Computer Architecture, Operating Systems, and Networks (2020)
Hand in Seven: Distribution of Primes

Peyman Afshani

March 24, 2020

1 Introduction

Prime numbers are among the most fascinating mathematical objects and there are still a lot of unanswered questions about them. One of the common intuitions about them is that they are distributed "like random numbers" unless there is a simple reason against it. Here, we would like to test this intuition. For instance, the only even prime number is 2 which means that no prime number will have 4, 6, or 8 as their last digit (in decimal) and the only prime number with 2 as its last digit is 2 itself. Similarly, there is only one prime number that ends with digit 5 (the number 5 itself). However, we cannot think of a simple reason why a prime number should not end in digits 1, 3, 7, or 9. So if our intuition is correct, then the prime numbers ending in each of these digits should be more or less evenly distributed. Here, you will write a program to test this intuition.

We have seen how using multiple threads we can greatly speed up the running time of computational tasks. However, a very big challenge is that changing a global variable or object using multiple threads can cause race conditions where the changes applied by one thread overwrite the changes done by another thread.

In this hand in, you will use non-blocking *compare and swap* instructions to avoid race conditions. Since this solution crucially relies on instructions supported by the hardware, you will need to implement parts of it in assembly language.

Remark. This hand in requires little coding but the greatest challenge in solving it lies in forming a *good* understanding of the fundamentals. So don't be disappointed if it takes you a bit of time to get started.

2 Compare and Swap

In this hand in, you can build on the previous hand in where you spawned a number of threads to find all the prime numbers smaller than `MAX`.

Try to use the following general framework for doing this. As a partial help, I have provided two incomplete pieces of code that you can download from here (you need to be logged into blackboard as before). However, if you prefer to write your own code from scratch, then you are absolutely free to do so.

2.1 A General Compare and Swap Technique

There are many ways to use compare and swap instructions. Here we describe a particular way to do it. This is not the most efficient way to do it but it offers a very general solution that can be easily applied to a lot of other problems. This solution is best when we expect very few race conditions.

Imagine we have a global object that we need to keep updated using n threads. In our case, this global object is of a struct type called `ptypes_t` defied as follow:

```
typedef struct ptypes_st {
  long n1;
  long n3;
  long n7;
  long n9;
} ptypes_t;
```

Whenever a thread discovers a new prime number p that ends in an odd digit j, it needs to increment the number of prime numbers ending in j by 1. These need to be stored in some global variable of type `ptypes_t`.

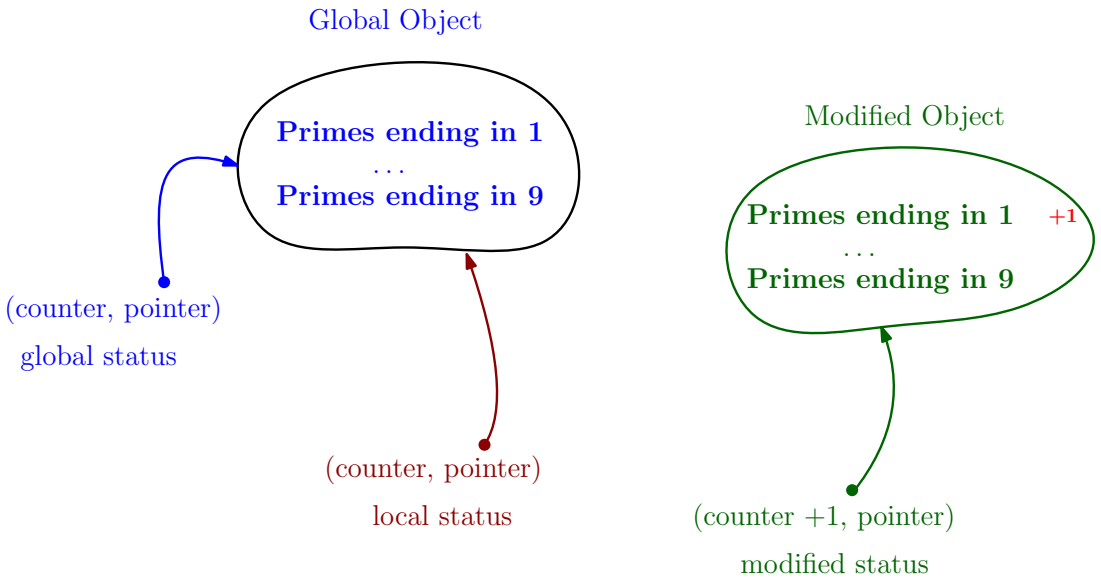

Figure 1:

Global variables. At the beginning, we define a global variable that includes a *64-bit counter* and a pointer to a `ptypes_t` type variable. Note that the pointer to the `ptypes_t` variable is also a 64 bit number. Thus, we can package the counter and the pointer in a struct that takes 128 bits. Let us call this struct `status_t`. We define a global variable of this type that all threads can access to. Let's call this variable the `global_status`. Thus, the `global_status` contains a (global) counter as well as a pointer to a (global) *object* of type `ptypes_t`. Think of the `global_status` variable as a (global counter, pointer to an object) pair. The global counter will keep track of how many times the object has been updated.

Finding a prime number. Now consider a running thread and assume the running thread finds a new prime number p.

