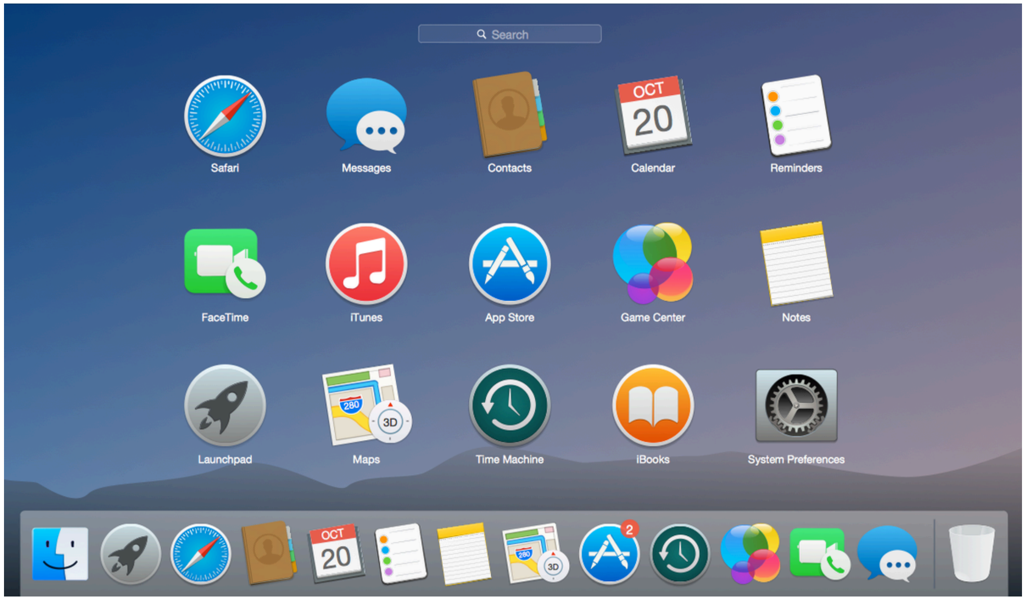
Understanding OS Structures

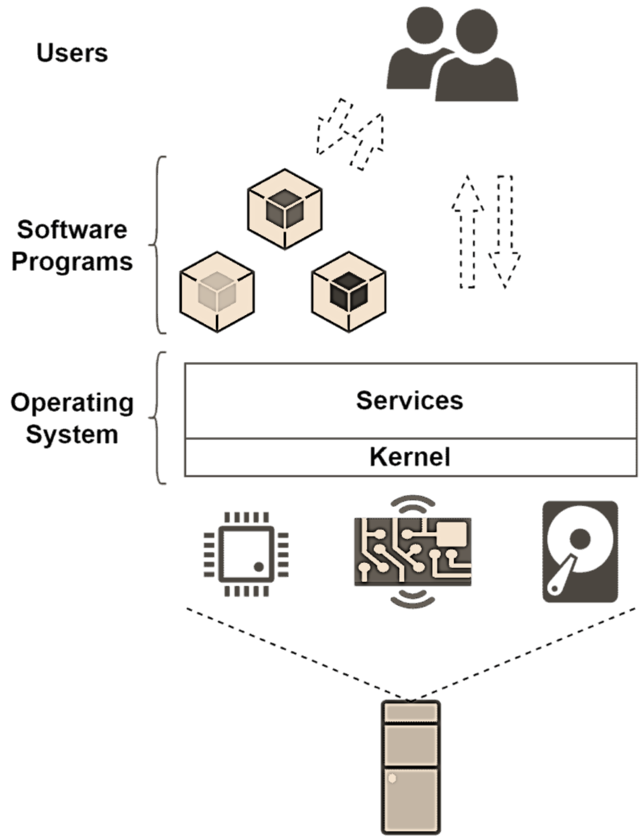
User Interface (UI)

- Can be Command-Line (CLI), Graphics User Interface (GUI), or Batch
- Allows user interaction with system services via system calls (typically written in C/C++)



System Services for Users

- •Program execution
- •I/O operations
- •File-system manipulation
- •Communications
- •Error detection



