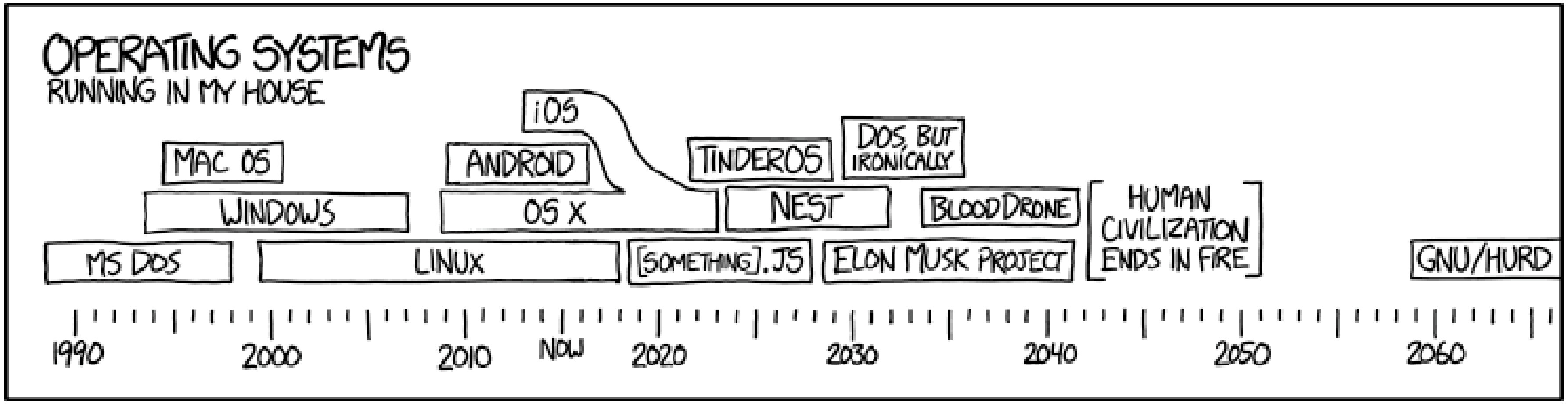
OS Structure

Kevin Webb

Swarthmore College

January 23, 2020

Relevant xkcd #1508:



One of the survivors, poking around in the ruins with the point of a spear, uncovers a singed photo of Richard Stallman. They stare in silence. "This," one of them finally says, "This is a man who BELIEVED in something."

Today's Goals

- Broad strokes: processes, resources, and protection
- Terminology (kernel, interrupts, traps, system calls, exceptions, …)
- Operating system structure and design patterns

Kernel vs. Userspace: Terminology

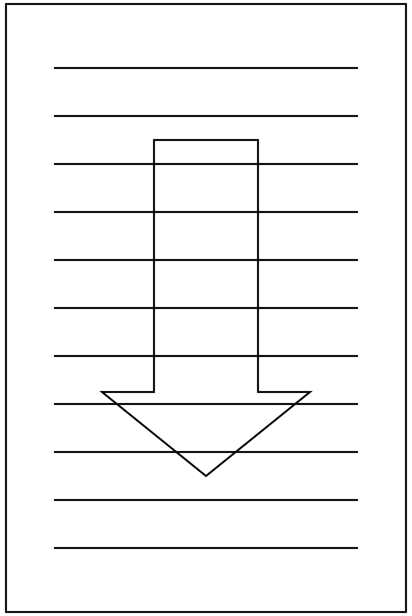
- “OS” & “Kernel” - interchangeable in this course
- Compiled Linux kernel: ~5-10 MB
- Fully installed system - a few GB
  - Most of this is user-level programs that get executed as processes
  - System utilities, graphical window system, shell, text editor, etc.

Primary Abstraction: The Process

- Abstraction of a running program
  - a dynamic “program in execution”

- Program: blueprint
- Process: constructed building

- Program: class
- Process: instance



Basic Process Resources

1. CPU Time – execute a stream of instructions

2. Main memory storage – store variables / scratch space

3. Input/Output (I/O) – interact with the outside world

4. Also: State (metadata) bookkeeping – kernel data structures
    - Programmer / user doesn't see this
    - Details next time…

