Chapter 2
Operating System Overview
Seventh Edition
By William Stallings
Learning Objectives

• The key functions of an operating system

• Discuss the evolution of operating system for early simple (Batch systems to modern complex systems)
A program that acts as an intermediary between a user of a computer and the computer hardware.

• **Convenience**
  • Makes the computer more convenient to use

• **Efficiency**
  • Allows computer system resources to be used in an efficient manner

• **Ability to evolve**
  • Permit effective development, testing, and introduction of new system functions without interfering with service
What Operating Systems Do

- Operating system is a resource allocator:
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
  - Not much visible to users how memory is allocated between different users?

- Operating system is a program controller:
  - Controls execution of programs to prevent errors and improper use of the computer
Operating System as Software

- Functions in the same way as ordinary computer software
- It is a program that executed by the processor, but with extra privileges
- OS relinquishes control of the processor to execute other programs and must depend on the processor to allow it to regain control
1) One set of OS services provides functions that are helpful to the user:

- **User interface**: almost all OS have a user interface (UI): varies between Command-Line, Graphics User Interface (GUI), Batch

- **Program development**: editors and debuggers

- **Program execution**: The system must be able to load a program into memory and to run that program.

- **Access to I/O devices** (I/O operation)

- **Controlled access to files**

- **System access**

- **Error detection and handling**
2) Another set of OS functions exists for ensuring the efficient operation of system itself via resource sharing

- **Resource allocation**: when multiple users running concurrently, resources must be allocated to each of the.
- **Accounting**: To keep track of which user use how much and what kinds of computer resources.
- **Protection and security**: The owner of information should be able to control use of that information
Operating System as Resource Manager

Figure 2.2 The Operating System as Resource Manager
**CLI:** Command Line Interface (CLI) or command interpreter (shell)
- Sometimes implemented in kernel
- Sometimes by systems program for example in unix it is a shell
- fetches a command from user and executes it
  - Command may be **built-in**, 
  - Command may be **another program**

**GUI:** User-friendly desktop interface
- Icons represent files, programs, actions, etc.

*Many operating systems now include both CLI and GUI interfaces*

Linux: command shells available (CLI); KDE as GUI
The MacOS X GUI
System Calls

- Interface between OS and user programs (to perform privileged operations)

- **Programming interface** to the services provided by the OS
  - i.e. interface provided to applications

- Are called by a running program to get services

- Typically written in a high-level language (C or C++)

- Machine dependent, but can be invoked by standard procedure libraries

- Even a simple program may make a lot of calls per second.
What are system calls used for?

Anything to do with:

- Accessing devices
  – Accessing files
  – Requesting memory
  – Setting/changing access permissions
- Communicating with other processes
- Stopping/starting processes
- Open a file
- Get data from the network
- Kill a process
Types of System Calls

- File management
- Device management
- Communications
- Protection
- Security
- Information maintenance
- Process control
- ...

## Linux system calls

Around 300-400 system calls

<table>
<thead>
<tr>
<th>Number</th>
<th>Generic Name of System Call</th>
<th>Name of Function in Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>exit</td>
<td>sys_exit</td>
</tr>
<tr>
<td>2</td>
<td>fork</td>
<td>sys_fork</td>
</tr>
<tr>
<td>3</td>
<td>read</td>
<td>sys_read</td>
</tr>
<tr>
<td>4</td>
<td>write</td>
<td>sys_write</td>
</tr>
<tr>
<td>5</td>
<td>open</td>
<td>sys_open</td>
</tr>
<tr>
<td>6</td>
<td>close</td>
<td>sys_close</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>39</td>
<td>mkdir</td>
<td>sys_mkdir</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Accessing and Executing System Calls

- System calls typically **not accessed directly** by programs
- Mostly accessed by programs via a **high-level Application Program Interface (API)** (i.e. a library) rather than direct system call use.
- APIs are wrapper functions for the system calls

- Three most common APIs are:
  - Win32 API for Windows,
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X),
  - Java API for the Java Virtual Machine (JVM)
API Support for System Calls

Your Program

User level code

API

System Calls

Kernel level code

... fd = open(...);

Your Program Code

open(...)
{...}

Standard C library Code

fopen(...)
{...}

sys_open(...)
{...}

Kernel Code
Why use APIs rather than system calls directly?

