

# IY5512 Computer Security

## Part 0: Preliminaries and introduction to computer systems

Chris Mitchell

[me@chrismitchell.net](mailto:me@chrismitchell.net)

<http://www.chrismitchell.net>

1

This course covers the topic of *Computer Security*.

This preliminary part of the course serves two main purposes:

- it introduces the content and format of the course material;
- it provides a brief introduction to computer systems, the main focus of this course.



## Preliminaries

- This course will be taught by Chris Mitchell.
- Contact details:
  - [me@chrismitchell.net](mailto:me@chrismitchell.net);
  - McCrea room 347;
  - ext. 3423
- If you want to speak to me, it is probably simplest to make an appointment via email.

2

The vast majority of this course will be taught by Chris Mitchell. My contact details are as follows:

[me@chrismitchell.net](mailto:me@chrismitchell.net); McCrea room 347; ext. 3423

Whilst I am happy to see you at any time, if you can find me in my office, I am often busy seeing other students and I only work half-time. So it is probably best to make an appointment via email.

## Outline of course

- The IY5512 course material is divided into the following main topics:
  0. Preliminaries and introduction to computer systems
  1. Introduction to computer security
  2. Design and evaluation
  3. Hardware security
  4. Software security
  5. Identification and authentication
  6. Authorisation
  7. Windows and Unix security

The eight main parts of the course are listed above. Each topic has a separate set of overheads and associated notes. In addition, the course involves a number of sets of coursework, which students should complete and hand in for assessment.



## Recommended texts I

- There are a rapidly growing number of books on Computer Security.
- The main recommended text for this course is:
  - D. Gollmann, *Computer Security*, 2011 (3rd edition).
- Useful background:
  - C. P. Pfleeger and S. L. Pfleeger, *Security in Computing*, 2007 (4th edition).
  - M. Bishop, *Computer Security*, 2003.
- There are many other useful books ...

4

The main book recommended for the course is:

- D. Gollmann, *Computer Security*, John Wiley & Sons, 2011 (3rd edition).

The following are also useful references:

- C. P. Pfleeger and S. L. Pfleeger, *Security in Computing*, Prentice Hall, 2007 (4th edition).
- M. Bishop, *Computer Security*, Addison-Wesley, 2003.

There is an ever-growing number of other books covering similar topics.

## Recommended texts II

- Course requires a basic understanding of how computers and operating systems work.
- If you do not have a background in computing you are **very strongly recommended** to do background reading to familiarise yourself with the basic concepts.
- Useful books include:
  - A. Tanenbaum, *Modern Operating Systems*, Prentice Hall;
  - A. Silberschatz, P.B. Galvin and G. Gagne, *Operating System Concepts*, John Wiley & Sons;
  - W. Stallings, *Operating Systems: Internals and Design Principles*, Prentice Hall.

This course requires a basic understanding of how computers and operating systems work.

If you do not have a background in computing you are **very strongly recommended** to do the necessary background reading to familiarise yourself with the basic concepts of computers and operating systems.

There are large number of textbooks covering this material. Useful books include:

- A. Tanenbaum, *Modern Operating Systems*, Prentice Hall;
- A. Silberschatz, P.B. Galvin and G. Gagne, *Operating System Concepts*, John Wiley & Sons;
- W. Stallings, *Operating Systems: Internals and Design Principles*, Prentice Hall.

There is also a wealth of online material.



## Course material

- Notes on the main parts of the course are available electronically at:  
<http://www.chrismitchell.net/IY5512>
- Other useful resources can be found at the same site.
- Copies of all coursework will be available from the same site. Solutions will also be made available.
- Coursework submission is via Moodle.
- **I only provide printed copies of this preliminary lecture.**

6

Notes on the main parts of the course are available electronically on the following web page:

<http://www.chrismitchell.net/IY5512>


Other useful resources can be found at the same site. All suggestions for additions and corrections to this site will be gratefully received!

Copies of all coursework will be available from the same site. Worked solutions to exercises will be made available in due course (again on the same site).

Please **submit** your coursework for formative feedback via the course Moodle page.

**Please note that, whilst I am providing printed copies of this preliminary lecture, you will have to make your own copies of subsequent parts of the course (if you want a hard copy, that is).**

Information Security Group



# Tutorials I

- Tutorials for IY5512 will take place in McCrea 229 on the following dates:
  - **Full-time students:** in one-hour group sessions on:
    - Tuesday 13th October 2015;
    - Tuesday 10th November 2015;
    - Tuesday 1st December 2015.
  - **Part-time students:** in an hour-long late afternoon session (starting after the lecture at 17:00) on:
    - Monday 12th October 2015;
    - Monday 9th November 2015;
    - Monday 30th November 2015.

7

Tutorials for IY5512 will take place in McCrea 229 on the following dates.


**Full-time students:** in 60-minute group sessions on each of:

- Tuesday 13th October 2015;
- Tuesday 10th November 2015;
- Tuesday 1st December 2015.

**Part-time students:** in a 60-minute late afternoon session (starting after the lecture at 17:00) on each of:

- Monday 12th October 2015;
- Monday 9th November 2015;
- Monday 30th November 2015.

Information Security Group



## Tutorials II

- IY5512 tutorial times for full-time students will be as follows:
  - Group 1: 10:00-11:00
  - Group 2: 11:00-12:00
  - Group 3: 12:00-13:00
  - Group 4: 14:00-15:00
- Please sign up for a tutorial group as soon as possible (arranged by Chez).
- Only come to the session that you sign up for.

8

The times of the IY5512 tutorials for full-time students will be as follows:

- Group 1: 10:00-11:00
- Group 2: 11:00-12:00
- Group 3: 12:00-13:00
- Group 4: 14:00-15:00

You should sign up for one of these tutorial groups, which apply to all first term courses, at the beginning of term (as arranged by Dr Ciechanowicz) and, apart from in exceptional circumstances, you should only come to the session that you sign up for.



## Assessment and feedback

- I will provide a number of sets of coursework during term.
- If feedback required, complete and return to me.
- Course result will be assessed purely on the basis of an examination, to be sat in May 2016.
- Examination will require you to answer three questions from five (examples of old examination papers are available from the library).

9

I plan to provide a number of sets of coursework during the term, one for each part of the course. If feedback is required on solutions, then these should be returned to me via Moodle by the deadline marked on the coursework sheet.

The final mark for this course will be assessed purely on the basis of an examination, to be sat in May 2016. The examination will require you to answer three questions from five (examples of previous years' examination papers are available from the library).

**Important note:** this is the sixth time I have taught this course and set the exam (starting in 2010/11), so do not expect precisely the same style of questions to arise in the examination as were set in the years before 2011.

## Acknowledgements

- Many slides used in this course are based on material previously prepared by Jason Crampton (although the errors are mine).
- His advice and input are gratefully acknowledged, as is the underlying help of Dieter Gollmann, the original teacher of this course.

Many of the slides used in this year's version of the course are based on material previously prepared by Jason Crampton (although any errors are due to me).

His advice and input are gratefully acknowledged, as is the underlying help of Dieter Gollmann, the original teacher of this course.

## Introduction to computer systems

- This preliminary part of the course provides a very brief introduction to computer systems.
- This is a very quick revision of material you need to know ...
- Not a full course (obviously)!

In the remainder of this preliminary part of the course, we provide a very brief introduction to computer systems.

The purpose is to provide you with a very quick reminder of some fundamental topics, understanding of which is necessary to understand the material in this course. This material is NOT intended as a full course – it is just a quick summary intended to highlight areas that you need to understand.

## Objectives

- Main goal is to introduce fundamental computing concepts, covering:
  - computer hardware;
  - software (programs); and
  - operating systems.
- Without a basic understanding of these topics, the rest of the course won't make sense.
- Understanding this material will enable you to:
  - identify security issues;
  - understand why vulnerabilities arise and how they can be addressed.

12

The main goal of this part of the course is to introduce a number of fundamental computing concepts, covering:

- computer hardware;
- computer software (programs); and
- operating systems.

Without a basic understanding of such matters, the rest of the course won't make sense; in particular having a basic understanding of how computers work will enable you to identify security issues, understand why vulnerabilities arise, and see how vulnerabilities can be addressed.

## Work for you ...

- If you have a computing background, then all this preliminary material will be trivial.
- If not, then:
  - this introduction is only a taster ...
  - you really need to do some background reading on computing, e.g. using:
    - the recommended books;
    - any other basic book (there are many);
    - huge range of web resources (e.g. Wikipedia)!

13

If you have a computing background, then you are likely to find all the material in this part of the course somewhat trivial.

If you don't have a computing background, then this introduction can only give you a taste of this material, and you really need to do some background reading on computing, e.g. by using:

- the recommended books;
- any other basic book (of which there are many);
- some of the huge variety of web resources (including Wikipedia!) ...

For example, if you look up 'operating system' under Wikipedia you will get a very nicely written article explaining in greater detail some of the concepts covered in this lecture.

# Agenda

- What do computers do?
- Some common notation
- Computer architectures
- Software
- Operating systems
- Resources

We divide the remainder of this introductory part of the course into the above-listed topics, starting with a general introduction to computers.

# Computers

- A computer is a very simple machine.
- It can:
  - store information; and
  - carry out sequences of simple instructions, specified as a **computer program**.
- Programs are collectively known as **software** (as opposed to **hardware**, i.e. electronics making up the computer).

A computer is, conceptually at least, a very simple machine that can store information, and carry out sequences of simple instructions, specified in the form of a **computer program**.

These programs operate on stored data and **inputs** (e.g. provided by a user) to give **outputs**.

Programs are collectively known as **software** (as opposed to the **hardware**, i.e. the physical electronics making up the computer).

## A simple program I

- Suppose we want to write a list of instructions enabling a child to multiply two numbers together.
- Suppose the child understands English and knows how to add and subtract.

As an example, suppose that we want to write a list of instructions that will enable an intelligent child to multiply two numbers together.

