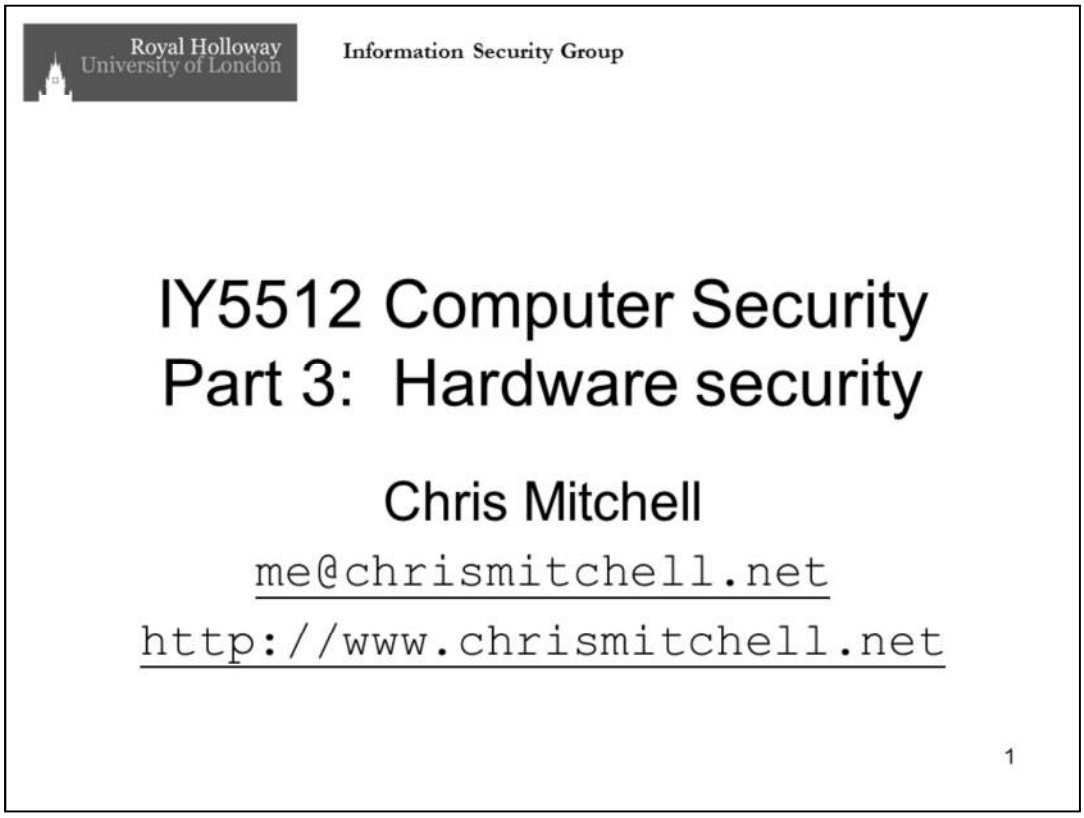
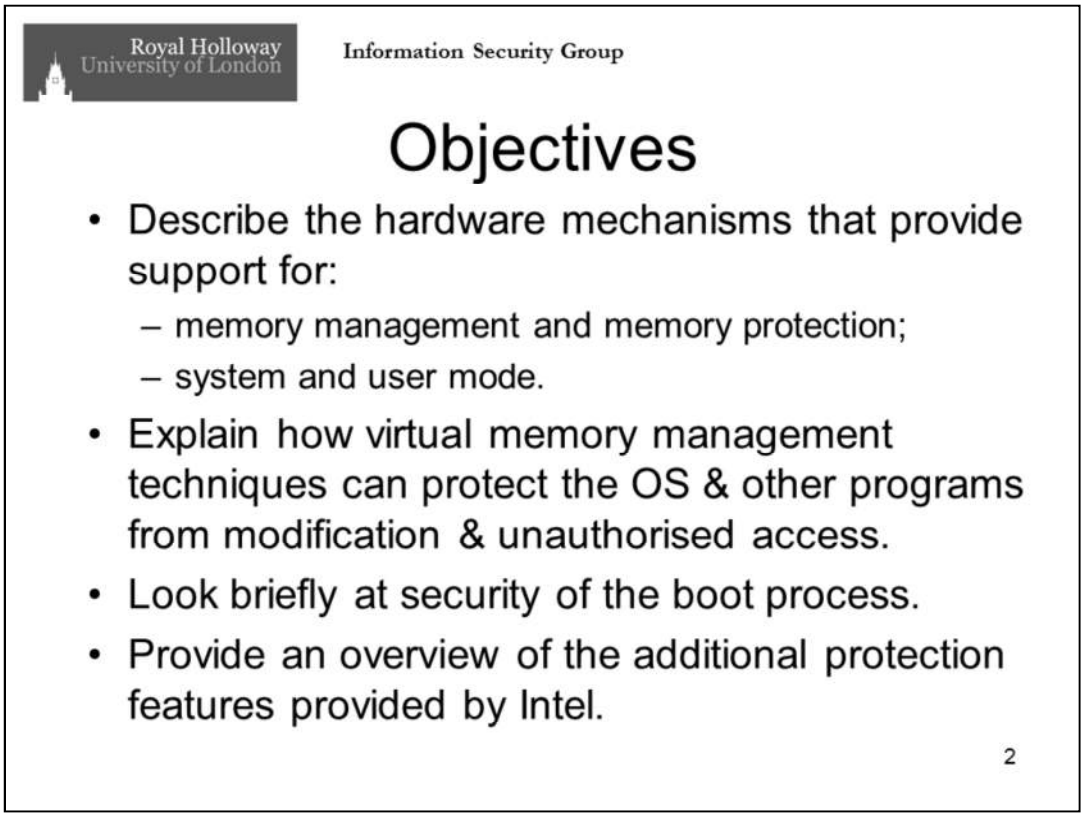

Royal Holloway
University of London
Information Security Group
IY5512 Computer Security
Part 3: Hardware security
Chris Mitchell
me@chrismitchell.net
http://www.chrismitchell.net
1



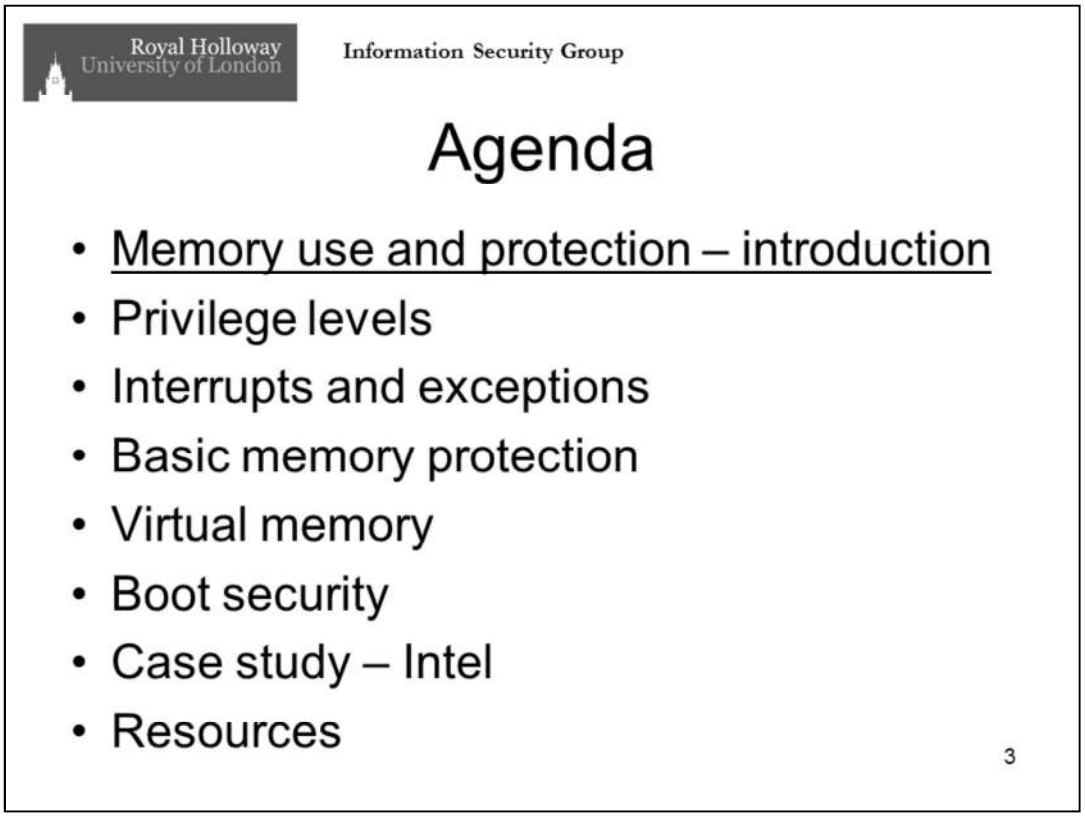
In this part of the course we consider the hardware mechanisms that provide support for:

- memory management and memory protection;
- the distinction between system mode and user mode.

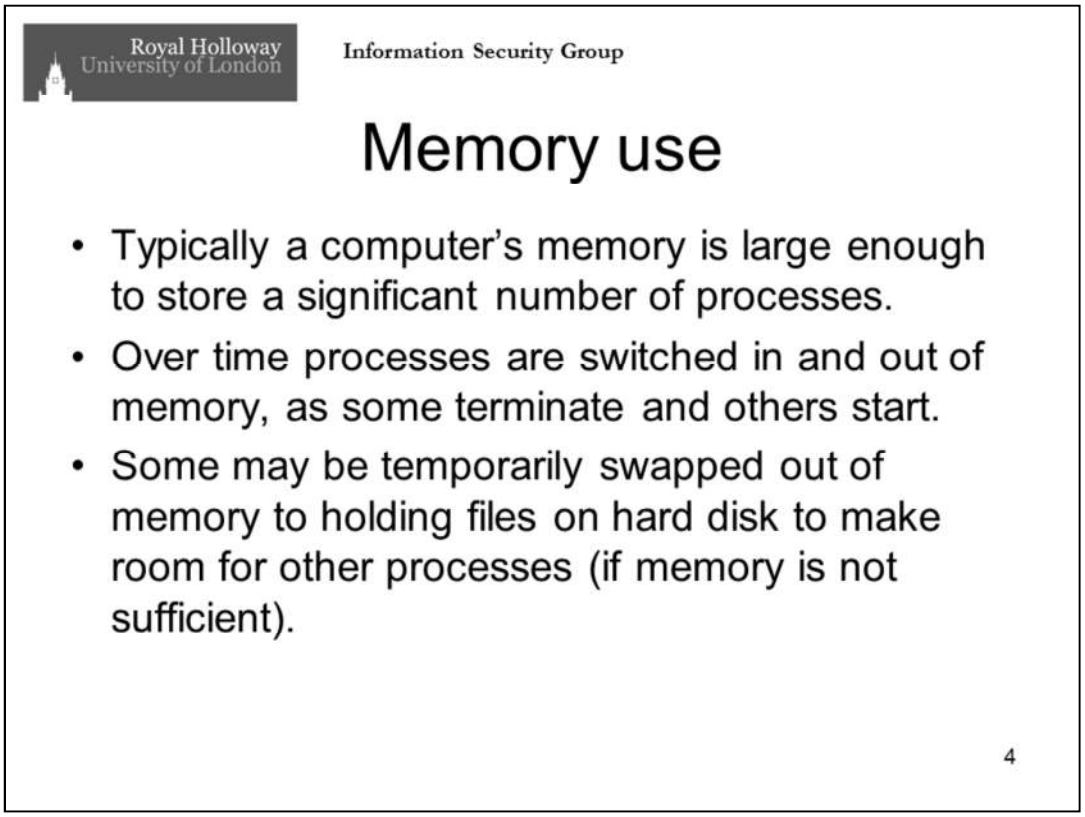
We will consider how virtual memory management techniques can be used to protect the operating system and other programs from modification and unauthorised access.

We consider the security of the boot process and the role of the BIOS in this process.

We also provide a brief overview of the additional protection features provided by the Intel processor architecture.



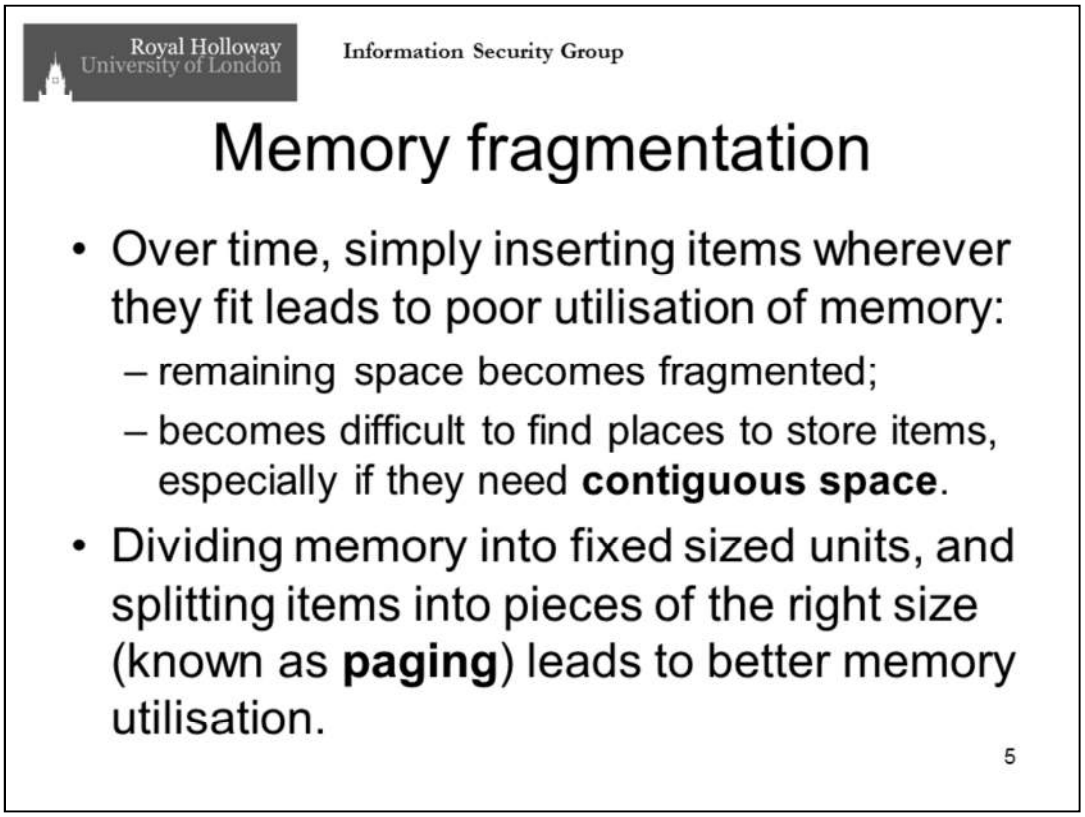
We start by looking at perhaps the most fundamentally important security functionality, namely memory protection.



Typically a computer's memory will be large enough to accommodate a significant number of processes.

Over time these processes will be switched in and out of memory, e.g. as some terminate and others start.

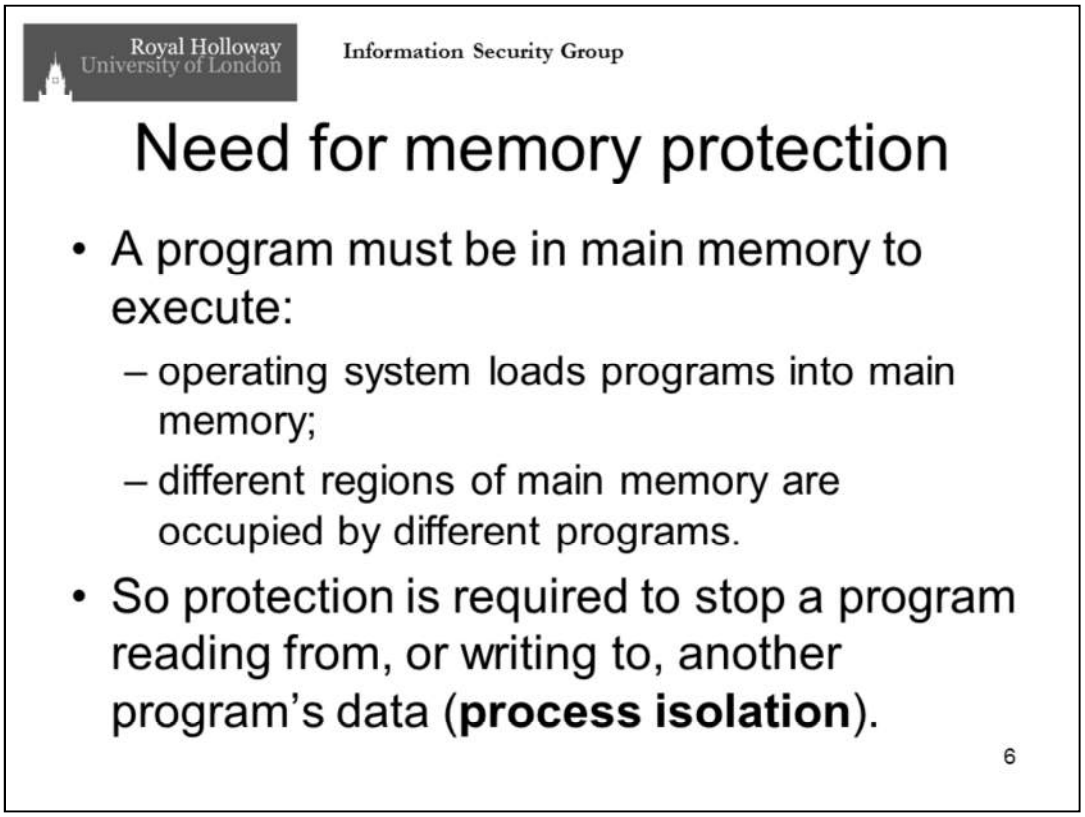
If the total amount of computer memory is not sufficient, some processes may be temporarily swapped out of memory to holding files on hard disk to make room for other processes. If memory is limited, this can seriously reduce the performance of the system, as it takes time to move data from memory to disk and vice versa.



Over time, simply inserting items wherever they will fit leads to poor utilisation of memory. In particular:

- the remaining memory space becomes fragmented;
- it becomes increasingly difficult to find places to store items, especially if they need **contiguous space** (i.e. a set of memory locations with consecutive addresses).

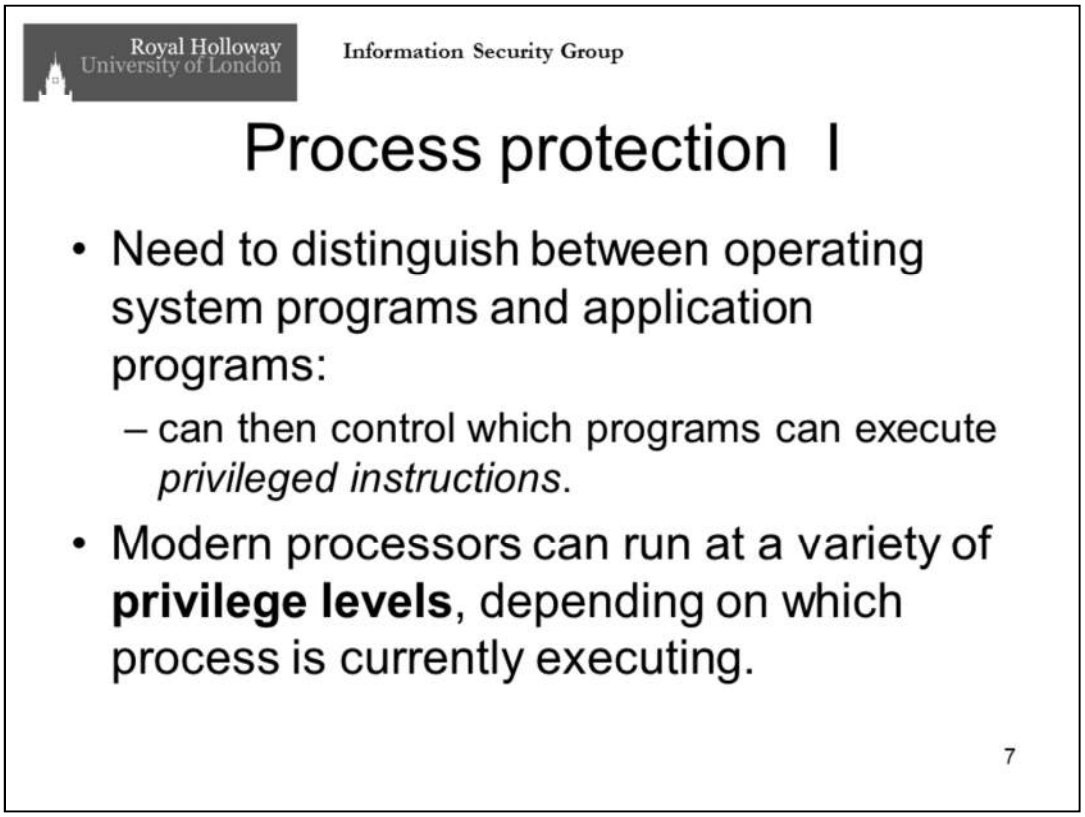
Dividing up computer memory into fixed sized storage units, and splitting items to be put into memory (e.g. processes) into pieces of the appropriate size leads to better utilisation of storage. Such a process is known as **paging**, where the fixed sized units of storage are called **pages**. Note that the pages allocated to a program can be scattered across main memory, and do not need to be adjacent. Also, the set of memory pages allocated to a process can vary during the lifetime of the process (as it is swapped in and out of memory).



In order to execute a program it must be in main memory.

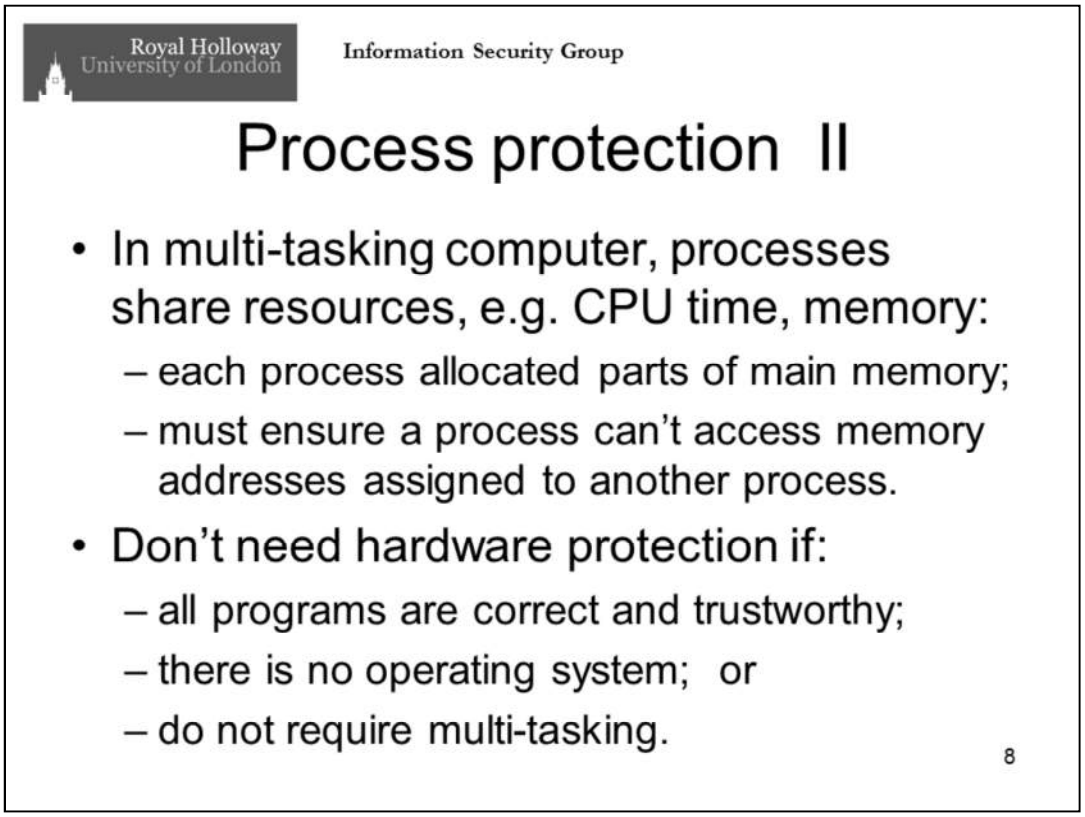
One of the operating system's jobs is to load programs into main memory.

Different regions of main memory are occupied by different programs, and hence the area of memory allocated to one program must be protected against access/interference by other programs running in other areas of memory. This is known as **process isolation**.



We also need to be able to distinguish between operating system programs and application programs. We can then control which programs are able to execute privileged instructions, i.e. instructions which would enable one application program to interfere with another, or to make uncontrolled accesses to resources.

Modern processors are able to run at a variety of **privilege levels**, depending on which process is currently executing. These privilege levels can be used to restrict access to critical functionality. Certain machine instructions are only available to programs running at a higher privilege level.



In a multi-tasking computer, processes share resources, such as CPU time and main memory:

• each process is allocated certain parts of main memory (using paging) and a certain proportion of the processor execution cycles;

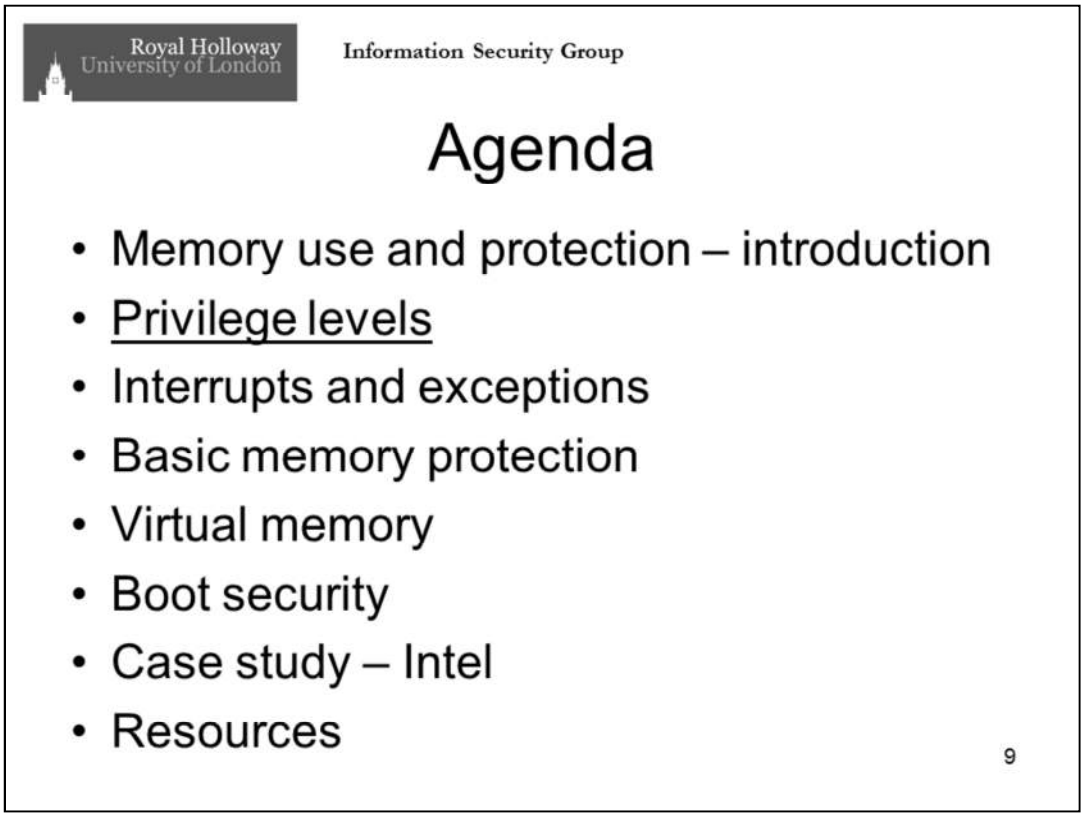
• the system design needs to ensure that one process cannot access memory addresses assigned to a different process or to the OS.

However, hardware protection is not required if:

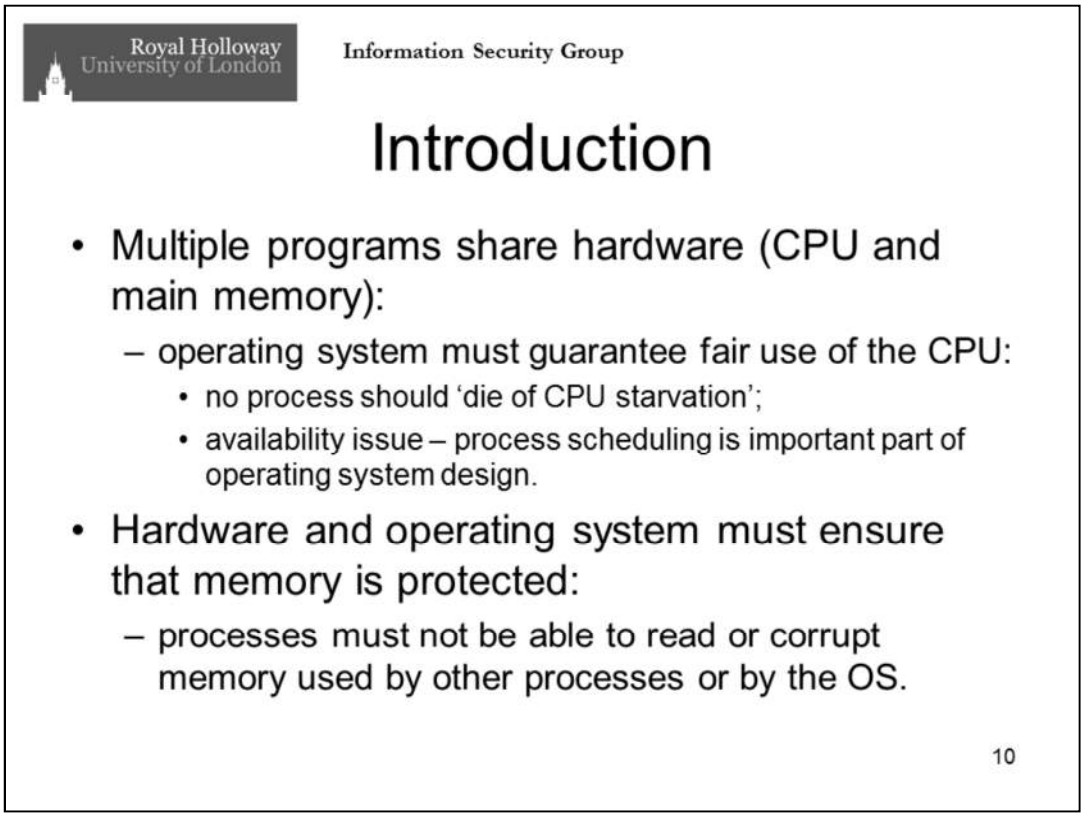
• we assume all programs are correct and trustworthy;

• there is no operating system; or

• we do not require multi-tasking.



We next look in more detail at the notion of privilege levels, i.e. different modes of execution of a processor. Each level offers executing programs differing levels of access to system resources (i.e. certain instructions cannot be executed in levels with lower privileges).

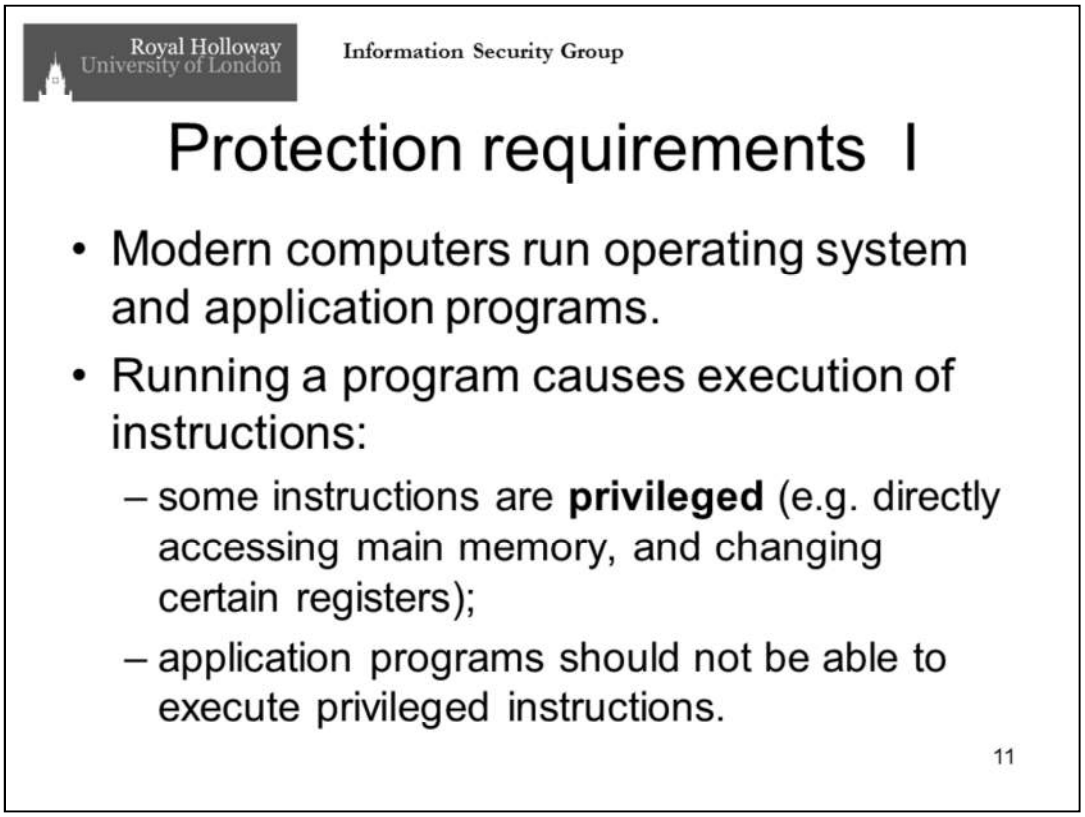


As already discussed, in a computer system, multiple programs share the system hardware (i.e. the CPU and the main memory). The operating system must guarantee fair use of the CPU:

• no process should 'die of CPU starvation', i.e. be deprived of any access to processor cycles;

• this is an availability issue - process scheduling is an important issue in operating system design.

The hardware and operating system must ensure that memory is protected. Specifically a process should not be able to read or corrupt memory used by other processes or by the operating system.



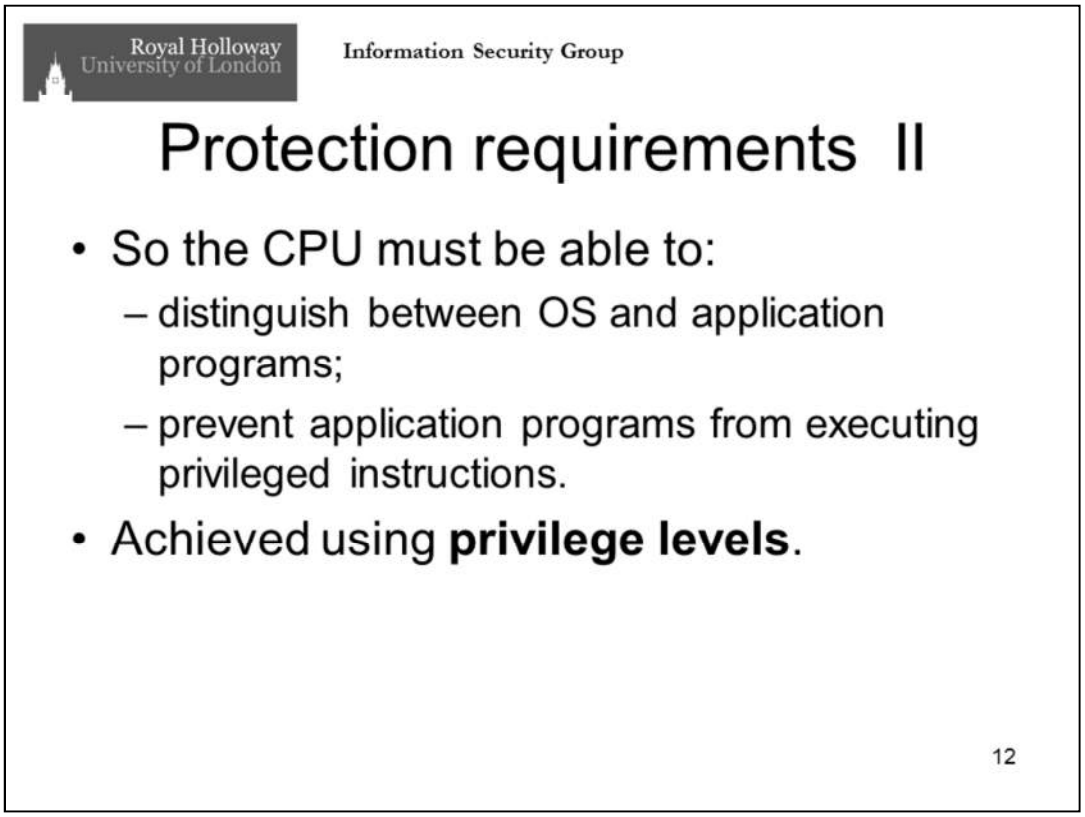
Modern computers run an operating system and application programs. Running a program causes the execution of instructions.

Some instructions are **privileged**, such as those involving:

• directly accessing main memory (i.e. without the addresses being translated using the virtual memory management functionality);

• changing certain registers.

