User– and Kernel Mode, System Calls, I/O, Exceptions

1 Introduction

- In the last two hardware lectures, we are going to look at the mechanisms provided by the CPU to manage the system hardware: the CPU itself, the memory and the peripheral devices.
- We will also look at the two privilege modes, **user mode** and **kernel mode**, the mechanisms to cross between then, and the reason why they exist in the first place.
- Thie latter leads on to the issue of security, and how the above provide basic system security mechanisms which can then be built on by the software layers above.

2 User Mode and Kernel Mode: Why?

- From a programmer's point of view, the system is the CPU used to execute instructions plus the memory used to hold instructions and data.
- This simplistic view might have been true when one person owned a microcomputer which ran one program at a time, but this is no longer the case.
- Now we have computers which:
  - run multiple programs at the same time, and each needs its own memory space.
  - switch the CPU quickly between programs to give the illusion that they are running at the same time.
  - hold documents for several users, with the expectation that each user can protect their own files.
  - allow multiple network connections simultaneously, where each connection may be dealing with sensitive data.
- If every program had unfettered access to the CPU, main memory and the peripheral devices, all concepts of separation of programs and the data in memory, on disk etc. would not exist.
  - A program could look at all memory locations, including that of other programs, as well as read all the data on all of the attached disks, and read all the data being sent across the network.
- To prevent this, we need the CPU to have at least two privilege levels.
- In **kernel mode**, the CPU has instructions to manage memory and how it can be accessed, plus the ability to access peripheral devices like disks and network cards. The CPU can also switch itself from one running program to another.
- In **user mode**, access to memory is limited to only some memory locations, and access to peripheral devices is denied. The ability to keep or relinquish the CPU is removed, and the CPU can be taken away from a program at any time.
- Now, all programs will be run in user mode, and this prevents them from accessing the data in other programs, as well as preventing the disk etc.
- But, what software runs in kernel mode, and how does it manage all the switching between the programs and ensuring that access to resources is correctly permitted/denied, as well as allocating resources equitably?

- This is the job of the **operating system**, which we will cover in the second half of the subject.

3 User Mode and Kernel Mode: How?

- In the CPU, there is a register or flag which records what mode the CPU is in: user or kernel mode.
  - On some CPU architectures like the Intel IA-32, there are several privilege levels known as *rings*.
- The CPU boots in kernel mode, with full access to the system hardware. It then proceeds to load and start the operating system running.
- At some point, the operating system prepares a program to run, e.g. by loading the instructions from the disk, setting up the memory for the program etc.
- Just before jumping to the first instruction, the CPU lowers its privilege level by marking the mode flag as user mode.
- Thus, when the program starts execution, it is already in user mode, and is limited in what hardware operations and accesses it can perform.

4 Switching from User to Kernel Mode

- It should be obvious that we don't want user-mode programs to easily switch back to kernel mode; that would make the two privilege levels useless.
- At the same time, a user-mode program does need to cross into kernel mode.
  - For example, when a program wants to read from disk, or get a line of text from the keyboard, it cannot do this in user mode.
  - The only software which can is the operating system.
  - The program needs to ask the operating system: "please access the hardware on my behalf and get me some input".
- We need a mechanism whereby:
  - a user-mode program can switch into kernel mode,
  - but have no control over the instructions which will be performed in kernel mode.
- This mechanism is the **system call**, which is implemented in the CPU as the **trap instruction**.
  - although different ISAs give this instruction different names, e.g. `syscall` on the MIPS ISA.
- Here is how it works:
  - The user-mode program places values in registers, or creates a stack frame with arguments, to indicate what specific service it requires from the operating system.
  - The user-mode program then performs the **trap** instruction.
  - Immediately, the CPU switches to kernel mode, and jumps to instructions at a fixed location in memory.
  - These instruction, which are part of the operating system, have memory protections so that they cannot be modified by user-mode programs, and may also be unreadable by user-mode programs.
  - The instructions, known as the trap or system call **handler**, read the details of the requested service + arguments, and then perform this request in kernel mode.