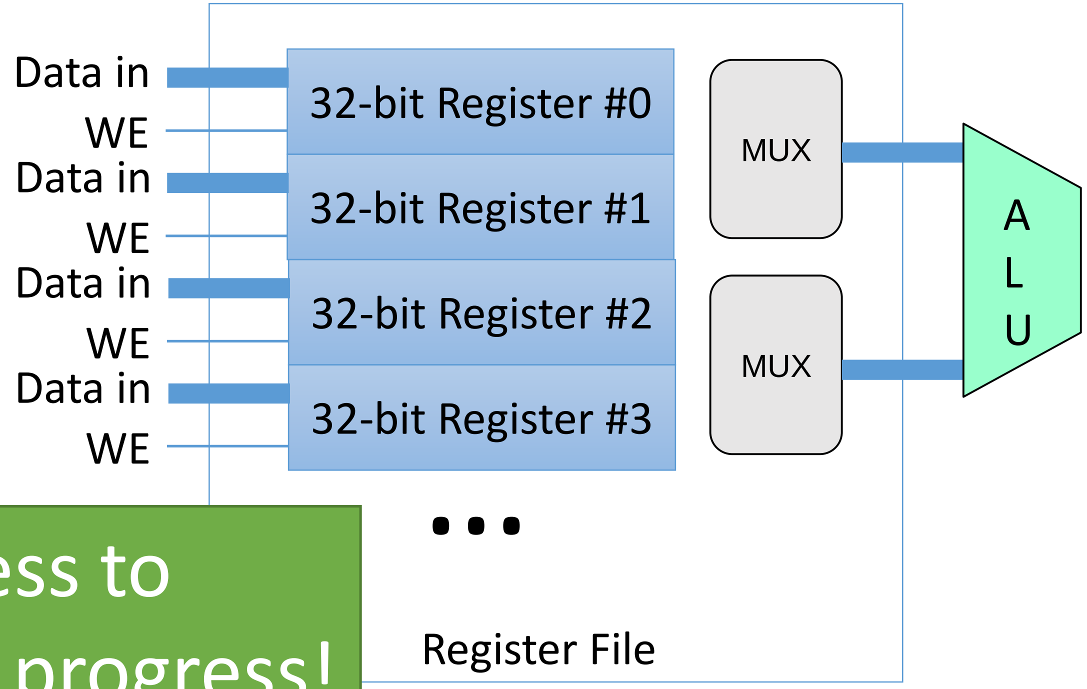
Process Resource: CPU Time

- CPU: Central Processing Unit

Program Counter (PC): **Memory address of next instr**

Instruction Register (IR): **Instruction contents (bits)**

- PC points to next instruction
- CPU loads instruction, decodes it, executes it, stores result
- Process "given" CPU by OS
  - Mechanism: context switch
  - Policy: CPU sche

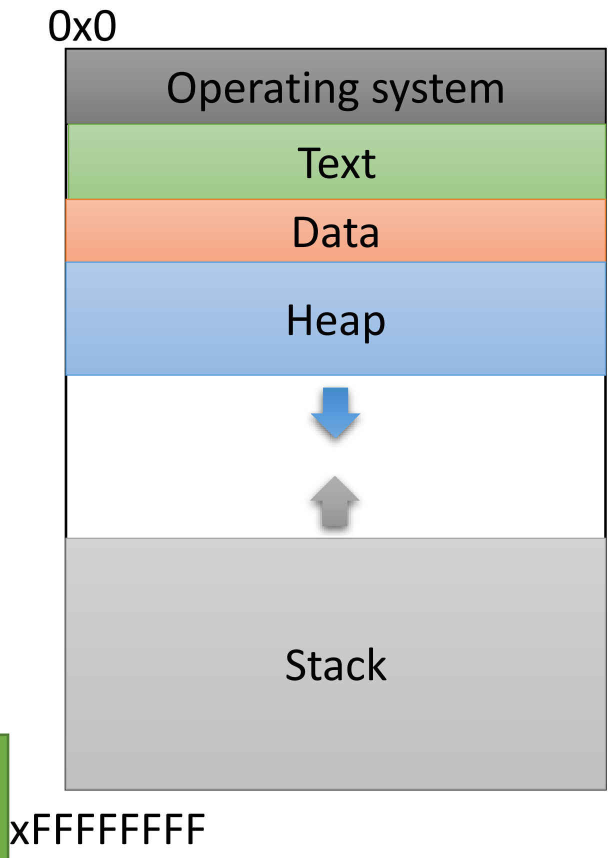


Required for process to execute and make progress!

Process Resource: Main Memory

- Process must store:
  - Text: code instructions
  - Data: static (known at compile time) variables
  - Heap: dynamically requested memory at runtime (malloc, new, etc.)
  - Stack: store local variables and compiler-generated function call state (e.g., saved registers)

Required for process to store instructions (+data)!



Process Resource: I/O

- Allows processes to interact with a variety of devices (i.e., everything that isn’t a CPU or main memory).

- Enables files, communication, human interaction, etc.

- Learn about or change the state of the outside world

Required?

Disk

etwork

Keyboard / Mouse

Reminder

1. Solo vote (quiet)
2. Small group discussion & group vote (loud)
3. Class discussion

Is I/O a requirement for processes?

A. Yes (why?)

B. No (why not?)

Same requirements for an Operating System?

- Previously, OS is: "System software that manages computer hardware and software resources and provides common services for computer programs."
- "OS" & "Kernel" - interchangeable in this course
- How does an OS / kernel fit in with this notion of processes?

Is the kernel a process? Should it be? Could it be?

