Week 1
Introduction

Pamela Delgado

February 20, 2019

(slides Willy Zwaenepoel)

Staff

- Instructor: Pamela Delgado
- TAs:
  - Laurent Bindschaedler
  - Jasmina Malicevic
  - Kristina Spirovska
  - Marios Kogias
  - Lucie Perrotta
- Secretary: Cecilia Bigler

Overall Goal of CS323 and CS323a

- CS323:
  - Learn principles of operating systems
- CS323a:

  NOT GIVEN THIS SEMESTER
  - See principles applied in one example, Linux

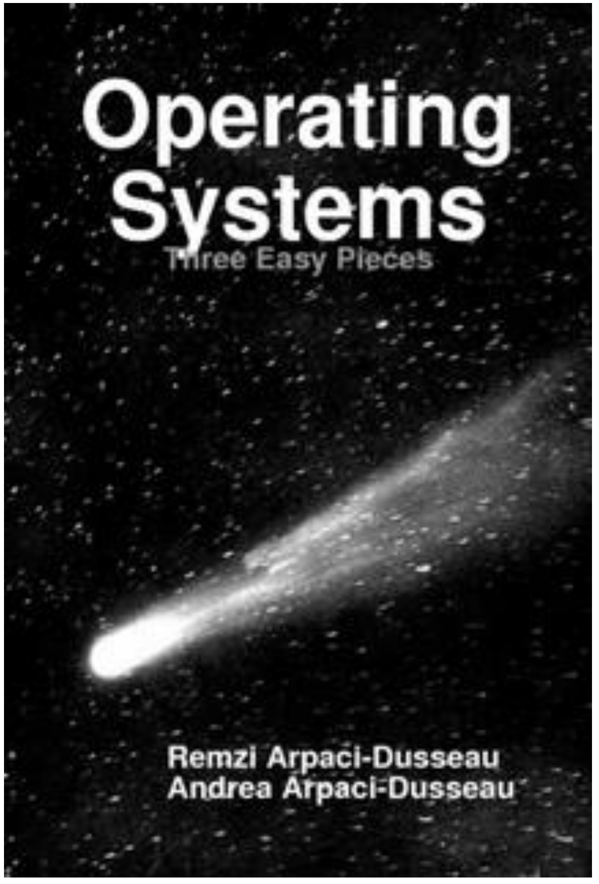
Method

- CS323:
  - Lectures / exercises

Slides

- Available before class on Moodle

Recommended Book for CS323



A *free* online book: http://pages.cs.wisc.edu/~remzi/OSTEP/

Prerequisites for CS323

- CS 206 – Concurrence
- CS 207 – Programmation orientée système

Work for CS323

- Weekly class meetings
- Weekly exercise sessions
- Midterm and final (2 hours, in class)

Tentative Class Schedule for CS323

Week	Date	Lecture	Date	Exercises
1	Feb 20	Intro	Feb 22	Intro
2	Feb 27	Process	Mar 1	Process
3	Mar 6	Process	Mar 8	Process
4	Mar 13	Process	Mar 15	Process
5	Mar 20	Memory	Mar 22	Memory
6	Mar 27	Memory	Mar 29	Memory
7	Apr 3	Memory	Apr 5	Memory
8	Apr 10	File System	Apr 12	Midterm Q/A
9	Apr 17	Midterm	Apr 19	Midterm review
	Apr 24		Apr 26	
10	May 1	File System	May 3	File system
11	May 8	File System	May 10	File System
12	May 15	File System	May 17	File System
13	May 22	Virtualization	May 24	Final Q/A
14	May 29	Final	May 31	

Grading for CS323

- 50% on midterm
- 50% on final

Questions?

Overview of Today's Lecture

- What does the OS do?
- Where does the OS live?
- OS interfaces
- OS control flow
- OS structure

What does an OS do?

A Bit of History

- Early days
  - Users program raw machine
- First “abstraction”
  - Libraries for scientific functions (sin, cos, …)
  - Libraries for doing I/O
- I/O libraries are the first pieces of an OS

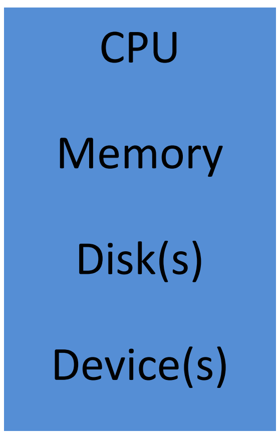
What does the OS do?

- Abstraction: makes hardware easier to use

What does the OS do?

- Abstraction: makes hardware easier to use

Hardware



What does the OS do?

- Abstraction: makes hardware easier to use

Hardware	Abstraction
CPU	Process
Memory	Address space
Disk(s)	File
Device(s)	Virtual device

What does the OS do?

- Abstraction: makes hardware easier to use

Hardware	Abstraction	OS component
CPU	Process	Process management
Memory	Address space	Memory management
Disk(s)	File	File systems
Device(s)	Virtual device	Device management

A Simple Example

- Write a photoshop application

- Easier to deal with files containing photos
- Than to deal with data locations on disk

- OS provides file abstraction
- Finds data locations on disk given file name

Another Simple Example

- Write a web server

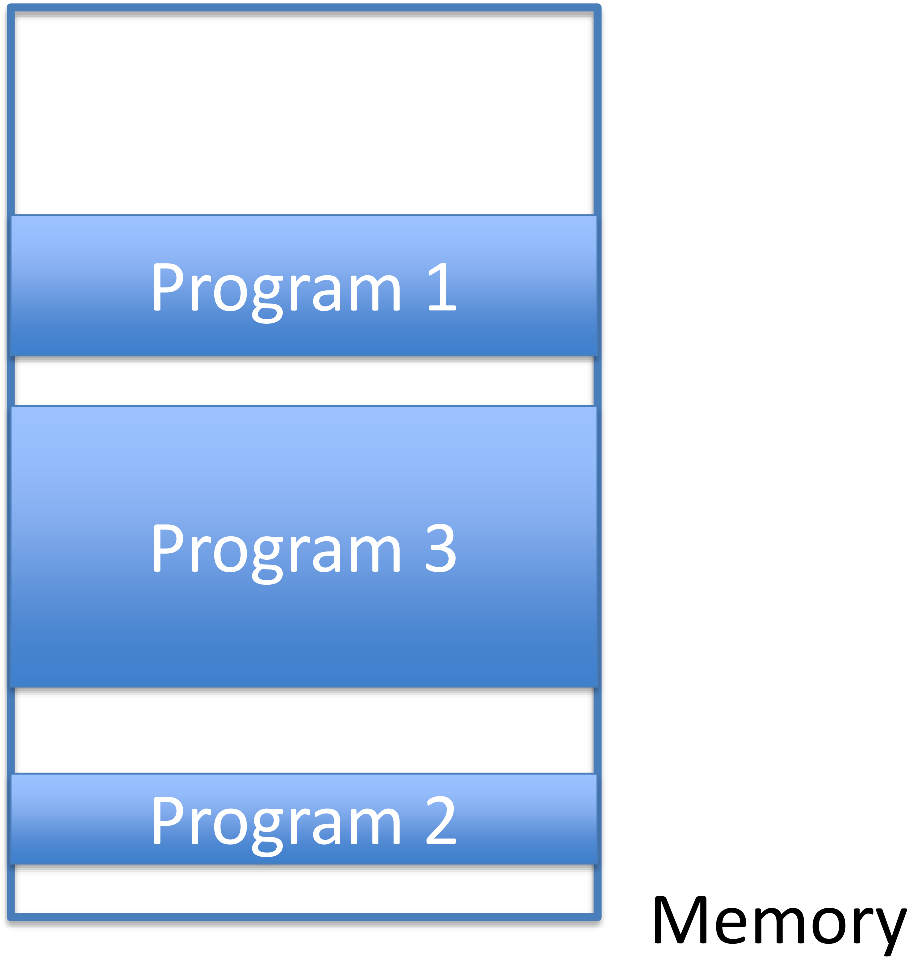
- Easier to deal with sending/receiving packets
- Than with NIC device registers