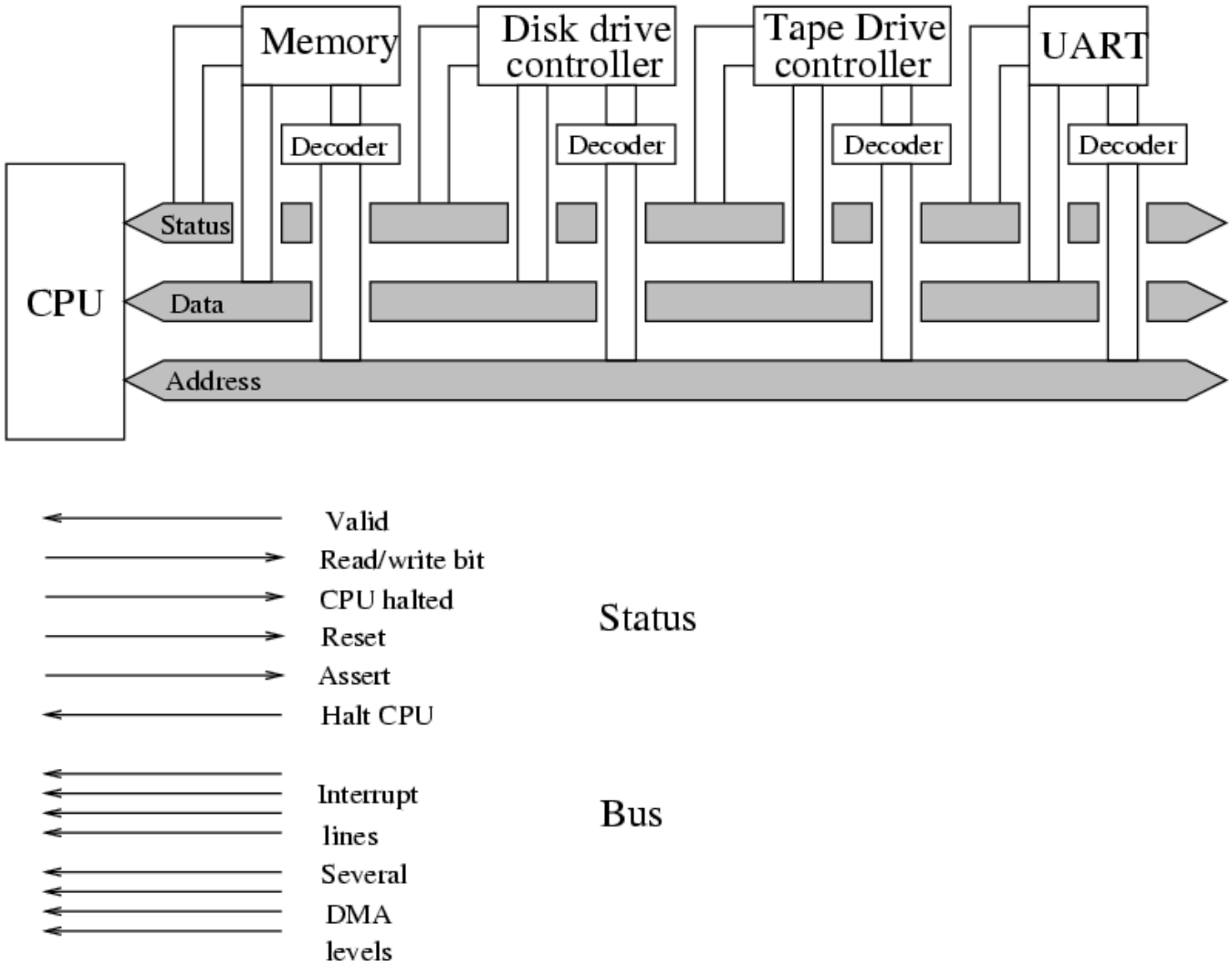
    - With the system call done, the operating system resets the mode to user-mode and returns from the system call, or there is an instruction to do both at the same time.
- From the point of view of the user-mode program, the trap instruction performs "magic" in a single instruction, with the results available at the next instruction. In reality, the CPU jumps in kernel mode to the system call handler, which does the work and returns to the program in user-mode.

