THE UNIVERSITY OF SYDNEY

COMP2129 Week 10 Tutorial

Performance

For this tutorial there are a number of code examples, contained in the **`week10_material.zip`** archive available on Piazza or eLearning. Go and download it now.

Exercise 1: Dining Philosophers

The **`philosophers.c`** file has an implementation of the dining philosophers in pthreads, with each philosopher modelled as a thread and each fork modelled as a mutex. Compile and run this program, and verify that it deadlocks.

- Why does it deadlock? Go through the conditions required for a deadlock.
- How can this be fixed? There is a very simple change to the program which will stop any deadlocks from occurring.
- Implement your solution, and verify that it does not deadlock, and that the dining philosophers will dine forevermore.

Exercise 2: Matrix multiplication

In this exercise the **`gprof`** profiling tool will be used to investigate a piece of code. Download the **`matrixmult.c`** code from Piazza or eLearning.

Compile the program using a profiling flag:

```
$ gcc -pg -g -lpthread -std=gnu99 -o matrixmult matrixmult.c
```

Analyse the program's performance using **`gprof`**, using the following steps:

1. Ensure the that **`-pg`** flag has been used to compile the program.
2. Execute the program as usual.
3. After the program has finished, execute the profiler:

   ```
   $ gprof ./matrixmult > mm.stats
   ```

4. View the saved information using **`less`**. You should observe that a negligible amount of time has been spent inside the **`trace()`** function, and most of the time has been spent inside the **`multiply()`** function.

Parallelise this program using pthreads. A good first step to parallelising matrix multiplication is to make each thread responsible for a set of rows in the result matrix. (You may wish to look up *block multiplication* if you're interested in a different way of parallelising matrix multiplication).

Compare the execution times of the original program and the parallel program.

Exercise 3: Sieving for primes

An efficient way of generating prime numbers is by using a *sieve*. The most commonly used sieve is called the Sieve of Eratosthenes. The Wikipedia article for this sieve is quite excellent, and you should go and look at the animated diagram at the start. This exercise will implement a sieve for the numbers in the range $[0, 100)$. (The square bracket means inclusive, the round bracket means not inclusive).

The sieve works by first assuming that every number is prime, finding the first prime, and crossing off all its multiples. Then it finds the next prime, and crosses off those multiples. When the sieve can't make any more progress, all that is left over are the primes. To start, assume that every number is prime: for this, create a **char** array with every element set to 1. The numbers 0 and 1 are not prime, so we don't set them to 1.

```
int maxnum = 100; // Sieve up to, but not including, this number.
char *isprime = calloc(maxnum, sizeof(char));

for (int i = 2; i < maxnum; i++) {
  isprime[i] = 1;
}
```

Next, find the first element in the **isprime** array which has not been marked off (is still set to 1), and mark off all its multiples. Then repeat for larger numbers: you only need to do this up to $\sqrt{maxnum}$ (why?).

```
for (int i = 0; i*i < maxnum; i++) {
  // If i is prime, mark off all its multiples.
}
```

Finally, iterate over the sieve and print out the prime elements. Verify your answer: there are 25 primes below 100, which are (in order)

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]$$

Exercise 4: Parallel blur

In this exercise, we want to utilize all 16 processors of our server. Download the file **input.tiff** and the file **blur.c** that contains the following C-code. This C-program reads the image **input.tiff** and produces an image **output.tiff**. Note that you can open tiff files with the program **eog** (eye of gnome image viewer).

1. Compile the program. You have to use the tiff library through the compiler option **-ltiff**. That is, to compile the program you can, for instance, type

```
$ gcc -g -ltiff blur.c -o blur \
    -I/labcommon/comp2129/local/include \
    -L/labcommon/comp2129/local/lib
```

(unfortunately, ucpu1 does not have libtiff, so we need to point the compiler to a version that we know exists).

2. Implement a multithreaded version of the program **`blur.c`**. Use 16 threads.

Exercise 5: Extension: Overflow

There is an executable file **`/labcommon/comp2129/bin/greeting`**, the source of which can be found in **`greeting.c`**. One line of the source (the secret phrase) has been replaced with a boring placeholder. The file is executable, not not readable. Your task is to run it, and give it a string which causes it to reveal its secret.

1. Recall that the **`scanf`** function is "unsafe" with strings. Why?
2. Look up what the characters **`0x41`**, **`0x42`**, **`0x43`**, and **`0x44`** are, either in an ASCII table or by writing a small program.
3. Recall that Intel computers are little-endian.

Good luck!