It first needs to make a (temporary) local copy of the `global_status`. In other words, it stores a *copy* package which we can think of as (copy of global counter, copy of the pointer to global object) pair. Let's call this the *local status*. Note that this local status includes only a copy of the pointer! Thus, the object has not been duplicated!

However, the thread also needs to make an actual copy of the object. It allocates space for a new `ptypes_t` variable, let's call this the *the modified object*, and it copies and then updates the correct number of primes from the `global_status`.

Using the `lock cmpxchg16b` Instruction. Observe that as long as the local status is the same as the `global_status`, then we know no other thread has updated the `global_status` yet. This is because the

counter in the `global_status` counts the number of updates done by the threads. Thus, we create a new variable of type `status_t` where the count has been incremented by one and the pointer points to the modified object. We call this the *modified status*. We can now use the `lock cmpxchg16b` Instruction. We define a new function called `my_cmpr_swap`. In this function and using the instruction `lock cmpxchg16b`, we perform the following operations atomically: if the local status equals the `global_status`, we copy our modified status, that is, the pair (counter+1, pointer to modified object), into the `global_status` and set the Zero flag, otherwise, we do not change the memory and do not set the Zero flag. After this, `my_cmpr_swap` function returns true if the Zero flag is set otherwise, it returns false.

If `my_cmpr_swap` function returns true, then we know that we have managed to update the `global_status` variable atomically to reflect the newly discovered prime number. However, if it returns false, then we know that some other thread has modified `global_status` variable (i.e., has discovered a new prime) while we were trying to set up our usage of the `lock cmpxchg16b` Instruction. In this case, we simply need to try again (update the local status, and update the modified object and status).

For a visual description of these concepts, see Figure 1.

Issues when using the `lock cmpxchg16b` Instruction. You have to remember that Intel is Little Endian. This has big a impact on how you need to use this instruction. See the template of the assembly file for some comments on this or take a look at the "Intel Exercises".

2.2 A Little Boost of Performance

To increase the performance of your program you can do the following optimization: each thread can store the number of primes it finds in its local variables (e.g., in four local variables n_1, n_3, n_7, and n_9). The thread can then update the global counts of the primes once it is done. This way, each thread performs only one update on the global object rather than every time a prime number is found.

2.3 Not Leaking Memory

A main component of the solution outlined here is that every time a thread wants to perform compare and swap, it needs to create a new object of type `ptypes_t` and thus it allocates new memory for that object. This means, we have to take care of another major headache of programming in C: we need to explicitly free unused memory. If we do not free memory, the threads continue to allocate more and more memory even though we are not using most of the allocated memory. Thus, the thread that performs a compare and swap should free the previously allocated and now unused memory, using the `free(void *)` function.

You will need to do this to make sure that your program does not leak memory (creates dead pockets of allocated but unused memory).

3 Part Three: Testing

Finally, execute and test your program for different number of threads. Make sure that you always get the same result. In this webpage you can find a table that lists the number of prime numbers for some powers ten. You can use this to do some basic testing.

Hint. To make sure that you understand how the compare and swap instruction works, write a small test program to test `my_cmpr_swap` function. An example of such a program can be found in compressed files you downloaded.

4 Part Four: Experiments

Submit your code together with your report. In your report, first briefly explain how your solution works and how did you implement it. Second, mention your CPU brand and its number of cores. Then,

start with a value `MAX` that guarantees that executing the program with only one thread takes some amount of time (e.g., 10 seconds). Then slowly increase the number of threads. Write down how long the program takes to run as well as the final result for each execution.

Include the result of a few test cases for the compare and swap function.

Finally, to test that your program does not leak memory use the `valgrind` command. To do that, first increase `MAX` to a large number and then run your program with a very large number of threads. `valgrind` can produce a lot of information but what we are looking for is a block of text titled `LEAK SUMMARY`. On my program I get this:

```
> valgrind  ./a.out 128
==29445== Memcheck, a memory error detector
...
==29445== LEAK SUMMARY:
==29445==    definitely lost: 0 bytes in 0 blocks
==29445==    indirectly lost: 0 bytes in 0 blocks
==29445==      possibly lost: 0 bytes in 0 blocks
==29445==    still reachable: 3,088 bytes in 2 blocks
==29445==         suppressed: 0 bytes in 0 blocks
==29445== Rerun with --leak-check=full to see details of leaked memory
==29445==
==29445== For counts of detected and suppressed errors, rerun with: -v
==29445== ERROR SUMMARY: 12 errors from 2 contexts (suppressed: 0 from 0)
```

If I change my program to leak memory, I can get something like this:

```
>./a.out 128
==29757== Memcheck, a memory error detector
...
==29757== LEAK SUMMARY:
==29757==    definitely lost: 2,382,896 bytes in 148,931 blocks
==29757==    indirectly lost: 0 bytes in 0 blocks
==29757==      possibly lost: 0 bytes in 0 blocks
==29757==    still reachable: 3,104 bytes in 3 blocks
==29757==         suppressed: 0 bytes in 0 blocks
==29757== Rerun with --leak-check=full to see details of leaked memory
==29757==
==29757== For counts of detected and suppressed errors, rerun with: -v
==29757== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

As you can see, I am now leaking almost all the memory I allocated! The problem with memory leaks is that if your program leaks memory, you might eventually run out of memory or waste a lot of memory in terms of "allocated but not accessible" memory. High level languages such as Java have specific tools called "garbage collectors" that automatically try to detect such lost memory and reclaim it. There is no such thing in C and thus you have to manage it yourself.