A. Yes it is, and it should be.

B. Yes it is, but it shouldn’t be.

C. No it isn’t, but it should be.

D. No it isn’t, and it can’t be.

E. Something else

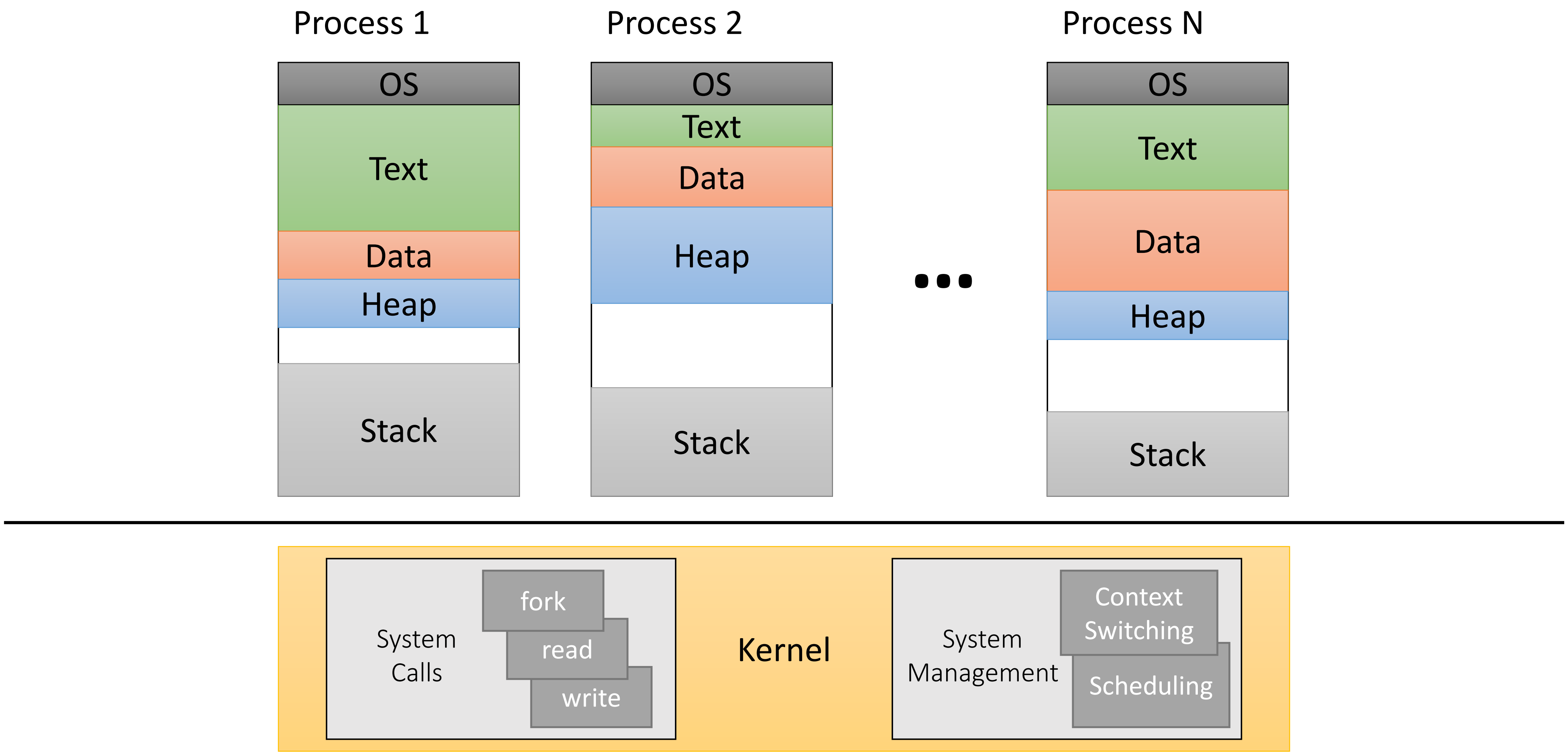
OS Kernel

- Many styles / ways to structure a kernel

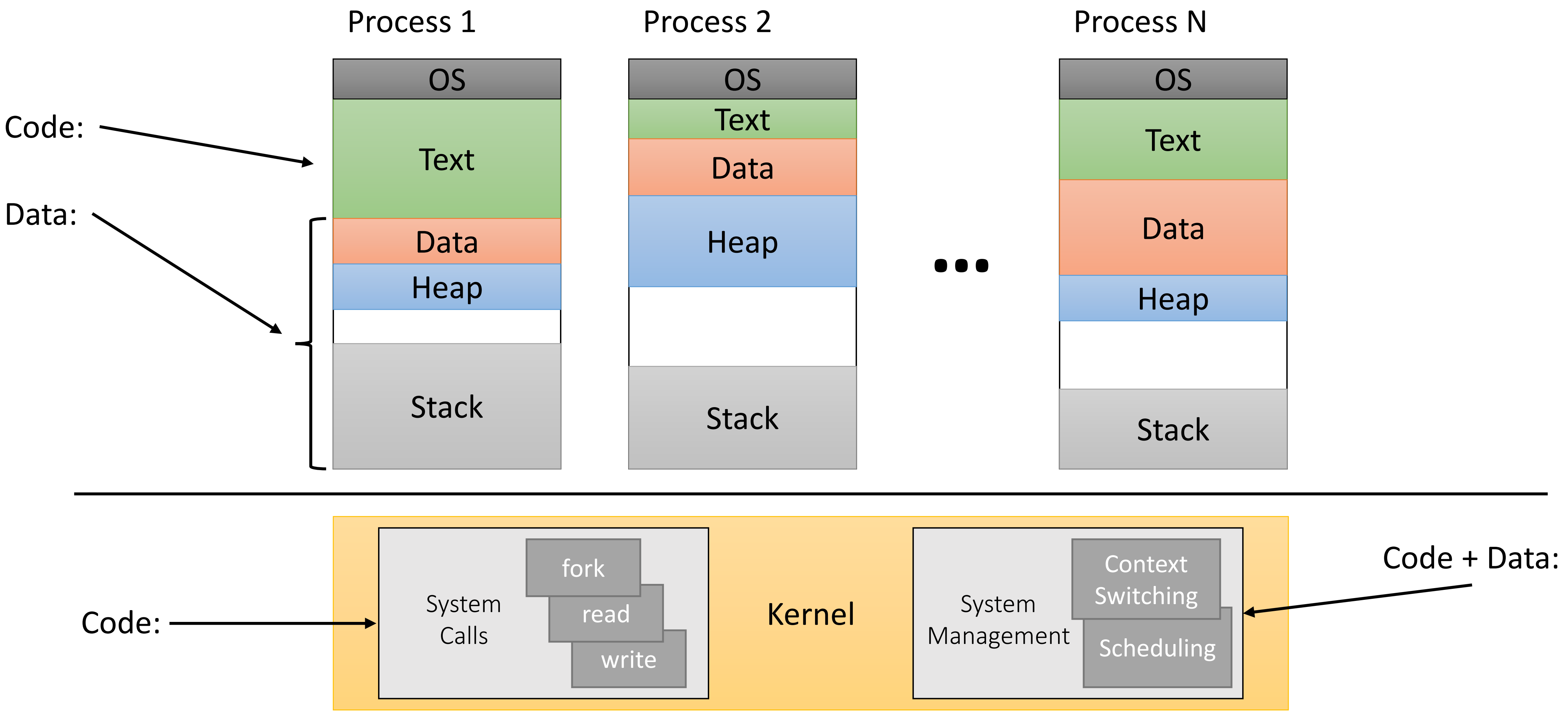
- Unless we say otherwise: assume the OS is <u>not</u> a process!
    - It's a special management entity – also implemented in software
    - It supports the user's processes, but is a special case with different needs