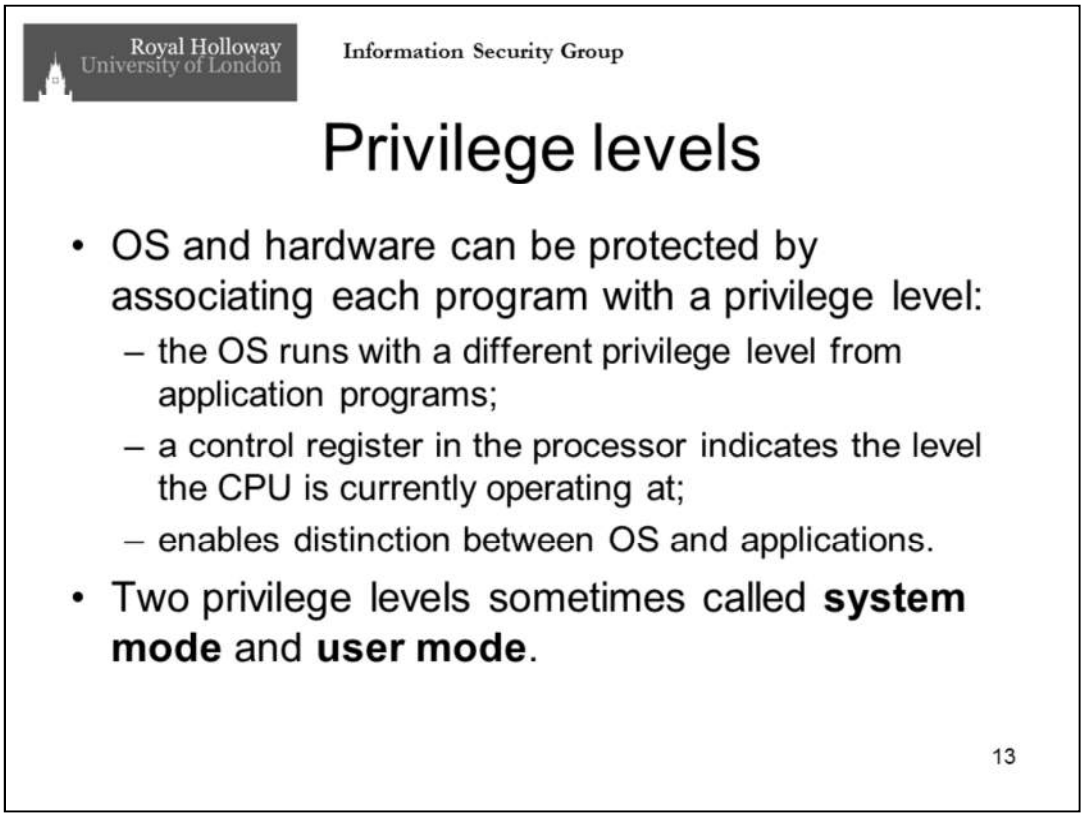
Application programs should not be able to execute privileged instructions.



In order to prevent application programs running privileged instructions, the CPU must be able to:

- distinguish between OS and application programs;
- prevent some programs from executing privileged instructions.

This is achieved using **privilege levels**.



The operating system and the hardware can be protected by associating each program with a privilege level:

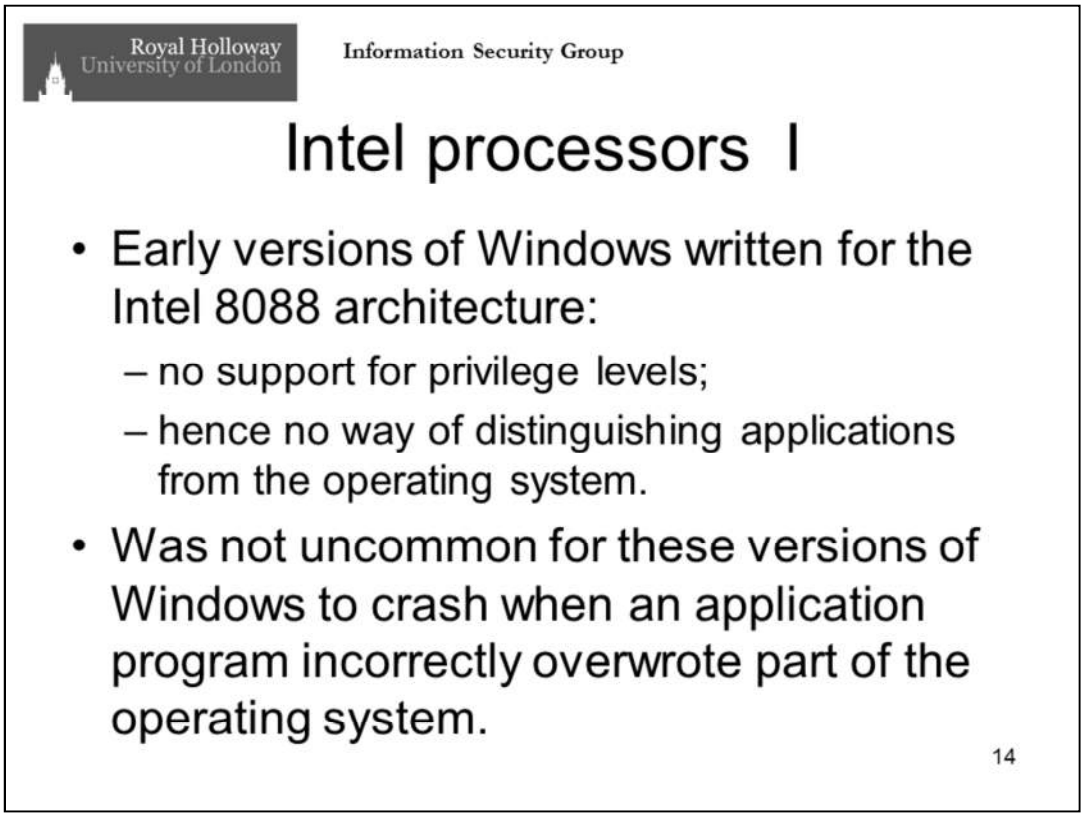
• the operating system runs with a different privilege level from application programs;

• one of the control registers on the processor indicates the level at which the CPU is currently operating;

• this enables a distinction to be made between operating system and applications.

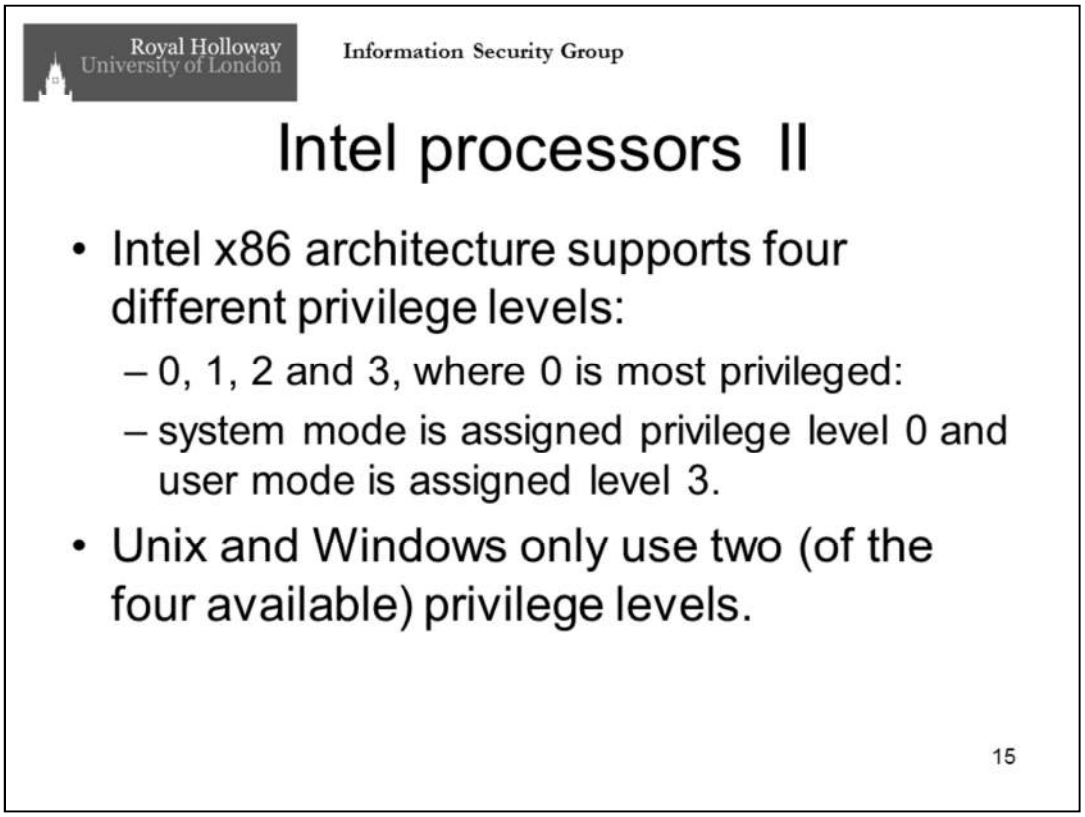
These two privilege levels are sometimes known as **system mode** and **user mode**.

Some instructions are not available if the processor is executing in system mode.



Early versions of Windows were written for the Intel 8088 architecture. In this architecture there is no support for privilege levels, and hence there is no way of distinguishing applications from the operating system.

When using these systems it was not uncommon for Windows to crash after an application program had incorrectly overwritten part of the operating system.



The more recent Intel x86 architecture actually supports a total of four different privilege levels. These are numbered 0, 1, 2 and 3, where 0 is the most privileged and 3 is the least privileged. System mode is assigned privilege level 0 and user mode is assigned level 3.

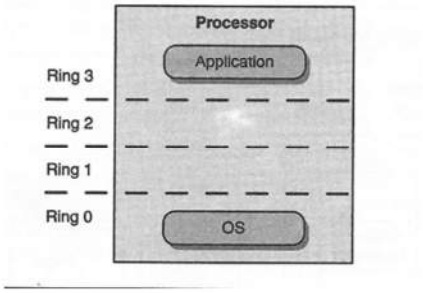
Both Unix and Windows only use two (of the four available) privilege levels, namely levels 0 and 3.

Royal Holloway University of London

Information Security Group

Intel privilege levels

- Four defined privilege levels (or **protection modes** or **rings**) for Intel processors: Rings 0, 1, 2 3.
- They separate the OS from applications.
- In principle, they also enable the OS to separate internal OS functions to help OS protect itself.

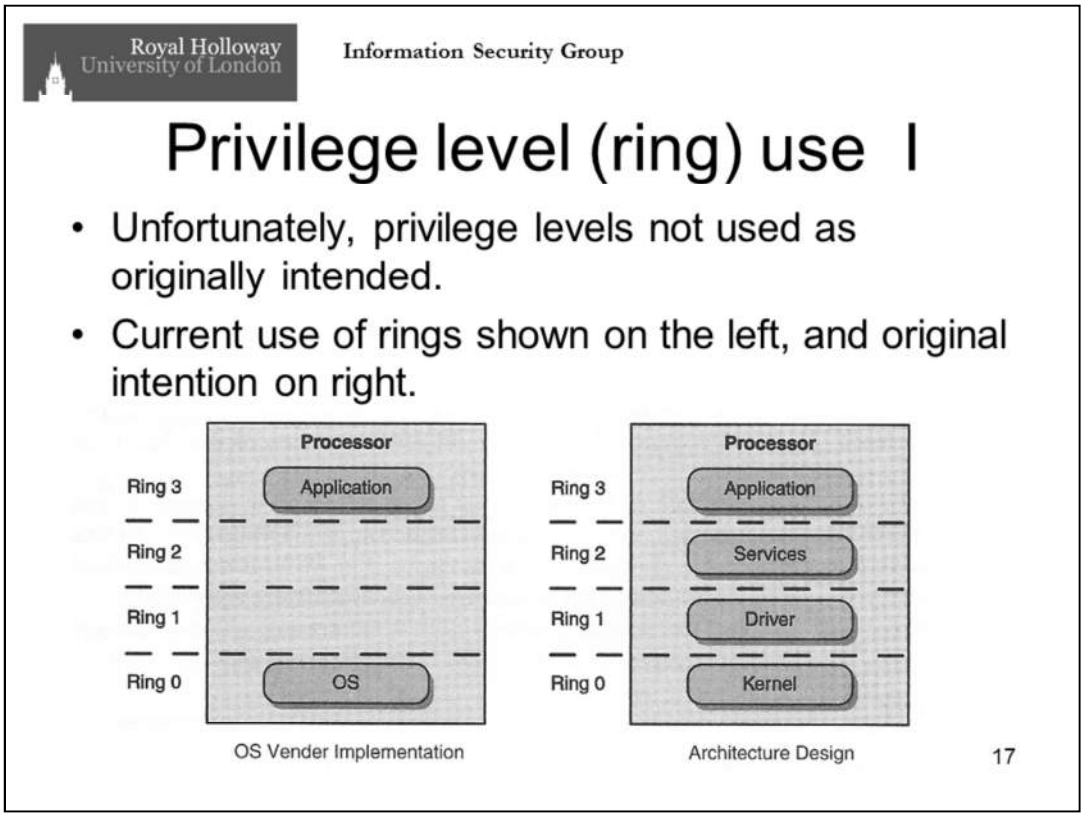


16

There are four defined privilege levels (also known as **protection modes** or **rings**) for Intel processors, known as Ring 0, Ring 1, Ring 2 and Ring 3.

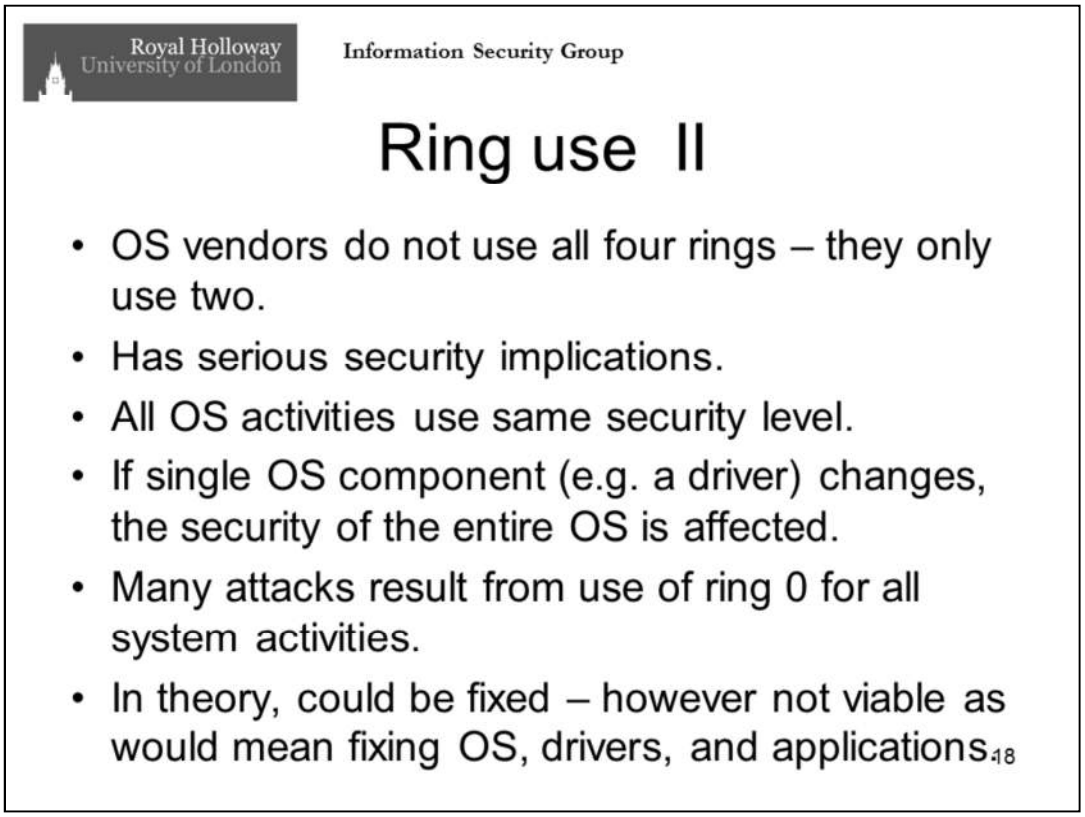
Their purpose is to separate the OS from applications running on the OS.

In principle they also enable the OS to separate (layer) internal OS functions to help the OS protect itself. However, in practice this does not happen.



Unfortunately, the four privilege levels (or rings as they are commonly known) provided by the Intel processor architecture have not really been used as was originally intended, so that the layering of OS functions is not achieved.

In the picture, the current use of the rings is shown on the left, and the original intention on the right. This picture applies for both Unix and Windows.



The OS vendors have not used all four rings – they only use two.

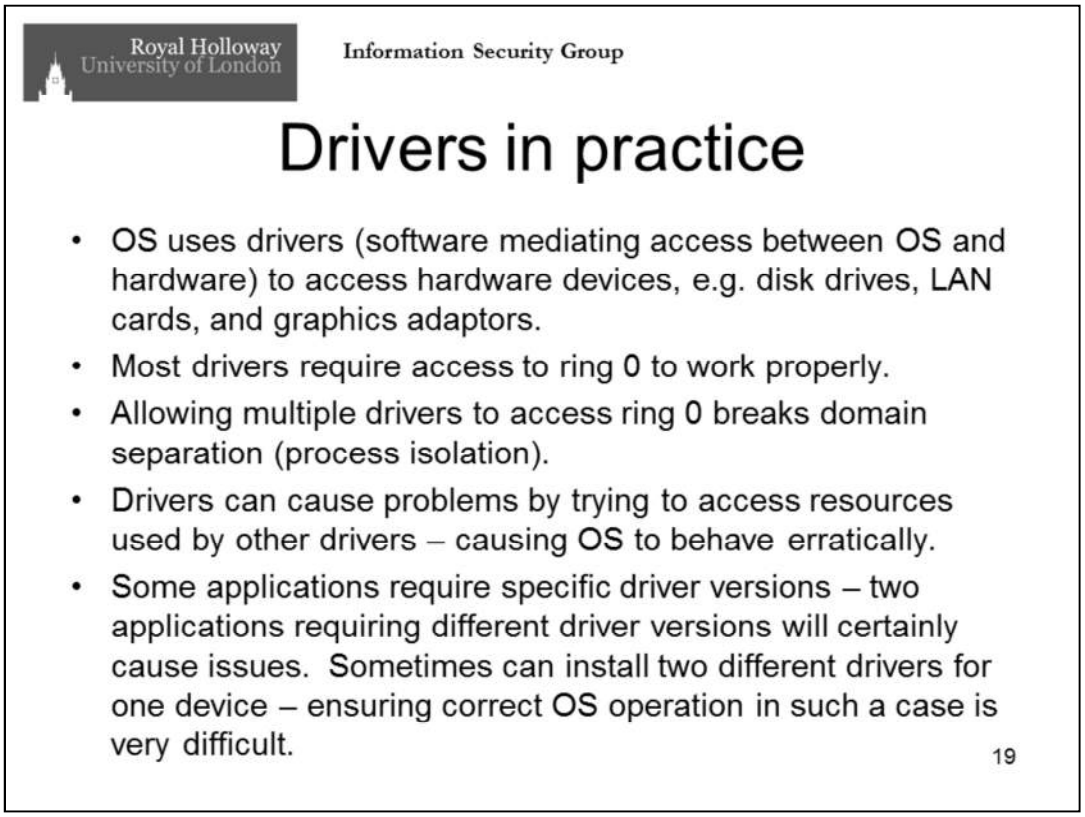
This has serious security implications.

All OS activities share the same hardware security level.

Every time a single OS component (e.g. a driver) changes, the security of the entire OS is affected.

Many attacks result from use of ring 0 for all system activities.

Whilst this could be fixed in principle, in practice it would require the OS, most drivers, and some applications to be rewritten – this is simply not a viable strategy.



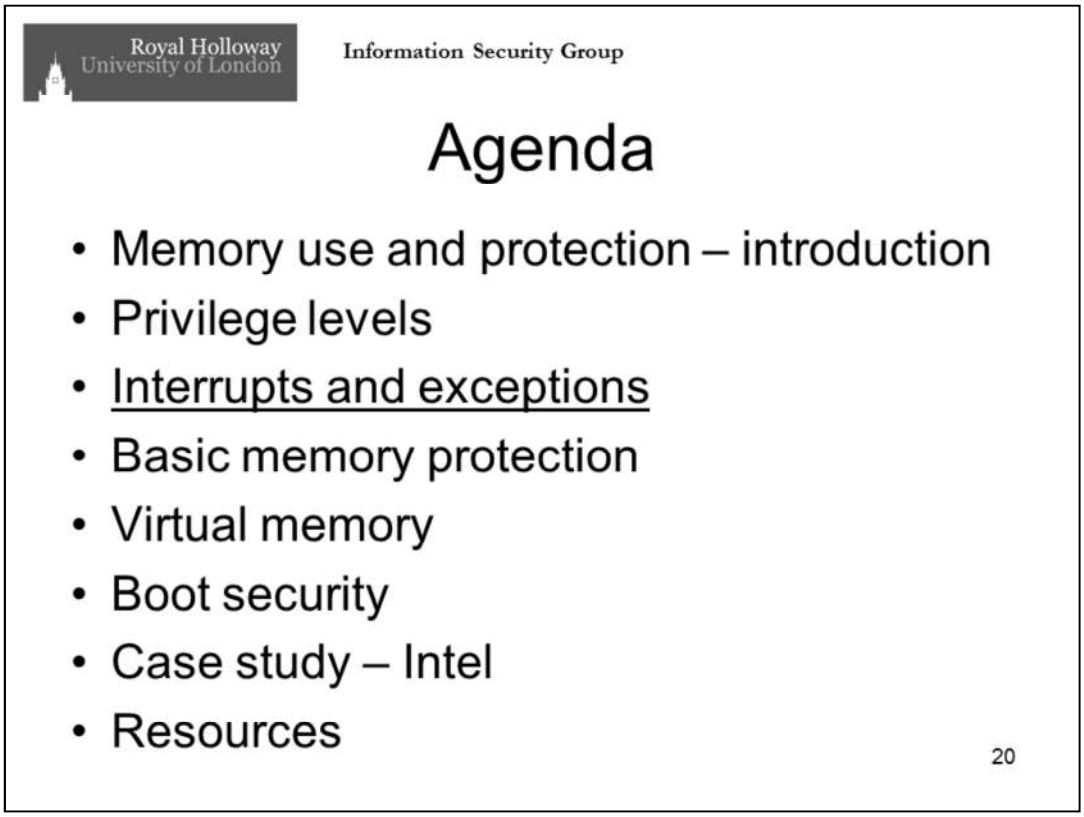
The OS requires drivers (i.e. pieces of software mediating access between OS and hardware) to access hardware devices, such as disk drives, LAN cards, and graphics adaptors.

Most drivers require access to ring 0 to work properly.

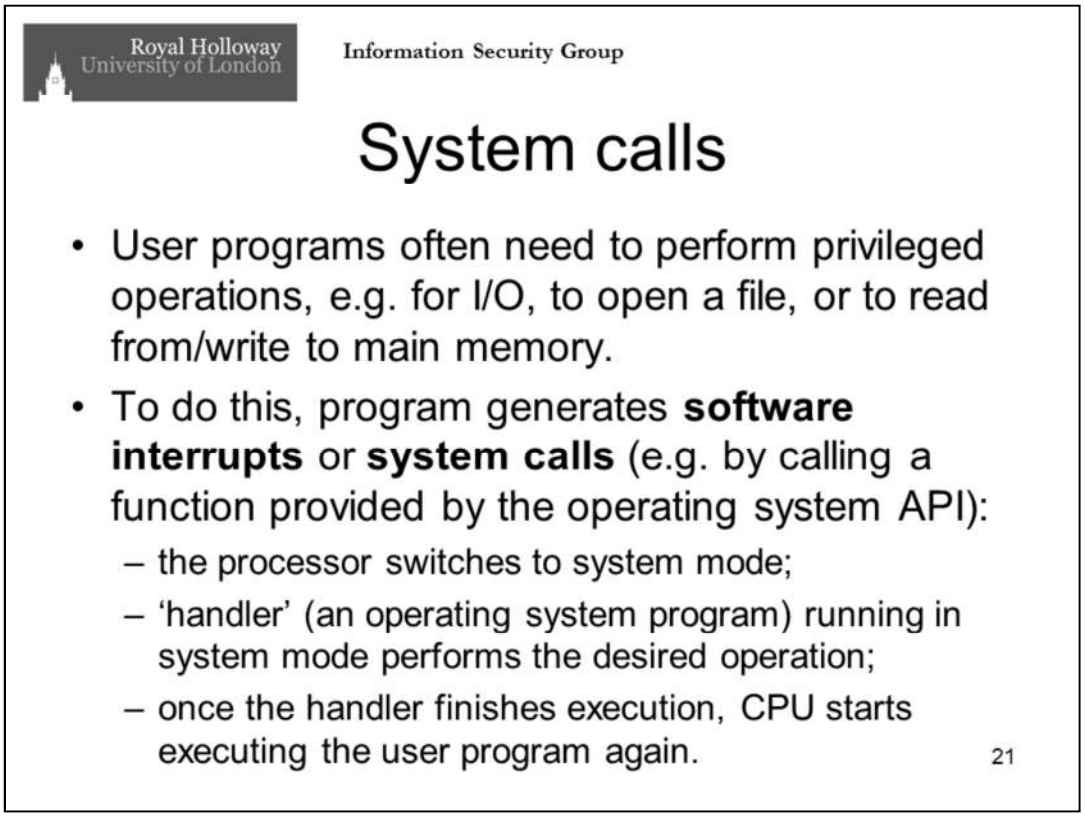
Allowing multiple drivers to access ring 0 breaks domain separation (process isolation).

Drivers can cause other problems by trying to access resources used by other drivers – causing the OS to behave erratically.

Some applications require specific versions of a driver – two applications requiring different driver versions will certainly cause issues. On occasion it is possible to install two different drivers for the same device – ensuring correct OS operation in such a case is very difficult.



Interrupts and exceptions are key functions of a processor that enable normal execution to be interrupted. These are of particular significance to security, since they can be misused to enable security breaches.



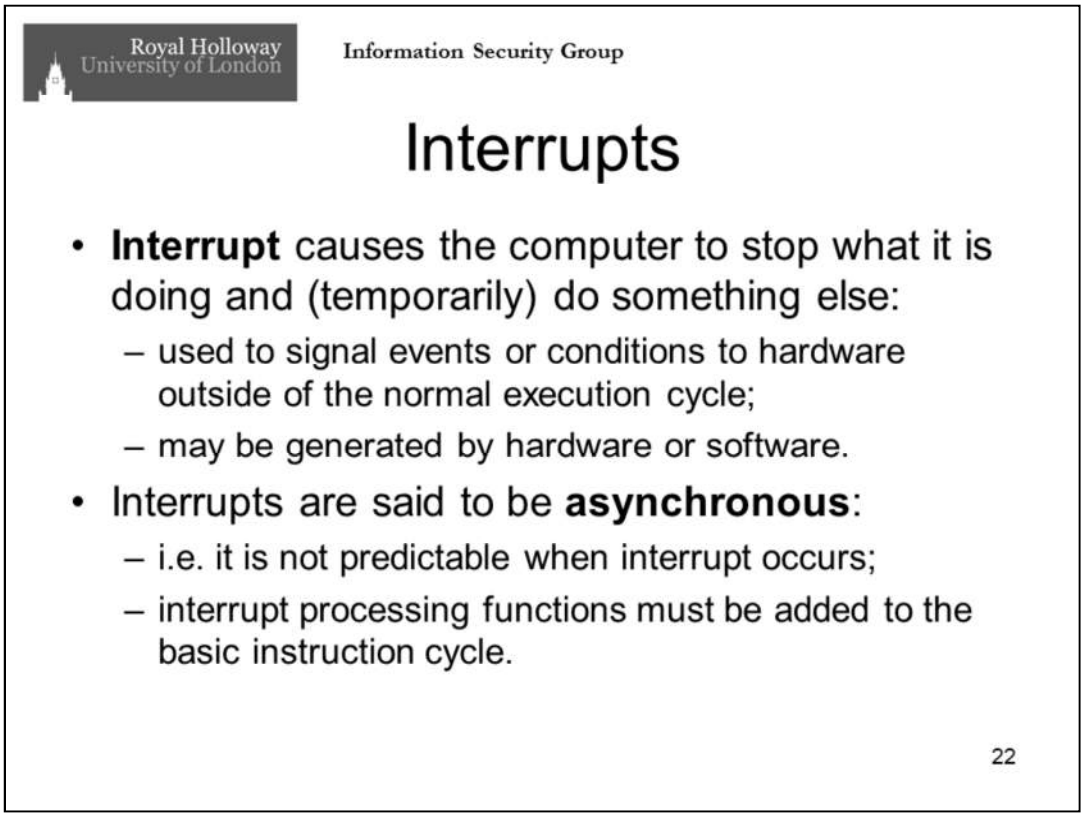
User programs often need to privileged operations (e.g. to perform input/output (I/O), to open files, or to read from/write to main memory. However, a user program running in user mode will not be able to execute the necessary instructions.

In order to achieve the desired objective, the program generates a **software interrupt** or **system call** (e.g. by calling a function provided by the operating system API). As a result:

• the processor switches to system mode;

• a 'handler' (an operating system program) running in system mode performs the desired operation;

• once the handler finishes execution, the CPU resumes execution of the user program (back in user mode).



An **interrupt** is an event that causes the computer to stop what it is doing and (temporarily) do something else in such a way that processing of the original task can be seamlessly resumed. They:

• are used to signal events or conditions to the computer hardware (the CPU) outside of the normal execution cycle;

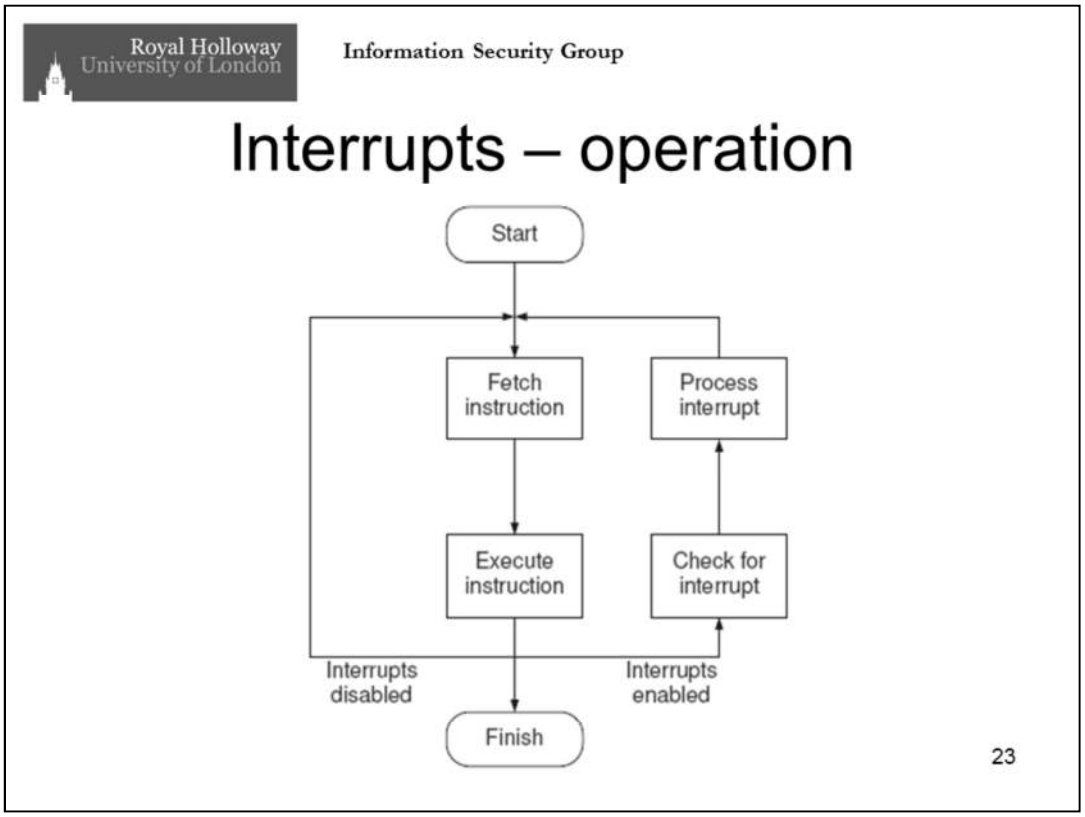
• may be generated by hardware or software;

• are hardware-specific.

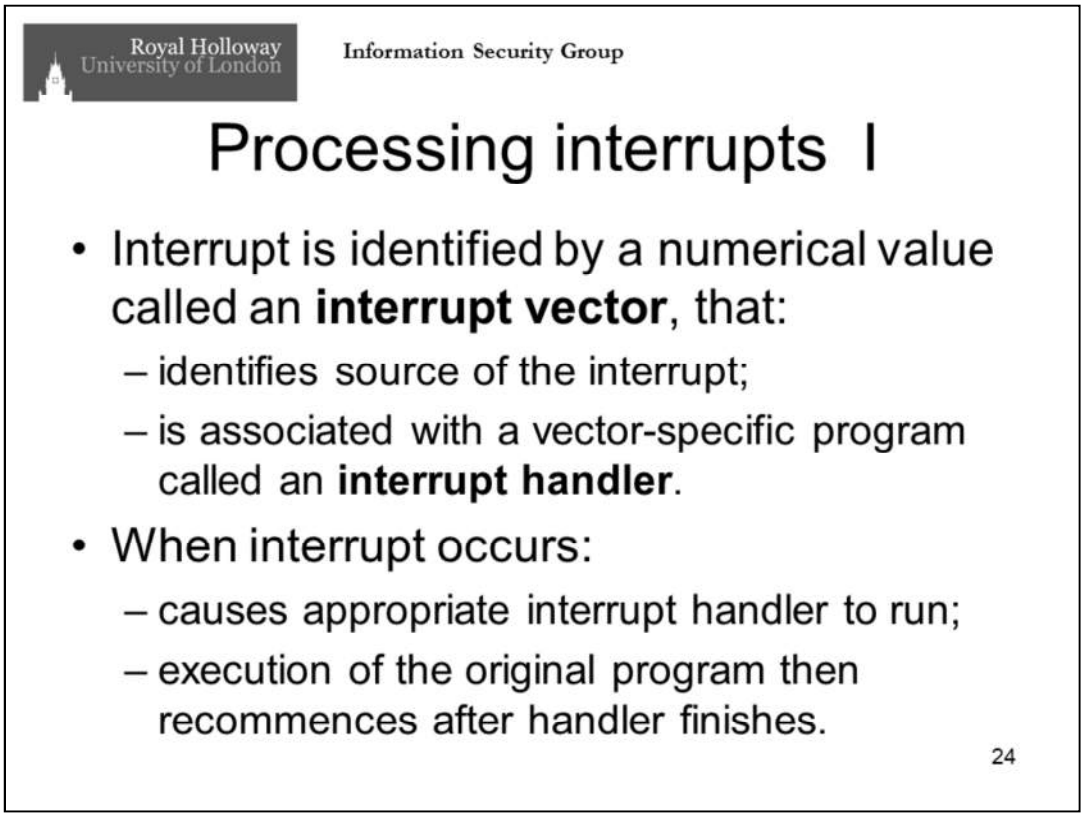
Interrupts are said to be **asynchronous**:

• that is, it is not predictable when an interrupt will occur;

• provisions for interrupt processing have to be added to the basic instruction cycle of the processor.



The diagram summarises how interrupts work from the processor's perspective. Fetching and executing instructions form the 'normal' processing cycle.

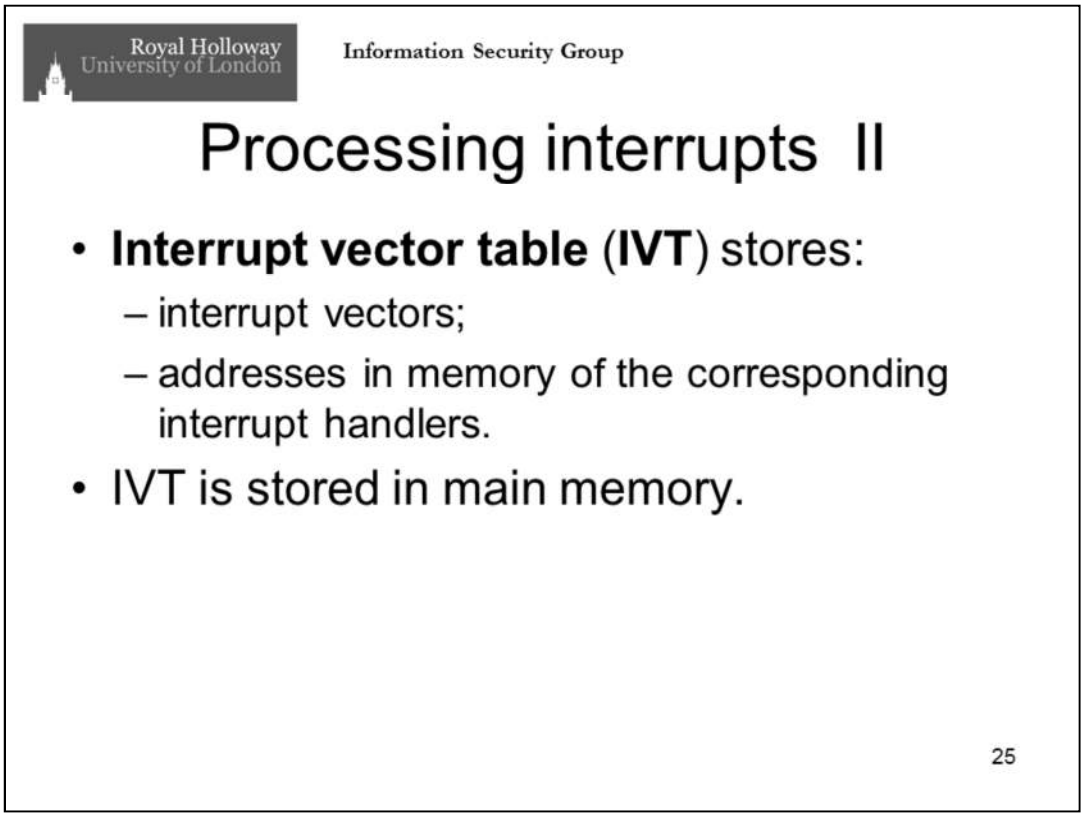


Every interrupt is identified by a numerical value called an **interrupt vector** that:

- identifies the source of the interrupt;
- is associated with a vector-specific program called an **interrupt handler**;

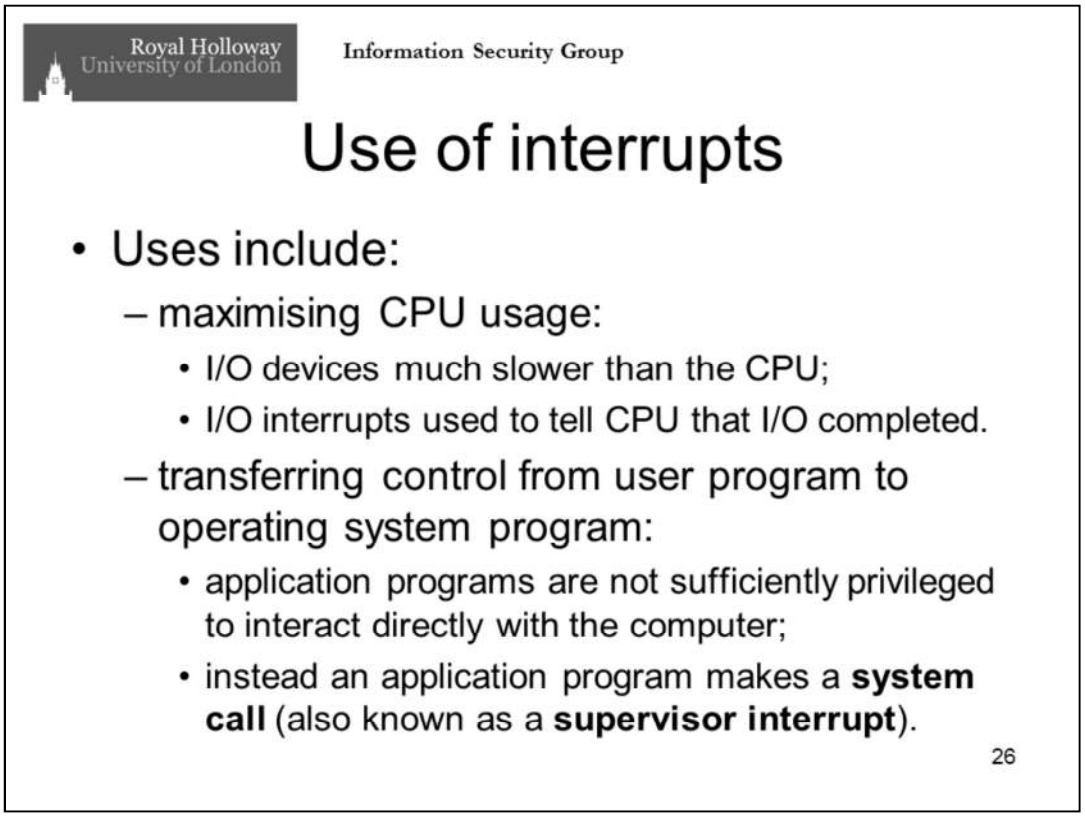
When the interrupt occurs:

- it causes the appropriate interrupt handler to be executed;
- execution of the original program then recommences once execution of the handler is completed.



