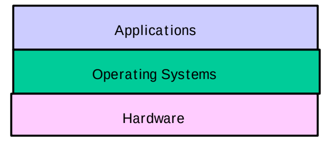
2: Architectural Underpinnings and Application Requirements

1/11/2004 8:59 PM

-1

OS Layer

- Remember OS is a layer between the underlying hardware and application demands
- OS functionality determined by both
  - Features of the hardware
  - Demands of applications



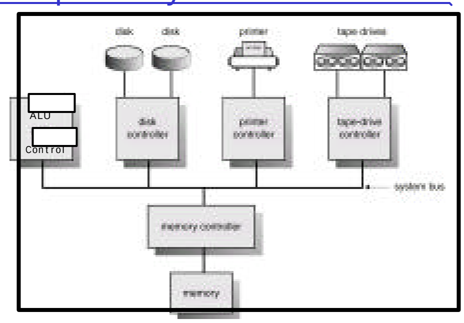
-2

Raw Materials

- What does the OS have to work to provide an efficient, fair, convenient, secure computing platform?
- Raw hardware
  - CPU architecture (instruction sets, registers, busses, caches, DMA controllers, etc.)
  - Peripherals (CD-ROMs, disk drives, network interfaces, etc.)

-3

Computer System Architecture



-4

CPU

- Registers
  - Local storage or scratch space
- Arthimetic logic unit (ALU)
  - Addition, multiplication, etc (integer and/or floating point)
  - Logical operations like testing for equality or 0
  - Operations performed by loading values into registers from memory, operating on the values in the registers, then saving register values back to memory
- Control unit
  - Cause a sequence of instructions, stored in memory to be retrieved and executed
  - Fetch instruction from memory, decode instruction, signal functional units to carry out tasks
  - PC = program counter contains memory address of instruction being processed
  - IR – instruction register – copy of the current instruction

-5

Bus and Memory

- Bus
  - Address lines, data lines, some lines for arbitration
  - Internal communication pathway between CPU, memory and device controllers
  - Sometimes one system bus; sometimes separate memory bus and I/O bus
- Memory
  - Both data and instructions must be loaded from memory into the CPU in order to be executed
  - To access memory, address placed in memory address register and command register written
  - Range of memory addresses? Size of data register? Determined by memory technology

-6

Devices

- Device controllers
  - Small processing units that connect a device to the system bus
  - Registers that can be read/written by CPU
    - command register (what to do), status register (is the device busy? Has the device completed a request?) , data register to store data bring written to the device or read from the device
- Device drivers
  - Software to hide the complexities of the device controller interface behind a higher level logical API
  - Example: read lba 10 instead vs. write command value 0x30 to command register, address 10 to address register,…

Better Raw Material?

- The "better" the underlying hardware, the better computing experience the OS can expose
- Certainly the faster the CPU, the more memory, etc. the better experience the OS can expose to applications
- Also there are some features that the hardware can provide to make the OS's job much easier
- Lets see if we can guess some…

Enforcing Protection

- If we want the operating system to be able to enforce protection and policies on all user processes, what can give the OS the power to do that?
  - Protected Instructions
  - Deny applications direct access to the hardware
  - Protected Mode of Execution (user vs kernel)
  - Memory protection hardware

Protected Instructions

- If you would look over the assembly language for a computer, you may notice that some instructions look pretty dangerous
  - Should any application be allowed to directly execute the halt instruction? Denial of service attack?
  - Should any application be allowed to directly access I/O devices? Read any ones files from disk?
- Hardware can help OS by designating some instructions as protected instructions that only the OS can issue
- How can the hardware tell whether it is OS (kernel) code or user code?

