HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2018
LAB 10: PTHREADS (PART II)

In this lab we work more with shared-memory parallelization using POSIX threads, also called Pthreads.

The following aspects are included in the lab:

(1) Using the return value from a thread function when calling `pthread_join()`.

(2) Using a condition variable to allow a thread to wait until another thread has done something.

(3) Using the second argument to `pthread_create()` to specify attributes for the created thread.

(4) Creating threads as as "joinable" or "detached".

(5) Parallelizing a few different kinds of computations using Pthreads.

1. Main Tasks

Task 1

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

Here we will check how the return value from a thread function can be retrieved by using the second argument to the `pthread_join()` function.

Modify the code in `threadtest.c` so that it dynamically allocates some memory (call `malloc()`) and puts some values in that memory block. Then return the pointer to that memory block – instead of "`return NULL`" you do e.g. "`return p`" where p is a pointer to the memory block you allocated.

So, now the thread has produced some data and returned a pointer to that data. How can the main thread get access to that data after the thread has completed? This can be done using the second argument to `pthread_join()`.

In main(), before the call to `pthread_join()`, declare a pointer variable of the same type that you used in the thread function. Then give the address of that pointer as the second argument to `pthread_join()`.

Date: February 15, 2018.

Then you should get a pointer to the data that was set up in the thread function. Test that it really is that data by printing out some of the values that you put there in the thread function – can you now access those same values in the main() function, after the `pthread_join()` call? Do you understand what is happening?

An alternative to returning the pointer from the thread function is to instead call the `pthread_exit()` function. Try this, and check that you can access the data in main() also in this case. It should work in the same way: according to the Pthreads documentation, returning a pointer from the thread function has the same effect as giving that pointer as the argument to `pthread_exit()`.

Note that since memory was allocated dynamically by calling `malloc()` in the thread function, we should free that memory by calling `free()` in the main() function when we are done using the memory block. You can use valgrind to check that this works, so that when you have added the `free()` call there are no memory leaks.

In the previous lab we saw how you can pass a pointer from the main thread to the created thread using the fourth argument to `pthread_create()`. Now you have seen that a pointer can also be passed in the other direction, from the created thread back to the main thread. It is up to the programmer how these possibilities are used, you may choose to use both ways, or if you prefer that you may use only the pointer passed by `pthread_create()` – you can decide that part of the data pointed to is going to be used for storing results of the thread's work.

Task 2

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

Here we will see how a *condition variable* can be used to allow a thread to wait until another thread has done something.

The code in `wait_test.c` that you are given works as follows: the main thread creates another thread, and the new thread enters a loop where it is checking if a global variable called `DoItNow` has been set to 1. The loop continues until it detects that `DoItNow` is 1. Then the thread function just does a printf statement and returns. Access to the `DoItNow` variable is protected by a mutex.

The main() function performs some work after creating the thread. Then it sets the `DoItNow` variable to 1 and then calls `pthread_join()` to wait for the other thread to finish. Since `DoItNow` has been set, the other thread finishes directly and the program ends.

Look at the code and play with it to make sure you understand what it is doing. Try removing the `DoItNow=1` line in main() and see what happens then. Do you understand why this happens? Then put the `DoItNow=1` line back so that the code works again.

This code now achieves that the created thread "waits" until `DoItNow` is set, but it is doing this by spinning around in a loop, so it is wasting CPU time. Look at

`top` and/or run the program with the `time` command to see how much CPU time it is using. You should see it using 200%, two cores are busy, one of them doing the work in main() and the other one running the loop in `thread_func()`.

Now we want to change this so that we get the same result, that the created thread waits until `DoItNow` has been set, but we want to achieve that without wasting CPU time while the thread is waiting. This can be done using a *condition variable*.

Declare a variable of the type `pthread_cond_t` at global scope, e.g. next to the mutex variable.

Then add calls to `pthread_cond_init` and `pthread_cond_destroy` for that variable in the beginning and end of main(), respectively. So far this looks similar to adding another mutex, the difference is only that the datatype and function names say "cond" instead of "mutex".

Now you can add a call to `pthread_cond_wait()` in side the loop in the `thread_func()` function. The `pthread_cond_wait()` function takes two arguments, a condition variable and a mutex, so if your condition variable is named c the call will look like this:

```
pthread_cond_wait(&c, &m);
```

The mutex is supposed to be locked when `pthread_cond_wait()` is called, so it makes sense to add it between the mutex lock and unlock calls in `thread_func()`. There, if it turns out that `DoItNow` is zero, then you can call `pthread_cond_wait()`.

When `pthread_cond_wait()` is called, the effect is that the mutex is released while the thread waits, but then when the thread awakes and the `pthread_cond_wait()` call finishes, the thread owns the mutex again, so that it can check the state of some variable(s) that are protected by the mutex, like the `DoItNow` variable in our case.