Services for Efficient OS Operation

- Resource allocation
- Accounting
- Protection and security

System Calls and APIs

- Accessed via APIs such as Win32, POSIX, Java
- Each system call has an associated number
- System call interface maintains a table indexed by these numbers

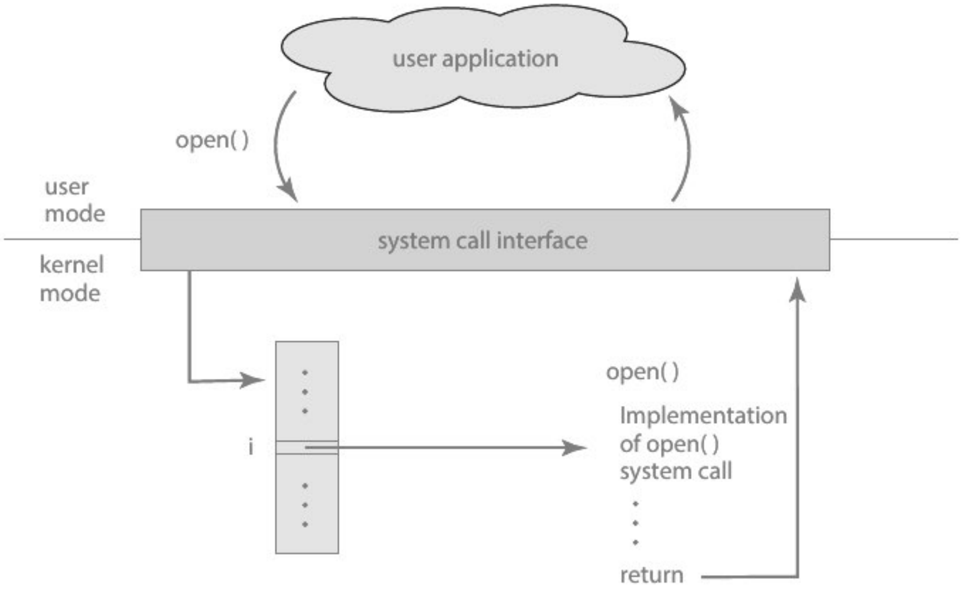


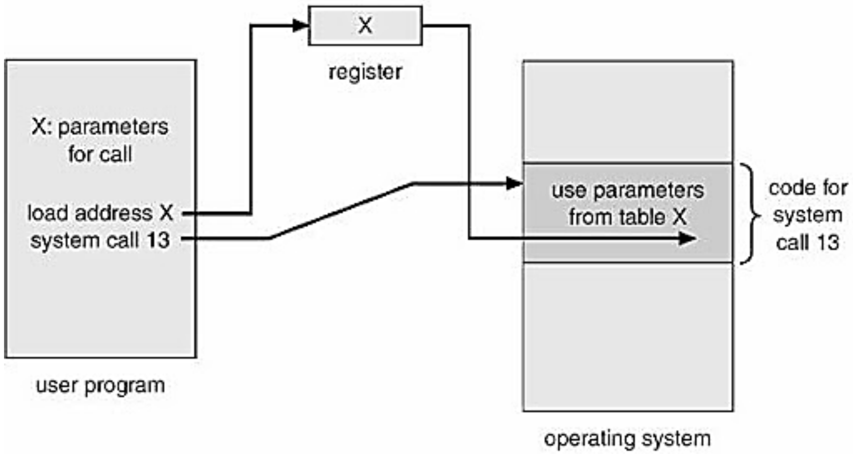
Figure 2.6 The handling of a user application invoking the `open()` system call.

Passing Parameters in System Calls

•Methods:

- Passing in registers
- Address of parameter stored in a block
- Pushed onto the stack by the program, popped off by the OS

•Block and stack methods allow for flexibility in the number and length of parameters

