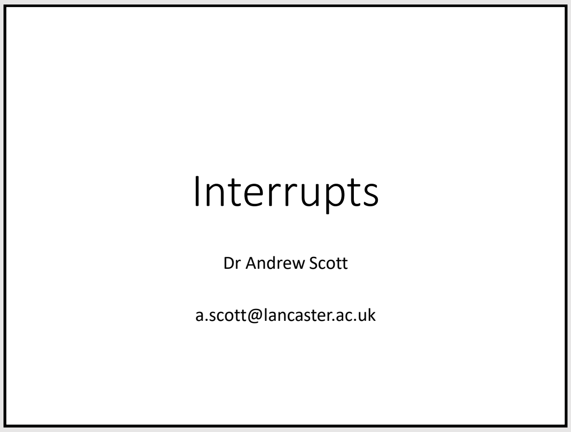
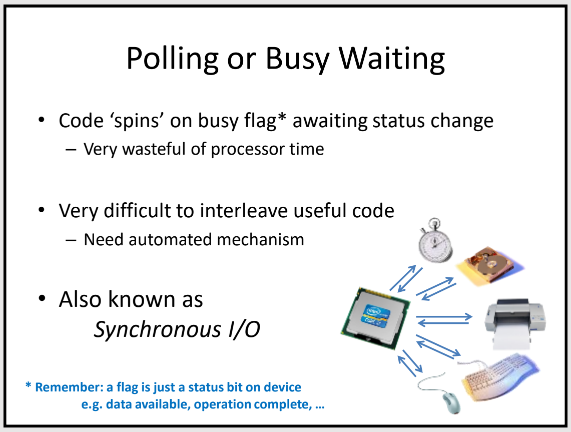


Interrupts

Dr Andrew Scott

a.scott@lancaster.ac.uk

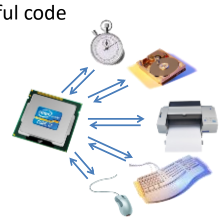
1



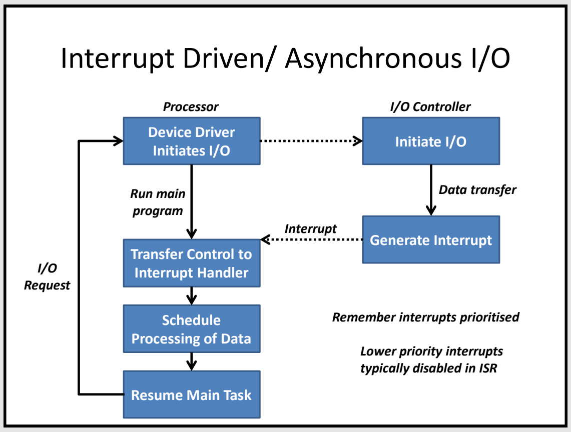
Polling or Busy Waiting

- Code ‘spins’ on busy flag* awaiting status change
  - Very wasteful of processor time
- Very difficult to interleave useful code
  - Need automated mechanism
- Also known as
  Synchronous I/O

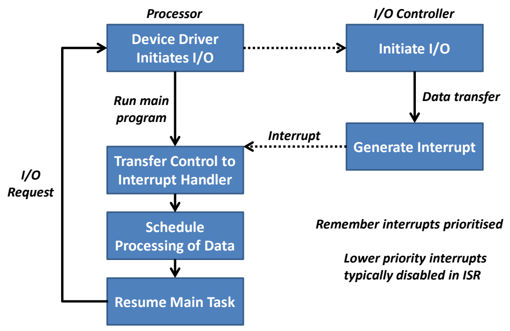
* Remember: a flag is just a status bit on device
e.g. data available, operation complete, …



2



Interrupt Driven/ Asynchronous I/O

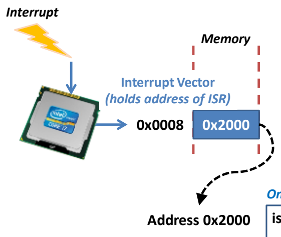


Remember interrupts prioritised

Lower priority interrupts typically disabled in ISR

3

Interrupt Service Routine (ISR)



Note we push a copy of all registers so C handler has access to all system state

Some interrupt systems by default

- Disable interrupts - typically do
- Push all registers - easy but slow
- Push few, if any - fast

One of the few places we need to use assembler

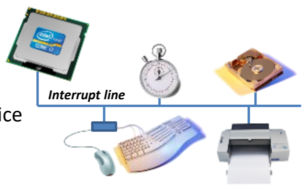
```
isr:    pushal    # all registers
        call      C_handler
        popal     # cf. pushal
        sti       # re-enable interrupts*
        iret      # restart main program
```

** iret instruction on many processors re-enables interrupts, so wouldn't need sti*

4

Interrupts

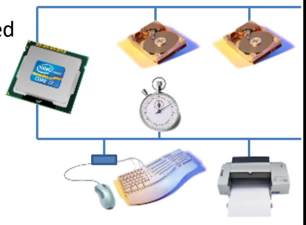
- If devices share interrupt/ interrupt line
  - Know at least one device changed status …better, but no idea which (generated interrupt)
    - Back to polling
- After interrupt
  - Must poll each device for data ready/ state change