We suppose that the child understands English and already knows how to add and subtract.

## A simple program II

1. Draw a box labelled  $x$  and write the first input in it;
2. Draw a box labelled  $y$  and write the second input in it;
3. Draw a box labelled *result* and write 0 in it;
4. If the contents of  $y$  is 0 go to step 8;
5. Add the contents of box  $x$  to the contents of box *result*;
6. Subtract one from the contents of box  $y$ ;
7. Go to step 4;
8. Finish.

17

One possible sequence of instructions is as follows:

1. Draw a box labelled  $x$  and write the first input in it;
2. Draw a box labelled  $y$  and write the second input in it;
3. Draw a box labelled *result* and write 0 in it;
4. If the contents of  $y$  is 0 go to step 8;
5. Add the contents of box  $x$  to the contents of box *result*;
6. Subtract one from the contents of box  $y$ ;
7. Go to step 4;
8. Finish.

Note that this is just one example of a list of instructions to achieve the desired objective. It is possible to write an unlimited number of different programs that achieve the same result.

## A simple program III

- Example models very closely how computers work.
- Computer memory divided into discrete 'boxes' or 'slots' – used to store working values (numbers or text).
- All computer needs to perform instructions must be stored in one of these boxes.
- These boxes are little like 'pigeonholes', each of which can store a piece of data.

18

Note that this example models very closely how computers work. In particular, computer memory is divided into discrete containers (called **memory locations**), which can be thought of as 'boxes', 'slots', or 'cells'. These memory locations can be used to store individual working values, such as whole numbers, floating point numbers, or characters (letters, digits, etc.) from a piece of text, e.g. for a word processor.

Everything the computer needs to perform a sequence of instructions must be stored in one of these memory locations. As an analogy, think about how individual cells in a spreadsheet work, or imagine them as pigeonholes within which data can be stored.

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

19

We now illustrate the operation of our 'program' using some simple inputs. Try using the program with some other (small) inputs and see whether it works.

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$  6

20

As we run through operation of the program, the results of the operations performed are shown on the left of the slides. The current step is shown in red.

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

 $x$   $y$

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

23

Note that this is a **decision** step. The ability of a computer to execute different sequences of instructions depending on computations it makes (or inputs it receives) enables computers to do very sophisticated tasks using a simple set of instructions.

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

## Running the simple program

- Suppose we want to multiply 6 and 3:
  1. Draw a box labelled  $x$  and write the first input in it;
  2. Draw a box labelled  $y$  and write the second input in it;
  3. Draw a box labelled *result* and write 0 in it;
  4. If the contents of  $y$  is 0 go to step 8;
  5. Add the contents of box  $x$  to the contents of box *result*;
  6. Subtract one from the contents of box  $y$ ;
  7. Go to step 4;
  8. Finish.

$x$    
 $y$    
*result*

36

The program terminates on step 8, and the desired result is contained in box *result*.

## Instructions

- Different computers carry out different types of instruction:
  - by analogy, our simple program is no use to someone who doesn't know how to add or doesn't understand English.
- Computers do not 'understand' programs:
  - by analogy, the person executing the instructions doesn't need to understand the instructions.
- If instructions are wrong, then computer will get the wrong answer. This has implications for security!
- Programs can often work for 'most cases' and only fail occasionally.

37

Different types of computer carry out different types of instruction; by analogy, our simple program is of no use to someone who doesn't know how to add or who doesn't understand English.

Computers do not 'understand' programs in the way that we might understand why a set of instructions is designed the way that it is. By analogy, it is not necessary for the person executing the instructions to understand the rationale for the instructions. There are many everyday analogies for this:

- as children we may have been taught simple techniques for multiplying numbers of more than one digit together – these typically consist of sets of rules which give the right answer, but as children we had no idea *why* they worked;
- knitting patterns consists of complex sequences of instructions which, if followed correctly, result in a garment of the right size and shape – however, the knitter does not need to know why the instructions are put together the way they are.

If the sequence of instructions is wrong (e.g. by poor design or as a result of a typing error) the computer will compute the wrong answer. This has major implications for security! Note especially that programs can often work for 'most cases' and only fail occasionally, meaning that software is often deployed that is faulty, but where the faults are very hard to find from testing.

## Writing a program I

- Next show how our list of instructions can be made into a program.
- Suppose a computer:
  - can accept input from external sources;
  - has a series of sequentially numbered memory **slots** (memory locations);
  - can access any slot and read from and/or update its contents;
  - memory slots used for storing data **and** programs.

38

We next see how a computer program is written, using our simple list of instructions for multiplication as an example.

We suppose that a computer:

- can accept input from external sources;
- has a series of sequentially numbered memory **slots** (memory locations);
- can access any slot and can then read from, and/or update, the contents of this slot;
- the computer memory is used both for storing data and for storing the program (i.e. the sequence of instructions the computer must follow).

## Writing a program II

- How can we write a program so the computer will execute the instructions and compute the product of two numbers?
- Suppose:
  - each slot may contain data or an instruction;
  - each variable ( $x, y, result$ ) is associated with a slot;
  - each instruction says which slots should be accessed;
  - each instruction says what do with the contents of the slots.

39

How can we write a program so that the computer will execute the instructions and compute the product of two numbers?

We make the following underlying assumptions (which model how real world computers operate):

- each slot may contain data or an instruction;
- each variable ( $x, y$  and  $result$  in our simple example) is associated with a slot;
- each instruction says which slots should be accessed;
- each instruction says what do with the contents of the slots.

## Writing a program III

1. Get input 1 and write it to slot 10 (*x* will be stored in slot 10);
2. Get input 2 and write it to slot 11 (*y* will be stored in slot 11);
3. Write 0 in slot 12 (*result* will be stored in slot 12);
4. If the contents of slot 11 is 0 go to instruction at slot 8;
5. Add the contents of slot 10 to the contents of slot 12;
6. Subtract one from the contents of slot 11;
7. Go to instruction at slot 4;
8. Finish;
9. (Not used);
10. (Storage for *x*);
11. (Storage for *y*);
12. (Storage for *result*);

The above program formalises the example program we looked at previously.

- Memory locations (slots) 1–8 contain the text or code section of the program.
- Memory location 9 is not being used.
- Memory locations 10–12 contain the data section of the program, i.e. where working data (variables) are located.

# Requirements I

- Machine instructions manipulate data held in memory, and can:
  - add two data items together;
  - test whether a data item is 0.
- A computer stores a program and its associated data ( $x$ ,  $y$  and *result*, for example) in memory:
  - this is basic idea behind the **von Neumann architecture** on which all modern ('stored-program') computers are based
  - the computer must have storage (or memory).

41

Machine instructions manipulate data held in memory, and can, for example:

- add two data items together;
- test whether a data item is 0.

A computer stores a program and its associated data ( $x$ ,  $y$  and *result*, for example) in memory. This is the basic idea behind the **von Neumann architecture** on which all modern ('stored-program') computers are based. The computer must have storage (or memory) which can be used to store the data and the program.

Note that, because the data and the program are stored in the same type of memory, a malfunctioning program might accidentally over-write part of the program instead of modifying the data. This is exactly the sort of problem we encounter in real life, and such an error can cause major security issues.

## Requirements II

- Computer must be able to keep track of program **execution**, including knowing:
  - which instruction is currently being executed;
  - which memory location is used to store the address of the next machine instruction to be executed.
- Done using a special memory location called the **program counter**.
- Everything has to be stated explicitly – a computer is very dumb!

42

A computer must be able to keep track of the **execution** of a program. In particular at any time it must know:

- which instruction is currently being executed;
- which memory location is used to store the address of the next machine instruction to be executed.

This is achieved using a special memory location called the **program counter**. The contents of the program counter indicate the location of the next instruction to be executed (every memory location has a unique numerical address).

Everything has to be explicitly stated – a computer has no intelligence; it is simply a machine following instructions.

## Going wrong I

- Things can go wrong in many ways!
- Any single, very trivial, error can cause the program to generate incorrect results.
- In our example, what happens if we ask the child to multiply 6 and 3.5 together?
  - What should we do about this possibility?
- This is a simple example of where the program is given inputs different to those expected by the program author.

43

Things can go wrong in many ways! Any single, very trivial, error can cause the program to generate incorrect results.

In our example, what happens if we ask the child to multiply 6 and 3.5 together? What should we do about this possibility?

This is an example of unexpected input to a program – i.e. the program is designed to cope with inputs equal to whole numbers, and may fail in unexpected ways if given something other than a whole number.

## Going wrong II

- What happens if the child can only count up to 1000 and we ask him/her to multiply 41 and 27 together?
  - What should we do about this possibility?
- What happens if we ask the child to multiply 6 and “ten” together?
  - What should we do about this possibility?
- Does this have implications for computer security? [Yes, of course it does!]

44

What happens if the child can only count up to 1000 and we ask him/her to multiply 41 and 27 together? What should we do about this possibility?

What happens if we ask the child to multiply 6 and “ten” together? What should we do about this possibility?

Does this have implications for computer security? [Yes, of course it does!]

This is another example of unexpected inputs to a program. This is a major cause of real world security problems – where the designer of a program doesn't think about what would happen if the program is given inputs other than those the designer expects it to be given, and doesn't include measures in the program to deal with such cases.

# Algorithms

- Programs implement *algorithms*.
- An algorithm is a list of precise instructions (steps) for calculating a function.
- Word derived from name Al-Khwarizmi (meaning 'native of Khwarazm'), a Persian Mathematician whose work introduced sophisticated mathematics to Europe.
- He wrote a treatise in Arabic in 825CE entitled '*On Calculation with Hindu Numerals*'.

Programs implement *algorithms*, where an algorithm is a list of precise instructions (steps) for calculating a function.

The term 'algorithm' is derived from the name Al-Khwarizmi (meaning 'native of Khwarazm'), a Persian Mathematician whose work introduced sophisticated mathematics to Europe. He wrote a treatise in Arabic in 825CE entitled '*On Calculation with Hindu Numerals*'.