The **Interrupt vector table** (**IVT**) is used to store the interrupt vectors and the addresses in memory of the corresponding interrupt handlers.

The IVT, and the interrupt handlers (i.e. the pieces of software run when an interrupt occurs), are stored in main memory, in an area reserved for OS use.



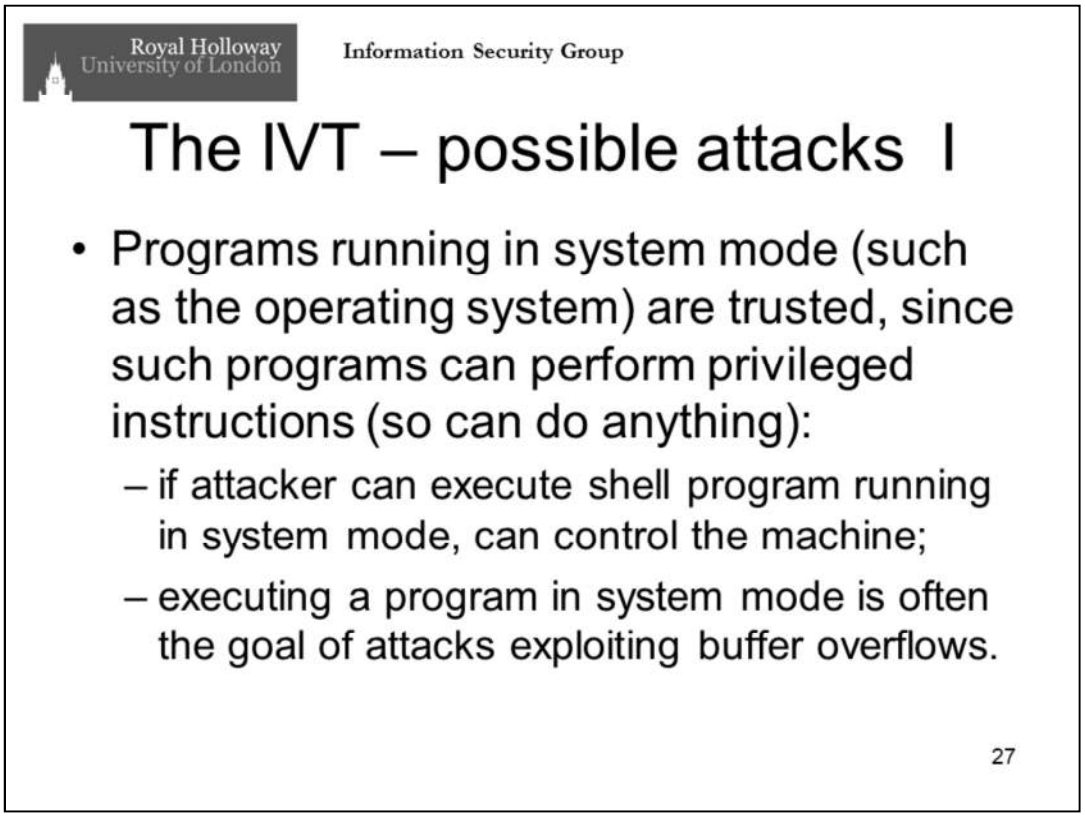
Uses of interrupts include:

• maximising CPU usage (i.e. so that processing of input/output can be interleaved with running programs):

- I/O devices are much slower than the CPU;
- I/O interrupts are used to tell the CPU that I/O has completed.

• transferring control from a user program to an operating system program:

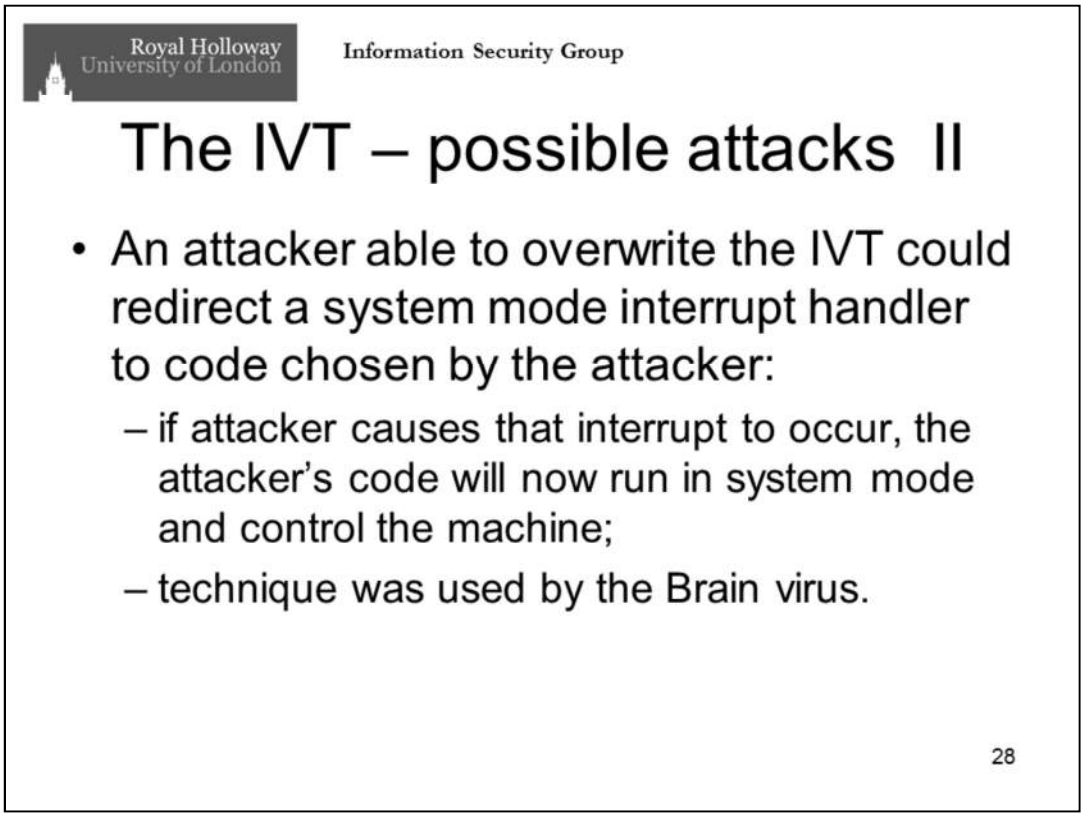
- application programs are not (or should not be) sufficiently privileged to interact directly with the computer;
- instead, as outlined on a previous slide, an application program must make a **system call** (also known as a **supervisor interrupt**).



Programs running in system mode (such as the operating system) are trusted, since such programs can perform privileged instructions (and can therefore do anything):

• if an attacker can execute a shell program running in system mode he can completely control the machine;

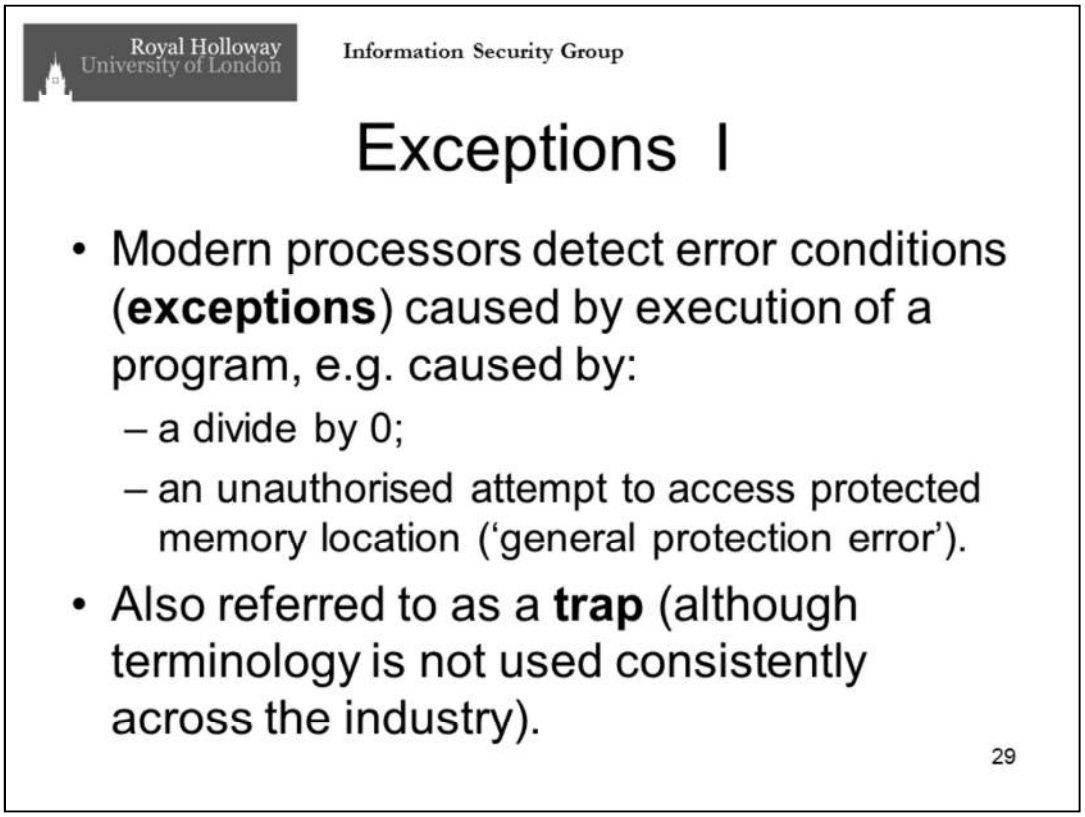
• executing a program in system mode is often the goal of attacks that exploit buffer overflows.



An attacker able to overwrite the IVT could redirect a system mode interrupt handler to code chosen by the attacker:

• if the attacker then causes that interrupt to occur, the attacker's code will now run in system mode and control the machine;

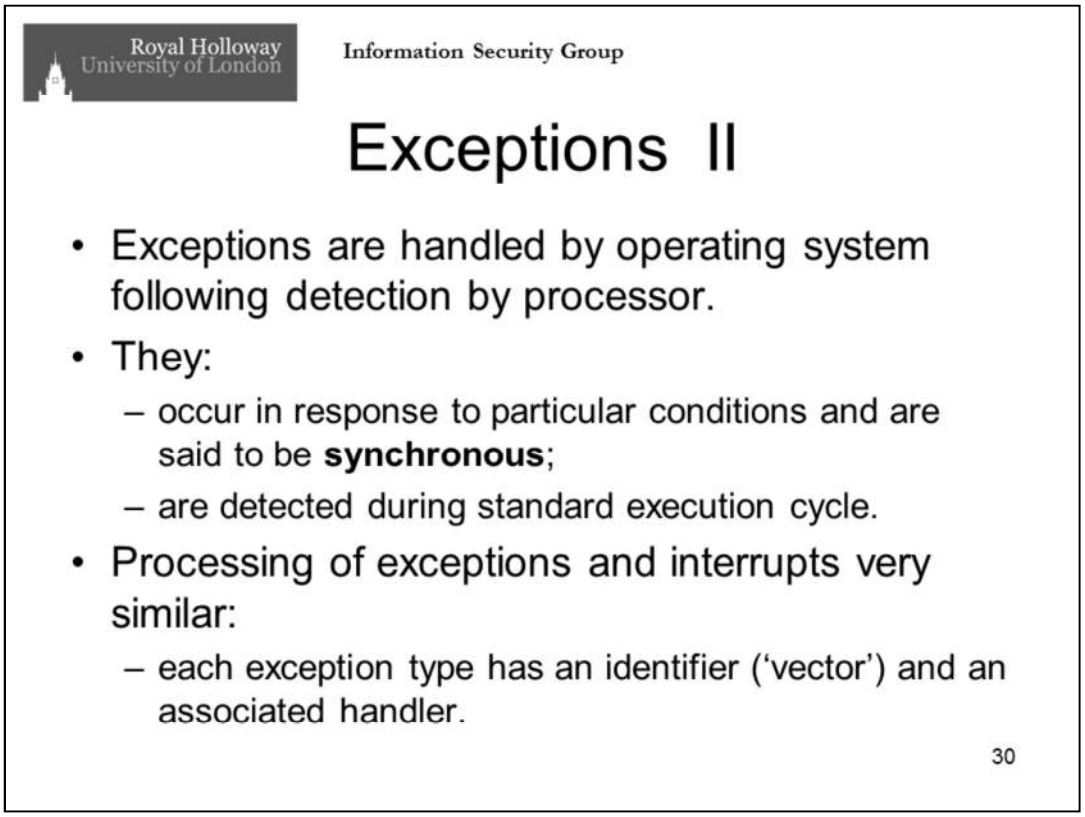
This technique was used by the Brain virus (and has been widely used by a range of exploits of system vulnerabilities).



All modern processors detect error conditions known as **exceptions** caused by the execution of a program. Such errors might, for example, be caused by:

- an attempt to divide a number by 0 (division by zero is undefined);
- an unauthorised attempt to access a protected memory location (giving rise to a 'general protection error').

Exceptions are also referred to as **traps**. There are wide variations in usage; for example, sometimes *trap* is used to refer to any interrupt, sometimes to any synchronous interrupt, sometimes to any interrupt not associated with input/output, and sometimes only to interrupts caused by instructions with trap in their names.

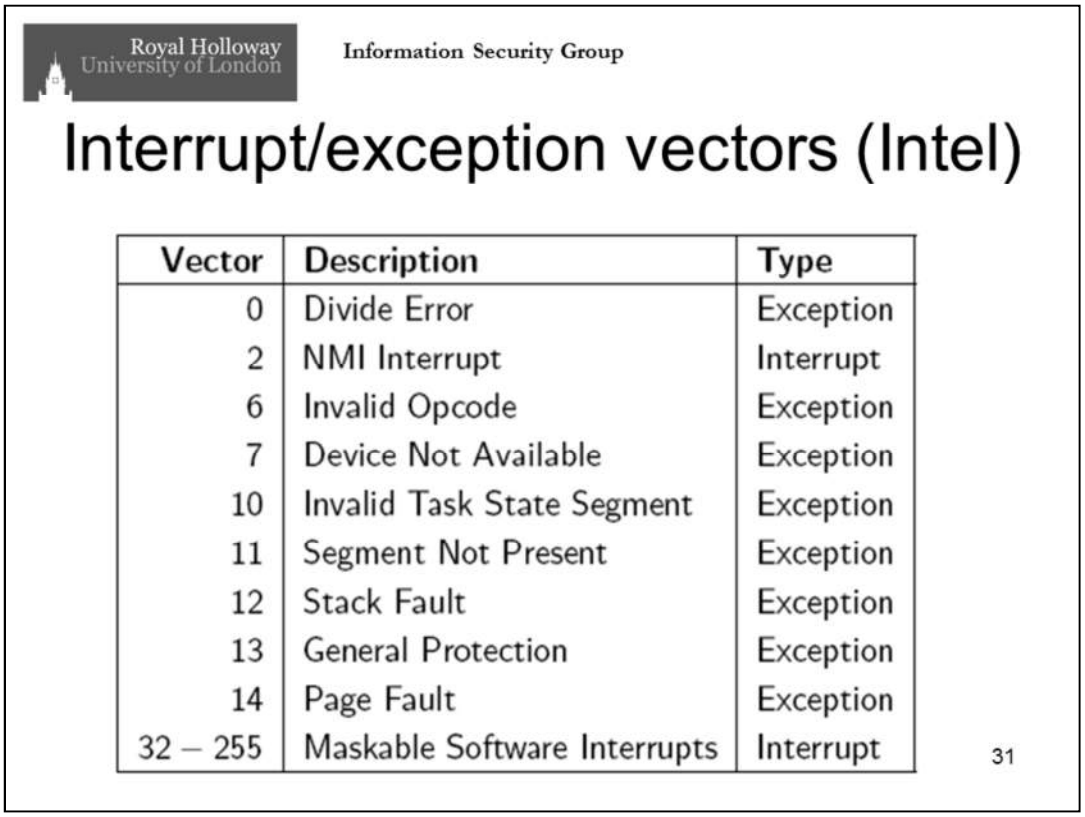


Exceptions are handled by the operating system following detection by the processor. They:

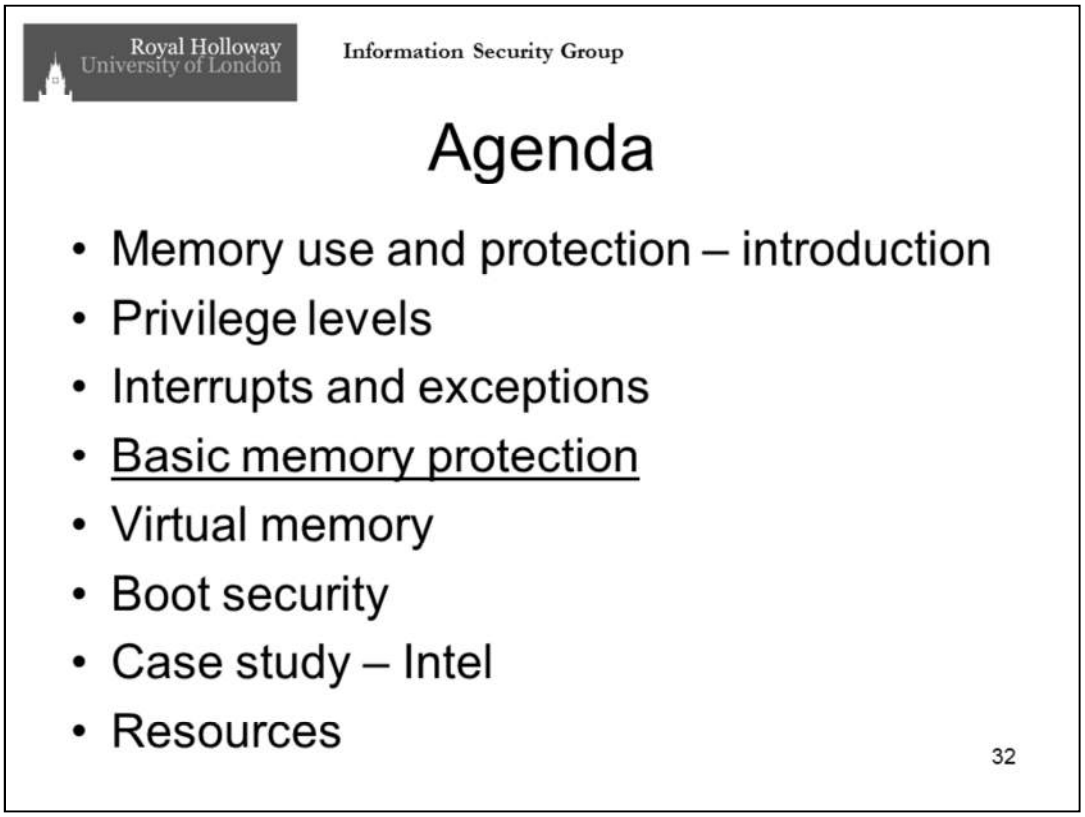
- occur in response to particular conditions and are said to be **synchronous**;
- are detected during the standard execution cycle.

Processing of exceptions and interrupts is very similar:

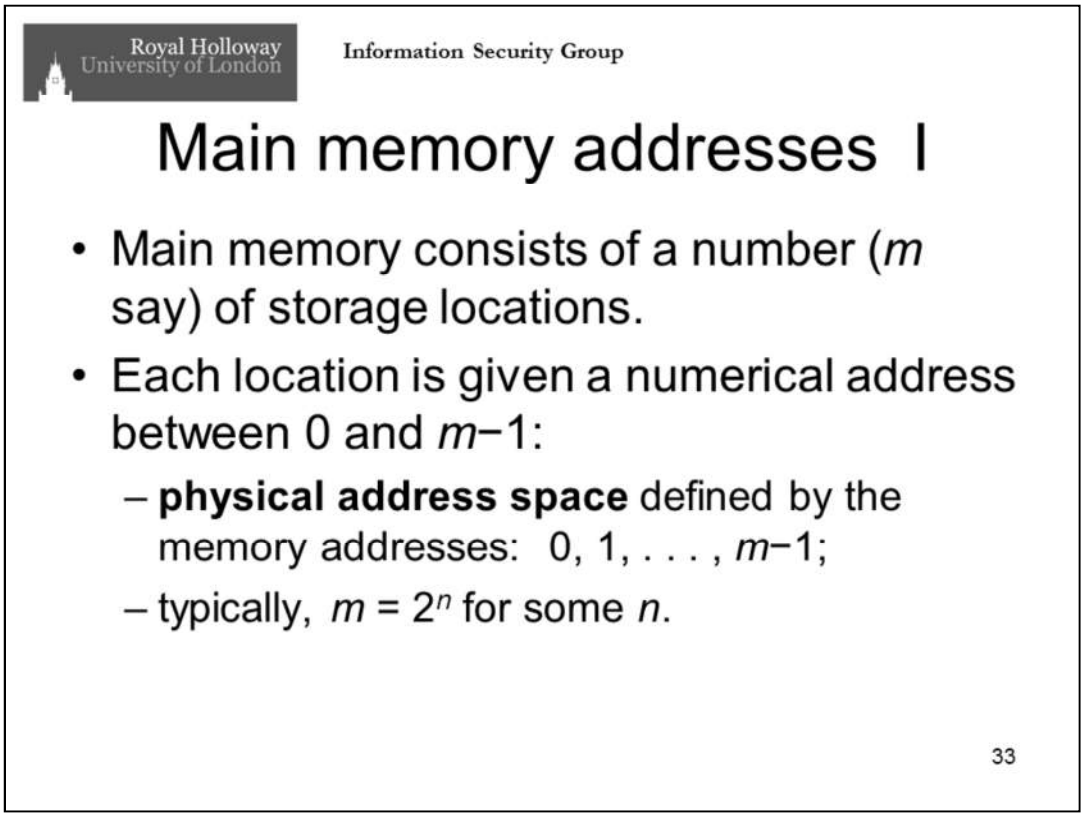
- each type of exception has an identifier ('vector"' and an associated handler, i.e. a piece of software which is executed when the relevant exception occurs.



As an example, the table specifies interrupt and exception vectors for the Intel Pentium processor family. Please note that this table is not examinable!



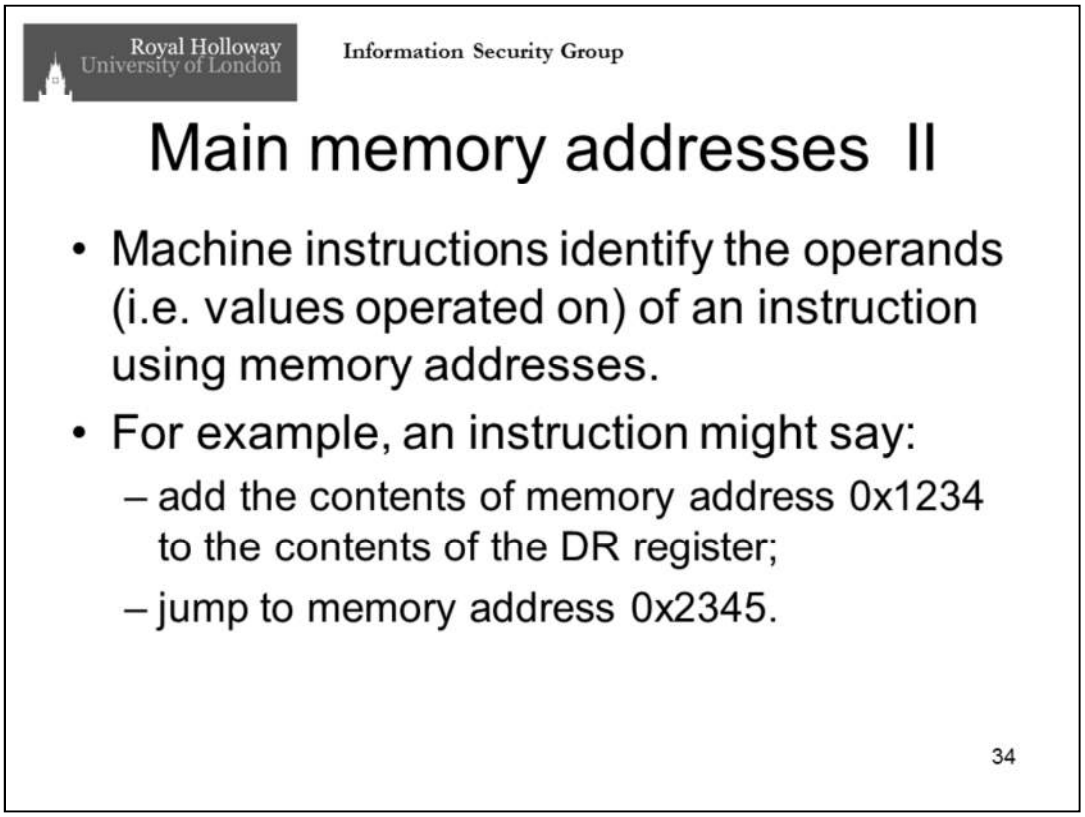
We now consider the main strategies that can be used to protect computer memory. We start in this section by examining 'historical' memory protection strategies, now replaced by virtual memory management based on paging (as described in the next section).



The main memory of a computer consists of a number (m say) of storage locations. Each location is given a numerical address between 0 and $m-1$:

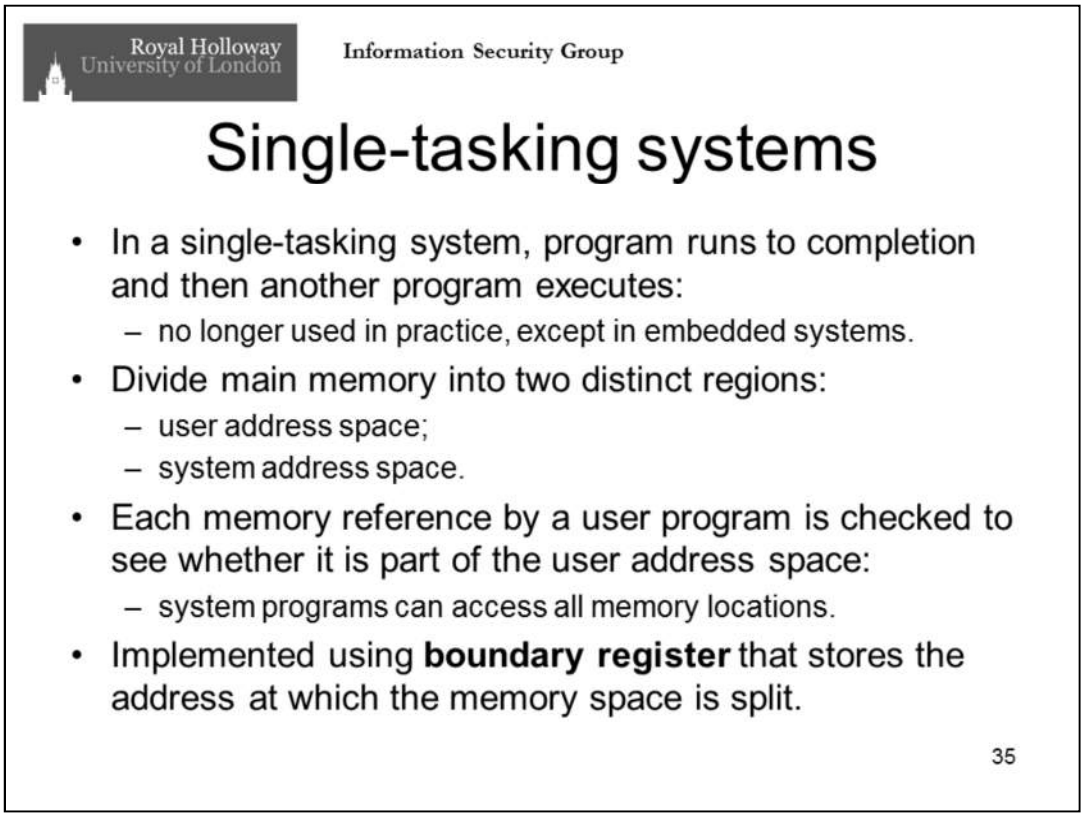
- the **physical address space** is defined by the list of memory addresses: 0, 1, . . . , $m-1$;
- typically, $m = 2^n$ for some n.

Typically, even though a computer **word** consists of a number of bytes (4 for 32-bit machines and 8 for 64-bit machines), each byte is given a unique address. Since a computer instruction may take up one or more words, the program counter will actually be increased by more than one as it executes an instruction.



Machine instructions identify the operands (i.e. the values being operated on) of an instruction using memory addresses. For example, a machine instruction might specify:

- add the contents of memory address 0x1234 to the contents of the DR register;
- jump to memory address 0x2345 (i.e. set the program counter to 0x2345).



In a single-tasking system, a program runs until completion and then another program can execute:

• such an approach is no longer used in practice, at least apart from in some simple embedded systems.

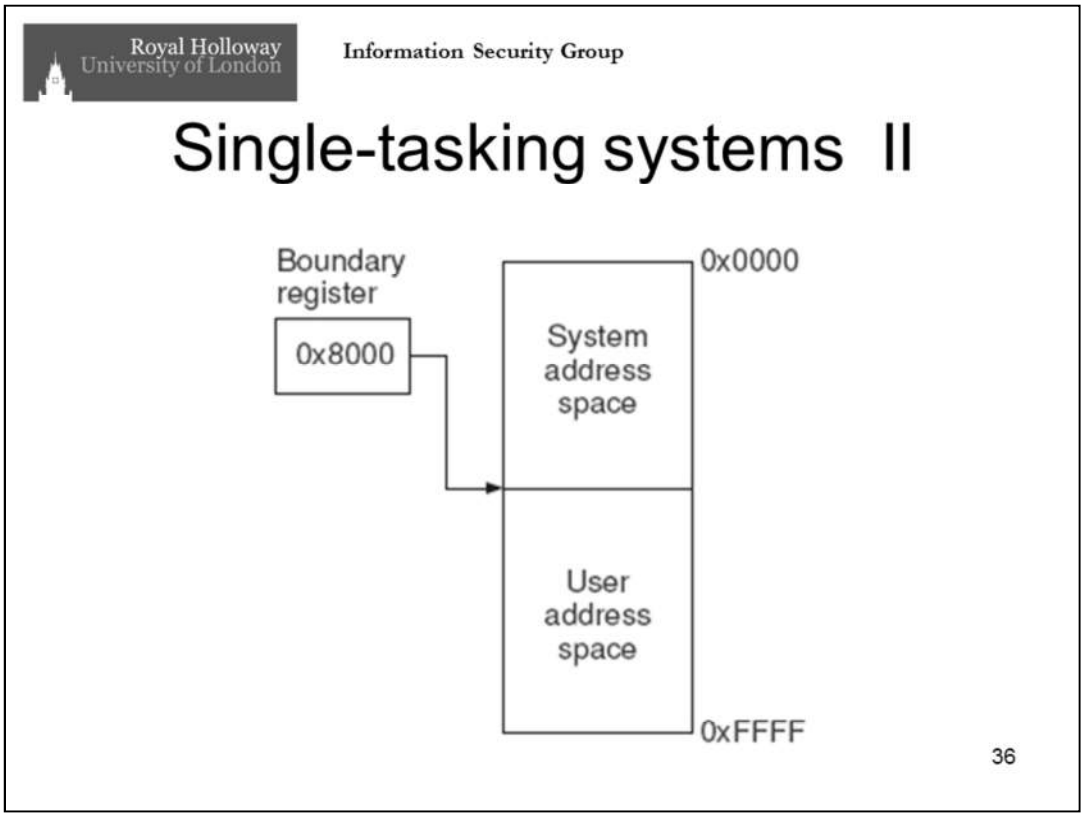
In such a system, memory protection is very simple. The main memory can be logically partitioned into two distinct regions:

• the user address space; and

• the system address space.

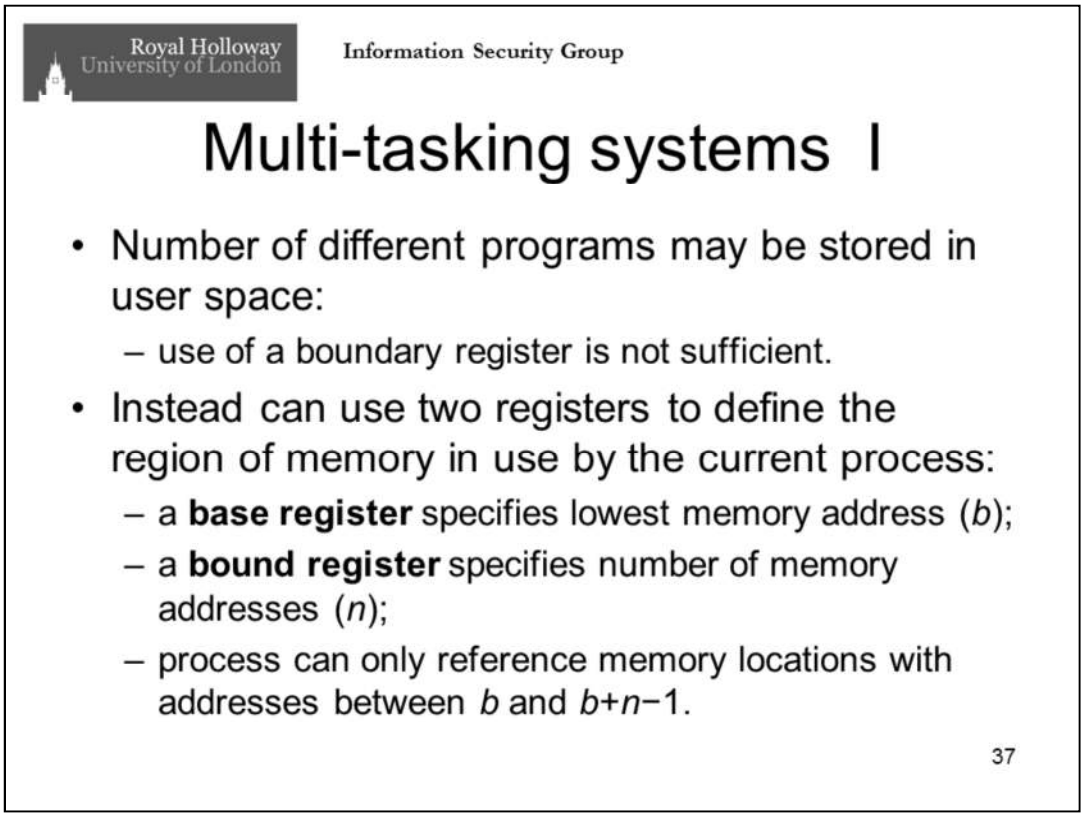
Each memory reference by a user program is checked to see whether it is part of the user address space (if not, then access is denied). System programs can access all the memory locations.

Such an approach can be implemented using a **boundary register** that stores the address at which the memory space is divided into two regions.



In the example shown, an attempt by a user program to access memory location:

- 0x89AB would be allowed;
- 0x1234 would generate a (memory) protection error.



In a multi-tasking system, a number of different programs may be stored in the user space. In such a case the use of a boundary register is not sufficient to provide the required degree of memory protection.