- The OS might create some processes to help itself out
    - e.g., Linux flushes buffered data to disks periodically
    - Other OS styles: kernel processes take a larger role, but still a "core" kernel

Kernel vs. Userspace: Model



Kernel vs. Userspace: Model



How/When should the OS Kernel’s code execute?

A. The kernel code is always executing.

B. The kernel code executes when a process asks it to.

C. The kernel code executes when the hardware needs it to.

D. The kernel code should execute as little as possible.

E. The kernel code executes at some other time(s).

Same Question, Different Resource

- “How much of the system’s memory should the OS use?”
- Hopefully not much… just enough to get its work done.
- Leave the rest for the user!

OS: Taking Control of the CPU

- The terminology here is, unfortunately, muddy.

"Trap"

1. System call – user requests service from the OS
2. Exception – user process has done something that requires help
3. (Hardware) interrupt – a device needs attention from the OS

System call often implemented as a special case of exception: execute intentional exception-generating instruction.

OS: Taking Control of the CPU

- The terminology here is, unfortunately, muddy.

"Trap"

1. System call – **user requests service from the OS**

2. Exception – user process has done something that requires help

3. (Hardware) interrupt – a device needs attention from the OS

System call often implemented as a special case of exception: execute intentional exception-generating instruction.

Why make system calls?

A. Performance: Kernel code executes faster / saves time.

B. Security: Programs can’t use kernel code or devices in unintended ways.

C. Usability: Kernel code is easier / adds value for programmers to use.

D. More than one of the above. (Which?)

E. Some other reason(s).

Common Functionality

- Some functions useful to many programs, some need to be protected
  - I/O device control
  - Memory allocation
- Place these functions in kernel
  - Called by programs (system calls)
  - Or accessed implicitly as needed (exceptions)
- What should these functions be?
  - How many programs should benefit?
  - Might kernel get too big?

How about a function like `printf()`?

A. `printf()` is a system call (why?)

B. `printf()` is not a system call (why not, what is it?)

- Some functions useful to many programs
  - I/O device control
  - Memory allocation
- Place these functions in kernel
  - Called by programs (system calls)
  - Or accessed implicitly as needed (exceptions)
- What should these functions be?
  - How many programs should benefit?
  - Might kernel get too big?

System Calls in Practice

- Often hidden from user by libraries (e.g., libc) for convenience
  - printf: performs a write() system call, but handles variable-length arguments
  - “raw” syscall does as little as possible. write(): move (already formatted) data

- How can you tell if a function is a syscall or belongs to a library?
  - Man page section number: 2 – syscall, 3 – library
  - Follow the trail of included header files

