CREATING AND INSPECTING THREADS

COURSE: REAL-TIME OPERATING SYSTEMS

Date: [06/03/2020]

Student 1: **Jordi Jaspers** Group: **EA – ICT**

Lecturer: **Jo Vliegen**

Table of Contents

[1 Introduction 2](#_Toc34387716)

[2 Program Description 2](#_Toc34387717)

[2.1 The MakeFile 2](#_Toc34387718)

[2.2 Defines & Includes 2](#_Toc34387719)

[2.3 Producer and consumer Principle 3](#_Toc34387720)

[2.4 Thread Creation 4](#_Toc34387721)

[2.5 Killing the threads + extra code snippets 5](#_Toc34387722)

[3 Results 6](#_Toc34387723)

# Introduction

In [computer science](https://en.wikipedia.org/wiki/Computer_science), a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a [scheduler](https://en.wikipedia.org/wiki/Scheduling_(computing)), which is typically a part of the [operating system](https://en.wikipedia.org/wiki/Operating_system). The implementation of threads and [processes](https://en.wikipedia.org/wiki/Process_(computing)) differs between operating systems, but in most cases a thread is a component of a process. [Multiple threads](https://en.wikipedia.org/wiki/Thread_(computing)#Multithreading) can exist within one process, executing [concurrently](https://en.wikipedia.org/wiki/Concurrent_computation) and sharing resources such as [memory](https://en.wikipedia.org/wiki/Shared_memory_(interprocess_communication)), while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its [dynamically allocated](https://en.wikipedia.org/wiki/Memory_management#HEAP) variables and non-[thread-local](https://en.wikipedia.org/wiki/Thread-local_storage) [global variables](https://en.wikipedia.org/wiki/Global_variable) at any given time.

# Program Description

This particular chapter will explain how the program is built. It gives a thorough description on each part of the code. The code will be divided in multiple sections to get a better understanding of the overall principle. Keep in mind that the advantaged of using multithreaded workload instead of using single threads is still the main topic of this report. The full code is accessible in the attached file.

## The MakeFile

If you want to run or update a task when certain files are updated, the make utility can come in handy. The make utility requires a file, Makefile (or makefile), which defines set of tasks to be executed. You may have used make to compile a program from source code. Most open source projects use make to compile a final executable binary, which can then be installed using make install. In our case, we would like to automate the “gcc” command so, the c-program gets compiled to an object file and gets linked to its respective binary file in one command.

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **Figure 2.1: The MakeFile** |   A black and silver text  Description automatically generated | The stated commands will execute the following:   * **Make** compiles and links the “.c” file * **Make CLEAN** Removes the “.o” and OUTPUT file * **./OUTPUT** Executes the binary file created through the compiler |

## Defines & Includes

In the C Programming Language, the #include directive tells the preprocessor to insert the contents of another file into the source code at the point where the #include directive is found. Include directives are typically used to include the C header files for C functions that are held outsite of the current source file. These files are for the most part “.h” files. In figure 2.2 you will see we are adding the <semaphore.h> and the <pthread.h> files to make use of the multithreading capabilities of the system. The only other important include is <time.h>. This is used to compare times later in the report. All the other includes are for basic capabilities of the program

|  |  |  |
| --- | --- | --- |
| On line 16 and 17 we defined some parameters. These are just static parameters that are used in the program. They tell us the amount of threads that are created, together with the maximum number the program is going to count to.  Line 22 is a static variable that stores a value the producer made. The consumer can request that value to be able to make the right calculation on that given value. The “m-lock” variable is made to lock the shelf once a consumer got that value, so no other consumer can get the same number. | |  | | --- | | **Figure 2.2: includes and defines** | |

## Producer and consumer Principle

A producer is someone who sells the goods or provides the service directly to the consumer with no involvement with a middleman. While the consumer goes directly to the producer to buy the product without going through any other channel. This principle is also used in the computing environment. Producers and consumer will share a fixed size buffer and they share this buffer. In the next particular example code, the producer will provide a number through a simple counter in a for-loop and saves it in the shelf variable. The value gets emitted or posted and the producer will wait until a consumer takes the value out of the shelf.

