Chapter 2: Operating-System Structures

肖 卿 俊
办公室：九龙湖校区计算机楼212室
电邮：csqjxiao@seu.edu.cn
主页：https://csqjxiao.github.io/PersonalPage
电话：025-52091022
Chapter 2: Operating-System Structures (7th Edition)

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines
- Operating System Generation
- System Boot
Chapter 3: Operating-System Structures (6th Edition)

- System Components
- Operating System Services
- System Calls
- System Programs
- System Structure
- Virtual Machines
- System Design and Implementation
- System Generation
Question about OS Services

- Name (as many as you can) system services you expect from an operating system
Question about OS Services

Name (as many as you can) system services you expect from an operating system:
- Process scheduling (or job scheduling)
- Inter-process communication (IPC)
- Memory management
  - Protection, sharing, demand paging
- File system for organizing external storage
- Access to I/O devices, e.g., microphones, speaker
- Access to the networks
Common System Components (and Types of System Calls)

- Process Management
- Main Memory Management
- File Management
- I/O System Management
- Secondary-Storage Management
- Networking
- Protection System
- Command-Interpreter System
Process Management

- A *process* is a program in execution.

- The operating system is responsible for the following activities in connection with process management.
  - Process creation and deletion.
  - Process suspension and resumption.
  - Provision of mechanisms for:
    - Process synchronization
    - Process communication
    - Deadlock handling
Main-Memory Management

- Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices.

- Main memory is a volatile storage device. It loses its contents in the case of system failure.
Main-Memory Management (Cont.)

- The operating system is responsible for the following activities in connections with memory management:
  - Keep track of which parts of memory are currently being used and by whom.
  - Decide which processes to load when memory space becomes available.
  - Allocate and deallocate memory space as needed.
File Management

- There are different types of physical media to store information. Each of them has its own characteristics and physical organization.
- Operating System provides a uniform logical view of information storage, i.e., file.
- A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.
The operating system is responsible for the following activities in connection with file management:

- File creation and deletion.
- Directory creation and deletion.
- Support of primitives for manipulating files and directories.
- Mapping files onto secondary storage.
- File backup on stable (nonvolatile) storage media.
Secondary-Storage Management

- Since main memory (primary storage) is volatile and too small to accommodate all data and programs permanently, the computer system must provide secondary storage to back up main memory.

- Most modern computer systems use disks as the principle on-line storage medium, for both programs and data.
Secondary-Storage Management

- The operating system is responsible for the following activities in connection with disk management:
  - Free space management
  - Storage allocation
  - Disk scheduling
The I/O subsystem consists of:
- A buffer-caching system
- A general device-driver interface
- Drivers for specific hardware devices
A *distributed* system is a collection processors that do not share memory or a clock. Each processor has its own local memory.

The processors in the system are connected through a communication network.

Communication takes place using a *protocol*. 
Networking (Distributed Systems)

- A distributed system provides user access to various system resources.

- Access to a shared resource allows:
  - Computation speed-up
  - Increased data availability
  - Enhanced reliability
Protection System

- Protection refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.

- The protection mechanism must:
  - distinguish between authorized and unauthorized usage.
  - specify the controls to be imposed and means for enforcement.
Many commands are given to the operating system by control statements which deal with:

- process creation and management
- I/O handling
- secondary-storage management
- main-memory management
- file-system access
- protection
- networking
The program that reads and interprets control statements is called variously:

- command-line interpreter
- shell (in UNIX)

Its function is to get and execute the next command statement.
Common System Components

- Process Management
- Main Memory Management
- File Management
- I/O System Management
- Secondary-Storage Management
- Networking
- Protection System
- Command-Interpreter System
Operating System Services

- Program execution – system capability to load a program into memory and to run it.
- I/O operations – since user programs cannot execute I/O operations directly, the operating system must provide some means to perform I/O.
- File-system manipulation – program capability to read, write, create, and delete files.
- Communications – exchange of information between processes
- Error detection – ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user programs.
Additional Operating System Functions

Additional functions exist not for helping the user, but rather for ensuring efficient system operations.