- OS provides packet abstraction
- Does the NIC device register manipulation

A Bit More History

- At some point, multiprogramming
- More than one program runs at the same time

Multiprogramming



Multiprogramming

- Need to protect programs from each other
- Need to protect OS from programs

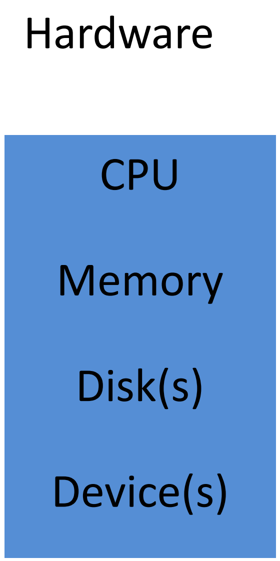
- Need to allocate/free memory

What does the OS do?

- Resource management: allocates hardware resources between programs

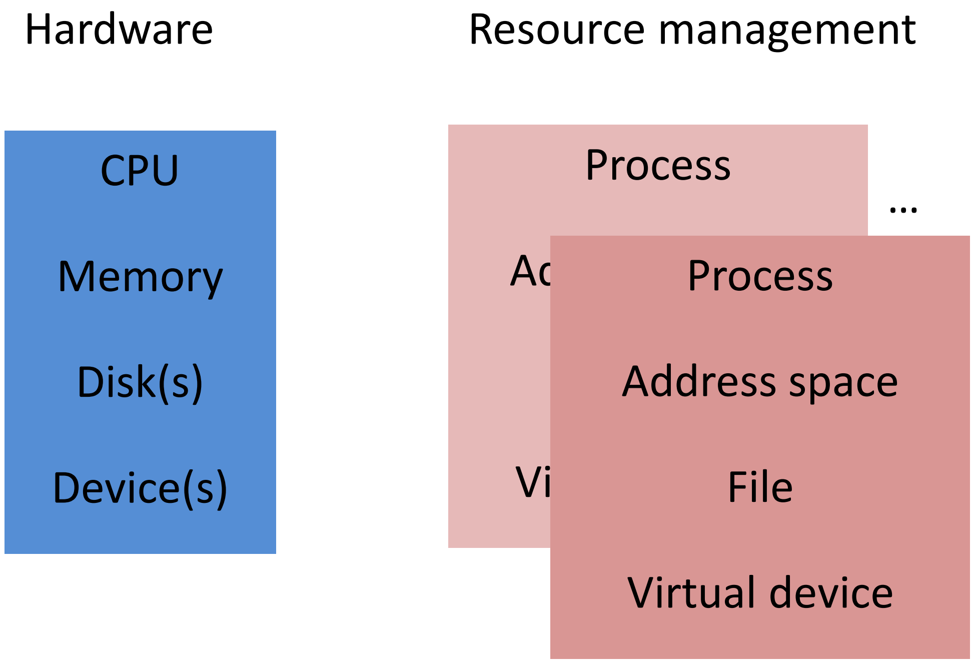
What does the OS do?

- Resource management: allocates hardware resources between programs



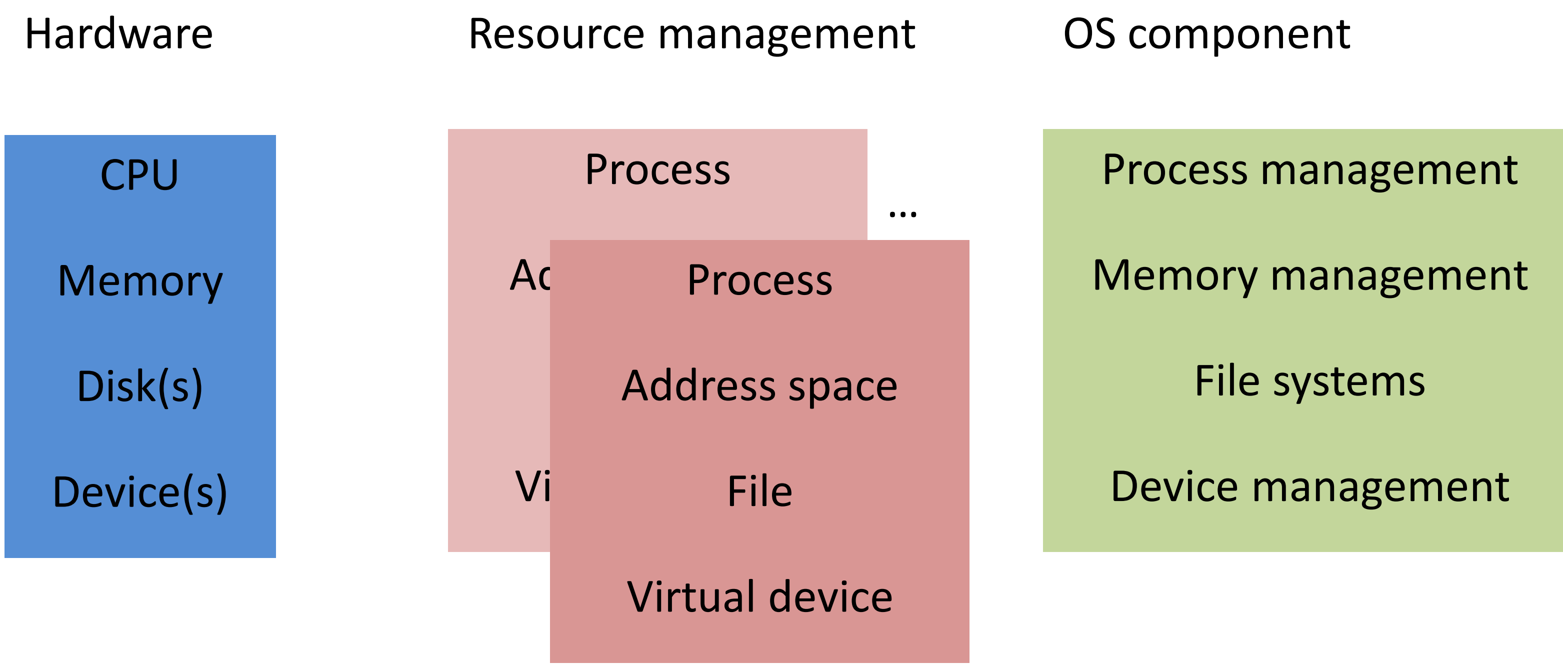
What does the OS do?

- Resource management: allocates hardware resources between programs



What does the OS do?

- Resource management: allocates hardware resources between programs



A Simple Example

- Many users want to compute

- OS allocates CPU to different users

Another Simple Example

- Many users want to use memory

- OS allocates memory between users

A Final Example

- Many files need to be stored on disk

- OS allocates disk space to files

What does the OS do?