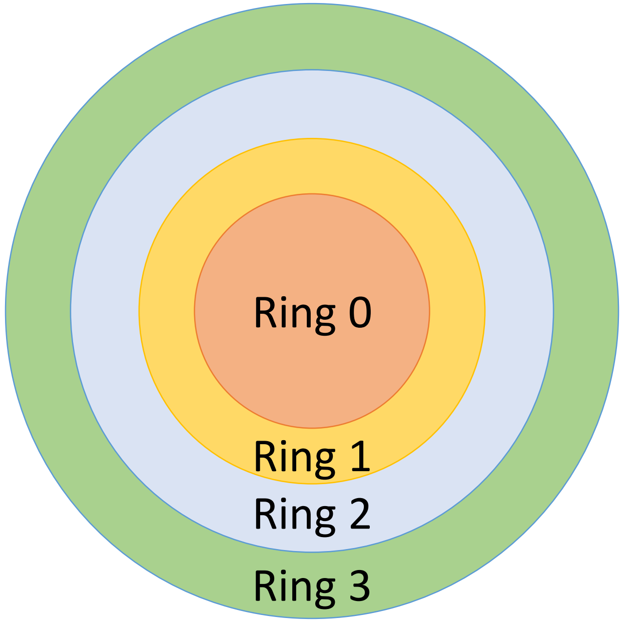
```
READ(2)        Linux Programmer's Manual        READ(2)

NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);
```

```
FREAD(3)                  Linux Programmer's Manual                  FREAD(3)

NAME
       fread, fwrite - binary stream input/output

SYNOPSIS
       #include <stdio.h>

       size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Syscall Protection Features

- Small syscalls: minimize attack “surface area” in trusted kernel code.

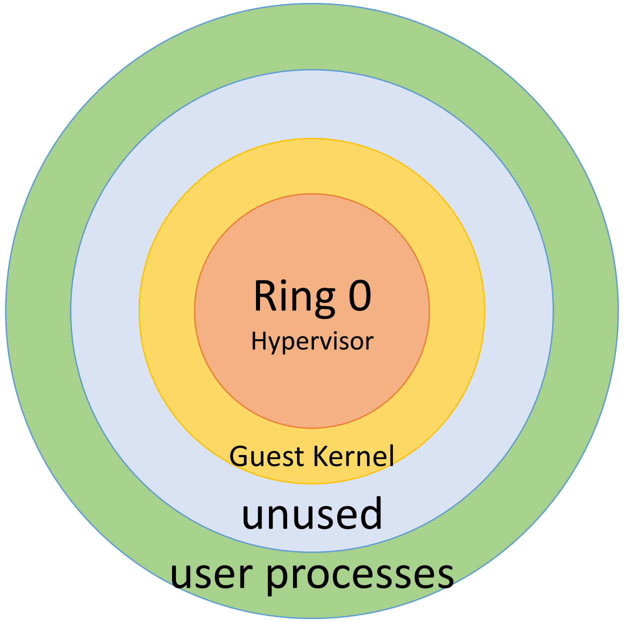
- Hardware mode: x86 / amd64 “rings”



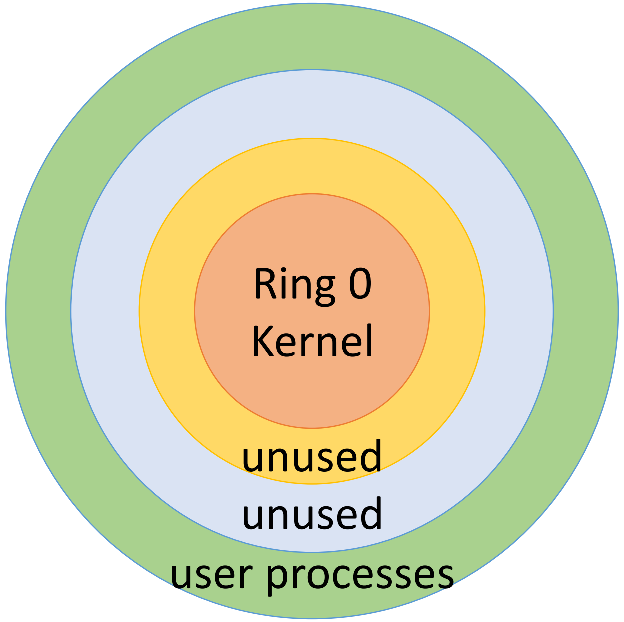
Syscall Protection Features

- Small syscalls: minimize attack “surface area” in trusted kernel code.

- Hardware mode: x86 / amd64 “rings”



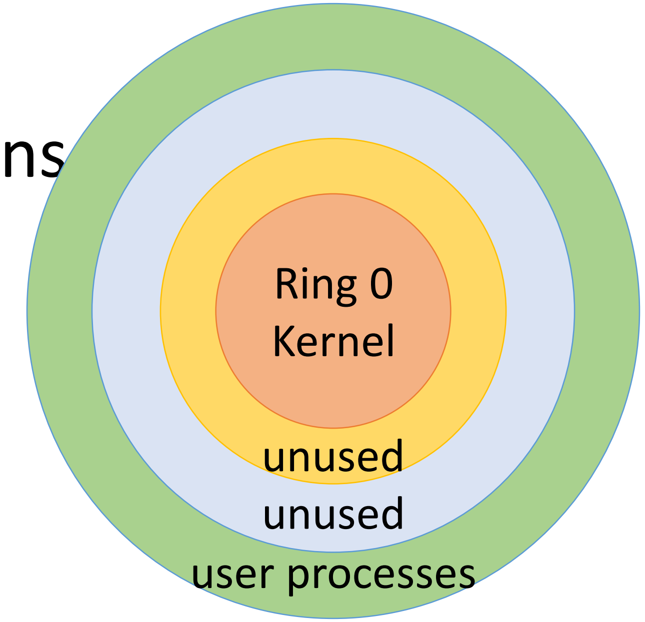
Virtualization



Traditional (Unix, Windows)

Syscall Protection Features

- Small syscalls: minimize attack "surface area" in trusted kernel code.
- Hardware mode: x86 / amd64 "rings"
- Lower numbered rings, more privileged instructions
- Well-defined syscall entry points
  - "amplify" power, switch mode to ring 0



Traditional (Unix, Windows)