# Agenda

- What do computers do?
- Some common notation
- Computer architectures
- Software
- Operating systems
- Resources

We next introduce some common mathematical notation used when describing computers and their operation. If this is all unfamiliar to you, then please spend some time reading around and familiarising yourself with these ideas.

## Bits

- Computers store data in form of sequences of ones and zeros, called **bits** (derived from **binary digits**).
- That is, a bit is either a 0 or a 1.
- Bit is the smallest unit of information.
- 1 or 0 can be thought of as 'yes or no', 'on or off', or 'true or false'.

Computers store and process data in the form of sequences of ones and zeros, where each one or zero is called a **bit** (the term is derived from **binary digit**). That is, a bit is either a 0 (zero) or a 1 (one). A bit is the smallest unit of information.

The 1 or 0 can be thought of as 'yes or no', 'on or off', or 'true or false', depending on the context.

## Bytes

- In computers, bits typically organised in groups of eight, where a sequence of eight bits is known as a **byte** (a byte has  $256=2^8$  possible values).
- Half a byte (4 bits) is sometimes called a **nibble** (Computer Scientists are fond of bad puns).
- A **kilobyte (kbyte)** ought to be 1000 bytes, but is actually 1024 bytes, because  $2^{10}=1024$ .
- Similarly a **megabyte (Mbyte)**, **gigabyte (gbyte)** and **terabyte (tByte)** are equal to  $2^{20}$ ,  $2^{30}$  and  $2^{40}$  bytes, respectively.

48

In computers, bits are typically organised in groups of eight, where a sequence of eight bits is usually known as a **byte** (a byte has  $256=2^8$  possible values). Sometimes a byte is referred to as an **octet**.

Half a byte (4 bits) is sometimes called a **nibble** (many Computer Scientists are fond of bad puns).

A **kilobyte (kbyte)** ought to be 1000 bytes, but is actually 1024 bytes, because  $2^{10}=1024$ .

Similarly a **megabyte (Mbyte)**, **gigabyte (gbyte)** and **terabyte (tByte)** are equal to  $2^{20}$ ,  $2^{30}$  and  $2^{40}$  bytes, respectively.

## Decimal representations I

- Usually write numbers in *decimal representation*.
- That is, the number 123 represents:
 
$$1 \times 100 + 2 \times 10 + 3 \times 1$$
- Positions of the digits give the number of units, tens, hundreds, thousands, and so on.
- *Place-value notation* (or *positional notation*) was invented in India; reached Europe via Islamic world – why we call numbers **Arabic numerals**.
- System has huge practical advantages compared to Roman numerals.

49

We normally write numbers using something known as a *decimal representation*.  
That is, the number 123 represents:

$$1 \times 100 + 2 \times 10 + 3 \times 1$$

The positions of the digits indicate the number of units, tens, hundreds, thousands, and so on.

This idea of *place-value notation* (or *positional notation*) was invented in India around 1500 years ago, and reached Europe via the Islamic world, which is why we refer to our numerals as Arabic numerals.

This system has huge practical advantages for computation by comparison with Roman numerals, the system which was in use in Europe prior to the arrival of the positional notation.

## Decimal representations II

- Value assigned to each digit corresponds to a power of 10.
- Rightmost position (**units**) corresponds to  $10^0=1$ , the next position to **tens** ( $10^1$ ), the next to **hundreds** ( $10^2$ ), then **thousands** ( $10^3$ ), and so on.

The value assigned to each digit in a decimal representation corresponds to a power of 10.

That is, the rightmost position (the **units**) corresponds to  $10^0=1$ , the next position to **tens** ( $10^1$ ), the next to **hundreds** ( $10^2$ ), then **thousands** ( $10^3$ ), and so on.



## Binary representations

- **Binary** is similar system where only 0 and 1 are used; each digit position corresponds to a power of 2 (i.e. 1, 2, 4, 8, 16, 32, 64, 128, and so on).

- For example, in binary:

101101

represents the (decimal) number 45.

- This is because:

$$45 = 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1.$$

- Computers represent numbers using binary (since everything is stored as bits).

51

We can devise a similar system (called **binary**) where only the digits 0 and 1 are used, and where each digit position corresponds to a power of 2 (i.e. 1, 2, 4, 8, 16, 32, 64, 128, and so on).

For example, in binary 101101 represents the (decimal) number 45.

This is because:

$$45 = 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1.$$

Computers represent numbers using binary, since everything inside the computer is stored as a sequence of bits.

## Hexadecimal representations

- Binary numbers may contain a lot of digits:
  - binary representation of 2508 is 1001 1100 1100 and requires 12 bits to write down.
- **Hexadecimal (hex)** is a base 16 number representation
  - decimal is base 10;
  - binary is base 2.
- Four binary digits can represent 16 values
  - hex digit can represent four binary digits;
  - for convenience often use hex rather than binary (all computers actually 'think' in binary).

52

Binary representations of numbers may contain a lot of digits, and hence writing numbers in binary is rather laborious. For example, the binary representation of 2508 is 1001 1100 1100, i.e. requires 12 bits to write down.

**Hexadecimal (hex)** is a base 16 number representation. By contrast decimal is base 10 and binary is base 2.

Four binary digits can represent 16 values and a hex digit can represent four binary digits. For convenience we often use hex rather than binary to write numbers as stored in a computer (remembering, of course, that all computers actually operate in binary).

## Notation

- To distinguish between decimal and hexadecimal, prefix hex numbers with 0x:
  - some textbooks instead use the suffix H to denote a hexadecimal number;
  - 0x10 and 10H both represent the decimal number 16.
- 2508 is 100111001100 in binary representation
  - to convert to hex we write it as 1001 1100 1100,
  - and convert each group to a hex digit to get 0x9CC.

In order to distinguish between decimal and hexadecimal, it is common to prefix a hexadecimal number with the string 0x:

- alternatively, some textbooks use the suffix H to denote a hexadecimal number;
- for example, 0x10 and 10H both represent the decimal number 16.


We know (from a previous slide) that 2508 is 100111001100 in binary representation. To convert to hex we write it as 1001 1100 1100, and then convert each group of four bits to a hex digit to get the hex representation 0x9CC.

# Agenda

- What do computers do?
- Some common notation
- Computer architectures
- Software
- Operating systems
- Resources

So far we have focussed on what computer systems can do. We next consider how they are put together.

Information Security Group



## What is a computer system?

- Computer system has three main components:
  - **Hardware:**
    - stores machine instructions (compiled programs) in main memory;
    - executes machine instructions;
  - **Operating system (OS):**
    - collection of computer programs;
    - provides interface between user and hardware;
    - provides interface between users and resources such as the file system and print services;
  - **Application programs (apps):**
    - provide specialised computing tools for the end user;
    - e.g. word processors, web browsers, database management systems, etc.

55

A computer system has three main components, as follows.

- The **Hardware**:
  - stores machine instructions (compiled programs) in main memory; and
  - executes machine instructions.
- The **Operating system** (the **OS**) is a collection of computer programs that:
  - provides an interface between a user and the hardware; and
  - provides an interface between the users and the system resources, such as the file system and print services.
- The **Application programs** (or **apps**):
  - provide specialised computing tools for the end user;
  - include word processors, web browsers, database management systems, games, etc.

## What is a computer?

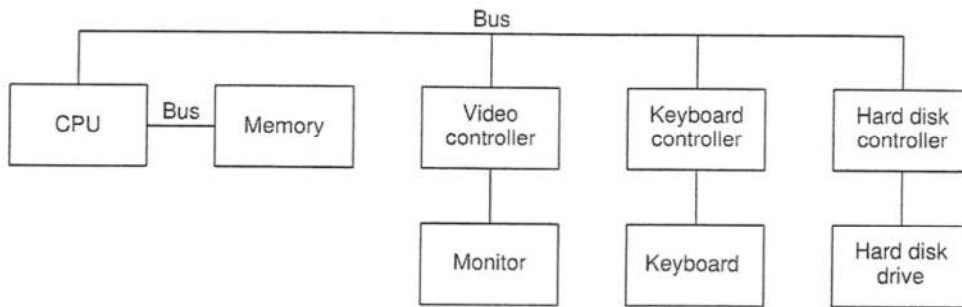
- Computer hardware has four main elements:
  - **processor** or **central processing unit (CPU)** – controls operation of the computer;
  - **main memory** – stores programs for execution and their associated data;
  - **input/output (I/O) controllers** – interfaces to peripherals (monitors, hard disks, etc.);
  - **system bus** – enables communications between other components.

56

An individual computer (i.e. the hardware component of a computer system) consists of four main elements:

- a **processor** or **central processing unit (CPU)** – controls the operation of the computer and actually performs the computations;
- the **main memory** – stores programs for execution and the data used by the program, i.e. data that the program reads to perform its calculations and writes as its output;
- the **input/output (I/O) controllers** – provide interfaces to peripheral devices (monitors, hard disks, printers, etc.);
- the **system bus** – provides a means of communication between the other components.

# The main components



This diagram shows a simplified version of the relationships between some of the main components of a computer.

## The processor

- Two main components:
  - **control unit** – performs the instruction cycle, i.e. it decides what happens next;
  - **arithmetic logic unit (ALU)** – responsible for performing arithmetic and boolean (logical) operations.
- Has a defined set of instructions which it can process.

58

A processor (CPU) incorporates two main components:

- the **control unit** – that is responsible for performing the instruction cycle, i.e. determining what should be done next;
- the **arithmetic logic unit (ALU)** – that is responsible for performing arithmetic and boolean (logical) operations, i.e. actually performing the computations.

Every processor has an associated **instruction set**, which is the collection of instructions that the processor can process. This instruction set is specific to the type of processor.