4.1 Trust in the Operating System

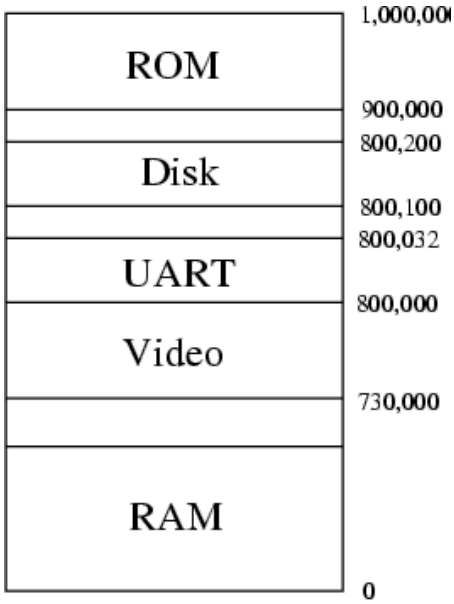
- Any software running in kernel mode has full access to everything. Therefore, we have to trust it: we have no choice.
- Therefore, we have to trust the system software which is the operating system (OS).
    - or more precisely, the part of the OS which run in kernel mode, known unsurprisingly as the **kernel**.
- We also need to ensure that, once running, the machine code which is the OS cannot be modified, or new code inserted into the OS which can be made to run in kernel mode.
- We will look at memory protection mechanisms in the next lecture.

5 Input/Output Devices: Interconnect and Access

- The computer must be able to do I/O, so as to store data on long-term storage devices, and to communicate with the outside world.
- However, we don't want the CPU to be burdened with the whole task of doing I/O, i.e controlling every electrical & mechanical aspect of every peripheral.
- Therefore, most devices have a *device controller* which takes device commands from the CPU, performs them on the actual device, and reports the results (and any data) back to the CPU.
- The CPU communicates with the device controllers via the address, data and control busses. Therefore, the device controllers usually appear to be memory locations from the CPU's point of view.
- Each device controller has a **decoder** which tells the device if the asserted address belongs to that device. If so, parts of the address and the data is written to/from the device.



- Usually, this means that the device controller is **mapped** into the computer's address space. And because main memory has its own decoder, we can say that the locations in main memory are also mapped into the computer's address space:



I/O Decoding Addresses

- Let's say we have an external modem connection to the Internet. To communicate with the modem, it is plugged into the computer's serial port. Inside the computer, this port is controlled by the

UART (universal asynchronous receive/transmit) device.

- In the diagram above, the UART address decoder would decode the addresses 800,000 to 800,031 on the address bus, which is 32 addresses. It will ignore addresses outside this range. For any addresses within the range, the decoder will pass the register values 0 to 31 to the UART. Assume this UART uses the following addresses:

Decoded Location	Real Location	Use of this location	Format of this location
0	800,000	Output format	Speed (4), Parity (2), Stop bits (2)
1	800,001	Output status register	
2	800,002	Output character	
3	800,003	Input format	Speed (4), Parity (2), Stop bits (2)
4	800,004	Input status register	
5	800,005	Input character	

- These special addresses are known as **device registers**, and are similar to the registers inside a CPU. The output status register controls the speed (bits per second) of the outbound traffic, and the framing of the data (parity and stop bits). The input status register does the same for the input traffic.
- Because a modem is so slow, compared to the CPU, we need a buffer to store one character as it is being transmitted, and also a buffer to store an input character as it arrives. The output and input character registers do this.
- How does the UART indicate that a character has arrived, or if the character in the output character register has been transmitted? The input and output status registers set various flag bits to indicate the success (or failure) of the last operation.
- With that out of the road, let's see how the operating system would transmit a character to the mode. Before anything else, the OS must set up the output transmit characteristics:
  - the CPU asserts the address 800,000 on the address bus.
  - it places a word of data onto the data bus. This describes the format of output to be used by the UART.
  - it asserts `write' on the r/w line.
  - it waits a period of time.
  - if no `valid address' returned, error.
- Then, to output a character, the character is sent to 800,002 as above. The UART *latches* the character off the data bus, and it is transmitted over the serial line at the bit rate set in the output format.
- Input from a device is more complicated. There are three types: polling, interrupts, and direct memory access (DMA). We will leave DMA until later.
- With **polling**, the UART leaves the input character in the input character register at the address 800,005 and an indicator that a character has arrived in the input status register at the address 800,004. The CPU must periodically scan (i.e. read) this address to determine if a character has arrived.
- This periodic polling causes problems, as the CPU must be checking the UART status registers.
- But remember: only the operating system in kernel mode can access these device registers.

- So, we must keep the CPU in kernel mode, and loop polling the input registers. But then, user-mode programs cannot run.
- The only alternative is to return to user mode, but if we do that, then the program may not call the operating system for ages, and several characters may arrive in from the modem. Only the last one will be there: all the other ones will be overwritten.