✓ **Platform independent:** System calls differ from platform to platform.

✓ **Upgrade:** The operating system may provide newer versions of a system call with enhanced features.

✓ **More functionality:** The API usually provides more useful functionality than the system call directly.

✓ **More flexibility:** The API can support multiple versions of the operating system and detect which version it needs to use at runtime.
Examples of Windows and Unix System Calls

<table>
<thead>
<tr>
<th>Process Control</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CreateProcess()</td>
<td>fork()</td>
</tr>
<tr>
<td></td>
<td>ExitProcess()</td>
<td>exit()</td>
</tr>
<tr>
<td></td>
<td>WaitForSingleObject()</td>
<td>wait()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File Manipulation</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CreateFile()</td>
<td>open()</td>
</tr>
<tr>
<td></td>
<td>ReadFile()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>WriteFile()</td>
<td>write()</td>
</tr>
<tr>
<td></td>
<td>CloseHandle()</td>
<td>close()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Device Manipulation</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SetConsoleMode()</td>
<td>ioctl()</td>
</tr>
<tr>
<td></td>
<td>ReadConsole()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>WriteConsole()</td>
<td>write()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Information Maintenance</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GetCurrentProcessID()</td>
<td>getpid()</td>
</tr>
<tr>
<td></td>
<td>SetTimer()</td>
<td>alarm()</td>
</tr>
<tr>
<td></td>
<td>Sleep()</td>
<td>sleep()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Communication</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CreatePipe()</td>
<td>pipe()</td>
</tr>
<tr>
<td></td>
<td>CreateFileMapping()</td>
<td>shmget()</td>
</tr>
<tr>
<td></td>
<td>MapViewOfFile()</td>
<td>mmap()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protection</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SetFileSecurity()</td>
<td>chmod()</td>
</tr>
<tr>
<td></td>
<td>InitializeSecurityDescriptor()</td>
<td>umask()</td>
</tr>
<tr>
<td></td>
<td>SetSecurityDescriptorGroup()</td>
<td>chown()</td>
</tr>
</tbody>
</table>
A major OS will evolve over time for a number of reasons:

- Hardware upgrades
- New types of hardware
- New services
- Fixes

Take advantage of hardware upgrades and new types/functions of hardware e.g. paging support or multicore.
Generations include:

- Serial Processing
- Simple Batch Systems
- Multiprogrammed Batch Systems
- Time Sharing Systems
Serial Processing

Earliest Computers:

- **No operating system**
  - programmers interacted directly with the computer hardware

- Computers run from a console with display lights (error messages), toggle switches, some form of input device (punch card, tape), and a printer (for output)

- Users have access to the computer in “series”

Problems:

- **Scheduling time:**
  - most installations used a hardcopy sign-up sheet to reserve computer time
    - time allocations could run short or long, resulting in wasted computer time

- **Setup time (very time consuming)**
  - a considerable amount of time was spent just on setting up the program to run
Simple Batch Systems

- Early computers were very expensive
  - important to maximize processor utilization
  - The central idea behind the simple batch-processing scheme: **Monitor**

- **Monitor**
  - user no longer has direct access to processor
  - job is submitted to computer operator who batches them together and places them on an input device
  - program branches back to the monitor when finished
Monitor Point of View

- **Resident Monitor** is software always in memory and available for execution
- Monitor: scheduling, privileged operations
- Users submit jobs to operator
- Operator batches jobs together
- Places them on an input device
- Software that controls the running programs
- Monitor reads in job and gives control
- Monitor controls the sequence of events
- Program branches back to monitor when finished (Job returns control to monitor)
Processor Point of View

- Processor executes instruction from the memory containing the monitor

- Executes the instructions in the user program until it encounters an ending or error condition

- “control is passed to a job” means processor is fetching and executing instructions in a user program

- “control is returned to the monitor” means that the processor is fetching and executing instructions from the monitor program
Job Control Language (JCL)

Special type of programming language used to provide instructions to the monitor (compiler, data)

