Lecture 3 — Operating System Structure; Traps

Jeff Zarnett & Carlos Moreno
jzarnett@uwaterloo.ca & cmoreno@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 18, 2017

Operating System Structure

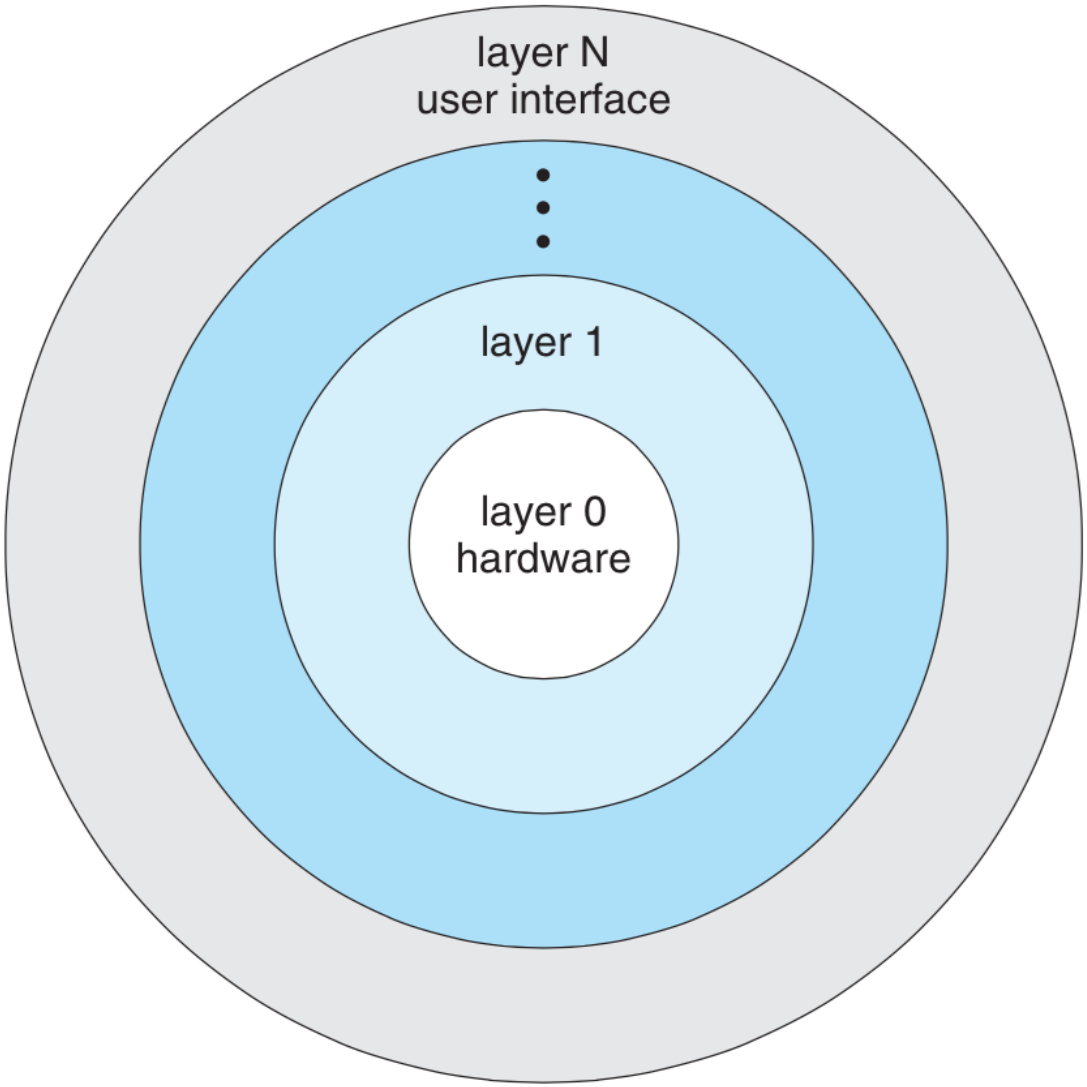
An operating system is large and complex and this is not the sort of thing programmers create at a whim with no planning.

It is possible to write an OS without a defined structure.

Like any large program, it will quickly become unmaintainable.

It will very likely be bug-riddled.