The consumer function waits in the while-loop until the producer has produced a number, so he can take it out of the shelf. It then does a *“sem\_post(&obj\_consumed)”* function to let the producer know he consumed that value. After consuming the value, it calculates if the number is a prime. If it is a prime number, it gets printed to the terminal.

|  |
| --- |
| **Figure 2.3: Producer and consumer Functions** |

A screenshot of a cell phone

Description automatically generated

## Thread Creation

Like indicated in the previous section, the producer will produce a value and the consumer will consume the produced value. So, we are depending the most on the consumers for all the calculation. It will be advantageous to create as much consumer as possible to process all the produced values. Anyways, it would not be advantageous to create multiple producers because of the fact that there will not be consumers enough consumers to process all the values.

|  |
| --- |
| **Figure 2.4: Thread Creation** |

A screenshot of a cell phone

Description automatically generated

Between line 96 and 102, the producer thread will be created. Because there is no for-loop, only one will be created. In contrary to the second part around line 107 to 114, the for-loop will loop until “NUMBER\_OF\_THREADS” is reached. So, the variable decides the amount of consumer threads. Everything before line 96 is the initialization before creating the actual threads. The “pthread\_join()” function shall suspend execution of the calling thread until the target thread terminates, unless the target thread has already terminated.

## Killing the threads + extra code snippets

|  |  |
| --- | --- |
| Killing a thread is as easy as calling the “pthread\_kill()” function followed with a “sem\_destroy()” function. The next code snippet shows how the functions are used in this example. Ofcourse, we end the program by returning a 0 code to tell it was success and nothing went wrong. | A close up of text on a screen  Description automatically generated |

|  |
| --- |
| **Figure 2.5: Thread killing** |

Eventually we would like to track the time and the output to compare all the values after multiple tests with a different amount of threads. The next code snippet indicates the general principle of a timer to track the time.

|  |  |
| --- | --- |
|  |  |

|  |
| --- |
| **Figure 2.6: File creation and timer** |

By implementing the “clock()” function the program will count the amount of clockpulses. That is not a readable time for humans. So at the end of the program we call a stop clock() function and then divide the amount of clock pulses with the CLOCK\_PER\_SEC. This will give the number of seconds it took for the program to run. “CLOCK\_PER\_SEC” is variable defined in the <time.h> include. It gets the clock of the computer that is running this document.

Finally, we would like to print all the results and the terminal output in a document to get an easy comparison later on. The document is opened in the beginning of the main function and closes at the end of the program.

# Results

The following graph indicates how the relation between ‘the elapsed real time’ the program consumes in function of ‘the number of threads that are contributing’. That graph gives a good representation of the advantageous of using multiple threads and if there will be saturation when using too much threads.

It is very clear there is a limit around 15 threads because the execution time goes up after that. But around 50 threads there is saturation at 385 seconds of execution time. We cannot deny that the improvement between 2 threads and 15 threads is major. This is the reason we use multiple threads.

The reason behind this is because the CPU in the computer the program ran on has 8 core – 16 thread CPU. The optimal thread count is going to be 16, any more will only slow down the processor. Compare it with the following example, Imagine you have a corridor that you can fit four people down, side by side. You want to move all the rubbish at one end, to the other end. The most efficient number of people is 4.

If you have 1 - 3 people then you are missing out on using some corridor space. If you have 5 or more people, then at least one of those people is basically stuck queueing behind another person all the time. Adding more and more people just clogs up the corridor, it does not speed up the acivity. So you want to have as many people as you can fit in without causing any queueing.

|  |  |  |
| --- | --- | --- |
|  | Counter Size: | 500000 |
| id | **Amount Threads** | **Execution Time** |
| 1 | 2 | 398,328696 |
| 2 | 4 | 374,050885 |
| 3 | 6 | 375,440889 |
| 4 | 10 | 370,759417 |
| 5 | 15 | 370,915610 |
| 6 | 25 | 378,949575 |
| 7 | 50 | 385,644386 |
| 8 | 100 | 384,872297 |
| 9 | 150 | 385,999499 |
| 10 | 200 | 387,510740 |
| 11 | 250 | 384,612100 |
| 12 | 300 | 382,930111 |