Protected Mode

- In addition to designating certain instructions as protected instructions, the hardware would need to be able to distinguish the OS from user apps
- Most architectures have a "mode" value in a protected register
  - When user applications execute, the mode value is set to one thing
  - When the OS kernel executes, the mode value set to something else
  - If code running in user mode, an attempt to execute protected instructions will generate an exception
  - Switching the mode value must of course be protected
- Some architectures support more protection modes than just user/kernel

Switching Modes

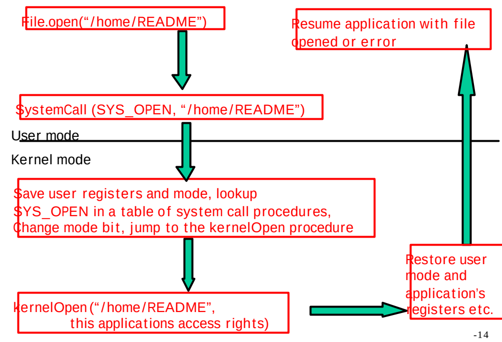
- So how do we switch between an OS running in kernel mode and an application running in user mode?
  - OS could set the mode bit to a different mode before allowing the application to run on the CPU
  - If an application needs to access a protected resource to accomplish its task (like read a file or send a message on the network), how can it do that at user mode?
- Once an application is running how can we force it to relinquish control?

System Calls

- If an application legitimately needs to access a protected feature (Ex. read a file from disk, it calls a special OS procedure called a "system call"
  - System call instruction executed with a parameter that designates specific call desired and any other parameters needed
  - The state of the user program is saved so that it can be restored (context switch to the OS)
  - Control passed to an OS procedure to accomplish the task and mode bit changed!
  - OS procedure runs at the request of the user program but can verify the user program's "rights" and refuse to perform the action if necessary
  - On completion of the system call, the state of user program including the old mode bit is restored

-13

System Call Illustrated



-14

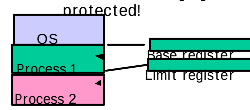
Memory Protection

- All code that executes on the CPU must be loaded into memory (its code, its data, etc.)
  - It is executed by setting the program counter register to point to the memory location of the next instruction to execute (add, jump, load, store, etc.)
- OS has its code in memory and so does each runnable user process
- Would we want a process to store random data into the OS's code or data segments? What about into another processes code or data segments?
- What prevents this?

-15

Simple Memory Protection Hardware

- Give each process a contiguous set of memory addresses to use and dedicate two registers to specifying the top and the bottom of this region
  - Of course, changing the base and limit register must be protected!



When process 1 executing, base and limit set to point to process 1's memory region if process 1 tries to load or store to addresses outside this region then hardware will transfer control to the OS

- Memory protection hardware generally more powerful that base and limit registers (page tables, TLB, etc.)

-16

Transferring Control to the OS

- A system call causes control to be transferred to the OS at the application's request
- Other things can cause control to be transferred to the OS but not at the application's request
  - Could be that the application did something wrong like tried to address memory it shouldn't or tries to divide by 0, etc.
  - Could be that a hardware device is requesting service

-17

Concrete Example: Intel CPU

- During OS initialization:
  - Interrupt Descriptor Table (IDT) loaded with handlers for each kind of interrupt
  - System call is interrupt vector 128 (0x80)
  - Kernel code segment is set to have privilege level 0 (user code runs at 3)
- Entry in IDT corresponding to vector 128 is set with:
  - Pointer to the kernel code segment and offset of the system call handler in this segment
  - Permission for code running at level 3 to invoke it
- To make system call, user level app:
  - Sets eax register to the system call number
  - Executes "int 0x80" instruction

-18

A Day in the Life of the OS

- When a machine reboots, the operating system will execute for some time to initialize the state of the machine and to start up certain system processes
- Once initialization is complete, the OS only executes when some “event” (e.g. system call, device interrupt) occurs that require its attention
- When an event occurs
  - The current state of the machine is saved
  - The mode changes to protected mode
  - An event handler procedure is executed (handlers for all possible events must be specified)

-19

Interrupts and Exceptions

- Two main types of events
- Exceptions are caused by software
  - Normal software requests for OS service are called “traps”
  - Software errors that transfer control to the OS are called “faults”
- Interrupts are caused by hardware (e.g. device notifies CPU that it has completed an I/O request)
- Warning: Understand the various types but don't worry too much about the names
  - Sometimes system calls called software interrupts
  - Sometimes say “trap to the OS” to handle a hardware interrupt

