Lecture - II

OS Structures

Tevfik Koşar

University at Buffalo
August 29th, 2013

Roadmap

• OS Design and Implementation
  - Different Design Approaches
• Major OS Components
  - Processes
  - Memory management
  - CPU Scheduling
  - I/O Management

OS Design Approaches

Operating System Design and Implementation

• Start by defining goals and specifications
• Affected by choice of hardware, type of system
  - Batch, time shared, single user, multi user, distributed
• User goals and System goals
  - User goals - operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals - operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
• No unique solution for defining the requirements of an OS
  → Large variety of solutions
  → Large variety of OS

Operating System Design and Implementation (Cont.)

• Important principle: to separate policies and mechanisms
  
  **Policy:** What will be done?
  
  **Mechanism:** How to do something?

  • Eg. to ensure CPU protection
    - Use Timer construct (mechanism)
    - How long to set the timer (policy)

  • The separation of policy from mechanism allows maximum flexibility if policy decisions are to be changed later

System Calls

• System calls are the only entry points into the kernel and system

  - User programs
  - Library functions & programs
  - shell, getcher, ls, read, more...
  - fork, open, read, system calls: rm, chmod, kill

• Programming interface to the services provided by the OS
• All programs needing resources must use system calls
• Most UNIX commands are actually library functions and utility programs (e.g., shell interpreter) built on top of the system calls
Example

- C program invoking printf() library call, which calls write() system call

```
#include <stdio.h>

int main ()
{
    ...
    printf ("Greetings\n");
    ...
    return 0;
}
```

Dual-Mode Operation

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
    - Mode bit provided by hardware
      - Provides ability to distinguish when system is running user code or kernel code
      - Protects OS from errant users, and errant users from each other
      - Some instructions designated as *privileged*, only executable in kernel mode
      - System call changes mode to kernel, return from call resets it to user

Transition from User to Kernel Mode

- How to prevent user program getting stuck in an infinite loop / process hogging resources
  - **Timer**: Set interrupt after specific period (1ms to 1sec)
    - Operating system decrements counter
    - When counter zero generate an interrupt
    - Set up before scheduling process to regain control or terminate program that exceeds allotted time

Questions

- At the system boot time, what should be the mode of operation?
- When to switch to user mode?
- When to switch to kernel mode?
- Which of these are mechanisms?
- Which of these are policies?

OS Design Approaches

- Simple Structure
- Layered Approach
- Microkernels
- Modules

Simple Structure

- No well defined structure
- Start as small, simple, limited systems, and then grow
- No well defined layers, not divided into modules
Simple Structure

- **Example: MS-DOS**
- Initially written to provide the most functionality in the least space
- Started small and grew beyond its original scope
- Levels not well separated: programs could access I/O devices directly
- Excuse: the hardware of that time was limited (no dual user/kernel mode)

**MS-DOS pseudolayer structure**

Layered Approach

- **Monolithic operating systems**
  - No one had experience in building truly large software systems
  - Problems caused by mutual dependence and interaction were grossly underestimated
  - Such lack of structure became unsustainable as OS grew
  - Early UNIX, Linux, Windows systems → monolithic, partially layered
- **Enter hierarchical layers and information abstraction**
  - Each layer is implemented exclusively using operations provided by lower layers
  - It does not need to know how they are implemented
  - Hence, lower layers hide the existence of certain data structures, private operations and hardware from upper layers

Simple Layered Approach

- The original UNIX
  - Enormous amount of functionality crammed into the kernel - everything below system call interface
  - "The Big Mess": a collection of procedures that can call any of the other procedures whenever they need to
  - No encapsulation, total visibility across the system
  - Very minimal layering made of thick, monolithic layers

**UNIX system structure**

Layered Approach

- Layers can be debugged and replaced independently without bothering the other layers above and below
  - Famous example of strictly layered architecture: the TCP/IP networking stack

**Theoretical model of operating system design hierarchy**

<table>
<thead>
<tr>
<th>Level</th>
<th>Name</th>
<th>Origin</th>
<th>Example Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Shell</td>
<td>User programming</td>
<td>Interfaces to shell language</td>
</tr>
<tr>
<td>1</td>
<td>Users</td>
<td>User processes</td>
<td>Quit, kill, suspend, restore</td>
</tr>
<tr>
<td></td>
<td>Devices</td>
<td>Device processes</td>
<td>Create, delete, read, write, ioctl</td>
</tr>
<tr>
<td></td>
<td>Devices</td>
<td>External devices</td>
<td>Open, close, read, write, ioctl</td>
</tr>
<tr>
<td>1</td>
<td>File</td>
<td>Files</td>
<td>Create, delete, open, close, read, write</td>
</tr>
<tr>
<td>1</td>
<td>Commands</td>
<td>Commands</td>
<td>Create, delete, open, close, read, write</td>
</tr>
<tr>
<td>2</td>
<td>Virtual memory</td>
<td>Segments, pages</td>
<td>Read, write, lock</td>
</tr>
<tr>
<td>3</td>
<td>Local file systems</td>
<td>Blocks of data, device classes</td>
<td>Read, write, lock, block</td>
</tr>
<tr>
<td>4</td>
<td>File systems</td>
<td>File systems</td>
<td>Read, write, lock, block</td>
</tr>
</tbody>
</table>

Full Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers.
  - The bottom layer (layer 0), is the hardware;
  - The highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
- THE system (by Dijkstra), MULTICS, GLUnix, VAX/VMS
Layered Approach

