Operating Systems – Week 2 Lecture Notes

Instructor: SDB
Theme: Execution Management
Topic: *System Calls and OS Structures*

Lecture Script

Welcome to Week 2!

Today, we'll get technical — we'll see how user programs request services from the OS via **system calls**, and how the OS is structured internally to handle those requests.

Think of a system call as a "doorway" into the kernel. When a user program wants to read a file, create a process, or allocate memory, it must use a system call — no exceptions.

We'll also look at different OS structure designs — how monolithic vs layered vs microkernel systems differ in handling those calls.

Let's start with system calls.

Core Concepts and Definitions

❖ *What is a System Call?*

A **system call** is the primary mechanism through which **user-mode programs request services from the kernel**.

It triggers a controlled transition from **user mode to kernel mode**, executes the service, then returns.

❖ *How System Calls Work*

1. Application code calls a library function (e.g., `printf()`)
2. Library internally invokes a system call (e.g., `write()`)
3. A **trap instruction** causes a switch to kernel mode
4. Kernel identifies the syscall number and dispatches the handler
5. Execution returns to user mode with results

Common Process-Related System Calls

System Call	Purpose
fork()	Create a new child process (clone of parent)
exec()	Replace current process memory with new program
wait()	Parent waits for child to terminate
exit()	Terminates process and returns status

Example: Running `ls`

What happens when you type `ls` in the shell?

```
shell (parent)
 ├── fork() → child process
 └── child calls exec("ls")
     └── kernel loads /bin/ls into process memory
         └── ls executes, exits, status returned to shell (via wait())
```

Each step involves a system call.

Live Trace Demo with strace

```
strace ls
```

Shows syscall-level trace:

```
execve("/bin/ls", ["ls"], ...) = 0
brk(NULL) = 0x...
openat(AT_FDCWD, ".", O_RDONLY|...)
...
```

Use this in class to show real-time kernel boundary crossings.

OS Structural Models

❖ *Monolithic Kernel*

- All OS components (file systems, device drivers, schedulers) run in kernel space.
- Fast but harder to maintain.

Example: Traditional Linux

❖ *Layered OS*

- OS is divided into layers: hardware interface at bottom, user interface at top.
- Each layer can call services only of the lower layer.

Example: THE OS (classic educational model)

❖ *Microkernel*

- Minimal kernel: only process mgmt, memory, IPC in kernel.
- Other components (drivers, FS, servers) in user space.

Example: Minix, QNX, modern macOS kernels (hybrid)

Glossary of Terms

Term	Definition
System Call	Interface for invoking OS services from user space
Trap	CPU instruction to switch to kernel mode
Kernel Mode	Privileged mode of execution
User Mode	Restricted mode for user applications
Fork	System call to create a process

Term	Definition
Exec	Replaces process memory with a new program
Microkernel	Minimal kernel architecture pushing components to user space
Monolithic	All OS code resides in a single kernel address space

---

Live Demos and Exercises (In-Class)

```
strace ls                    # Show system call sequence
man 2 fork                   # Review manual page of syscall
gcc fork_demo.c -o fork_demo   # Run sample C fork/exec/wait code
```

Optional code snippet for demo:

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

Exploration Topics

- How syscall tables are implemented (syscall numbers)
- How syscall wrappers work in libc
- Kernel-bypass systems: DPDK, RDMA
- Compare IPC in monolithic vs microkernel systems
- Research syscall overhead benchmarks

☑Summary

- System calls bridge user programs to the kernel
- fork(), exec(), wait(), and exit() are foundational
- OS structures vary in modularity and performance trade-offs
- Tools like strace make syscall behavior observable and teachable

---

✍ Review & Exercises

1. Draw the control flow of fork → exec → wait → exit
2. What happens if a program calls exec() but not wait()?
3. Write a C program that forks and prints child/parent PIDs
4. Trace system calls in a GUI app (strace gedit)
5. Compare microkernel vs monolithic with 2 pros and cons each
6. Try modifying a shell to log each syscall invoked (advanced)
