HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2022
LAB 8: PTHREADS (PART I)

This lab gives an introduction to parallel programming using POSIX threads, also called Pthreads. The following aspects are included in the lab:

(1) Creating threads

(2) Waiting for a thread to finish

(3) Using the input argument to the thread function for input and/or output

(4) Dividing work between threads

(5) Checking what speedup we get when varying the number of threads used

(6) Using mutex variables to achieve safe access to shared data

1. Main Tasks

Task 1

The code for this task is in the `Task-1` directory.

Here you are given a small code that uses Pthreads. In the main() function, the main thread calls `pthread_create()` to create a new thread. The third argument to `pthread_create()` is a function pointer. That is how we decide what the thread should do: the new thread that is created will run that function. In this case the function is called `the_thread_func()` but of course the function could have any name you want, just like any function in your program.

After creating the thread, the main() function calls `pthread_join()` which means that it waits for the other thread to finish.

To get access to the Pthreads functionality you need to include the `pthread.h` header file.

When building a program that uses Pthreads, you need to link with the Pthreads library, which is done by adding `-lpthread` when linking, in the same way as we need to link with e.g. `-lm` when using math functions.

Date: January 20, 2022.

Look at the code, compile it and run it and try to understand what is happening. Add some printf statements inside the `the_thread_func()` so that you can see that it is really running.

For documentation of all Pthreads functions, see for example this page:
http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

You can also check the documentation of each Pthreads function using the "man" command, e.g. `man pthread_create` (provided that the necessary "man" info has been installed on the computer you are using; if not, just use the web page above instead).

Note that we are now looking at a threaded program, where several threads execute at the same time. The main thread, executing the main() function, will continue its execution after the `pthread_create()` call. At the same time, the newly created other thread will be executing the `the_thread_func()` function. By adding some printf statements in both these places we can see that both are happening.

To verify that both the main() function and `the_thread_func()` are actually being executed simultaneously, add some work so that it takes some time for them, and have printf statements before and after they do their work. For example, you can add something like this in main() after the `pthread_create()` call:

```
printf("main() starting doing some work.\n");
long int i;
double sum;
for(i = 0; i < 1000000000; i++)
  sum += 1e-7;
printf("Result of work in main(): sum = %f\n", sum);
```

and something similar inside `the_thread_func()`. Now, you should be able to see that both threads start their work at roughly the same time, then one of them will finish before the other one depending on how much work you give each of them.

Try giving more work to the main thread, then you should see that `the_thread_func()` finishes first. Then try changing the amount of work so that the work in main() completes first. In that case, the main thread will continue to the `pthread_join()` call, which will not return until `the_thread_func()` finishes.

Use the "top" command to check the CPU usage of your program when it is running. If you have two threads both working at the same time, you should see "top" showing 200 % in the CPU column. Note that this requires that the computer you are running on actually has 2 cores that are available so that the operating system can let both threads run simultaneously. (If there is only one core available, you will see 100 % even if your program uses two threads.)

When you feel that you understand how this works, try adding one more thread, running another thread function. To do that, the following is needed: create a new function like `the_thread_func()` but with another name, perhaps `the_thread_func_B()`. Then, in the main() function, add another `pthread_t` variable, perhaps `threadB`, and another call to `pthread_create()` specifying `threadB` and `the_thread_func_B()`. In the end of main() you also need another call to `pthread_join()`, waiting for the other thread to finish.

Now you have a program that has three threads executing simultaneously: the main thread plus your two additional threads.

Add some work and some printf statements in the new thread function also. Then run the program and convince yourself that all three threads are really executing simultaneously.

Task 2

The code for this task is in the `Task-2` directory.

Here, the code you are given is similar to the initial code in the previous task, but now the code in main() is also supplying a pointer to some input data for the thread that is created.

Look at "`man pthread_create`" to see which arguments the `pthread_create` function expects. In the previous task we just used the first and third arguments, while the second and fourth arguments were NULL. Now we will make use of the fourth argument, called `arg` in the documentation. This is an argument of type `void*` that lets us specify a pointer that will be given as input to the thread function for the new thread we are creating. The pointer can be used in any way the programmer decides; to specify some input parameters for the thread's work, to give a location where the thread should put its computed results, or both.

In the example code in this task, we give the thread some input data in the form of a small array of doubles. Look at the code in main(): it sets up an array called `data_for_thread` and puts some values there, then the `data_for_thread` pointer is used as the fourth argument to `pthread_create`.

The pointer given as the fourth argument to `pthread_create` will then be available to the thread via the input argument to the thread function. Note that the thread function, in our case the function called `the_thread_func`, has an input argument of type `void*`. The programmer decides what data, if any, should be given as input to the thread and gives that pointer to the `pthread_create` function. Then, when the thread is running, it can access that data using the `void*` argument to the thread function.

The reason why the `void*` type is used for this is to make it general; we can use any pointer type we want, the Pthreads library will not care about that but just supply the pointer to our thread function, then it is up to the programmer to cast that to the pointer type that is to be used in practice.