- what compiler to use
- what data to use

$JOB
$FTN
...
[FORTRAN program]
...
$LOAD
$RUN
...
[data]
...
$END
Desirable Hardware Features

Memory protection for monitor
• while the user program is executing, it must not alter the memory area containing the monitor

Timer
• prevents a job from monopolizing the system

Privileged instructions
• can only be executed by the monitor

Interrupts
• gives OS more flexibility in controlling user programs
Simple Batch Systems: Modes of Operation

User Mode
- user program executes in user mode
- certain areas of memory are protected from user access
- certain instructions may not be executed

Kernel Mode
- monitor executes in kernel mode
- privileged instructions may be executed
- protected areas of memory may be accessed
Simple Batch System Overhead

- Processor time alternates between execution of user programs and execution of the monitor

- Sacrifices:
  - some main memory is now given over to the monitor
  - some processor time is consumed by the monitor

- Despite overhead, the simple batch system improves utilization of the computer
Uniprogramming

In simple batch OS the processor is often idle!

- The processor spends a certain amount of time executing, until it reaches an I/O instruction; it must then wait until that I/O instruction concludes before proceeding
- There must be enough memory to hold the OS (resident monitor) and one user program
- When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O
Multiprogramming

- Multiprogramming
  - also known as multitasking
  - memory is expanded to hold three, four, or more programs and switch among all of them
Multiprogramming

- Processor has more than one program to execute
- The sequence in which the programs are executed depends on their relative priority and whether they are waiting for I/O
- After an interrupt handler completes, control may not return to the program that was executing at the time of the interrupt
Multiprogramming Example

For a simple batch environment, these jobs will be executed in sequence: Tree jobs comp complete after 30 minutes.

For Multiprogramming: these jobs will be executed concurrently

<table>
<thead>
<tr>
<th>Sample Program Execution Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type of job</strong></td>
</tr>
<tr>
<td>Heavy compute</td>
</tr>
<tr>
<td>Duration</td>
</tr>
<tr>
<td>Memory required</td>
</tr>
<tr>
<td>Need disk?</td>
</tr>
<tr>
<td>Need terminal?</td>
</tr>
<tr>
<td>Need printer?</td>
</tr>
</tbody>
</table>
Device Utilization Histograms

(a) Uniprogramming

(b) Multiprogramming
## Effect Of Multiprogramming on Resource Utilization

<table>
<thead>
<tr>
<th></th>
<th>Uniprogramming</th>
<th>Multiprogramming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor use</td>
<td>20%</td>
<td>40%</td>
</tr>
<tr>
<td>Memory use</td>
<td>33%</td>
<td>67%</td>
</tr>
<tr>
<td>Disk use</td>
<td>33%</td>
<td>67%</td>
</tr>
<tr>
<td>Printer use</td>
<td>33%</td>
<td>67%</td>
</tr>
<tr>
<td>Elapsed time</td>
<td>30 min</td>
<td>15 min</td>
</tr>
<tr>
<td>Throughput</td>
<td>6 jobs/hr</td>
<td>12 jobs/hr</td>
</tr>
<tr>
<td>Mean response time</td>
<td>18 min</td>
<td>10 min</td>
</tr>
</tbody>
</table>
Time-Sharing Systems

- For many applications the interaction with users is essential.
- Processor time is shared among multiple users
- Using multiprogramming to handle multiple interactive jobs
- Multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation
Batch Multiprogramming vs. Time Sharing

Both batch processing and time sharing use **multiprogramming**. The key differences are:

<table>
<thead>
<tr>
<th>Principal objective</th>
<th>Batch Multiprogramming</th>
<th>Time Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source of directives to operating system</td>
<td>Maximize processor use</td>
<td>Minimize response time</td>
</tr>
<tr>
<td></td>
<td>Job control language commands provided with the job</td>
<td>Commands entered at the terminal</td>
</tr>
</tbody>
</table>

Table 2.3  Batch Multiprogramming versus Time Sharing
Memory Management

- The OS has five principal storage management responsibilities:
  - Process isolation
  - Automatic allocation and management
  - Support of modular programming
  - Protection and access control
  - Long-term storage
Virtual Memory

- A facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available

- Conceived to meet the requirement of having multiple user jobs reside in main memory concurrently
Paging

- Allows processes to be comprised of a number of fixed-size blocks, called pages

- Program references a word by means of a virtual address
  - consists of a page number and an offset within the page
  - each page may be located anywhere in main memory

- Provides for a dynamic mapping between the virtual address used in the program and a real (or physical) address in main memory
Virtual Memory

Main Memory
Main memory consists of a number of fixed-length frames, each equal to the size of a page. For a program to execute, some or all of its pages must be in main memory.

