System Calls and Operating System Structures

Week 2

SDB

Autumn 2025

Agenda

1. What are System Calls?
2. System Call Lifecycle and Examples
3. OS Structures: Monolithic, Layered, Microkernel
4. Performance and Modularity Trade-offs
5. Summary and Q&A

Learning Goal

Understand how user programs interact with the OS and how OS structure affects performance and extensibility.

What is a System Call?

Definition

A system call is a controlled interface through which user programs request services from the operating system kernel, such as file access, process creation, or device control.

What is a System Call?

Definition

A system call is a controlled interface through which user programs request services from the operating system kernel, such as file access, process creation, or device control.

Common Categories and Examples:

- **Process Control:** `fork()`, `exec()`, `exit()`
- **File Operations:** `open()`, `read()`, `write()`
- **Device I/O:** `ioctl()`, `read()`, `write()`
- **Info Maintenance:** `getpid()`, `alarm()`, `gettimeofday()`

What is a System Call?

Definition

A system call is a controlled interface through which user programs request services from the operating system kernel, such as file access, process creation, or device control.

Common Categories and Examples:

- **Process Control:** `fork()`, `exec()`, `exit()`
- **File Operations:** `open()`, `read()`, `write()`
- **Device I/O:** `ioctl()`, `read()`, `write()`
- **Info Maintenance:** `getpid()`, `alarm()`, `gettimeofday()`

Did You Know?

Most system calls in Linux are thin wrappers around a lower-level syscall instruction that triggers a CPU trap to kernel mode.

Example: Process Creation in Linux

C code using system calls:

```c
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child Process\n");
    } else {
        printf("Parent Process\n");
    }
    return 0;
}
```

Example: Process Creation in Linux

C code using system calls:

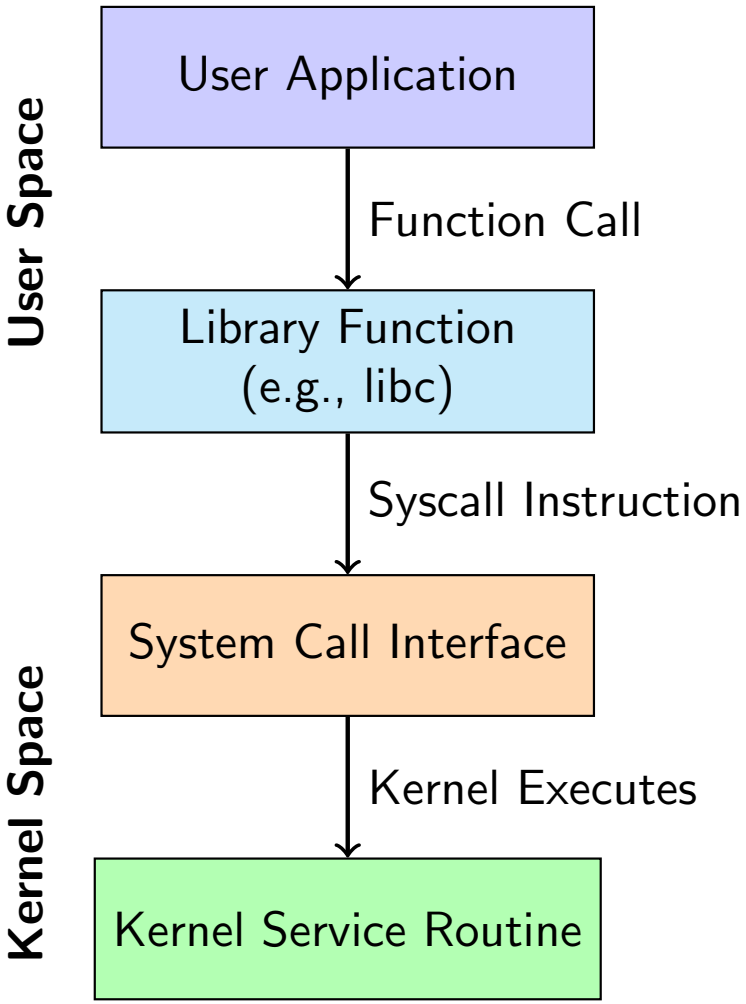
```c
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child Process\n");
    } else {
        printf("Parent Process\n");
    }
    return 0;
}
```

Behind the Scenes:

- `fork()` creates a new process by duplicating the calling process.
- Internally, Linux uses the `clone()` syscall to implement `fork()`.
- The child gets a new PID and a copy of the parent's memory space.

System Call Flow Diagram



User Application
Initiates requests for system-level services (e.g., file access, networking) via high-level APIs.

Library Function (libc)
Provides standard interfaces (e.g., `open()`, `read()`) that abstract away syscall details.

System Call Interface
Acts as a controlled gateway to the kernel. Validates requests and switches to kernel mode.

