COMP2129 Week 9 Tutorial

Thread Synchronisation

Exercise 1: Timing code

The execution of a program can be timed by using the `time` bash command. Taking the `race.c` example from the last tutorial, we can measure how long it takes to run by running the `time` (`man time`) command:

```
$ gcc -std=c99 race.c -o race -lpthread
$ time ./race
counter  = 10685323
expected = 40000000

real	0m0.132s
user	0m0.481s
sys	0m0.000s
```

The *real* time is the wall-clock time taken to run the program. The *user* time is the total CPU time used, and the *system* time is the time spent in system calls or the kernel. Here you can see that the CPU time spent is larger than the wall-clock time spent, so this was executing on multiple cores simultaneously.

Sometimes it is better to have finer-grained measures of time, for example if your program has a large setup phase, and you only want to measure some operation which occurs after that. For this you can use the `clock()` (`man 3 clock`) function.

```c
#include <time.h>

clock_t tick, tock;
tick = clock();
// Run some code in the middle here
tock = clock();
printf("Time elapsed: %f seconds\n", (double) (tock - tick) / CLOCKS_PER_SEC);
```

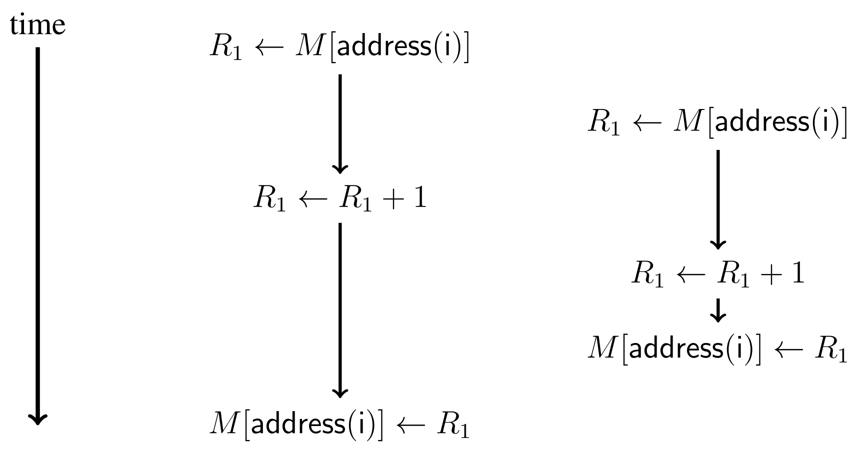
Use the above example to code to time how long it takes to `pthread_create()` all of the worker threads in `race.c`.

Exercise 2: Fixing a race condition using locking

In the previous example of `race.c`, there was a data race. This occurred because two threads were modifying the same global variable at once. The operation `i++` is broken down into three steps as listed below (M denotes the main memory, R_1 denotes a register.)

1.	$R_1 \leftarrow M[\mathsf{address(i)}]$	load the value of `i`
2.	$R_1 \leftarrow R_1 + 1$	increment the value by 1
3.	$M[\mathsf{address(i)}] \leftarrow R_1$	store the incremented value

Because of the indeterminism executing both threads, it is possible that the value of `i` is either incremented by 2 which would be the expected outcome, or is incremented by 1 if a race condition occurs. Such an example of a race condition is shown below, with the two threads in the middle and right column.


time
$R_1 \leftarrow M[\mathsf{address(i)}]$
$R_1 \leftarrow R_1 + 1$
$M[\mathsf{address(i)}] \leftarrow R_1$
$R_1 \leftarrow M[\mathsf{address(i)}]$
$R_1 \leftarrow R_1 + 1$
$M[\mathsf{address(i)}] \leftarrow R_1$

If the operation `i++` was protected by a lock, and all the threads agreed to acquire the lock before incrementing `i`, the race condition would vanish. Pthreads provides a *mutex* (stands for *mutual exclusion*) primitive for this reason.

