G52OSC
OPERATING SYSTEMS AND CONCURRENCY

Introduction to Concurrency

Dr Jason Atkin

Module overview - reminder

- Module was originally two modules
  - Operating systems
  - Concurrency
- Integrated into one module
  - Half of module is Operating Systems, G53OPS
    - Taught by Dr Geert De Maere
  - Other half is: Concurrency, using operating system functionality
    - Taught by Dr Jason Atkin
- My thanks and appreciation to Brian Logan for the previous concurrency course slides

What do we mean by concurrency?

Types of concurrency

Concurrency

- We can all do different things at the same time
  - Distributed processing
- Sometimes **one** of us can try to do **multiple** things, **switching between them**
  - Time sharing / switching
- And sometimes we really can do multiple things at the same time
  - E.g. writing a document while waiting on hold on a phone (does this count? listening?)

Types of Concurrency

- Exactly the same issues occur with computers as with people doing multiple things
  - Plus some really **can** do **multiple** things **at the same time**, even when not just ‘waiting’
- **Multiple machines** (different computers) doing things at the same time
- **Multiple processes** (different ‘running programs’) doing things at the same time
- **Individual processes** perhaps doing multiple things at the same time (multiple threads)

Definitions for these discussions

- **Machine:** a physical entity which has one or more processors and is physically independent of other machines
  - Communication method is needed between machines
- **Processor:** a ***core*** of a CPU, which (in theory) can do only one thing at once (we'll ignore pipelining for the moment)
  - A machine could have multiple processors which potentially can share some or all of the memory
- **Terminology issue:** do not think of 'multi-core processor' as one processor, think 'multiple processors' (a useful abstraction)

Distributed Processing

- Processes on **different (multiple) machines** doing things at the same time
  - Each has their own processor(s) so do not slow each other down
    - Other processes on the same machine may though
  - Need to communicate to coordinate?
    - Harder when processes are distributed
    - Do they need to wait for each other?
  - Need to share resources? (where are these?)

Multiple processes

- ***Multiple processes*** (different programs?) doing things at the same time on the ***same machine***
  - Limits to the simultaneous operations
    - Theoretically one operation per processor (/core)
  - What if not enough processors?
    - Something must manage them – scheduling problem
- Types of scheduling:
  - ***Cooperative multitasking***: process says '*I've done enough for a while, I'm at a great place to have a break, give someone else a go*' (**`yield()`**?)
  - ***Preemptive multitasking***: task can be interrupted at (almost) any time, to allow the next one to have a go

Multiple threads

- **Single process** can do **multiple things simultaneously** – using multiple threads

- Multiple ***threads*** within a process:
  - Same limits to the simultaneous operations as processes, and same scheduling problem
    - Theoretically one operation per processor (/core)
  - Some potential advantages over processes
    - More things can usually be shared
    - Potential runtime resource usage benefits
    - Potentially less work for operating system
    - Easier for programmer? Some data implicitly shared

Potential Problems

- **Same *conceptual* problems in each case**
  - Distributed by: machine, process, thread
  - Some problems more important in some cases
    - Communication vs data sharing balance?
  - Some problems more easily solved than others
  - Often different practical solutions (i.e. different code/functions)
- **Communication**
  - Tell others **what** you are doing, or **results** of doing so
    - Messages (queues/storage)?
    - Via shared data/memory?
- **Data sharing**
  - Simultaneous changes? (What is result then?)
  - Atomic actions? (uninterruptable vs interrupt in middle)
  - Multiple/local copies? Sharing changes/integrity issues?

Some classifications then…

- Distributed processing
  - Communication is important
  - Resource changes usually have to be explicitly shared
- Multi-processing
  - Running processes (or threads) across multiple processors, with easily-shared resources (e.g. memory)
- Multi-tasking
  - Timesharing a single processor (/core)
  - Simulated concurrency

Concurrent programs

A program

Load A
Increment A
Load B
Add A to B
Store B
Store A
Load C
Decrement C
Store C

- All programs run operations one at a time, in a specified order
- Thread of execution/control, works through the operations
  - **One thread: sequential program**
  - **Multiple: concurrent program**
- Note: (important later)
  - Compiler can reorder your code, keeping the effects
    - The ones that it knows about anyway
  - Processor can reorder it at runtime

Examples of concurrent programs

- GUI-based applications: e.g., javax.swing
- Mobile code: e.g., java.applet
- Web services: HTTP daemons, servlet engines, application servers
- I/O processing: concurrent programs can use time which would otherwise be wasted waiting for slow I/O
- Real Time systems: operating systems, transaction processing systems, industrial process control, embedded systems etc.
- Parallel processing: simulation of physical and biological systems, graphics, economic forecasting etc.