Kernel Service Routine
Executes privileged operations like I/O, memory management, and process control.

System calls enforce privilege separation, ensuring secure interaction between user and kernel space.

System Call Categories

Grouped by Functionality:

- **Process Control**
  Create, terminate, wait, fork, exec
- **File Management**
  Create, delete, read, write, open, close
- **Device Management**
  Request, release, attach, detach
- **Information Maintenance**
  Get/set time, process ID, user ID, system info
- **Communication**
  Pipes, sockets, signals, shared memory

System Call Categories

Grouped by Functionality:

- **Process Control**
  Create, terminate, wait, fork, exec
- **File Management**
  Create, delete, read, write, open, close
- **Device Management**
  Request, release, attach, detach
- **Information Maintenance**
  Get/set time, process ID, user ID, system info
- **Communication**
  Pipes, sockets, signals, shared memory

Think!

Which of these categories do you think is most performance-sensitive? Why?

System Call Categories

- **Process Control**
  Create, terminate, wait, fork, exec
  e.g., `fork()`, `execvp()`
  Note: Security-critical; affects process isolation.
- **File Management**
  Create, delete, read, write, open, close
  e.g., `open()`, `read()`
  Note: Performance-sensitive; frequent I/O operations.
- **Device Management**
  Request, release, attach, detach
  e.g., `ioctl()`, `mount()`
  Note: Security-sensitive; interacts with hardware.
- **Information Maintenance**
  Get/set time, PID, UID, system info
  e.g., `getpid()`, `gettimeofday()`
  Note: Lightweight; used for bookkeeping.
- **Communication**
  Pipes, sockets, signals, shared memory
  e.g., `pipe()`, `send()`, `kill()`
  Note: Performance-sensitive; used in IPC and networking.

System Call Categories

- **Process Control**
  Create, terminate, wait, fork, exec
  e.g., `fork()`, `execvp()`
  Note: Security-critical; affects process isolation.
- **File Management**
  Create, delete, read, write, open, close
  e.g., `open()`, `read()`
  Note: Performance-sensitive; frequent I/O operations.
- **Device Management**
  Request, release, attach, detach
  e.g., `ioctl()`, `mount()`
  Note: Security-sensitive; interacts with hardware.
- **Information Maintenance**
  Get/set time, PID, UID, system info
  e.g., `getpid()`, `gettimeofday()`
  Note: Lightweight; used for bookkeeping.
- **Communication**
  Pipes, sockets, signals, shared memory
  e.g., `pipe()`, `send()`, `kill()`
  Note: Performance-sensitive; used in IPC and networking.

Think!

Which of these categories do you think is most performance-sensitive? Why?

OS Structure Types

Operating systems can be architected in different ways, each with trade-offs in performance, modularity, and reliability.

Three Classical Architectures:

- **Monolithic:** All OS services run in kernel space as one large binary.
- **Layered:** OS is divided into layers, each built on top of the lower one.
- **Microkernel:** Only essential services in kernel; others run in user space.

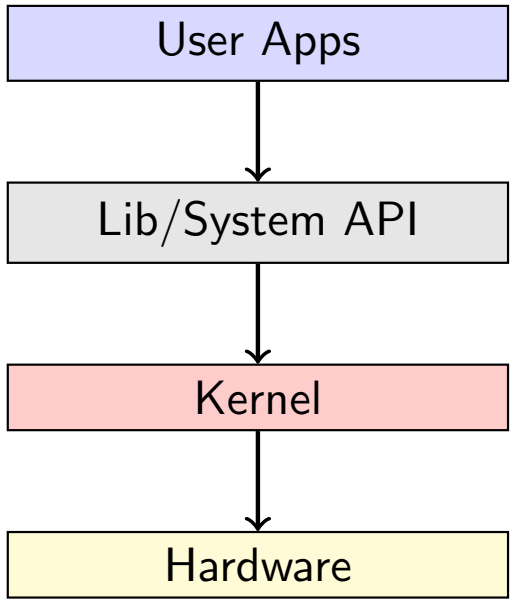


Figure: A Typical Monolithic OS.

OS Structure Types

Operating systems can be architected in different ways, each with trade-offs in performance, modularity, and reliability.

Three Classical Architectures:

- **Monolithic:** All OS services run in kernel space as one large binary.
- **Layered:** OS is divided into layers, each built on top of the lower one.
- **Microkernel:** Only essential services in kernel; others run in user space.