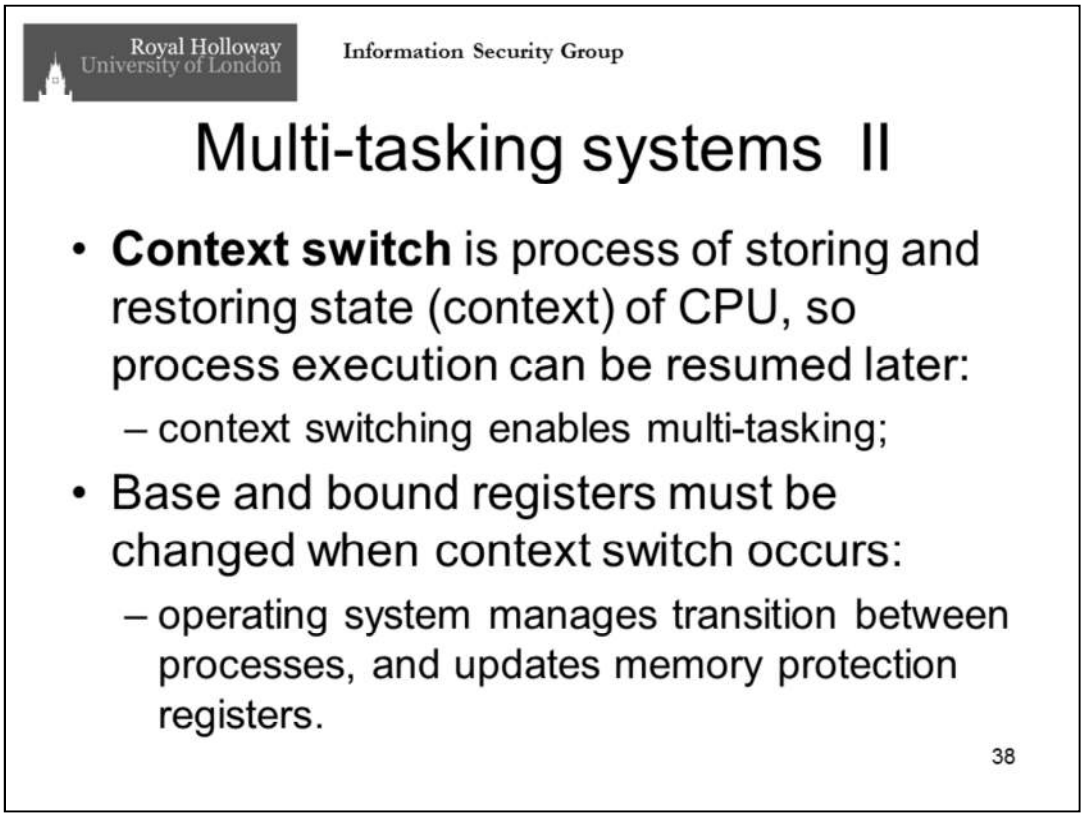
Instead, a simple approach involves using two registers to define the region of memory in use by the current process:

• a **base register** specifies the lowest memory address (*b*);

• a **bound register** specifies the number of memory addresses (*n*) allocated to this process, i.e. the length of the process in memory;

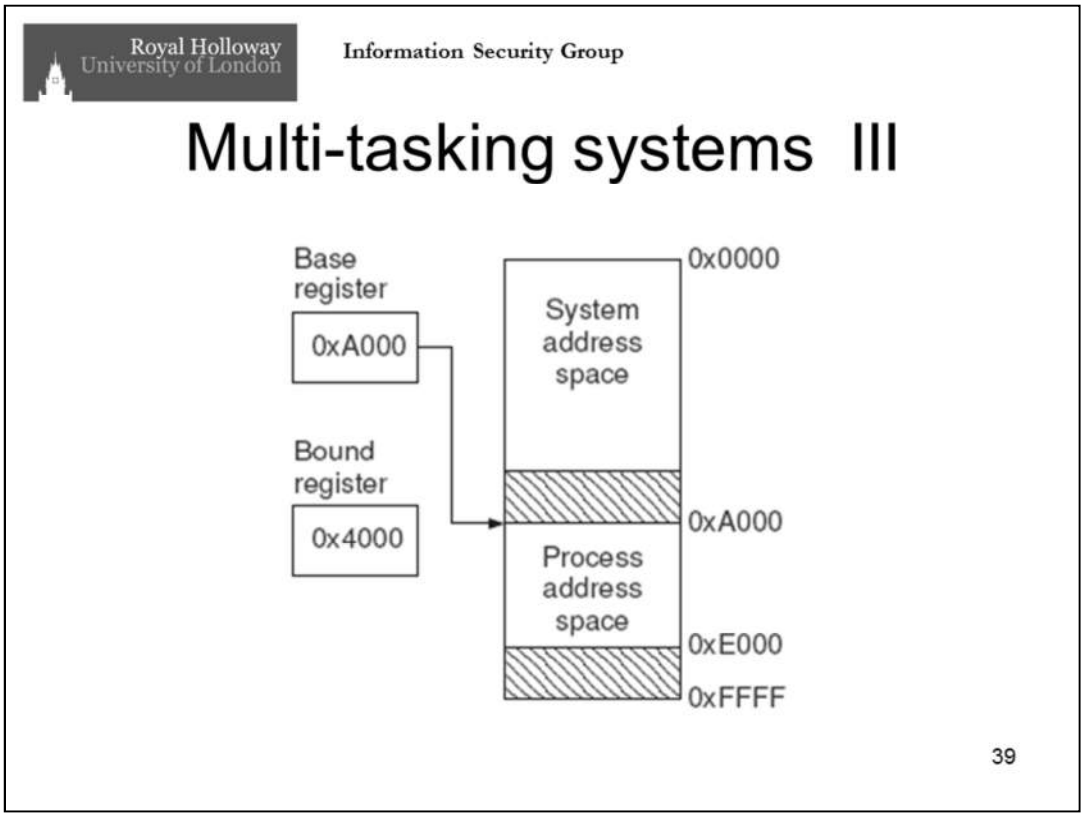
• the process can only reference memory locations with addresses between *b* and $b+n-1$.

In this simple means of memory allocation, each process must be given contiguous memory space, i.e. the memory allocated to a process must all be in one block.



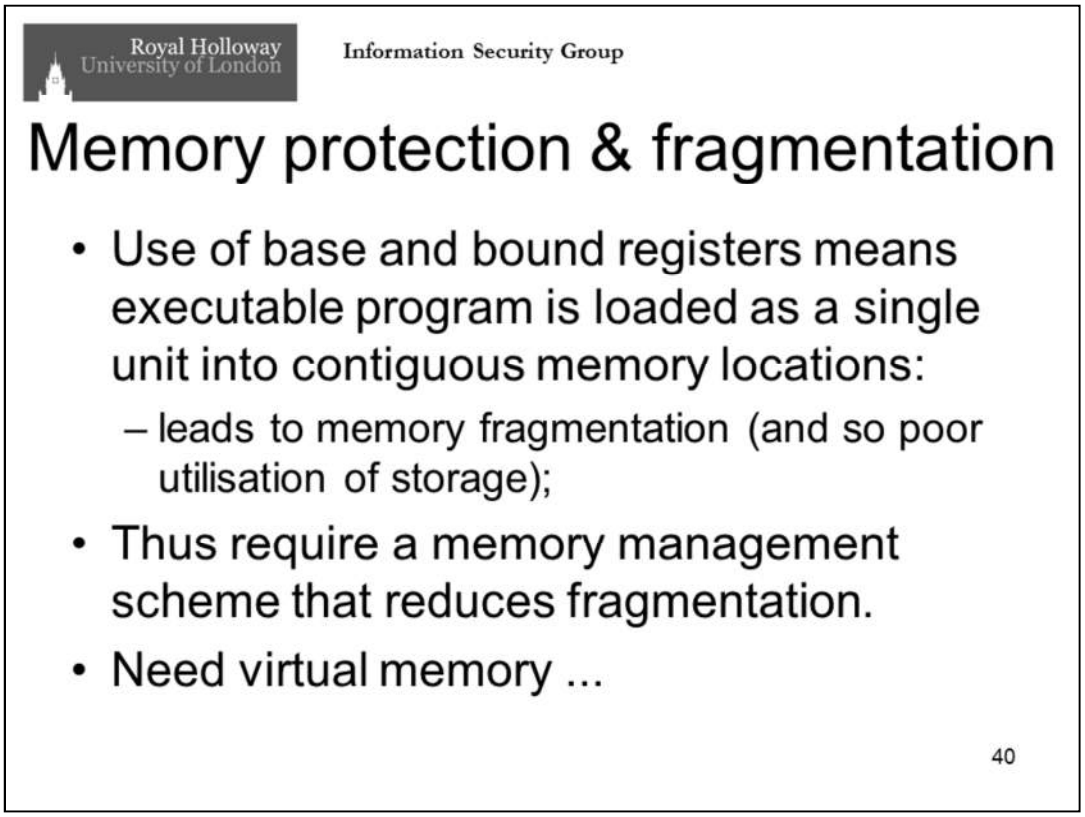
A **context switch** is the process of storing and restoring the state (context) of a CPU so that execution can be resumed from the same point at a later time. Context switching enables multi-tasking.

In the simple model we just looked at for memory management for multitasking, the base and bound registers must be changed when a context switch occurs. The operating system has to manage the transition from one process to another, and update the memory protection registers accordingly.



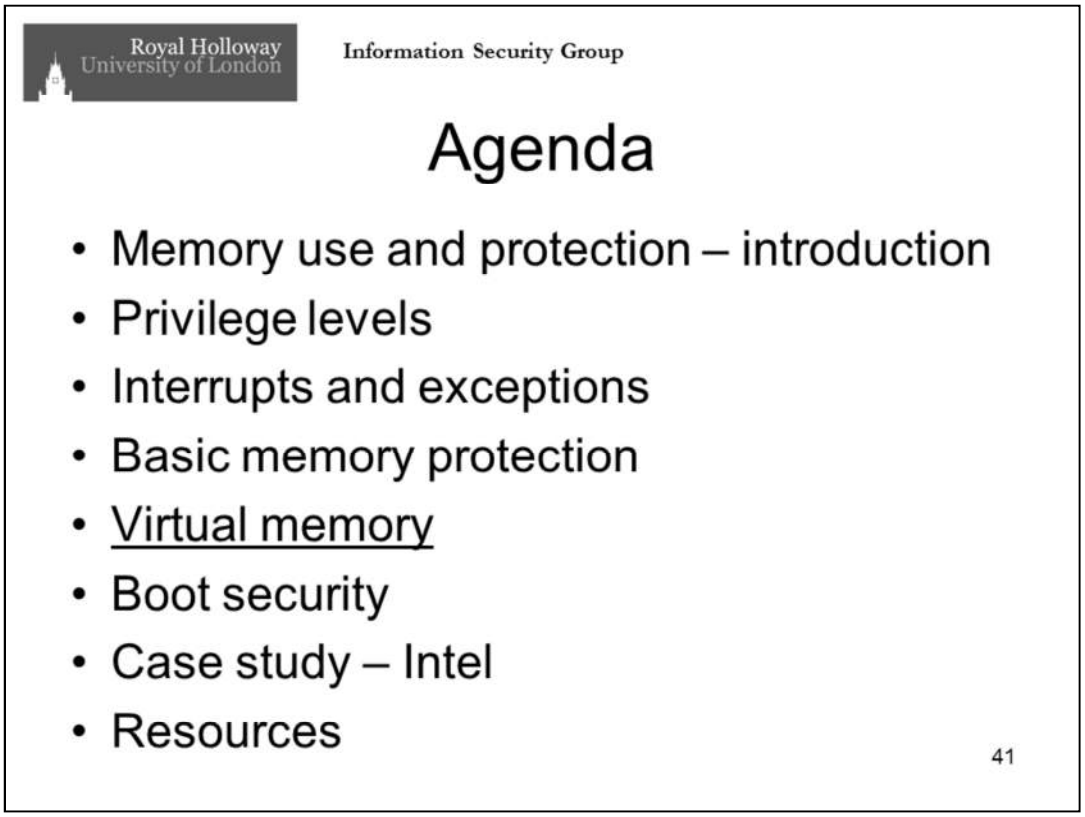
In this example, an attempt by a user program to access memory location:

- 0xABCD would be allowed;
- 0x89AB would generate a protection error (since this address is outside the process address space);
- 0x1234 would generate a protection error (since this address is outside the process address space).

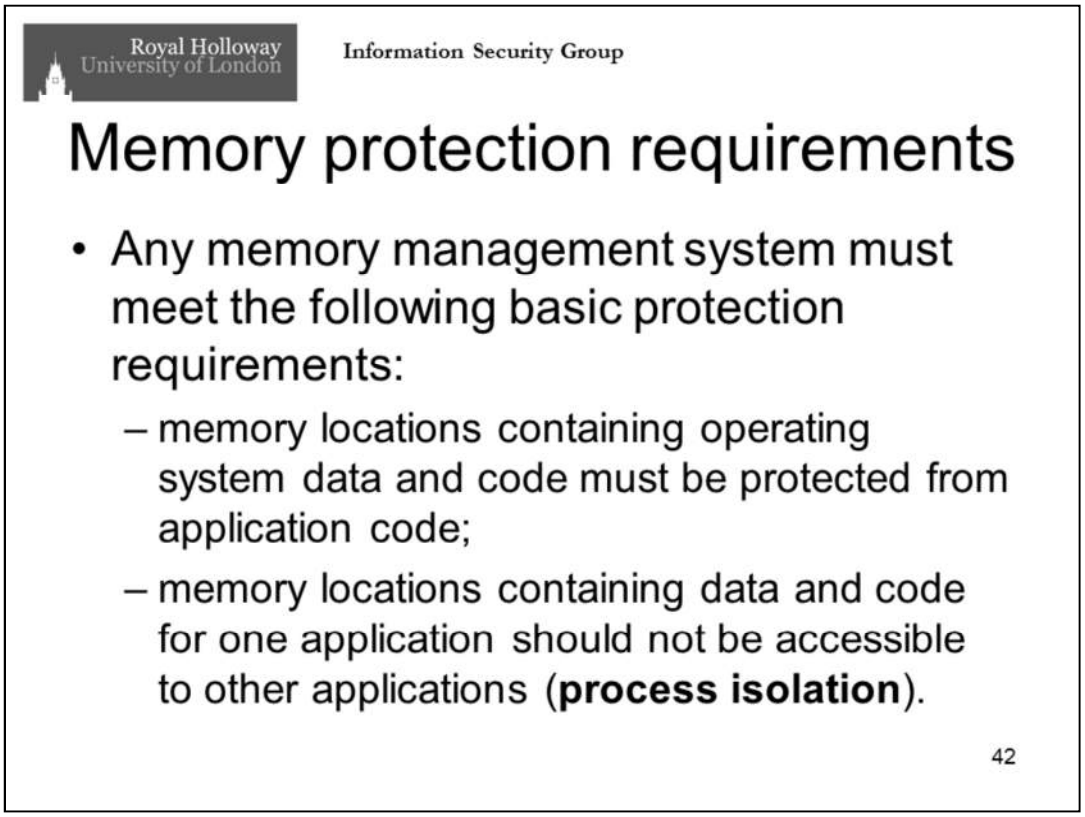


The use of base and bound registers requires the executable program to be loaded as a single unit into contiguous locations in main memory. Such an approach to storage allocation leads to fragmentation of the available memory space, and hence poor utilisation of storage.

As a result we require a memory management scheme that reduces fragmentation. This leads naturally to the next topic, i.e. **virtual memory**.



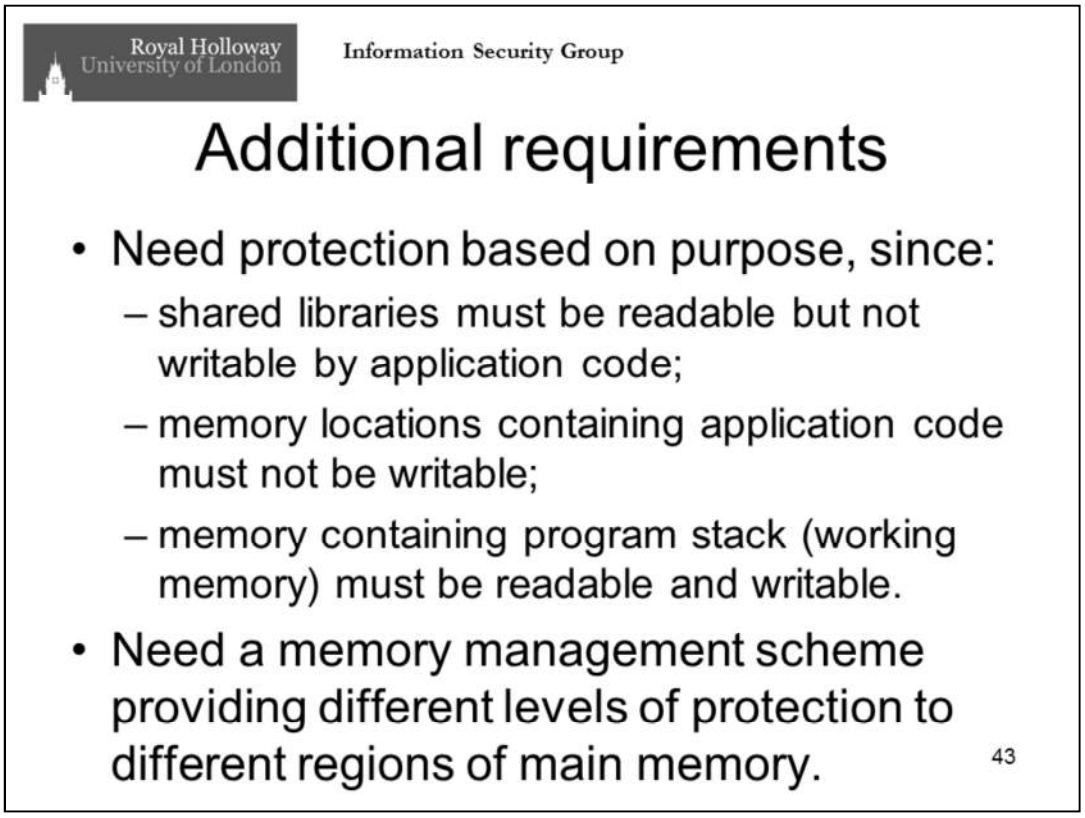
We now describe the means of memory management used in modern computers, namely virtual memory management, which relies on the use of paging and hardware address translation using the MMU (which itself relies on a table stored in main memory).



The basic requirements for memory protection are as follows:

• memory locations containing operating system data and code must be protected from application code;

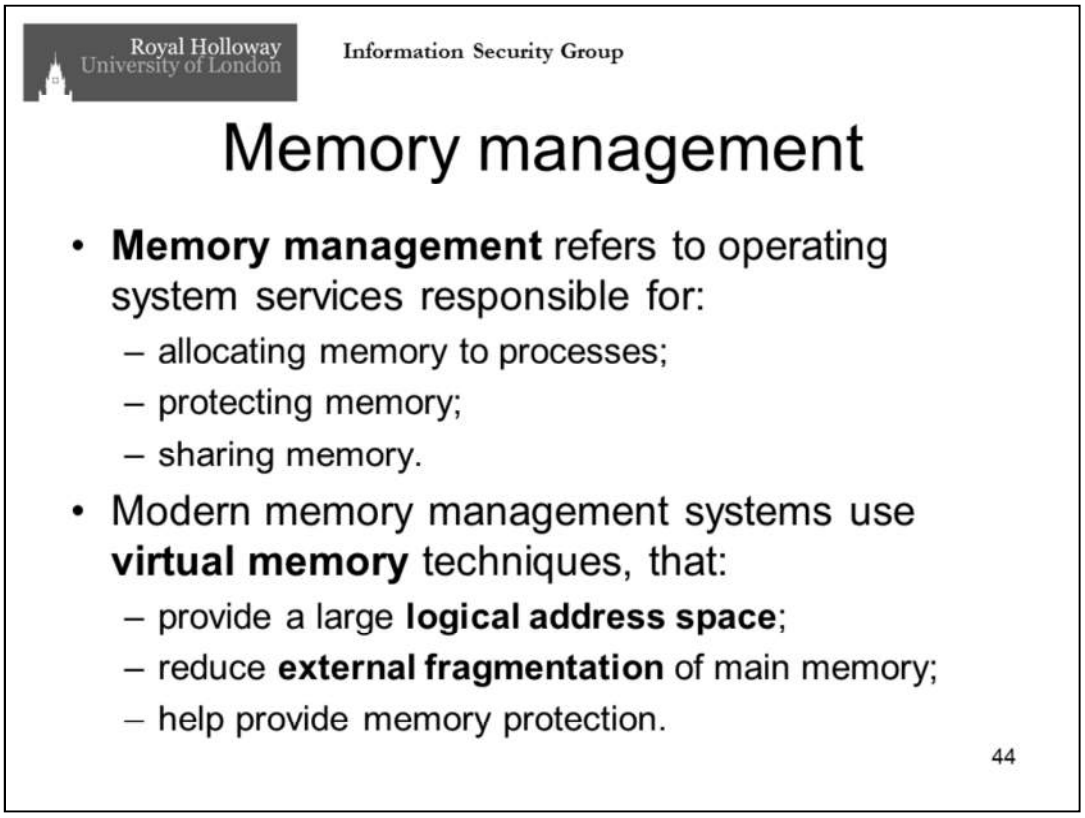
• memory locations containing data and code for one application should not be accessible to other applications (i.e. what is called **process isolation**).



For the following reasons, we need a memory protection scheme based on purpose:

- shared libraries must be readable but not writable by application code;
- memory locations containing application code must not be writable;
- memory locations containing the program stack (a type of working memory for a program) must be readable and writable.

As a result we need a memory management scheme that provides different levels of protection to different regions of main memory, according to the purpose for which the memory is being used.



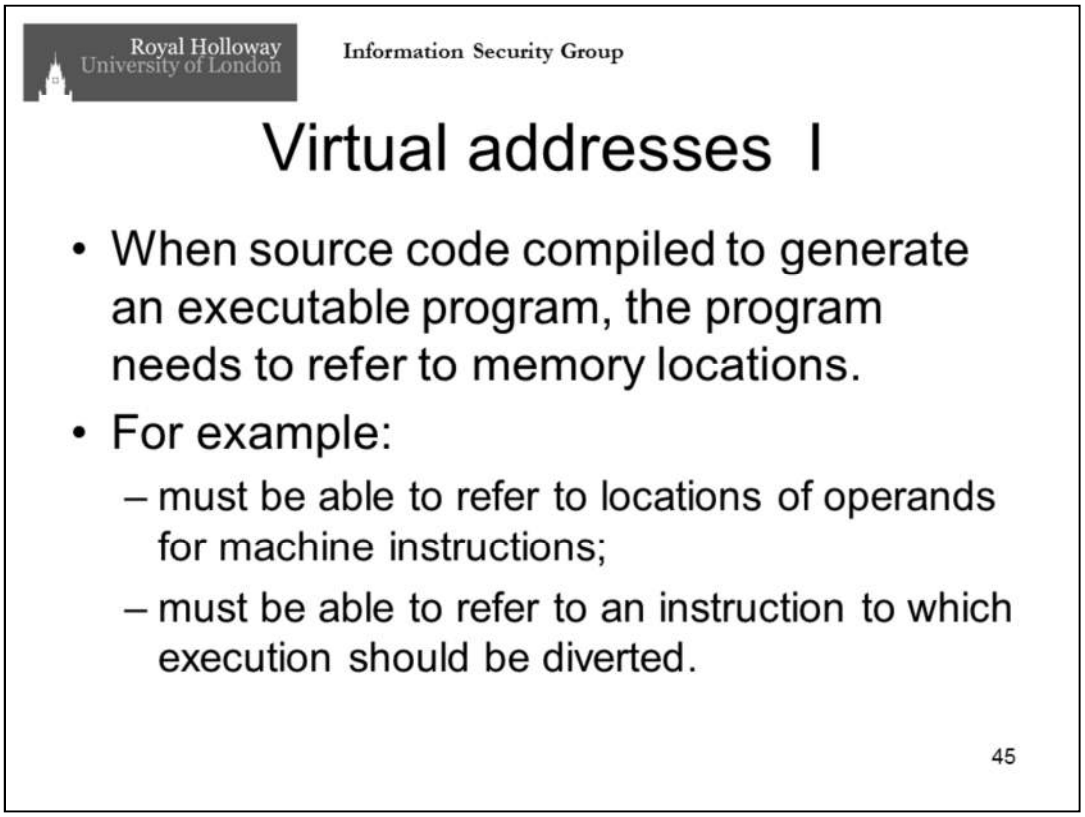
Memory management is a collection of operating system services that are jointly responsible for:

- the allocation of memory to processes;
- the protection of memory; and
- sharing of memory.

Modern memory management systems make use of virtual memory techniques, which help to:

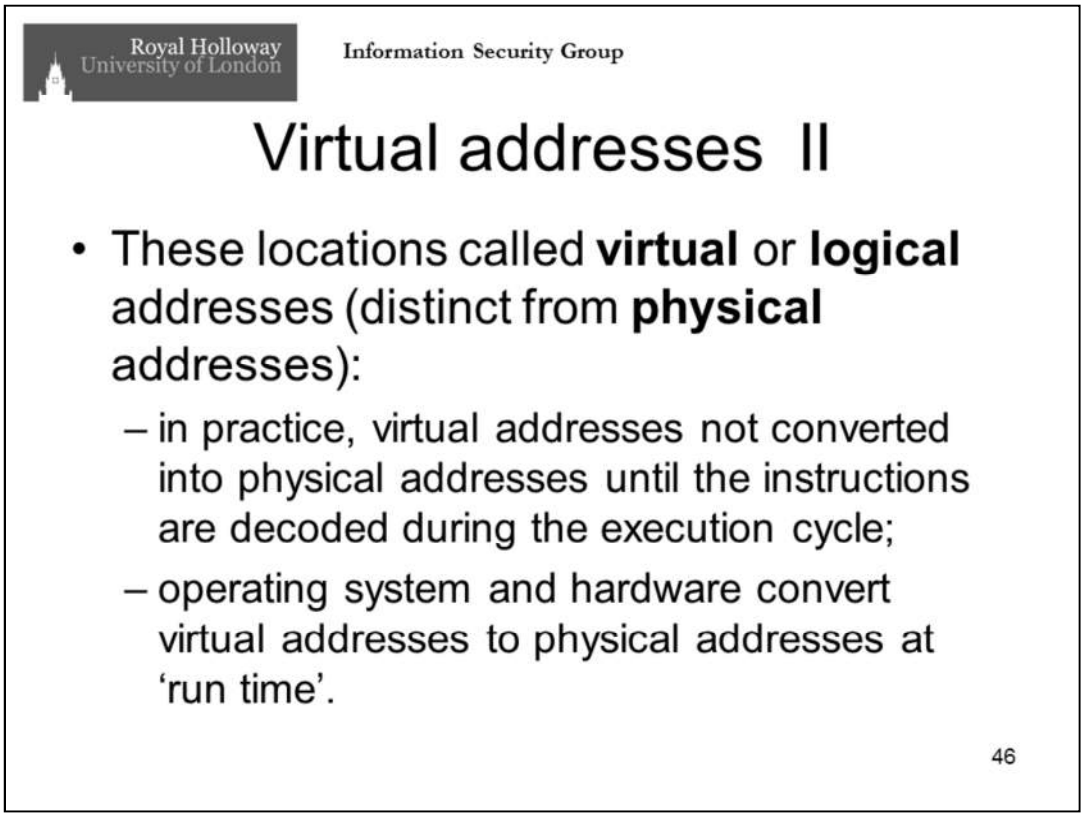
- provide a large **logical address** space;
- reduce **external fragmentation** of main memory;
- provide memory protection.

External fragmentation refers to the case where available (free) main memory becomes divided into many small pieces, which are too small to be used. It can occur if the memory allocation process allocates (and de-allocates) memory in variable size pieces.



When source code is compiled to generate an executable program, it is necessary for the executable program to refer to memory locations within the space allocated to it. For example, the program:

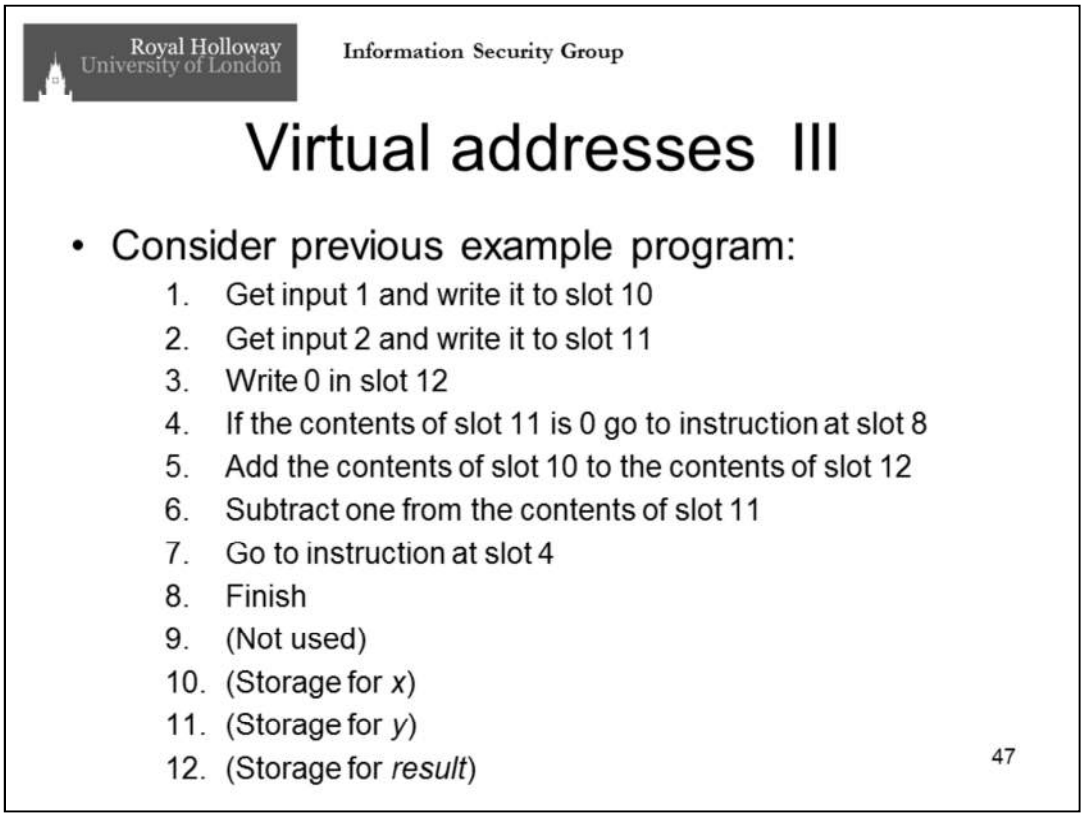
- must be able to specify the locations of operands for machine instructions;
- must be able to specify the location of an instruction to which execution should be diverted.



These locations are called **virtual** or **logical** addresses (which are distinct from **physical** addresses).

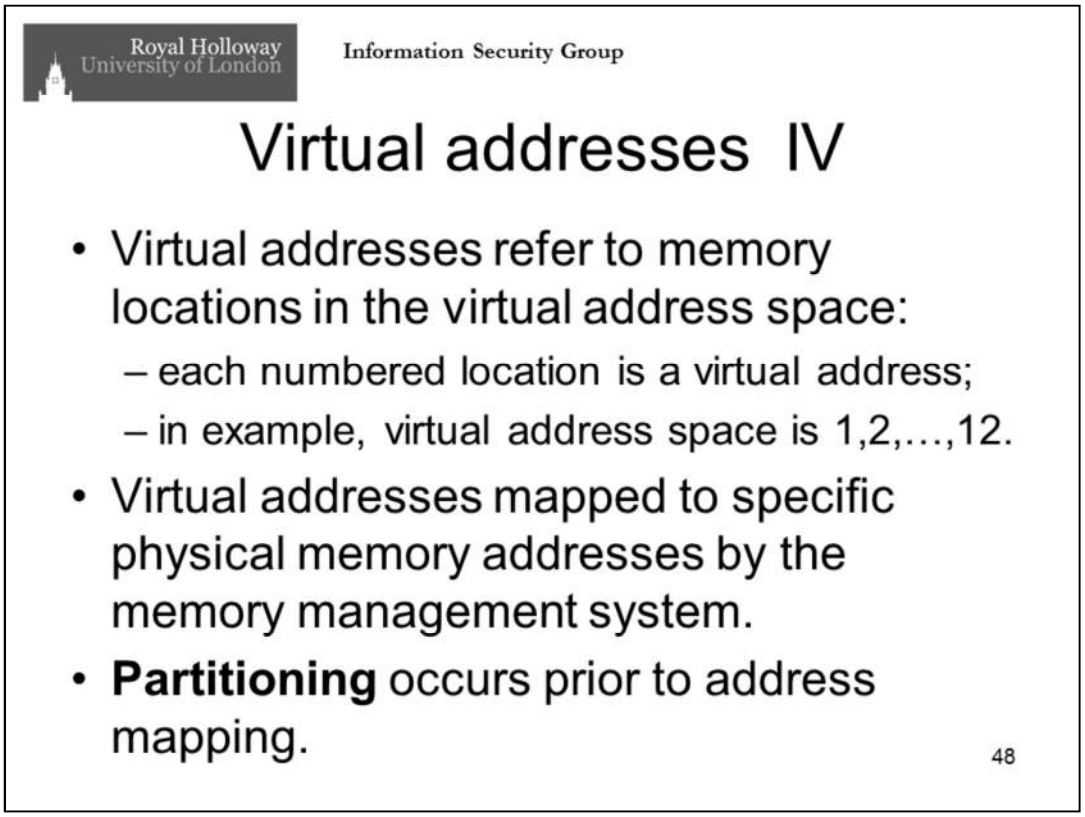
In practice, the virtual addresses are not converted into physical addresses until the instructions are decoded during the execution cycle. It is the job of the operating system and the hardware to correctly convert virtual addresses to physical addresses.

Address conversion is performed by special purpose hardware built into the CPU (the MMU), and it uses tables stored in the computer's main memory to do the conversions (translations).



Consider the previous example program:

1. Get input 1 and write it to slot 10
2. Get input 2 and write it to slot 11
3. Write 0 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*)

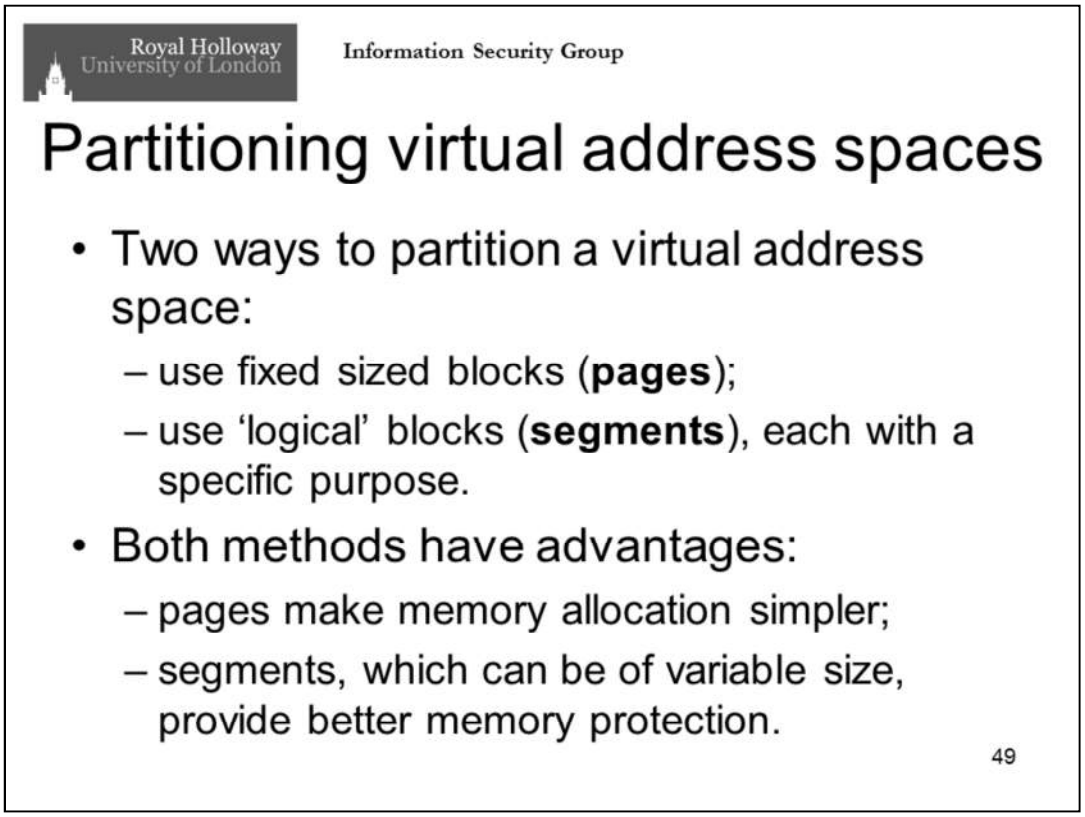


Virtual addresses refer to memory locations in the virtual address space:

- each numbered location is a virtual address;
- in this example the virtual address space is the sequence 1, 2, …, 12.

The virtual addresses are mapped to specific physical memory addresses (as the program is being executed) by the memory management system.

Program is typically **partitioned** (divided up) into blocks prior to mapping virtual addresses into physical addresses. Each block of program is then allocated a block of physical memory.



