Computer Systems
Lecture 18

Interrupts

Dr Lito Michala, Dr José Cano
School of Computing Science
University of Glasgow
Spring 2020

Outline

- Interrupts
- Using interrupts to catch errors
- Concurrent processes
- How interrupts are implemented

Control constructs

- Control constructs determine the order of execution of statements
- We have seen some high level control constructs
  - if b then S
  - if b then S1 else S2
  - while b do S
  - for var := expr1 to expr2 do S
  - procedure
  - And there are plenty more
- These are implemented using just a couple of low level control constructs
  - goto L
  - if b then goto L
- But there is one more low level primitive: interrupts

Another kind of control: losing control!

- Control constructs built on goto and if-then-goto let the program decide what to do next

- Sometimes we want something else (not the program) to decide what to do next

Interrupts

- The hardware provides interrupts which are used to implement processes
- An interrupt is an automatic jump
- It goes either to the operating system or to an error handler
- But it is not the result of a jump instruction
  - It happens automatically when an external event occurs
- The program that was running never jumped to the OS
  - The processor just stops executing its instructions, and starts executing the OS instead
- It's like talking to a group of people, and suddenly you get interrupted!

What causes an interrupt

- An error in a user program
  - e.g. overflow, result of arithmetic is too large to t in a registers

- **A trap**: this is an explicit jump to the operating system, but the program doesn't specify the address

- **An external event**: a disk drive needs attention now, or the timer goes off

What happens when an interrupt occurs

- The computer is a digital circuit

- Without interrupts, it repeats forever
  - Fetch the instruction at the address in the pc register
  - Execute the instruction

- With interrupts, it repeats this forever:
  - Check to see if there is an interrupt request
  - If there is, savepc := pc, pc := address of code in operating system
  - Fetch the instruction at the address in the pc register
  - Execute the instruction

- Since the pc has been modified, the next instruction will not be part of the program that was interrupted, it will be the operating system

Saving state

- Remember, an interrupt is a jump to the OS
- This requires setting the address of OS in the pc register
- But if we simply assign a value to pc, the computer has forgotten where the interrupted program was
- Therefore the hardware must "save state": savepc := pc
- The OS has a special instruction that enables it to get the value of savepc

How interrupts are used

- Interrupts can be used to catch errors in a program, e.g. arithmetic overflow (the result is too big to t in a register)
  - If an overflow occurs (or divide by zero, or some other error) we want the program to jump to an error handler

- Trap is similar to an interrupt, and is used to request service from the OS
  - User program can't halt the machine, but uses trap to ask the OS to stop running the program

- They can be used to provide quick service to an Input/Output device
  - A disk drive may generate an interrupt when the spinning platter reaches a certain point, and it needs service right away (within a tight deadline)

- Interrupts are used to implement concurrent processes
  - The operating system gives each process a time slice in round-robin order, so each process makes progress

Interrupts and programming languages

- Most programming languages don't provide the ability to work directly with interrupts

- But many programming language provide exceptions
  - Without an exception handler, a division by 0 might terminate the program
  - In the program, you can set an exception handler: a procedure to execute if a division by 0 occurs
  - The compiler might implement this in several different ways:
    - It could put in explicit comparison and conditional jumps to check each division
    - Or it could set up an interrupt handler (this requires negotiation with the operating system)

Catching errors

- As a program runs, it may accidentally produce an error
- Two examples
  - An arithmetic instruction produces a result that's too large to fit in a register: this is called overflow
  - A divide instruction attempted to divide by 0
- It's better to detect the error and do something about it
- This makes software robust
- If the program keeps going, it's likely to produce wrong results and it won't tell
- Two approaches for catching errors (use one or the other)
  - Explicit error checking
  - Interrupts

Explicit error checking

- Most computers have a condition code register with a bit indicating each kind of error
- Sigma16 uses R15, and a bit in R15 indicates whether overflow occurred
- Every time you do an add (or other arithmetic instruction), that bit is set to 0 if it was ok, and 1 if there was overflow
- You can check for this with a conditional jump, and then take appropriate action
- Of course, you have to decide what the appropriate action is!

```
add R2,R5,R4            ; x := a + b
jumpovfl TooBig[R0]     ; if overflow then goto TooBig
```

Problems with explicit error checking

- You have to put in the *jumpovfl* after every arithmetic instruction
- This makes the program considerably longer
- It's also inefficient: those conditional jumps take time
- It is “fragile”: if you forget the jumpovdl even once in a big program, that program can malfunction

A better approach: interrupts!

