Lab 3 Report

Tobias L & Özgun M (Group 5)

April 14, 2016

The purpose of this lab was to learn how to work with POSIX-threads and setting the priority of the threads. The main task was to make a thread sleep without using the conventional sleep-functions in time.h such as clock_nanosleep() and sleep().

1 busy_wait (A)

To put a delay on a thread without using these particular functions a spinning busywait-function was created to make the processor artifically be delayed, but still execute other tasks.

```
void busy_wait(const long nsec)
{
    long diffns = 0;
    int diffs = 0;
    struct timespec ts1, ts2;
    clock_gettime(0, &ts1);
    clock_gettime(0, &ts2);
    while (diffns < nsec)
    {
        clock_gettime(0, &ts2);
        diffns = ts2.tv_nsec - ts1.tv_nsec;
        diffs = ts2.tv_sec - ts1.tv_sec;
        diffns += diffs * BILLION;
    }
}
```

As one can see from the algorithm above, we have a parameter which takes a long argument. It's const because as the program was running the variable was for some reason changed during execution time. The while loop is running for as long as the time-difference is less than the argument nanoseconds. In the while-loop a difference check is running to compare the two timers and how long they've progressed through the timer. This makes an artificial "sleep" but instead of stopping the thread it's running an unnecessary process, which is an unnecessary strain on the CPU.

2 Two tasks with priorities (B)

In this section of the lab, there were a few observations that had to be made in regards to thread-creation and priority assignment. Task1 was controlling the 1st LED and Task2 was controlling the 2nd LED.

2.1 Creating two tasks at different times (B.1)

2.1.1 First create Task1 and then Task2

2.1.2 First create Task2 and then Task1

Creating one or the other task before the other doesn't make any difference.

2.2 Priority assignment I (B.2)

Task1 in this test was using the busy_wait-function to delay the processor, while Task2 was using a normal nanosleep-function. The observation produced the following results:

2.2.1 Creating Task1 before Task2

High priority Task1 runs and finish, then Task2 starts and finish

Low priority Task1 starts and Task2 starts estimated 0.5 seconds after. The program alterantes between the tasks till they return. The reason they're alternating is because of nanosleep in Task2. While Task2 sleeps, it allows Task1 to resume until Task2 is awake.

2.2.2 Creating Task2 before Task1

High priority Task1 and Task2 starts together. The program alternates between the tasks till they return. The reason of the alteranting is because of nanosleep in Task2. While Task2 sleeps, it allows Task1 to resume until Task2 is awake.

Low priority Task2 starts and instantly pauses. Task1 runs and finish. Task2 resumes and finish.

2.3 Priority assignment II (B.3)

Task1 and Task2 were both using the busy_wait-function in this test. The observation produced the following results:

2.3.1 Higher priority to Task1

Task1 starts and doesn't let Task2 start before Task1 is done.

2.3.2 Higher priority to Task2

Task1 starts. Task2 starts and interupts/pauses Task1. (Sometimes Task2 interupted Task1 when the LED was on, so Task1 continued to shine while Task2 was running. This is caused by the higher priority for Task2. When Task1 was just about to preemptive, Task2 started before it was able to shut down. So it would run parallell to Task1.). Task2 runs until it reaches its finish time THEN Task1 continues to run.

3 Three tasks with synchronization (C)

In this exercise three tasks had to be made. Two tasks (Task1 and Task2) were going to be from the previous exercises and Task3 was going to read in a value from terminal. The value that was read was going to determine the brightness of the LEDs that was controlled by Task1 and Task2 respectively. Instead of reading a float value, a struct was created which the attributes LED-number and the value of the brightness.

```
typedef struct
{
    int led;
    int brightness;
} Light;
```

The value that is entered is still in the format of n.m where n is the led-number and m is the brightness-value.

Since the threads all share the same global variable it had to be locked by a global mutex, to prevent the threads from accessing the variable at the same time. Testing this program at the end caused some unexpected errors with the thread-cancel - functions which was solved by entering handle SIG32 nostop
to prevent the program from breaking in the middle of closing the program.

Task3 was reading a value by scanf. The read value would only put into the global variable if it met the conditions for execution, otherwise it would just stay in the local variable until it was overwritten by the next input. The tasks that were reading the value of the global variable also had a mutex lock, when it was reading and had met the conditions for proceeding. One optimisation that could be done is to check if the condition is fulfilled before going into the specific function that runs the LEDs. The code can be seen in separate files.