Week 10

Processes

What are processes?

- A process is a single instance of an executed program
- For example, typing **ls** into your terminal causes your shell to **spawn** a new *process*
- Processes are containerised from other processes - i.e. they have separate memory. Registers are saved when we switch between processes, individual processes have their own view of registers.
- Processes are managed by the OS.

Each process has a unique ***pid*** or **process ID**.

Processes

How do we spawn separate processes from a C program?

- **posix_spawn**

- now_q10.c

Concurrency vs Parallelism

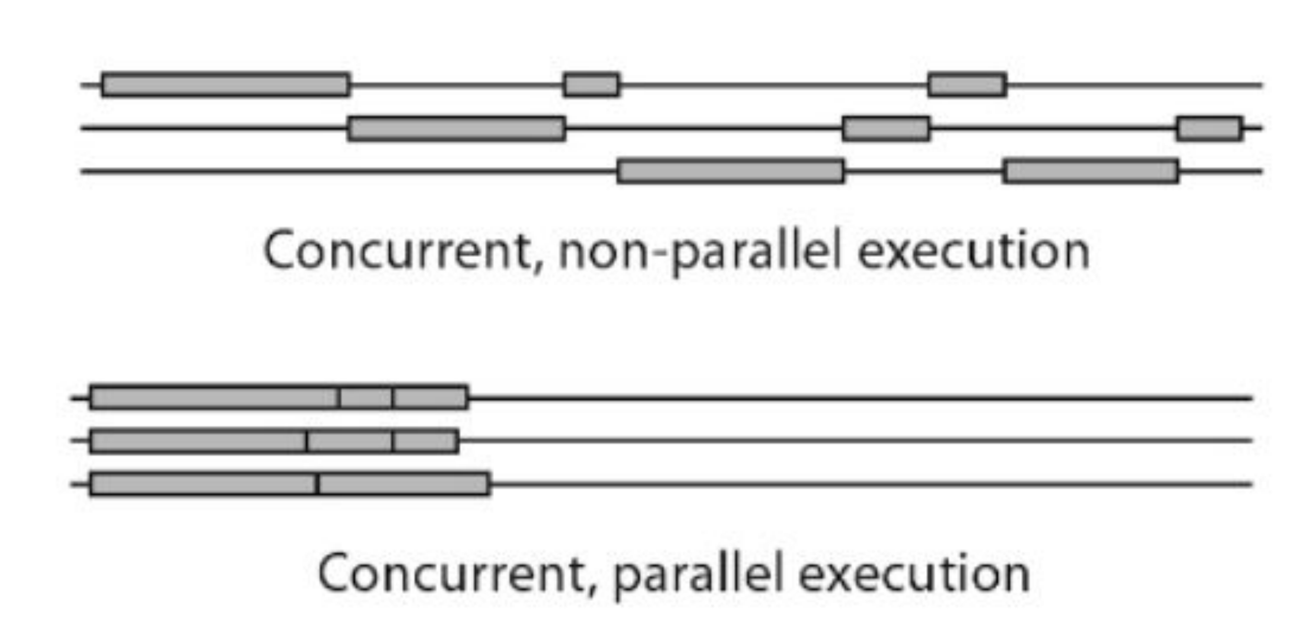
Concurrency:

- The ability for a computer to have multiple tasks at the same time
    - These tasks don’t have to be done truly simultaneously - instead we can just rapidly switch between them
    - Each task is executed on a “thread”

Parallelism:

- Parallelism requires multiple cores to allow different tasks to run truly simultaneously

Concurrency vs Parallelism



Concurrent, non-parallel execution
Concurrent, parallel execution

Writing concurrent programs

- We need a library: **<pthread.h>** (POSIX threads)
- Also need to compile with a flag: **-pthread** with gcc/dcc

- program_q5.c
- hello_q4.c

Writing concurrent programs

Why do we care about concurrency?