- Resource allocation – allocating resources to multiple users or multiple jobs running at the same time.
- Accounting – keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics.
- Protection – ensuring that all access to system resources is controlled.
System Calls

- System calls provide the interface between a running program and the operating system.
  - Generally available as assembly-language instructions.
  - Languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C, C++)

https://en.wikipedia.org/wiki/System_call#Typical_implementations
http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html
Operating System as Design

User

System

Platform support, Device Drivers

Portable OS Library

System Call Interface

Portable OS Kernel

Application / Service

Compilers

Word Processing

Web Browsers

Email

Web Servers

Databases

Software

Hardware

x86

PowerPC

ARM

PCI

Ethernet (10/100/1000)

802.11 a/b/g/n

SCSI

IDE

Graphics
Example of System Calls (1/2)

- **cp** is the command line tool in Linux, which makes a copy of your files or directories.
  - For instance, let's say you have a file named **picture.jpg** in your **working directory**, and you want to make a copy of it called **picture-02.jpg**. You would run the command:

  ```
  cp picture.jpg picture-02.jpg
  ```

- System call sequence to copy the contents of one file to another file
Example of System Calls (2/2)

- System call sequence to copy the contents of one file to another file

Example System Call Sequence
- Acquire input file name
- Write prompt to screen
- Accept input
- Acquire output file name
- Write prompt to screen
- Accept input
- Open the input file
  - if file doesn't exist, abort
- Create output file
  - if file exists, abort
- Loop
  - Read from input file
  - Write to output file
  - Until read fails
- Close output file
- Write completion message to screen
- Terminate normally
Why Execute a System Call from a Trap (Interrupt)?

- Protection is achieved via dual mode (user mode vs. kernel mode) and system call.
  - A system call is executed from a trap, via a trap-handler, and ended by a return-from-trap.

http://www.linux.it/~rubini/docs/ksys/ksys.html
System Call Implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers

- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
The caller needs know nothing about how the system call is implemented

- Just needs to obey API and understand what OS will do as a result call

- Most details of OS interface hidden from programmer by API
  - Managed by run-time support library (set of functions built into libraries included with compiler)
API – System Call – OS Relationship

user application

open ()

user mode

system call interface

kernel mode

open ()

Implementation of open ()

system call

return
Standard C Library Example

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

```c
#include <stdio.h>
int main ()
{
    
    printf ("Greetings");
    
    return 0;
}
```
Use of A System Call to Perform I/O

1. trap to monitor
2. perform I/O
3. return to user

case n

system call n

user program

resident monitor

read

...
### System Calls for the Linux 2.2 Kernel

- On the left are the numbers of the system calls. This number will be put in register `%eax`.
- On the right are the types of values to be put into the remaining registers before calling the trap 'int 0x80'.
- After each syscall, an integer is returned in `%eax`.

<table>
<thead>
<tr>
<th>%eax</th>
<th>Name</th>
<th>Source</th>
<th>%ebx</th>
<th>%ecx</th>
<th>%edx</th>
<th>%esx</th>
<th>%edi</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sys_exit</td>
<td>kernel/exit.c</td>
<td>int</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>sys_fork</td>
<td>arch/i386/kernel/process.c</td>
<td>struct</td>
<td>pt_regs</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>sys_read</td>
<td>fs/read_write.c</td>
<td>unsigned</td>
<td>int</td>
<td>char *</td>
<td>size_t</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>sys_write</td>
<td>fs/read_write.c</td>
<td>unsigned</td>
<td>int</td>
<td>const char *</td>
<td>size_t</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>sys_open</td>
<td>fs/open.c</td>
<td>const char *</td>
<td>int</td>
<td>int</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>sys_close</td>
<td>fs/open.c</td>
<td>unsigned</td>
<td>int</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>sys_waitpid</td>
<td>kernel/exit.c</td>
<td>pid_t</td>
<td>unsigned int</td>
<td>int</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call

- Three general methods used to pass parameters to the OS
System Call Parameter Passing (Cont.)

- Simplest: pass the parameters in *registers*
  - In some cases, may be more parameters than registers
- Parameters stored in a *block*, or table, in memory, and address of block passed as a parameter in a register
  - This approach taken by Linux and Solaris
- Parameters placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system

Block and stack methods do not limit the number or length of parameters being passed
Parameter Passing via Table

- **X**: parameters for call
- **load address X**
- **system call 13**
- **register**
- **use parameters from table X**
- **code for system call 13**

User program

Operating system
What is OS Structure?

- The way the OS software is organized with respect to the applications that it serves and the underlying hardware that it manages.
  - **Monolithic kernel** (单内核, 宏内核, 巨内核)
  - **Microkernel system structure** (微内核)
  - **Hybrid kernel, and Monolithic kernel with modules**
Goal of OS Structure Design

- Protection: within and across users + the OS itself
- Performance: time taken to perform the services
- Flexibility: Extensibility => Not one size fits all
- Scalability: performance ↑ if hardware resources ↑
- Agility: adapting to changes in application needs and/or resource availability
- Responsiveness: reaching to the external events
DOS-like Structure

- MS-DOS – written to provide the most functionality in the least space
  - Performance: Access to system services is like a procedure call
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
  - Bad Protection: an error of application can corrupt the OS
  - Not divided into modules
MS-DOS execution (Single Program)

(a) At system startup          (b) running a program
DOS-like Structure (cont.)

