Summary of chapter 2 – The Kernel Abstraction:

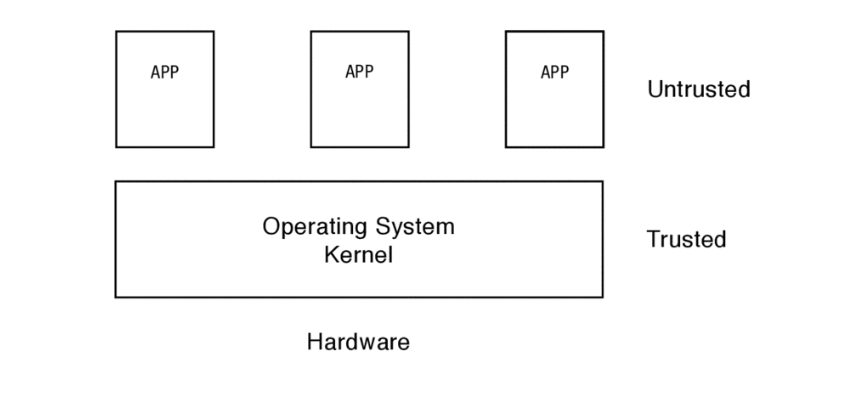
First of all, what is an operating system?

An operating system is a system software that manages the computer’s hardware and software and provide common services for computer programs.

It’s vital for the OS to have good protection. That’s one of the central roles of the OS. We want to isolate misbehaving applications, and separate users so they do not disturb or do harm on each other. As mentioned in chapter 1, protection is essential to achieving several of the OS goals:

* Reliability – Does it what it supposed to do?
* Security – Limit the applications scope.
* Privacy – Each users must be isolated and separate.
* Fair resource allocation – No program should be able to gather any amount of processing time etc.

Implementing protection is the job of the operating system kernel.



It’s important to mention the definition of a process; the execution of an application with restricted rights. Another definition is that a process is an instance of a program.

## 2.1 The process abstraction

To run a program, the OS copies the instructions and data from the executable image (a sequence of machine instructions) into physical memory.

*The execution stack* holds the state of local variables during procedure calls.

*The heap* is also a memory region, for any dynamically allocated data structures the program might need.

To run multiple copies of the same program, the operating system can make multiple copies of the program’s **instructions**, **static** **data**, **heap** and **stack** in memory. Most operating systems *reuses* memory wherever possible. So they only store one single copy of a program’s instructions when a program is run several times.

A program can have zero, one or more processes executing it. For each instance, there is a process with its own copy of the program in memory.

The OS stores information about a process on a data structure called the **process control block.** It holds information such as where it is stored in memory, where its executable image resides on disk, which user executed it, what privileges and so on.

Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.

## 2.2 Dual Mode Operation

A good way to define what the dual mode operation is to say we have two modes of operations:

*The kernel mode* and *the user mode*.

User mode has not privileged instructions, whereas the kernel mode has access to privileged instructions.

What hardware is necessary to let the OS protect applications and users from one another, yet also let user code run directly on the processor? At a minimum, we need three things:

* **Privileged instructions –** All potentially unsafe instructions are prohibited when executing in user mode
* **Memory protection –** All memory access outside of a process’ valid memory region are prohibited when executing in user mode.
* **Timer interrupt –** The kernel must be able to periodically regain control from the current process

Every process has two extra registers, called base and bound. Base specifies the start of the process’ memory region, and the bound specifies the end. The kernel executes without the base and bound registers; it has access to the full memory region.

The use base and bound can provide protection, but not the following important features:

* **Expandable heap and stack –** The amount of memory allocated to a program is fixed. The OS may change the bound.
* **Memory sharing –** Memory cannot be shared between different processes.
* **Physical memory address -**
* **Memory fragmentation**

Therefore, most modern processors introduce virtual addresses. Every process’ memory will now start at the same place (for instance zero). Each process think that it has the entire machine to itself. The hardware translates these virtual addresses to physical memory locations.

This also makes the heap and stack expendable. If either the heap or stack grows beyond its allocated region, the OS can move it to a different larger region in physical memory, but it’s still on the same virtual address.

