

## Lab#2: Introduction mutex synchronization and Ncurses library

Leonardo Fusser (1946995)

### Objectives:

- Learn how to use the Ncurses library.
- Learn how to use the mutex to synchronize two threads.
- Learn how to improve CPU usage.

**Material:** Windows - Visual Studio

**To hand in:** No report to hand-in. Answers to all questions.

- Indented and commented multi-source file on Git Hub.
- All functions must have a prolog header.
- All files must have their prolog headers: file name, description, date, author, and version history.
- Use of symbolic constant is mandatory, all macros in capital letters.
- Answer all questions by filling in the lab sheets using a computer. Submit this document to GitHub.
- Don't erase lines of code of a previous step but comment it out.

If the requirements are not fulfilled, the student will be asked to re-submit.

### File system structure:

The file system organization is as listed below. You must populate only the file in **orange**.

```
➤ pdcourses_test
  ➤ laboratory
    ➤ ncurses_init.c
    ➤ project_files
      ➤ mainLab2.c
    ➤ header
      ➤ public.h
      ➤ ncurses_init.h
```

## **Lab Work**

Accept the following invitation:

<https://classroom.github.com/a/ETfJ3rgP>

You will get a repo for the next few labs:

<https://github.com/Embedded-OS-Vanier/lab2-mutex-ncurses-your-name>

Clone it.

### **Ncurses crash course:**

ncurses (new curses) is a programming library providing an application programming interface (API) that allows the programmer to write text-based user interfaces in a terminal-independent manner.

Example:

```
init_ncurses();  
/* Menu */  
attron(GREEN_WHITE);  
mvprintw(2,0, "Test program"); // prints the message at line 2 column 0  
mvprintw(3,0, "Use the following keys:"); // prints at line 3 column 0  
mvprintw(4,0, "Exit: x - Up: Q - Down: A"); // prints at line 4 column 0  
attroff(GREEN_WHITE);  
refresh(); /* Print it on to the real screen */
```

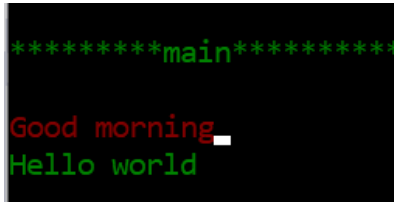
Console output result:

```
Test program  
Use the following keys:  
Exit: x - Up: Q - Down: A
```

## Part 1: Synchronization

Create two threads. Each thread must print a message forever in a loop. The loop must not have any delay (no Sleep() call).

Example:



```
*****main*****
Good morning_
Hello world
```

Run the program for at least 10 seconds.

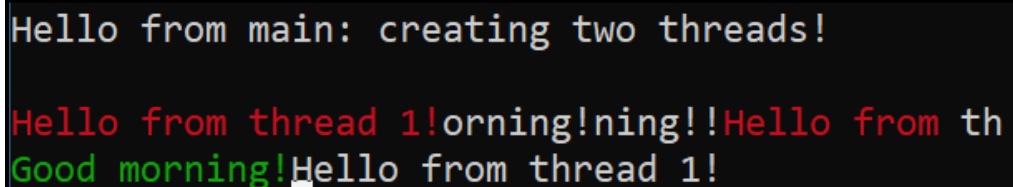
Since both threads are simultaneously printing to the console, there should be visible synchronization conflict. Explain:

After creating the two threads as indicated above, there is visible synchronization issues on the console output. The hardcoded messages being printed from the two thread routines seem to be constantly fighting each other for printing time. With the way the two thread routines are structured, they do not terminate and return back to the main thread. Refer to screenshot below.

The reason why this is occurring is because the two thread routines are not given enough time to complete printing their hardcoded messages to the console output. When one thread routine starts to print, its message being printed to the console will not finish in time before the thread routine hands over running time to the other thread routine. The other thread routine then takes over (goes into the running state) and has similar behaviour to the other thread routine.

A simple solution would be to use blocking delays inside the two thread routines, and that would ensure that there is no printing time conflicts between them.