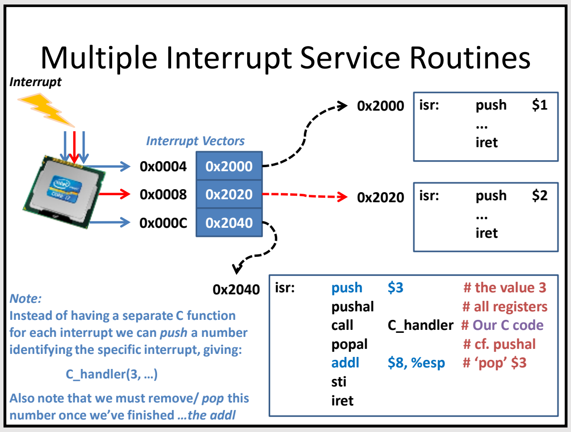
5

Multiple Interrupt Lines

- Narrows search when subsequently polling
- Groups similar devices
- More support required on processor
  - Multiple lines/ wires
    - Cost, complexity, …
  - Multiple ISRs



6



Multiple Interrupt Service Routines

```
isr:    push    $1
        ...
        iret
```

```
isr:    push    $2
        ...
        iret
```

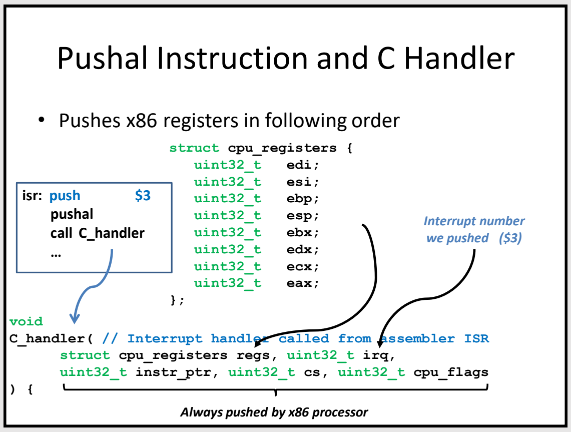
```
isr:    push    $3           # the value 3
        pushal               # all registers
        call    C_handler    # Our C code
        popal                # cf. pushal
        addl    $8, %esp     # 'pop' $3
        sti
        iret
```

Note:
Instead of having a separate C function for each interrupt we can *push* a number identifying the specific interrupt, giving:

C_handler(3, …)

Also note that we must remove/ *pop* this number once we've finished …*the addl*

7



Pushal Instruction and C Handler

- Pushes x86 registers in following order

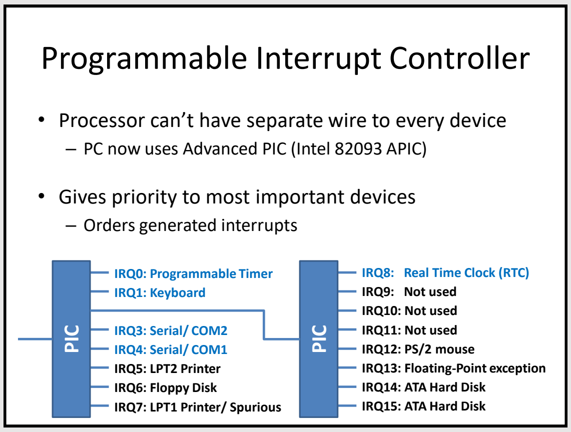
```
isr: push   $3
     pushal
     call C_handler
     ...
```

```
struct cpu_registers {
   uint32_t   edi;
   uint32_t   esi;
   uint32_t   ebp;
   uint32_t   esp;
   uint32_t   ebx;
   uint32_t   edx;
   uint32_t   ecx;
   uint32_t   eax;
};

void
C_handler( // Interrupt handler called from assembler ISR
      struct cpu_registers regs, uint32_t irq,
      uint32_t instr_ptr, uint32_t cs, uint32_t cpu_flags
) {
```

Interrupt number we pushed ($3)
Always pushed by x86 processor

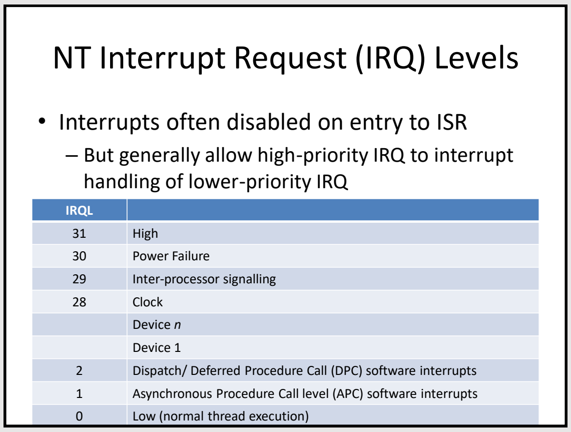
8



Programmable Interrupt Controller

- Processor can't have separate wire to every device
  - PC now uses Advanced PIC (Intel 82093 APIC)
- Gives priority to most important devices
  - Orders generated interrupts