Syscall Entry vs. Userspace Function Call

- syscall behavior is different from userspace code, where to execute a new function we just specify which instruction to jump to.

Userspace instructions (from CS 31):

`pushl`	Create space on the stack and place the source there.	`subl $4, %esp` `movl src, (%esp)`
`popl`	Remove the top item off the stack and store it at the destination.	`movl (%esp), dst` `addl $4, %esp`
`call`	1. Push return address on stack 2. Jump to start of function	`push %eip` `jmp target`
`leave`	Prepare the stack for return (restoring caller's stack frame)	`movl %ebp, %esp` `popl %ebp`
`ret`	Return to the caller, PC ← saved PC (pop return address off the stack into PC (eip))	`popl %eip`

Syscall Entry vs. Userspace Function Call

- **Takeaway: the cost of making a function call and returning in userspace isn't that big – just a few instructions.**

Userspace instructions (from CS 31):

`pushl`	Create space on the stack and place the source there.	`subl $4, %esp` `movl src, (%esp)`
`popl`	Remove the top item off the stack and store it at the destination.	`movl (%esp), dst` `addl $4, %esp`
`call`	1. Push return address on stack 2. Jump to start of function	`push %eip` `jmp target`
`leave`	Prepare the stack for return (restoring caller's stack frame)	`movl %ebp, %esp` `popl %ebp`
`ret`	Return to the caller, PC ← saved PC (pop return address off the stack into PC (eip))	`popl %eip`

Syscall Entry Points

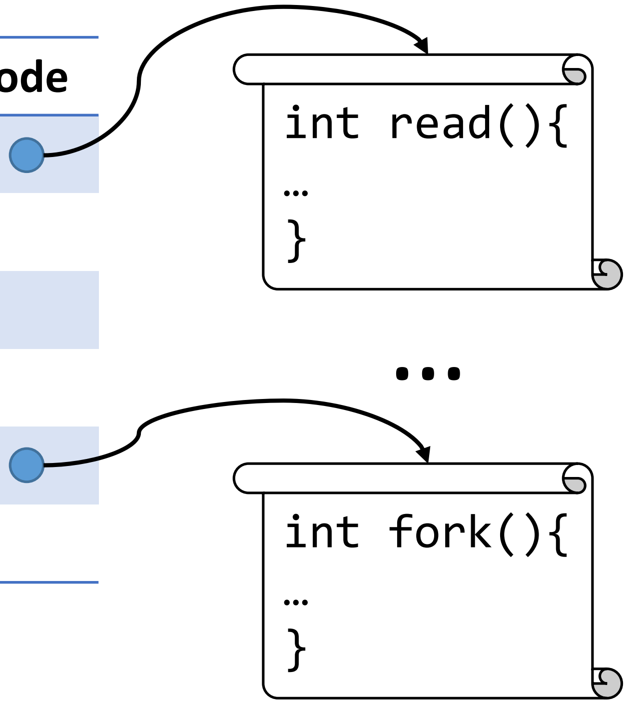
- Switching into the kernel means we guarantee kernel code will start running at a fixed point in the code – the beginning of a function.

- Guarantees we will run an entire function, not just some part of it (your userspace process is no longer in control of the CPU).

Making a System Call

- Each system call has a unique number. OS keeps a table.

Syscall Number	Syscall Name	Code
0	read	
1	write	
2	open	
…		
57	fork	
…		



Making a System Call

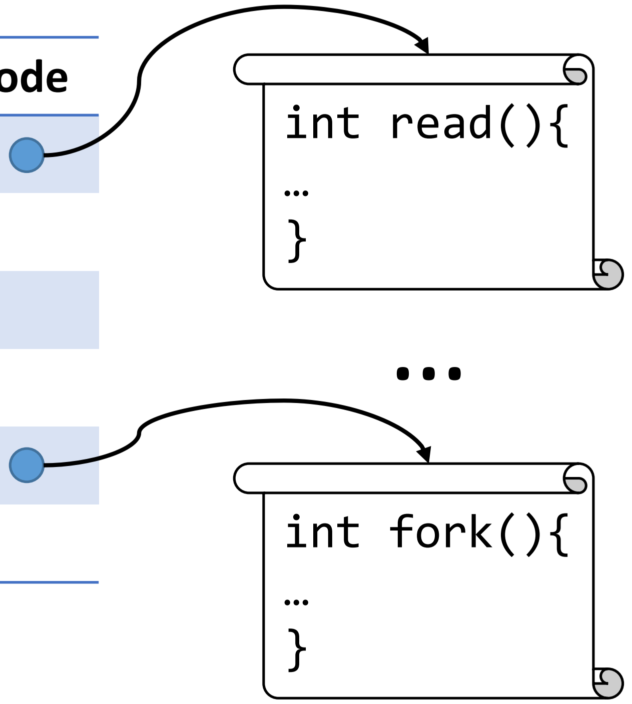
- Each system call has a unique number. OS keeps a table.

To make a system call:

1. place desired syscall number in the agreed-upon location (e.g., register).

2. initiate system call (special instruction – often intentional exception).

Syscall Number	Syscall Name	Code
0	read	
1	write	
2	open	
…		
57	fork	
…		