Modify the code in `the_thread_func` so that it does something with the input data: cast the input argument `arg` to `(double*)` and use printf to print the three values there. Does it work? Are you able to get access to the data from main() in this way?

When you understand how that works, try creating another thread that is running the same thread function, but given different input data. To do this, start by creating a second array of doubles, e.g. `data_for_thread_B`, and set some other values

there. Then create another thread, similarly to how you did it in the previous task but this time using the same thread function, the difference between the threads is now only that they get different input pointers.

This is a very common way of using pthreads in practice: you have several threads that run the same thread function, and you use the input argument to tell each thread what it should be doing, for example letting the threads work on different parts of an array of input data.

Task 3

The code for this task is in the `Task-3` directory.

Here we will look at how the runtime of a program can be affected by our decisions about how to divide work between threads.

The code for this task performs some very simple computation, just adding the number 7 many times in a loop, and in the end reports the final sum. The program does part of the work in the main() function, and part of the work in `the_thread_func` that is run by another thread. The amount of work done in those two places is determined by the constants `N1` and `N2`. We can change how the work is divided by changing the values of `N1` and `N2`, and the final result should be the same as long as the sum of `N1` and `N2` is kept the same.

Look at the code and try to understand how it works. Then insert timing measurements in the code (use the `get_wall_seconds()` function from previous labs) and run it. Change how the work is divided by changing `N1` and `N2`, keeping the sum of `N1` and `N2` fixed. Is the time reduced when you let both threads do approximately the same amount of work? (Don't compile with -O3 as the compiler then figures out what the loop is actually doing and removes it.)

Note that this again requires that there are at least 2 cores available for your program to use. If there is only one core available, then probably the division of work will not matter much since only one of the threads will be able to run at a time. The operating system will then let both threads run, but they will have to take turns; when one thread is running, the other one will be sleeping. On the other hand, if there are at least 2 cores available, then we expect an improvement in the real time by a factor of 2 when the work is split evenly compared to the case when one thread does all the work.

Task 4

First write a serial code that uses the most straightforward algorithm for determining how many prime numbers there are in a given range of numbers, from 1 to M where M is an input argument to your program. Simply use a loop where you naively test how many numbers each number is divisible by. (Or take any of the simpler algorithms from the first lecture.)

Then parallelize your program using Pthreads: divide the work between two threads.

Task 5

Write a program which uses N threads, where N is an input parameter to your program. For example, you can create N threads by calling `pthread_create` in a `for` loop. Organize your threads by letting each of them have a unique index, like thread 0, thread 1, thread 2, etc. When the main() function creates a thread, provide the thread index as an input parameter to the thread function (the thread function takes a pointer as input, you can for example use a struct with info for each thread and provide a pointer to that struct). Let each thread call a function which prints the index of the current thread.

Task 6

Modify your program from the Task 4 so that it uses a variable number of threads. Let the program accept two input arguments: the first argument is M as in Task 4, and the second argument is the number of threads to use, similar to N in Task 5.

When your program is working, try running it for different numbers of threads and measure the timings. Ideally, you should see a speedup up to the number of cores available on the computer. For example, if there are 4 cores available, the runtime should be reduced to 25 % when using 4 threads. Can you see such an improvement, if not can you explain why?

Also try running your program with larger numbers of threads, more than the number of cores available. Do you see any further speedup in that case?

Task 7

When using Pthreads, it is possible for each thread to create new threads. So not only the main thread can create threads; you can have a code where the main thread creates some threads, and each of them in turn creates several other threads.

To check how this works, write a program that lets the main thread create two threads and waits for them to finish. Each of those two threads also create two new threads and wait for those to finish.

This means that your program will be using seven threads in total: the main thread, plus the two threads it creates, plus 2*2 additional threads created by those. Add printf statements inside each thread function to see what is happening.

So, Pthreads allows us to create threads that create threads that create threads, and so on. This can be useful sometimes, e.g. when parallelizing some hierarchical,

recursive algorithm. However, the most common usage in practice is to have the main thread create all the necessary threads.

Task 8

The code for this task is in the `Task-8` directory.

In this task we will illustrate the use of mutex variables in a multi-threaded program. Here we have a global variable `sum` which is initialized to 0. The program creates N threads, where N is an input parameter to the program. Note that all threads have access to the variable `sum`. Threads are executing the function `the_thread_func` which is doing some job and adding the result of this job to the global variable `sum`.

Try to run the program with the same number of threads a few times. Which result do you get? Do you get the same result each time?

The variable `sum` is a shared variable; it can be accessed by all the threads. Since all threads update this shared piece of data, we need a mutex for mutual exclusion. Check the following functions:
`pthread_mutex_init`
`pthread_mutex_destroy`
`pthread_mutex_lock`
`pthread_mutex_unlock`
and modify your code using a mutex so that each time you run your program, you will get the correct value of `sum`.