- Concurrency can allow our programs to perform certain actions simultaneously that were previously tricky for us to do as COMP1521 students.
- For example, with our current C knowledge, we cannot execute any code while waiting for input (with, for example, **scanf**, **fgets**, etc.).

- feed_me_q6.c

Data races

- We see that running **counter_q7.c** often gives differing values each individual execution, and always less than the expected 10000

Why does this happen?

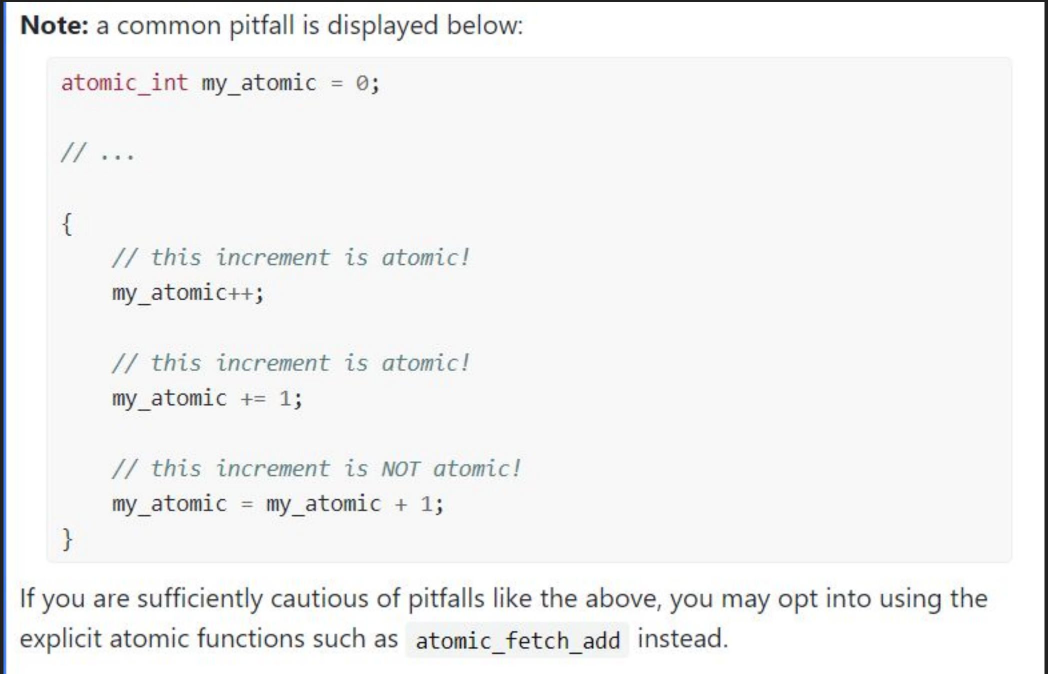
- We have a data race where multiple threads try to access data at the same time - in this case, both threads could have obtained the initial value of global_total rather than a value which has been updated by another thread
- Demo here: https://web.cse.unsw.edu.au/~xavc/data-race/

Fixing data races - mutual exclusion

- Mutual exclusion prevents threads from running certain code which access the same data at the same time
- Look at the **pthread_mutex_t** type, as well as **pthread_mutex_lock** and **pthread_mutex_unlock**

Fixing data races - atomic types

- We can make our increment an atomic operation, so it is no longer possible for threads to interleave partway through an increment.



Note: a common pitfall is displayed below:

```
atomic_int my_atomic = 0;

// ...

{
    // this increment is atomic!
    my_atomic++;

    // this increment is atomic!
    my_atomic += 1;

    // this increment is NOT atomic!
    my_atomic = my_atomic + 1;
}
```

If you are sufficiently cautious of pitfalls like the above, you may opt into using the explicit atomic functions such as `atomic_fetch_add` instead.

Deadlocks

- When two or more threads or processes are unable to proceed because each is waiting for the other to release a resource.

- **deadlock.c**