The `pthread_cond_wait()` call means that the thread that called `pthread_cond_wait()` will sleep, not using any CPU time, until some other thread wakes it up by calling `pthread_cond_signal()`. So, in our case, to wake the thread up we should call `pthread_cond_signal()` in main() after we have set `DoItNow` to 1. Do this, and then check if the program now works as it should.

Again, look at `top` and/or run the program with the `time` command to see how much CPU time it is using. If `pthread_cond_wait()` is used properly, you should now see that the program uses only 100% CPU. Does it work?

When it works, try doing some changes to check that it behaves as you expect. For example, try removing the `pthread_cond_signal()` call – what happens then? Do you understand why?

Task 3

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

In Pthreads the threads can be defined as joinable or detached. What is the difference of joinable and detached? Run the program `join.c` (first build it using make) and study the program flow. What happens if we set the attribute to detached (change `PTHREAD_CREATE_JOINABLE` to `PTHREAD_CREATE_DETACHED`) and remove the `pthread_join` calls? What happens if we also remove the `pthread_exit` call in the end of main()?

Task 4

The code for this task is in the `Task-4` directory.

Here we will take another look at how condition variables can be used.

As noted above, a condition variable allows a thread to block (sleep) until a specified condition is reached. In the program `synch.c` a global barrier is implemented using a condition variable, study and run the program. What happens if we remove the barrier, how will the output change?

Note that the idea of a "barrier" in a threaded program means that all threads should synchronize by waiting for the others at that point in the program; no threads should continue after the barrier until all other threads have reached the barrier.

Another way to implement a barrier would be to constantly read a global variable until it changes. This is implemented in `spinwait.c`. First compile `spinwait.c` without any compiler optimization, run it and check that it works. Then try compiling `spinwait.c` using the `-O3` optimization option, and then run it. Why does this not work? How can we fix this? (*Hint:* What is the meaning of the keyword `volatile` in C?)

To see explicitly what the compiler is doing differently when the `volatile` keyword is used, it is interesting to compare the assembly code output of the part of the program that corresponds to the while-loop waiting for the (mystate==state) condition to become true. (This is easiest to do with only a moderate level of compiler optimization; use the lowest level of compiler optimization that gives the problem. If `-O1` is enough to give the problem, then use `-O1`.) What does the loop look like when compiler optimization is turned on and `volatile` is not used? Looking at the assembly code, can you understand why the program hangs in that case? How does the assembly code change when `volatile` is used? Can you understand why the program no longer hangs then?

Task 5

The code for this task is in the `Task-5` directory.

The program in `pi.c` computes π in parallel using numerical integration. A numerical way to compute π is the following:

$$\int_0^1 \frac{4}{1+x^2} dx = [4\, arctan(x)]_0^1 = \pi$$

If we use the midpoint rule, we can compute the above integral as follows (see

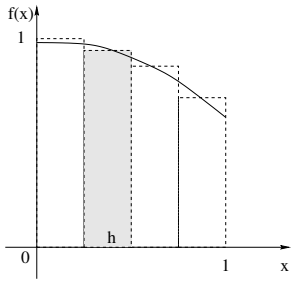

FIGURE 1.1. Numerical integration with the midpoint rule.

Figure 1.1):

$$\int_0^1 \frac{4}{1+x^2} dx = h \sum_{i=1}^{n} \frac{4}{1+x_i^2} = h \sum_{i=1}^{n} \frac{4}{1+((i-1/2)h)^2}.$$

The trivial parallel implementation of the latter formula is that if we have p threads available, we slice the sum into p pieces, attach one interval to each of them to compute a partial sum, and then each thread adds its partial sum to a global sum. Alternatively, instead of having a global (mutex-protected) sum variable we may let each thread store its result in a separate location and then let the main thread sum up the results after the other threads have finished their work. Parallelize the computations in the program `pi.c` with appropriate Pthreads functions.

Task 6

The Enumeration-sort algorithm. The code for this task is in the `Task-6` directory.

For each element (j) count the number of elements (i) that are smaller ($a(i) < a(j)$). This gives the rank of a(j), i.e. anew(rank(j))=a(j).

Finding the rank of an element is an independent and perfectly parallel task, i.e., finding ranks of two different elements can be computed concurrently. This strategy is implemented in the program `enumsort.c`. Study, compile (make) and run the program.

The overhead in creating a thread for finding the rank of one element is too high and the efficiency becomes poor. A better approach is to create tasks where the

rank of several elements are found by one thread within the task. Implement this strategy and compare with the first approach.

What is the computational complexity of the enumeration-sort algorithm? Is it a good algorithm, compared to e.g. the merge sort algorithm that you have worked with in some of the previous labs?

Task 7

Matrix-matrix multiplication. The code for this task is in the `Task-7` directory.

Consider the program `matmul.c` (build it using "make"). Here, two matrices A and B are multiplied and the result is the matrix C. Find the parallelism in the computations and divide the work between threads. What is the maximal speedup you get from your parallelization on 1000x1000 matrices? For how small matrices is it worth doing the parallelization? (Remark: the code is far from optimal due to cache performance, but for our purposes here, in this lab exercise, we focus on parallelization.)