-20

Overlapping I/O and Computation

- If we want the OS to be able to efficiently keep the CPU busy, then I/O devices need to be able to operate independently
- Even if CPU can do other work while I/O is pending, system is still inefficient if CPU constantly needs to check for I/O completion (polling)
  - Interrupts
  - DMA
  - Buffering

-21

Interrupt Driven I/O

- CPU uses special instructions or writes to special memory addresses (memory mapped I/O) to initiate the I/O request
- Device will perform the request while the CPU does other work
- When the request is complete, the device will send an interrupt signal to the CPU via a shared bus
- Interrupt causes control to transfer to the OS (even if an application is in the middle of execution)
- Interrupt handler saves the context of the current process and then uses the interrupt type to index into a vector table of routines
- Control switches to the procedure registered in the table to handle the specific interrupt

-22

Interrupting interrupts?

- What happens if get another interrupt while processing one? Information about first interrupt could be lost
- Disable interrupts while processing an interrupt
- When finished processing an interrupt, check other devices with pending requests for a “done” status

-23

Intel Architecture's PIC

- Programmable Interrupt Controller (PIC) is a chip that offloads some interrupt processing from the main CPU
- Serves a referee to prioritize interrupt signals and allows devices to prevent conflicts
  - Device interrupts go to the PIC; PIC determines which device raised the interrupt; Sends interrupt to the CPU with a value indicating the interrupt service routine to invoke
  - If multiple interrupts, PIC will buffer them and send them one at a time to the CPU
- Treated by the main CPU as a peripheral

-24

Request Processing With Interrupts

- To issue a request, OS executes the "top half" initiates request processing
  - Check if device is available
  - If so write command, address and data registers
  - Stores info about the request issued
  - CPU returns to other processing; device controller gets busy working on request
- When request is done, "bottom half" completes request
  - device controller interrupts the CPU, finds interrupt handler and retrieves info stored about the request
  - CPU copies data from the device controller registers to main memory if needed
  - Sets device status to available

DMA

- Still if we want to transfer large chunks of data, CPU will still need to be very involved
  - For each small chunk of data, CPU must write a command to the command and address registers and transfer data to/from the data register
  - Very regular pattern
- DMA or Direct Memory Access automates this process and provides even greater overlap of computation and I/O
  - Tell device controller with DMA: Starting memory address and length and it will get each piece directly from memory as it needs it
  - Scatter/gather list: don't limit it to single start/length

Buffering

- Still more can be one to overlap computation and I/O
- What if I/O is slow enough and requested frequently enough, all processes may be waiting for I/O
  - I/O bound vs compute bound jobs
- For writes, copy data to a buffer and then allow process to continue while data is written from buffer to device
  - If system crashes?
- For reads, read data ahead in anticipation of demand

Memory Mapped I/O

- For each device, set aside a range of memory that will be mapped to the registers of the device
- The CPU thinks it is reading/writing memory locations (same instructions, same addressing scheme)
- Without memory mapped I/O, CPU needs a way to name each register on each device controller
  - Special instructions? Device/register addresses?
  - Required knowledge of number and type of devices at design time

Regaining the CPU

- If a user application is running on the CPU, what can the OS do to make it yield the CPU after its turn?
  - Timer (clock) operation
  - Timer generates interrupts on a regular interval to transfer control back to the OS
- What will the OS due when it regains control? Give another application a chance to run
  - Which one? Scheduling
  - How? Context Switch
  - More on this later…

Synchronization

- When we write a program, we think about adjacent instructions happening in order without interruption
- We've seen lots of things that can interrupt the execution of a process (timers, I/O request completion, etc.)
  - Most times this is ok; the state of our process is restored and the illusion is maintained
  - But sometimes it is really important that two things happen together with no interruption
  - Specifically if two processes are sharing resources
    - Example: two processes updating a shared database of account balances; one reads balance and adds $100, one reads balance and removes $100