6 Interrupts

- An alternative way for the system to find out when input has arrived, or when output has been completed, is to use **interrupts**.
- If a computer uses interrupts for I/O operations, a device will assert an *interrupt line* on the status bus when an I/O operation has been completed. Each device has its own interrupt line.
- For example, when a character arrives, the UART described above asserts its interrupt line. This sends a signal in to the CPU along the status bus. If the interrupt has priority greater than any other asserted interrupt line, the CPU will stop what it is doing, and jump to an **interrupt handler** for that line. This interrupt handler is a section of machine code placed at a fixed location in main memory.
- Here, the interrupt handler will collect the character, do something with it and then return the CPU to what it was doing before the handler started i.e the program running before the interrupt came in.
- Generally speaking, interrupt handlers are a part of the operating system, and run in kernel mode.
- Interrupts are prioritised. The CPU is either running the current program, or dealing with the highest interrupt sent in from devices along the status bus. If an interrupt's priority is too low, then the interrupt will remain asserted until the other interrupts finish, and the CPU can handle it. Alternatively, if a new interrupt has a priority higher than the one currently being handled by the CPU, then the CPU diverts to the new interrupt's handler, just as it did when it left the running program.
- The CPU has an internal status register which holds the value of the current interrupt being handed. Normal programs run at a level below the lowest interrupt priority, i.e interrupt priority 0.

6.1 Interrupt Vectors

- Just as with the TRAP instruction, to ensure that the CPU goes back to what it was doing, the value of the Program Counter before the interrupt is stored on the stack or in a register before the CPU jumps to the first instruction in the interrupt handler.
- Because a low-priority interrupt can itself be interrupted by a higher-priority interrupt, old values of the Program Counter are *stacked* in interrupt-level on the stack. The last instruction in an interrupt handler must unstack an old PC value, and put it back into the Program Counter. The **ReTurn from Interrupt** or RTI instruction is what is used here to do this.
- Each interrupt level has its own interrupt handler. How does the CPU know where each handler is stored in main memory? A table of *vectors* is kept in main memory for each interrupt level. It holds the address of the first instruction in the appropriate interrupt handler.

Address	Holds Address Of	For Example
0	Reset Handler	1,000,870
1	IRQ 1 – Keyboard	1,217,306
2	IRQ 2 – Mouse	1,564,988

15	IRQ 15 – Disk	1,550,530
16	Zero Divide	1,019,640
17	Illegal instruction	1,384,200
18	Bad mem access	1,223,904
19	TRAP	1,758,873

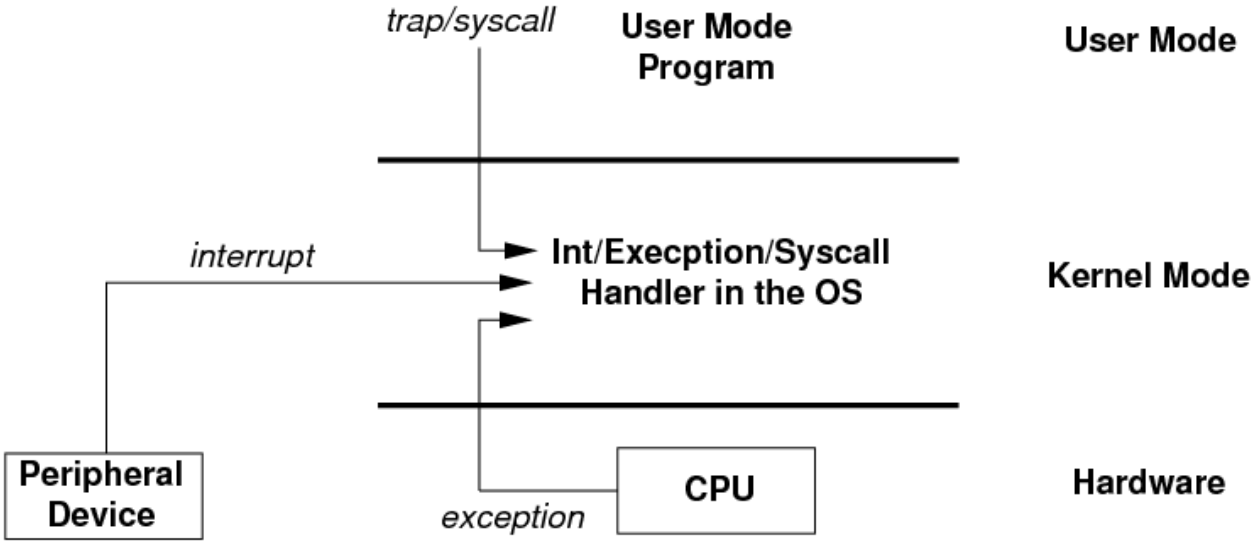
- The above fictitious table shows where the vectors might be stored in main memory, their value (i.e. where the first address of each interrupt handler is), and what interrupt is associated with each.