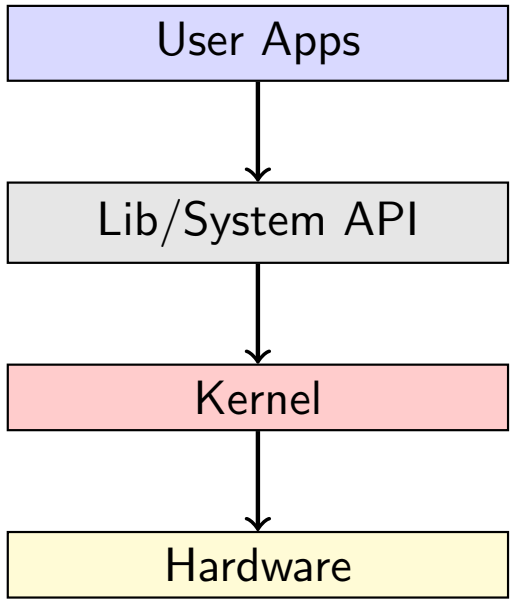


Figure: A Typical Monolithic OS.

Did You Know?

The original UNIX was monolithic, but modern OSes like macOS and Windows use hybrid kernels to balance performance and reliability.

OS Architecture Comparison

1. Monolithic Kernel
All OS services (file system, memory, device drivers) run in kernel space as a single large binary.
Example: Early UNIX, Linux
Pros: Fast due to direct function calls
Cons: Poor modularity; bugs can crash entire system

2. Layered Architecture
OS is divided into hierarchical layers, each built on top of the lower one.
Example: THE OS (Dijkstra), early Windows NT
Pros: Clear separation of concerns
Cons: Rigid structure; performance overhead

3. Microkernel
Only essential services (e.g., IPC, scheduling) run in kernel space; others (e.g., drivers, file system) run in user space.
Example: MINIX, QNX, seL4
Pros: High modularity, better fault isolation
Cons: Performance overhead due to IPC

4. Hybrid Kernel (Modern)
Combines monolithic speed with microkernel modularity.
Example: Windows NT, macOS (XNU)
Pros: Balanced performance and modularity
Cons: Increased complexity

Trade-off: *Performance vs. Modularity and Maintainability*

Key Takeaways

- **System Calls** serve as the secure interface between user applications and the OS kernel, enabling controlled access to hardware and services.
- **OS Structure** directly impacts performance, modularity, and fault isolation. The choice of architecture influences how the OS handles complexity and failures.
- **Real-World Usage:**
  - Linux uses a *hybrid monolithic* model — fast but extensible.
  - Microkernels (e.g., QNX, MINIX) are favored in embedded and safety-critical systems for their reliability and modularity.

Key Takeaways

- **System Calls** serve as the secure interface between user applications and the OS kernel, enabling controlled access to hardware and services.
- **OS Structure** directly impacts performance, modularity, and fault isolation. The choice of architecture influences how the OS handles complexity and failures.
- **Real-World Usage:**
  - Linux uses a *hybrid monolithic* model — fast but extensible.
  - Microkernels (e.g., QNX, MINIX) are favored in embedded and safety-critical systems for their reliability and modularity.

Reflect

How does the OS structure influence the way you design applications or system tools? Would you prefer performance, modularity, or fault tolerance?

Next Week Preview: Process Management

Topics to Explore:

- **Process Lifecycle:** States (new, ready, running, waiting, terminated) and transitions
- **Process Control Block (PCB):** Data structure storing process metadata (PID, state, registers, etc.)
- **Scheduling Queues:** Ready, waiting, and job queues; how processes are selected for execution
- **Context Switching:** Saving and restoring process state during CPU switches
- **Hands-on:** Shell-based process creation using `fork()`, `exec()`, and tracing with `strace`, `ps`, `top`

Next Week Preview: Process Management

Topics to Explore:

- **Process Lifecycle:** States (new, ready, running, waiting, terminated) and transitions
- **Process Control Block (PCB):** Data structure storing process metadata (PID, state, registers, etc.)
- **Scheduling Queues:** Ready, waiting, and job queues; how processes are selected for execution
- **Context Switching:** Saving and restoring process state during CPU switches
- **Hands-on:** Shell-based process creation using `fork()`, `exec()`, and tracing with `strace`, `ps`, `top`

Prep Tip

Try running `ps -ef`, `top`, and `strace ls` on your system. Observe how processes are created, scheduled, and interact with the kernel in real time.

Outline

Discussion Prompt

Which OS architecture would you choose for:

- A desktop OS like Ubuntu?
- A safety-critical OS in a car or drone?
- A distributed file system controller?

Discussion Prompt

Which OS architecture would you choose for:

- A desktop OS like Ubuntu?
- A safety-critical OS in a car or drone?
- A distributed file system controller?

Activity

Discuss with a partner. Justify your choice based on performance, reliability, and extensibility.

Quiz: System Calls and OS Structures

Test Your Understanding:

1. What is the difference between a library function and a system call?
2. Why do system calls require a switch to kernel mode?
3. Which OS structure offers the best modularity and fault isolation?
4. True or False: Microkernels are slower because they use message passing.
5. Challenge: Can you name a real-world OS that uses a microkernel architecture?