- Major difficulty with layering
  - ... appropriately defining the various layers!
  - layering is only possible if all function dependencies can be sorted out into a Directed Acyclic Graph (DAG)
  - however there might be conflicts in the form of circular dependencies ("cycles")

Circular dependency on top of a DAG

Layered Approach

- Circular dependencies in an O/S organization
  - example: disk driver routines vs. CPU scheduler routines
    - the device driver for the backing store (disk space used by virtual memory) may need to wait for I/O, thus invoke the CPU-scheduling layer
    - the CPU scheduler may need the backing store driver for swapping in and out parts of the table of active processes

- Other difficulty: efficiency
  - the more layers, the more indirections from function to function and the bigger the overhead in function calls
  - backlash against strict layering: return to fewer layers with more functionality

Microkernel System Structure

- The microkernel approach
  - a microkernel is a reduced operating system core that contains only essential O/S functions
  - the idea is to minimize the kernel by moving up as much functionality as possible from the kernel into user space
  - many services traditionally included in the O/S are now external subsystems running as user processes
    - device drivers
    - file systems
    - virtual memory manager
    - windowing system
    - security services, etc.

Layered OS vs Microkernel

- The modular approach
  - many modern operating systems implement kernel modules (Modern UNIX, Solaris, Linux, Windows, Mac OS X)
  - this is similar to the object-oriented approach:
    - each core component is separate
    - each talks to the others over known interfaces
    - each is loadable as needed within the kernel
  - overall, modules are similar to layers but with more flexibility (any model could call any other module)
  - modules are also similar to the microkernel approach, except they are inside the kernel and don’t need message passing

**Examples:** QNX, Tru64 UNIX, Mach (CMU), Windows NT
Modular Approach

- Modules are used in Solaris, Linux and Mac OS X

Hybrid Systems

- Many real OS use combination of different approaches
- Linux: monolithic & modular
- Windows: monolithic & microkernel & modular
- Mac OS X: microkernel & modular

Mac OS X Structure - Hybrid

- BSD: provides support for command line interface, networking, file system, POSIX API and threads
- Mach: memory management, RPC, IPC, message passing

Major OS Components

- Processes
- Memory management
- CPU Scheduling
- I/O Management

Processes

- A process is the activity of executing a program
  - Pasta for six
    - boil 1 quart salty water
    - stir in the pasta
    - cook on medium until "al dente"
    - serve
  - CPU
  - Input data
  - Program
  - Process
  - Thread of execution

Processes

- **Multitasking** gives the illusion of parallel processing (independent virtual program counters) on one CPU

(a) Multitasking from the CPU’s viewpoint

(b) Multitasking from the processes’ viewpoint = 4 virtual program counters

Pseudoparallelism in multitasking

Processes

- **Operating System Responsibilities:**
- The O/S is responsible for managing processes
  - the O/S creates & deletes processes
  - the O/S suspends & resumes processes
  - the O/S schedules processes
  - the O/S provides mechanisms for process synchronization
  - the O/S provides mechanisms for interprocess communication
  - the O/S provides mechanisms for deadlock handling

CPU Scheduling

- **Operating System Responsibilities:**
- The O/S is responsible for efficiently using the CPU and providing the user with short response times
  - decides which available processes in memory are to be executed by the processor
  - decides what process is executed when and for how long, also reacting to external events such as I/O interrupts
  - relies on a scheduling algorithm that attempts to optimize CPU utilization, throughput, latency, and/or response time, depending on the system requirements
I/O Management

Device-independent software

Layers of the I/O subsystem

Two I/O Methods

Synchronous
- After I/O starts, control returns to user program only upon I/O completion.
  - Wait instruction idles the CPU until the next interrupt.
  - Wait loop (contention for memory access).
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing.

Asynchronous
- After I/O starts, control returns to user program without waiting for I/O completion.
  - Device-status table contains entry for each I/O device indicating its type, address, and state.
  - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.

Operating System Responsibilities:
The O/S is responsible for controlling access to all the I/O devices:
- hides the peculiarities of specific hardware devices from the user
- issues the low-level commands to the devices, catches interrupts and handles errors
- relies on software modules called "device drivers"
- provides a device-independent API to the user programs, which includes buffering

Memory Management

Main memory
- large array of words or bytes, each with its own address
- repository of quickly accessible data shared by the CPU and I/O devices
- volatile storage that loses its contents in case of system failure

Performance of Various Levels of Storage

- Movement between levels of storage hierarchy can be explicit or implicit
Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- If cache is smaller than storage being cached
  - Cache management - important design problem
  - Cache size and replacement policy

Migration of Integer A from Disk to Register

- Multitasking environments must be careful to use most recent value, not matter where it is stored in the storage hierarchy
- Multiprocessor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
  - Several copies of a datum can exist

Memory Management

- Operating System Responsibilities:
  The O/S is responsible for an efficient and orderly control of storage allocation
  - ensures process isolation: it keeps track of which parts of memory are currently being used and by whom
  - allocates and deallocates memory space as needed: it decides which processes to load or swap out
  - regulates how different processes and users can sometimes share the same portions of memory
  - transfers data between main memory and disk and ensures long-term storage

Summary

- OS Design Approaches
  - Mechanism vs Policy
  - Monolithic Systems,
  - Layered Approach, Microkernels, Modules
- Major OS Components
  - Processes
  - CPU Scheduling
  - I/O Management
  - Memory management
- Reading Assignment: Chapter 3 from Silberschatz

Questions?

Acknowledgements

- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from UNR