## Registers I

- Special memory locations reserved for very particular purposes:
  - very limited capacity (less than 100 bits);
  - extremely quick to access;
  - very limited in number (less than 40).
- Types of registers:
  - data registers;
  - index registers;
  - control registers.

59

Every processor has a number of special storage locations called **registers**. Registers are reserved for very particular purposes. They:

- have very limited capacity (they have less than 100 bits each);
- can be accessed very quickly by the processor;
- are very limited in number (typically there are less than 40 of them).

Types of registers include:

- data registers;
- index registers; and
- control registers.

The exact nature of the set of registers will depend on the particular type of processor.

## Registers II

- Uses of registers include:
  - program counter (instruction pointer);
  - memory management;
  - stack management;
  - storage of intermediate results in calculations.

Within a computer, registers are used for a range of purposes, including:

- the program counter (instruction pointer), which points to the memory location containing the instruction currently being executed;
- memory management (i.e. deciding which portions of the computer memory are used for which programs);
- stack management (where the stack is a special area of memory reserved for temporary storage);
- storage for intermediate results in calculations.

## Machine instructions I

- A computer program is a sequence of instructions that can be understood by a processor.
- Different processors have different instruction sets.
- A program can only run on a particular type of processor.

As we have previously outlined, a computer program is a sequence of machine instructions that can be understood by a processor.

Again as mentioned previously, different processors have different instruction sets. As a result, any individual program can only run on a particular type of processor.

## Machine instructions II

- A machine instruction specifies:
  - the operation to be executed (the **op code**);
  - the location(s) of the operand(s) in memory – numbers of memory locations referred to as **addresses**.
- For example, an instruction might specify:

Add (the contents of memory address)  $x$  to (the contents of memory address)  $y$

62

A machine instruction specifies:

- the operation to be executed (the **op code**);
- the location(s) of the **operand(s)** in memory (where the operands are the values used as inputs to the operation). It is important to note that the numbers of memory locations are referred to as **addresses**.

For example, an instruction might specify:

Add (the contents of memory address)  $x$  to (the contents of memory address)  $y$

## Examples of instructions

- Typical examples of machine instructions include:
  - Transfer data to a register from a specified location;
  - Transfer data from a register to a specified location;
  - Jump (transfer execution) to a specified location;
  - Perform an arithmetic or logical operation on one or more values at specified locations;
  - Transfer data to/from an I/O device;
  - Change contents of a control register.

63

Typical examples of machine instructions include:

- Transfer data to a register from a specified location in computer memory (recalling that each memory location has a unique address);
- Transfer data from a register to a specified location in computer memory;
- Jump (transfer execution) to a specified location in computer memory, i.e. modify the program counter, which determines which instruction will be performed next;
- Perform an arithmetic or logical operation on one or more values at specified locations, and store the result somewhere, e.g. in a specific register;
- Transfer data to or from a specified input/output (I/O) device;
- Change the contents of a control register.

## Programming languages I

- Fundamentally, a program is a sequence of machine instructions.
- This is what is executed on a machine.
- Can write programs using machine instructions, but it is very difficult and time consuming.
- In practice, programs written in **high-level programming languages**: automatically translated to machine instructions.

64

Up to now we have talked about a program as consisting of a sequence of machine instructions. This is correct in the sense that this is what is executed on a machine.

However, whilst it is possible to write programs directly using machine instructions, it is very difficult and time consuming.

In practice, programs are written using **high-level programming languages**, which are subsequently automatically translated into sequences of machine instructions which can then be executed by the processor.

## Programming languages II

- High-level programming languages (e.g. C, C++, Java, Pascal, Basic) have many advantages:
  - easier to use;
  - errors less likely since programmer can focus on purpose of the program rather than details of instruction set;
  - machine independent (aiding portability).

65

These high-level programming languages (e.g. C, C++, Java, Pascal, Basic) have many advantages:

- they are much easier to use;
- errors are less likely since the programmer can focus on the purpose of the program rather than the technicalities of the instruction set;
- they are, to a greater or lesser extent, machine independent, thereby aiding portability.

# Compilers

- Programs which convert programs in a high-level language into sequences of machine instructions known as **Compilers**.
- Compiler can include checks for poor programming practices (likely to lead to security issues).
- Compilers have settings (e.g. to include debug code).

66

Programs which can convert programs written in a high-level language into sequences of machine instructions are known as **Compilers**.

A compiler can include checks for poor programming practices (e.g. those which are likely to lead to security issues).

Compilers can have various settings, e.g. so that when developing a new program, the compiler includes lots of extra checks to help in **debugging**, i.e. the process of removing errors from a program while it is being written.

## The instruction cycle I

- Machine instructions stored in sequential computer memory locations:
  - the **program counter** is a register containing address of the next machine instruction to be executed;
  - the program counter is incremented during each instruction cycle;
  - a **jump** in execution is simply a way of modifying this counter.

67

The machine instructions are stored in sequential locations in computer memory:

- the **program counter** is a register that contains the address of the next machine instruction to be executed;
- the program counter is incremented during each instruction cycle;
- a **jump** in execution is simply a way of modifying this counter, so that execution transfers from one part of a program to another part.

## The instruction cycle II

- The **control unit** is responsible for performing the **instruction cycle**:
  - **fetches** an instruction from the memory location specified by the program counter and stores it in the **instruction register**;
  - **decodes** the instruction to determine what action the CPU should take;
  - **executes** the instruction.

68

The **control unit** is responsible for performing the **instruction cycle**:

- it **fetches** an instruction from the memory location (address) specified by the program counter and stores it in the **instruction register**;
- it **decodes** the instruction to determine what action the CPU should take;
- it **executes** the instruction.

## Security issues I

- Changes to the instruction register must be strictly controlled:
  - if attacker can change contents of this register, can choose which instruction is executed.
- Typically, changes to control registers (and other *privileged* operations) can only be made by the operating system (a trusted party).

69

Changes to the instruction register must be strictly controlled. If a malicious party is able to change the contents of this register, then he/she can determine which instruction is executed next.

Typically, things are arranged so that changes to control registers (and other **privileged** operations) can only be made by the operating system, i.e. by a trusted entity.

## Security issues II

- Similarly, changes to the program counter must be strictly controlled:
  - an attacker that can change this register can decide which instructions are executed;
  - protecting memory locations (registers and main memory) that control which machine instructions are executed is an essential part of computer security.

70

In a similar way, changes to the program counter must be strictly controlled:

- an attacker that is able to change the contents of this register can decide which machine instructions are executed;
- protecting memory locations (registers and main memory) that control which machine instructions are executed is an essential part of computer security.

# Agenda

- What do computers do?
- Some common notation
- Computer architectures
- Software
- Operating systems
- Resources

## Computer programs

- To operate, computers run programs.
- Many vulnerabilities caused by errors in programs, including:
  - inadequate validation of inputs;
  - incorrect program logic.
- So to understand computer security and vulnerabilities, must understand how programs operate.

72

As we have seen, in order to perform tasks, computers run programs.

Many vulnerabilities, i.e. weaknesses in a system that can be exploited to attack the system, are caused by errors in programs, including:

- inadequate validation of inputs;
- incorrect program logic.

In order to fully understand computer security and the vulnerabilities in systems that can give rise to attacks on systems, it is thus necessary to understand how programs operate, at least at a high level.

## Programming languages

- Programs not normally written using machine instruction set.
- Instead written using a high-level language.
- Many types of programming language, and many ways of converting programs to machine instructions.
- Two particularly important approaches: **compiled** and **interpreted** languages.

73

As mentioned previously, although computers have a defined instruction set, programs are not normally written using this instruction set.

They are instead written using a high-level language.

There are many types of programming language, and many ways of converting programs in these high-level languages to machine instructions.

We briefly review two particularly important approaches, namely **compiled** and **interpreted** languages.

## Compiled programs I

- Compilers typically used with so called *imperative* or *procedural* programming languages such as C and Pascal.
- Very widely used.
- Firstly the program written to obtain the **source code**. [Software of all types is often referred to as **code**].
- Cannot be executed by a computer.

74

Compilers are typically used with so called *imperative* or *procedural* programming languages, such as C, C++ and Pascal.

Such languages are very widely used.

Firstly the program is written to produce the **source code**. Software of all types is often referred to simply as **code**.

This source code cannot be executed directly on a computer.

## Compiled programs II

1. Compile source code (using platform-specific compiler) into **object code**:
  - generates machine instructions and memory;
  - references for machine instructions;
  - may involve linking and compiling external functions provided by standard libraries.
2. Load object code into main memory:
  - may involve linking to external functions (e.g. in form of .dll or .so files).
3. Execute program.

75

There are three main steps in the use of compiled programs.

1. Compile the source code (using a platform-specific compiler) into **object code**:
  - compiler generates machine instructions and memory references for machine instructions;
  - may involve linking and compiling external functions provided by standard libraries.
2. Load the object code (the program) into the computer's main memory:
  - this may involve **linking to external functions**, e.g. provided by the operating system, e.g. in the form of .dll (dynamic linked library) or .so files.
3. Execute the program.

## Interpreted programs I

- **Interpreters** typically used with programs written in *declarative* (including *functional* and *logic*) programming languages such as Haskell or Prolog.
- As with compilers, first the program is written to produce the **source code**.

76

**Interpreters** are typically used with programs written in *declarative* (including *functional* and *logic*) programming languages such as Haskell or Prolog.

As with the use of compilers, first the program is written to produce the **source code**. It is what happens next that differentiates interpreted from compiled code

...

## Interpreted programs II

- Program then executed using an interpreter, a program that 'runs' (interprets) the instructions contained in the source code.
- Interpreter can be thought of as a **virtual machine** that provides an interface between the source code and the hardware.
- Some languages, notably **Java**, are part compiled (into Java **bytecode**) and part interpreted (by a Java virtual machine).