- Abstraction: makes hardware easier to use
- Resource management: allocates hardware resources between programs

- OS does *both* at the same time

What Is and What Is Not in the OS

- Web browser: only abstraction
  - Not considered part of the OS
- Graphics library: only abstraction
  - Not considered part of the OS
- Device driver: both
  - Part of the OS
- Printer server: both
  - Part of the OS

Where does the OS live?

A Bit of Computer Architecture: CPU: Dual-Mode Operation

- Kernel mode vs. user mode
- Mode bit provided by hardware

Kernel Mode

- Privileged instructions:
  - Set mode bit
  - …
- Direct access to all of memory
- Direct access to devices

User Mode

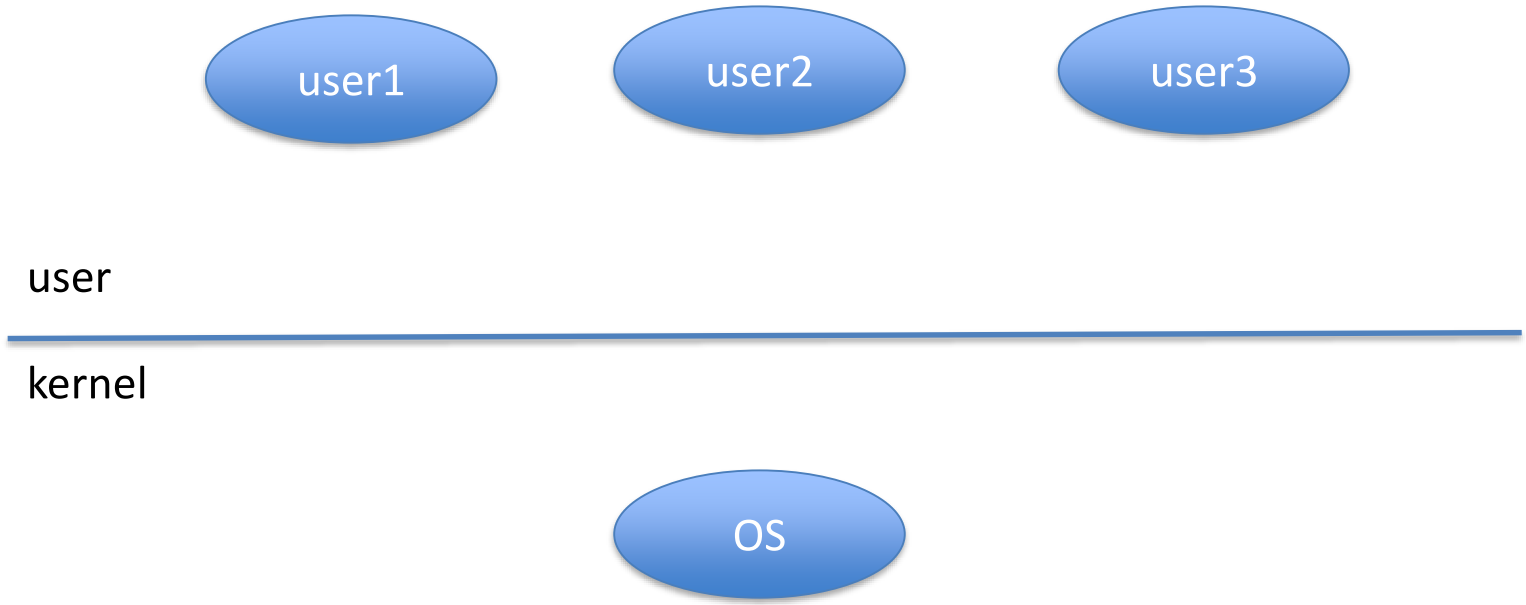
- No privileged instructions:
  - Set mode bit
  - …
- No direct access to all of memory
- No direct access to devices

In General

- OS runs in kernel mode
- Applications run in user mode

- This allows OS
  - To protect itself
  - To manage applications/devices

User/OS Separation



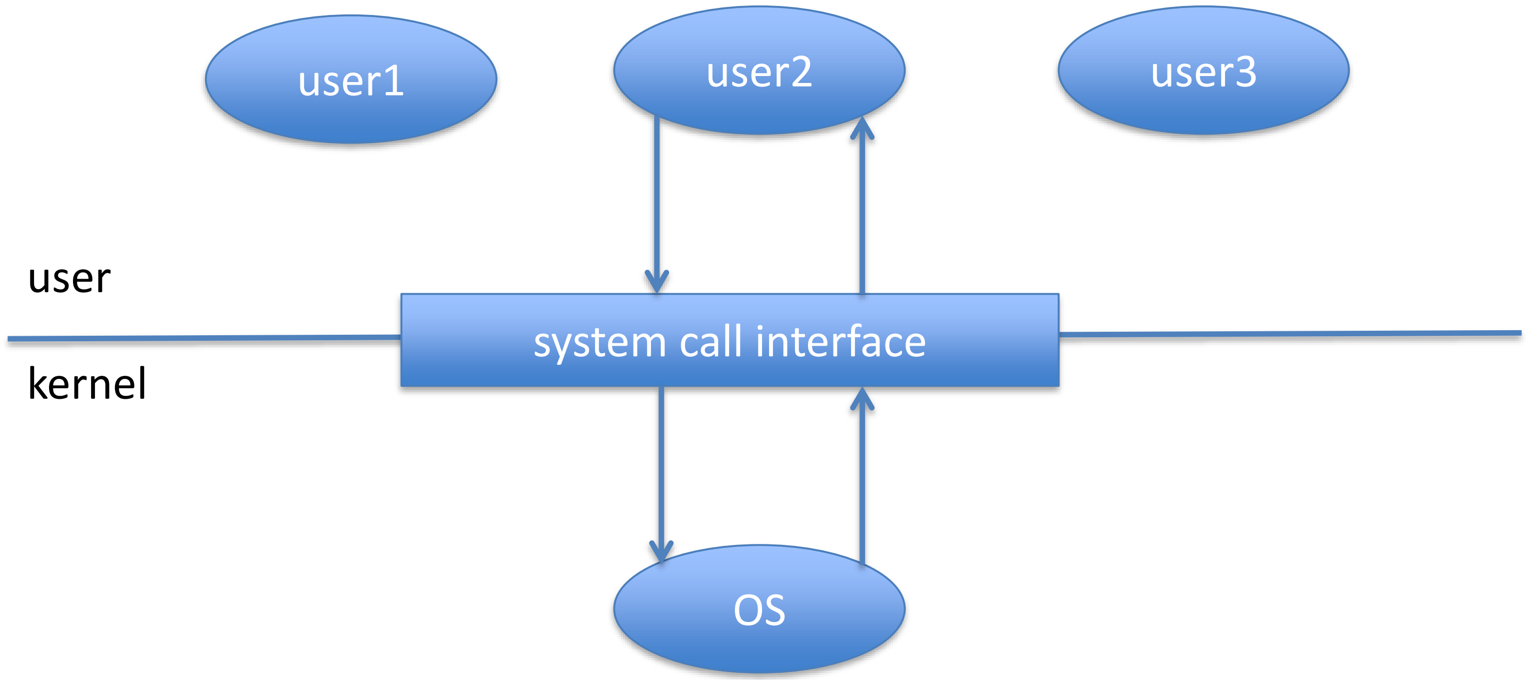
From Kernel to User Mode

- By the OS setting the mode bit to user
- Usually as a by-product of an instruction

From User to Kernel Mode

- By a device generating an interrupt
- By a program executing a trap or system call

System Calls:
Across User/Kernel Boundary



System Calls

- Are the *only* interface from program to OS
- Narrow interface essential for integrity of OS

Example System Calls

- Process management
- Memory management
- File systems
- Device management
- …

System Calls in Linux?

