THE UNIVERSITY OF SYDNEY

COMP2129 Week 11 Tutorial

Threads and Parallelism

Exercise 1: Program Design

Using a pen and paper, explain how you would write the following UNIX utilities: **`echo`**, **`sort`**, **`uniq`**. You should explain it in a high-level way, but also mention any specific C functions which have to be used. You don't have to implement any command-line flags, just the basic functionality. Here is an example of how to explain how to write **`cat`**:

1. If there are no arguments, just copy the contents of **`stdin`** to **`stdout`**, by using an **`fread()`**, **`fwrite()`** loop until **`fread()`** returns **`EOF`**, then exit.
2. If there are arguments, go through the argument list, trying to open each string as a file in read mode, for example **`fopen(argv[1], "r")`**. If the file did not open successfully, write an error message to **`stderr`** using **`perror()`**. If the file did open successfully, copy its contents to **`stdout`** using an **`fread()`** **`fwrite()`** loop.
3. Return status code 1 if any file could not be opened, and 0 otherwise.

Your tutor will come around during the tutorial and get you to explain one of the utilities, as well as perhaps further implementation details, such as sensible sizes for buffers and how to handle any errors.

Exercise 2: Dining Philosophers 2

The **`philosophers.c`** file in the **`week10_material.zip`** archive has an implementation of the dining philosophers in pthreads, with each philosopher modelled as a thread and each fork modelled as a mutex. In the previous tutorial, this was solved by imposing a locking order on the mutexes. This time, solve it by using a global semaphore which only allows a certain number of philosophers to attempt to eat at any one time.

Exercise 3: Diagnosing a deadlock

There are four threads in the following program, labelled T_1 to T_4, which are (amongst other operations) locking and unlocking three mutexes labelled A, B, and C. Their order of locking is given here:

$$T_1 : lock(A), lock(B), lock(C), unlock(A), unlock(B), unlock(C)$$
$$T_2 : lock(A), lock(B), lock(C), unlock(C), unlock(B), unlock(A)$$
$$T_3 : lock(B), lock(C), unlock(B), unlock(C)$$
$$T_4 : lock(C), lock(A), unlock(A), unlock(C)$$

Which threads have the potential to deadlock here? Draw two possible interleavings of instructions that cause at least two threads to go into a deadlock.

Exercise 4: False Sharing

Download the file **`falseshare.c`** from Piazza or eLearning.

- Describe what the program is doing.
- Identify the parallel and sequential parts of this program.
- Compile and run the program using the **`time`** utility, for 1, 2, 3, and 4 threads. Does the program speed up as much as it should? Why/why not?
- Identify where false sharing is occurring in this program, and different ways of mitigating it.
- Implement two ways of mitigating false sharing, and use the **`time`** utility again to ensure your program speeds up as you expect.

Exercise 5: Limits of parallelism

Amdahl's law gives limits on the maximum possible speedup of programs. What is the maximum possible speedup of the following programs:

1. 10% sequential, 90% parallel, with 4 processors used.
2. 90% sequential, 10% parallel, with 32 processors used.
3. 5% sequential, 95% parallel, with an unlimited number of processors.

Discuss some reasons why the maximum theoretical speedup is not often achieved in computations.

Exercise 6: Barrier Sync

On a manufacturing line, three robot arms are able to work independently on one "unit" at a time. After they are all finished, a manager advances the conveyor belt while the robot arms stay idle. This process repeats indefinitely (or at least until factory close time). Each robot arm can complete its task independently of the other arms, however each robot arm must wait for the conveyor belt to stop moving before it can begin work again.

The file **`robots.c`** (available on Piazza and eLearning) gives a correct implementation of this, however it is not very efficient, since it uses **`pthread_create()`** and then **`pthread_join()`** many times to start and stop threads. Modify the program so that there are only four threads launched by **`pthread_create()`**: the three robot workers, and an extra "manager" thread, which is responsible for waiting until the robots have finished, and advancing the conveyor belt. Make the four threads synchronise using barriers (refer to slides 52 and 53 from the Week 9 lecture), so that the program does not use **`pthread_join()`** at all.

Hint: This problem may be easier if you use two barriers, one for the robots to reach before the conveyor belt moves, and one for after the conveyor belt has finished moving.