- Most computers (including Sigma16) can also perform an interrupt if an overflow (or other error) occurs
- The circuit checks for overflow (or other error) after every arithmetic operation
  - If the error occurred, the circuit performs an interrupt
- The OS then decides what to do
  - User program can tell the OS in advance “in case of overflow, don’t kill me, but jump to this address: TooBig”
  - There is a special trap code for making this request
- In some programming languages, this is called “setting an exception handler” or “catching exceptions”
- There is a special control register with a bit that species whether overflow should trigger an interrupt

Why are interrupts better than explicit checking?

- Interrupts guarantee that every operation is checked
- It is **faster**: the circuit can do this checking with essentially zero overhead
- It is **easier**: the programmer doesn't have to worry about it
- The **program is shorter**: don't need a jump after every arithmetic instruction

Interrupts and processes

- One of the central features provided by an OS is concurrent processes
  - A process is a running program
  - Think of a program as a document: it's just sitting there
  - A process is the actions that happen when a program is executed: it has its variables, the variables change over time, Input/Output happens, …
  - Several different processes may be running on the same program (e.g. multiple tabs on a web browser)
  - Each process has its own variables

- Processes are implemented using interrupts

- The idea: the OS gives a user program a time slice

- The user is interrupted, and the OS can then run a different program

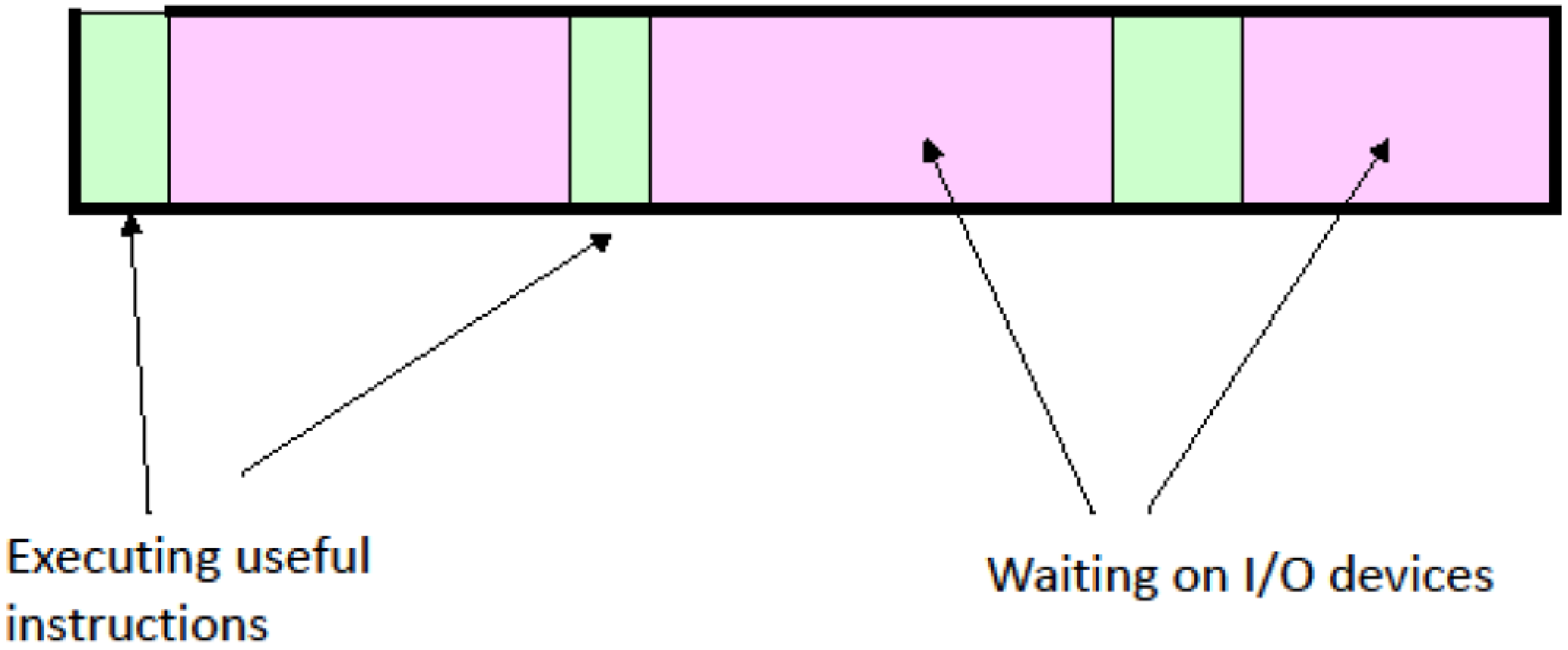
Waiting for I/O = wasted time

- Motivation for processes comes from I/O

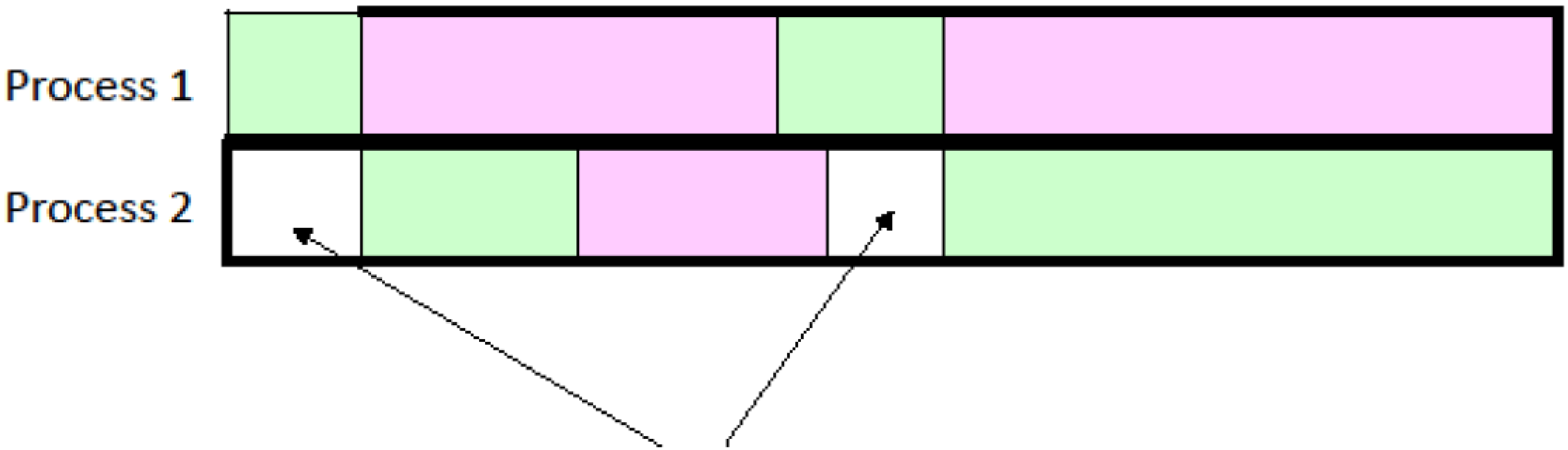
- The problem
  - Instructions execute quickly: typically about 0.3ns (about 3×10^9 per second)
  - Input/output is much slower, especially if mechanical devices are involved
  - An I/O operation runs slower than an instruction by a factor ranging from 10^4 to 10^8
  - For comparison, a supersonic jet fighter is only 10^3 times faster than a turtle

- If a program does *compute… print… compute…* it is likely to spend a lot of time waiting for the I/O

A process must sometimes wait



Don't wait: switch to another process



Ready—*not actually executing, but ready to go when the processor becomes available*

Don't wait: switching to another program

- When a program needs to perform I/O, it
  - Requests the operating system to do the I/O
  - The OS starts the I/O but doesn't wait for it to finish
  - The OS then allows a different program to run for a while
  - Eventually, when the I/O operation finishes, the OS allows the original program to resume
- This leads to an operating system running a large number of separate programs
- Each running program is called a process