System call number	System call name
0	restart_syscall
1	exit
2	fork
3	read
4	write
5	open
6	close
7	waitpid
8	creat
9	link
10	unlink
…	

System Calls in Linux?

System call number	System call name
…	
350	name_to_handle_at
351	open_by_handle_at
352	clock_adjtime
353	syncfs
354	sendmmsg
355	setns
356	process_vm_readv
357	process_vm_writev

System Call Implementation

- Architecture-specific, example for x86

- Traditionally, software interrupt instruction
  - "int 0x80"
  - Raises interrupt 128
- More recently, special instructions
  - "sysenter" (on Intel)
  - "syscall" (on AMD)

System Call Identification

- Unique system call number

System call number	System call name
0	restart_syscall
1	exit
2	fork
3	read
4	write
5	open
6	close
…	

To Perform a Given System Call

- Architecture-specific, example for x86
- Put system call number in register %eax
- Execute system call instruction

System Call Parameter Passing

- Again, architecture-specific

- Put in designated registers
- Put on the stack
- Put in table and have register point to it

In Linux/x86

- System call number in %eax register
- Parameters in registers
- If more parameters, register used as pointer

Question

- Ever called the OS?

Question

- Ever called the OS?
  - Yes, of course, e.g., any file system operation.
- Ever written a system call instruction?

Question

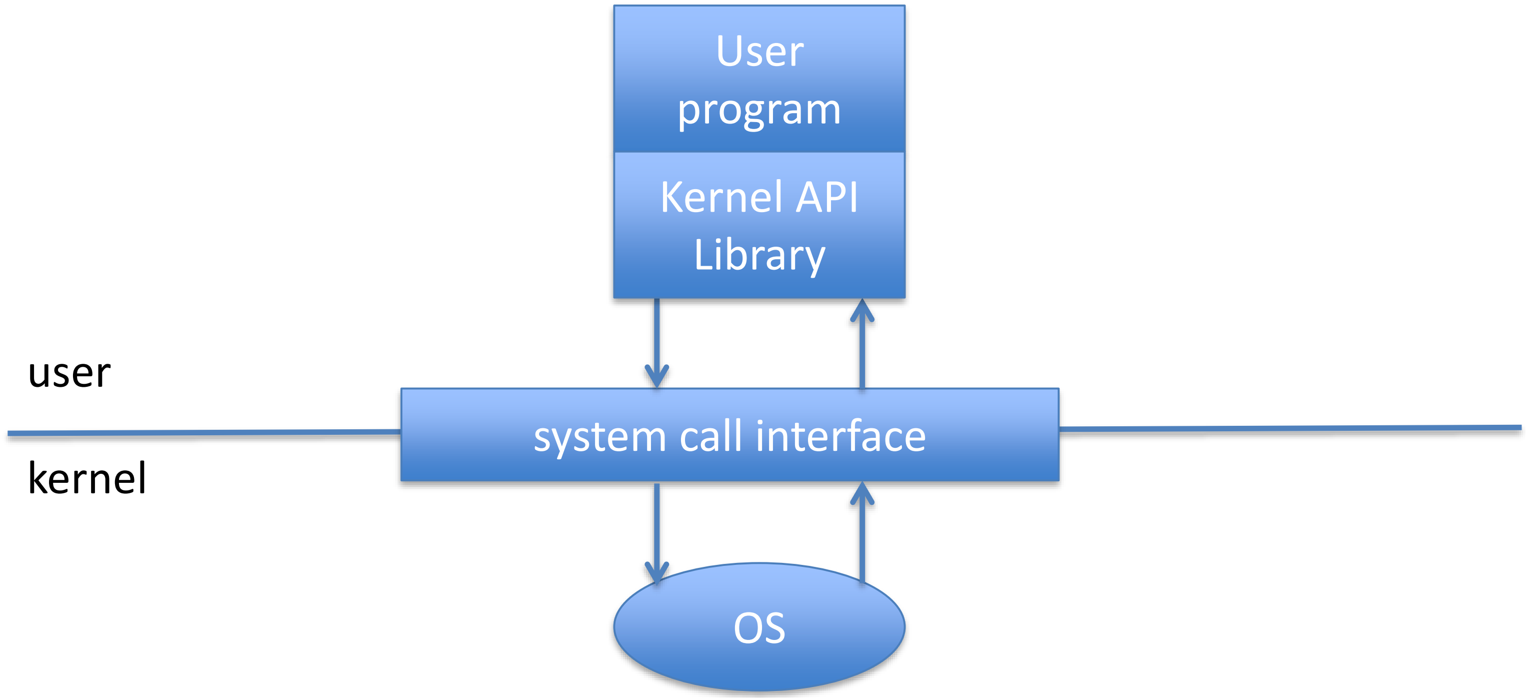
- Ever called the OS?
  - Yes, of course, e.g., any file system operation.
- Ever written a system call instruction?
  - I doubt it
- How so?

Answer: Kernel API

- A set of function calls that wrap system calls
- Easier to use
- More portable

- Example: Linux Kernel API

Kernel API


User program
Kernel API Library
user
system call interface
kernel
OS

Linux Kernel API

- Process management
  - fork(), exec(), wait(), …
- Memory management
  - mmap(), munmap(), sbrk(), …
- File system
  - open(), close(), read(), write(), …
- Device management
  - ioctl(), read(), write(), …
- Other examples
  - getpid(), alarm(), sleep(), chmod(), …

What Do Wrapper Functions Do?

- At the time of the call
  - Put arguments in registers
  - Put system call number in register %eax
  - Execute system call instruction
- At the time of the return
  - Take return value out of register
  - Return

Kernel API

```
main() {
    …
    write(…)
    …
}

write(…) {
    …
    execute system call instruction
    …
}
```

Question

- Ever called the OS?
  - Yes, of course, e.g., any file system operation.
- Ever written a system call instruction?
  - I doubt it
- Have you ever had to invoke the kernel API?

Question

- Ever called the OS?
  - Yes, of course, e.g., any file system operation.
- Ever written a system call instruction?
  - I doubt it
- Have you ever had to invoke the kernel API?
  - Maybe, maybe not