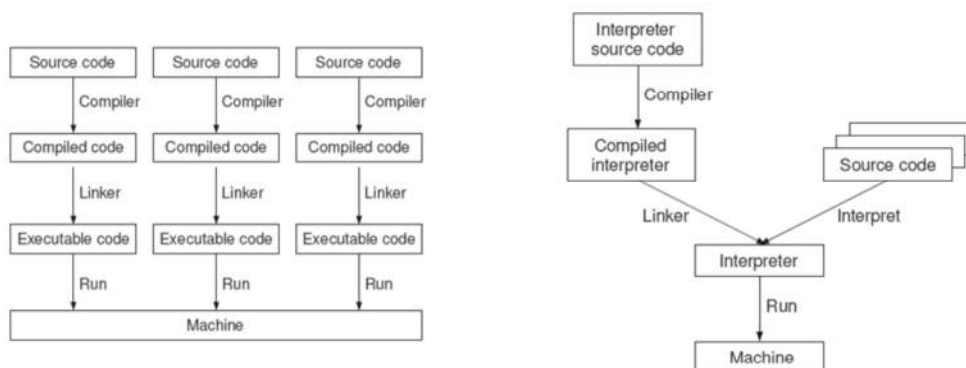
77

The program is then immediately executed using an interpreter, a program that 'runs' (interprets) the instructions contained in the source code. That is, the instructions in the high level language are translated into machine instructions 'on the fly', instead of in advance.

An interpreter can be thought of as a **virtual machine** that provides an interface between the source code and the hardware. This virtual machine is an imaginary computer that can directly operate on the commands of the high level programming language.

Some languages, notably **Java**, are part compiled (into Java **bytecode**) and part interpreted (by the Java virtual machine).

# Creating programs



The diagram shows the two ways in which high-level language programs are converted to machine instructions. The compiler approach is shown on the left, and the interpreter approach on the right.

## Functions: modular code

- Programs typically written in small blocks:
  - each block performs well-defined, specific task;
  - blocks are called **functions** (or subroutines, procedures, modules or classes, depending on the programming language).
- Decomposing programs into smaller functions has two main advantages:
  - functions can be written once and re-used in many different programs;
  - programs are easier to maintain.

79

Computer programs are typically written in small blocks:

- each block performs a well-defined, specific task;
- blocks are called **functions** (or subroutines, procedures, modules or classes, depending on the programming language).

Decomposing programs into smaller functions has two main advantages:

- functions can be written once and re-used in many different programs;
- the resulting programs are easier to maintain, since it is easier to understand, test and maintain a small self-contained piece of code which performs a specific function than a large monolithic program.

Such a **modular approach** is deemed good practice in **software engineering** (the name for the discipline of writing programs).

## Modular code: a simple example

- Suppose want a program that exponentiates (raises one integer to the power of another):
  1. Draw a box labelled  $z$  and write the first input in it
  2. Draw a box labelled  $w$  and write the second input in it
  3. Write a box labelled *result1* and write 1 in it
  4. If the contents of  $w$  is 0 go to step 8
  5. **multiply** the contents of box  $z$  with the contents of box *result1*
  6. Subtract one from the contents of box  $w$
  7. Go to step 4
  8. Finish
- To implement **multiply**, can re-use the instructions previously written for multiplying two numbers together.

80

Suppose that you want to develop a program that exponentiates integers (i.e. that raises one integer to the power of another integer). The following program is a possible way of performing such a computation.

1. Draw a box labelled  $z$  and write the first input in it
2. Draw a box labelled  $w$  and write the second input in it
3. Write a box labelled *result1* and write 1 in it
4. If the contents of  $w$  is 0 go to step 8
5. **multiply** the contents of box  $z$  with the contents of box *result1*
6. Subtract one from the contents of box  $w$
7. Go to step 4
8. Finish

The program works by multiplying the value inserted in  $z$  by itself over and over again (the number of times this happens is set by the value inserted in  $w$ ).

To implement the **multiply** instruction (in step 5), we can re-use the instructions previously written for multiplying two numbers together.

## Modular code: implications

- Developer of exponentiate program does not need to develop code for multiplication.
- Can re-use a function developed previously.
- The multiply code and the exponentiate code can then be tested and debugged independently of each other.

The developer (author) of the exponentiate program does not need to develop his or her own code for multiplication, and can instead re-use the multiply function developed previously.

The multiply code and the exponentiate code can then be tested and debugged independently of each other.

## Variable types

- Variables are used in high-level programs to store working data.
- Can be given memorable names.
- Variables can be used to store various types of information, e.g. integers, strings of characters, floating point numbers, ...
- In a high-level language program, the **type** of a variable is used to define what data it stores.

82

When writing computer programs in a high-level language we use **variables** to store working data. These variables can be given memorable names which reflect their purpose.

Variables can be used to store various types of information, e.g. integers, strings of characters, floating point numbers, ...

In a high-level language program, the **type** of a variable is used to define the nature of the data stored in that variable.

When a high-level language program is translated into a sequence of machine instructions, the variable names are converted into numerical memory addresses.

## Function signatures

- Programmer using a function does not need to know how it is implemented; only needs to know:
  - what inputs to supply and the outputs that the function will **return**.
- The **signature** (or **type**) of a function specifies the types of its inputs and outputs.
- The signature `int power(int a, int n)` defines a function **power** that takes two integers as inputs and returns an integer.

83

A computer programmer using a pre-written function does not need to know how the function is implemented. The programmer only needs to know what inputs to supply to a function, and the outputs that the function will **return**. That is, the programmer can treat a function as a 'back box', with known inputs and outputs.

The **signature** (or **type**) of a function specifies the types of its inputs and outputs. The signature `int power(int a, int n)` defines a function **power** that takes two integers as inputs and returns an integer.

## Example – a simple function

- Function `power` takes two integers as inputs (integers  $a$  and  $n$ ) and returns an integer (stored in the variable `result`):

```
function power(int  $a$ , int  $n$ )  
  int  $result = 1$ ;  
  for  $i = 1$  to  $n$  do  
     $result = result * a$ ;  
  end for  
  return  $result$ ;  
end function
```

84

The `power` function takes two integers as inputs (integers  $a$  and  $n$ ) and returns an integer (stored in the variable `result`):

```
function power(int  $a$ , int  $n$ )  
  int  $result = 1$ ;  
  for  $i = 1$  to  $n$  do  
     $result = result * a$ ;  
  end for  
  return  $result$ ;  
end function
```

In this function there is a 'for loop', terminated by the 'end for' statement. The syntax shown simply means execute the instructions in the loop a total of  $n$  times – firstly with  $i=1$ , secondly with  $i=2$ , and so on, until the last time when  $i=n$ .

The function computes the exponential  $a^n$ , i.e.  $a$  raised to the power  $n$ .

## Using a function

- Function `main` calls `power` using inputs `x` and `y` and assigns the result to variable `x`:

```
function main
int x = 3;
int y = 5;
x = power(x, y);
print(x);
return 0;
end function
```

85

The program `main` (which is itself a function) calls the function `power` using inputs `x` and `y`, and assigns the result to the variable `x`:

```
function main
int x = 3;
int y = 5;
x = power(x, y);
print(x);
return 0;
end function
```

Note that the 'return 0' line is just a programmatic convenience. It doesn't mean anything, since no function calls `main` – it is the 'top level' function.

## Shared libraries

- Shared library is language-specific set of functions giving 'toolbox' for programmer, e.g. containing:
  - string-handling functions;
  - mathematical functions;
  - drawing and windowing functions (GUI design);
  - cryptographic functions.
- Saves developers from 'reinventing the wheel'.

86

A shared library is a language-specific suite of functions that provides a 'toolbox' for programmers. The functions in such a library will be designed to do things that are needed by many programs. Examples of types of functions in such a library include:

- string-handling functions;
- mathematical functions;
- drawing and windowing functions (for Graphical User Interface (GUI) design);
- cryptographic functions.

Such a library saves developers from 'reinventing the wheel'. That is, a developer can write a program which calls these functions, without having to write new code to implement them.

## Application program interfaces (APIs)

- Application program interface (API) is a collection of function signatures:
  - implementation of functions in the library invisible to application programmer;
  - programmer doesn't need to know how the functions work, only what inputs they expect and what outputs they produce.

87

An application program interface (API) is a collection of function signatures:

- the implementation of the functions in the library is typically invisible to the application programmer;
- the programmer doesn't need to know how the functions work, only what inputs they expect and what outputs they produce.

## APIs – other uses

- APIs have other uses.
- An API can be used to give access to 'lower level' functionality to the programmer, e.g.:
  - operating system APIs enable application programmers to use kernel-level functions which they do not have full access to;
  - often the source code for the OS kernel is not available.
- API can give access to sensitive functions in a controlled way.

88

Apart from giving access to 'useful functions, APIs have other uses. An API is often used to 'expose', i.e. give access to, so called lower level functionality to the programmer. For example:

- operating system APIs enable application programmers to use kernel-level functions, to which they do not have full access (since they are very powerful);
- often the source code for the OS kernel is not available.

That is, an API can give access to sensitive functions in a controlled way.

## An example API

- A cryptographic library might include an `encryptMessage` function:
  - inputs are a message (variable type string), the name of a cryptographic algorithm (variable type string), and a key (variable type integer);
  - output is ciphertext (variable type string);
  - programmer can write `c = encryptMessage(m, 'DES', k)`, and avoid writing a program to implement DES;
  - need to trust implementer of crypto API to write DES program correctly!

89

A cryptographic library might include an `encryptMessage` function:

- the inputs are a message (variable type string), the name of a cryptographic algorithm (variable type string), and a key (variable type integer);
- the output is the ciphertext (variable type string);
- a programmer can write `c = encryptMessage(m, 'DES', k)`, and avoid writing a program to implement DES;

The user of a library needs to trust the implementer of the crypto API to write the DES program correctly!

## Data structures

- **Data structure** is a particular way of storing and organising data in a computer (a tool of a high level programming language):
  - used in almost every program or software system;
  - essential components of many algorithms, and enable efficient management of huge amounts of data;
  - examples include arrays, stacks, linked lists, trees, ...
- Enable programs to be written which operate on data structures as components, rather than simply on single data elements.

