Lab#1: Introduction to multi-threading

Leonardo Fusser (1946995)

Objectives:
- Learn how to create multiple POSIX theads.
- Learn how to synchronize threads using pthread_join() and pthread_exit()

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.

Lab Work

Part 1: Visual Studio and GitHub

Visual Studio 2019 Installation

Open the following file and follow the steps:

Install_Visual Studio 2019 Installation_CPP_GitHub extension.pdf

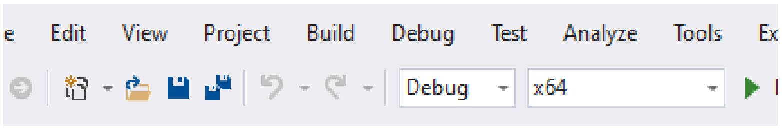
Working with Git Hub and Visual Studio

Open the following file and follow the steps:

Work with Git Hub_v2.pdf

Add path (x64 machine):

Make sure V.Studio is in x64 mode:

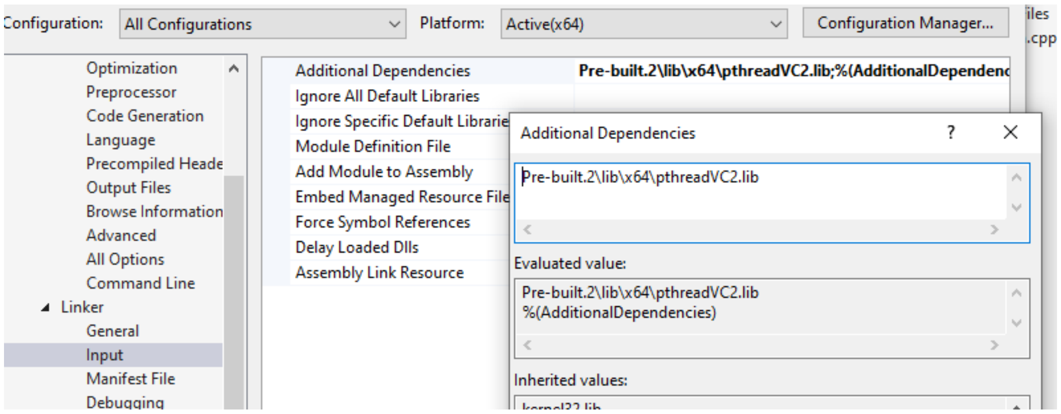


Open project properties:

In Linker-> input

Add:

Pre-built.2\lib\x64\pthreadVC2.lib



Configuration: All Configurations Platform: Active(x64)

Optimization
Preprocessor
Code Generation
Language
Precompiled Heade
Output Files
Browse Information
Advanced
All Options
Command Line
Linker
General
Input
Manifest File
Debugging

Additional Dependencies	Pre-built.2\lib\x64\pthreadVC2.lib;%(AdditionalDependenc
Ignore All Default Libraries	
Ignore Specific Default Librarie	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

Additional Dependencies

Pre-built.2\lib\x64\pthreadVC2.lib

Evaluated value:

Pre-built.2\lib\x64\pthreadVC2.lib
%(AdditionalDependencies)