Traditional UNIX Structure



Layered Structure

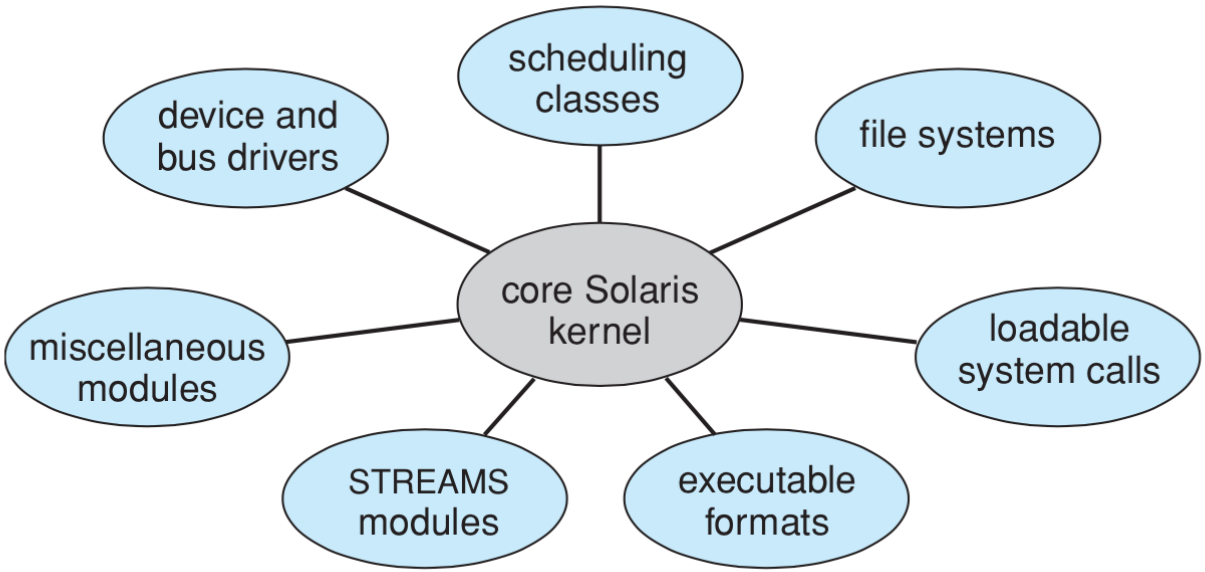
All the modern OSes tend to follow this structure, with modifications.

Another structural element within a layer: modules.

Just as we have classes and packages: modules group functionality.

Modules might be static, or loadable & swappable.
This might be during operation or with a reboot.

Sun Solaris Structure



Solaris supports loadable modules.

