically allocate array of 3 ints

// these three commands will do the same:
arr[1] = 4;
p[1] = 4;
*(p+1) = 4;

free(p2);  // always deallocate your memory!
```

Pointers and dynamic memory allocation are powerful tools, but they can be confusing. When using pointers always keep in mind the scope of your memory. You should only use dynamically allocated memory when the size or the lifetime of your memory block are variable during runtime. When both the size and the lifetime of your memory block are known at compile time, you should use normal variables and arrays instead.

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.

3.1

In this assignment, you will let the LEDs on the Pi 'breath' in and out once. For this, you will make use of barriers (PThreads supports these), and PWM (described above).
The objectives for this assignment are:

1. Create a thread that slowly fades the LEDs from unlit to lit.
2. Create a thread that slowly fades the LEDs from lit to unlit.
3. Start both threads at the same time.
4. Create a barrier and use this barrier to ensure the second thread does not start working before the first thread has completed.

3.2

In this assignment, you will implement a queue data structure that will be used to provide data to your LEDs in Assignment 3.3. As we will be using multiple threads to access this queue, it is vital that it is thread-safe, which you will test.

We'll provide you with some code to get you started. You will have to implement the empty functions yourself. To ensure multiple threads can use the queue at the same time, please make adding and removing elements thread-safe through the use of mutual exclusion.

```
struct Node {
    int brightness, duration; //Stored values
    struct Node* next;        //Next node in line
};

struct Queue {
    struct Node* front;       //First node in the queue
    struct Node* back;        //Last node in the queue
    pthread_mutex_t mutex;    //Used for mutual exclusion
};

//Adds a new node to the back of the queue
void addToQueue(struct Queue* queue, int brightness, int duration) {
    // Implement
}

//Removes a node from the front of the queue
//The memory of the removed node has to be freed
//Returns the values of the removed node via pointers
//If the queue is empty return -1 for both the brightness and the duration
void removeFromQueue(struct Queue* queue, int* pBrightness, int* pDuration) {
    // Implement
}

//Returns the length of the queue
int queueSize(struct Queue* queue) {
    // Implement
}

//Initializes the values of an already allocated queue struct
void initQueue(struct Queue* queue) {
    // Implement
}
```

Your objectives for this assignment are:

1. Implement the queue data structure provided above.
2. Use the mutex to make the functions thread safe: the functions can be called from different threads without problems.
3. Spawn two threads, and have these threads both add 100,000 elements to the same queue simultaneously.
4. Afterwards, print the size of the queue and make sure it is 200,000 elements.

5. Spawn two threads, and have these threads both remove 100,000 elements from the same queue simultaneously.

6. Each time you run `removeFromQueue` check if -1 is returned for the brightness and duration. This should not be the case, because the same amount of nodes is added and removed.

7. Afterwards, print the size of the queue and make sure it is 0 elements.

8. Please explain in the comments why mutual exclusion is necessary here. It is interesting to also see what happens without mutual exclusion (but please submit the version with mutual exclusion).

3.3

In this assignment, you will re-use the queue you built in assignment 3.2 to provide data to the LEDs. You're going to build a small system where you can easily PWM-control all four leds, where each led has a separate queue and thread, and there is another fifth thread to handle user input. You will also quickly look at piping, where the output of one process is continuously provided as input to another.

We have provided a small textfile called lightshow.txt on Brightspace. This file stores numbers in the format `LED(0-3) Brightness(0-100) Duration(ms)`. You can use this text file (by piping it into your own program) to quickly test your code!

Your objectives for this assignment are:

1. Create four queues, one for each LED.

2. Create four threads, one for each LED, that fetch one by one the elements from their respective queues and set the LED to the indicated brightness for the specified duration. When the thread's queue is empty, turn off the corresponding LED.

3. Reuse the same functions for each LED. You are not allowed to have multiple functions with the same functionality only for different LEDs.

4. Create one additional thread, which accepts command line input of the form `LED(0-3) Brightness(0-100) Duration(ms)`; these data are then added to the correct LED's queue.

5. Ensure your program works correctly by piping output from the `lightshow.txt` into your own program.

Bonus Exercise: How to screw it all up

In this assignment, you will try to do some bad things with threads. We want you to create two small programs (so you upload a .zip file: 3_b.zip) that demonstrate some fatalities that can occur when multiple threads come into play. Please demonstrate in these programs - clearly marked - an instance of thread starvation and an instance of thread deadlock between three threads.
Your objectives for this assignment are:

1. Create a program with two or more threads, where starvation occurs (on the raspberry pi) for one of these threads.

2. Create a program with three threads, where deadlock occurs between these threads.
3. Ensure it is clearly visible to us, the TAs, that starvation and deadlock indeed occur.