Concurrent processes

- A process is a running program
- At an instant of time, the computer is physically executing just one instruction (which belongs to one process)
- From time to time (around 100 or more times per second), the system will transfer control from one process to another one (called a process break)
- Time scale
  - At the scale of a nanosecond (10^{-9} second) the computer is executing just one instruction of one process; all other processes are doing nothing
  - At the scale of human perception (10^{-2} second) it appears that all the processes are making smooth processes
- A motion picture is just a sequence of still photographs but displaying them rapidly gives the impression of continuous motion

Operating system kernel

- A process does not transfer control to another process
- How could it?
  - When you write a program, you don't know what other programs will be running when this one is!
- A process break means
  - Running process jumps to the operating system kernel
  - The kernel is the innermost, core, central part of the OS
  - The kernel has a table of all the processes
  - On Windows: right-click the toolbar, launch the Task Manager, click Processes tab
  - The kernel chooses another process to run and jumps to it

Events that can trigger an interrupt

- There is a timer that “bings” periodically
  - Each time it goes o it generates an interrupt

- When an Input/Output device has competed a read or write, it generates an interrupt

Interrupts and preemptive scheduling

- When the operating system jumps to a user process, it sets a timer which will “go off” after a set amount of time
  - e.g. 1ms = 10^{-3} second

- When does a running process jump to the operating system?
  - When the timer goes off
  - When the process makes an I/O request

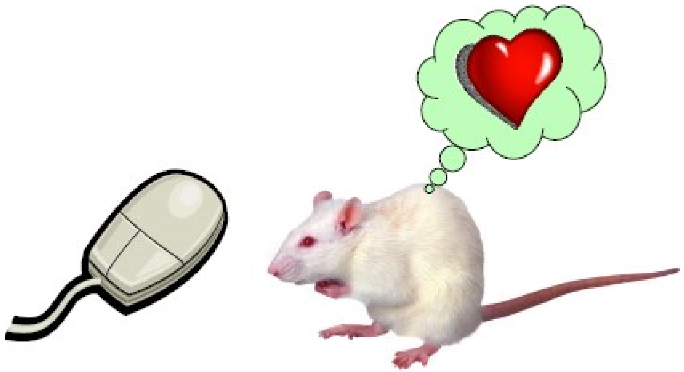
- This guarantees that the process won't run forever and freeze the system even if it goes into an infinite loop

The Scheduler

- The core of an operating system is the scheduler
- It maintains a list of all the processes
- When an interrupt occurs
  - The process that was running stops executing instructions: it has been interrupted
  - The OS takes any necessary action (e.g. service the I/O device)
  - Then the OS jumps to the scheduler
  - The scheduler chooses a different process to run
  - It sets the timer and jumps to that process

Mouse

- The mouse isn't connected to the cursor on the screen!
- When you move the mouse, it generates an interrupt
- The OS reads the mouse movement
- Then it calculates where the cursor should be and redraws it
- This happens many times per second, giving the illusion of smooth movement



How interrupts are implemented

- Interrupts cannot be implemented in software!
- The processor (the CPU) repeatedly goes through a sequence of steps to execute instructions
- This is the control algorithm and it's performed by a digital circuit in the processor (the control circuit)
- Interrupts are implemented by the control circuit

Control

- We have seen the RTM, which executes operations
  - e.g. reg[d] := reg[a] + reg[b]

- This is the core of a processor!

- We have seen the control registers: pc, ir, adr

- The processor uses these to keep track of what it is doing

The Control Algorithm

- The behaviour of the entire processor is defined by a control algorithm
- We can describe this using a special notation (which looks like a simple programming language, but it is not a program)
  - Notations
  - The control algorithm
- We can implement the control algorithm using flip flops and logic gates

Registers

- **pc** (program counter): contains address of the next instruction
- **ir** (instruction register): contains the current instruction (or first word of an RX instruction)
- **adr** (address register) holds the effective address for RX instructions
- **reg[a]** (register file): contains 16 registers for use by user program

Notation

- **pc, ir, adr**: contents of these 16-bit registers
- **ir_op,ir_d, ir_a, ir_d**: 4-bit fields in the ir
- **reg[x]**: the register in the register file with address x
- **mem[x]**: the memory location with address x

Infinite loop

- In hardware, we need infinite loops

- The computer should never stop executing instructions!

```
repeat forever
  action
  action
  ...
  action
```