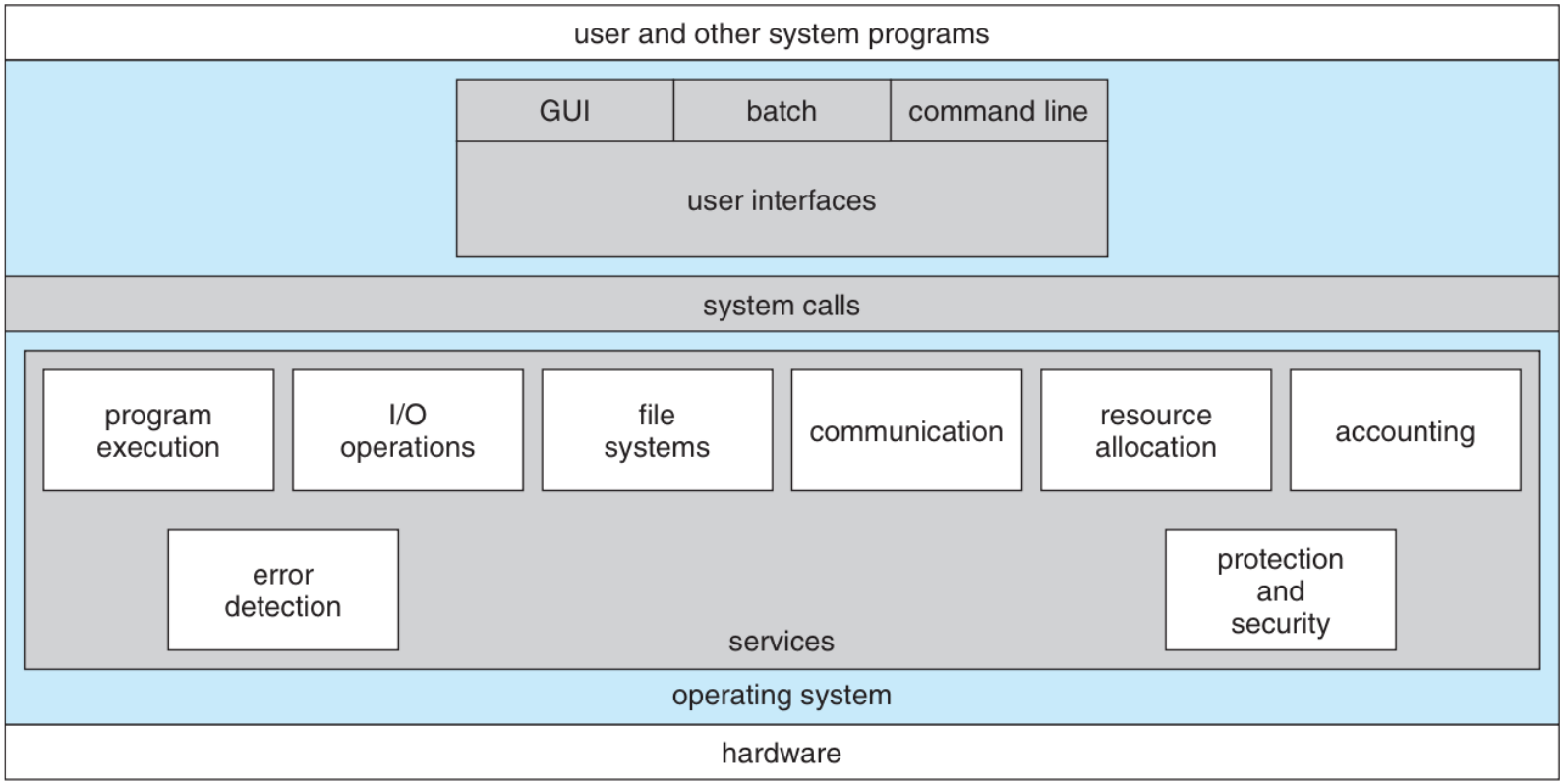
Layered Operating System with Modules

Provides the environment for the execution of programs.

Supports them in their execution by providing various services.

Some are there for convenience; others are mandatory.

Layered Operating System with Modules



Layered Operating System with Modules

Starting from the top:

- User Interface
- System Calls
- Program Execution
- I/O Operations
- File Systems
- Communication
- Resource Allocation
- Accounting
- Error Detection
- Protection and Security

Invoking System Calls

Some services run automatically, without user intervention.

In other cases, we want specifically to invoke them. How?

It’s a trap!

Trap

Operating systems run on the basis of interrupts.

A trap is a software-generated interrupt.

Generated by an error (invalid instruction) or user program request.

Trap

If it is an error, the OS will decide what to do.

Usual strategy: give the error to the program.

The program can decide what to do if it can handle it.

Often times, the program doesn't handle it and just dies.

User Mode and Kernel Mode

Already we saw user mode vs. supervisor (kernel) mode instructions.

Supervisor mode allows all instructions and operations.

Even something seemingly simple like reading from disk or writing to console output requires privileged instructions.

These are common operations, but they involve the OS every time.

User Mode and Kernel Mode

Modern processors track what mode they are in with the mode bit.

At boot up, the computer starts up in kernel mode as the operating system is started and loaded.