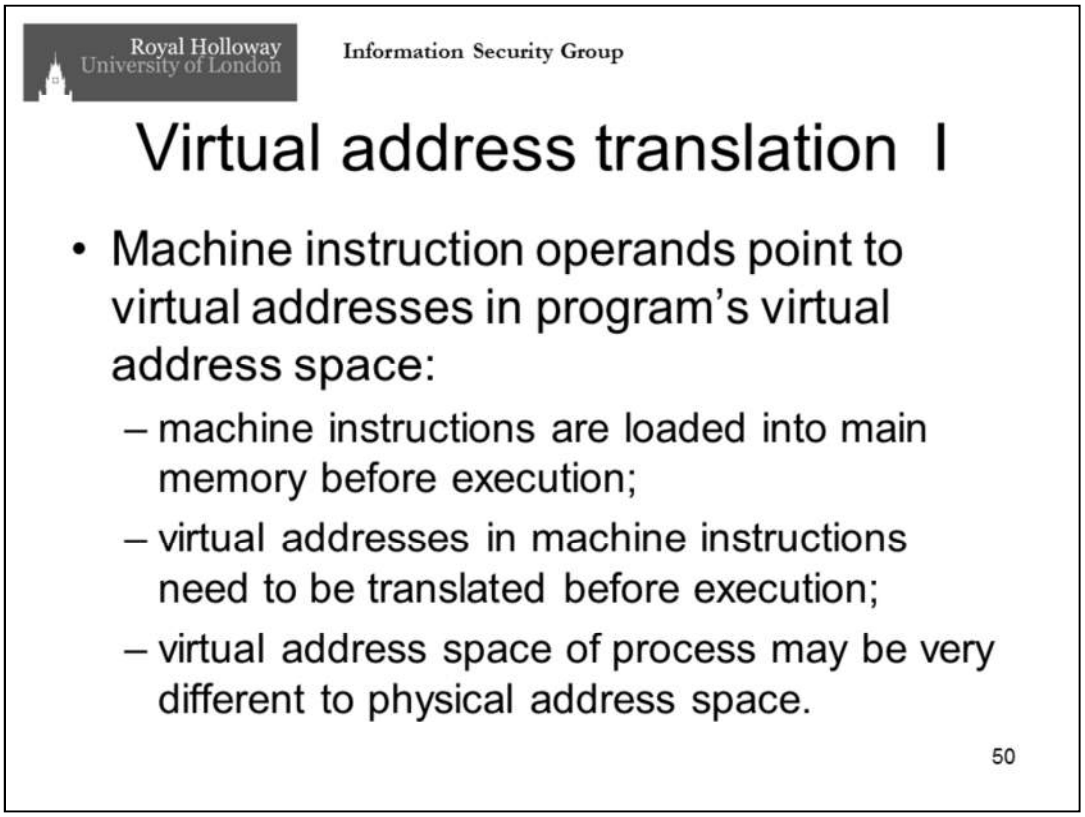
There are two widely used ways of partitioning a virtual address space, i.e. to allocate different portions of a program to different portions of physical memory. These are to:

- use fixed sized blocks (**pages**);
- use 'logical' blocks (**segments**), each with a specific purpose.

Both methods have advantages:

- pages make memory allocation simpler;
- segments, which can be of variable size, can potentially provide better memory protection.

It is possible to combine the two approaches (typically using segmentation first, where the segments are defined at the time of creating the executable code, and then paging).

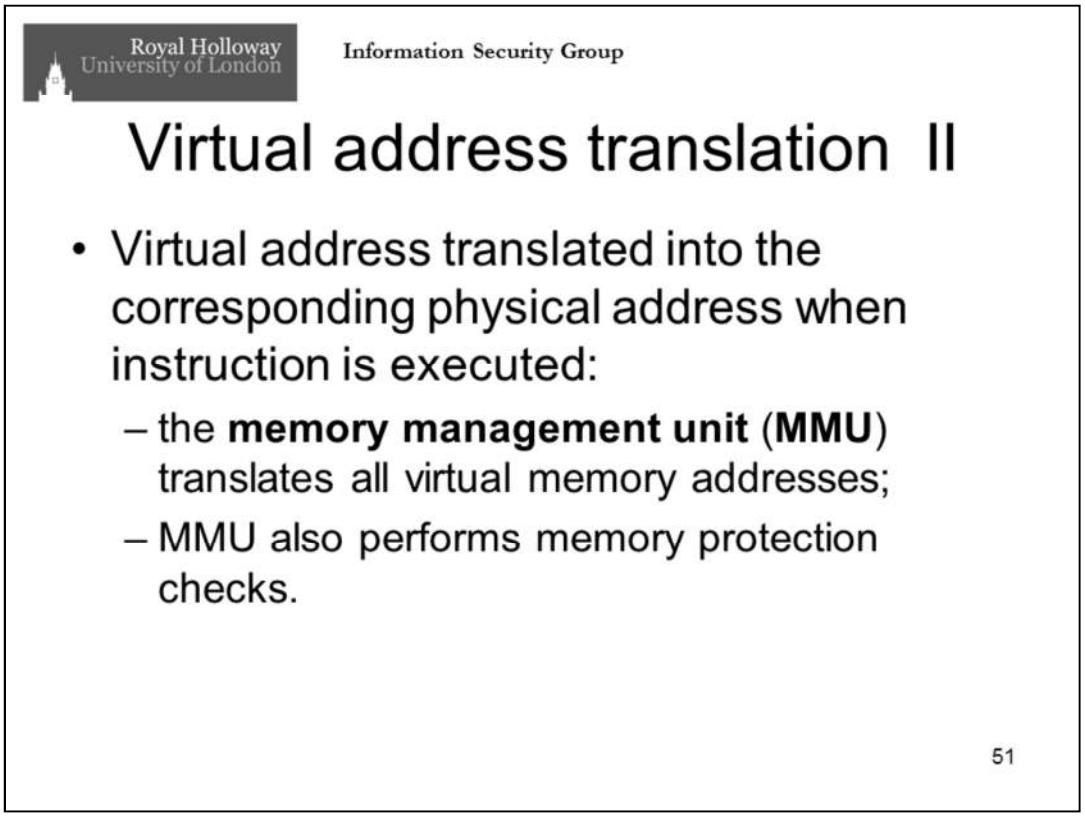


The operands of a machine instruction are references to virtual memory addresses (in a program's virtual address space):

• machine instructions are loaded into main memory before execution;

• the virtual addresses in machine instructions need to be translated before execution;

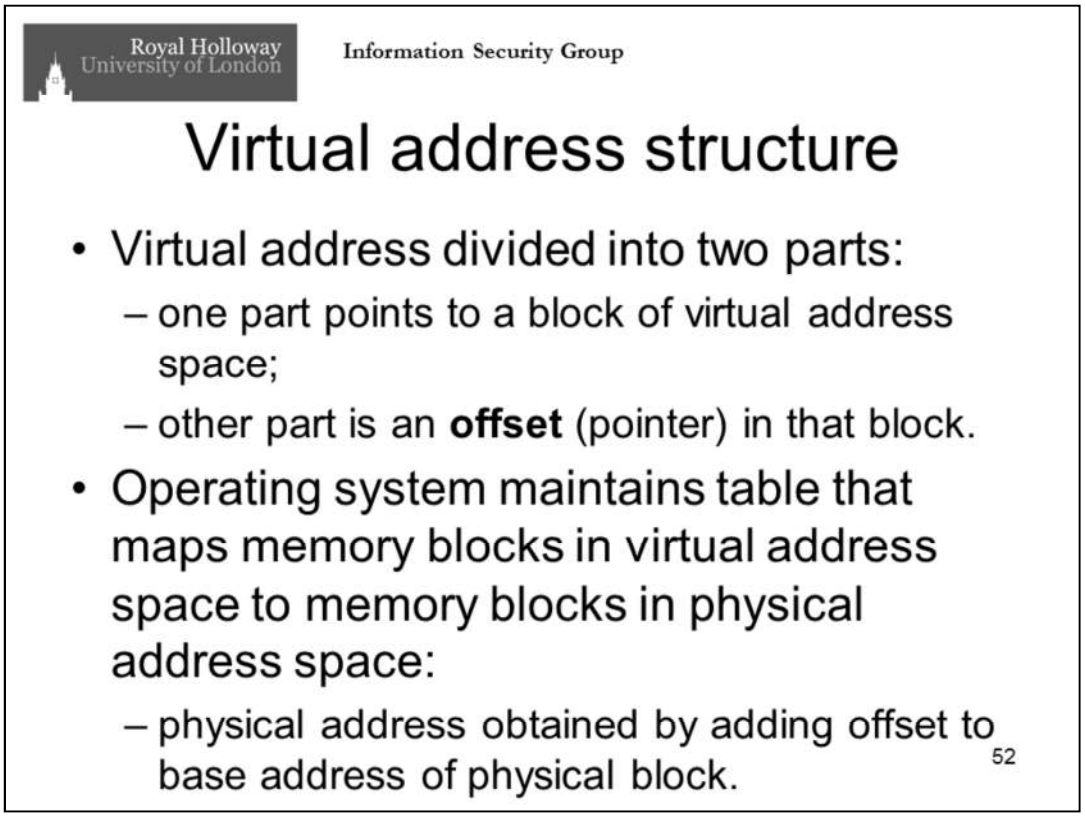
• the virtual address space of a process may be very different to the physical address space.



A virtual address is translated into the corresponding physical address when the instruction containing the address is executed:

• the **memory management unit** (**MMU**), part of the CPU, translates all virtual memory addresses;

• the MMU also performs memory protection checks.



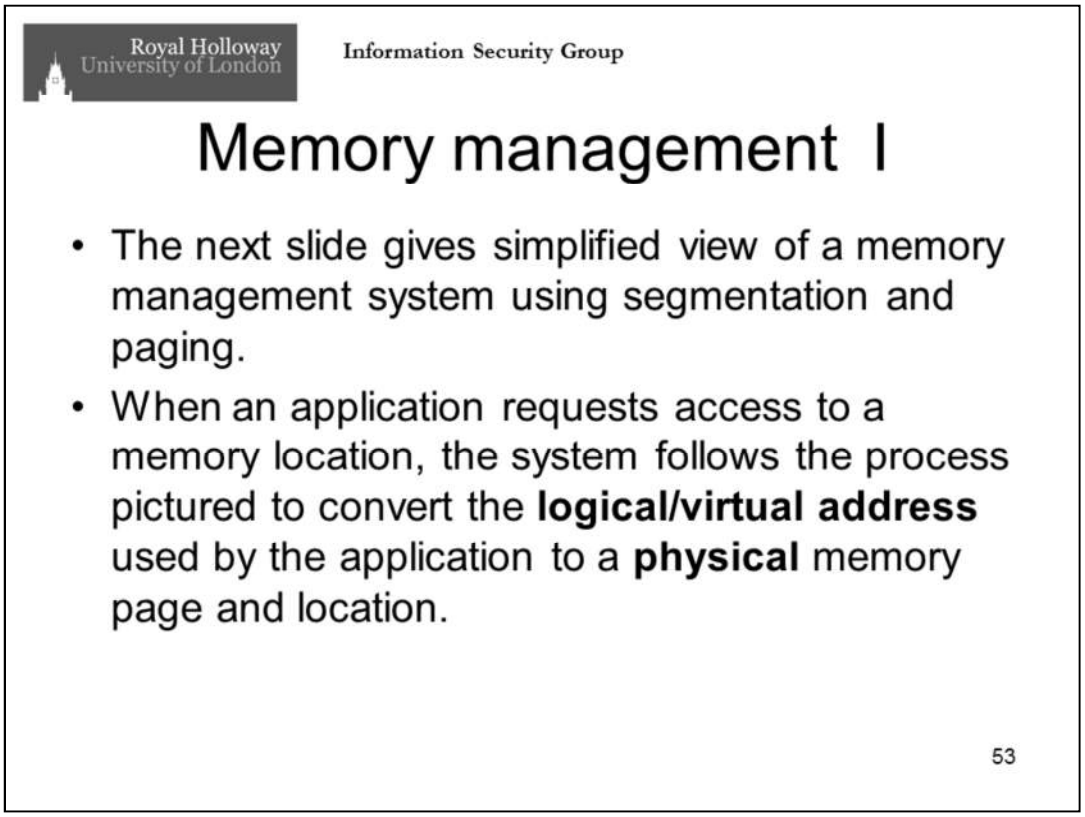
As part of the process of translation, a virtual address is divided into two parts:

- one part identifies a particular block of the virtual address space;
- the other part is equal to an *offset* (i.e. an address) within that block.

The operating system maintains a look-up table which maps blocks of memory in the virtual address space into blocks of memory in the physical address space:

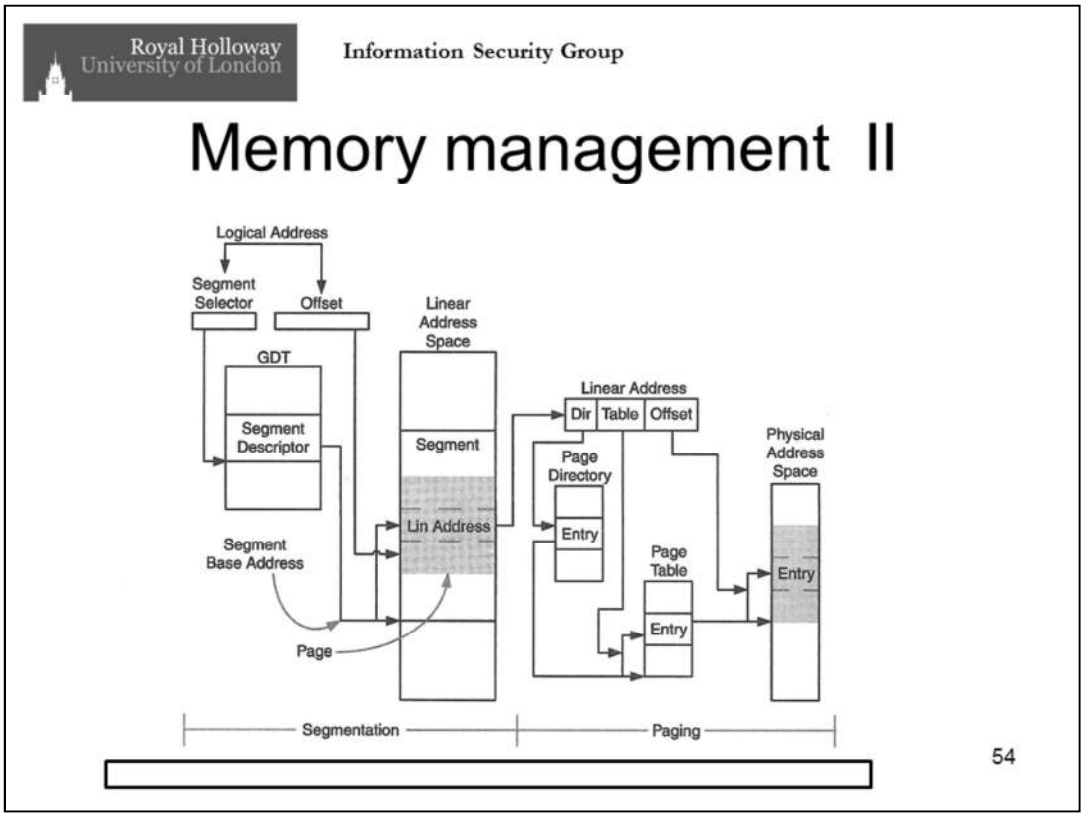
- a virtual address is converted into a physical address by finding the base address of the corresponding physical block and adding the offset.

This is all described in greater detail in the next few slides.



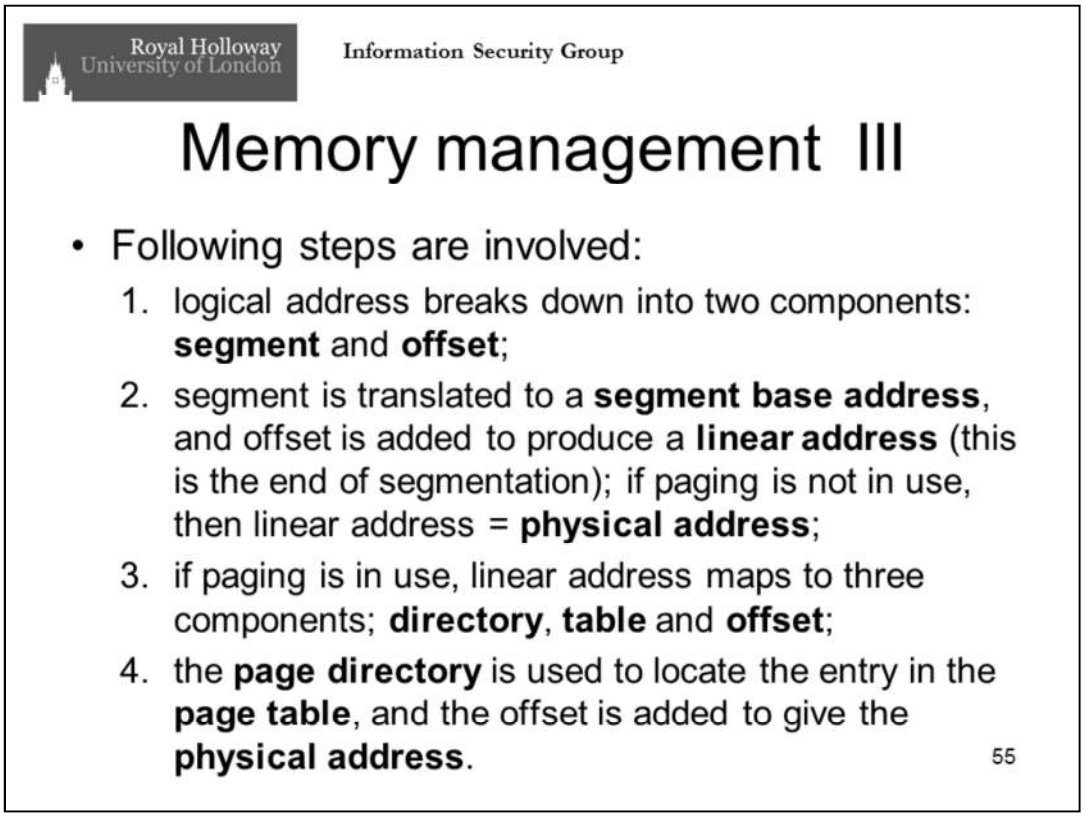
The next slide gives a simplified view of a memory management system using a combination of segmentation and paging.

As discussed previously, when an application requests access to a memory location, the system follows the process pictured to convert the **logical/virtual address** used by the application to a **physical** memory page and location.



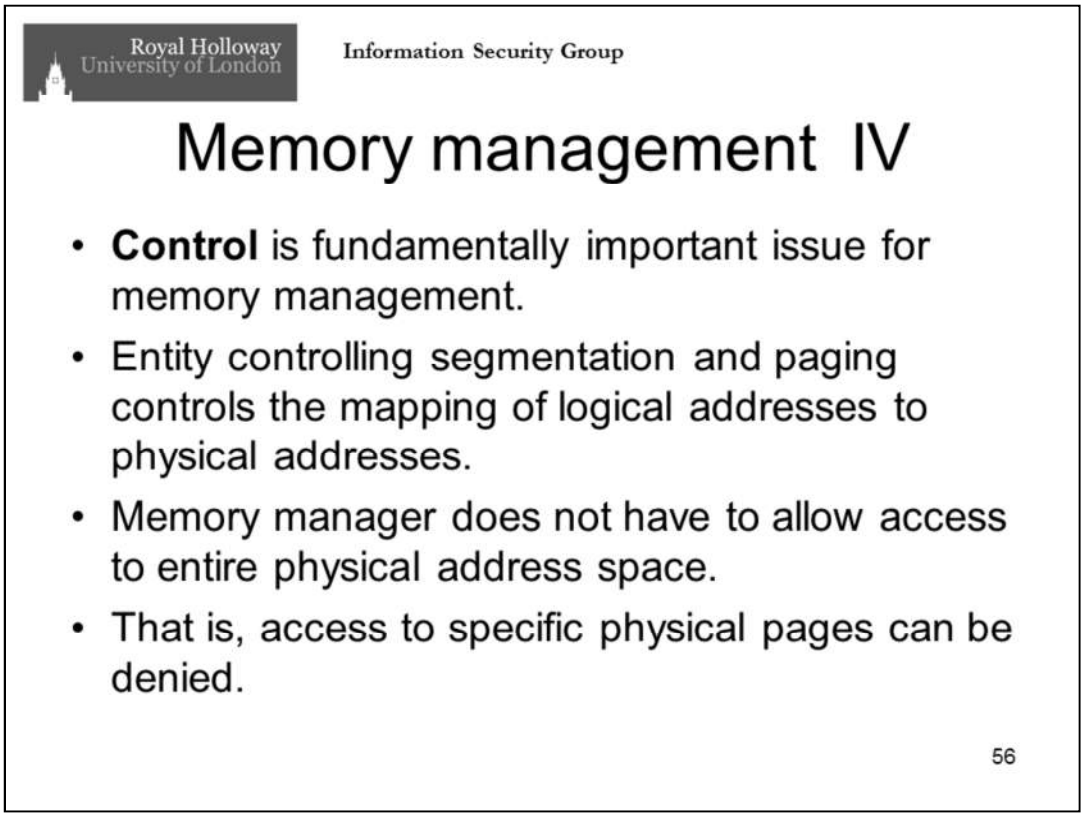
The diagram shows the operation of both **segmentation** and **paging**.

Note that GDT stands for **Global Descriptor Table**. This table is used by Intel x86-family processors to define the characteristics of the **memory segments**, including their base address, size, and access privileges such as executability and writability.



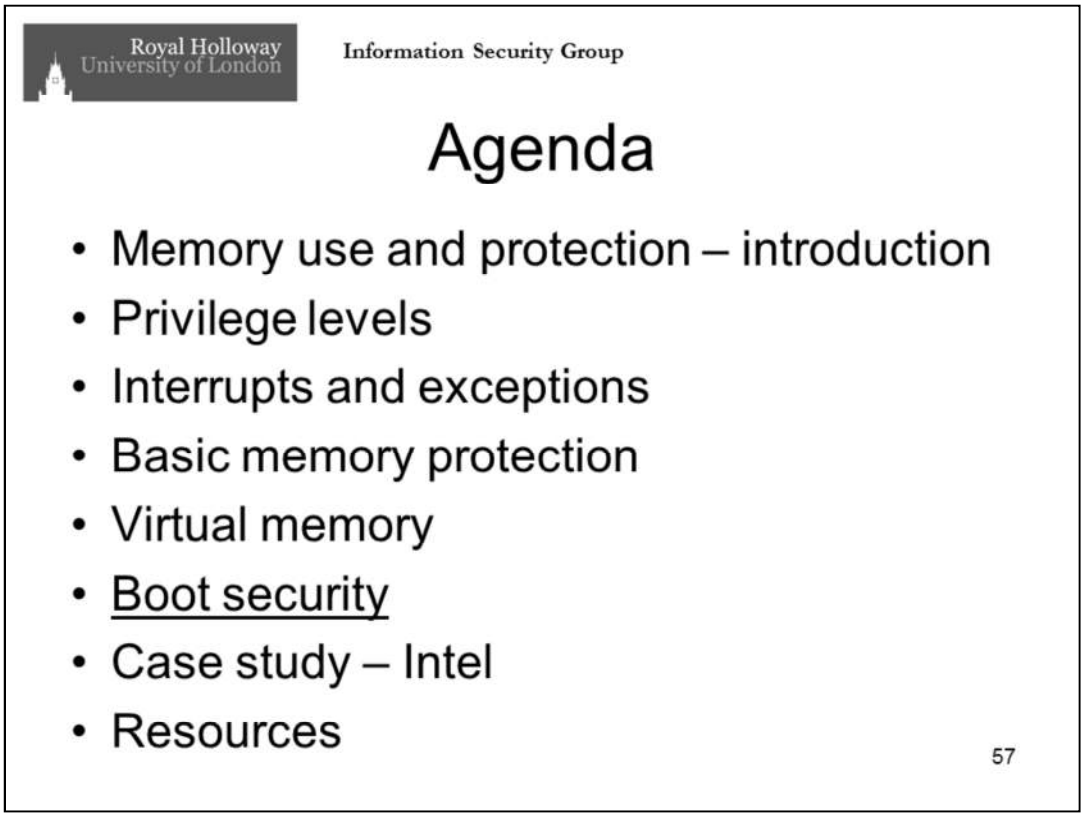
The following steps are involved:

1. the logical address breaks down into two components: **segment** and **offset**;
2. the segment is translated using the GDT to a **segment base address**, and the offset is added to produce a **linear address** (this is the end of segmentation task); if paging is not in use then the linear address is equal to the **physical address**;
3. if paging is in use, the linear address maps to three components; **directory**, **table** and **offset**;
4. the **page directory** is used to locate the entry in the **page table**, and the offset is added to give the **physical address**.



Control is a fundamentally important issue for memory management. The entity controlling segmentation and paging controls the mapping of logical addresses to physical addresses.

The memory manager does not have to allow access to the entire physical address space. That is, access to specific physical pages can be denied.



Memory protection ensures security within a running OS. In this section we consider a slightly different topic, namely how to ensure that the OS is running successfully after we start up a computer.

The boot process, i.e. the process which causes the OS to run after system start up, underlies OS security. We look at the main threats to boot security, as well as possible mitigations.

The discussion here is largely based on NIST SP800-147 and SP800-147B, available at:

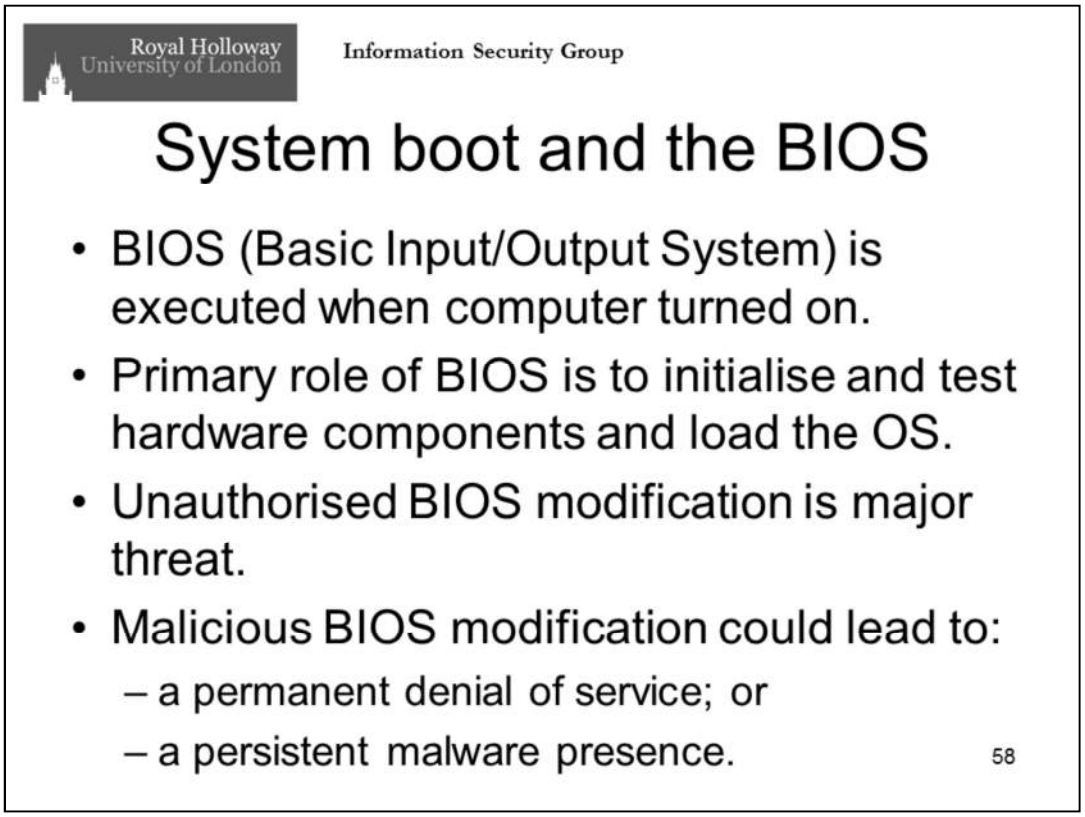
http://csrc.nist.gov/publications/nistpubs/800-147/NIST-SP800-147-April2011.pdf

and

http://dx.doi.org/10.6028/NIST.SP.800-147B

All NIST SPs are available here:

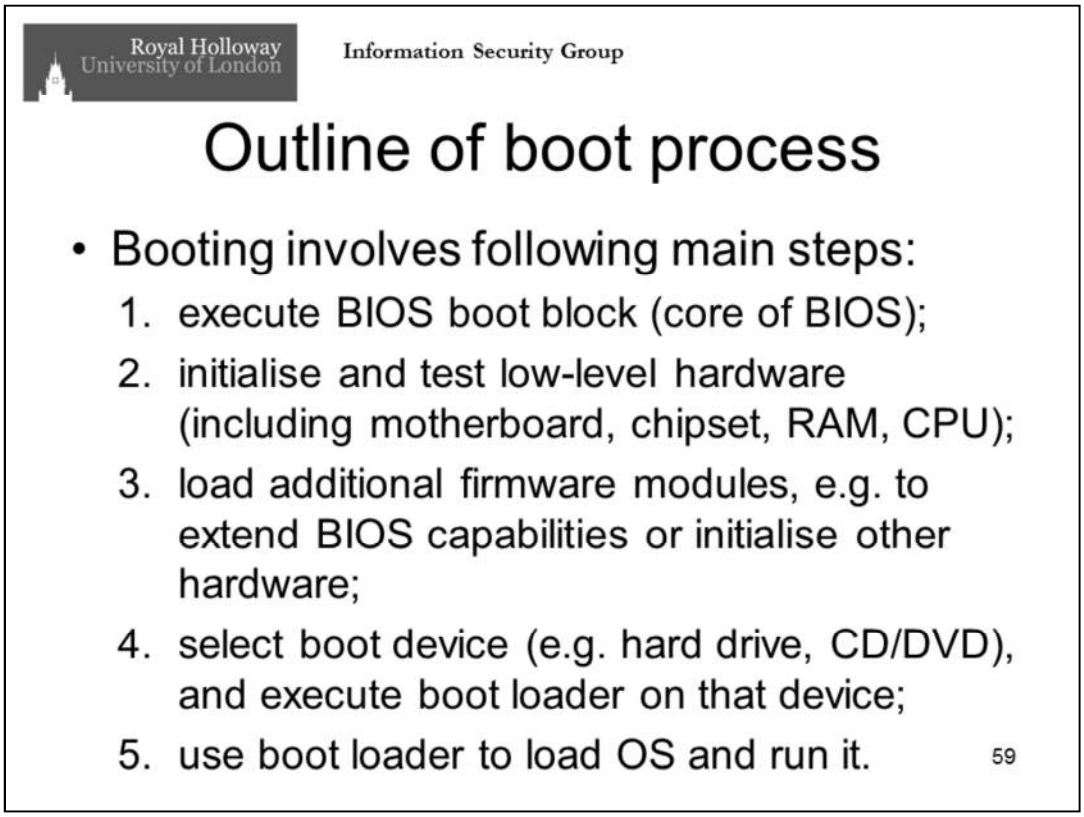
http://csrc.nist.gov/publications/PubsSPs.html



The system BIOS (Basic Input/Output System) is the first software executed on the main CPU when a computer is switched on. The primary role of the system BIOS on modern machines is to initialise and test hardware components and load the operating system. IOS is an example of **firmware**, i.e. software stored in persistent (non-volatile) memory, e.g. ROM or flash memory.

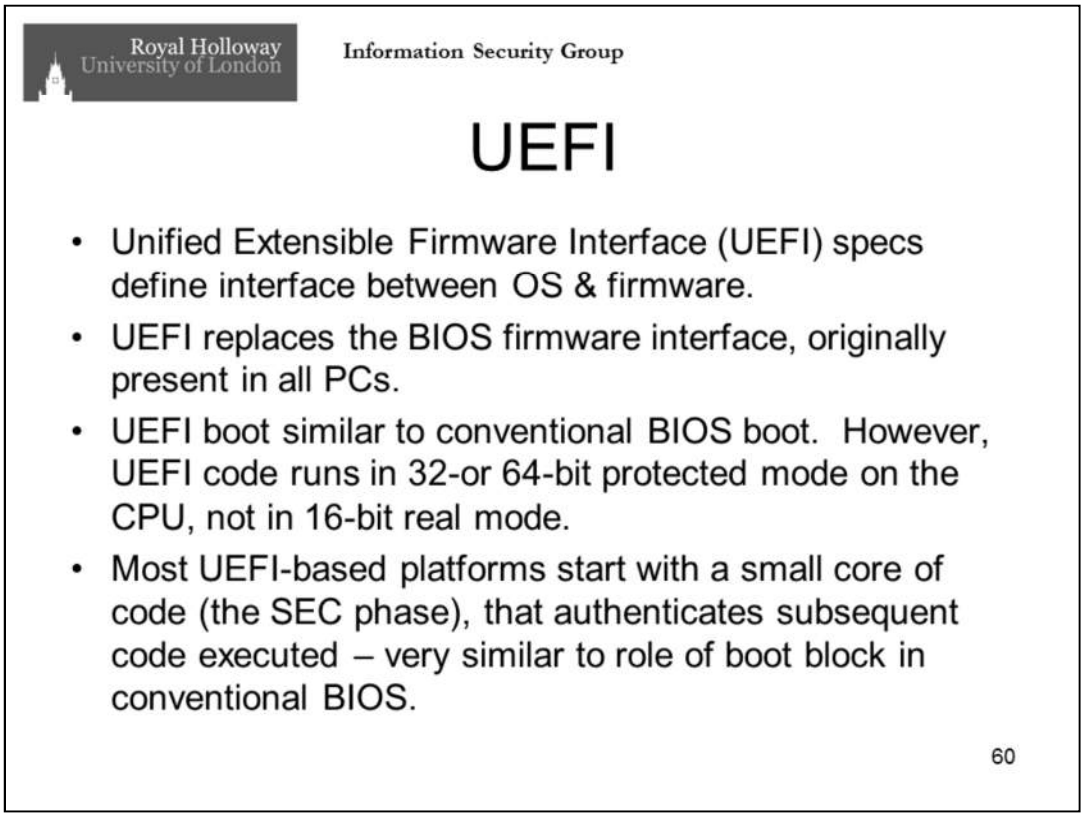
Unauthorised modification of the BIOS firmware, e.g. by malicious software, is a major threat because of the BIOS's uniquely privileged position within modern computing architectures. Malicious BIOS modification could lead to a permanent denial of service (i.e. a system which cannot boot) or a persistent malware presence on the machine.

The BIOS is typically developed by both original equipment manufacturers (OEMs) and independent BIOS vendors, and is distributed to end-users by motherboard or computer manufacturers. Manufacturers frequently update system firmware to fix bugs, patch vulnerabilities, and support new hardware.



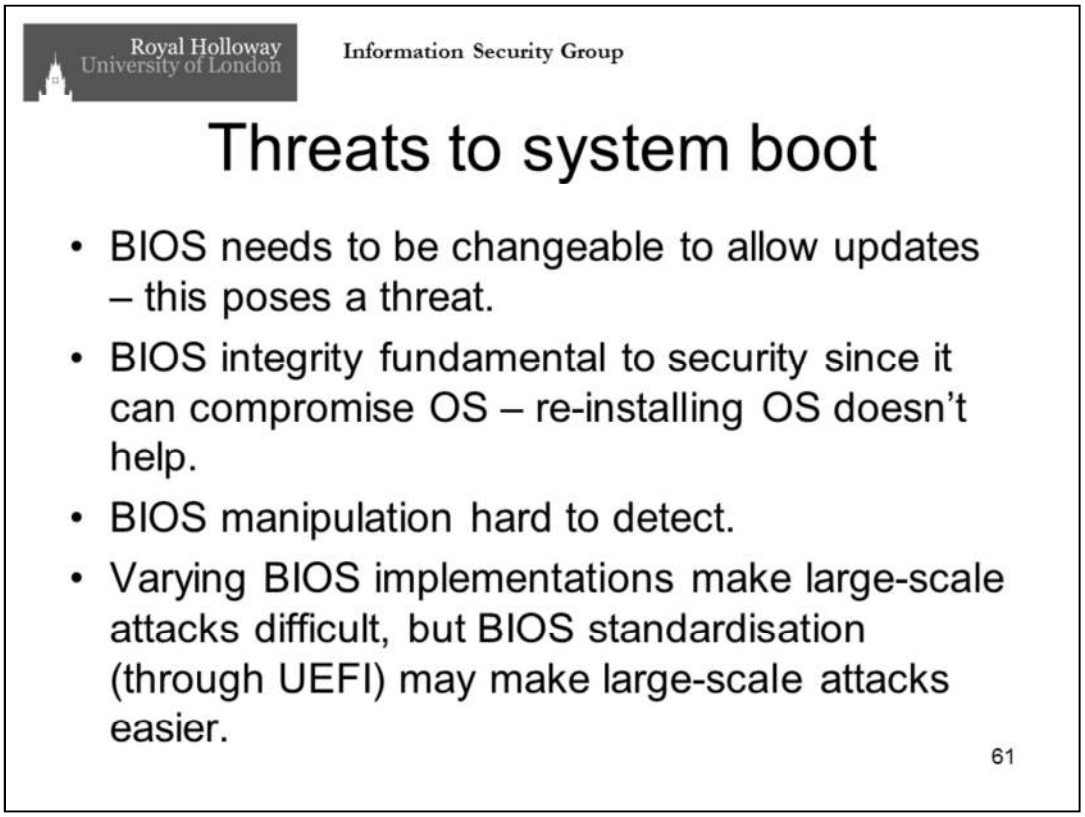
The primary function of the system BIOS is to initialize important hardware components and to load the operating system. This process is known as **booting**. The boot process of the system BIOS typically involves the following steps:

1. **Execute Core Root of Trust:** The system BIOS may include a small core block of firmware that executes first and is capable of verifying the integrity of other firmware components. This has traditionally been called the **BIOS Boot Block**. For trusted computing applications, it may also contain the Core Root of Trust for Measurement (CRTM).
2. **Initialize and Test Low-Level Hardware:** Very early in the boot process the system BIOS initialises and tests key pieces of hardware on the computer system, including the motherboard, chipset, memory and CPU.
3. **Load and Execute Additional Firmware Modules:** The system BIOS executes additional pieces of firmware that either extend the capabilities of the system BIOS or initialise other hardware components necessary for booting the system. These additional modules may be stored within the same flash memory as the system BIOS or they may be stored in the hardware devices they initialise (e.g. a video card or a local area network card).
4. **Select Boot Device:** After the system hardware has been configured, the system BIOS searches for a boot device (e.g. a hard drive, optical drive or USB drive) and executes the **boot loader** stored on that device.
5. **Load Operating System**: While the system BIOS is still in control of the computer, the boot loader begins to load and initialise the operating system kernel. Once the kernel is functional, control of the computer system transfers from the system BIOS to the operating system.



The Unified Extensible Firmware Interface (UEFI) specifications define a software interface between an operating system and platform firmware. UEFI is intended to replace the BIOS firmware interface, originally present in all PCs. In practice, most UEFI firmware images provide legacy support for 'old style' BIOS services. UEFI can support remote diagnostics and repair of computers, even without another operating system.

The UEFI boot process follows a similar flow to the conventional BIOS boot process. One difference is that UEFI code runs in 32-or 64-bit protected mode on the CPU, not in 16-bit real mode, as is often the case with a conventional BIOS. Most UEFI-based platforms start with a small core block of code that has the primary responsibility of authenticating subsequent code executed on the computer system. This is very similar to the role of the boot block in a conventional BIOS. This part of the boot process is known as the Security (SEC) phase, and it serves as the core root of trust in the computer system.