10

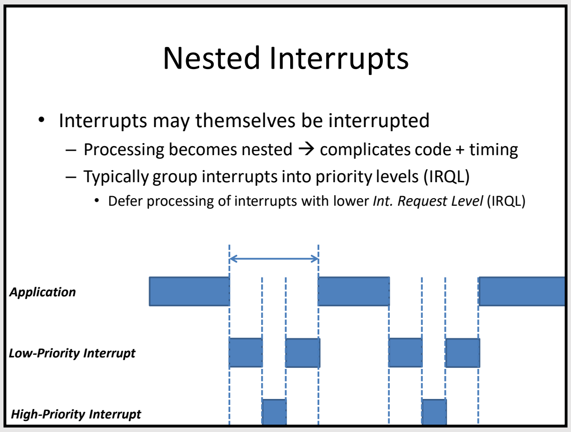


NT Interrupt Request (IRQ) Levels

- Interrupts often disabled on entry to ISR
  - But generally allow high-priority IRQ to interrupt handling of lower-priority IRQ

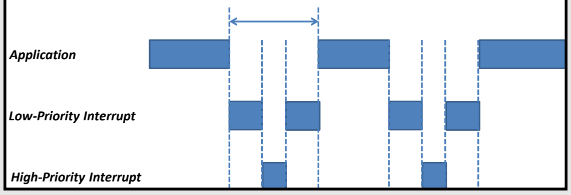
IRQL	
31	High
30	Power Failure
29	Inter-processor signalling
28	Clock
	Device *n*
	Device 1
2	Dispatch/ Deferred Procedure Call (DPC) software interrupts
1	Asynchronous Procedure Call level (APC) software interrupts
0	Low (normal thread execution)

13

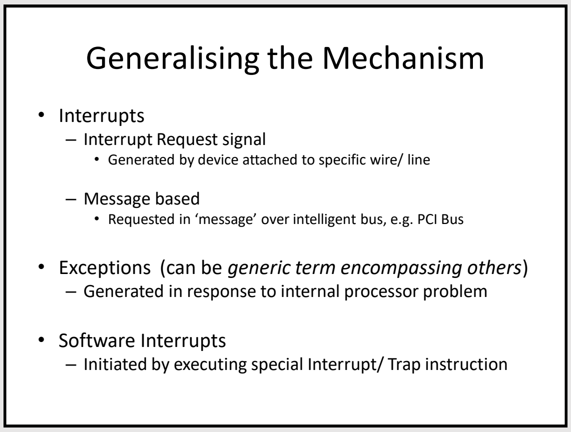


Nested Interrupts

- Interrupts may themselves be interrupted
  - Processing becomes nested → complicates code + timing
  - Typically group interrupts into priority levels (IRQL)
    - Defer processing of interrupts with lower *Int. Request Level* (IRQL)



14



Generalising the Mechanism

- Interrupts
  - Interrupt Request signal
    - Generated by device attached to specific wire/ line
  - Message based
    - Requested in 'message' over intelligent bus, e.g. PCI Bus
- Exceptions (can be *generic term encompassing others*)
  - Generated in response to internal processor problem
- Software Interrupts
  - Initiated by executing special Interrupt/ Trap instruction

15

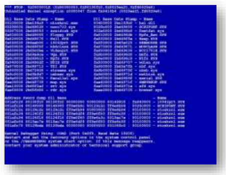
x86 Exception List

```
const char *
x86Exception[ ] = {
   "Division by zero",                                  //  0 F  -
   "Single step",                                       //  1 FT -
   "Non-Maskable Interrupt (NMI)",                      //  2 I  -
   "Breakpoint",                                        //  3 T  -
   "INTO executed with overflow flag set",              //  4 T  -
   "BOUND executed with index outside array bounds",    //  5 F  -
   "Invalid Op-code",                                   //  6 F  -
   "Device/ Co-processor not available",                //  7 F  -
   "Double fault - no handler for exception",           //  8 A  0
   "Co-processor Segment overrun (pre i486)",           //  9 F  -
   "Invalid Task State Segment (TSS)",                  // 10 F  E
   "Segment not present",                               // 11 F  E
   "Stack/ Segment Fault",                              // 12 F  E
   "General Protection Fault (GPF)",                    // 13 F  E
   "Page Fault",                                        // 14 F  E
   "Exception 0x0F - reserved by Intel",                // 15    -
   "x87 Floating Point exception",                      // 16 F  -
   "Alignment check (Ring 3: user mode)",               // 17 F  0
   "Machine check (internal processor error)",          // 18 A  ?
   "SIMD Floating Point exception",                     // 19 F  -
```

16

When Exceptions go Wrong

- Interrupts MUST be correctly handled
- If no interrupt service routine set for interrupt
  - Processor raises ***Double Fault*** interrupt
  - Unknown state can be hard to recover from
    - Blue Screen of Death
- If double fault isn't handled
  - Processor ***Triple Faults***
  - Causes machine restart



17