Answer: The Language Library

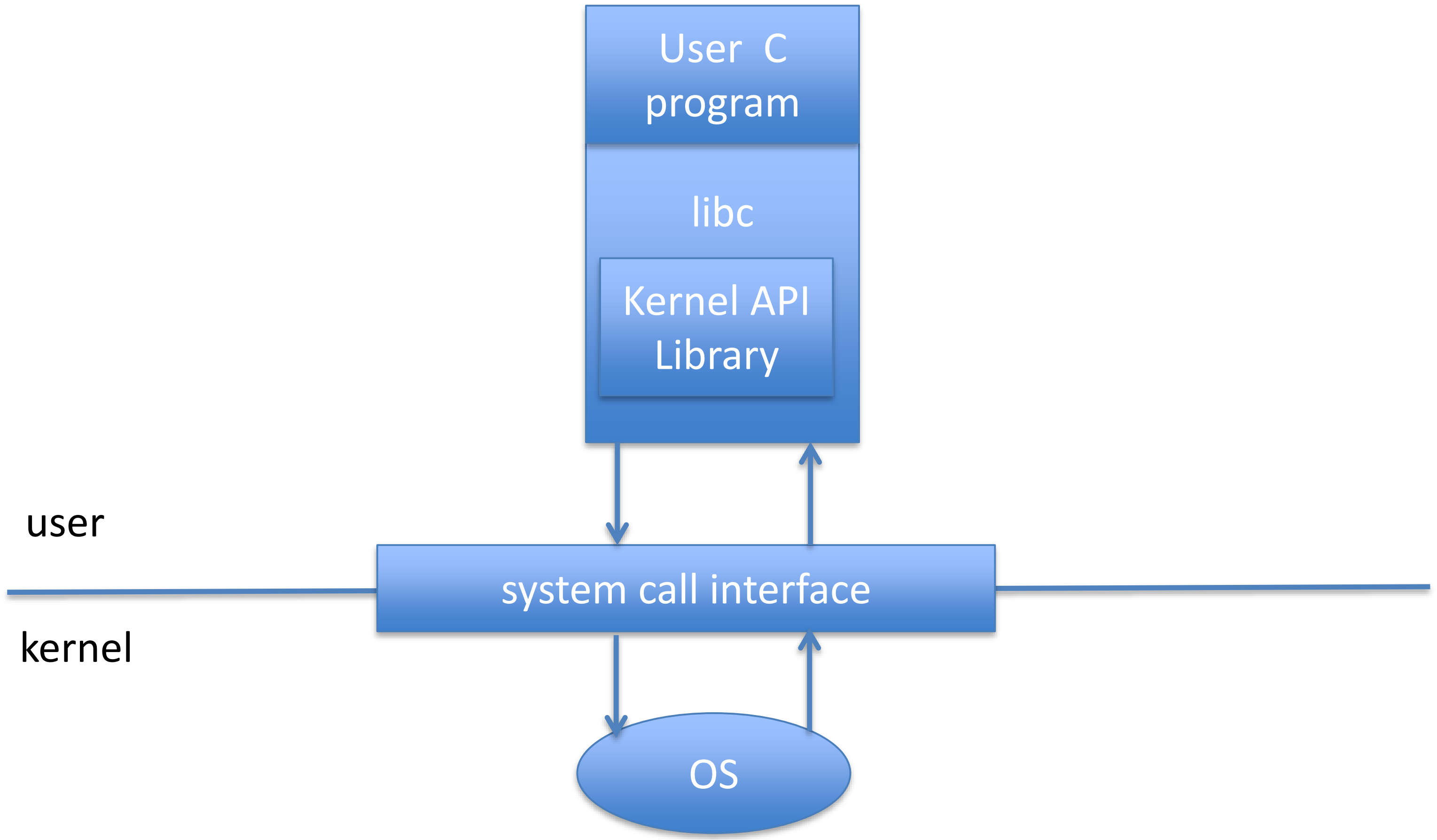
- A language-specific library
- Wraps the kernel API

- Classic example: the standard C library libc

libc

- printf, sprintf, fprintf, …
- getchar, putchar, …

libc


User C program
libc
Kernel API Library
user
system call interface
kernel
OS

libc

```
#include <stdio.h>
main() {
    …
    printf(…)
    …
}

printf(…) {
    …
    write(…)
    …
}

write(…) {
    …
    execute system call instruction
    …
}
```

Please Note!

- libc makes system call look like function call
- It is *not* a function call
- It is a user – kernel transition
  - From one program (user) to another (kernel)
  - Much more expensive

Traps

- Trap is generated by CPU as a result of error
  - Divide by zero
  - Execute privileged instruction in user mode
  - Illegal access to memory
  - …
- Works like an “involuntary” system call
  - Sets mode to kernel mode
  - Transfers control to kernel
- Identified by a trap number

Interrupts

- Generated by a device that needs attention
  - Packet arrived from the network
  - Disk i/o completed
  - …
- Identified by an interrupt number
  - Roughly speaking, identifies the device

OS Control Flow

OS Control Flow: Event-Driven Program

- Nothing to do

} Do nothing

OS Control Flow: Event-Driven Program

- Nothing to do } Do nothing

- Interrupt (from device)
- Trap (from process)
- System call (from process}

} Start running

What does the hardware do on a system call *i*?

- Puts the machine in kernel mode
- Sets the PC = SystemCallVector[i]

- SystemCallVector is a predefined location

What does the hardware do on trap *i*?

- Puts the machine in kernel mode
- Sets the PC = TrapVector[i]

- TrapVector is a predefined location

What does the hardware do on interrupt *i*?

- Puts the machine in kernel mode
- Sets the PC = InterruptVector[i]

- InterruptVector is a predefined location

Kernel Code: Initialization

```
SystemCallVector[1] = address of syscall 1 handler
routine
SystemCallVector[2] = address of syscall 2 handler
routine
….

TrapVector[1] = address of trap 1 handler routine
TrapVector[2] = address of trap 2 handler routine
…

InterruptVector[1] = address of interrupt 1 handler
routine
InterruptVector[2] = address of interrupt 2 handler
routine
…
```

Kernel Code: Main Loop

```
forever {
    wait for something to happen (HALT instruction)
    }
```

(Simplified) Execution Flow

- User executes system call *i*
- Hardware
  - Puts machine in kernel mode
  - Sets PC to SystemCallVector[i]
- Kernel
  - Executes system call *i* handler routine
  - Executes a return from kernel instruction
- Hardware
  - Puts machine in user mode
- User executes instruction after system call

OS Design Goals

- Correct abstractions
- Performance
- Portability
  - Architecture-dependent
  - Architecture-independent
- Reliability
- Other considerations:
  - E.g., on mobiles, energy conservation

A Short Note About Reliability

- OS must never fail
- Must carefully check inbound parameters
- For instance, inbound address parameter must be valid