## 2.3 Types of Mode Transfer

*User to kernel mode*

There are three reasons for the kernel to take control from a user process.

Asynchronous:

**Interrupt:** An interrupt is an *asynchronous* signal to the processor that some external event has occurred and may require its attention. The processor hardware then saves the current execution state and starts execution a interrupt handler in the kernel.

Each different type of interrupts requires different handlers.

Synchronous:

**Processor exceptions:** A hardware event caused by a *user program* behaviour that causes a transfer of control to the kernel. The hardware saves the current execution state and starts running a exception handler in the kernel. Examples: divide by zero, or a process attempts to perform a privileged instruction.

Synchronous:

**System calls**: a user program request the kernel to perform an operation on its behalf. A system call is any procedure provided by the kernel that can be called from user level.

*Kernel to User Mode*

**New process.** To start a new process, the kernel must copy the program into memory, set the program counter to the first instruction, set the stack pointer to the base of the user stack and switch to user mode.

**Resume after an interrupt, processor exception or system call.** When the kernel is done doing its work regarding either an interrupt, processor exception or system call it changes the mode back to user level.

**Switch to a different process.** In some cases the kernel switches to a different process than the one before the interrupt. The kernel needs to save the process state in the PCB. The kernel can then resume a different process by loading its sate and then switch to user mode.

**User-level upcall.** Many OS provide user programs the ability to receive asynchronous notification of events. It’s similar to kernel interrupt handling, except at user level.

## 2.4 Implementing Safe Mode Transfer:

I think that context switch is the process of storing the state of a process, so it can be restored and executed later.

Context switch must be carefully crafted and relies on hardware support. Most OS has a common sequence of instruction both for entering the kernel and for returning to user level.

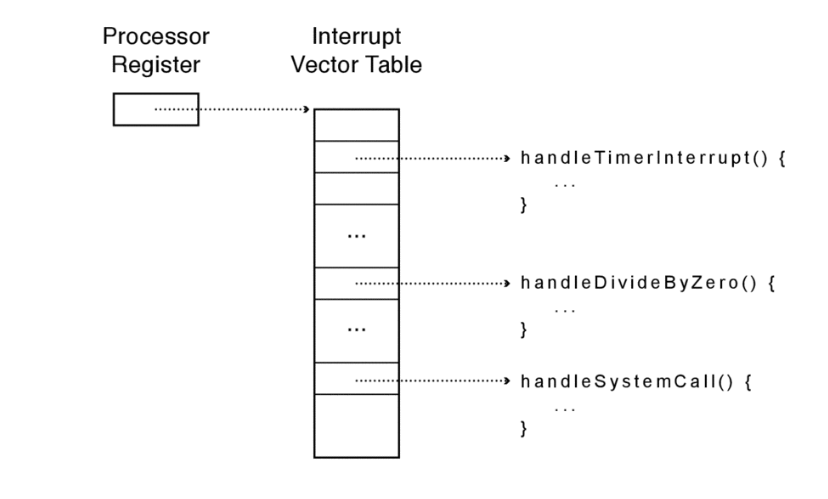
At a minimum, this common sequence must provide:

* **Limited entry to the kernel –** User programs cannot be allowed to jump to arbitrary locations in the kernel.
* **Atomic changes to the processor state –** When in user mode, the stack and the program counter point to memory region in the user process, but in kernel mode the program counter and stack point to memory locations in the kernel. Transitioning between the two is atomic – the mode, program counter, stack and memory protection are all changed at the same time.
* **Transparent, restartable execution –** The OS must be able to restore a process’ state after an interrupt occurred.

**Interrupt vector table:**

How does the processor know what code to run when the event is an divide-by-zero exception, a file read system call or a timer interrupt?

The processor has a special register that point to an area of the kernel memory called **the interrupt vector table**.



The interrupt vector table is an array of pointers, pointing to different handlers in the kernel. The general term is an **interrupt handler**.

**Interrupt stack:**

When an interrupt occur, where should the process’ state be saved, and what stack should the kernel’s code use?