No protection between the applications and the OS, which are in the same address space.

Managed by the OS.
Monolithic Structure

Each App in its own hardware address space

OS in its own hardware address space

Managed by the OS

App 1   App 2   ......   App n

OS
Services and Device Drivers

Hardware

Managed by the OS
Monolithic Architecture
Example: UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
## UNIX System Structure

### System-call interface to the kernel
- signals terminal handling
- character I/O system
- terminal drivers
- file system
- swapping block I/O
- system
- disk and tape drivers
- CPU scheduling
- page replacement
- demand paging
- virtual memory

### Kernel interface to the hardware
- terminal controllers
- terminals
- device controllers
- disks and tapes
- memory controllers
- physical memory

<table>
<thead>
<tr>
<th>(the users)</th>
</tr>
</thead>
<tbody>
<tr>
<td>shells and commands</td>
</tr>
<tr>
<td>compilers and interpreters</td>
</tr>
<tr>
<td>system libraries</td>
</tr>
</tbody>
</table>
Design Philosophy: 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.
Layered Operating System

- Layer N
  - User interface

- Layer 1

- Layer 0
  - Hardware
Gain and Loss of Monolithic Structure

- **Loss of Protection in DOS-like Structure**
  - Unacceptable for a general-purpose OS

- **Monolithic Structure**
  - Reduce performance loss by consolidation

- **But …**
  - Monolithic structure => no customization

- **Need for customization**
  - Applications of video game and computing prime numbers may have different needs for CPU scheduling, file access or memory management
Microkernel OS Structure

Communication takes place between user modules using IPC-based message passing

App 1  App 2  ......  App n

OS Services

File System  Memory Management  CPU Scheduler

Each App in its own hardware address space
Each service in its own address space
Simple abstraction
- Address space
- IPC

Hardware

Microkernel
Advantage of Microkernel-based Design

- App 1
- App 2
- ……
- App n

OS Services
- File System 1
- Memory Management
- CPU Scheduler
- File System 2

Microkernel

Hardware

Extensibility
Pros and Cons of Microkernel System Structure

■ Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure

■ Detriments:
  - Performance overhead of user space to kernel space communication
Why Performance Loss

■ Border Crossing
  ◆ Change in locality, e.g., memory address space
  ◆ Copy data between user and system spaces

Microkernel

App 1

OS Services

File System 1

Microkernel

Monolithic

App 1

OS Services
Question

Based on discussion thus far ....

<table>
<thead>
<tr>
<th>Feature</th>
<th>DOS-like OS</th>
<th>Monolithic OS</th>
<th>Microkernel OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extensibility</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>Protection</td>
<td></td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Performance</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>
Hybrid Kernel Approach

The idea behind a hybrid kernel is to have a kernel structure similar to that of a microkernel, but to implement that structure in the manner of a monolithic kernel.

In contrast to a microkernel, all (or nearly all) operating system services in a hybrid kernel are still in kernel space.
Hybrid Kernel Example: Windows NT Client-Server Structure

- Workstation service
- Server service
- Security

- Integral subsystems
- Environment subsystems

- Win32 Application
- POSIX Application
- OS/2 Application

User mode

Executive Services

- I/O Manager
- Security Reference Monitor
- IPC Manager
- Virtual Memory Manager (VMM)
- Process Manager
- PnP Manager
- Power Manager
- Window Manager
- GDI

Object Manager

Executive

Kernel mode drivers

Microkernel

Hardware Abstraction Layer (HAL)

Kernel mode

Hardware
Hybrid Kernel Example: Mac OS X Structure

![Diagram of Mac OS X structure]

- Application environments and common services
- BSD
- Mach
Hybrid Kernel Example: OS/2 Layer Structure

- application
- application
- application

- application-programming interface
- API extension
- subsystem
- subsystem
- subsystem

- system kernel
  - memory management
  - task dispatching
  - device management

- device driver
- device driver
- device driver
- device driver

https://en.wikipedia.org/wiki/OS/2
Monolithic Kernel with Dynamically Loadable Modules

- Most modern operating systems implement kernel modules
  - Uses 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, similar to layers but with more flexible
Solaris Modular Approach

◆ Solaris is a Unix operating system originally developed by Sun Microsystems.
◆ Kernel type is Monolithic with dynamically loadable modules

https://en.wikipedia.org/wiki/Solaris_(operating_system)
Linux is also a modular monolithic kernel.
Question about Virtualization

What comes to your mind when you hear about the word “virtualization”

- Memory Systems
- Data Centers
- Java Virtual Machine
- Virtual Box
- IBM VM/370
- Google Glass
- Cloud Computing
- Dalvik JVM
- VMWare Workstation
- The movie “Inception”

https://en.wikipedia.org/wiki/Virtualization
https://en.wikipedia.org/wiki/VirtualBox
Concept of Virtualization

Virtualization refers to the act of creating a virtual (rather than actual) version of something, including:
- hardware platform virtualization,
- memory virtualization,
- CPU virtualization,
- storage virtualization,
- network virtualization, etc.
Virtual Machines

- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory.