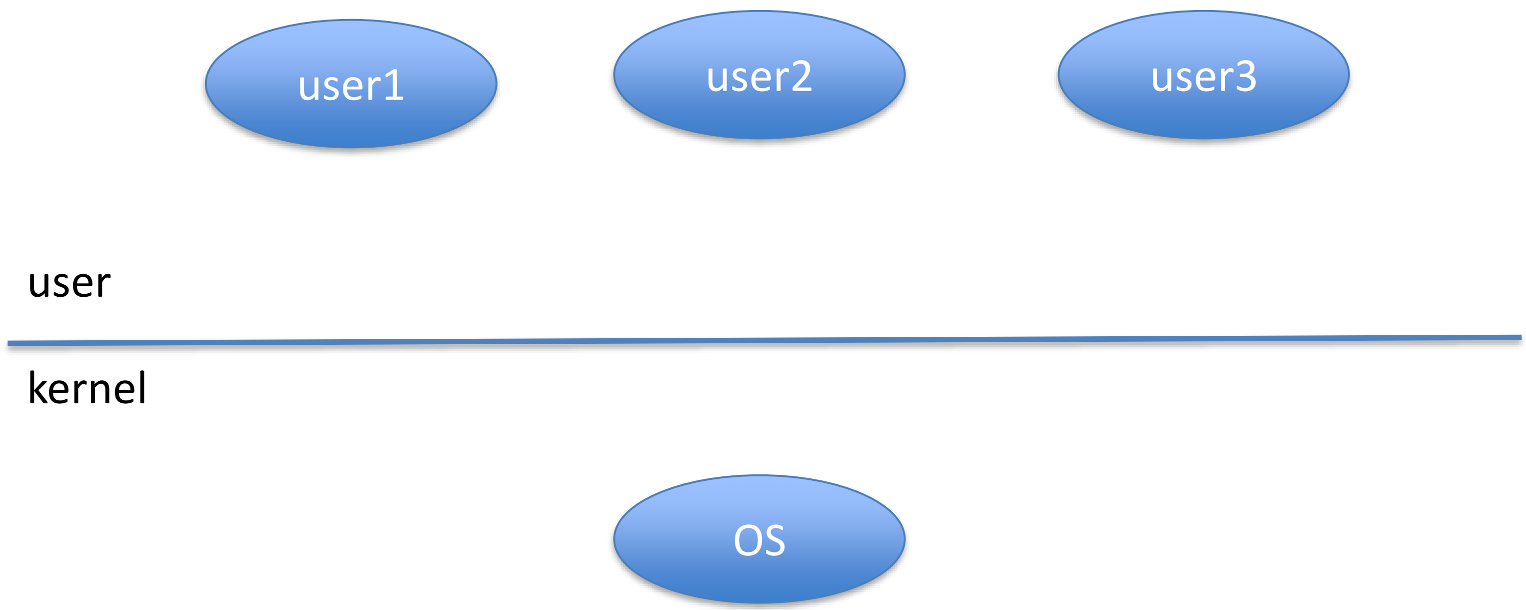
(Simplified) Execution Flow

- User executes system call *i*
- Hardware
  - Puts machine in kernel mo
  - Sets PC to SystemCallVector[i]
- Kernel
  - Executes system call *i* hander routine
  - Executes a return from kernel instruction
- Hardware
  - Puts machine in user mode
- User executes instruction after system call

includes check on inbound parameters

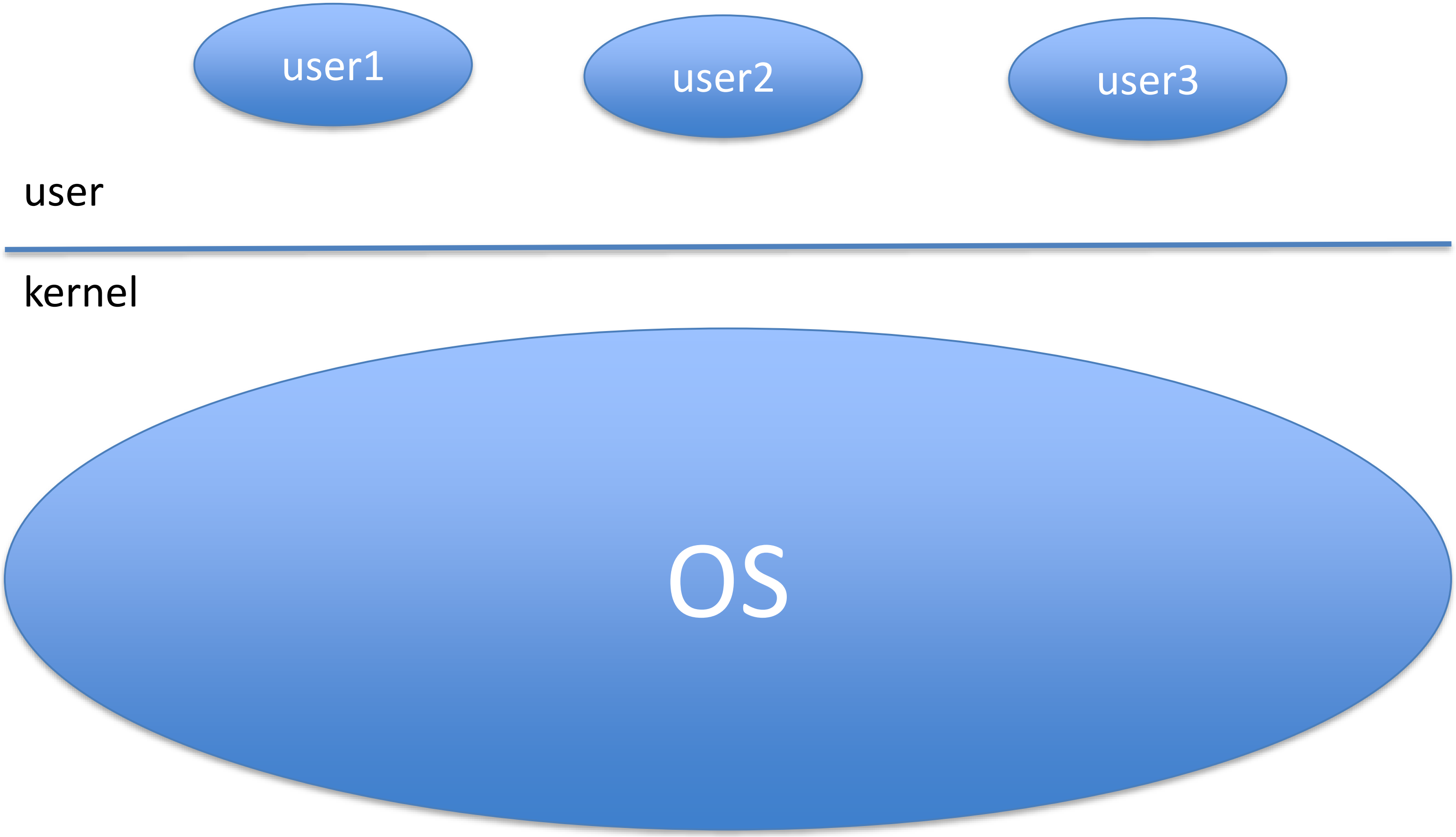
OS Structure

User/OS Separation



This approach is called the “monolithic OS”