Reasons for concurrency

- It makes the program easier to write
  - Switching between two tasks you want to do at the same time is complex
    - Much easier to code them as two threads and let the operating system switch them
  - Utilise time when process is waiting for input
    - May be easier to code it to ‘wait’ than to explicitly make it do something else in the meantime
    - E.g. ‘read 10 bytes and return when you have done so’ vs ‘is there another byte there yet? If so then read, else …’
- You want the speed (not always possible)
  - i.e. You want to use **multiple processors** at once
    - You **need** to separate the threads of control for each
    - But may still have resource interdependencies ☹

Module aims

New techniques needed

- **Concurrent programs are intrinsically more complex than single-threaded programs**:
  - When more than one activity can occur at any time, program execution is necessarily nondeterministic;
    - Code may execute in surprising orders—any order that is not explicitly ruled out is allowed
  - A field set to one value in one line of code in a process may have a different value *before the next line of code is executed in that process;*
    - Program cannot assume that shared resources do not change
- ***Writing concurrent programs requires new programming techniques***

Half-module aims

- “To convey a basic understanding of the concepts, problems, and techniques of concurrent programming and concurrency in operating systems”
  - “Understanding of the concepts, problems and techniques of concurrent programming”
  - Examples of concurrency problems and their solutions
  - Classic problems and solution methods (patterns)
  - Understanding of concurrency primitives
- “To show how these can be used to write simple concurrent programs”
  - “The ability to write simple concurrent programs”
  - “Enhanced programming skills”
  - You will be examining, running and writing C and Java programs
  - You will be using operating system API functions

Content of this half module

1. Introduction and fundamentals
2. Windows GUI and Message Loop basics
   – Plus Linux equivalents
   – Already know Java basics
3. Shared data and lock-free programming
4. Critical Sections and Mutexes
5. Events, Signals, MessageQueues
6. Semaphores
7. Monitors (and Java), Thread Pools, etc
8. Correctness and examples
9. Summary and wrapping up

Scope of half-module

- We will concentrate on implementations within Windows C
  - Will also compare against Linux C and Java
- Why?
  - I could choose C or Java – you have seen these
  - This is G52OSC not G52CON, so I wanted to be closer to the operating system
  - I can show some things in C that are not possible in Java (means I can give examples you can run)
    - E.g. “test and set” operation, importance of location in memory (for cache)
  - You get to understand how application programming works in more detail

Links with Operating Systems

- We may need to use operating facilities to aid multi-processing
  - E.g. lock facilities, shared memory, etc
    - First part of coursework

- Operating systems need to take concurrency into account
  - Operating systems are concurrent systems
    - Second part of coursework

Module structure

Structure

- 4 lectures per week
- 1 lab

- 25% Coursework
- 75% Exam : Do 3 from 5 questions
  - 3 Operating Systems
  - 2 Concurrency

Labs and Coursework

- The labs will teach you:
  - How to use various concurrency primitives
  - The basics of Windows programming
  - Help you with (shared) memory allocation
- Lectures will introduce these concepts
- There is a coursework worth 25%
  - Remember that this is a 20 credit module
  - Equivalent to 50% of a 10 credit module
- Coursework uses the things from the labs

Concurrency Books

- Recommendations from previous Concurrency Module
  - I think you can get away with the web resources
- Andrews (2000), Foundations of Multithreaded, Parallel and Distributed Programming, Addison Wesley
- Lea (2000), Concurrent Programming in Java: Design Principles and Patterns, (2nd Edition), Addison Wesley
- Ben-Ari (1982), Principles of Concurrent Programming, Prentice Hall
- Andrews (1991), Concurrent Programming: Principles & Practice, Addison Wesley
- Burns & Davis (1993), Concurrent Programming, Addison Wesley
- Magee & Kramer (1999), Concurrency: State Models and Java Programs, John Wiley