7 Exceptions

- We have seen that there are two ways to get into kernel mode:
  1. A user–mode program can execute a TRAP instruction to perform a system call. From the program's point of view, they know that the operating system will perform the request, but have no idea how long it will take.
  2. An interrupt can arrive, raising the CPU's interrupt level from 0 to some number N. The user–mode program is oblivious to the fact that the CPU has been taken away from it. The operating system will deal with the I/O, and eventually return back to the user–mode program.
- The latter has one important implication: there is no guarantee of the time interval between any two user–mode instructions, as an interrupt can arrive between them and force a change into kernel mode!
- There is a third way to get into kernel mode, and that is when an **exception** occurs on the CPU.
- If the running program performs some illegal operation, e.g. a divide by zero, then the CPU issues an exception. Again, the CPU switches to kernel mode, and is diverted to some location unknown to the program where the machine code exists to deal with the exception.
- Unlike system calls, which are predictable because they require a specific instruction, exceptions can occur at any time in a program's execution.
- For example, the MIPS `div $t0, $t1, $t2` instruction may or may not cause an exception: it depends on the value in $t2. If it occurs, the exception will occur during the instruction's execution.
- The exception handler's job is to decide if the problem is recoverable or not. If the problem cannot be recovered, the program is terminated without the CPU ever returning back to the program.
- If the exception can be recovered, the handler takes the steps required to do this, and then returns the CPU back to the culprit instruction, so that it can be performed again. In this situation, the program will have no idea that anything went wrong.
- As an example of a recoverable error, let's say a program has been allocated 1Mbyte of stack memory, but it is running a recursive function which has used nearly all of the stack.
- On the next function call, the $sp is decremented and falls below the allocated memory. Any reads or writes to the stack will access the stack which doesn't exist, and this will cause an exception.
- The exception handler, seeing the access just below the stack memory, can allocate some more memory to the stack, and allow the program to restart. As with system call handlers, exception handlers are placed in the operating system and run in kernel mode.

7.1 Exceptions to Inform of User–Mode Violations

- Exceptions are caused by the program when it performs something illegal. Just like an interrupt, an exception can occur at any time.
- Rather than coming from the program as with a system call, the CPU detects the illegal action and sends an exception to *itself*.



- Each CPU is designed to detect certain illegal actions. Examples are:
  - illegal maths operations: divide by zero, overflow.
  - data alignment issues: misaligned mutibyte data reads or writes.
  - invalid memory access: access to memory which does not exist.
- Exceptions can also be used to detect when a user–mode program attempts to perform an operation which is kernel–mode only, e.g.
  - access to memory locations which are only available in kernel mode, e.g. the device register locations.
  - attempt to perform certain CPU instructions which can only be performed in kernel mode.
- As with the other illegal actions, these cause an exception which diverts the CPU to kernel mode to deal with the problem, usually by terminating the running program.

7.2 Virtualising the CPU

- In a later subject, we will tackle the topic of virtualisation, where a CPU is virtualised on top of itself.
- This is done, for example, to run Windows on top of Linux, or to run multiple copies of Windows on top of Windows.
- Each simulated instance is known as a **virtual machine**.
- How is this done? We are going to have to give the virtual machine the ability to run both in user mode and in kernel mode.
- The solution is to run the virtual machine in user mode only, but fool it into thinking that it can do kernel mode actions.
- Now, when the virtual machine tries to do a kernel–mode–only action:
  - An exception is sent by the CPU to itself.
  - The CPU is diverted to the *real* operating system running on the hardware.

  - The OS looks at what the simulated virtual machine was trying to do.
  - The OS performs the action, or simulates the action, e.g. an access to a virtual disk might be translated into an access to a file which holds the virtual disk's contents.
  - The OS cleans up, and returns back to the virtual machine in user mode.
  - The virtual machine thinks that it was successful in performing the kernel–mode action.
- Every time you run VMWare or VirtualBox or Qemu or the other machine simulators, this is exactly what happens thousands of times a second!

File translated from TEX by TTH, version 3.85.
On 14 Feb 2012, 11:30.