It looks more like this



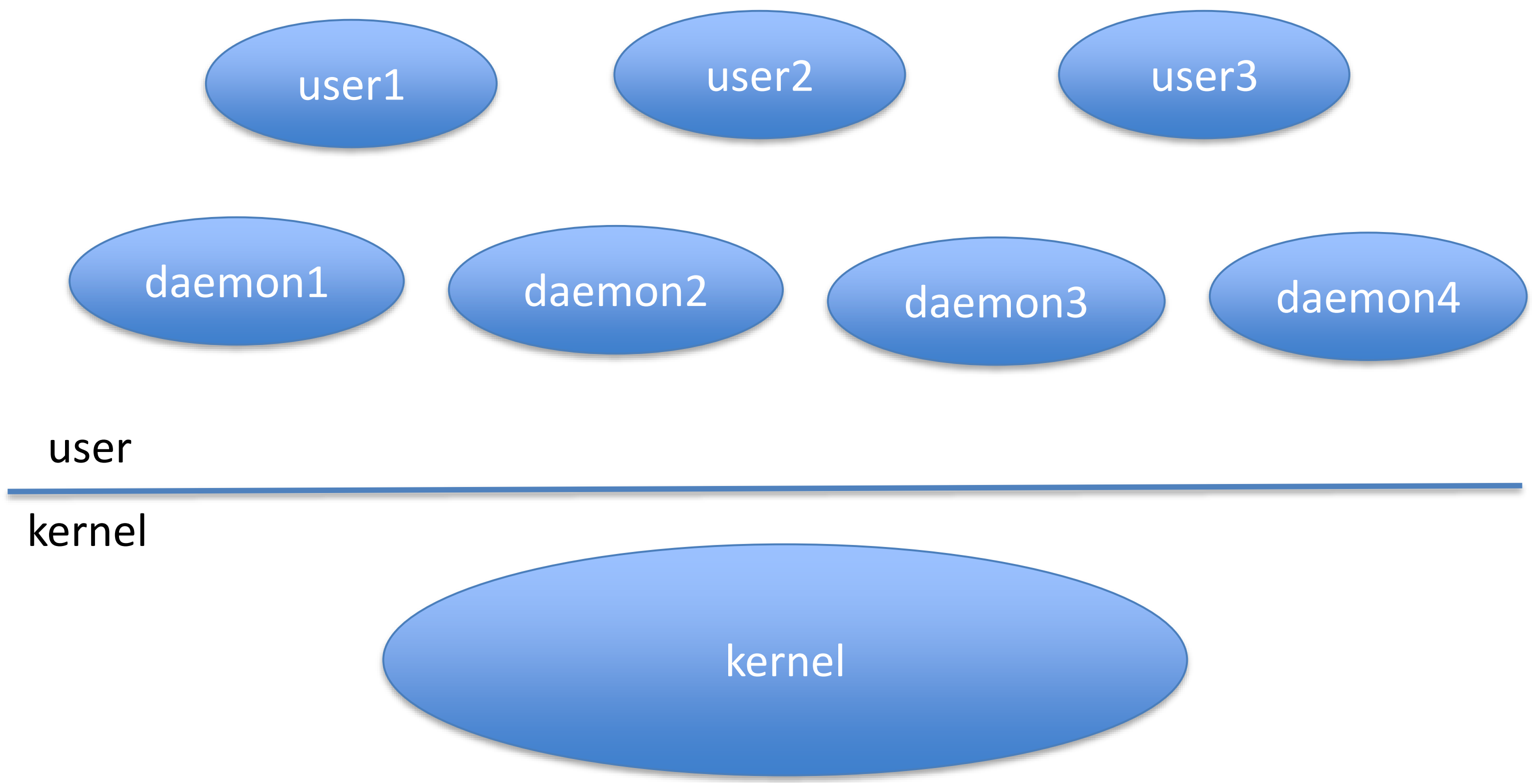
Downside of Monolithic OS

- The OS is a huge piece of software
  - Millions of lines of code and growing
- Something goes wrong in kernel mode
  - Most likely, machine will halt or crash
- Incentive to move stuff out of kernel mode

No need for entire OS in kernel mode

- Some pieces can be in user mode
  - No need for privileged access
  - No need for speed
- Example: daemons
  - System log
  - Printer daemon
  - Etc.

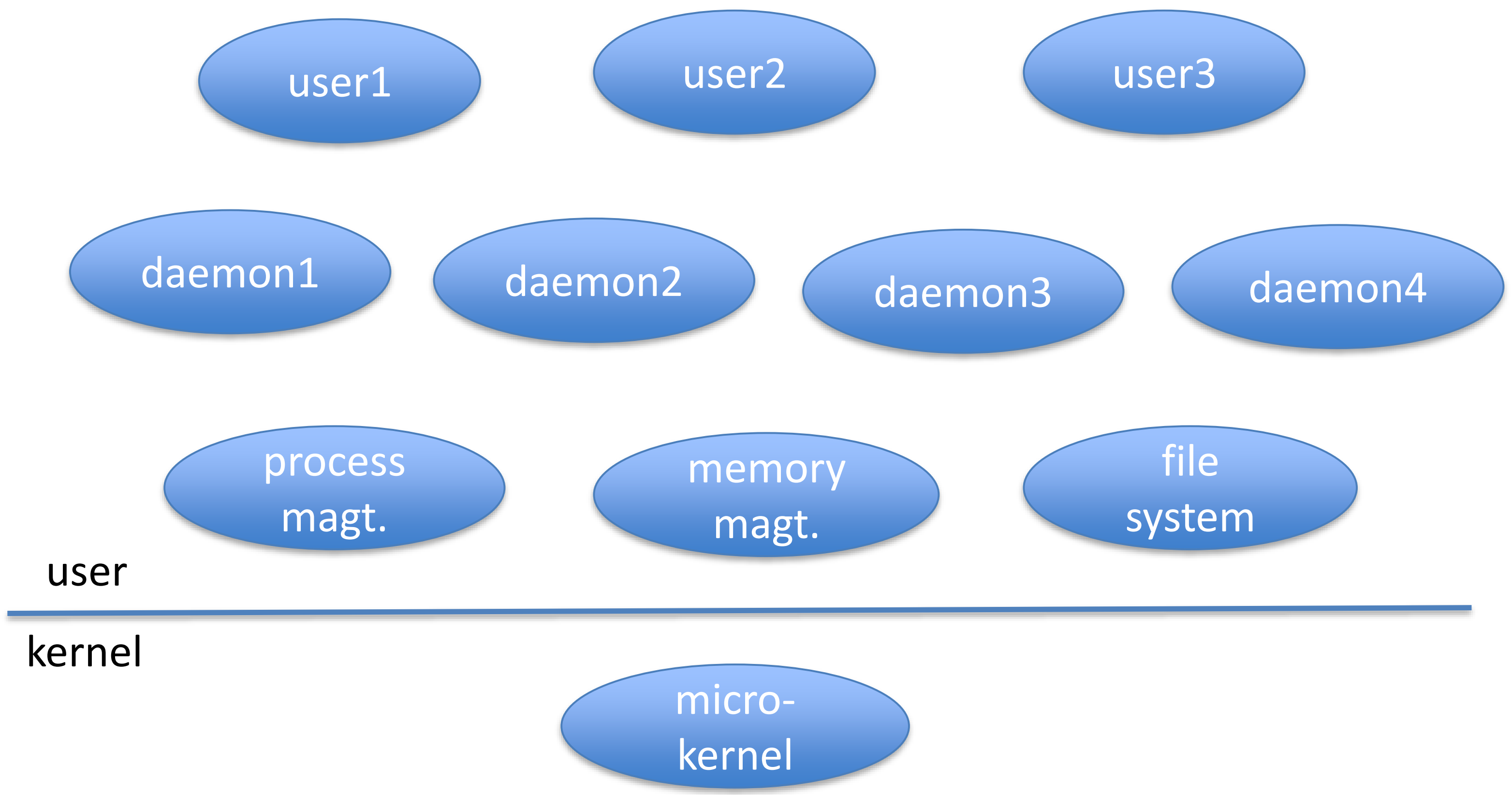
User/OS Separation: Systems Programs



The Ultimate Minimum: Microkernel

- Absolute minimum in kernel mode
  - Interprocess communication primitives
- All the rest in user mode

Microkernel



In Practice

- Microkernels have failed commercially
  - Except for niches like embedded computing
- The “systems programs” model has won out

The Price: Lines of Code in Linux Kernel

Summary

- What does the OS do?
- Where does the OS live?
- OS interfaces
- OS control flow
- OS structure