Case dispatch

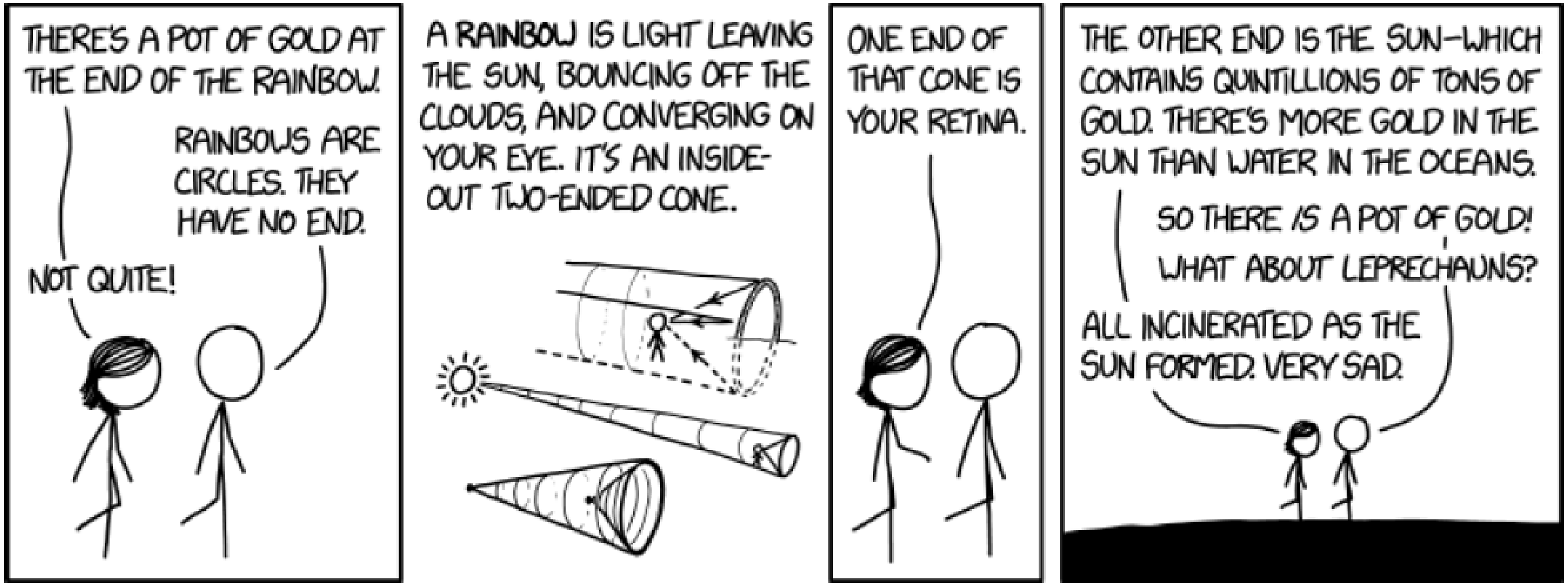
- We often have an operation code, a binary number, with k bits (e.g. 4 bits)
- There are 2^k alternative actions to take, depending on the value of the code

```
case opcode
  0: action
  1: action
  ...
  15: action
```

Control algorithm

```
repeat forever
  if interrupt_request
    then savepc := pc
         pc := 0 ; address of interrupt handler in OS
    else ir := mem[pc], pc := pc + 1                    ; fetch instruction
         case ir_op
         0: reg[ir_d] := reg[ir_a] + reg[ir_b]          ; add
         1: reg[ir_d] := reg[ir_a] - reg[ir_b]          ; sub
         2: reg[ir_d] := reg[ir_a] * reg[ir_b]          ; mul
         ... more RRR instructions are similar
         ...
         15: adr := mem[pc], pc := pc + 1               ; displacement
             adr := adr + reg[ir_a]                     ; effective address
             case ir_b
             0: reg[ir_d] := adr                        ; lea
             1: reg[ir_d] := mem[adr]                   ; load
             2: mem[adr] := reg[ir_d]                   ; store
             3: pc := adr                               ; jump
             ... more RX instructions are similar
             ...
```

the end of the rainbow



https://xkcd.com/1944/