Since the two thread routines are always in the ready state, the scheduler time slices the running time for them forever.



```
Hello from main: creating two threads!
Hello from thread 1!orning!ning!!Hello from th
Good morning!Hello from thread 1!
```

Figure 1. Output on console when the two thread routines try to print a message using the "mvprintw()" function. As explained above, the two threads are not given enough time to print their hardcoded messages to the console, resulting in the garbage shown above. The two threads never terminate and return to the main thread (as expected).

Modify your thread by creating a mutex protected function print():

```
void print(int l, int c, char *str);
```

The first parameter specifies the line number. The second parameter specifies the column number. And the last parameter points to the message to print.

Test your new function.

Does it fix the problem? Explain

Now, after creating a mutex protected print function for "mvprintw()", there are no more visible synchronization issues on the console output. The hardcoded messages being printed from the two thread routines are not fighting each other for printing time anymore. They appear normally on the console output, with little to no garbage being produced. Refer to screenshot below.

Using a mutex print function fixes the issue seen before because of how the mutex inside it works. With the new function, the mutex inside protects "mvprintw()" whenever it is used. This is possible because of the mutex lock and unlock functions found inside the mutex print function. With the mutex protection, only one message (from one of the two thread routines) can be printed at a time, which will avoid the garbage console output we saw before.

In a way, the mutex print function acts like a blocking delay, since the console cannot be overwhelmed with prints from the two thread routines.

The two thread routines will still be constantly in the running state and the scheduler will still have to time slice the running time for them forever.

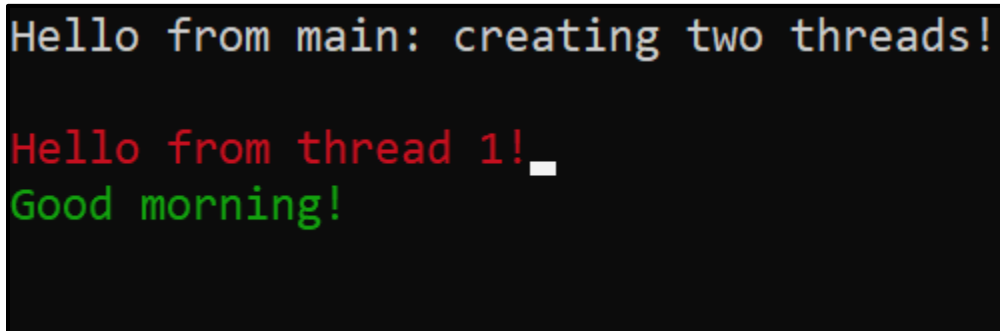


Figure 2. Output on console when the two thread routines use a mutex print function to display a message to the console. As explained above, the two thread routines are no longer fighting each other for printing time and there is no more garbage output on the console.

Give a quick demo to the teacher.

## Part 2: CPU usage

While the program runs, open the task manager, and find out the CPU usage in %

Name	Status	43% CPU
Apps (10)		
> Notepad++ : a free (GNU) source code editor (32 bit)		0%
pdcurses_test.exe (32 bit) (2)		X %
> Task Manager		5.5%
> Windows Explorer		2.0%

Is the CPU usage high? Explain

While running the program with the two thread routines, the amount of system resources needed jumps to a high amount. This can be seen in the task manager in the Windows OS. In my case, when the program runs, it uses around 34% of the CPU in order to run properly. This is not good for the system and is a bad design. See screenshot below.

The reason this occurs is simply because there are no blocking delays found in the two task routines. The effect is similar to when crude NULL loops are used inside thread routines. When the thread routines run, nothing slows them down, so they hog the system, which in turn causes the CPU load to jump to a significant level.

At this level, mutexes cannot prevent the CPU from being exhausted. Only blocking delays are able to do so, since they alter the behavior of the scheduler.

The two thread routines enter the running state, but never enter the blocked state, so the scheduler time slices the running time for them forever; the two thread routines will run forever and will not give the CPU a break. This also means that there is not much spare processing time left for the system to use.

