Architecture Support for OS

CSCI 444/544 Operating Systems
Fall 2008

Agenda

Hardware Review
- various components
  • CPU, memory, disk
- hardware support
  • modes, memory protection, and interrupts

OS Structure
• Monolithic approach
• Layered approach
• Microkernel approach
• Kernel modules
• Virtual machine

A Simple PC

CPU

1. Fetch unit
2. Decode unit
3. Execute unit

Three-stage pipeline
Memory Hierarchy

<table>
<thead>
<tr>
<th>Typical access time</th>
<th>Typical capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 nsec</td>
<td>&lt;1 KB</td>
</tr>
<tr>
<td>2 nsec</td>
<td>1 MB</td>
</tr>
<tr>
<td>10 nsec</td>
<td>64-512 MB</td>
</tr>
<tr>
<td>100 sec</td>
<td>5-50 GB</td>
</tr>
<tr>
<td></td>
<td>20-100 GB</td>
</tr>
</tbody>
</table>

Structure of a Disk Drive

- Surface 7
- Surface 6
- Surface 5
- Surface 4
- Surface 3
- Surface 2
- Surface 1
- Surface 0

Bus

Architecture Support for OS

These features were built primarily to support OS:

- Modes of execution (OS protection)
- Memory protection
- Interrupts and exceptions
- I/O control operations
- Timer (clock) operation
Modes of Execution

OS code is stored in memory
- What if a user program modifies OS code or data?

Introduce modes of execution
- Instructions can be executed in user mode or kernel (system) mode

A special register holds which mode the CPU is in

Certain instructions can only be executed when in kernel mode

Likewise, certain memory locations can only be written when in kernel mode
- Only OS code is executed in kernel mode
- Only OS can modify its memory
- The mode register can only be modified in kernel mode

Memory Protection

OS must protect user programs from each other
- maliciousness, ineptitude

Simplest scheme: base and limit registers
- base and limit registers are loaded by OS before starting program

Paging, segmentation, virtual memory
- page tables, page table pointers
- translation looksaside buffers (TLBs)
- page fault handling

Crossing protection boundaries

So how do user programs do something privileged?
- e.g., how can you write to a disk if you can’t execute I/O instructions?

User programs must call an OS procedure
- OS defines a sequence of system calls
- how does the user-mode to kernel-mode transition happen?

There must be a system call instruction, which:
- throws a software interrupt, which vectors to a kernel handler
- passes a parameter indicating which system call to invoke
- saves caller’s state (regs, mode bit) so they can be restored
- OS must verify caller’s parameters (e.g., pointers)
- must be a way to return to user mode once done

Traps

A trap is a special instruction that forces the PC to a known address and sets the mode into kernel

Foundation of the system call

![Diagram of system call process]
OS Control Flow

After the OS has booted, all entry to the kernel happens as the result of an event:
- event immediately stops current execution
- changes mode to kernel mode, event handler is called

Kernel defines handlers for each event type:
- specific types are defined by the architecture
  - e.g.: timer event, I/O interrupt, system call trap
- when the processor receives an event of a given type, it
  - transfers control to handler within the OS
  - handler saves program state (PC, regs, etc.)
  - handler functionality is invoked
  - handler restores program state, returns to program

Interrupts and Exceptions

Two main types of events: interrupts and exceptions:
- Exceptions are caused by software executing instructions
  - e.g., the x86 `int` instruction
  - e.g., a page fault, or an attempted write to a read-only page
  - an expected exception is a "trap", unexpected is a "fault"
- Interrupts are caused by hardware devices
  - e.g., device finishes I/O
  - e.g., timer fires

I/O Control

Issues:
- how does the kernel start an I/O?
  - special I/O instructions
  - memory-mapped I/O
- how does the kernel notice an I/O has finished?
  - Polling (synchronous)
  - Interrupts (asynchronous)

Interrupts are basis for asynchronous I/O:
- device performs an operation asynchronously to CPU
- device sends an interrupt signal on bus when done
- in memory, a vector table contains list of addresses of kernel routines to handle various interrupt types
- CPU switches to address indicated by vector index specified by interrupt signal

Timers

How can the OS prevent runaway user programs from hogging the CPU (infinite loops?)
- use a hardware timer that generates a periodic interrupt
- before it transfers to a user program, the OS loads the timer with a time to interrupt
  - "quantum" – how big should it be set?
- when timer fires, an interrupt transfers control back to OS
  - at which point OS must decide which program to schedule next

Should the timer be privileged?
- for reading or for writing?
Monolithic Kernels

Traditionally, OS’s are built as a monolithic entity:
- Single linked binary
- Any function can call any other function
- All in one place with no protection between components

Layered approach

Idea: Implement OS as a set of layers
The first description of this approach was Dijkstra’s THE system (1968)
- Layer 5: Job Managers
  - Execute users’ programs
- Layer 4: Device Managers
  - Handle devices and provide buffering
- Layer 3: Console Manager
  - Implements virtual consoles
- Layer 2: Page Manager
  - Implements virtual memories for each process
- Layer 1: Kernel
  - Implements a virtual processor for each process
- Layer 0: Hardware
Each layer can be tested and verified independently

Monolithic Design

Many modern OSes fall into this category: Unix, Windows XP
Major advantage:
- Good performance, easy for kernel developers, well-understood, high-level of protection between applications
Disadvantages:
- No isolation between kernel components
- Not (safely and easily) extensible
- As system scales, it becomes
  - Hard to modify
  - Hard to maintain
What is the alternative?
- Find a way to organize the OS in order to simplify its design and implementation

Problems with Layering

Strict hierarchical structure is too inflexible
- Real systems have “uses” cycles
  - File system requires virtual memory services (buffers)
  - Virtual memory would like to use files for its backing store
- Poor performance
  - Each layer crossing has overhead associated with it
Microkernels

Design Philosophy: protected kernel code provides minimal “small, clean, logical” set of abstractions
• processor control,
• virtual memory
• communication protection

Organize the rest of OS as user-level processes
– e.g., file system “server”

Processes communicate using message-passing
– Like a distributed system

Examples: Mach, Chorus, QNX

Microkernels: Pros vs Cons

Advantages
• Simplicity
  – Core kernel is very small
• Extensibility
  – Can add new functionality in user-mode code
• Reliability
  – OS services confined to user-mode programs

Disadvantages
• Poor performance
  – Message transfer operations instead of system call

State of the Art: Kernel Modules

Basic idea: users can supply modules, which run directly in the kernel’s address space

Advantages:
• Good performance
• Extensibility

Disadvantages:
• Modules can compromise security, reliability
  – Device drivers cause 85% of crashes in Windows 2000!

Solaries Loadable Modules

• Similar to a layered system, but more flexible
• Like the microkernel approach, but more efficient
Virtual Machine
Abstract the hardware of a single computer into several different execution environments

System Boot
- Small piece of code — **bootstrap loader**, locates the kernel, loads it into memory, and starts it
- Quite often two-step process where a simple bootstrap loader at fixed location loads **boot block**, which in turn loads the kernel
- When power initialized on system, execution starts at a fixed memory location
  - Firmware used to hold initial boot code