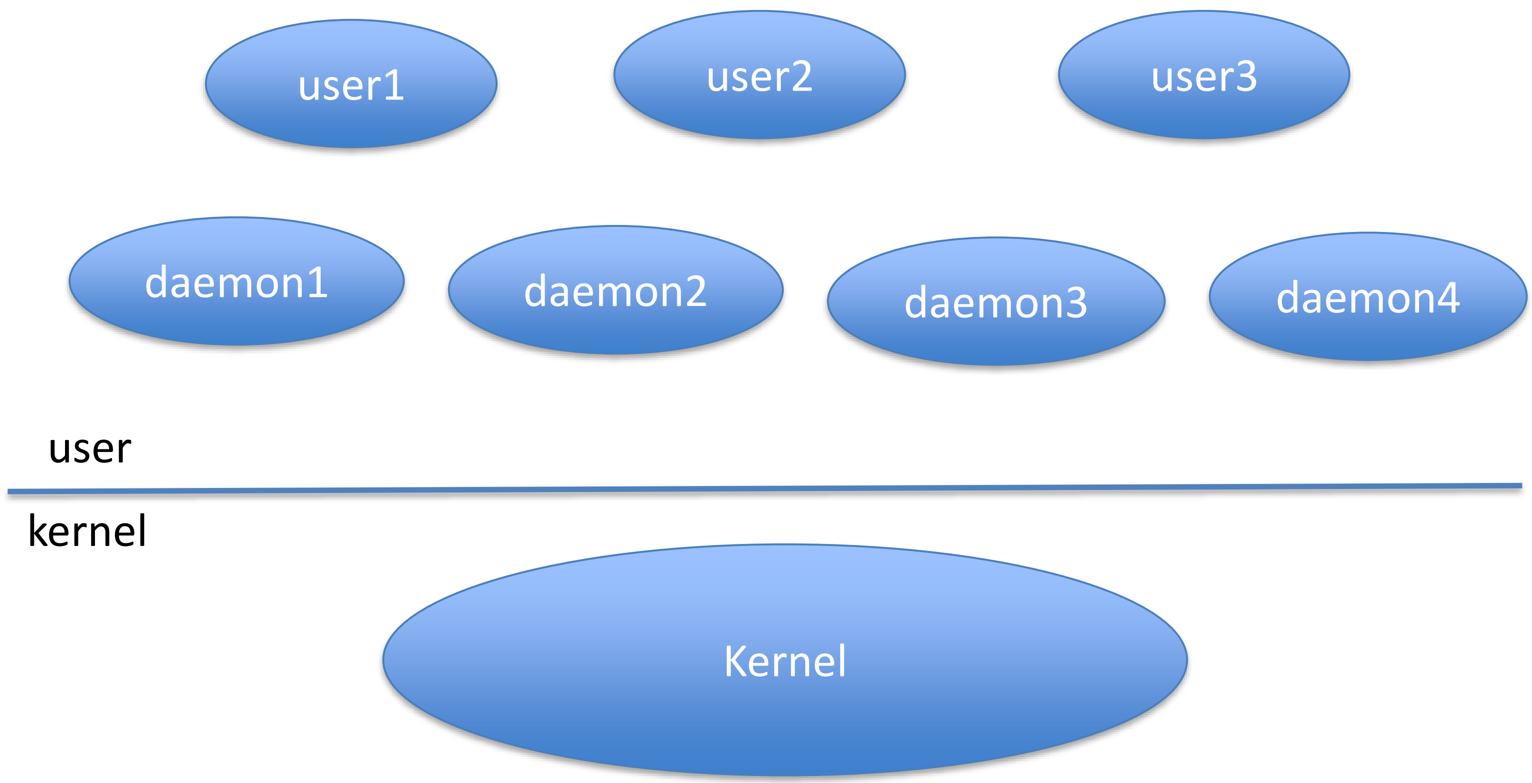
Summary: What does the OS do?

- Abstraction
- Resource management

Summary: OS Structure

- In user mode:
  - Applications
  - Systems programs
- In kernel mode:
  - Kernel

Summary: OS Structure



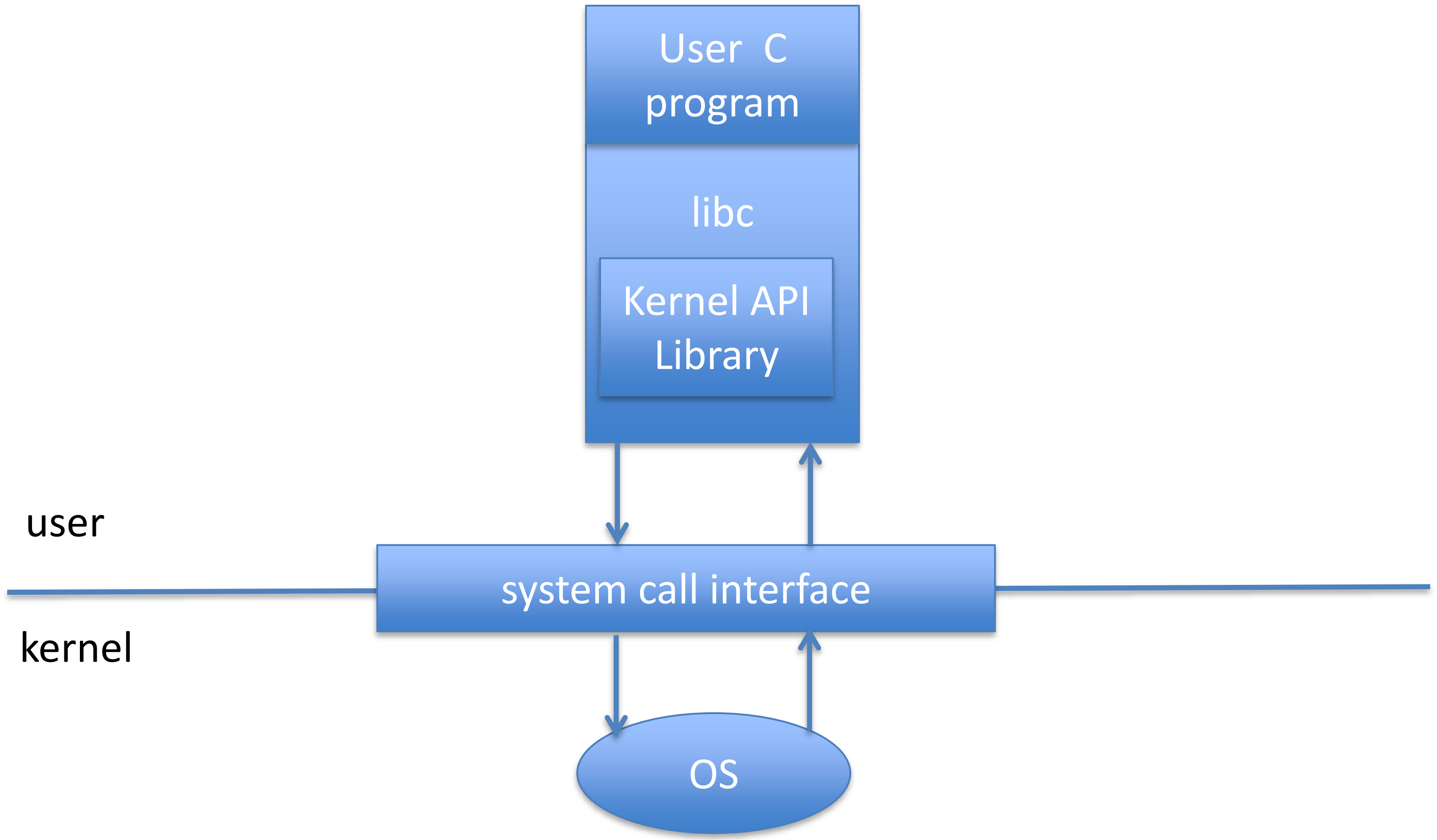
Summary: Where does the OS live?

- OS in (hardware) kernel mode
- Programs in (hardware) user mode

Summary: OS APIs

- System call
- Kernel API
- Language library

Summary: OS API



Summary: OS Control Flow

- Event-driven
- Idle loop
- Broken by system call, trap or interrupt