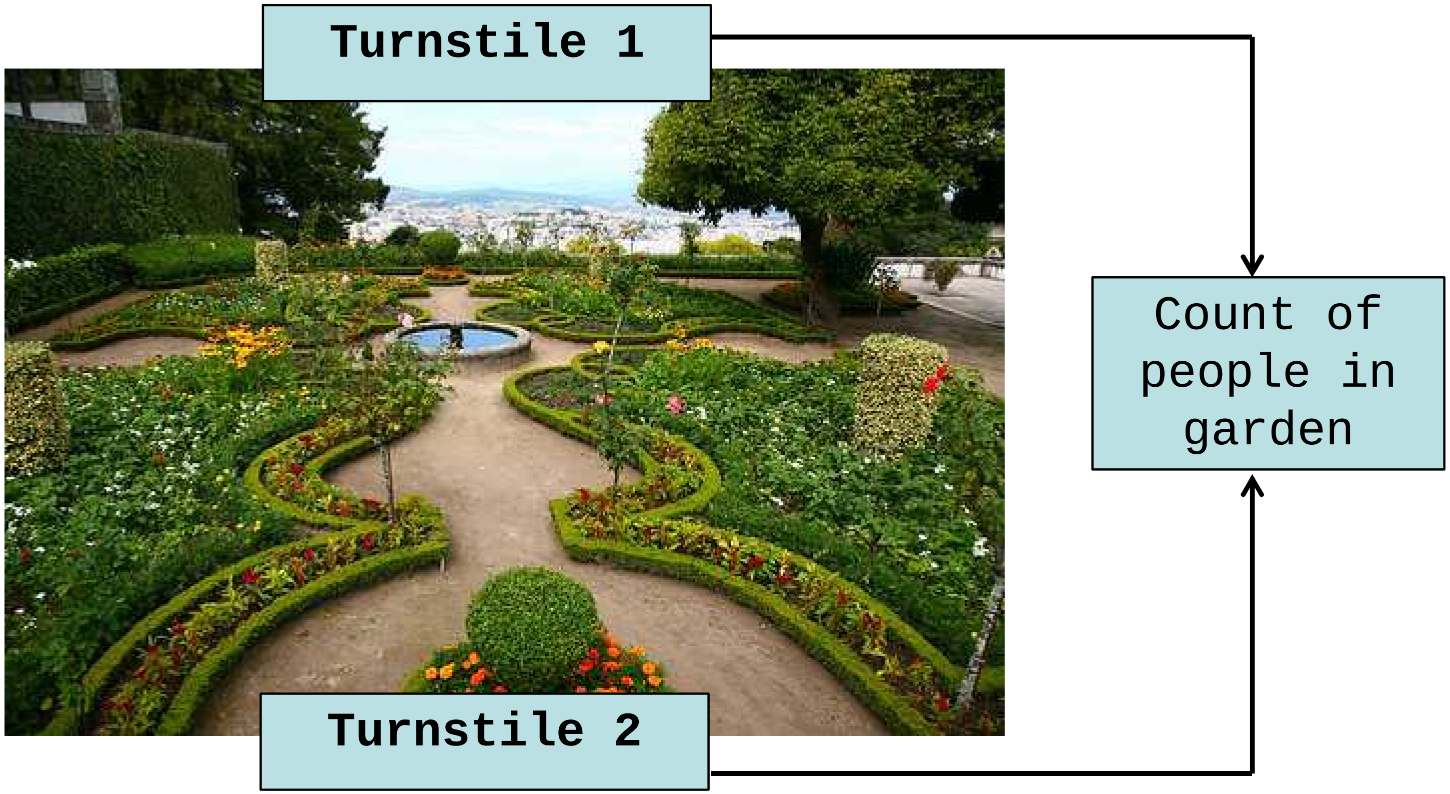
Windows programming books

- Problem is that most are C++ rather than C, since classes make programming much easier
- **The web/MSDN is a more than adequate resource**
- Concurrent programming on Windows, Joe Duffy, Addison Wesley
- Programming Windows, 5th Edition, Charles Petzold (a classic), Microsoft Press
- Windows via C/C++ 5th Edition, Richter and Nasarre, Microsoft Press
- Multicore Programming, Design and Implementation for C++ Developers, Hughes and Hughes, Wrox
- For the operating system implementation side: Windows Internals (various editions), Microsoft Press

An example

An example concurrency problem

- Ornamental garden problem:
  - Search for: "ornamental garden" concurrency
- Each should increment a **shared** counter when someone enters the garden