90

A **data structure** is a particular way of storing and organising data in a computer (a tool of a high level programming language). Data structures are used in almost every program or software system. They are essential components of many algorithms, and they make possible to manage and process large volumes of data in efficient and simple ways.

Examples of data structures include arrays, stacks, linked lists, trees, heaps, etc.

The use of data structures enables programs to be written which operate on the data structures as components, rather than simply on single data elements. This can greatly simplify and clarify the task of the programmer.

## Data structures – stacks

- A **stack** is a last in, first out (LIFO) data structure, analogous to a stack of documents on a desk.
- A stack has two fundamental operations:  
**push** and **pop**:
  - push operation adds an item to the top of the list, or initialises the stack if it is empty;
  - pop operation removes an item from the top of list, and returns this value to the caller.

91

A **stack** is a last in, first out (LIFO) data structure, analogous to a stack of documents on a desk.

A stack has two fundamental operations associated with it: **push** and **pop**:

- the push operation adds an item to the top of the list, or initialises the stack if it is empty;
- the pop operation removes an item from the top of the list, and returns this value to the caller.

## Data structures – heaps

- A **heap** is a tree-based data structure, in which every node has a **key**.
- It satisfies the **heap property**:
  - if  $B$  is a child node of  $A$ , then  $\text{key}(A) \geq \text{key}(B)$ ;
- This implies that an element with the greatest key is always in the root node.

92

A **heap** is a tree-based data structure, in which every node in the tree has an associated **key**.

It satisfies the **heap property**:

- if  $B$  is a child node of  $A$ , then  $\text{key}(A) \geq \text{key}(B)$ ;

This implies that an element with the greatest key is always in the root node.

Note that a tree is a special type of graph. A graph consists of a set of nodes (vertices) and a set of edges between defined pairs of nodes. A tree has the property that there are no circuits in the graph.

# Agenda

- What do computers do?
- Some common notation
- Computer architectures
- Software
- Operating systems
- Resources

## What is an operating system?

- From perspective of an application program, operating system (OS) acts as a virtual machine on which application runs:
  - a computer has a very limited set of instructions;
  - the operating system extends the set of instructions that are available to programmers;
  - the operating system hides the details of the particular hardware on which the operating system runs.
- Note that description is loosely Unix based – but Windows is quite similar.

94

From the perspective of an application program, we can think of the operating system (OS) as an extended or virtual machine:

- a computer has a very limited set of instructions;
- the operating system extends the set of instructions that are available to programmers;
- the operating system hides the details of the particular hardware on which the operating system runs.

Note that the descriptions in the next few slides are loosely based on Unix; however, at least at this high level, Windows is quite similar.

## Resource management

- Operating system can also be thought of as a **resource manager**, managing access to and protecting:
  - the computer hardware;
  - services and programs;
  - data.

The operating system can also be thought of as a **resource manager**, that manages access to and protects:

- the computer hardware;
- a set of services and programs;
- stored data.

It also typically provides a user interface (e.g. in the form of a **shell** or **desktop**), so that a human user can choose which applications run, manage files, install and remove applications, etc.

## Hardware management

- Operating system manages interaction between the computer and peripheral devices.
- Each device has a **device controller** attached to bus:
  - device controller contains a **buffer** (temporary storage for data) and control registers.
- **Device driver** is a piece of software that:
  - communicates with the device controller;
  - enables data to be transferred between buffer and main memory.
- Device drivers are device and OS-specific.

96

An operating system manages the interaction between the computer and peripheral devices.

Each device has an associated **device controller** attached to the system bus:

- the device controller contains a buffer (temporary storage for data) and control registers.

A **device driver** is a piece of software that:

- communicates with the device controller; and
- provides functions to write data to the buffer from main memory and read data from the buffer and write it to main memory.

Device drivers are device and OS-specific.

# Interrupts

- Peripheral devices communicate with computer using **interrupts**.
- **Interrupt vector table (IVT)** stores each interrupt vector along with the location of program for dealing with interrupt:
  - IVT is stored in main memory;
  - IVT controls the flow of execution.
- The IVT is an attractive target for an attacker:
  - if attacker can change IVT, can execute any code;
  - IVT must only be changed by OS (or other trusted entities).

97

Peripheral devices communicate with the computer by means of special processor functions known as **interrupts**. As the name suggests, interrupts interrupt (stop) the current flow of execution of a program, and cause a different program to run. Each type of interrupt has a unique interrupt vector, i.e. a numerical value which indicates the type of interrupt

The **interrupt vector table (IVT)** stores each interrupt vector along with the location of the program for dealing with the interrupt:

- the IVT is stored in main memory (in an area reserved for use by the OS);
- the IVT controls the flow of execution.

The IVT is an attractive target for an attacker:

- if an attacker can change the IVT he can execute code of his choosing;
- the IVT must only be changed by the operating system (or other trusted entities).

## Multi-tasking

- Modern processors support **multi-tasking**:
  - memory and processor time are shared between multiple programs to maximise the use of CPU;
  - for example, when one program is waiting for I/O, another program uses the CPU;
  - no more than one program is executed by the CPU at any moment in time.

98

All modern processors support **multi-tasking**:

- memory and processor time are shared between multiple programs to maximise the use of CPU processing cycles;
- for example, when one program is waiting for I/O, another program can use the CPU;

Although it may appear as if multiple processes are operating in parallel at the same time, in fact no more than one program is executed by the CPU at any moment in time. [This situation changes in modern **multi-core** processors which have more than one CPU – one processor chip can contain many CPUs].

# Processes I

- Concept of **process** (or **task**) is fundamental to multi-tasking computers:
  - a process encapsulates all information required to run a program;
  - OS uses processes to manage execution of multiple programs on single CPU.

The concept of a **process** (or **task**) is fundamental to multi-tasking computers:

- a process encapsulates all the information required to run a program;
- the OS uses processes to manage the execution of multiple programs on the same CPU.

## Processes II

- Process defined by:
  - machine instructions (**text section**);
  - the contents of the CPU registers;
  - any global variables (**data section**);
  - contents of the process's stack;
  - contents of the process's heap;
  - pointers to resources (such as files) being used by the process.

100

A process is defined by:

- the machine instructions (the **text section**);
- the contents of the CPU registers;
- any global variables (the **data section**);
- the contents of the process-specific stack;
- the contents of the process-specific heap;
- pointers to resources (such as files) that are currently being used by the process.

A **call stack** (also known as an **execution stack**, **control stack**, **function stack**, or **run-time stack**, and often referred to as just **the stack**) stores information about the active functions of a program. Although maintenance of the call stack is important for the proper functioning of most software, the details are normally transparent to the author of a program written in a high-level language. The main purpose of a call stack is to keep track of the address to which each active function should return control when it finishes executing. There is usually a single call stack for each running program (or, more accurately, with each thread of a process).

More generally, the **stack** is a section of computer memory used to store all the variables that are declared and initialised before a program is run. The **heap** is the section of computer memory used to store the variables created or initialised during execution of a program. That is, heap is an area of memory used for **dynamic memory allocation**. Blocks of memory can be allocated and freed in an arbitrary order. The pattern of allocation and size of blocks is not known until run time.

## Process scheduling

- Two most important computer resources are CPU time and memory:
  - machine instructions are executed by the CPU;
  - OS must ensure that all programs get access to CPU.
- Process scheduling is used to determine which program uses the CPU next:
  - based on process priority;
  - no process should 'die' of processor 'starvation'.
- Assigning CPU to process done by **dispatcher**:
  - process allocated a time slice by dispatcher.

101

The two most important resources of a computer are CPU time and memory:

- machine instructions are executed by the CPU;
- the OS must ensure that all programs gain access to the CPU.

Process scheduling is used to determine which program next uses the CPU:

- this is typically based on the priority of a process (a numerical value indicating how important the process is), as well as some kind of 'round robin' system, i.e. so that processes of equal priority are given CPU time in turn;
- no process should 'die' of processor 'starvation'.

The assignment of the CPU to a process is done by the **dispatcher**:

- the process is allocated a time slice (i.e. a specific period of time for it to execute) by the dispatcher.

## Process state

- Process has an **execution state**:
  - maintained by the operating system;
  - defines how the process is treated by the operating system.
- Most common process states are:
  - running;
  - ready;
  - blocked.

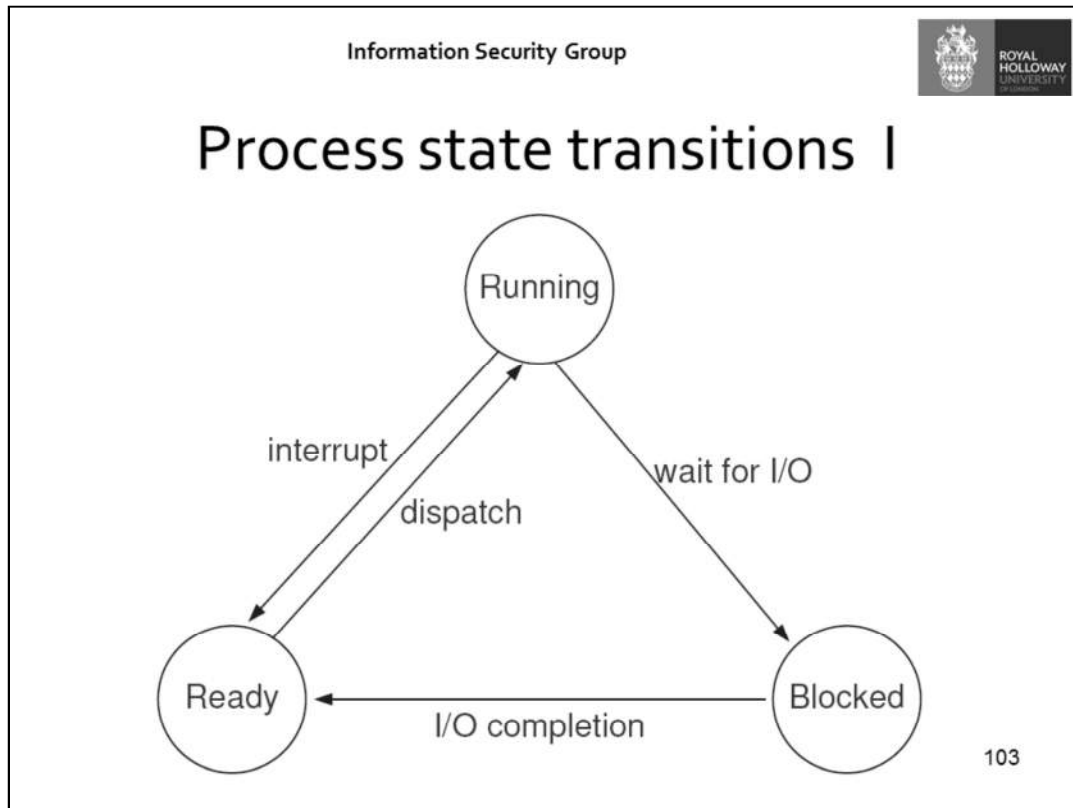
102

Every process has an **execution state**. This state:

- is maintained by the operating system;
- defines how the process should be treated by the operating system.