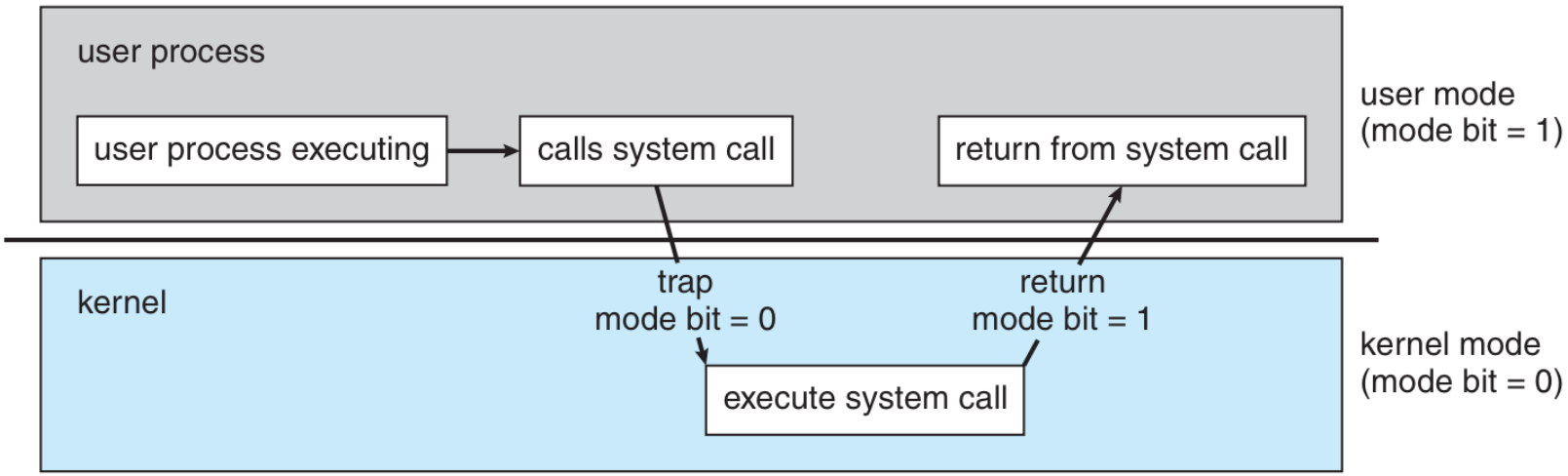
User programs are always started in user mode.

When a trap or interrupt occurs, and the operating system takes over, the mode bit is set to kernel mode.

When it is finished the system goes back to user mode before the user program resumes.

Example: Text Editor Printing

Suppose a text editor wants to output data to a printer.



User Mode and Kernel Mode: Motivation

Why do we have user and supervisor modes, anyway?

Uncle Ben to Spiderman: “with great power comes great responsibility”.

Same as why we have user accounts and administrator accounts.

To protect the system & its integrity against errant and malicious users.

User Mode and Kernel Mode: Motivation

Multiple programs might be trying to use the same I/O device at once.

Program 1 tries to read from disk. This takes some time.

If Program 2 wants to read from the same disk, the operating system forces Program 2 to wait its turn.

Without the OS, it would be up to the author(s) of Program 2 to check and wait patiently for it to become available.

Works if everyone plays nicely.

Without enforcement of the rules, a program will do something nasty.

User Mode and Kernel Mode: Motivation

There is a definite performance trade-off.

Switching from user to kernel mode takes time.

The performance hit is worth it for the security.

Example: Reading from Disk

C code to perform a read on a UNIX system.

`read` takes three parameters:

1. the file (a file descriptor, from a previous call to `open`);
2. where to read the data to; and
3. how many bytes to read.

```
int bytesRead = read( file, buffer, numBytes );
```

Note that `read` returns the number of bytes successfully read.

Example: Reading from Disk

In preparation for a call to `read` the parameters are pushed on the stack.
This is the normal way in which a procedure is called in C(++).

`read` is called; the normal instruction to enter another function.

The `read` function will put its identifier in a predefined location.

Then it executes the `trap` instruction, activating the OS.

Example: Reading from Disk

The OS takes over and control switches to kernel mode.

Control transfers to a predefined memory location within the kernel.

The trap handler examines the request: it checks the identifier.

Now it knows what system call request handler should execute: read.

That routine executes.

When it is finished, control will be returned to the `read` function.
Exit the kernel and return to user mode.

`read` finishes and returns, and control goes back to the user program.

System Call Summary

The steps, arranged chronologically, when invoking a system call are:

1. The user program pushes arguments onto the stack.
2. The user program invokes the system call.
3. The system call puts its identifier in the designated location.
4. The system call issues the `trap` instruction.
5. The OS responds to the interrupt and examines the identifier in the designated location.
6. The OS runs the system call handler that matches the identifier.
7. When the handler is finished, control exits the kernel and goes back to the system call (in user mode).
8. The system call returns control to the user program.