Lab 4: Threads and caches

Introduction to Studies in Embedded Systems (1DT086)

—

Advanced Computer Science Studies in Sweden (1DT032)

—

Autumn 2019, Uppsala University

1 Introduction

1.1 Goals

In this lab you will practice some basic thread management using pthreads. You will observe some effects of caches and speed-up due to parallelization. You will also use the Sense HAT joystick and see the use of threads for handling blocking I/O operations. In an optional task you can observe (and fix) some simple data races.

1.2 Preparations

Read through these instructions **carefully** in order to understand what is expected of you. You should have finished *Lab 2* and have a working Linux system on your Raspberry Pi before doing this lab.

1.3 Report

For this lab you have to hand in a written report. Each group hands in one joint report, where *all* group members have participated in *all* parts of the report, and can individually explain every part if asked. These instructions will make it clear what should go into the report.

The report *must* be handed in in PDF format, and be written in LaTeX. Use the template `report-template.tex` that is found on the Student Portal. If you much prefer, you may use some other LaTeX template instead.

2 Warming up

This section does not contain any tasks that need to be considered for the report, but it is highly recommended that you go through it anyway.

2.1 pthreads

For a very short (and very incomplete) reference to the pthreads API, see the slides from the last lecture. There are many guides that are more complete available online, for example this one. To get started with pthreads, try the simple C program in Listing 1 (it is also available on the Student Portal in the file `small_pthreads_example.c`).

Listing 1: A simple example program using pthreads.

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  void *function_to_run(void *arg) {
5      int n = *(int *) arg;
6      printf("I'm the new thread! I got %d passed to me.\n", n);
7      return NULL;
8  }
9
10 int main() {
11     pthread_t mythread;
12     int n = 5;
13
14     printf("Creating a new thread...\n");
15     if (pthread_create(&mythread, NULL, function_to_run, &n)) {
16         fprintf(stderr, "Error creating new thread.\n");
17         return -1;
18     }
19
20     printf("Waiting for the new thread to terminate...\n");
21     pthread_join(mythread, NULL);
22     printf("New thread has terminated.\n");
23
24     return 0;
25 }
```

This program will have two threads:

1. The default thread that was created with the process and that executes the `main` function.
2. A second thread created on line 15 that executes `function_to_run`.

Note how we pass an argument to the new thread as the last argument `&n` to `pthread_create`. This argument must always be a pointer without type, i.e., a void pointer `(void *)`. We actually pass an `int` pointer `(int *)`, but this is ok as its type will automatically be cast[1] to a void pointer as needed.

[1] In C, to *cast* or *type cast* is to change the type of some value. It is sometimes done automatically, as on line 15, and sometimes explicitly, as on line 5.

In `function_to_run` the parameter `arg` is a void pointer, but since we know it is in fact pointing to an integer, we just cast it back to an `int` pointer with `(int *) arg`. On the same line we dereference the newly re-cast pointer with the dereferencing operator `*`, and store the integer value in a new local variable called `n` as follows: `int n = *(int *) arg`.

To compile a program using pthreads properly with GCC, you need to pass it the `-pthread` flag.

```
gcc -Wall -pthread myprogram.c -o myprogram
```

Compile and run the program in Listing 1 and see that it works as expected.

2.2 The Sense HAT joystick

In Task 2 you will use the small joystick on the Sense HAT board. The joystick device is mapped by your Raspberry Pi Linux system to the special file `/dev/input/event0`.[2] By reading from this file it is possible to get the input events from the device.

Before using the joystick, you should stop the script that prints information on the LED matrix when the joystick is pressed. Use `systemctl` for this. If you only `stop` the script (and don't `disable` it), the script will automatically run again when you restart the Raspberry Pi, which is probably what you want.

```
sudo systemctl stop sensehat-info
```

The input events that can be read from `/dev/input/event0` have a special format that encodes the type of event (*press*, *release* or *hold*), the direction of the event (*north*, *south*, *west*, *east* or *down*) as well as a time stamp.

To help you parse the events and enable you to access the device from your C programs, there is a (very) small library available on the Student Portal. It consists of the files `joystick.c` and `joystick.h`. There is also a small example program available that shows how to use these.

Download the `joystick_example.tar.gz` archive from the Student Portal and copy it to your Raspberry Pi with `scp`. On your Raspberry Pi, unpack it using `tar`.[3]

[2] If, for whatever reason, the joystick gets mapped to another device file on your system, update the file `joystick.h` with the other filename.

[3] Don't use `sudo` with `tar` if you want the extracted files to be owned by your user.