Name	Status	100% CPU
> pdcurses_test.exe (32 bit) (2)		33.9%

Figure 3. Performance impact on my system while running the program. As explained above, since there are no blocking delays found in the two thread routines, they will hog the system similar to how crude NULL loops do as well. In my case, the program consumes around 34% of the CPU in order to print the messages to the console correctly.

To give slack time to the system, add blocking delay in each thread. The blocking delay does not need to be long, 1mS should suffice.

Again, while the program runs, find out the CPU usage in %

Is the CPU usage high? Explain

With the blocking delays added to the two thread routines, the amount of system resources needed no longer jumps to a high amount. This can be seen in the task manager in the Windows OS. In my case, when the program runs, it now uses only around 0% of the CPU in order to run properly. This is a much better improvement than what was seen previously. This is also a much better system design. See screenshot below.

The reason this occurs is simply because there are now blocking delays found inside the two task routines. This time, when the thread routines run, the blocking delays slow them down, so they can't hog the system anymore, which in turn causes the CPU load to drop significantly.

The two thread routines enter the running state, but now they enter the blocked state from time to time because of the blocking delays found inside them. The scheduler no longer time slices the running time for them forever, which in turn, makes the two thread routines give the CPU a break. Furthermore, the addition of the blocking delays allows for more spare processing time left for the system to use.



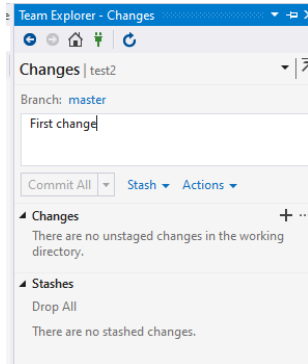
*Figure 4. Performance impact on my system while running the program with the added blocking delays. As explained above, since there are blocking delays found in the two thread routines, they will no longer hog the system like what was seen before. In my case, the program consumes only around between 0.2% and 0% of the CPU in order to print the messages to the console correctly.*

Give a quick demo to the teacher.

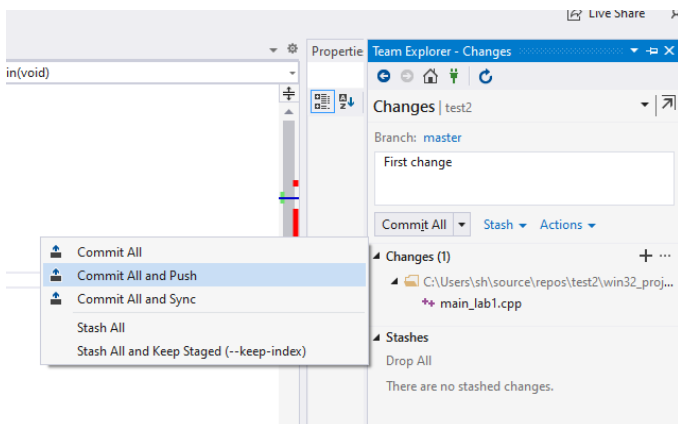
## Upgrading your repo:

Now it is time to update your repo.

To commit and upload the change, click Change Tab. Write a commit message:



Commit all and push to the repo:



At this point, your repo is populated with all the files.

***Make sure the repo in the web page is populated!***

Voila!

Approval before leaving!

### **After lab questions:** (to answer after the lab)

**Q1-** In part 1 explain the jumbled text that was printed to the console.

- In short, the reason why there was jumbled text/garbage being printed on the console output is because the two thread routines were constantly fighting for printing time. One thread routine would start printing a message to the console and then the other thread routine would interrupt the thread routine that was printing the message and decide to print its own message and vice versa. This behaviour repeated itself forever.

Refer to the detailed explanations found in part 1.

**Q2-** In part 2 explain the CPU usage drop.

- To summarize, the CPU load dropped a significant amount because there were blocking delays added to the two thread routines. This ensured that the two thread routines no longer hogged the system and cause too much stress for the CPU. If there were no blocking delays added, the two thread routines would just hog the system since nothing is slowing them down, and in turn cause an overload for the CPU.

Refer to the detailed explanations found in part 2.