Computer Fundamentals: Operating Systems, Concurrency

Dr Robert Harle
This Week

- The roles of the O/S (kernel, timeslicing, scheduling)
- The notion of threads
- Concurrency problems
- Multi-core processors
- Virtual machines
Traditionally

- A single program for a single user at a single time
Operating System
A single computer for multiple users each executing a single program
Microprocessors (early 80s)

- A dedicated machine for each person running a single program
Multitasking (80s+)

- A dedicated machine for each person running multiple programs
OS Functions

- A modern OS does a lot
- Abstracts hardware (allows you to write code to e.g. access HDD and takes care of the different HDDs for you)
- Schedules processes
- Allocates main memory (to individual processes)
- Provides library of useful functions (e.g. get system time, load file, etc)
- Enforces security
- May provide libraries to create a GUI
The kernel is the part of the OS that runs the system

- Just software
- Handles process scheduling (see later)
- Access to hardware
- Memory management

Very complex software – when it breaks... game over.
Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (0% complete)

EIP: 0060:[c03ca9af] Not tainted ULI
EFLAGS: 00010246 (2.6.8-prep)
EIP is at find_isa_irq_pin+0x8/0x5d
eax: 00000000 ebx: 00000000 ecx: 00000003 edx: 00000003
esi: c75f0000 edi: 01234567 ebp: c75f0000 esp: c75fe8c
ds: 007b es: 007b ss: 006b
Process reboot (pid: 3505, threadinfo=c75f0000 task=d88b01b0)
Stack: c011acf0 01234567 01234567 00000000 c75f0000 01234567 c0117a3 00000000
c013d63 c03252a29 df6a61b8 c13ee000 00c59fe0 d481ee0 00800001 dbf9400
00c59fe0 dcd3300c d481ee0 c805700d 00000000 d480a164 dcd3300c dbf9400
Call Trace:
[[c011acf0]] disable_10_APIC+0x16/0x1b6
[[c0117a3]] _machine_restart+0x6/0x6c
[[c013d63]] sys_reboot+0x19a/0x50f
[[c015700d]] _handle_mm_fault+0xe5/0x229
[[c011ce75]] do_page_fault+0x1a5/0x1f4
[[c01864b7]] _destroy_inode+0x36/0x45
[[c0181fab]] dput+0x33/0x4f3
[[c0168a36]] __fput+0xc9/0xee
[[c0167163]] filp_close+0x59/0x5f
[[c0310c7b]] syscalls_call+0x7/0xb
Code: a2 f6 9f 5f e4 89 37 c8 78 47 c8 78 47 c8 78 47 9b 53 2b 9b 53 2b 8f 34 11
6c 36 24 6d 2c 0c 4b 26 14 2a 14 8f 67 4e 35 6d 2c 8c <63> 17 01 6d 2c 0c 46 13
80 46 13 80 46 13 00 63 17 01 6d 2c 0c
The kernel allocates chunks of main memory to each process. It tries to prevent a program from accessing anything outside its allocation.
Modern OSes allow us to run many programs at once ("multitask"). Or so it seems. In reality a CPU time-slices:

- Each running program (or "process") gets a certain slot of time on the CPU
- We rotate between the running processes with each timeslot
- This is all handled by the OS, which schedules the processes. It is invisible to the running program.
Context Switching

- Every time the OS decides to switch the running task, it has to perform a **context switch**
- It saves all the program's context (the Fetch Execute stuff like program counter, register values, etc) to (main) memory
- It loads in the context for the next program
- Obviously there is a time cost associated with doing this...
Choosing a Timeslot Size

- The computer is more efficient: it spends more time doing useful stuff and less time context switching
- The illusion of running multiple programs simultaneously is broken
- Appears more responsive
- More time context switching means the overall efficiency drops
Relinquishing a Timeslot Early

- Sometimes a process is stuck waiting for something to happen (e.g. data to be read from disk)
- The process is “blocked”
  - Should release (yield) its timeslot
  - How can we know when to unblock it?

A blocks and yields

A B C A B C D B C D B
We could periodically check ("poll") to see whether the data is there.

Essentially keep scheduling the process even though it will mostly be doing pointless checks.