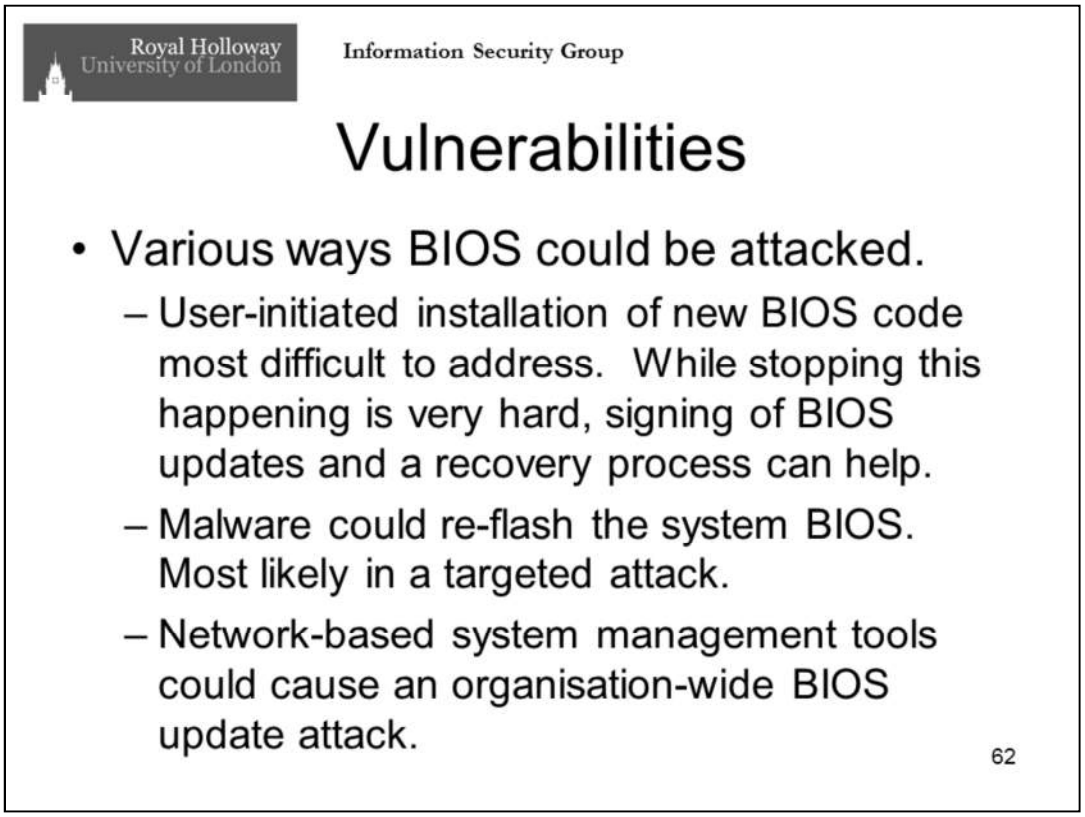
BIOS updates may be necessary, e.g. to correct bugs or to add support for new types of hardware. This involves loading new code into the persistent (flash) memory in the computer. That is, the BIOS needs to be changeable.

Since it is the first code that is executed by the main CPU, the BIOS is a critical security component of a computer. While the system BIOS, possibly with the use of a Trusted Platform Module (TPM), can verify the integrity of firmware and software executed later in the boot process, typically all or part of the system BIOS itself is implicitly trusted.

The system BIOS is a potentially attractive target for attack. Malicious code running at the BIOS level could have a great deal of control over a computer system. It could be used to compromise any components that are loaded later in the boot process, including the boot loader, hypervisor, and/or operating system.

The BIOS is stored on non-volatile memory that persists between power cycles. Malware written into a BIOS could be used to re-infect machines even after new operating systems have been installed or hard drives replaced. Because the system BIOS runs early in the boot process with very high privileges on the machine, malware running at the BIOS level may be very difficult to detect. Because the BIOS loads first, there is no opportunity for anti-malware products to authoritatively scan the BIOS.

BIOS exploits are likely to be highly system-specific, i.e. directed at a specific BIOS version or certain hardware components (e.g. a particular motherboard chipset). By contrast, most malware targets software executing at or above the OS kernel, where it can attack larger classes of machines. BIOS-level malware may be more likely to be used in targeted attacks on high-value computer systems. The move to UEFI-based BIOS may make it easier for malware to target the BIOS in a widespread fashion, as these BIOS implementations are based on a common specification.

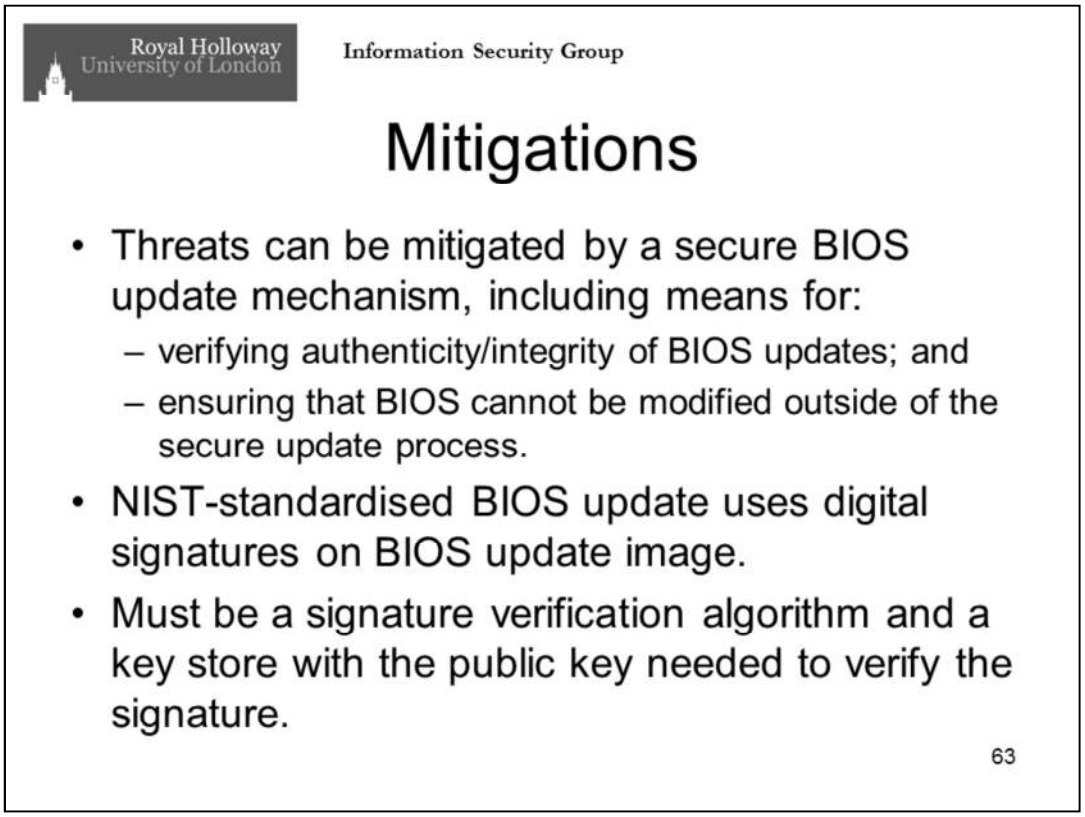


One of the most difficult threats to prevent is a user-initiated installation of a malicious system BIOS. User-initiated BIOS update utilities are often the primary method for updating the system BIOS. It is almost impossible to prevent users from installing unapproved BIOS images if they have physical access to the computer system. Security processes may be able to detect and remediate the unapproved system BIOS, such as initiating a recovery process to restore to an approved BIOS.

Malware could leverage weak BIOS security controls or exploit vulnerabilities in the system BIOS itself to re-flash or modify the system BIOS. General-purpose malicious software is unlikely to include this functionality, but a targeted attack on an organization could be directed towards an organization's standard system BIOS. The malicious BIOS can be delivered to the system either over a network, or using removable media (e.g. USB).

Network-based system management tools could also be used to launch an organisation-wide attack on system BIOSs. For example, consider an organisation-maintained update server for the organization's deployed system BIOS; a compromised server could push a malicious system BIOS to computer systems across the organisation.

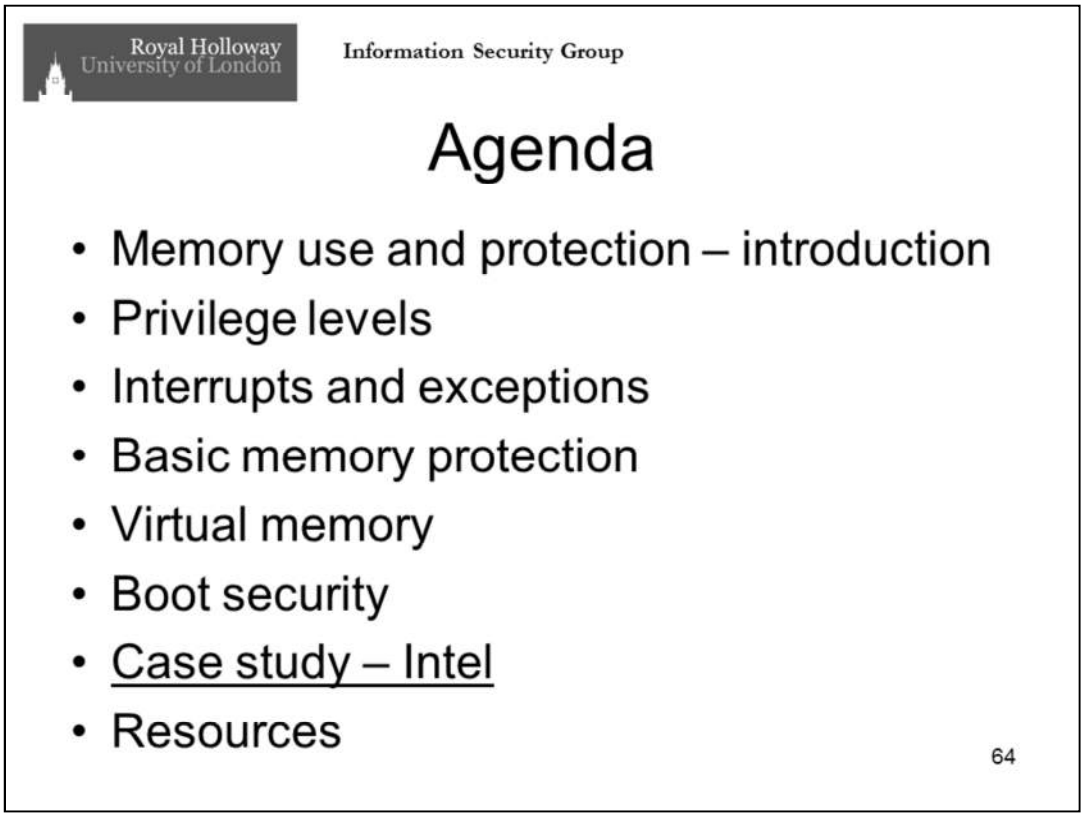
Any of these approaches could be used to cause the BIOS to 'roll back' to an authentic but vulnerable system BIOS. This is a particularly insidious attack, since the 'bad' BIOS is authentic (i.e. it was shipped by the manufacturer).



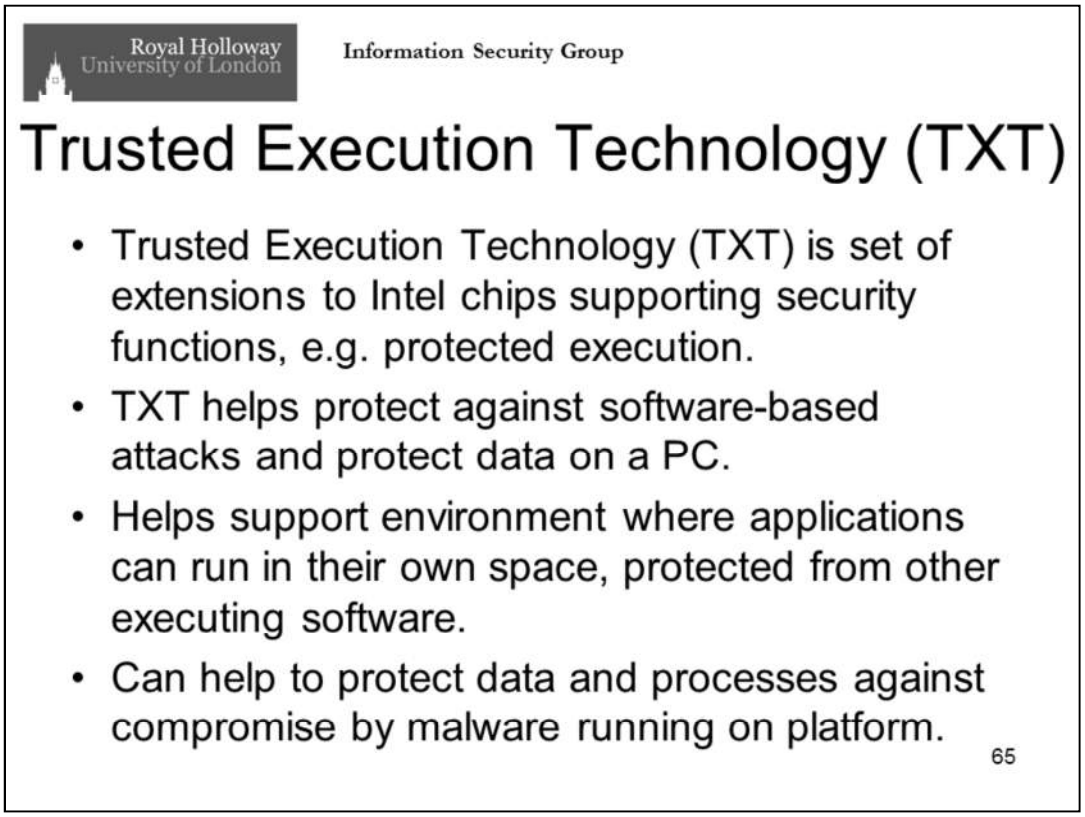
The threats described on the previous slide can largely be mitigated by implementing a secure BIOS update mechanism. A secure BIOS update mechanism includes:

- a process for verifying the authenticity and integrity of BIOS updates; and
- a mechanism for ensuring that the BIOS is protected from modification outside of the secure update process.

Authentication verifies that a BIOS update image was generated by an authentic source and is unaltered. The NIST-standardised authenticated BIOS update mechanism uses digital signatures to ensure the authenticity of the BIOS update image. To update the BIOS using the NIST-approved authenticated BIOS update mechanism, there need to be a Root of Trust for Update (RTU) that contains a signature verification algorithm and a key store that includes the public key needed to verify the signature on the BIOS update image. The key store and the signature verification algorithm must be stored in a protected fashion on the computer system, modifiable only using an authenticated update mechanism or a secure local update mechanism.



We conclude this introduction to hardware security issues by looking at security measures that are included in the Intel processor family, whose architecture is used in all PCs and Apple computers. We look at how Intel chooses to implement some of the general memory protection techniques we have just described. We also look at some additional features provided to support secure virtualisation.



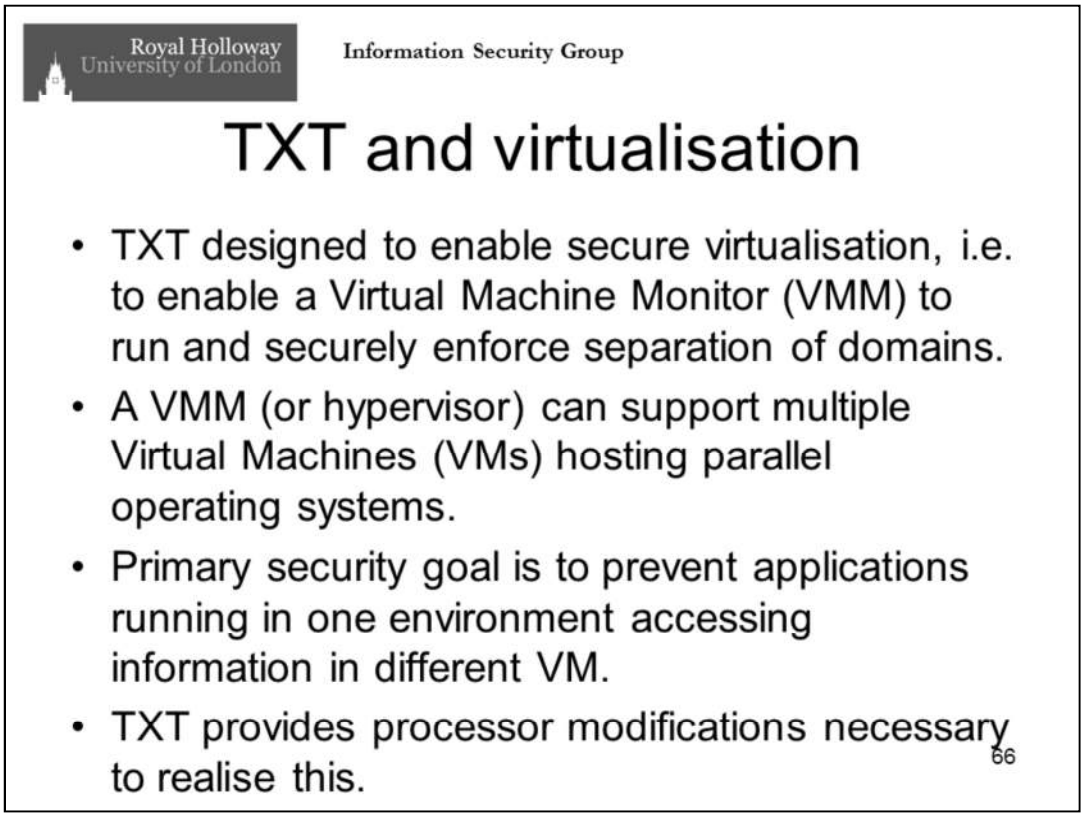
Intel's Trusted Execution Technology (TXT) for safer computing, formerly named LaGrande Technology (LT), is a set of hardware extensions to Intel processors and chipsets that support security capabilities such as measured launch and protected execution.

TXT provides hardware-based mechanisms that help protect against software-based attacks and protect the confidentiality and integrity of data stored or created on a PC.

It does this by enabling an environment where applications can run in their own space, protected from other executing software.

This can help to protect data and processes from being compromised by malware running on the platform.

We use TXT and LT interchangeably.

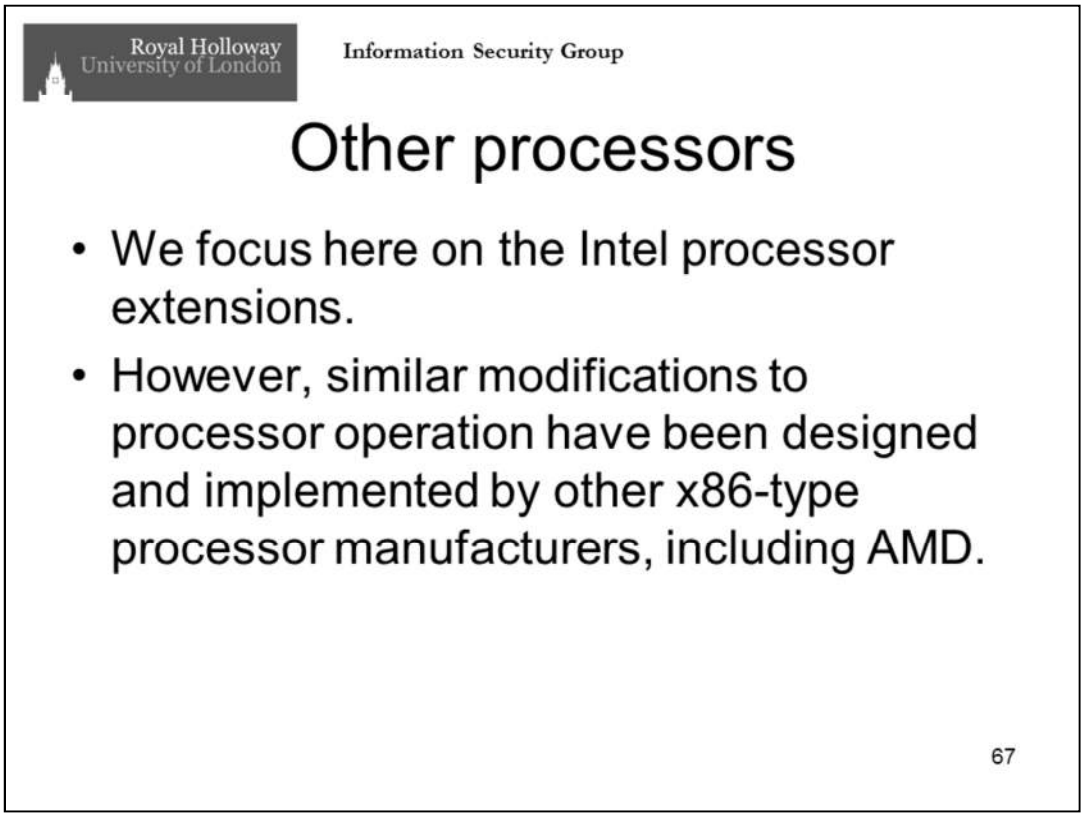


It is important to realise that TXT is designed to enable secure virtualisation, i.e. to enable a Virtual Machine Monitor (VMM) to run and securely enforce separation of domains. TXT also supports use of the Trusted Platform Module (TPM), but we do not discuss this further here.

A VMM (or hypervisor) can support multiple Virtual Machines (VMs) hosting parallel operating systems.

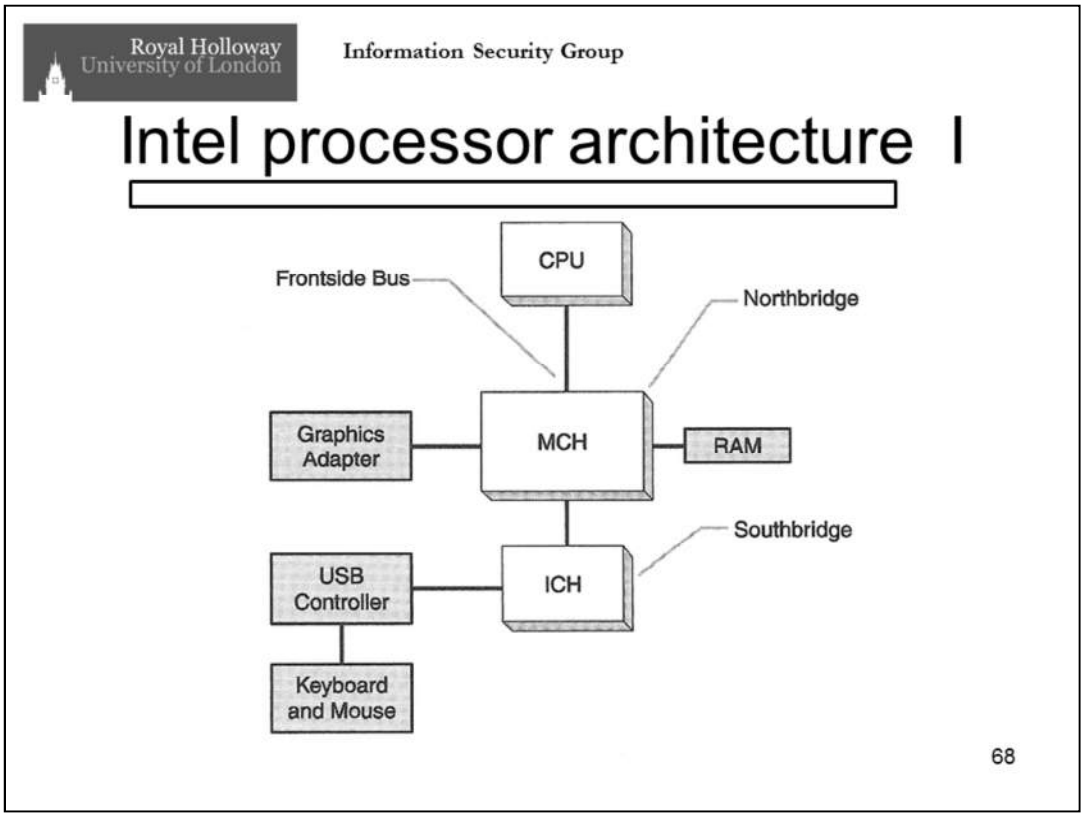
The primary security goal is to prevent applications running in one environment accessing information handled in different VMs.

TXT provides some of the key processor modifications necessary to realise this security goal.

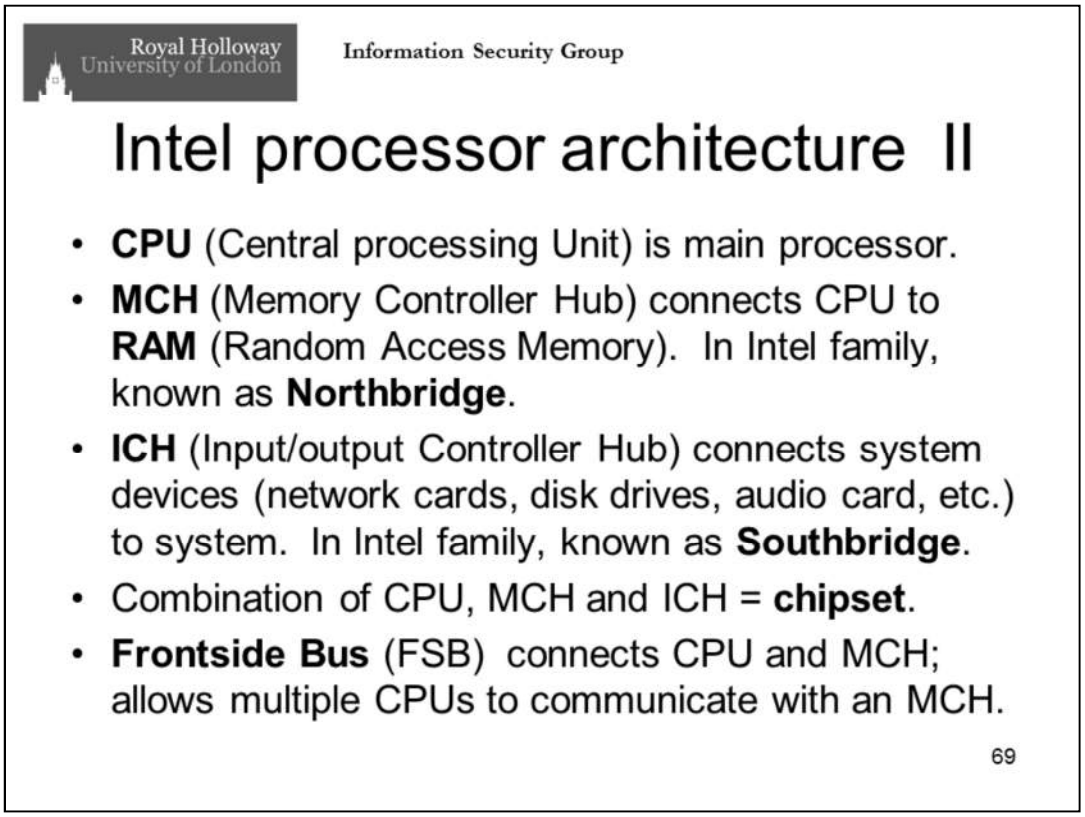


In this course we focus on the Intel processor extensions.

However, similar modifications to processor operation have been designed and implemented by other manufacturers of processors conforming to Intel's x86 architecture, including AMD – whose extensions are known as AMD-V.



The diagram shows a high-level overview of the Intel processor architecture. The terms used are explained on the next slide.



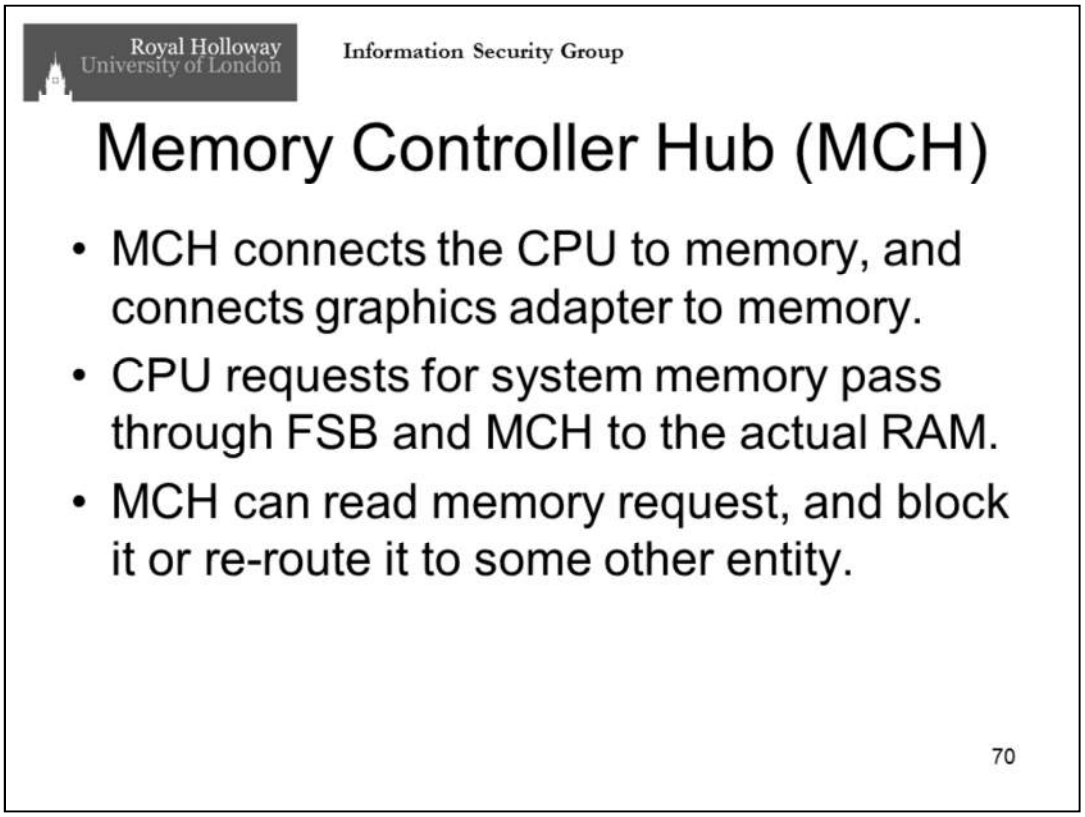
The **CPU** (Central processing Unit) is the main processor.

The **MCH** (Memory Controller Hub) connects the CPU to the **RAM** (Random Access Memory). In the Intel family this is also known as the **Northbridge**.

The **ICH** (Input/output Controller Hub) connects system devices (network cards, disk drives, audio components, etc.) to the system. In the Intel family this is also known as the **Southbridge**.

The combination of a CPU, a MCH and an ICH is often referred to as a **chipset**.

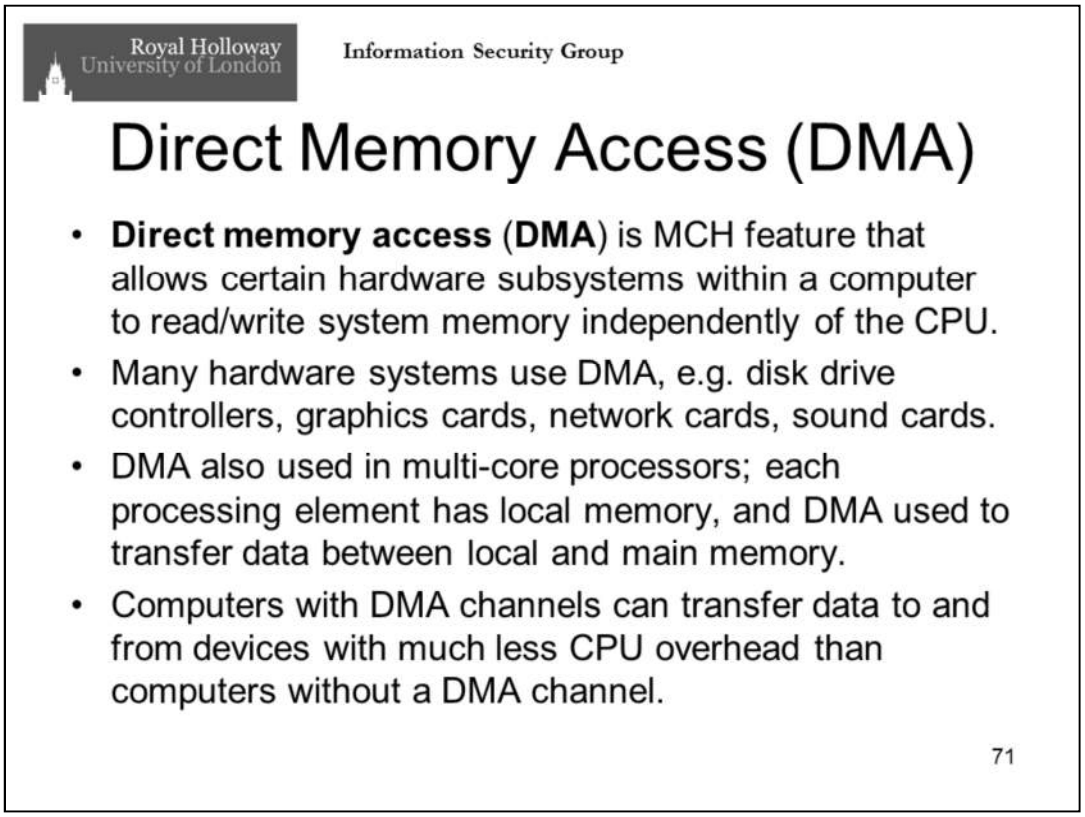
The **Frontside Bus** (FSB) connects the CPU and MCH, allowing multiple CPUs to communicate with a single MCH.



The MCH connects the CPU to memory, and also connects the graphics adapter to system memory.

All CPU requests for system memory pass through the FSB and MCH to the actual RAM.

The MCH can read the memory request, and block it or re-route it to some other entity.

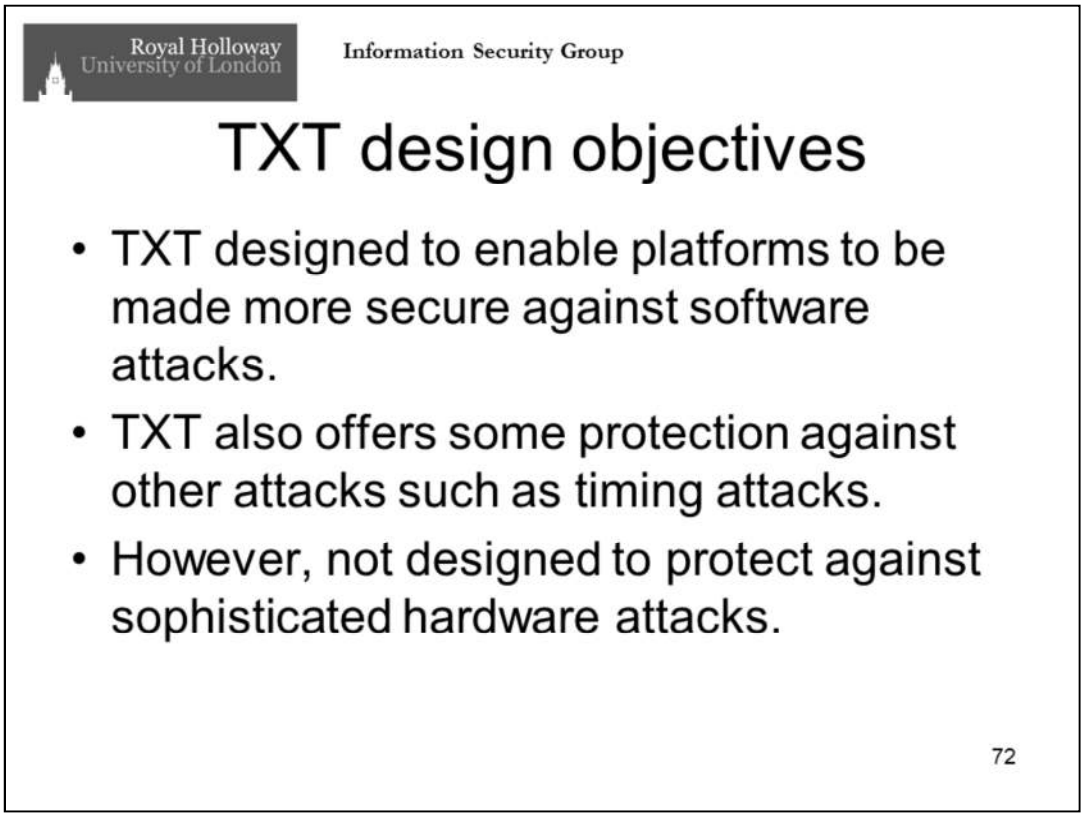


Direct memory access (**DMA**) is an MCH feature that allows certain hardware subsystems within a computer to read/write system memory independently of the CPU.

Many hardware systems use DMA, including disk drive controllers, graphics cards, network cards and sound cards.

DMA is also used for intra-chip data transfer in multi-core processors, where each processing element is equipped with a local memory and DMA is used to transfer data between the local memory and the main memory.