Quiz: System Call Internals

Challenge your understanding:

1. What is the role of the syscall number in the system call interface?
2. How does the OS validate arguments passed from user space?
3. Why is it unsafe to allow user programs to directly invoke kernel functions?
4. True or False: All system calls are synchronous and blocking.
5. Bonus: What is the difference between a trap and an interrupt?

Quiz: OS Structure Design Trade-offs

Apply your knowledge:

1. Why might a microkernel be preferred in a mission-critical embedded system?
2. What are the performance implications of message passing in microkernels?
3. How does layering improve maintainability but potentially reduce performance?
4. True or False: In a monolithic kernel, a bug in one module can crash the entire system.
5. Bonus: Can you name a real-world OS that uses a hybrid kernel model?

Exercise: Trace a System Call

Objective: Understand how a system call travels from user space to kernel space and back.

Task

Use `strace` or `dtrace` to trace the execution of a simple program that opens and reads a file. Document the system calls made and identify:

- The syscall names and arguments
- The transition point to kernel mode
- The return values and error handling

Exercise: Trace a System Call

Objective: Understand how a system call travels from user space to kernel space and back.

Task

Use `strace` or `dtrace` to trace the execution of a simple program that opens and reads a file. Document the system calls made and identify:

- The syscall names and arguments
- The transition point to kernel mode
- The return values and error handling

Bonus: Compare the trace output on Linux vs. macOS or Windows (if available).

Explore Further: Advanced Topics

Want to go deeper? Consider exploring:

- How system calls are implemented in assembly (e.g., `syscall` instruction)
- Kernel modules and loadable drivers
- System call filtering and security (e.g., `seccomp`, `ptrace`)
- OS structure in real-world kernels (Linux, Minix, QNX)
- Performance benchmarking of syscall overhead

Explore Further: Advanced Topics

Want to go deeper? Consider exploring:

- How system calls are implemented in assembly (e.g., `syscall` instruction)
- Kernel modules and loadable drivers
- System call filtering and security (e.g., `seccomp`, `ptrace`)
- OS structure in real-world kernels (Linux, Minix, QNX)
- Performance benchmarking of syscall overhead

Challenge

Can you write a minimal kernel module that logs system calls?

System Call Example: File I/O

Objective: Read from a file and write its contents to standard output using system calls.

```c
#include <fcntl.h>
#include <unistd.h>

int main() {
    char buffer[128];
    int fd = open("example.txt", O_RDONLY);
    int n = read(fd, buffer, sizeof(buffer));
    write(STDOUT_FILENO, buffer, n);
    close(fd);
    return 0;
}
```

Syscalls used: `open()`, `read()`, `write()`, `close()`

System Call Example: Process Creation

Objective: Create a child process and execute a new program.

```c
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        execlp("ls", "ls", "-l", NULL);
    } else {
        wait(NULL);
    }
    return 0;
}
```

Syscalls used: `fork()`, `execlp()`, `wait()`

System Call Example: Interprocess Communication

Objective: Use a pipe to send data from parent to child.

```c
#include <unistd.h>

int main() {
    int fd[2];
    pipe(fd);
    if (fork() == 0) {
        close(fd[1]);
        dup2(fd[0], STDIN_FILENO);
        execlp("wc", "wc", "-w", NULL);
    } else {
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
        close(fd[1]);
    }
    return 0;
}
```

Syscalls used: `pipe()`, `fork()`, `dup2()`, `execlp()`, `write()`

System Call Example: Memory Mapping

Goal: Map a file into memory and read from it.

```c
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int fd = open("example.txt", O_RDONLY);
    char *data = mmap(NULL, 100, PROT_READ,
        MAP_PRIVATE, fd, 0);
    write(STDOUT_FILENO, data, 100);
    munmap(data, 100);
    close(fd);
    return 0;
}
```

Syscalls: `open()`, `mmap()`, `munmap()`, `write()`, `close()`

System Call Example: Signals

Goal: Handle and send a signal between processes.

```c
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

void handler(int sig) {
    write(STDOUT_FILENO, "Signal caught!\n", 15);
}

int main() {
    signal(SIGUSR1, handler);
    if (fork() == 0) {
        kill(getppid(), SIGUSR1);
    } else {
        pause(); // Wait for signal
    }
    return 0;
}
```

Syscalls: `signal()`, `kill()`, `pause()`, `getppid()`

System Call Example: Error Handling

Goal: Demonstrate how to handle syscall errors using `errno`.

```c
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main() {
    int fd = open("nonexistent.txt", O_RDONLY);
    if (fd == -1) {
        printf("Error: %s\n", strerror(errno));
    }
    return 0;
}
```

Syscalls: `open()`, `strerror()`, `errno`