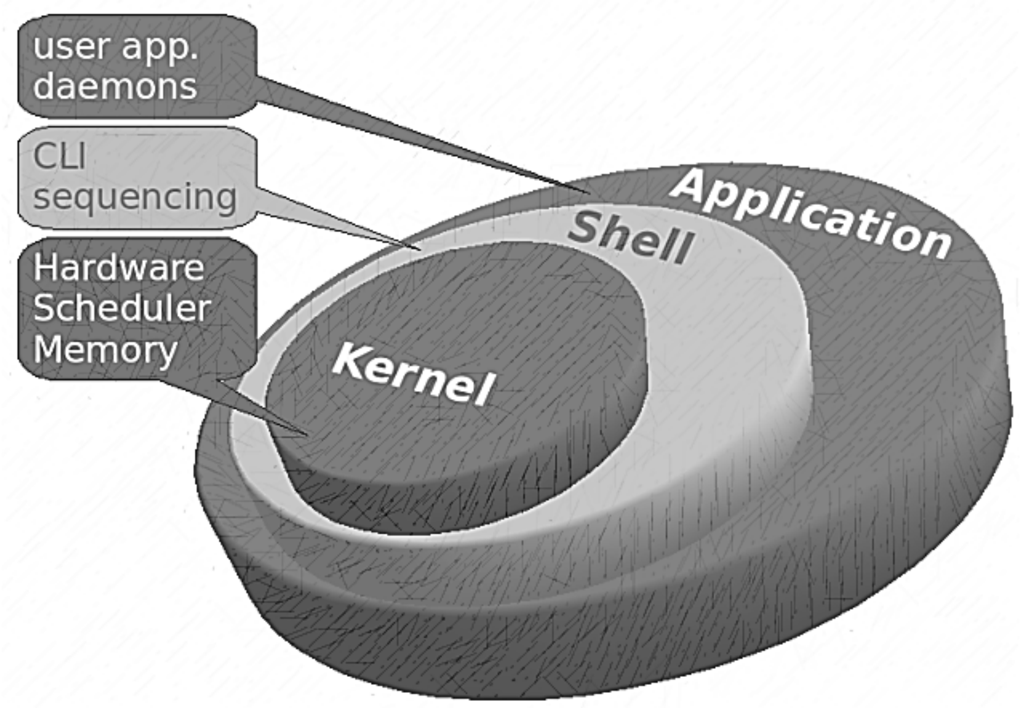


Types of System Calls

- **Process control:** end, abort, load, execute, create/terminate process, wait, allocate/free memory
- **File management:** create/delete file, open/close file, read, write, get/set attributes
- **Device management:** request/release device, read, write, logically attach/detach devices
- **Information maintenance:** get/set time, get/set system data, get/set process/file/device attributes
- **Communications:** create/delete communication connection, send/receive, transfer status information

Understanding Operating System Components

- Operating systems facilitate user interaction with computer hardware and resource management.
- Two fundamental components: Shell and Kernel.



Shell

Definition: User interface for interacting with the OS.

Functionality: Accepts user commands, translates them for the kernel.

Features: Command history, tab completion, scripting.

Shell Commands in C++ (Windows vs. Unix)

Function	Windows Command	Unix Command
List Directory Contents	system("dir")	system("ls -l")
Create Directory	system("mkdir new_dir")	system("mkdir new_dir")
Remove Directory	system("rmdir new_dir")	system("rmdir new_dir")
Copy File	system("copy source.txt destination.txt")	system("cp source.txt destination.txt")
Move File	system("move source.txt destination.txt")	system("mv source.txt destination.txt")
Delete File	system("del file.txt")	system("rm file.txt")
Print Working Directory	system("cd")	system("pwd")
Display File Content	system("type file.txt")	system("cat file.txt")

Kernel

Definition: Core component managing system resources.

Responsibilities: Memory management, process scheduling, device management.