In this exercise, you need to fix up `race.c` so that only one thread increments the variable `counter` at a time. A mutex can be created by declaring an initialising a `pthread_mutex_t`:

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
```

For this program, the mutex should be a global, and it should be initialised in `main()`, before any threads are started. Threads can ask for the lock by calling `pthread_mutex_lock()` on the mutex, and release it by calling `pthread_mutex_unlock()`.

```
pthread_mutex_lock(&lock);
counter++;
pthread_mutex_unlock(&lock);
```

Finally, after all the threads are finished, the mutex needs to be destroyed (otherwise memory leaks or other resource leaks can occur).

```
pthread_mutex_destroy(&lock);
```

You should set `BIGNUM` to `100000` for this exercise. Locking is expensive!

Exercise 3: Monte Carlo

In computational physics (and other fields), the Monte Carlo method relies on random sampling to get numeric results. The code below approximates π by randomly sampling points in the box $0 \leq x, y \leq 1$ in the plane, and counting the number of points which satisfy $x^2 + y^2 \leq 1$. The ratio of the points within the quarter circle to the total points in the box should (assuming our random number generator is "good") converge to $\pi/4$, the ratio of the areas. The following code is available on eLearning or Piazza as **monte-carlo.c**.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define ITERATIONS (100000000)

int main(void) {
  unsigned int state = 1;
  int count = 0;
  for (int i = 0; i < ITERATIONS; i++) {
    double x = (double)rand_r(&state)/(double)RAND_MAX;
    double y = (double)rand_r(&state)/(double)RAND_MAX;
    if (x*x + y*y <= 1)
      count++;
  }

  double pi = ((double)count / (double)ITERATIONS) * 4;
  printf("Estimated Pi: %.8f\n", pi);
  printf("Actual Pi    : %.8f\n", M_PI);
  return 0;
}
```

Compile using the flags **-std=gnu99 -lpthread**, and use the **time** shell command to time the execution of the program (it should take a few seconds). Check that it gives a good value of π. Your task is to parallelise this program so that it uses every available core on your machine. Check **/proc/cpuinfo** to check how many cores are available to use, and use that many worker threads.

Here we had to use the **rand_r()** function to generate random numbers, since if every thread used **rand()**, then they all would be accessing its internal state, which would become a source of contention and make the program run slowly. **rand_r()** lets the caller specify where the internal state is stored (in this case, in the variable **state**), which lets each thread hold it's own state. Make sure each thread uses a different initial random seed for the **rand_r** function, otherwise the program will just be going over exactly the same points again and again, and a worse estimate for π could occur. You should not need to use mutexes here, only creating and joining threads.

After you've finished, check that you get a similar answer for π, and check that the speedup you experience is proportional to the number of threads you launched. You should see user time exceed real time by a significant amount.

Monte Carlo integration (and a lot of other forms of numeric integration) is fantastic for being a embarrassingly parallel problem: no points depend on any other points. (Although the distribution of numbers *needs* to be uniform for it to work).

Exercise 4: Task Dependencies

Assignment 1 had five parts. The fifth part was to do with logging errors, and will not be considered here. The first four parts had some tasks dependent on others, while some parts were independent, and could be processed in parallel.

Draw a task dependency graph of assignment 1, parts 1 to 4, and compare your answer with a friend. Some tasks have a strict one-to-one producer-consumer relationship, and some tasks may be consuming results from multiple producers. Highlight which parts are pipeline parallism, task parallelism, or data parallelism.

Exercise 5: Extension: more general integration

Mathematically, what we did in the earlier Monte Carlo exercise was defined a function $f(x, y)$, by

$$f(x,y) = \begin{cases} 1, & x^2 + y^2 \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

and then approximated the value of the integral

$$\frac{\pi}{4} = \int_0^1 \int_0^1 f(x,y)\,\mathrm{d}x\mathrm{d}y$$

Can you see how this could generalise to other functions? Try approximating the value of the integral

$$\frac{2}{3} = \int_0^1 \int_0^1 x^2 + y^2\,\mathrm{d}x\mathrm{d}y$$

using the program you developed earlier. You may have to use `double` in places that an `int` was used before, since the function now takes on non-integer values.