Disk
Secondary memory (disk) can hold many fixed-length pages. A user program consists of some number of pages. Pages for all programs plus the operating system are on disk, as are files.

Figure 2.9  Virtual Memory Concepts
Virtual Memory Addressing

Figure 2.10 Virtual Memory Addressing
The nature of the threat that concerns an organization will vary greatly depending on the circumstances.

The problem involves controlling access to computer systems and the information stored in them.

Main issues:
- Availability
- Confidentiality
- Data integrity
- Authenticity
Scheduling and Resource Management

- Key responsibility of an OS is managing resources

- Resource allocation policies must consider:
  - fairness
  - efficiency
  - differential responsiveness
Demands on operating systems require new ways of organizing the OS.

Different approaches and design elements have been tried:

- Microkernel architecture
- Multithreading
- Symmetric multiprocessing
- Distributed operating systems
- Object-oriented design
Microkernel Architecture

- Assigns only a few essential functions to the kernel:
  - address spaces
  - interprocess communication (IPC)
  - basic scheduling

- The approach:
  - simplifies implementation
  - provides flexibility
  - is well suited to a distributed environment
Multithreading

- Technique in which a process, executing an application, is divided into threads that can run concurrently

**Thread**
- dispatchable unit of work
- includes a processor context and its own data area to enable subroutine branching
- executes sequentially and is interruptible

**Process**
- a collection of one or more threads and associated system resources
- programmer has greater control over the modularity of the application and the timing of application related events
Symmetric Multiprocessing (SMP)

- Term that refers to a computer hardware architecture and also to the OS behavior that exploits that architecture

- Several processes can run in parallel

- Multiple processors are transparent to the user
  - these processors share same main memory and I/O facilities
  - all processors can perform the same functions

- The OS takes care of scheduling of threads or processes on individual processors and of synchronization among processors
SMP Advantages

**Availability**
- failure of a single process does not halt the system

**Performance**
- more than one process can be running simultaneously, each on a different processor

**Incremental Growth**
- performance of a system can be enhanced by adding an additional processor

**Scaling**
- vendors can offer a range of products based on the number of processors configured in the system
Multiprogramming

Multi programming

Multi processing

Figure 2.12 Multiprogramming and Multiprocessing
OS Design

Distributed Operating System

- Provides the illusion of
  - a single main memory space
  - single secondary memory space
  - unified access facilities
- State of the art for distributed operating systems lags that of uniprocessor and SMP operating systems

Object-Oriented Design

- Used for adding modular extensions to a small kernel
- Enables programmers to customize an operating system without disrupting system integrity
- Eases the development of distributed tools and full-blown distributed operating systems
Virtual Machines and Virtualization

- Virtualization
  - enables a single PC or server to simultaneously run multiple operating systems or multiple sessions of a single OS
  - a machine can host numerous applications, including those that run on different operating systems, on a single platform
  - host operating system can support a number of virtual machines (VM)
    - each has the characteristics of a particular OS and, in some versions of virtualization, the characteristics of a particular hardware platform
Virtual Memory Concept

Figure 2.13 Virtual Memory Concept
Virtual Machine Architecture

Process perspective:
• the machine on which it executes consists of the virtual memory space assigned to the process
• the processor registers it may use
• the user-level machine instructions it may execute
• OS system calls it may invoke for I/O
• ABI defines the machine as seen by a process

Application perspective:
• machine characteristics are specified by high-level language capabilities and OS system library calls
• API defines the machine for an application

OS perspective:
• processes share a file system and other I/O resources
• system allocates real memory and I/O resources to the processes
• ISA provides the interface between the system and machine
Process and System Virtual Machines

Figure 2.14 Process and System Virtual Machines
Process and System Virtual Machines

(b) System VM

Figure 2.14 Process and System Virtual Machines
A multiprocessor OS must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors.