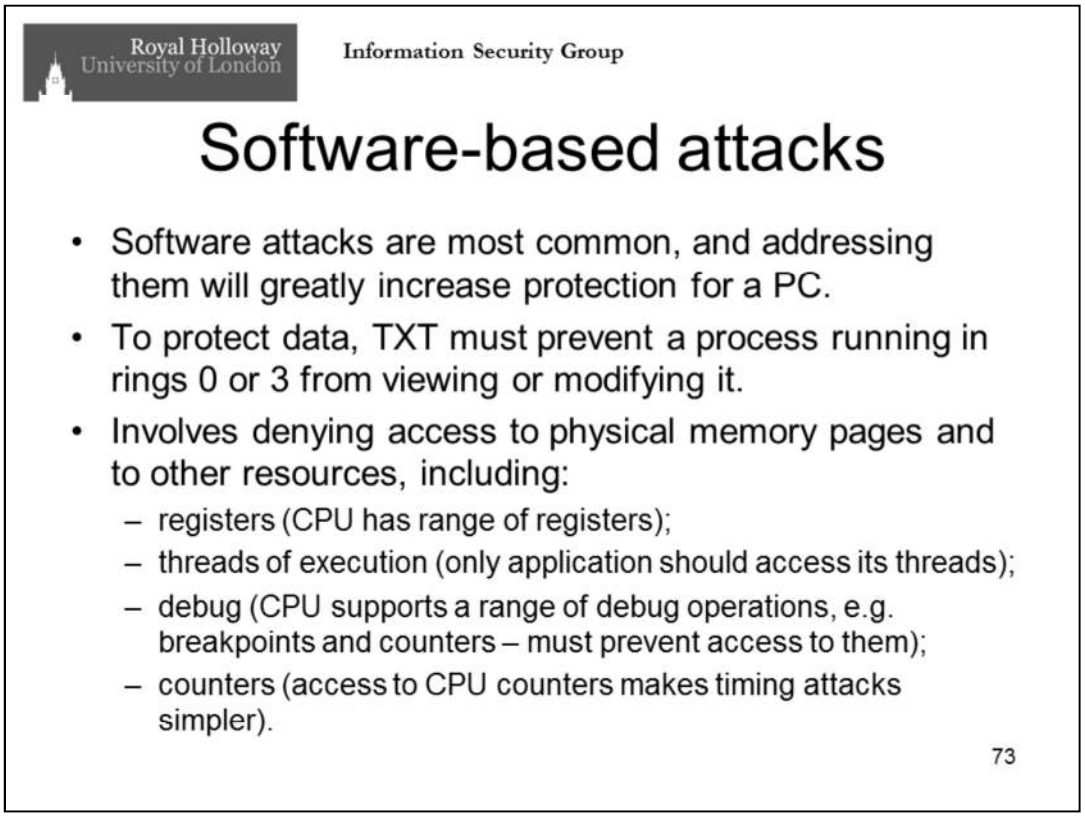
Computers that have DMA channels can transfer data to and from devices with much less CPU overhead than computers without a DMA channel.



TXT is designed to enable platforms to be made more secure against software attacks.

TXT also offers some protection against other attacks such as timing attacks.

However, it is not designed to protect against sophisticated hardware attacks.

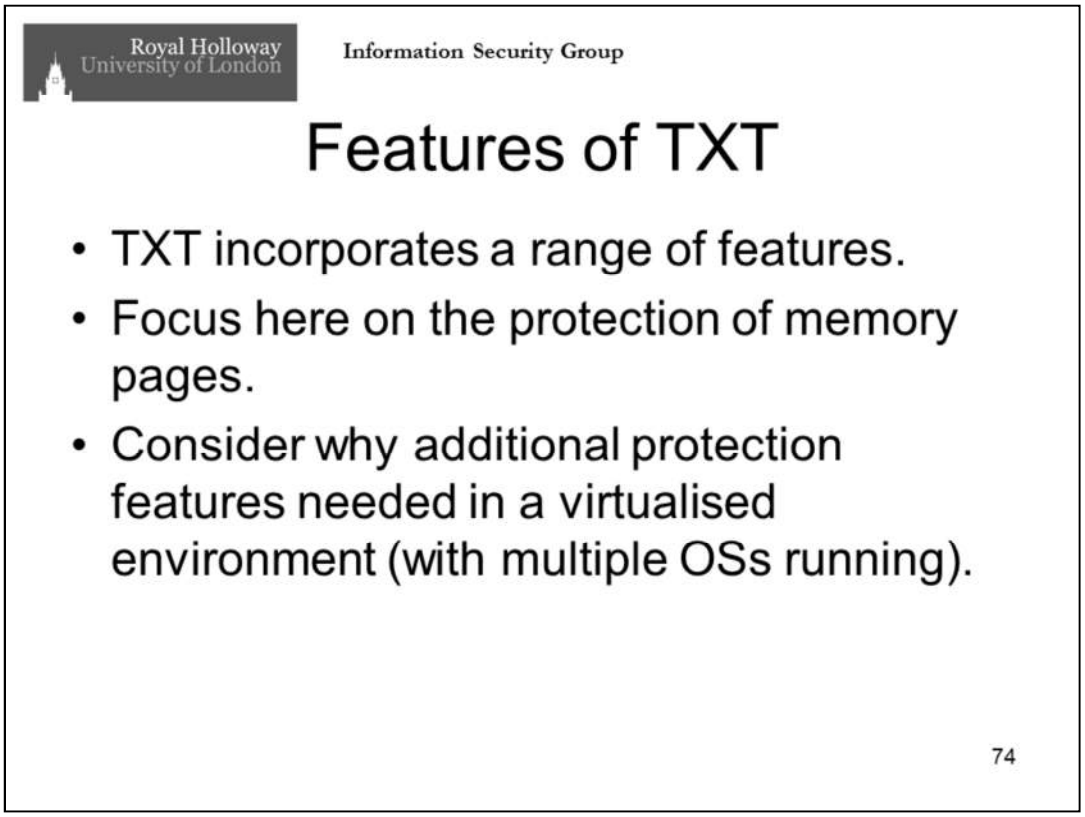


Software attacks are most common, and addressing them will greatly increase protection for a PC.

While data is being protected by TXT, it must protect it by preventing any process running in rings 0 or 3 from viewing or modifying the data.

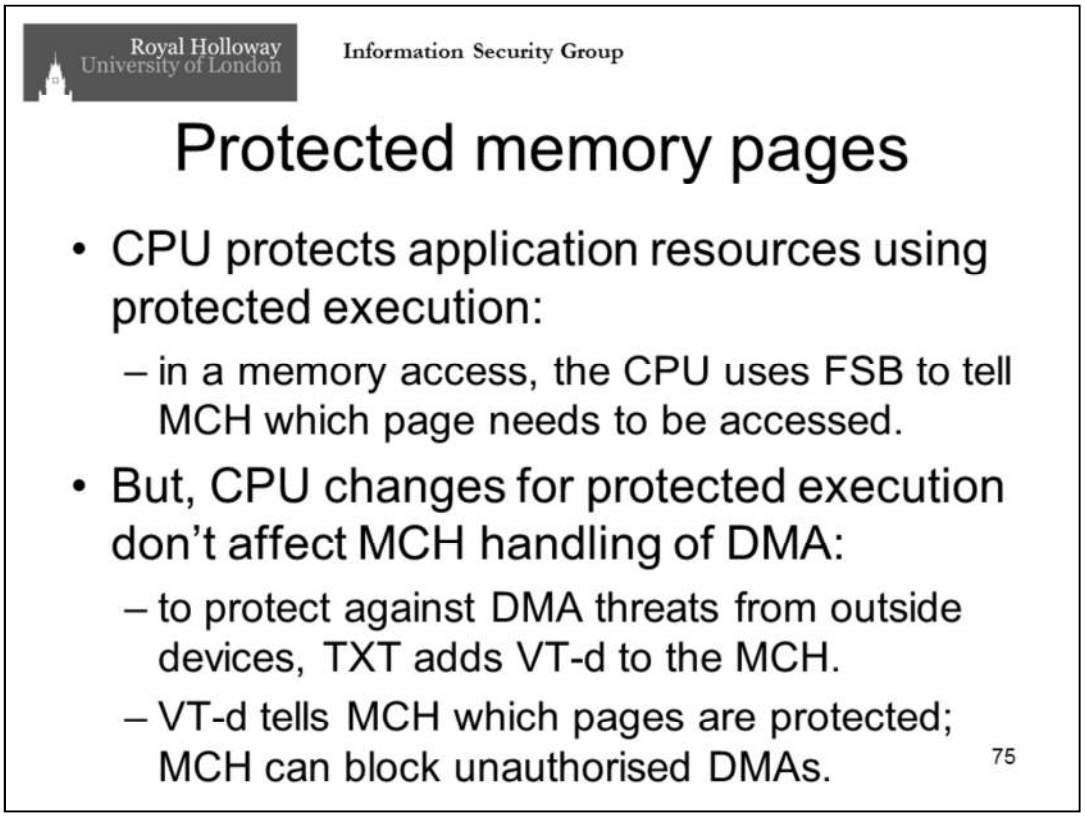
This not only involves denying access to a physical memory page – other resources affect application data, including:

- registers (CPU has range of registers);
- threads of execution (only an application should have access to its threads);
- debug (CPU supports a range of debug operations, e.g. breakpoints and counters – must prevent access to these resources);
- counters (access to CPU counters makes timing attacks simpler).



TXT incorporates a range of features. We focus here on the protection of memory pages.

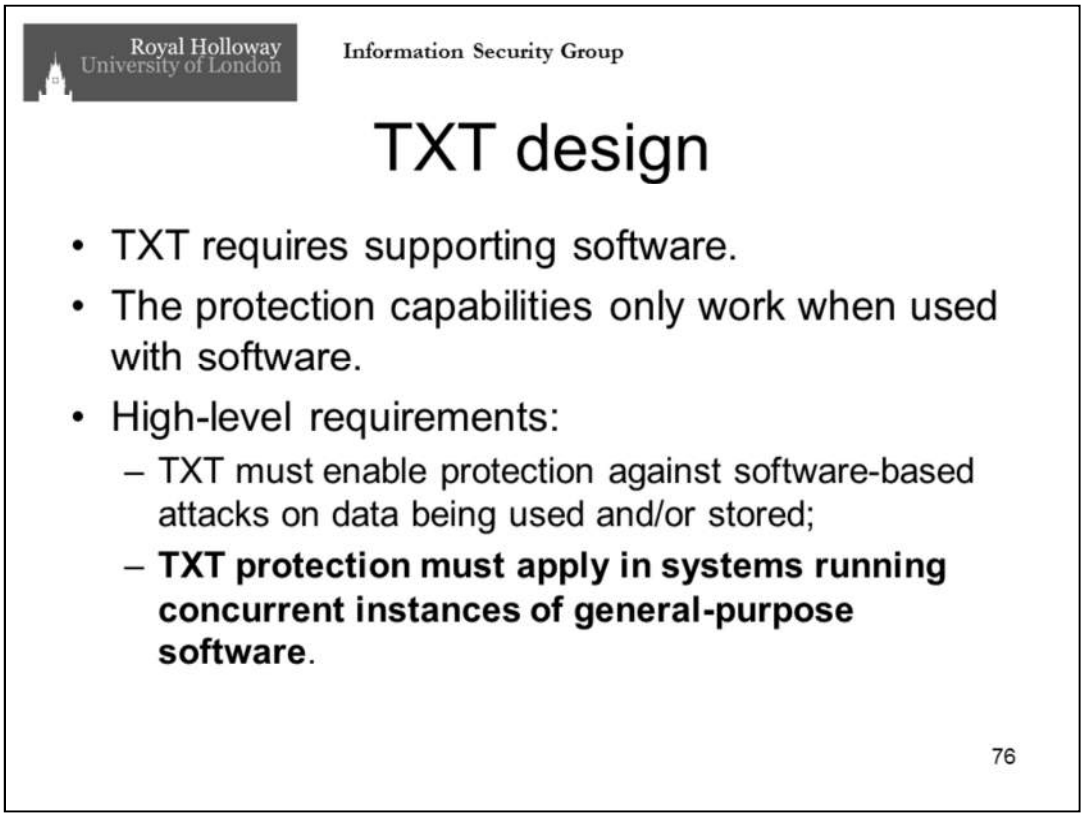
In particular we consider why additional protection is particularly needed in a virtualised environment, i.e. where multiple instances of a general purport operating system may be running concurrently.



The CPU protects application resources using protected execution. When memory accesses occur, the CPU uses the FSB to indicate to the MCH which page needs to be accessed.

However, the CPU changes to support protected execution do not change how the MCH handles direct memory access (DMA).

To protect against threats posed by DMA from outside devices, TXT adds the VT-d functionality table to the MCH. This functionality informs the MCH which memory pages are being protected by the CPU, and the MCH uses VT-d to block unauthorised DMAs.

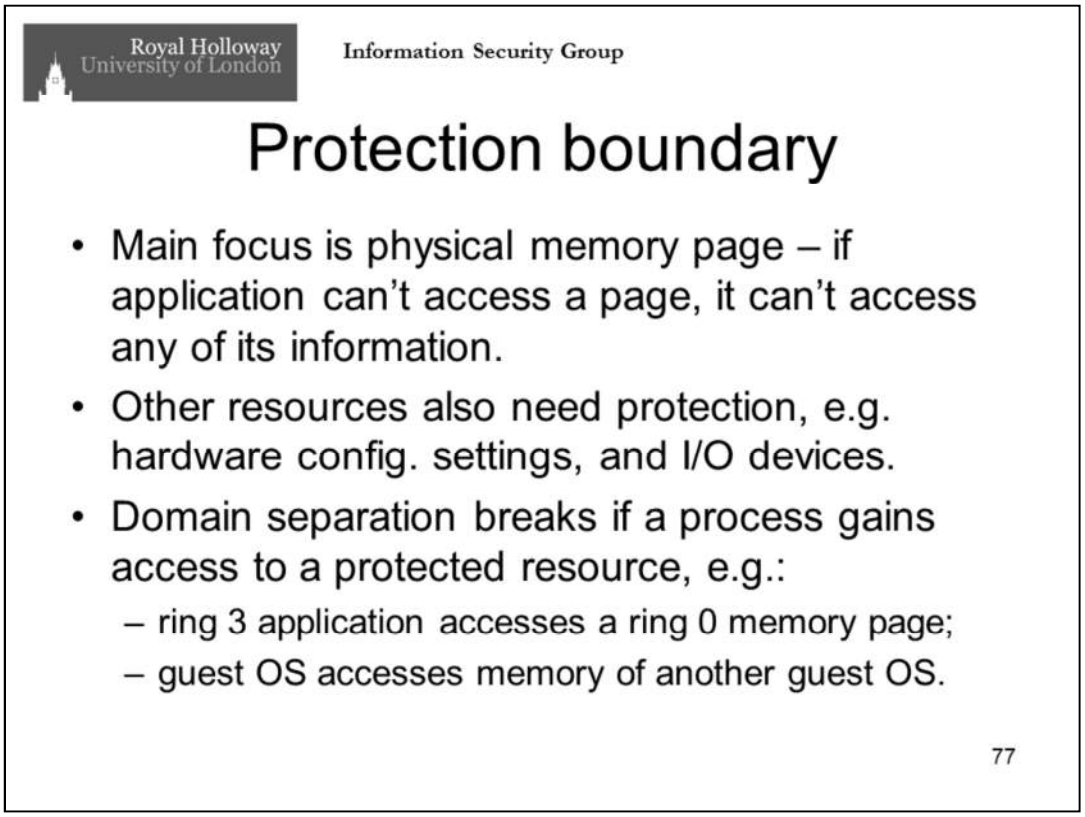


We next briefly review the main requirements used to motivate the design of TXT. It is important to note that TXT requires supporting software – the TXT protection capabilities can only be realised when used in combination with appropriate controlling software.

The high-level requirements for TXT include the following:

• a TXT system must enable protection against software-based attacks on data being used and/or stored;

• **TXT protection must be available in systems running concurrent instances of general-purpose software (i.e. must support strong separation of environments running on VMMs.**



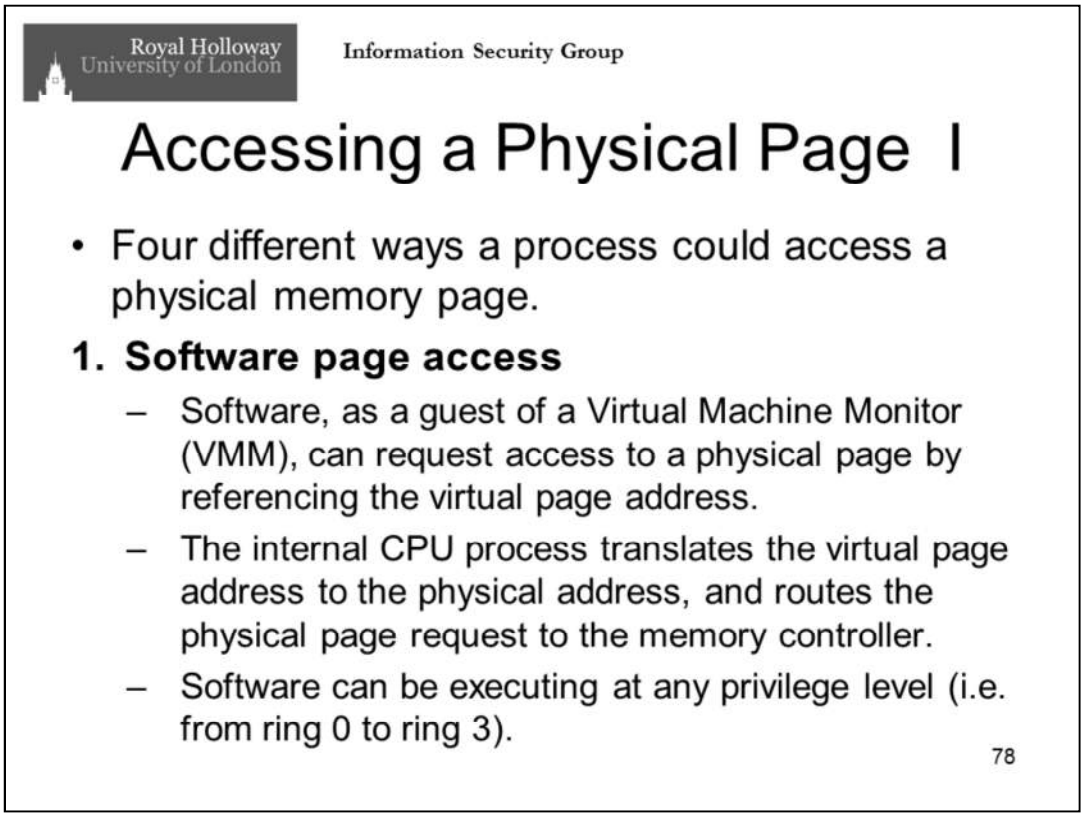
The primary focus of TXT protection is the physical memory page – if an application cannot access a page, then it cannot access any of the information it holds.

However, other resources also require protection, including hardware configuration settings, input/output devices, and other system entities.

From a TXT perspective, domain separation breaks if a (software) process gains access to a protected resource. Examples of possible security breaches include:

• if a ring 3 application accesses a protected ring 0 memory page; or

• if an application running in one guest OS (running on a VMM) accesses the memory of another guest OS without permission.



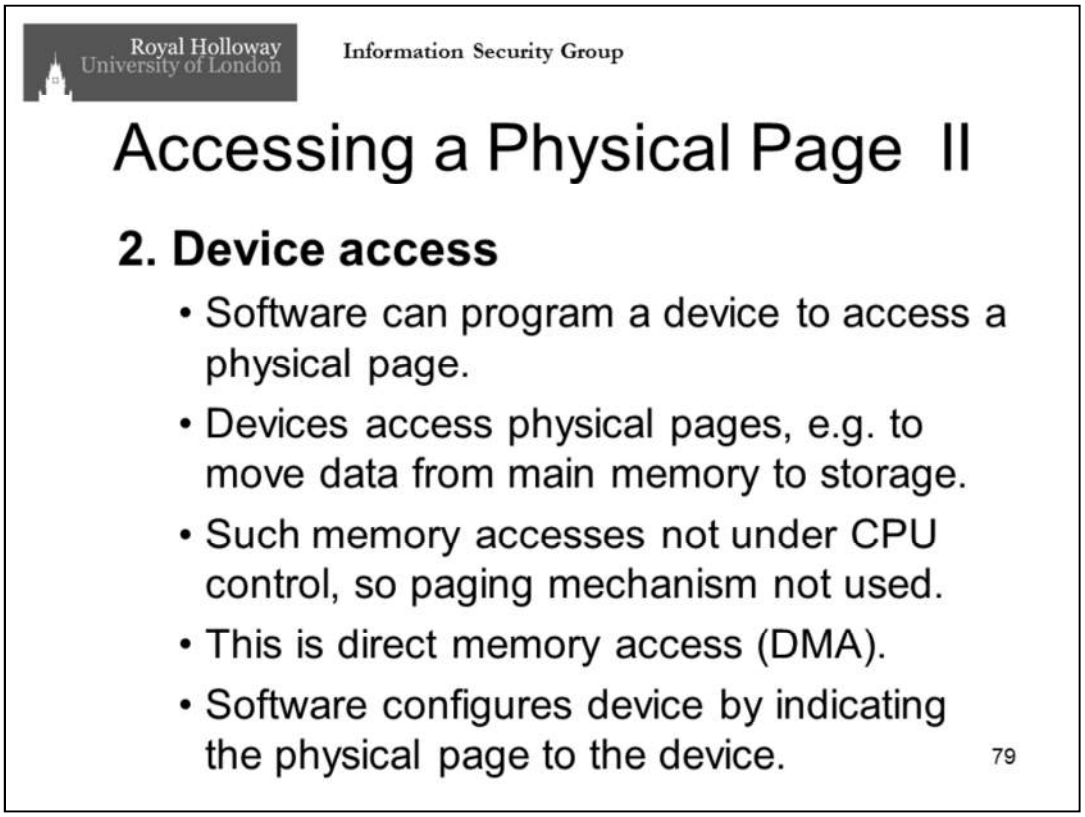
We briefly review four different ways in which a process could access a physical memory page.

1. Software page access

The software, as a guest of a Virtual Machine Monitor (VMM) can request access to a physical page by referencing the virtual page address.

The internal CPU process translates the virtual page address to the actual physical address, and routes the physical page request to the memory controller.

The software can be executing at any privilege level (i.e. from ring 0 to ring 3).



2. Device access

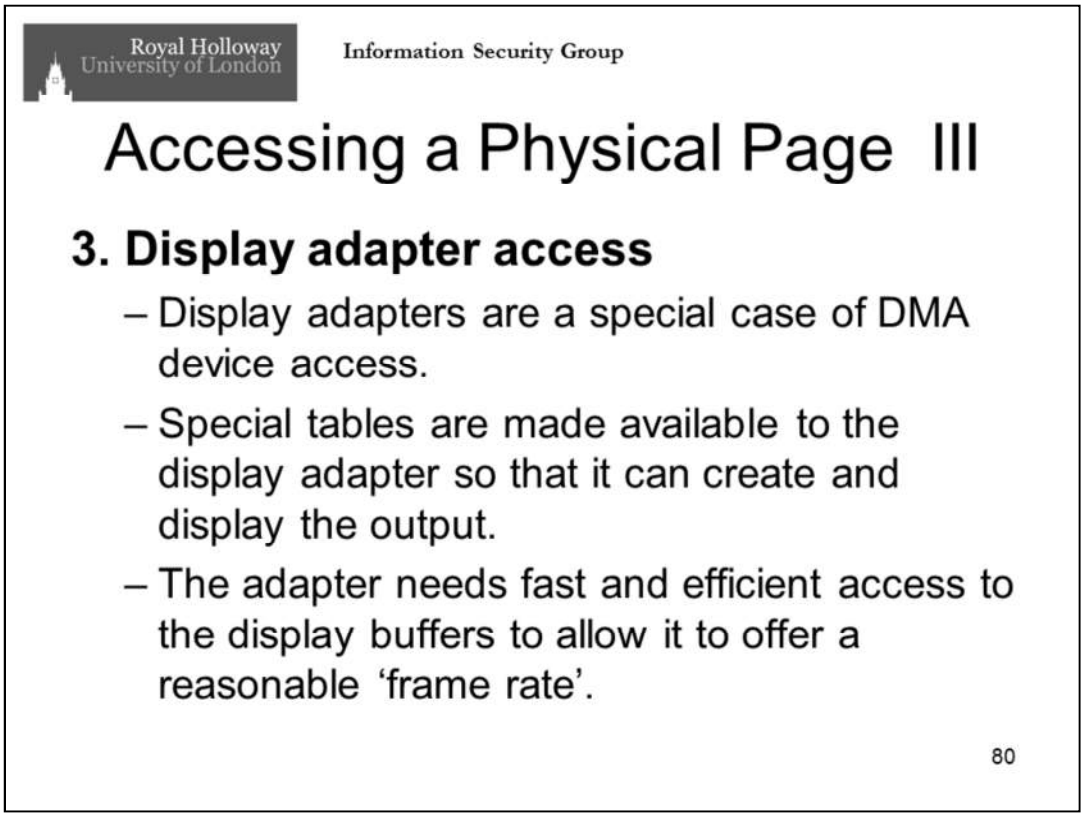
The software can program a device to access a physical page.

Devices commonly access physical pages to move information from main system memory to storage or a network device.

Such memory accesses are not under CPU control, so the paging mechanism is not used.

This is direct memory access (DMA).

Software configures the device by indicating which physical page the device should access.

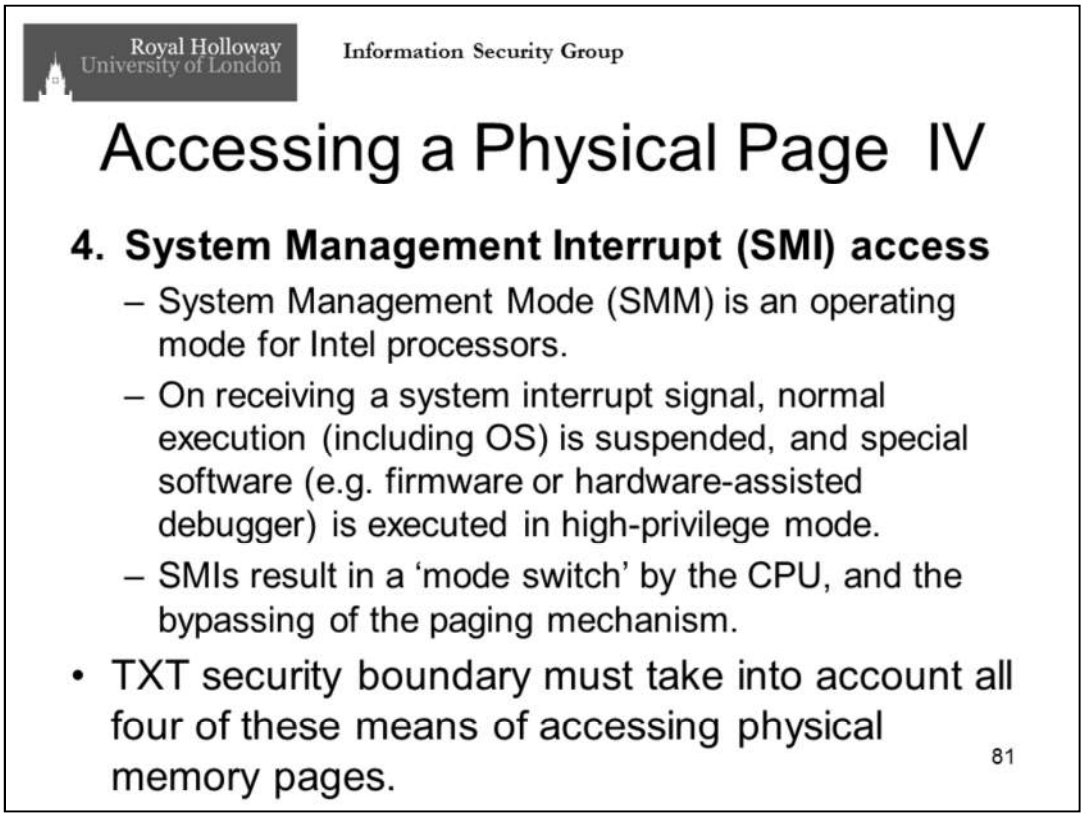


3. Display adapter access

Display adapters are a special case of DMA device access.

Special tables are made available to the display adapter so that it can create and display the output.

The adapter needs fast and efficient access to the display buffers to allow it to offer a reasonable 'frame rate'.



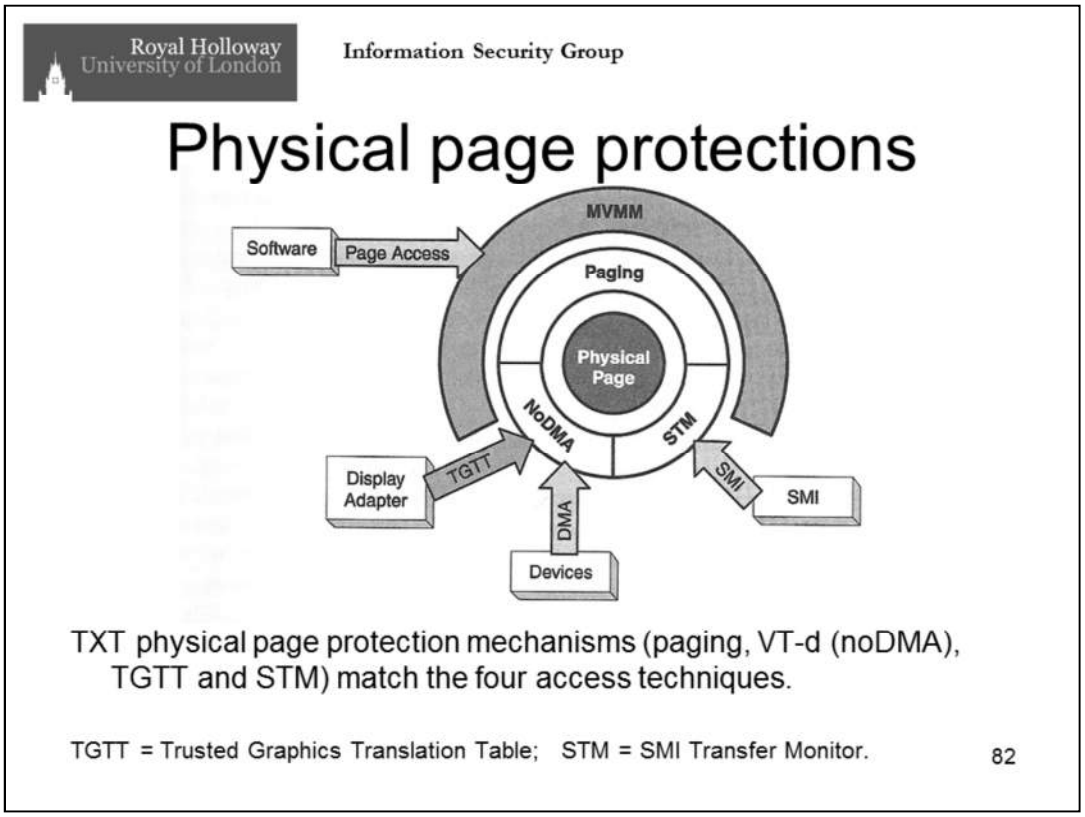
4. System Management Interrupt (SMI) access

System Management Mode (SMM) is an operating mode for Intel processors.

On receiving a system interrupt signal, all normal execution (including the operating system) is suspended, and special separate software (usually firmware or a hardware-assisted debugger) is executed in high-privilege mode.

SMIs result in a 'mode switch' by the CPU, and the bypassing of the paging mechanism.

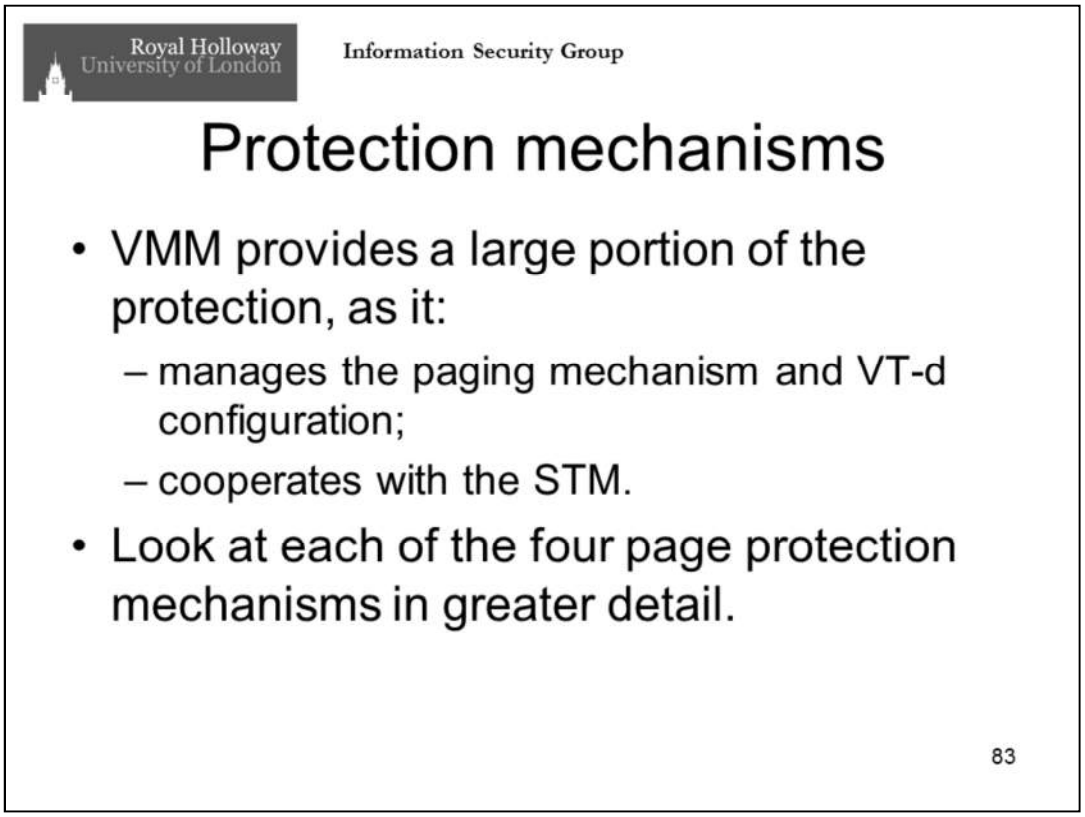
The TXT security boundary must take into account all four of these means of accessing physical memory pages.



The picture shows the four TXT physical page protection mechanisms, i.e.

- Paging,
- Virtualisation Technology for Directed I/O (VT-d), shown as noDMA,
- the Trusted Graphics Translation Table (TGTT), and
- the SMI (System Management Interrupt) Transfer Monitor (STM).

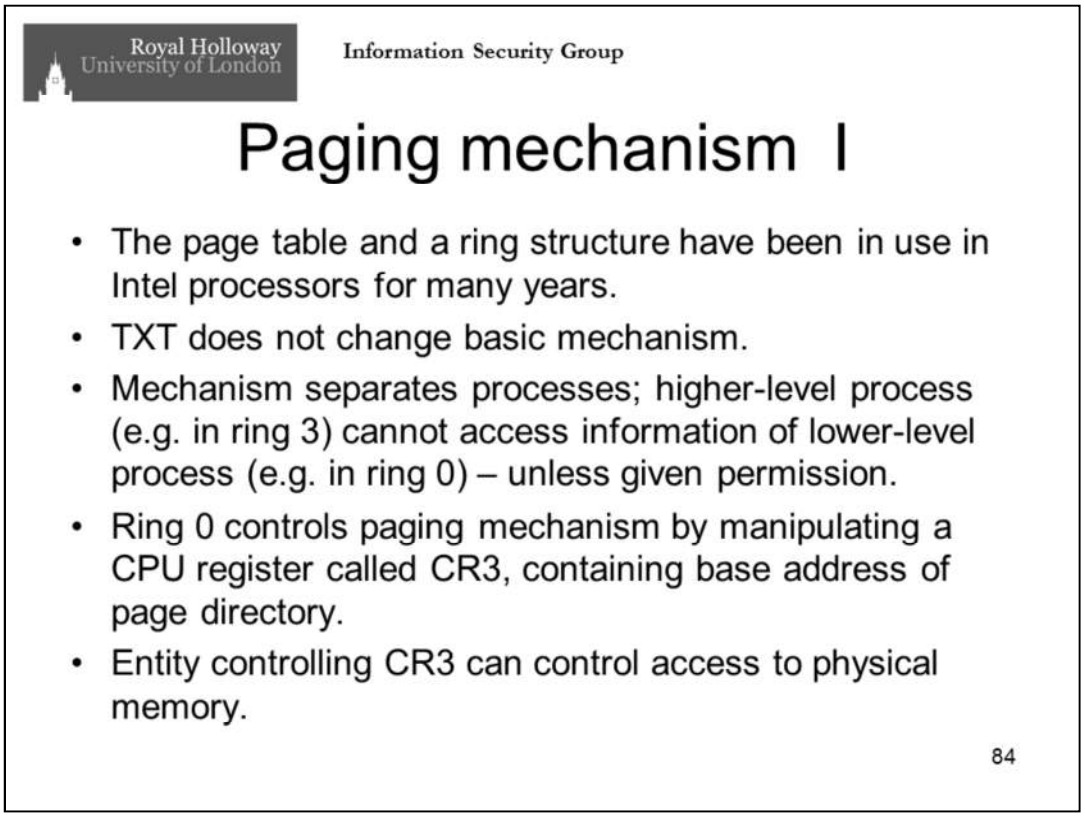
These are present to securely manage the four access techniques we have just described. We describe these protection mechanisms in the following slides.



The VMM provides a large portion of the protection, as it:

- manages the paging mechanism and the configuration of the VT-d mechanism;
- cooperates with the STM.

We next look at each of the four page protection mechanisms in greater detail.