- A virtual machine system provides an interface identical to the underlying bare hardware, creating an illusion of multiple (virtual) machines.

- User Drives:

  - Kim
  - APP1
  - Windows

  - Piero
  - APP2
  - Linux

  - Bala
  - APP3
  - Windows

  Shared Hardware Resources
A hypervisor or virtual machine monitor (VMM) is a piece of computer software, firmware or hardware that creates and runs virtual machines.
A hosted hypervisor takes the layered approach to its logical conclusion. It treats underlying hardware and the host operating system kernel as though they were all hardware.
Virtual Machine Implementation 2: Hosted Hypervisor

This is how it works:

- Your main OS runs like usual (Windows in LiLi's case). This OS is called "Host OS".
- A virtualization software (e.g., VirtualBox or VMWare) will launch a second OS on top of the first one.
- The virtualization software will trick the second OS and give him some virtual hardware.

Each OS, no matter virtual or host, is not aware of the other's existence.
Virtual Machine Implementation 2: Hosted Hypervisor

- The resources of the physical computer are shared to create the virtual machines
  - CPU scheduling can create the appearance that users have their own processor
  - Spooling and a file system can provide virtual disk, virtual card readers, and virtual line printers
  - A normal user time-sharing terminal serves as the virtual machine operator’s console
VMware Architecture

<table>
<thead>
<tr>
<th>application</th>
<th>application</th>
<th>application</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td>guest operating system (free BSD)</td>
<td>guest operating system (Windows NT)</td>
<td>guest operating system (Windows XP)</td>
<td></td>
</tr>
<tr>
<td>virtual CPU</td>
<td>virtual CPU</td>
<td>virtual CPU</td>
<td></td>
</tr>
<tr>
<td>virtual memory</td>
<td>virtual memory</td>
<td>virtual memory</td>
<td></td>
</tr>
<tr>
<td>virtual devices</td>
<td>virtual devices</td>
<td>virtual devices</td>
<td></td>
</tr>
</tbody>
</table>

virtualization layer

host operating system (Linux)

CPU

memory

I/O devices
Virtual Machines Advantages

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.

- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
Virtual Machines Disadvantage

- The virtual machine concept is difficult to implement due to the effort required to provide an exact duplicate to the underlying machine.

- Other lightweight virtualization mechanisms are available.

https://en.wikipedia.org/wiki/Virtualization
Java Virtual Machine

- Compiled Java programs are platform-neutral bytecodes executed by a Java Virtual Machine (JVM).
- JVM consists of
  - class loader
  - class verifier
  - runtime interpreter
- Just-In-Time (JIT) compilers increase performance
Java Virtual Machine

java .class files
↓
class loader
↓
verifier
↓
java interpreter

host system
Java Virtual Machine (Cont.)

Java Application

byte code

Java VM (interpreter)

Dynamic compilation

JIT Compiler

native code

Win32  OS/2  Linux x86  AIX  OS/400  Linux PPC  OS/390  Linux 390  Win64  Linux IA-64

IA-32  POWER/PowerPC  S/390  IA-64
MS-DOS Layer Structure

application program

resident system program

MS-DOS device drivers

ROM BIOS device drivers
FreeBSD Running
Multiple Programs

- kernel
- process B
- interpreter
- free memory
- process C
- process D
Communication Models

Communication may take place using either message passing or shared memory.
Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.
- SYSGEN program obtains information concerning the specific configuration of the hardware system.
Operating System Generation (cont.)

- **Booting** – starting a computer by loading the kernel
- **Bootstrap program** – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution
System Boot

- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
  - When power is initialized on system, execution starts at a fixed memory location
    - **Firmware** is used to hold the initial boot code
Design and Implementation of OS not “solvable”, but some approaches have proven successful

Internal structure of different operating systems can vary widely

Start by defining goals and specifications
Operating System Design and Implementation (cont.)

- Affected by choice of hardware, type of system

- *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
Important principle to separate

**Policy:** What will be done?
**Mechanism:** How to do it?

Mechanisms determine how to do something, policies decide what will be done

The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later