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
• 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

• 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

```c
#include <stdio.h>
int main ()
{
    printf("Greetings");
    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)

Layered Approach

- **Monolithic operating systems**
  - no one had experience in building truly large software systems
  - the problems caused by mutual dependence and interaction were grossly underestimated
  - such lack of structure became unsustainable as O/S 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

---

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

- 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.

Layered Approach

<table>
<thead>
<tr>
<th>Level</th>
<th>Name</th>
<th>Objects</th>
<th>Example Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Shell</td>
<td>Use programming environment</td>
<td>Statements in shell language</td>
</tr>
<tr>
<td>12</td>
<td>User processes</td>
<td>User processes</td>
<td>Quit, kill, suspend, resume</td>
</tr>
<tr>
<td>11</td>
<td>Directories</td>
<td>Directories</td>
<td>Create, destroy, attach, detach, search, list</td>
</tr>
<tr>
<td>10</td>
<td>Devices</td>
<td>External devices, such as printers, displays, and keyboards</td>
<td>Open, close, read, write</td>
</tr>
<tr>
<td>9</td>
<td>File system</td>
<td>Files</td>
<td>Create, destroy, open, close, read, write</td>
</tr>
<tr>
<td>8</td>
<td>Communications</td>
<td>Pipes</td>
<td>Create, destroy, open, close, read, write</td>
</tr>
<tr>
<td>7</td>
<td>Virtual memory</td>
<td>Segments, pages</td>
<td>Read, write, fetch</td>
</tr>
<tr>
<td>6</td>
<td>Local secondary store</td>
<td>Blocks of data, device channels</td>
<td>Read, write, allocate, free</td>
</tr>
<tr>
<td>5</td>
<td>Primitive processes</td>
<td>Primitive processes, semaphores, ready list</td>
<td>Suspend, resume, wait, signal</td>
</tr>
<tr>
<td>4</td>
<td>Interrupts</td>
<td>Interrupt-handling programs</td>
<td>Invoke, mask, unmask, retry</td>
</tr>
<tr>
<td>3</td>
<td>Procedures</td>
<td>Procedures, call stack, display</td>
<td>Mark stack, call, return</td>
</tr>
<tr>
<td>2</td>
<td>Instruction set</td>
<td>Evaluation stack, microprogram interpreter, stack and array data</td>
<td>Load, store, add, subtract, branch</td>
</tr>
<tr>
<td>1</td>
<td>Electronic circuits</td>
<td>Registers, gates, buses, etc.</td>
<td>Clear, transfer, activate, complement</td>
</tr>
</tbody>
</table>
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

(a) Layered kernel

(b) Microkernel
Microkernel System Structure

- **Benefits of the microkernel approach**
  - **extensibility** — it is easier to extend a microkernel-based O/S as new services are added in user space, not in the kernel
  - **portability** — it is easier to port to a new CPU, as changes are needed only in the microkernel, not in the other services
  - **reliability & security** — much less code is running in kernel mode; failures in user-space services don’t affect kernel space

- **Detriments of the microkernel approach**
  - again, performance overhead due to communication from user space to kernel space
  - not always realistic: some functions (I/O) must remain in kernel space, forcing a separation between “policy” and “mechanism”

- Examples: QNX, Tru64 UNIX, Mach (CMU), Windows NT

Modular Approach

- **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
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

  Program

  thread of execution

  CPU

  input data

  Process
Processes

- It can be interrupted to let the CPU execute a higher-priority process

  Pasta for six
  - boil 1 quart salty water
  - stir in the pasta
  - cook on medium until “al dente”
  - serve

  First aid
  - Get the first aid kit
  - Check pulse
  - Clean wound with alcohol
  - Apply band aid

  CPU (changes hat to “doctor”)

  Program

  thread of execution

  hmm... now where was I?

  CPU (back to “chef”)

  Process

  input data

  . . . and then resumed exactly where the CPU left off
Processes

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

(a) Multitasking from the CPU’s viewpoint

| job 1 | job 2 | job 3 | job 1 | job 2 | job 4 | job 3 | job 1 |

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

**Pseudoparallelism in multitasking**

Processes

- **Timesharing** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive computing**
  - Response time should be < 1 second
  - Each user has at least one program loaded in memory and executing \(\Rightarrow\) **process**
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

Two I/O Methods

- After I/O starts, control returns to user program only upon I/O completion ➔ synchronous
  - 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.

- After I/O starts, control returns to user program without waiting for I/O completion ➔ asynchronous
  - System call - request to the operating system to allow user to wait 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.
Two I/O Methods

Synchronous

Asynchronous

I/O Management

- **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

The storage hierarchy

Performance of Various Levels of Storage

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

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>registers</td>
<td>cache</td>
<td>main memory</td>
<td>disk storage</td>
</tr>
<tr>
<td>Typical size</td>
<td>&lt; 1 KB</td>
<td>&gt; 16 MB</td>
<td>&gt; 16 GB</td>
<td>&gt; 100 GB</td>
</tr>
<tr>
<td>Implementation technology</td>
<td>custom memory with multiple ports, CMOS</td>
<td>on-chip or off-chip CMOS-SRAM</td>
<td>CMOS DRAM</td>
<td>magnetic disk</td>
</tr>
<tr>
<td>Access time (ns)</td>
<td>0.25 – 0.5</td>
<td>0.5 – 25</td>
<td>80 – 250</td>
<td>5,000,000</td>
</tr>
<tr>
<td>Bandwidth (MB/sec)</td>
<td>20,000 – 100,000</td>
<td>5000 – 10,000</td>
<td>1000 – 5000</td>
<td>20 – 150</td>
</tr>
<tr>
<td>Managed by</td>
<td>compiler</td>
<td>hardware</td>
<td>operating system</td>
<td>operating system</td>
</tr>
<tr>
<td>Backed by</td>
<td>cache</td>
<td>main memory</td>
<td>disk</td>
<td>CD or tape</td>
</tr>
</tbody>
</table>
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.
Acknowledgements

• “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne

• “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings

• “Modern Operating Systems” book and supplementary material by A. Tanenbaum

• R. Doursat and M. Yuksel from UNR