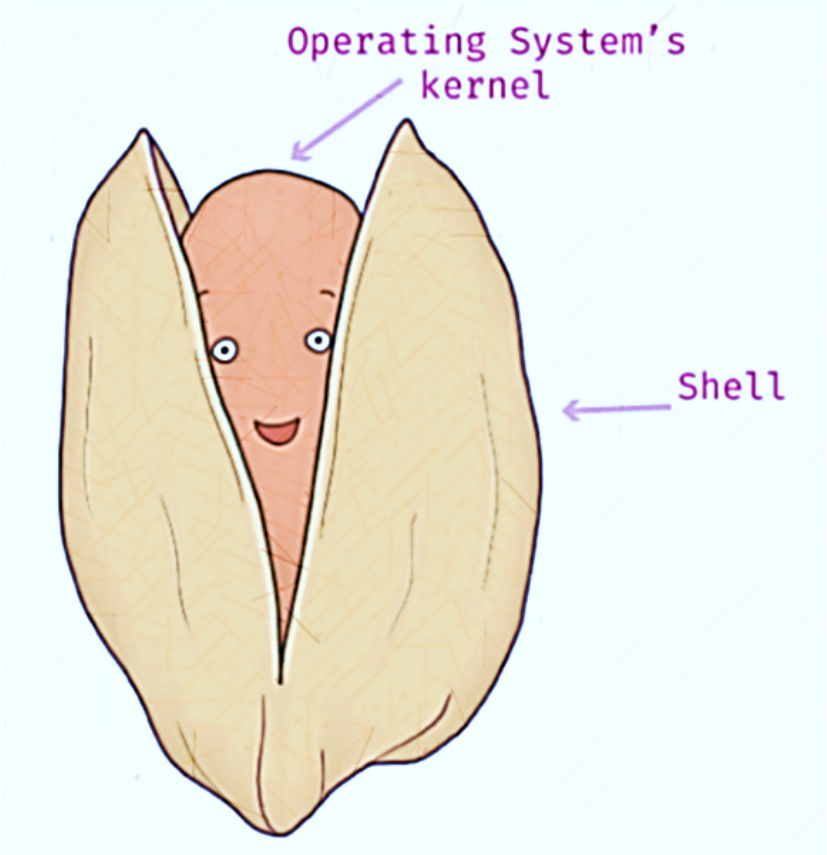
Interaction: Directly communicates with hardware.

System Calls

System Call	Description
fork()	Creates a new process by duplicating the calling process.
exec()	Replaces the current process image with a new process image.
wait()	Makes the calling process wait until one of its child processes terminates.
exit()	Terminates the calling process and returns a status to the parent process.
open()	Opens a file or device and returns a file descriptor.
read()	Reads data from a file descriptor into a buffer.
write()	Writes data from a buffer to a file descriptor.

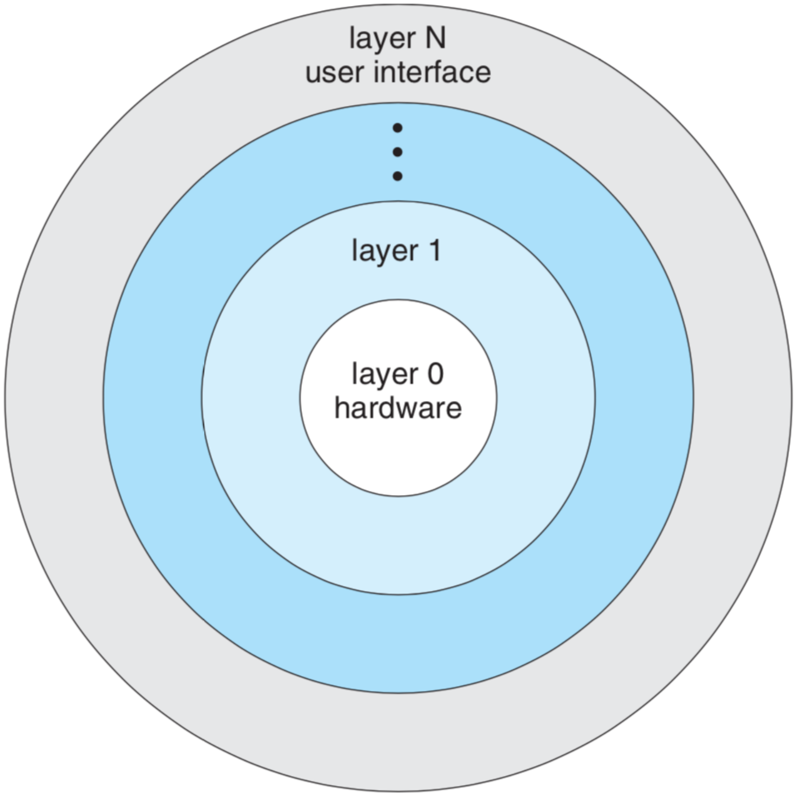
Difference between Shell and Kernel

- **Shell**: User interface, interprets user commands.
- **Kernel**: Core system component, manages hardware and resources.