Hardware support for Synchronization

- Need a way to guarantee that a sequence of instructions occur at once – at least with respect to other entities that are accessing the same data
- Solution 1: Disable Interrupts
  - Until re-enabled, instruction sequence will run to completion
  - Would you like to allow applications to do this?
- Solution 2: Provide Locks
  - Acquire lock, perform sequence, release lock
  - Sequence may be interrupted but interruption not visible to others because they wait to acquire the lock

-31

Building Locks

- Acquiring a shared lock is the same problem as updating a shared bank balance

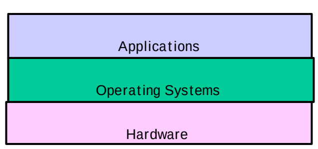
Read balance ($300)	Is lock free? (yes)
Read balance ($300)	Is lock free? (yes)
Decrement $100 ($200)	Write "I've got lock"
Increment $100 ($400)	Write "I've got lock"
Write balance ($200)	Proceed to access
Write balance ($400)	Proceed to access
Withdrawal lost!	Concurrent access violating lock!

- Hardware can provide a grouping of instructions that it will guarantee to happen atomically
  - Test and set, read/modify/write
  - From these build locks, from locks build any atomic unit

-32

OS Layer

- OS functionality determined by both
  - Features of the hardware
  - Demands of applications



-33

Programmers/users demand performance

- Users want to realize the full "advertised" capability of a hardware resource
  - If they have a disk capable of 20 MB/sec transfer rate, then they would like to be able to read files at that rate
  - If they have a network interface card capable of 100 Mbit/sec transmission rate, then they would like to be able to send data at that rate
- Operating System usually provide the desired functionality at a cost of some overhead (tax like the government)
  - Avoid seek and rotational delay when reading/writing to the disk
  - Avoid control messages sent over the network
  - Use a minimum of memory/disk space
- Programmers/users want that tax to be at a minimum

-34

Performance Optimization

- Operating systems try to optimize their algorithms to minimize the "tax" on applications
- What algorithms minimize the tax? That is a hard question – depends on what your workload is
- Example: What data do you keep in memory?
  - LRU is generally good but is exactly the wrong thing for large sequential accesses
- Optimize for the "common" case? Adapt? Let applications give hints?

-35

OS Goals

- So operating systems should:
  - Abstract the raw hardware
  - Protect apps from each other
  - Not allow applications to monopolize more that their fair share of system resources
  - Provide desired functionality
  - Expose the raw capability of the hardware, minimizing the "tax"
  - Optimize for the expected (any?) workload
  - Be simple enough that the code executes quickly and can be debugged easily
- Does this sound like a big job to anyone?

-36

Outtakes

Programmers/users demand functionality

- Operating systems provide commonly needed functionality
  - Programmers want stable storage, want to be able to share contents with other apps => file system with naming scheme shared by all processes
  - Programmers don't want to deal with paging their own code and data in and out of limited physical memory (and want protection/isolation from other processes) => virtual memory
  - Programmers want running processes to be able to communicate (not complete protection and isolation) => shared memory regions, pipes, sockets, events
  - Users don't want a single task to be able to monopolize the CPU => preemptive scheduling
  - Users want to be able to designate high and low priority processes => priority scheduling
  - ....

Application demands exceed OS functionality?

- Not all applications are happy with the operating system's services
- Many things an operating system does, application programmers could do on their own if they were sufficiently motivated
- Examples:
  - Databases traditionally ask for a raw disk partition and manage it themselves (who needs the FS?)
  - User-level thread libraries can be more efficient than kernel level threads

Application Moves Into the OS

- If a computer system is going to be used, for one application, can avoid overhead of crossing user/kernel protection boundary by putting the application in the kernel

Driving forces for OS development?

- Many times platform implies operating system; system hardware usually marketed more than OS
- Choice of OS for the PC platform is not the norm
- Even on PC platform, what drives OS development
  - Application mix, stability, politics bigger factors than OS features?
  - OS features driven by stability and ease of porting/writing apps
- All this implies OS you use every day doesn't follow the bleeding edge like hardware