Easy but obviously inefficient.
Interrupts

- Modern systems support **interrupts**
- Just signals that something has happened. An **interrupt handler** is associated with each interrupt
- E.g. HDD raises an interrupt to say it's done getting data → scheduler unblocks the process
Platforms

- Almost all significant programs make use of the library functions in an OS (e.g. to draw a window)
- Our machine code needs not only a specific instruction set, but also the relevant operating system (with its libraries) installed
- So software is typically compiled for a specific **platform**: a (architecture, OS) pair
  - x86/Windows
  - ARM/Windows
  - x86/Linux
  - ARM/iOS
  - X86/OSX
Threads and Concurrency!
Sometimes a program needs to do background tasks whilst still performing a foreground task.

E.g. run an intensive computation but still process mouse events in case the user hits cancel.

Processes have **threads**: effectively sub processes that run and are scheduled independently.
- Threads run independently but **share memory**

![Diagram showing processes and threads](image)
Multiple CPUs

- Ten years ago, each generation of CPUs packed more in and ran faster. But:
  - The more you pack stuff in, the hotter it gets
  - The faster you run it, the hotter it gets
  - And we got down to physical limits anyway!!

- Some systems had multiple CPUs to get speed up
Multicore CPUs

- Modern system contain chips with multiple cores: multiple CPUs in a single package
- Connections shorter → faster
- Lower power

Main Memory

![Diagram showing connections between CPU cores and main memory.]
Two cores run completely independently, so a single machine really *can* run two or more applications simultaneously. 

BUT the real interest is how we write reliable programs that use *more* than one core or thread.

This is hard because they use the same resources, and they can then interfere with each other.

Those sticking around for IB CST will start to look at such *concurrency* issues in far more detail. We will just look at...
Race Conditions

\[ c = c + 1; \]

\[ c = 5 \]

\[ c = c - 1; \]
Race Conditions

Thread 1

LOAD c x
ADD #1 x
STORE x c

Thread 2

LOAD c x
SUB #1 x
STORE x c

Main memory

c=5
Race Conditions

Thread 1

LOAD c x
ADD #1 x
STORE x c

Thread 1 Register

5
6
6

Thread 2

LOAD c x
SUB #1 x
STORE x c

Thread 2 Register

6
5
5

Main Memory

5
5
6
6
6
5
Race Conditions

Thread 1
- LOAD c x
- ADD #1 x
- STORE x c

Thread 1 Register
- 5
- 6
- 6

Thread 2
- LOAD c x
- SUB #1 x
- STORE x c

Thread 2 Register
- 5
- 4
- 4

Main Memory
- 5
- 5
- 4
- 6
Race Conditions

Thread 1

LOAD c x
ADD #1 x
STORE x c

Thread 1 Register

Thread 2

LOAD c x

SUB #1 x
STORE x c

Thread 2 Register

Main Memory

5
5
5
5
6
6
4
4
5
5
5
6
6
4
Race Conditions

- When we have two or more threads sharing a piece memory the result can depend on the order of execution
  → “Race condition”

- Hard to detect (non-deterministic)
- Hard to debug
- Generally just hard
Solving Race Conditions

- Risky sets of operations like this must be made atomic
- i.e. no context switching once the code block is started
- *Not trivial* → much of CST IB devoted to this

LOAD c x
ADD #1 x
STORE x c
Aside: The Value of Immutability

- If something is immutable, the race conditions go away since you can only read it → remember this for OOP
Emulation

- Go back 20 years and emulators were all the rage: programs on architecture X that simulated architecture Y so that programs for Y could run on X.
- Essentially interpreters, except they had to recreate the entire system. So, for example, they had to run the operating system on which to run the program.

- Now computers are so fast we can run multiple **virtual machines** on them.
- Allows us to run multiple operating systems simultaneously!
Virtualisation

- This is time-sharing reinvented, with steroids
- Underpins the internet services we have today

Heavy load

- Windows 7
- Windows 7
- Windows xp
- Windows 7
- Ubuntu
- Android

Light load

- Windows 7
- Windows 7
- Ubuntu
So what have we learnt?

- Operating systems are complex pieces of software
- They are really a collection of management processes, each in charge of a different thing
- Multitasking is faked through timeslicing
- Multiple cores within a CPU were introduced to boost performance on multitasking systems
- All this parallelism leads to lots of tricky concurrency issues that we're still trying to bottom out.