```
tar -xvf joystick_example.tar.gz
```

The archive contains the joystick library files, `joystick.c` and `joystick.h`, as well as the example program `joystick_example.c` and a `Makefile`. Study these files to understand what they do (at least the files `joystick_example.c` and `joystick.h`, the details of `joystick.c` are less important).

The `joystick.h` header only defines three functions and some constants. The functions `open_joystick_device` and `close_joystick_device` opens and closes the device file, respectively. The `read_joystick_input` function gets events from the device. This function will read a single event and blocks the calling thread until there is some input to return. The return value is of the type `struct js_event` that is defined in `joystick.h`.

Compile the example program with `make` and try it out to see that it works as expected. Stop the process by pressing Ctrl+c.

```
make
./joystick_example
```

2.3 The `time` command

A convenient way to see how long it takes for a process to terminate is to use the `time` command. You give it the program to execute as an argument, and when that process has finished, `time` will print some timings. You can try it with any program, for example `ls`.

```
time ls
```

After the output of `ls` is the timings from the `time` command, looking something as follows.

```
real    0m0.008s
user    0m0.001s
sys     0m0.007s
```

The `real` field tells you how long it took for the process to terminate measured in *wall clock time*. That is, how much actual time passed between start and finish.

The `user` field tells you how long the process's threads have spent executing the program's code in total, measured in *CPU time*. That is, the amount of

time that some CPU core has been busy executing some thread belonging to the process. If the process has several threads, their CPU time is summed for the `user` field. If the process has several threads and the CPU has several cores, it is possible for the `user` field to be greater than the `real` field. Conversely, if the process is mostly idling or has to wait for threads from other processes to be scheduled by the OS, the `user` field can be much smaller than the `real` field.

The `sys` field is like the `user` field, except that it instead counts the CPU time that threads from the process are consuming in *kernel mode* by invoking system calls (opening files etc.). The `user` and `sys` fields together give you the total CPU time of the process.

You can try the `time` command on the example pthreads program above, and on the joystick example program. Convince yourselves that the reported timings make sense.