On most processors, we have the **interrupt stack**. When an interrupt, processor exception or a system call causes a context switch into the kernel, the hardware changes the stack pointer to the base of the kernel’s interrupt stack. The hardware then saves the interrupted process’ state by pushing it’s registers onto the stack. Then the kernel’s handler is called.

**Interrupt masking:**

Interrupt arrive asynchronously. So potentially the processor could be executing a process, meanwhile an interrupt would show up. Therefore, we have an instruction called disable interrupts. However, this is not quite correct; the interrupt is only deferred (masked) and ignored. Once a corresponding enable interrupt instruction is executed, any pending interrupts are delivered to the processor.

If multiple interrupts arrive while interrupts are disabled, the hardware delivers them in turn when interrupt are re-enabled.

**Hardware support for saving and restoring registers**

The hardware saves the value of the processors’ stack pointer, program counter and processor status word *before* jumping through the interrupt vector table to the interrupt handler.

## Implementing Secure System calls:

Any time a process needs to perform an action outside of its protection domain — to create a new process, read from the keyboard, or write a disk block — it must ask the operating system to perform the action on its behalf, via a system call.

System calls provide the illusion that the OS Kernel is a set of library routines available to the user program. However, implementing system calls requires the OS to define a calling convention – how to name the system calls, pass arguments, and receive return values across the user/kernel boundary.

***THIS NEEDS TO BE MORE CLEAR. AS OF RIGHT NOW I AM GOING TO KEEP READING THE REST OF THE CHAPTER AND HOPEFULLY COME BACK WITH MORE INSIGHT. BEST REGARDS, MAX***

## Starting a new process:

To start running a new process, the kernel must:

* Allocate and initialize the process control block.
* Allocate memory for the process
* Copy the program from disk into the newly allocated memory
* Allocate a user-level stack for user-level execution
* Allocate a kernel-level stack for handling system calls, interrupts and processor exceptions.

To start running the program, the kernel must also

* Copy arguments into user memory
  + For example: When you click on a file in Windows, the window manager asks the kernel to start the application associated with the file, passing it the file name. Then the file name is the argument (?)
* Transfer control to user mode

## Implementing Upcalls

Applications can benefit from being told when event occurs that need their immediate attention. We need to virtualize some part of the kernel so that applications can behave more like operating system. We call virtualized interrupts and exceptions **upcalls**. In windows, they are *asynchronous events.*

There are several uses for immediate event delivery with upcalls:

* **Preemptive user-level threads** 
  + An application may run multiple tasks, or threads, in a process. A user-level thread package can use a periodic timer upcall as a trigger to switch tasks, to share the processor more evenly among user-level tasks or to stop a runaway task, e.g., if a web browser needs to terminate an embedded third party script.
* **Asynchronous I/O notification**
  + A system call starts, but return immediately. Later, the application can poll the kernel for I/O-completion, or a separate notification can be sent via an upcall to the application when the I/O completes.
* **Interprocess communication**
  + A kernel upcall is needed if a process generates an event that needs the instant attention of another process. For example for logout- to notify applications that they should save file data and clearly terminate.
* **User-level exception handling**
  + Many applications can handle their own exceptions handling routines. For example that files are saved before the application shuts down. The OS therefore needs to inform the application when it receives a processor exception so the application runtime handles the event.
* **User-level resource allocation**
  + Many applications may be able to optimize their behaviour to differing amounts of CPU time or memory

## Case Study: Booting an Operating System Kernel

When a computer boots, it sets the machine’s program counter to start executing at a pre-determined position in memory. Since the computer is not yet running, the initial machine instructions must be fetched and executed immediately after the power is turned on before the system has had a chance to initialize its DRAM. Instead, systems typically use a special read-only hardware memory (Boot ROM) to store their boot instructions. On most x86 personal computers, the boot program is called the BIOS, for “Basic Input/Output System”.

The bios reads a fixed-size block of bytes called the bootloader. Once the BIOS has copied the bootloader into memory the first instruction is executed. Then, the bootloader in turn loads the kernel into memory and jumps to it.

When the kernel starts running it can initialize its data structures, such as setting up the interrupt vector table to point to the various interrupt, processor execution and system call handlers.

The kernel starts it first process, usually the login page.