Example function

```
#define NUMBER_THREADS 2

// Windows name for an unsigned long - 4 bytes
volatile DWORD dwTotal = 0;

// Function called by every thread
DWORD WINAPI thread_function( LPVOID lParam )
{
      for ( int i = 0; i < 1000000; i++ )
      {
            dwTotal++;
      }
      return 0;
}
```

Asides: windows things

```
volatile DWORD dwTotal = 0;
```

- Use **volatile** (standard C specifier) when the variable may be accessed from multiple threads
- DWORD specifies a size of 4 bytes, and is a name for the unsigned long type

```
DWORD WINAPI thread_function
       ( LPVOID lParam )
```

- WINAPI is a flag to tell the compiler the format to lay out the function call on the stack
- LPVOID means void* (long pointer to void)

What is the result when run twice?

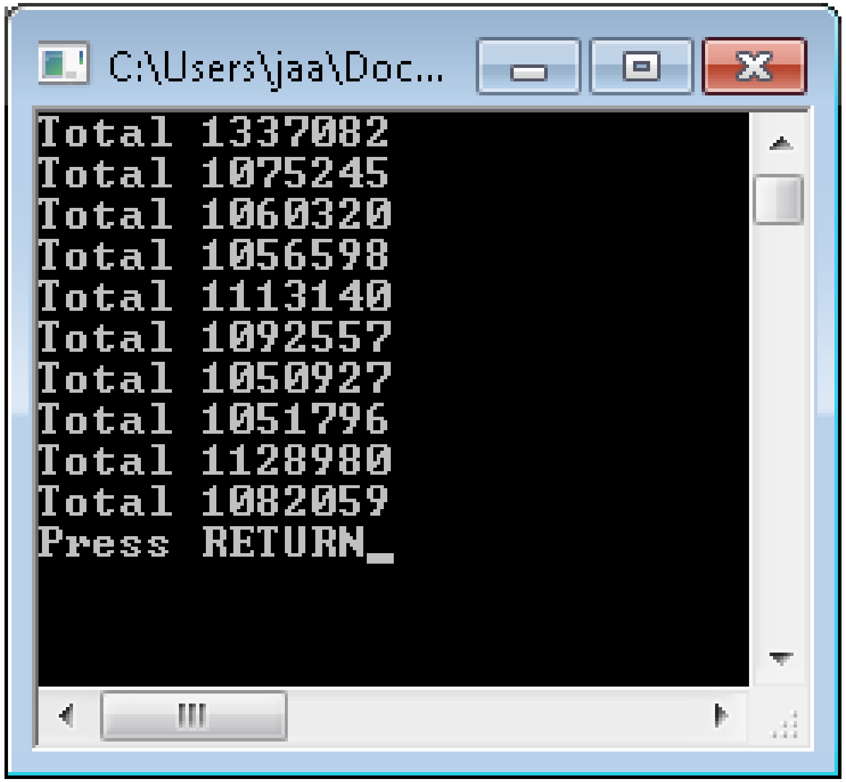
```
// Function called by every thread
DWORD WINAPI thread_function(LPVOID lParam)
{
    for ( int i = 0; i < 1000000; i++ )
    {
        dwTotal++;
    }
    return 0;
}
```

Each function increments the variable one million times

Run the function twice, at the same time

What will the result be?

Results



- Should have been 2,000,000

- Problem:

  `dwTotal++;`

  - Get current value
  - Increment
  - Write new value back

Results not what we expect. A problem to investigate.

Next Lecture

- C and Pointers
  - We NEED to thoroughly understand the basics of pointers to work with memory
  - Remember: Memory layout matters!
- #include and #define
  - Conditional compilation
- Pointers
  - Address of operator
  - Copying/assigning pointers
  - Dereferencing
  - Arrays
  - C-style strings

A simple program – RED matters

```
#define WIN32_LEAN_AND_MEAN
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_RUNS 2

volatile DWORD dwTotal = 0;

DWORD WINAPI my_function(
      LPVOID lpParam )
{
   for ( int i = 0;
      i < 1000000;  i++ )
   {
      dwTotal++;
   }
   return 0;
}
```

```
int main(int argc, char* argv[])
{
   int iTN = 0;
   dwTotal = 0;
   for ( iTN = 0;
         iTN < NUM_RUNS;
      ++iTN )
   {
      my_function( … );
   }
   printf("Total %d\n",dwTotal);

   printf( "Press RETURN" );
   while ( getchar() != '\n' )
      ;
    return 0;
}
```