```
time ./small_pthreads_example
time ./joystick_example
```

3 *Task 1:* Using multiple threads for speed-up

In this task you will observe how using multiple threads can speed up a program, in this case a Monte Carlo algorithm for estimating the value of π.

A Monte Carlo algorithm uses random sampling to estimate an answer. Here we will estimate the value of π by generating uniformly random points inside a square and determining what fraction of those points that end up inside a maximal inscribed circle (see Figure 1). The relation between the area A_s of the square and the area A_c of the circle is

$$\frac{\pi}{4} = \frac{A_c}{A_s}.$$

In other words, if we know the value of the fraction A_c/A_s, then we can calculate the value of π as

$$\pi = 4 \times \frac{A_c}{A_s}.$$

We can estimate the value of the fraction A_c/A_s by generating uniformly random points inside the square and counting the fraction of those points that are also inside the circle.

$$\frac{A_c}{A_s} \approx \frac{\#\text{points in circle}}{\#\text{total points}}.$$

Therefore we can also estimate π as

$$\pi \approx 4 \times \frac{\#\text{points in circle}}{\#\text{total points}}.$$

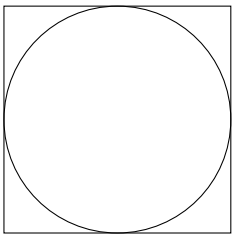

Figure 1: A maximal circle inscribed in a square. The area A_c of the circle is $\pi/4$ times the area A_s of the square.

Question 1.1 —— *Does the above Monte Carlo algorithm for estimating the value of π seem like a good candidate for speed-up with parallelization using several threads? Clearly explain and motivate your answer.*

Download the file `monte-carlo-pi.c` from the Student Portal and copy it to your Raspberry Pi. Open the file and study it until you understand what it does. Compile it and run it with *one thread* for the Monte Carlo algorithm (i.e., let `NUM_THREADS` be defined as `1`).

Take timings of the process with the `time` command. Also observe the behavior of `htop` in another SSH session while you run the program, especially the load on the CPU cores displayed at the top.

Question 1.2 —— *How long does it take for the process to finish in wall clock time with one thread? How much CPU time has the process consumed in total? What could you observe about the load on the CPU cores from* `htop`*?*

Repeat the above for 2, 4, and 8 threads running the Monte Carlo algorithm (modify `NUM_THREADS` and recompile the program in between).

Question 1.3 —— *How long does it take for the process to finish in wall clock time with 2, 4, or 8 threads? How much CPU time has the process consumed in total in each case? What could you observe about the load on the CPU cores from* `htop`*?*

Question 1.4 —— *Explain all the above observations as clearly as possible, taking into account the number of processor cores on the Raspberry Pi.*

(*Note:* The above Monte Carlo algorithm was chosen as an illustrative example for how to speed up computations using multiple threads, it is not actually an efficient way to calculate digits of π.)

4 *Task 2:* Multiple threads for handling I/O

Multiple threads in a program is not only used to achieve a speed-up on parallel computer architectures (like multicore processors), but also to easily and efficiently handle I/O operations that are slow or blocking.

It is common, for example, in frameworks for graphical user interfaces to have one or more threads that are waiting for input events from the user while other threads are doing other work. In this task you will do something similar by creating a thread that waits for input events from the Sense HAT joystick while another thread is continuously doing other work.

Download the archive `io_handling.tar.gz` from the Student Portal. Copy it to your Raspberry Pi and unpack it. It contains a program `io_handling.c` as well as the library files for handling the LED matrix and joystick that you have seen before—`led_matrix.c`, `led_matrix.h`, `joystick.c`, and `joystick.h`—and also a `Makefile`.

The program `io_handling.c` is not multi-threaded. It just contains code for moving some lighted LEDs (the "snek") over the LED matrix over and over. Open the file and study it until you understand how it works. Compile it (with `make`) and run it to see that it behaves as expected. Stop the program by pressing Ctrl+c.

Your task is now to edit this program to create another thread that handles input from the joystick while the main thread moves the snek. In order to do so, first create a new function `void *handle_input(void *arg)` that does the following in an infinite loop.

1. Wait for an input event from the joystick by calling the blocking function `read_joystick_input`.
2. If the event is *not* of type `JOYSTICK_PRESS`, ignore it and go back to 1.
3. Otherwise, do the following depending on the direction of the event, and then go back to 1.
   - `DIRECTION_WEST`: Change the color of the snek to `RGB565_MAGENTA`.
   - `DIRECTION_EAST`: Change the color of the snek to `RGB565_YELLOW`.
   - `DIRECTION_NORTH`: Increase the length of the snek by one (to a maximum of `NUM_LEDS`).
   - `DIRECTION_SOUTH`: Decrease the length of the snek by one (to a minimum of 1).
   - `DIRECTION_DOWN`: Terminate the program after closing both the LED matrix and joystick device files properly. For this it might be convenient to create another global variable with which the main thread can be notified.

It may be helpful to take a look at `joystick_example.c` from Section 2 for hints about how to read input events from the joystick.

Then edit the program to create a new thread that runs your newly created function `handle_input` in parallel to the main thread. The main thread should create this new thread before it starts moving the snek.

Verify that the program works as expected: That the snek keeps moving over the LED matrix continuously, and that input events to the joystick are handled without delay.

Listing 2.1 —— *Your* `.c` *file for the above program.*

5 *Task 3:* Cache behavior

In this task you will observe some cache behavior and the difference in speed of different levels of the memory hierarchy. The Raspberry Pi 3 model B has a Broadcom BCM2837 CPU, which is a quad-core ARM Cortex A53. It has private L1 caches per core, one for data and one for instructions. It also has an L2 cache that is shared between the cores and uses a (pseudo-)random cache replacement policy.