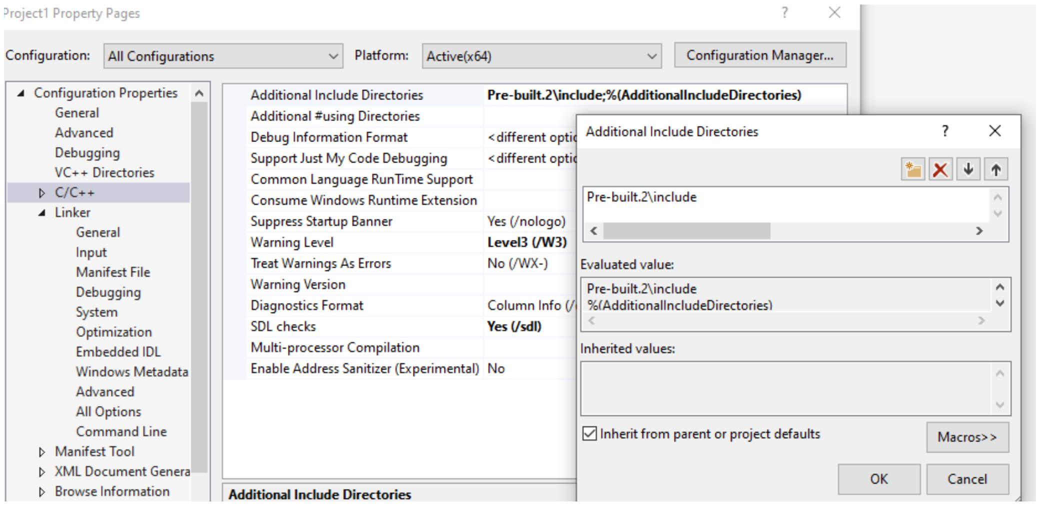
Inherited values:

kernel32.lib

In C/C++ -> Additional Include Directories

Add:

Pre-built.2\include



Project1 Property Pages

Configuration: All Configurations Platform: Active(x64)

Configuration Properties
General
Advanced
Debugging
VC++ Directories
C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Windows Metadata
Advanced
All Options
Command Line
Manifest Tool
XML Document Genera
Browse Information

Additional Include Directories	Pre-built.2\include;%(AdditionalIncludeDirectories)
Additional #using Directories	
Debug Information Format	<different opti
Support Just My Code Debugging	<different opti
Common Language RunTime Support	
Consume Windows Runtime Extension	
Suppress Startup Banner	Yes (/nologo)
Warning Level	Level3 (/W3)
Treat Warnings As Errors	No (/WX-)
Warning Version	
Diagnostics Format	Column Info (/
SDL checks	Yes (/sdl)
Multi-processor Compilation	
Enable Address Sanitizer (Experimental)	No

Additional Include Directories

Additional Include Directories

Pre-built.2\include

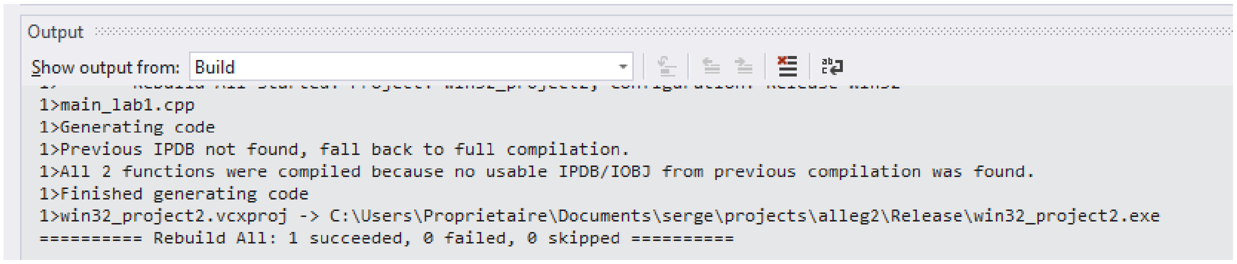
Evaluated value:

Pre-built.2\include
%(AdditionalIncludeDirectories)

Inherited values:

☑ Inherit from parent or project defaults

Also, it should compile successfully when you build it:



```
1>main_lab1.cpp
1>Generating code
1>Previous IPDB not found, fall back to full compilation.
1>All 2 functions were compiled because no usable IPDB/IOBJ from previous compilation was found.
1>Finished generating code
1>win32_project2.vcxproj -> C:\Users\Proprietaire\Documents\serge\projects\alleg2\Release\win32_project2.exe
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

Part 2: Multi- threading

One thread

In the main, declare and create a thread:

```
int main(void){
    pthread_t thread_id;
    printf("In main: creating a thread \n");
    pthread_create(&thread_id, NULL, thread1, NULL);
    Sleep(5000); // Blocks 5 seconds
    printf("Main: returns! \n");
}
```

Create a thread function `thread1` that prints a message every half a second forever. e.g., `Hello from thread1 .`

```
void* thread1(void* threadid){
    ...
}
```

Explain the console output

Does it print “`Main: returns!`”? Explain.

The console output shows that the message from inside the thread routine prints 10 times before the thread routine returns back to the main thread. A final message is displayed on the console indicating that the thread routine returned back to the main thread (main thread terminated).

This would make sense since the thread routine has only 5 seconds allocated in the running state before it is forced to return back to the main. This is because of the sleep blocking delay in the main thread.

The system behaves like this: the main thread is in the blocked state (for 5 seconds) which causes the thread routine to be in the running state (goes from ready to running) before it returns back to the main thread (after 5 seconds and is terminated). The main thread then returns from the blocked state (becomes unblocked), executes some final lines of code, and terminates.

Each print on the console from the thread routine lasts 500mS and prints for 10 times, so that would make the total running time for the thread routine 5 seconds (due to blocking delay).

Now in the main, replace the blocking delay by `pthread_exit().`
Don't erase lines of code of the previous step but comment it out. See code snippet below:

```
int main(void){
    ...
    //Sleep(5000); // Blocks 5 seconds
     pthread_exit(NULL);
    printf("Main: returns! \n");}
```

Explain the console output.

Does it print "`Main: returns!"?` Explain.

When the sleep blocking delay is replaced with "`pthread_exit()`", the console never reaches the point where a message is displayed indicating that the thread routine returned back to the main thread. This time, the thread routine is endlessly printing a message found inside of the thread routine.

This is because of how "`pthread_exit()`" affects the behaviour of the main thread. With this change applied in the main thread, the main thread essentially terminates without executing any last few lines of code but lets the thread routine run endlessly. The thread routine runs endlessly because of the infinite printing loop found inside of the thread routine.

In other words, the system behaves like this: the main thread terminates which causes the thread routine to be in the running state (goes from ready to running). The thread routine will not be able to return to the main thread since it got terminated. The thread routine will remain in the running state forever.

Note: even if the thread routine had a finite amount of time of being in the running state, the console will still never reach the point where a message is displayed indicating that the thread routine has returned back to the main thread. The same behaviour described above here will be observed.

Modify the thread function to print a message 10 times at a period of half a second. Therefore, the thread should terminate after 5 seconds (10 * 0.5 seconds).

Now, in the main, replace `pthread_exit()` by `pthread_join(thread1, NULL)`

```
int main(void){
     ...
     pthread_join(thread1, NULL);
     printf("Main: returns! \n");
}
```

Explain the console output.

Does it print "`Main: returns!`"? Explain.

When "`pthread_exit()`" is replaced with "`pthread_join()`", and when the structure of the thread routine printing changed, the console now reaches the point where a message is displayed indicating that the thread routine returned back to the main thread. This time, the thread routine does not endlessly print a message found inside of the thread routine (because it can only print 10 times, with the same duration seen before).

This is because of how "`pthread_join()`" affects the behaviour of the main thread. With this change applied in the main thread, the main thread no longer terminates without executing any last few lines of code, but it is placed in a blocked state waiting for the thread routine to run until completion beforehand.

In other words, the system behaves like this: the main thread goes from the running state to blocked state which causes the thread routine to be switched to the running state (goes from ready to running). The thread routine will be able to return to the main thread since it will only last a finite amount of time. When the thread routine completes and terminates (returns back to the main thread), the main thread will then return from the blocked state (becomes unblocked), and it will execute any last few lines of code before terminating all together.

Two threads

Add another thread `thread2` almost identical to the first one. The only difference is that it prints a message 10 times at a period of one second. Therefore, `thread2` should terminate after 10 seconds (10 * 1 seconds).

Don't make the main join thread 2 yet:

```
int main(void){
     ...
     pthread_join(thread1, NULL); // joints only thread1 – not thread2 yet
     printf("Main: returns! \n");
}
```

Explain the console output.

Does it print "`Main: returns!`"? Explain.

When a second thread is added, the console output shows that the thread1 routine is the only one that gets terminated correctly before returning back to the main thread. The thread2 routine seems to run in parallel with the thread1 routine, but the thread2 routine does not terminate correctly. The thread1 routine prints its message 10 times (same duration seen before), whereas the thread2 routine only prints its message around 6 times (not the complete 1 second duration).

This is because of how "`pthread_join()`" affects the behaviour of the main thread. Simply put, since there is only one "`pthread_join()`" in the main thread (for the thread1 routine only), the main thread is in the blocked state until the thread1 routine terminates. Since "`pthread_join()`" does not exist for the thread2 routine, the main thread will never wait for it to terminate and return back. Instead, the thread2 routine is forced to terminate along with the thread1 routine because the main thread would have terminated after returning from the blocked state (in blocked state while it was waiting for the thread1 routine to terminate).

In other words, the system behaves like this: the main thread goes from the running state to the blocked state which causes the two thread routines to go from the ready state to the running state. The thread1 routine will be in the running state until it terminates and returns back to the main thread. When the thread1 routine returns back to the main thread, the main thread becomes unblocked and then terminates shortly afterwards. The thread2 routine will also be in the running state right until the point where the main routine terminates. When the main thread terminates, so will the thread2 routine.

Explain how to make `thread2` complete its 10 second loop.

In order for the thread2 routine to complete its 10 second loop, it also must be able to terminate correctly and return back to the main thread (by having its own "`pthread_join`" in the main thread).

When the thread2 routine has its own "`pthread_join`" in the main thread, the main thread will now not only wait for the thread1 routine to terminate and return, but it will also do the same thing for the thread2 routine (main thread remains in blocked state until the two thread routines return to the main thread so that it can terminate all together).

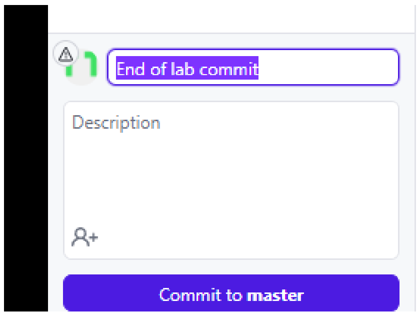
Modify the code to make thread2 complete its 10 second loop.

Give a quick demo of the last part 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 modified files.

Make sure the repo in the web page is populated!

Voila!

Approval before leaving!

After lab questions: (to answer after the lab)

Q1- Explain the differences between `pthread_join()` and `pthread_exit()`.

- The main difference between "`pthread_join()`" and "`pthread_exit()`" is how the main thread (initial thread) behaves.

  When "`pthread_join()`" is used in the main thread, the main thread becomes blocked and waits for any threads to run until completion (waiting for them to terminate) and return to the main thread. Afterwards, the main thread will become unblocked, and it will execute any final lines of code before terminating.

  When "`pthread_exit()`" is used in the main thread, the main thread terminates immediately without executing any final lines of code. But, if there are any other threads besides the main thread, they can still be in the running state even though the main thread has terminated.

Q2- Is it possible for a thread to keep running even if the main() has terminated?

- Yes, it is possible to keep a thread running even if the main thread has terminated.

  As described above, it is possible for a thread to remain in the running state if the main thread has terminated. This is only possible if "`pthread_exit()`" is used in the main thread.

  On the other hand, by using "`pthread_exit()`" in the main thread, "`pthread_exit()`" will terminate the main thread immediately before the main thread can execute any final lines of code.

Q3- What happens to the running threads when the main terminates?

- Simply put, when the main thread terminates, usually all other threads are terminated as well.

  The only exception is if "`pthread_exit()`" is used in the main thread. If "`pthread_exit()`" is used in the main thread, then the other threads will not terminate along with the main thread. Refer to above for complete explanation.