The most common process states are:

- running;
- ready (to run);
- blocked (waiting for something to occur before it can run).



The diagram shows the three common execution states of a process, as well as the possible transitions between these states.

## Process state transitions II

- When program starts executing, new process created and added to **ready list**:
  - when it reaches the top of the ready list and CPU becomes available, process enters the running state;
  - process may enter the blocked state (waiting for I/O to complete);
  - otherwise process may be interrupted (time slice ends or higher priority process needs the processor) and returns to ready state.

104

When a program starts executing, a new process is created and added to the list of processes in the ready state, the **ready list**.

When it reaches the head of the ready list and the CPU becomes available the process enters the running state;

The process may enter the blocked state (waiting for I/O to complete);

Otherwise the process may be interrupted (time slice ends or higher priority process needs the processor) and returns to the ready state, and would typically be added to the bottom of the ready list (depending on its priority).

## Process control block

- OS needs information about process so can resume execution, including:
  - execution state;
  - value of program counter and other registers;
  - memory management information;
  - I/O resources, e.g. files, in use by process.
- Information stored in a process descriptor or **process control block**:
  - created, maintained and protected by the OS.

105

The operating system requires a lot of information about each process so that it can resume execution when that process enters the ready state, including:

- the execution state of the process (i.e. whether it is running, blocked or ready);
- the value of the program counter and other registers;
- memory management information (i.e. which areas of main memory are being used by the process);
- references to I/O resources such as the set of files in use by the process.

This information is stored in a process descriptor or **process control block**, which is created, maintained and protected by the operating system.



## Program management

- **Utility software** used to create new programs:
  - **compiler** generates object code from source code;
  - **linker** creates executable program from object code;
  - **loader** (part of OS) writes contents of executable program into main memory.
- Most machine instructions do something to operands in registers or main memory:
  - part of linking and loading is to generate appropriate memory addresses for these operands.

106

**Utility software** is used to create new programs. Such software includes:

- a **compiler**, which generates object code from source code;
- a **linker**, which creates an executable program from object code;
- a **loader** (part of the OS), which is responsible for loading the contents of an executable program (typically in a file) into main memory.

Recall that most machine instructions do something to operands, which are values that are stored either in registers or in main memory. Part of the linking and loading processes is to generate appropriate memory addresses for these operands.

## Program address space I

- Source code uses variables and constants.
- However, object code uses addresses in which variables and constants are stored.
- Set of addresses referenced in the object code is called the **address space**.
- Address space does not generally refer to actual physical memory locations.
- Address space is **virtual** and addresses must be translated to physical memory addresses.

107

Source code makes references to variables and constants. By contrast, object code makes references to addresses in which variables and constants are stored.

The set of addresses referenced in the object code is called the **address space**.

The address space does not generally refer to actual physical memory locations.

The address space is **virtual** and addresses must be translated to physical computer memory locations at some stage. That is, a piece of software (an **executable**) will be constructed in such a way that it is designed to work in a fixed address space – however, it may be loaded into any part of the computer's main memory.

## Program address space II

- Recall previous example:
  1. Get input 1 and write it to slot 10 (*x* will be stored in slot 10)
  2. Get input 2 and write it to slot 11 (*y* will be stored in slot 11)
  3. Write 0 in slot 12 (*result* will be stored in slot 12)
  4. If the contents of slot 11 is 0 go to instruction at slot 8
  5. Add the contents of slot 10 to the contents of slot 12
  6. Subtract one from the contents of slot 11
  7. Go to instruction at slot 4
  8. Finish
  9. (Not used)
  10. (Storage for *x*)
  11. (Storage for *y*)
  12. (Storage for *result*)
- The virtual addresses range from 1 to 12.

108

We recall the previous example:

1. Get input 1 and write it to slot 10 (*x* will be stored in slot 10)
2. Get input 2 and write it to slot 11 (*y* will be stored in slot 11)
3. Write 0 in slot 12 (*result* will be stored in slot 12)
4. If the contents of slot 11 is 0 go to instruction at slot 8
5. Add the contents of slot 10 to the contents of slot 12
6. Subtract one from the contents of slot 11
7. Go to instruction at slot 4
8. Finish
9. (Not used)
10. (Storage for *x*)
11. (Storage for *y*)
12. (Storage for *result*)

In this example, the virtual addresses range from 1 to 12. However, this does not mean that the program must always be located in the part of the computer memory with these addresses (this would be very inflexible).

## Virtual address translation

- When might address translation occur?
  - At compile/link time:
    - requires compiler/linker to know where program will be loaded in main memory;
    - very rare today.
  - At load time:
    - requires the code to remain in main memory once loaded;
    - rare in multi-tasking systems because code may be re-located during execution.
  - At run time:
    - requires hardware support to translate addresses during execution.

109

There are various possible occasions when address translation might occur.

- At compile/link time:
  - this requires the compiler/linker to know where in main memory the program will be loaded;
  - such an approach is very rare today, since it is very inflexible (it means a program must always occupy same part of physical memory).
- At load time:
  - this requires the code to remain in main memory once it has been loaded;
  - this is rare in multi-tasking systems because code may be re-located during execution.
- At run time:
  - this requires hardware support to translate addresses during the execution cycle;
  - modern processors provide support for run-time address translation, which is how modern computers work.

# Memory management I

- Executable programs are loaded into main memory before execution.
- Data is loaded into main memory before it is read or written.
- Certain operating system data structures are stored in main memory.

Executable programs are loaded into main memory before execution.

Data is loaded into main memory before it is read or written.

Certain operating system data structures are also stored in the computer's main memory.

## Memory management II

- OS must manage these competing demands for storage, and must ensure:
  - that confidential data is not read by unauthorised processes;
  - that sensitive data is not modified by unauthorised processes.
- **Memory protection** used to ensure that process only reads from, or writes to, memory locations which it is authorised to.

111

The operating system has to manage all these competing demands for storage. In particular, the operating system has to ensure that:

- confidential data is not read by unauthorised processes;
- sensitive data is not modified by unauthorised processes.

**Memory protection** is used to ensure that a process only reads from, or writes to, memory locations for which it has the necessary authorisation.

## Virtual memory management

- Virtual address space provides a way of partitioning main memory:
  - can use virtual memory management to isolate particular regions of memory;
  - can use virtual memory management for memory protection;
  - can also specify different types of access for different regions of memory, e.g.:
    - a region containing a program can be read only;
    - a region containing data can be read/write.

112

A virtual address space provides a way of partitioning main memory:

- can use virtual memory management to isolate particular regions of memory;
- can use virtual memory management for memory protection;
- can also specify different types of access for different regions of memory, e.g.:
  - a region containing a program can be read only;
  - a region containing data can be read/write.

## Secondary storage

- Computer systems store lots of data:
  - main memory cannot be used to store huge volumes of data processed by computers;
  - main memory is volatile, whereas data often needs to be stored for long periods;
  - so need way of organising and storing large quantities of data.
- **Secondary storage** is provided by persistent, high-capacity, storage media:
  - e.g. disk drives, floppy disks, USB sticks;
  - has much slower access speed than main memory.

113

Computer systems typically store large volumes of data:

- main memory cannot be used to store the huge volumes of data processed by computers;
- main memory is volatile, whereas a data typically needs to be stored for long periods of time;
- hence we need a way of organising and storing large quantities of data.

**Secondary storage** consists of persistent, high-capacity storage media such as disk drives, floppy disks, USB sticks. These forms of secondary storage have much slower access speeds than main memory (particularly in terms of latency, i.e. delay).

## File management

- Secondary storage typically organised using a **file system**:
  - an operating system supports one or more file systems;
  - file system provides a way of organising, storing, retrieving and protecting data.
- File is a block of data with some meaning:
  - the file system manages storage and retrieval of files;
  - a (computer) file is synonymous with a paper document.
- Each file associated with **metadata**:
  - length, time of creation, time of last modification, owner-ID, access permissions, ...

114

Secondary storage is typically organised using a **file system**:

- an operating system will support one or more file systems;
- a file system provides a way of organising, storing, retrieving and protecting data.

A file represents a block of data that forms a logical unit:

- the file system manages the storage and retrieval of files;
- a (computer) file is synonymous with a paper document.

Every file is associated with a range of **metadata** containing information about the file, such as:

- length, time of creation, time of last modification, owner-ID, access permissions, ...

Different types of file will have different associated metadata types.

## File operations

- File system used to store data in, and retrieve data from, files:
  - provides a number of functions that can be called by application programs to interact with files;
  - typical file operations include `create`, `delete`, `open`, `close`, `read`, `write`, `append`.