Download the file `cache.c` from the Student Portal and copy it to your Raspberry Pi. This program creates arrays of different sizes and times how long it takes to access 100 000 000 elements of each array. It does this by traversing the array as many times as is needed (smaller arrays will therefore get fully traversed more times than larger arrays).

In order to mostly defeat the caches' ability to exploit spatial locality (and cache prefetching[4]), the accesses into the arrays follow a random order. This is achieved by letting the values of the elements correspond to indices in the array, and randomly shuffling them so that starting at any index and selecting the next index from the value at the current index, we create a big cycle covering all elements. See Figure 2 for an example of such an array with just 10 elements.

Open the file `cache.c` and study it until you understand how the program works. Compile it and run the program.

Question 3.1 —— *What timings did you get from the program? Try to explain the reported timings with respect to the memory hierarchy and the program's behavior.*

[4]Modern processors try to analyze the pattern of the program's memory accesses and predict which data will be used soon. It can then bring those cache lines into the cache even before they have been requested. This is called cache prefetching. It often works well, but not on "random" access patterns such as in this array.

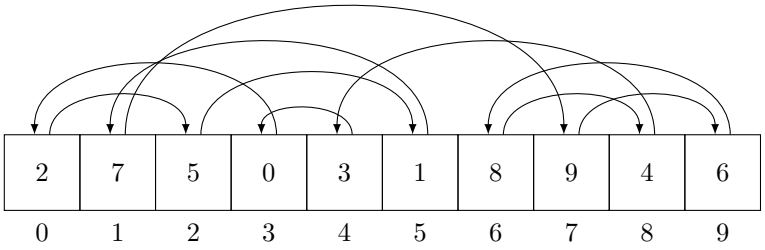

Figure 2: A randomly shuffled array where following values to the next index creates a big cycle.

Question 3.2 —— *Based on the timings and your knowledge of the Raspberry Pi's memory hierarchy structure, guess the sizes of the L1 data and L2 caches. Motivate your guesses carefully.*

Now you should edit the above program to instead walk through the arrays sequentially and see the benefits of spatial locality (as well as cache prefetching). Add another function `int *make_ordered_array(size_t size)` that works like `make_random_array` except that it generates arrays where the value at every index is the next index, with wrap-around to zero. The generated arrays should look as in Figure 3, but with varying sizes.

In the program's `main` function, replace the call to `make_random_array` with a call to your new `make_ordered_array`. Compile and run the program again.

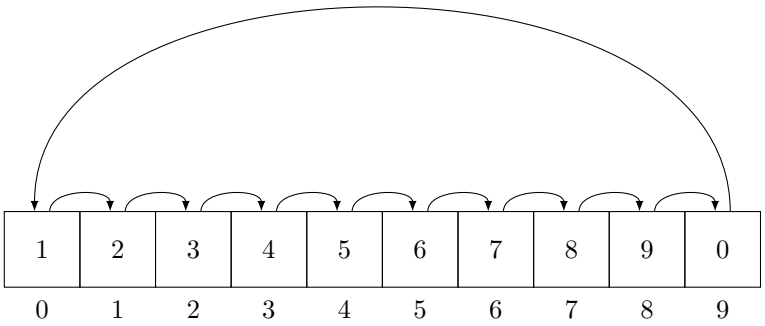

Figure 3: An ordered array where following values to the next index creates a sequential walk through the array.

Listing 3.3 —— *Your* `.c` *file for the above program.*

Question 3.4 —— *What timings did you get from the edited program? Explain why these differ from the original program.*

6 *Stretch goal:* Race conditions

(This task is completely optional. You don't need to include it in your report.)

In this task you will observe—and fix—a race condition in a multi-threaded program. You will also observe the penalty in execution time that may have to be paid for such fixes.

Race conditions in software occur when variations in the timing and scheduling of threads cause different end results. These are often difficult to reason about, and often lead to bugs that are hard to test for.

Download the file `race.c` from the Student Portal and copy it to your Raspberry Pi. This program creates two threads that both increment the same global counter variable in a loop. Open the file and study it until you understand how the program works.

Compile and run the program for different number of increments per thread (modify the constant `COUNT_PER_THREAD`). Do this for at least the values 10, 100, 1000, 10 000, and 100 000 000.

To ponder ——— *What output do you get from the program for different per-thread increment counts? Try to understand why this is not always the output that one could expect, and what causes this issue. Try to understand why this seems to be less likely to occur for some per-thread increment counts than others.*

To fix the above problem, we can use `mutexes` to sequentialize parts of the threads' execution. Write an edited version of the program that uses a mutex object to protect the updates to the counter variable from being performed by more than one thread at a time.

First you have to declare a mutex variable and initialize it. You can make it global for easy access from both threads.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Then protect the entire for-loop in the function `loop_increment` by locking the mutex before the loop and unlocking it after. You lock the mutex with a call to `pthread_mutex_lock(&lock);`. This call will lock the mutex directly if it was unlocked, otherwise it will block until some other thread unlocks it before

continuing.

Unlocking the mutex is done with a call to `pthread_mutex_unlock(&lock);`. This call never blocks.

The mutex object is kept inside the OS kernel, and you should properly dispose of it by calling `pthread_mutex_destroy(&lock);` before terminating the process.

Compile and run the program again for different values of `COUNT_PER_THREAD` and make sure that the final count is now always twice this value.

Unfortunately, while mutexes and other forms of synchronization between threads can remove bugs due to race conditions, they also often negatively affect performance by reducing parallelization and by the (sometimes considerable) overhead of the operations themselves.

Make a third version of the above program where the twin calls to the functions `pthread_mutex_lock` and `pthread_mutex_unlock` are instead *inside* the loop, protecting only the actual increment statement `counter++;`. Compile and run this program and make sure that it also avoids the race condition.

Now, for each of the three versions of the program (no mutex, locking/unlocking outside the loop, and locking/unlocking inside the loop), run it with the `time` command for a value of 100 000 000 for `COUNT_PER_THREAD`.

> **To ponder** ——— *What timings does the* `time` *command report for the three versions of the program? Try to understand why the results differ the way they do.*