OS Layered Approach

- Divided into layers (levels), with hardware at the bottom (layer 0) and the user interface at the top (layer N)
- Each layer uses functions and services of lower layers



Virtual Machines

USES A LAYERED APPROACH

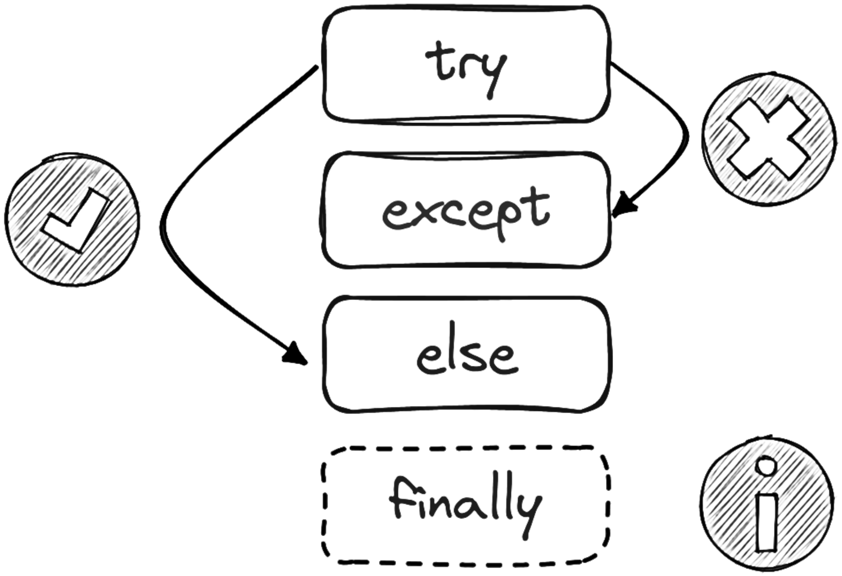
TREATS HARDWARE AND OS KERNEL AS HARDWARE

HOST CREATES THE ILLUSION OF A DEDICATED PROCESSOR AND MEMORY FOR EACH PROCESS

EACH GUEST HAS A 'VIRTUAL' COPY OF THE COMPUTER

Error Handling

- Application failures generate core dump files capturing the memory of the process
- OS failures generate crash dump files containing kernel memory



Activity

- Write a program that executes five different shell commands. For example:
    - **dir** (List contents of current directory)
    - **echo "Hello, World!"** (Print "Hello, World!")
    - **mkdir test_directory** (Create a new directory named "test_directory")
    - **type example.txt** (Display contents of a text file named "example.txt")
    - **cd** (print the current working directory)
- Experiment with different commands and observe the output.
- Comment your code and provide explanations for each command you execute.
- Share your experience and any challenges you faced.
- Discuss the importance of using shell commands in programming and real-life applications.
- Highlight any security concerns or best practices when executing shell commands from a program.

-

Sample Answer

1.Experience & Challenges:

1. Experience: Seamless interaction with the OS.
2. Challenges: Ensuring cross-platform compatibility.

2.Importance:

1. Vital for file manipulation, process management, and network operations.
2. Streamline workflows and automate tasks in real-life applications.

3.Security & Best Practices:

1. Guard against command injection attacks.
2. Validate user inputs and restrict access to sensitive resources.
3. Thoroughly test commands before deployment.

```
#include <cstdlib>
#include <iostream>

int main() {
    // 1. List contents of current directory
    std::cout << "Listing contents of current directory:\n";
    system("dir");

    // 2. Print "Hello, World!"
    std::cout << "\nPrinting 'Hello, World!':\n";
    system("echo Hello, World!");

    // 3. Create a new directory named "test_directory"
    std::cout << "\nCreating directory 'test_directory':\n";
    system("mkdir test_directory");

    // 4. Display contents of a text file named "example.txt"
    std::cout << "\nDisplaying contents of 'example.txt':\n";
    system("type example.txt");

    // 5. Change directory to "test_directory"
    std::cout << "\nChanging directory to 'test_directory':\n";
    system("cd test_directory");

    // Optional: Print current working directory
    std::cout << "\nCurrent working directory:\n";
    system("cd");

    return 0;
}
```