The file system is used to store data in, and retrieve data from, files. It provides a number of functions that can be called by application programs to interact with files. Typical operations provided by a file system include `create`, `delete`, `open`, `close`, `read`, `write`, `append`.

## File open

- **File open** function used to make a file available to an application program.
- After file open request, system:
  - checks program is permitted to access the file;
  - loads file data into the process memory space;
  - adds reference (a **file handle**) to list of files being accessed by the program, where the file handle includes access control information.
- Calls to open function trigger a security check:
  - verifies that the process that invoked the open function is authorised to open the requested file.

116

The **file open** function (a **system call**) is used to make a file available to an application program. When it receives a file open request, the system:

- checks that the program is permitted to access the file (**authorisation**);
- loads the file data into the process memory space;
- adds a reference (a **file handle**) to the list of files being accessed by the program, where the file handle includes access control information that governs how the program can access the file (e.g. the program may be allowed to read the file but not modify it).

All calls to the open function will trigger a security check. This check will verify that the process that invoked the open function is authorised to open the requested file.

## Directories I


- **Directory** is a user-friendly way of organising files in a computer system:
  - provides structured view of the computer system, making it easier to manage files;
  - it is a special type of file containing information about other files (i.e. the files contained in that directory);
  - directory names may form a **hierarchical namespace** to simplify locating resources.

117

A **directory** is a user-friendly way of organising files in a computer system:

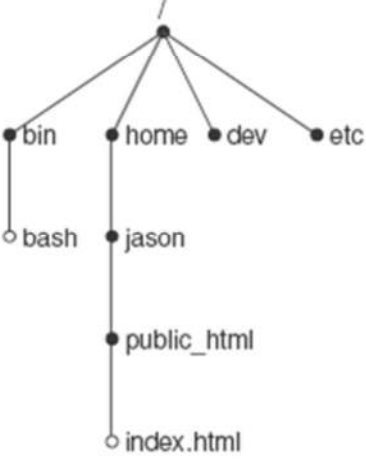
- it provides a structured view of the computer system, making it easier to manage files;
- the directory is itself a special type of file that contains information about other files (namely the files contained in that directory);
- directory names will typically form a **hierarchical namespace** to make it easier to locate resources.

Information Security Group



## Directories II

- Hierarchical namespace can be visualised as a tree:
  - tree has a **root directory**;
  - tree has the property that every (child) node has a unique parent node;
  - parent node may have more than one child;
  - each file has a unique name called the **absolute path name**.



```
graph TD; Root["/"] --- bin; Root --- home; Root --- dev; Root --- etc; bin --- bash; home --- jason; home --- public_html; home --- index_html["index.html"]
```

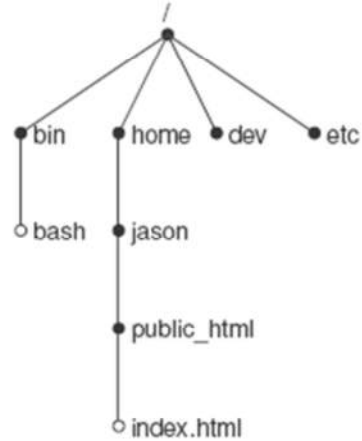
118

A hierarchical namespace can be visualised as a tree-like structure:

- the tree has a **root directory**;
- the tree has the property that every (child) node has a unique parent node;
- the parent node may have more than one child;
- each file has a unique name called the **absolute path name**, derived from the nodes on the path from the root to the node corresponding to the file.

## Directories III

- Each process has a working directory or current directory:
  - determined by the user account information at authentication time;
  - default current directory of a new process is that of its parent process;
  - relative path name defines a path from the current directory;
  - in example shown, `/public_html/index.html` is the path name relative to the directory `/home/jason`



119

Each process has a working directory or current directory:

- this working/current directory is determined by the user account information at authentication time;
- the default current directory of a new process is that of its parent process;
- a relative path name defines a path from the current directory;
- in the example shown, `/public_html/index.html` is the path name relative to the directory `/home/jason`

## User interfaces I

- Operating system provides way to enable users to easily interact with system resources:
  - called a **shell** (command-line interface (CLI)) or **desktop** (graphical user interface (GUI));
  - CLI is used to enter commands to run programs, print files, create or copy files, ...;
  - GUI enables a user to interact with windows and icons to run programs, print files, create or copy files, ...

120

The operating system provides an environment that enables the user to easily interact with system resources:

- often called a **shell** (in the case of a command-line interface (CLI)) or **desktop** (in the case of a graphical user interface (GUI));
- a CLI is used to enter commands to run programs, print files, create or copy files, ...;
- a GUI enables a user to interact with windows and icons to run programs, print files, create or copy files, ...

## User interfaces II

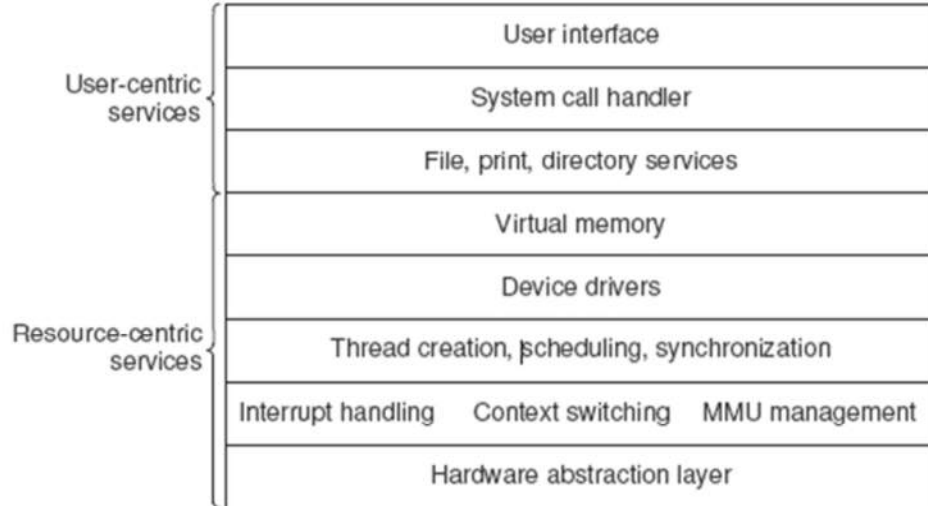
- User interface is a program run after a user has been successfully authenticated:
  - process created by this program is associated with security-related information for that user;
  - programs launched from that interface are associated with the same information;
  - programs are thus associated with users (enabling access control and audit).

121

The user interface is a program that is run after a user has been successfully authenticated:

- the process that is created when this program is run (recall that running any program involves creating a process) is associated with the security-related information for the user that has been authenticated;
- any programs launched from that interface are associated with the same information;
- in this way programs are associated with users, and so user-based access control and audit can be achieved.

# Summary of OS services



The picture summarises the services provided by a 'typical' OS.

## Hardware virtualisation I

- **Hardware virtualisation** involves hiding physical characteristics of a computing platform from executing software.
- Software executes on an 'abstract' (virtualised) computing platform.
- The software that provides the virtualisation is called a **hypervisor** or **virtual machine monitor (VMM)**.

123

**Hardware virtualisation** involves hiding the physical characteristics of a computing platform from executing software. In such a case, the software executes on an 'abstract' (virtualised) computing platform.

The software that provides the virtualisation layer is called a **hypervisor** or **virtual machine monitor (VMM)**.

## Hardware virtualisation II

- VMM creates simulated computer environment, a **virtual machine**, for **guest software**.
- Guest software not just user applications; many VMMs allow execution of complete OSs.
- Guest software executes as if running on physical hardware, except access restrictions may apply to:
  - physical system resources (such as network access, display, keyboard, and disk storage);
  - some peripherals, or only a subset of the device's native capabilities might be available.

124

A VMM creates a simulated computer environment, known as a **virtual machine**, for its **guest software**.

The guest software is not limited to user applications; many VMMs allow the execution of complete operating systems.

The guest software executes as if it were running directly on the physical hardware, except that certain access restrictions may apply to:

- physical system resources (such as network access, display, keyboard, and disk storage);
- some peripherals, or only a subset of the device's native capabilities may be available.

## Virtualisation – properties

- Virtualisation necessarily involves a loss of performance:
  - resources required to run the hypervisor;
  - need for translation of instructions.
- There is reduced performance on the virtual machine compared to running 'native' on the physical machine.

Virtualisation necessarily involves a loss of performance, because of the resources required to run the hypervisor/VMM, and the need for translation of instructions.

As a result, there is reduced performance on the virtual machine compared to running 'native' (i.e. in the non-virtualised way) on the physical machine.

## Virtualisation – advantages

- **Server consolidation** allows many small physical servers to be replaced by one larger physical server:
  - increases utilisation of costly hardware resources such as CPU.
- Hardware is consolidated but OSs are not:
  - each OS runs inside a virtual machine;
  - large server can host many such guest virtual machines.

126

**Server consolidation** allows many small physical servers to be replaced by one larger physical server:

- increases utilisation of costly hardware resources such as CPU.

The hardware is consolidated, but OSs are not:

- each OS runs inside a virtual machine;
- a large server can host many such guest virtual machines.

# Agenda

- What do computers do?
- Some common notation
- Computer architectures
- Software
- Operating systems
- Resources

## Further reading

- A. Tanenbaum, *Modern Operating Systems*, chapter 1.
- A. Silberschatz, P.B. Galvin and G. Gagne, *Operating Systems Concepts*, chapters 1 and 2.
- ... many other introductory books.

The following books are recommended:

- A. Tanenbaum, *Modern Operating Systems*, chapter 1.
- A. Silberschatz, P.B. Galvin and G. Gagne, *Operating Systems Concepts*, chapters 1 and 2.

However, there are many other textbooks covering similar ground, and of course there are many very valuable resources on the web.