Key design issues:

- **Simultaneous concurrent processes or threads**:
  - Kernel routines need to be reentrant to allow several processors to execute the same kernel code simultaneously.

- **Scheduling**:
  - Any processor may perform scheduling, which complicates the task of enforcing a scheduling policy.

- **Synchronization**:
  - With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide.

- **Memory management**:
  - The reuse of physical pages is the biggest problem of concern.

- **Reliability and fault tolerance**:
  - The OS should provide graceful degradation in the face of processor failure.
Multicore OS Considerations

- The design challenge for a many-core multicore system is to efficiently harness the multicore processing power and intelligently manage the substantial on-chip resources efficiently.

- Potential for parallelism exists at three levels:
  - Hardware parallelism within each core processor, known as instruction parallelism.
  - Potential for multiprogramming and multithreaded execution within each processor.
  - Potential for a single application to execute in concurrent processes or threads across multiple cores.
Grand Central Dispatch

- Developer must decide what pieces can or should be executed simultaneously or in parallel

Grand Central Dispatch (GCD)
- implemented in Mac Os X 10.6
- helps a developer once something has been identified that can be split off into a separate task
- thread pool mechanism
- allows anonymous functions as a way of specifying tasks
Virtual Machine Approach

- Allows one or more cores to be dedicated to a particular process and then leave the processor alone to devote its efforts to that process.

- Multicore OS could then act as a hypervisor that makes a high-level decision to allocate cores to applications but does little in the way of resource allocation beyond that.
Summary

- Operating system objectives and functions:
  - convenience, efficiency, ability to evolve
  - user/computer interface
  - resource manager

- Evolution:
  - serial processing, simple batch systems, multiprogrammed batch systems, time sharing systems

- Process
- Memory management
  - real address, virtual address
- Scheduling and resource management
- Multithreading
A Brief History of Operating Systems
Evolution of Operating Systems
**First Generation: 1945-1955 (1/3)**

- No operating system
- Vacuum tubes and plugboards

ENIAC (Electronic Numerical Integrator And Computer): the first electronic general-purpose computer.

Human was the operator and programmer.

Computer were programmed by physically re-wiring it; later, through stored programs (von Neumann architecture).

Programs written in machine or assembly language.
Problems:

1 - Serial processing: users had access to the computer one by one in series.

2 - Users have to write again and again the same routines.
A programmer would first write the program on paper (in FORTRAN or assembly), then punch it on cards.
Second Generation: 1955-1965 (2/3)

Separation between operators and programmers.

- The programmer: prepares her/his job off-line.
- The operator: runs the job and delivers a printed output.
Problems:

- A lot of CPU time is still wasted waiting for I/O instructions to complete.

- I/O devices much slower than processor.
Third Generation: 1965-1980 (1/2)

Integrated Circuits (ICs)
- **Multiprogrammed** batch systems.

- Load two jobs in memory: while one job is waiting for I/O, the processor could switch to the other job.

- Expand to three, four or more jobs.

- Jobs are kept in main memory at the same time and the CPU is multiplexed among them or multiprogrammed.
Fourth Generation: 1980-Present (1/3)

Personal Computers (PCs)
- From multiple users back to a single user.

- Multitasking a central feature of modern PC operating systems.

- PC systems emphasize user convenience.
Fourth Generation: 1980-Present (3/3)

- GNU (GNU's Not Unix!): 1983
- Mac OS: 1984
- Microsoft Windows: 1985
- Linux: 1991
Summary

- Operating system objectives and functions:
  - convenience, efficiency, ability to evolve
  - user/computer interface
  - resource manager

- Evolution:
  - serial processing, simple batch systems, multiprogrammed batch systems, time sharing systems

- Process
- Memory management
  - real address, virtual address
- Scheduling and resource management
- Multithreading
- Symmetric multiprocessing (SMP)
  - distributed OS
  - object oriented design