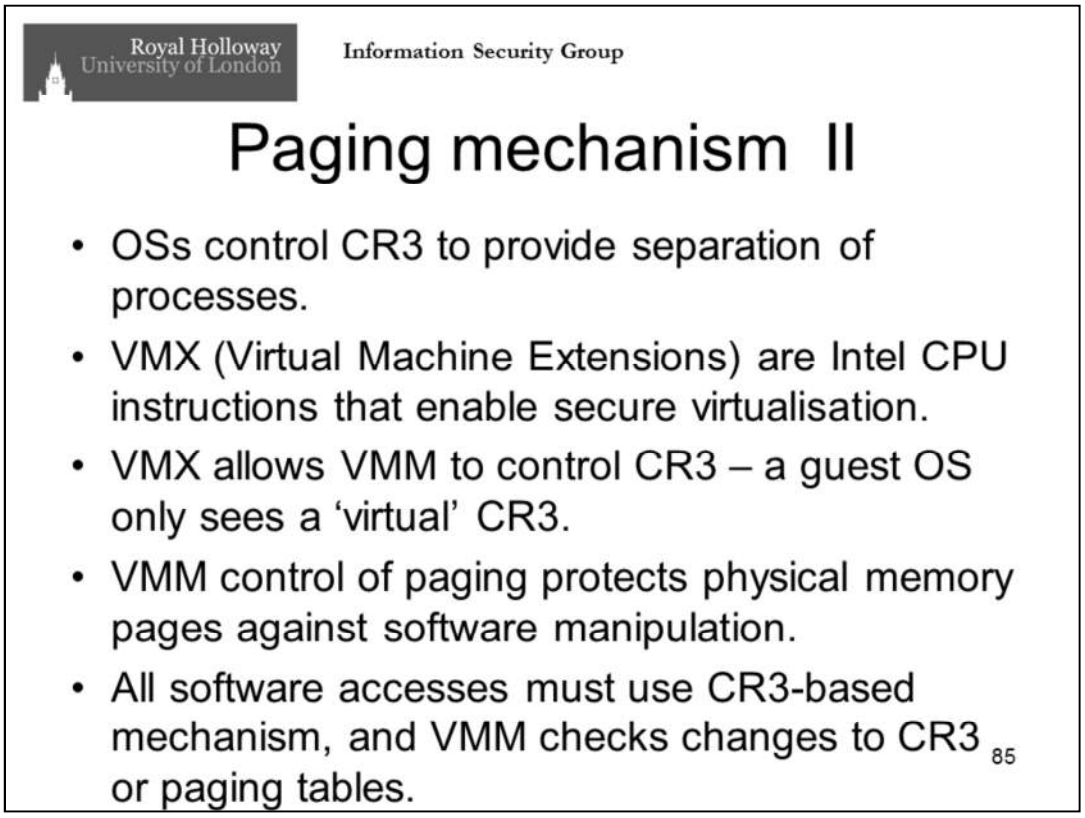
This mechanism, using a page table and a ring structure, has been in use in Intel processors for many years.

TXT does not change the basic mechanism.

The mechanism separates processes using rings, so that a higher-level process (e.g. in ring 3) has no access to the information of a lower-level process (e.g. in ring 0) – unless given permission.

The ring 0 processes control the paging mechanism by manipulating a CPU register called CR3 – this register contains the base address of the page directory.

Any entity controlling CR3 can control access to physical memory.



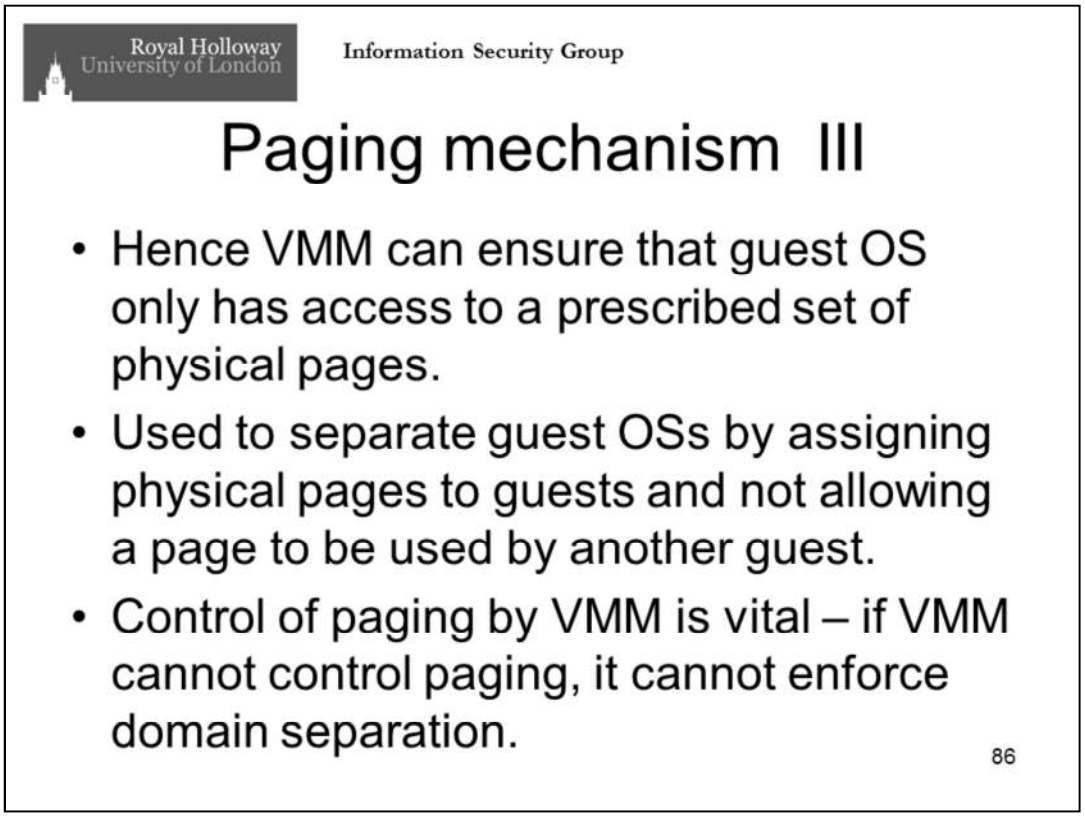
Operating systems control CR3 to provide separation of processes.

VMX (Virtual Machine Extensions) are Intel CPU instructions that enable secure virtualisation.

VMX allows a VMM to control CR3 – a guest operating system only sees a 'virtual' CR3.

VMM control of the paging mechanism protects physical memory pages against software manipulation.

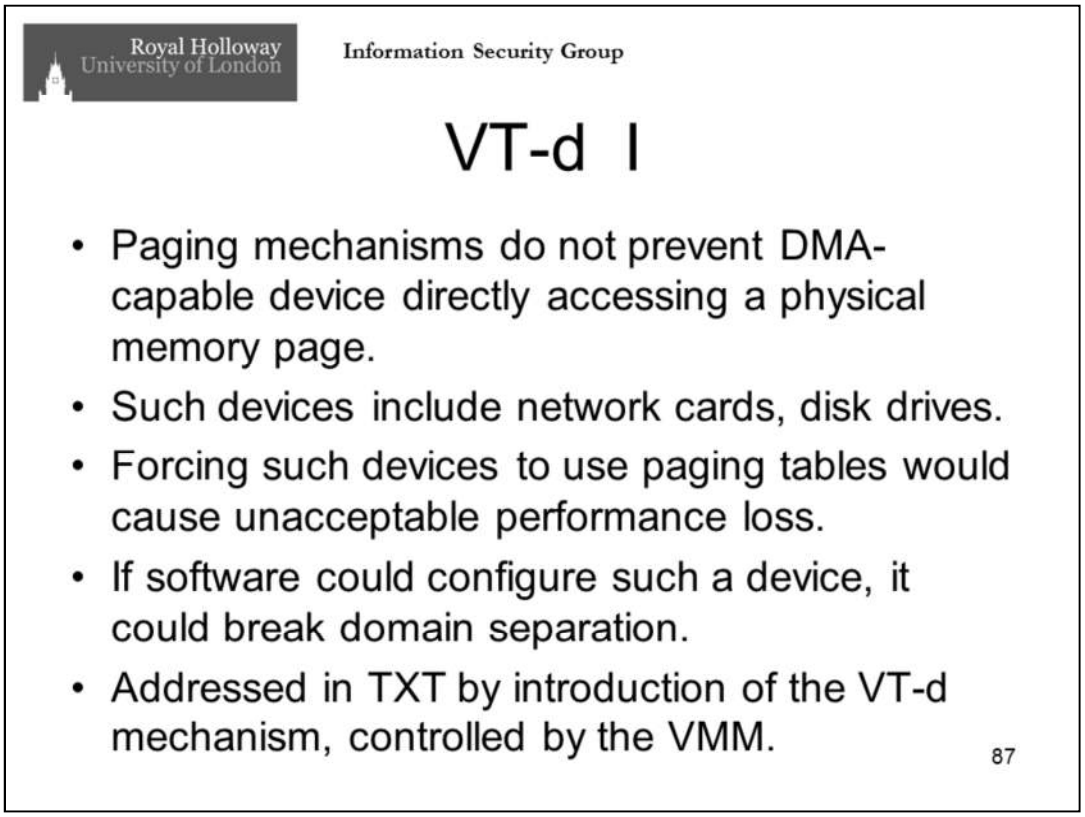
All software accesses must use the mechanism based on CR3, and the VMM validates any changes to CR3 and the paging tables.



This mechanism enables the VMM to ensure that a guest OS only has access to a prescribed set of physical pages.

This control is used to separate guest OSs, by assigning a physical page to a guest and never allowing such a page to be used by any other guest.

Control of the paging mechanism by the VMM is fundamentally important – if the VMM cannot control paging then it cannot enforce domain separation.



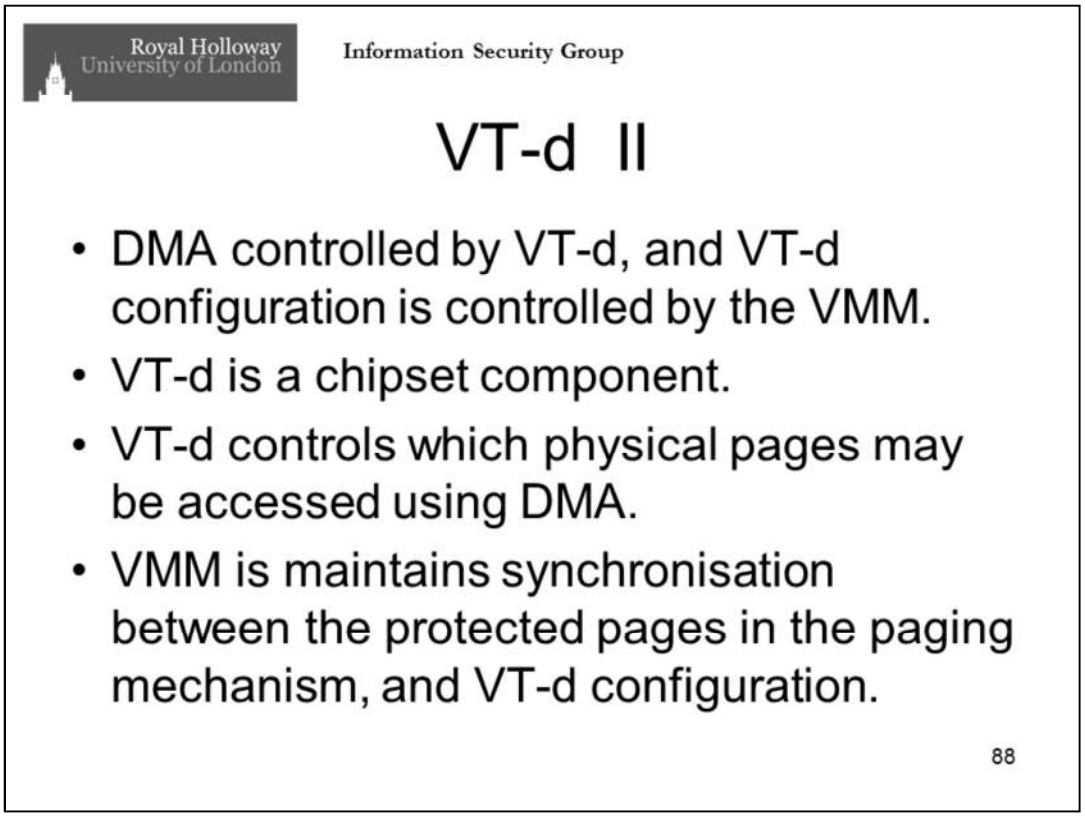
The paging mechanisms do not control the ability of a DMA-capable device to directly access a physical memory page.

Such devices include network cards and disk drives.

Forcing such devices to use the paging tables would cause an unacceptable performance hit (they need high speed access to memory).

Configuration of such a device by malicious software would allow the software to break the domain separation established by the VMM and the paging mechanism.

This is addressed in TXT by the introduction of the VT-d mechanism, controlled by the VMM.

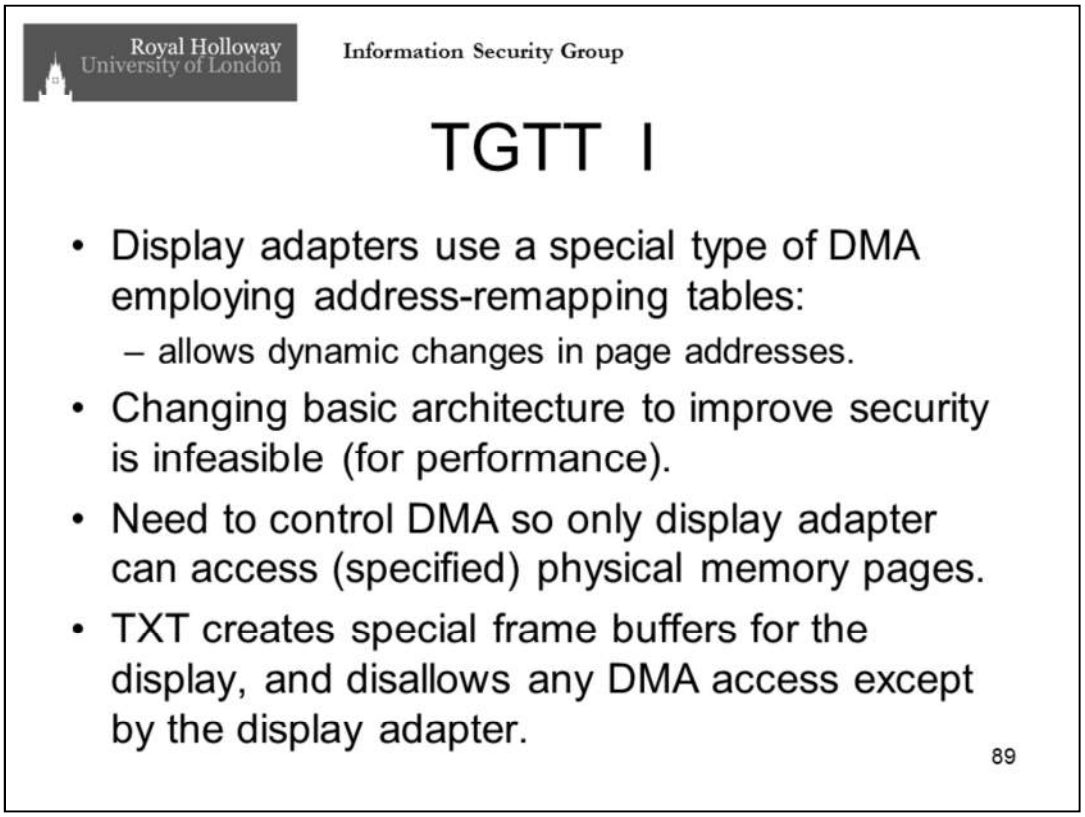


That is, while a DMA access bypasses the VMM and the paging mechanism, DMA is controlled by the VT-d mechanism, and the configuration of VT-d is (in turn) controlled by the VMM.

VT-d is a chipset component.

VT-d controls which physical pages may be accessed using DMA.

The VMM is responsible for maintaining synchronisation between the protected pages in the paging mechanism, and the configuration of VT-d.

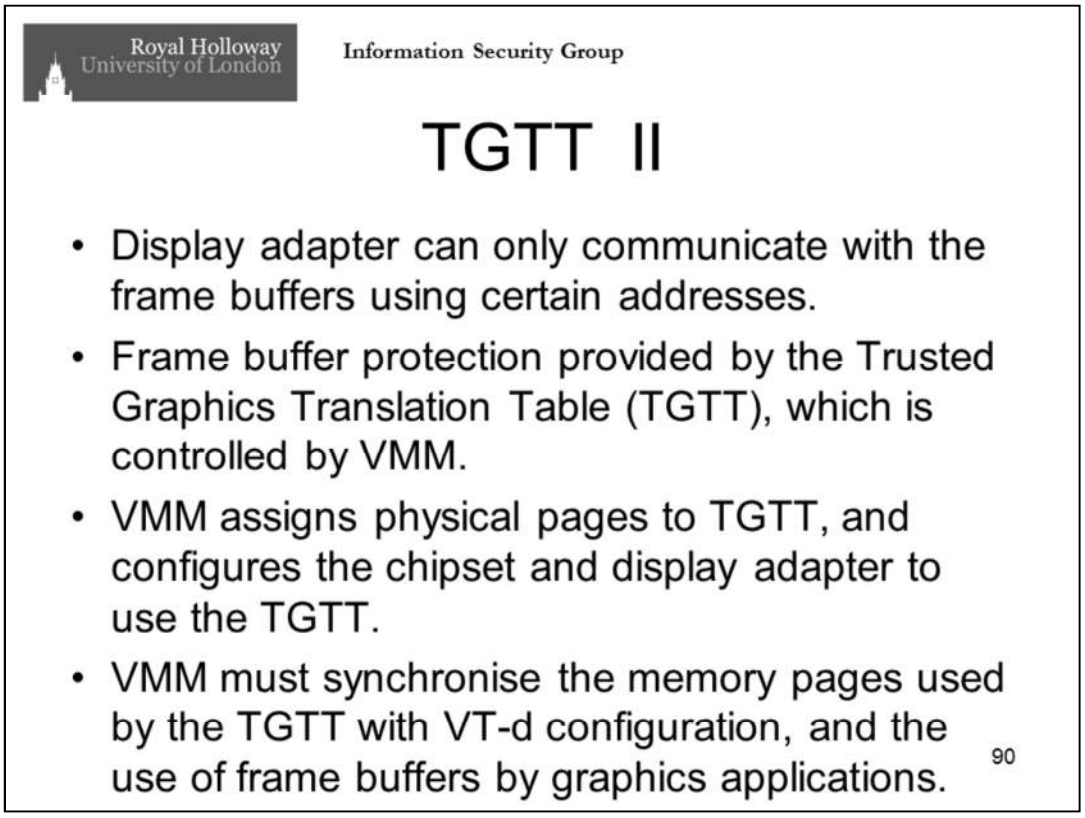


To create and display screens quickly and efficiently, a display adapter uses a special type of DMA employing address-remapping tables to define what is written where. Remapping allows dynamic changes in page addresses to meet the needs of the display adapter.

Changing the basic architecture to improve security is infeasible (for performance reasons).

We need to control this special type of DMA in such a way that only the display adapter is given access to (specified) physical memory pages, and so that no other devices can DMA to these pages.

To achieve this, TXT creates special frame buffers for the display, and disallows any DMA access except by the display adapter.

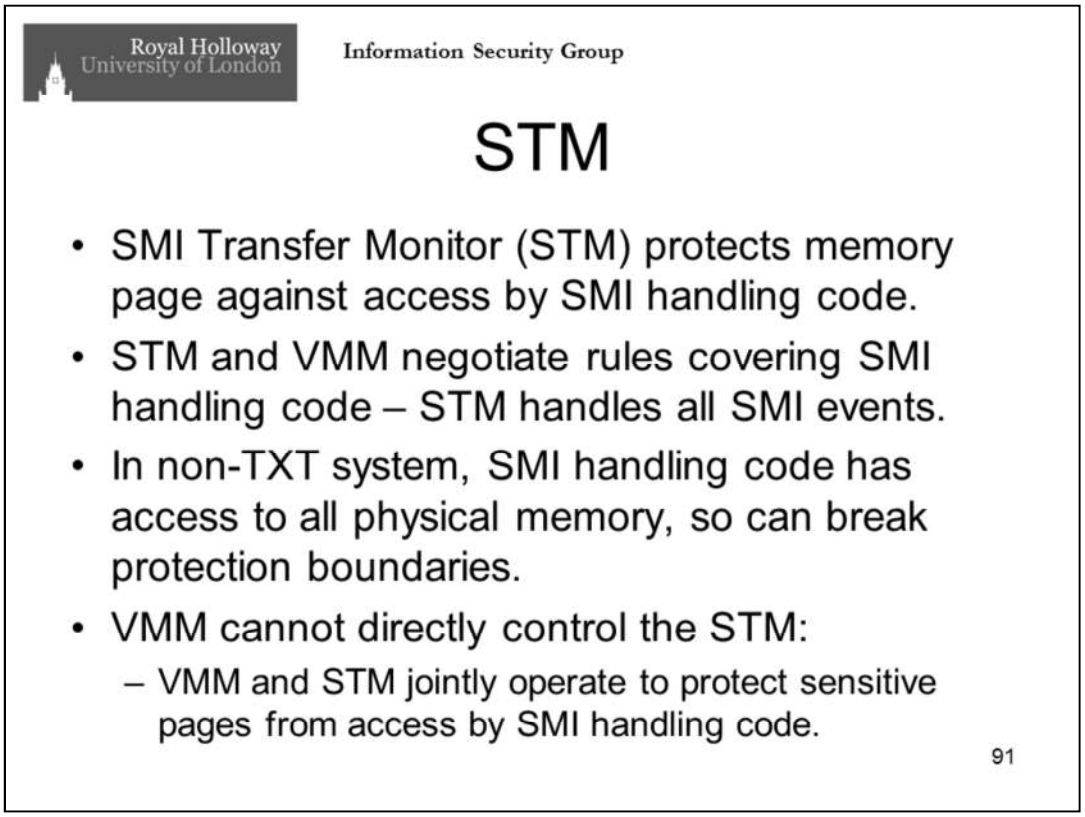


Also, the display adapter can only communicate with the frame buffers using certain addresses.

The frame buffer protection is provided by the Trusted Graphics Translation Table (TGTT), which is itself controlled by the VMM.

The VMM assigns the physical pages to the TGTT, and configures the chipset and display adapter to use the TGTT.

The VMM must synchronise the memory pages used by the TGTT with the VT-d configuration and the use of frame buffers by graphics applications.

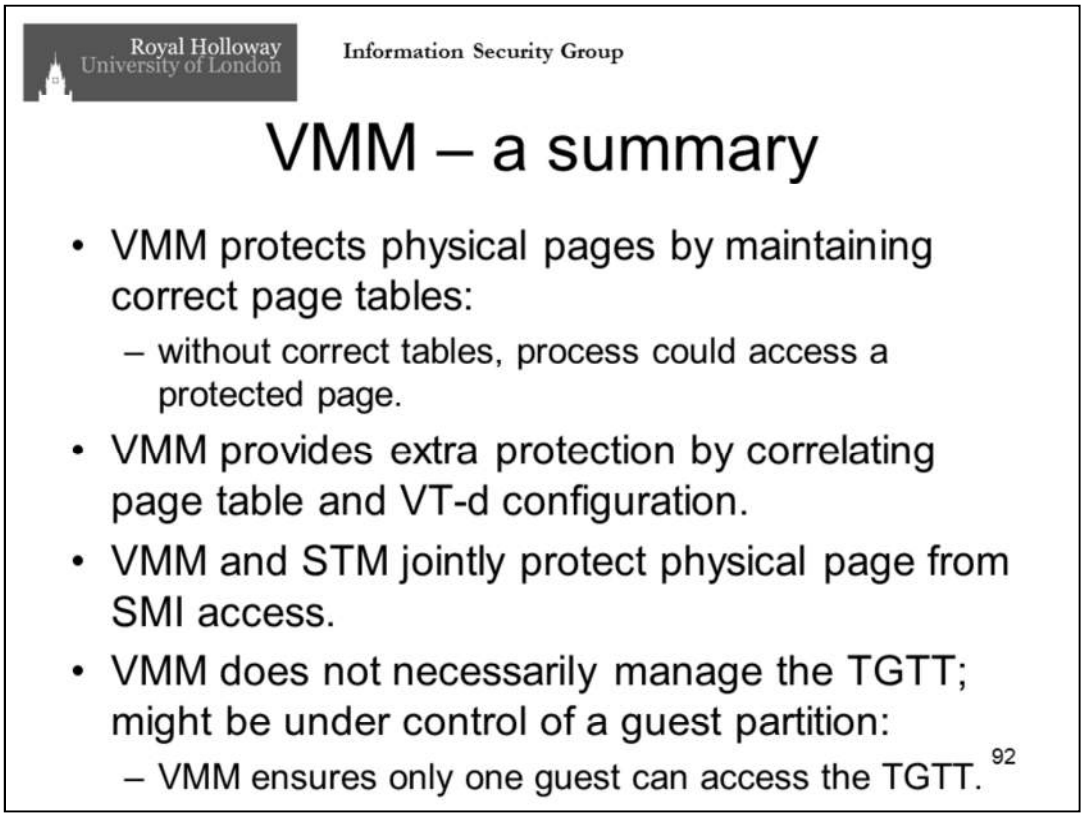


The SMI Transfer Monitor (STM) protects a memory page against access by normal SMI handling code.

The STM and VMM negotiate rules governing what SMI handling code should and should not do – the STM then handles all SMI events.

In a non-TXT system, SMI handling code has access to all physical memory, and can thus break protection boundaries.

The VMM cannot directly control the STM. However, the VMM and STM jointly operate to protect sensitive pages from access by SMI handling code.

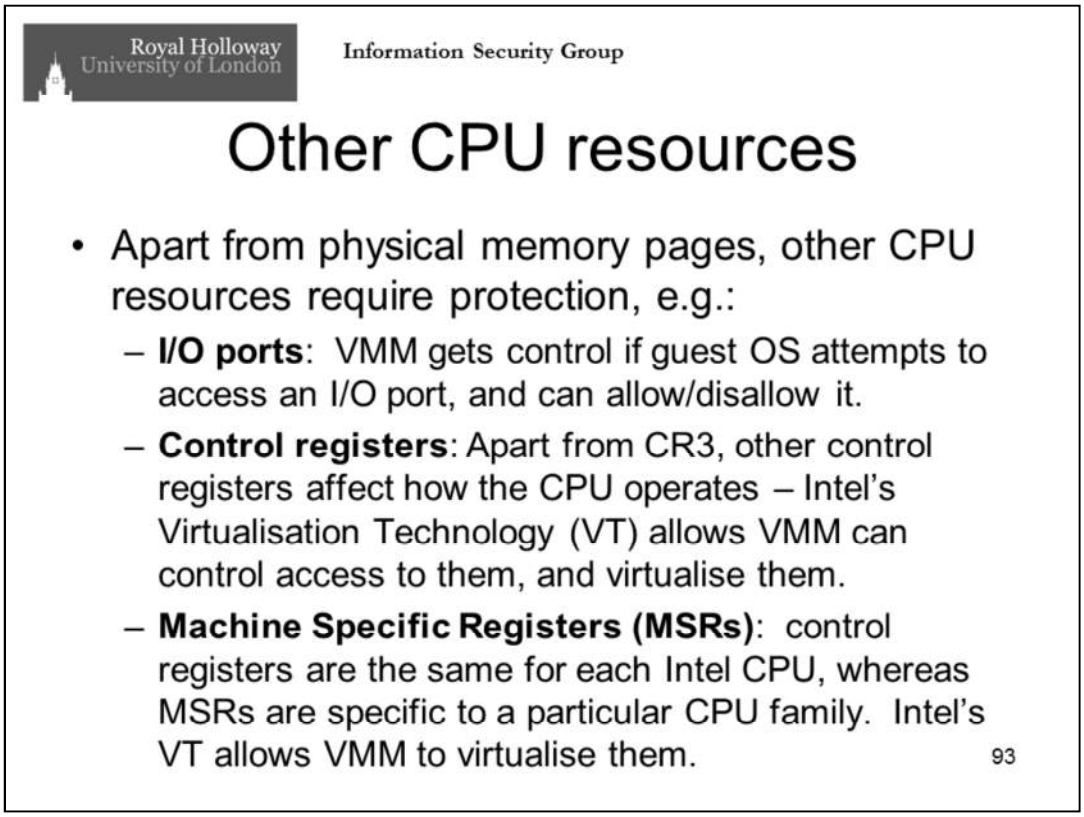


The VMM protects physical pages by maintaining correct page tables – without maintenance of correct tables, software processes could gain access to a protected physical page.

The VMM provides additional protection by enforcing correlation between page tables and the VT-d configuration.

The VMM, working with the STM, protects physical page from SMI access.

The VMM does not necessarily manage the TGTT, as it could be under the control of a guest partition. However, the VMM does ensure that only one guest has access to the TGTT.



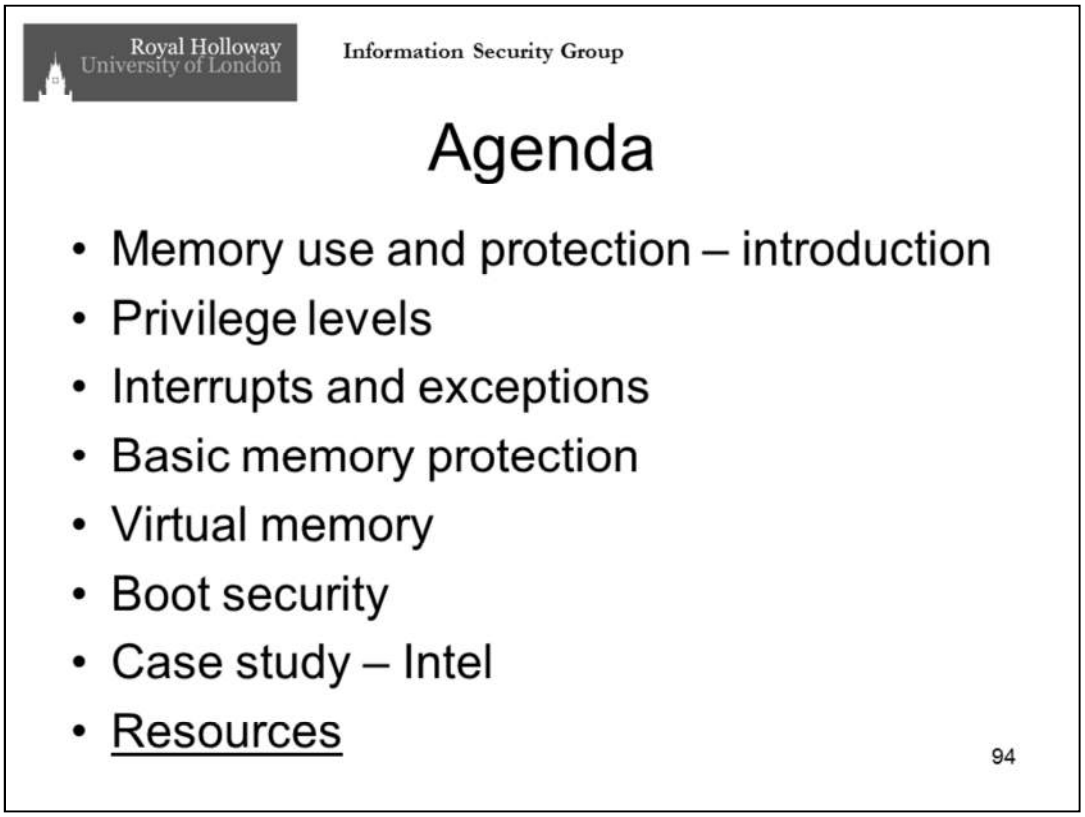
Apart from physical memory pages, other CPU resources require protection.

These include:

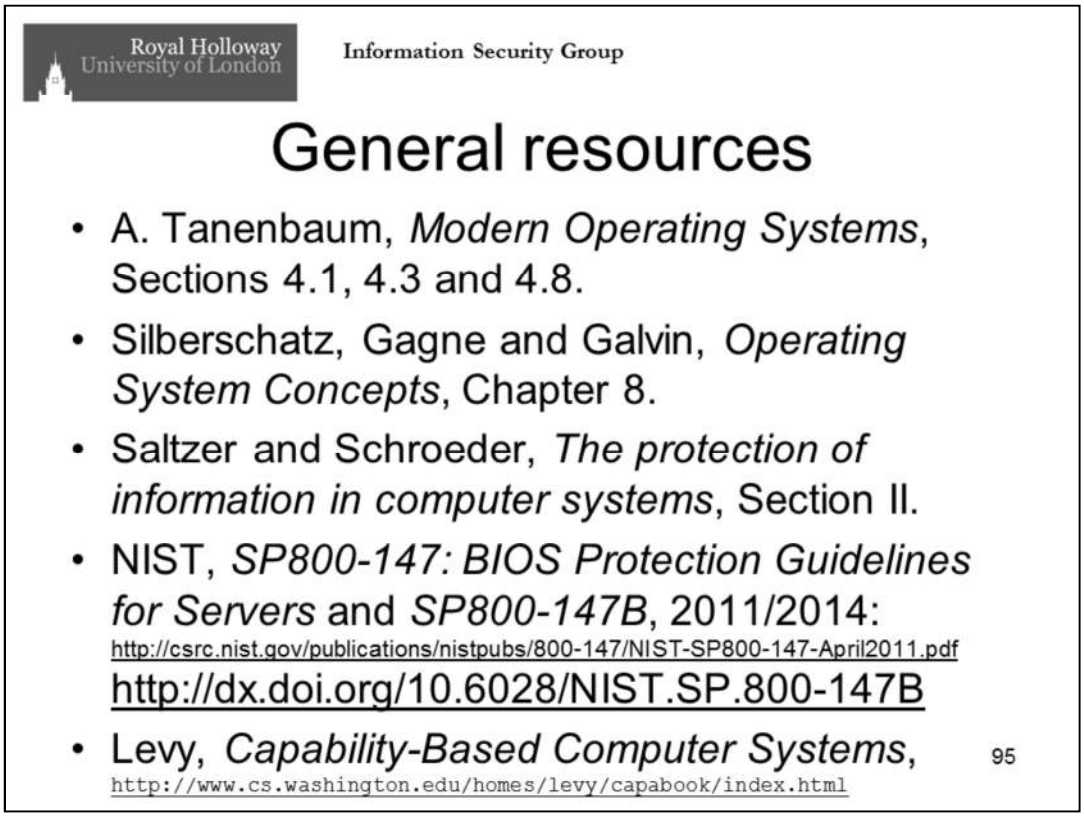
• **Input/output ports**: the VMM gets control whenever a guest OS attempts to access an I/O port, and can allow/disallow this attempt.

• **Control registers**: Apart from CR3, there are other control registers that affect how the CPU operates – Intel's Virtualisation Technology (VT) allows the VMM can control access to them, and provide virtualisation of them.

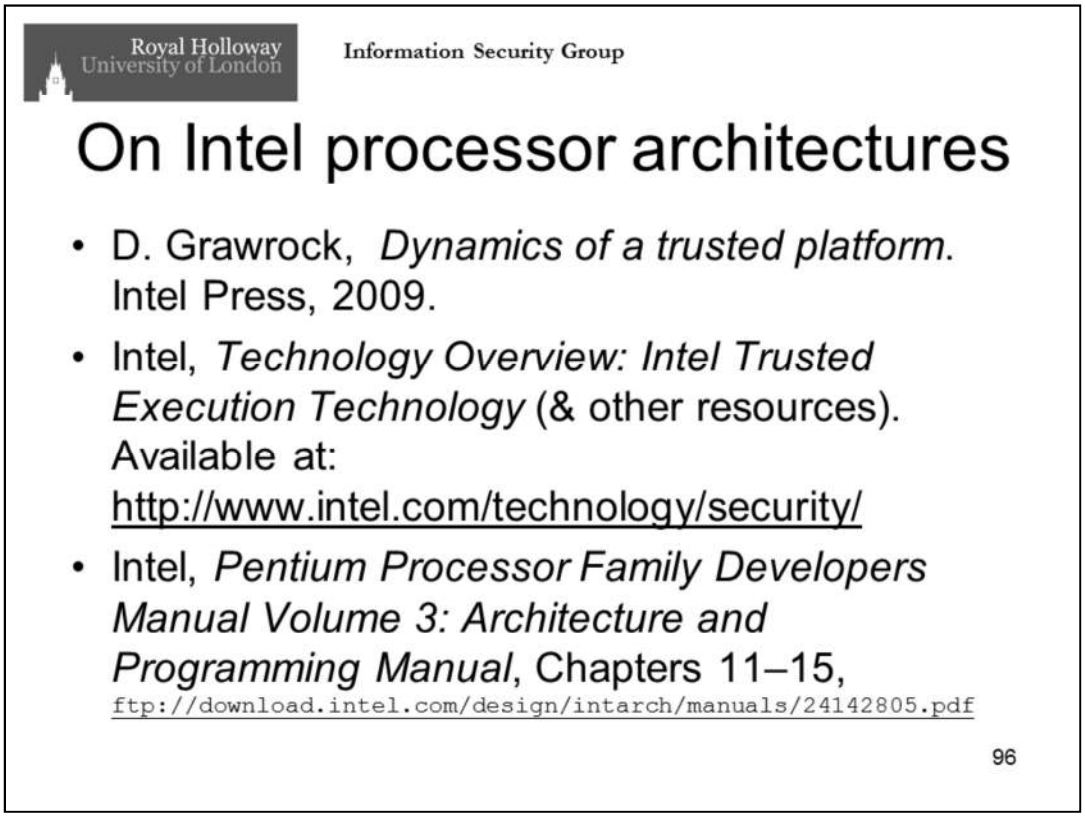
• **Machine Specific Registers (MSRs)**: the control registers are the same for each Intel CPU, whereas the MSRs are specific to a particular CPU family. Intel's VT allows the VMM to virtualise them.



We conclude by listing resources relevant to this part of the course.



- A. Tanenbaum, *Modern Operating Systems*, Sections 4.1, 4.3 and 4.8.
- Silberschatz, Gagne and Galvin, *Operating System Concepts*, Chapter 8.
- Saltzer and Schroeder, *The protection of information in computer systems*, Section II.
- NIST, *SP800-147: BIOS Protection Guidelines*, April 2011. Available at: http://csrc.nist.gov/publications/nistpubs/800-147/NIST-SP800-147-April2011.pdf
- NIST, *SP800-147B: BIOS Protection Guidelines for Servers*, August 2014. Available at: http://dx.doi.org/10.6028/NIST.SP.800-147B
- Levy, *Capability-Based Computer Systems*. Available at: http://www.cs.washington.edu/homes/levy/capabook/index.html



- D. Grawrock, *Dynamics of a trusted platform*. Intel Press, 2009.
- Intel, Technology Overview: Intel Trusted Execution Technology (and other resources). Available at: http://www.intel.com/technology/security/
- Intel, *Pentium Processor Family Developers Manual Volume 3: Architecture and Programming Manual*, Chapters 11–15,. Available at: ftp://download.intel.com/design/intarch/manuals/24142805.pdf