System Call Cost

- Compared to a normal userspace function call, cost is relatively high.
- Worth the cost to processes to get access to protected resources.
- Programmer should be careful not to make too many syscalls in performance-critical sections of code.

Structure of a Kernel

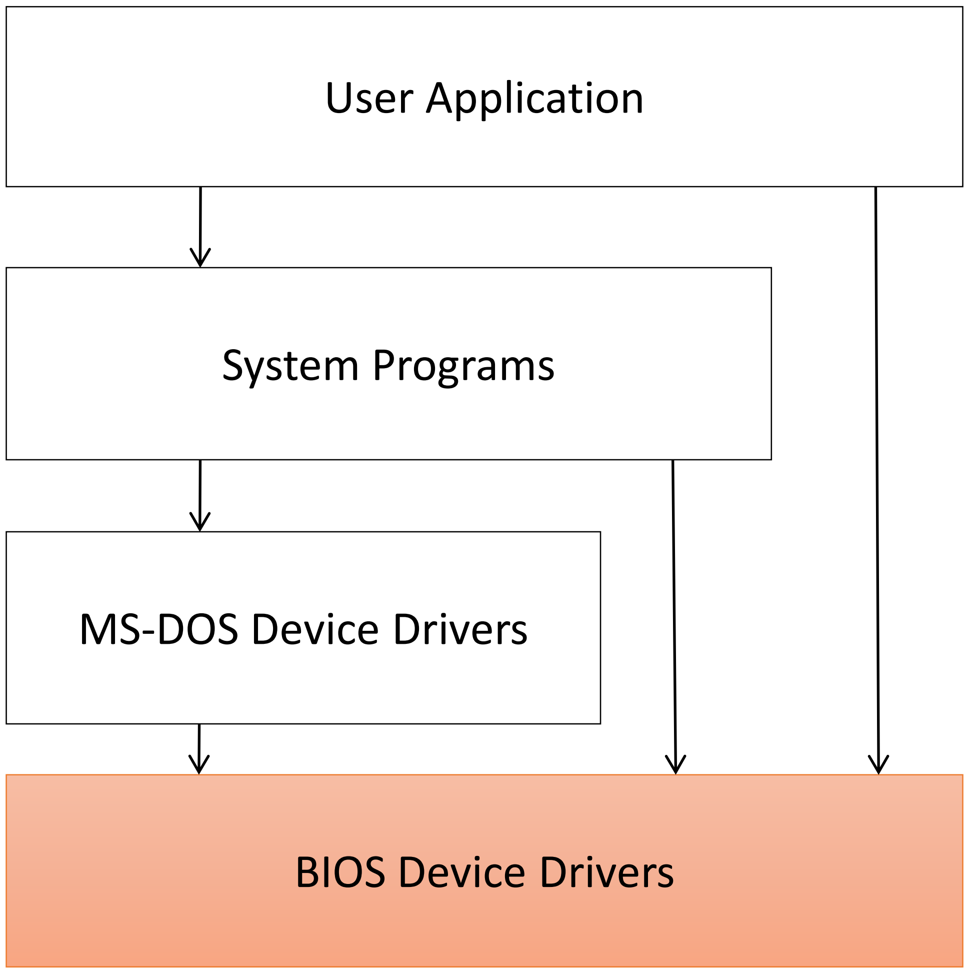
- Simple (MS-DOS, early UNIX)
- Monolithic + Modules (Linux, Windows 9x)
- Microkernel (Mach)
- Hybrid (Windows NT, XNU/OS X)

Structure of a Kernel

- Simple (MS-DOS, early UNIX)
- Microkernel (Mach)
- Hybrid (Windows NT, XNU/OS X)

There is no one-size-fits-all solution!

Simple (MS-DOS)



What’s problematic about this simple model?

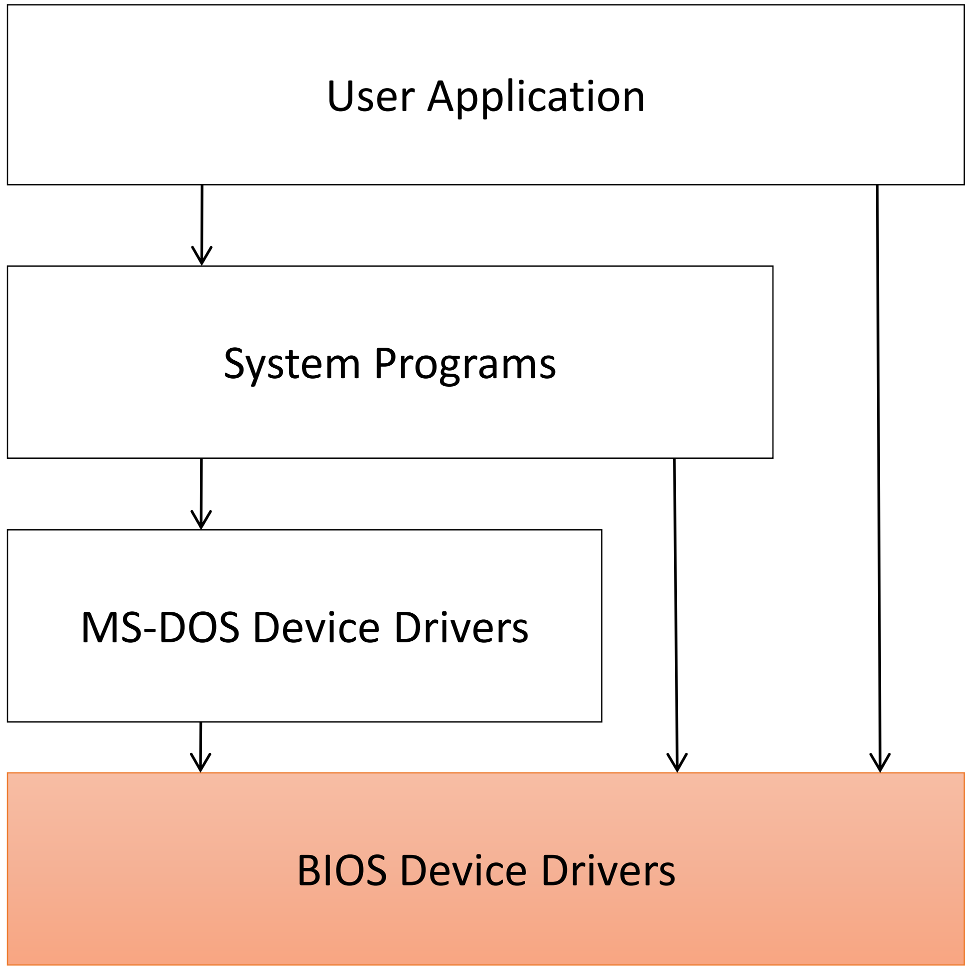
A. Insecure

B. Inefficient

C. Hard to add functionality

D. More than one of the above

E. Something else



What's problematic about this simple model?

A. Insecure

B. Inefficient

C. Hard to add functionality

D. More than one of the above

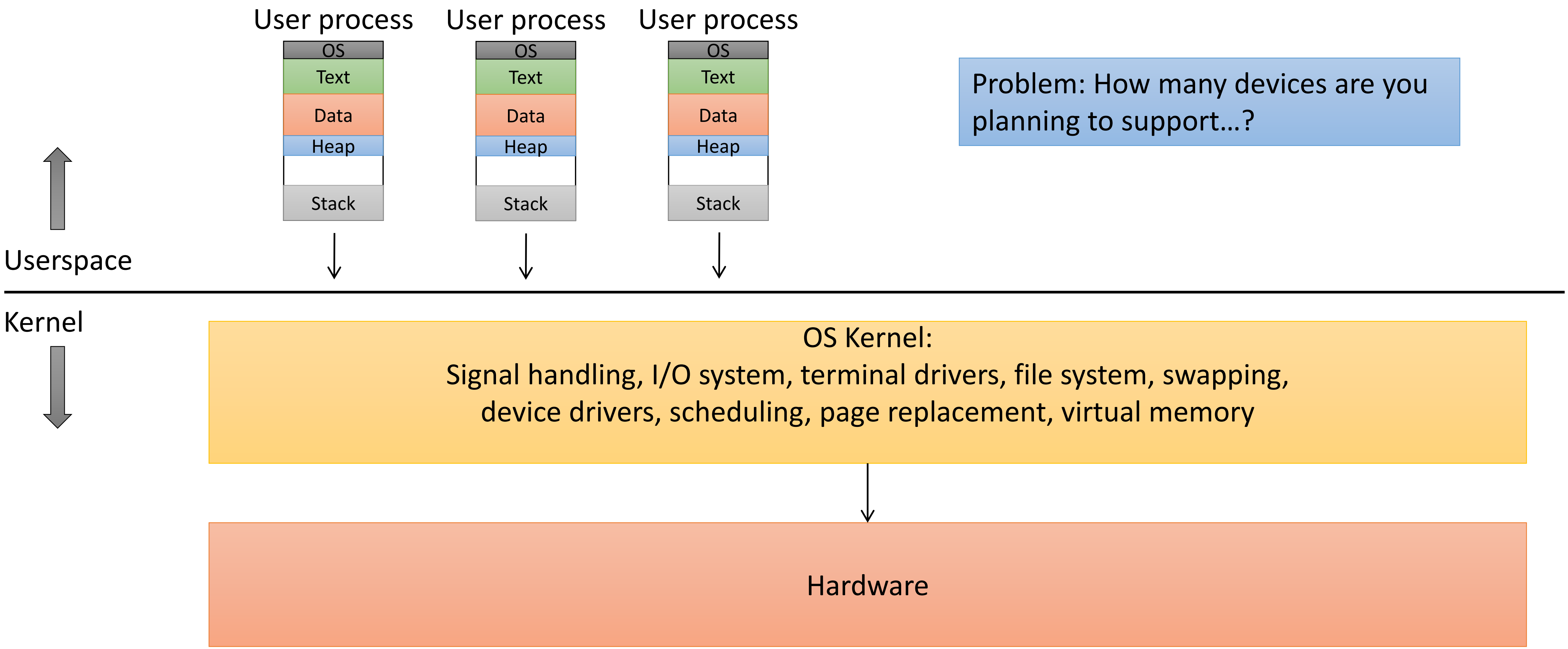
E. Something else

Solution: add the protection features we talked about earlier (or something similar)!

Most importantly: Limit user's entry into important stuff.

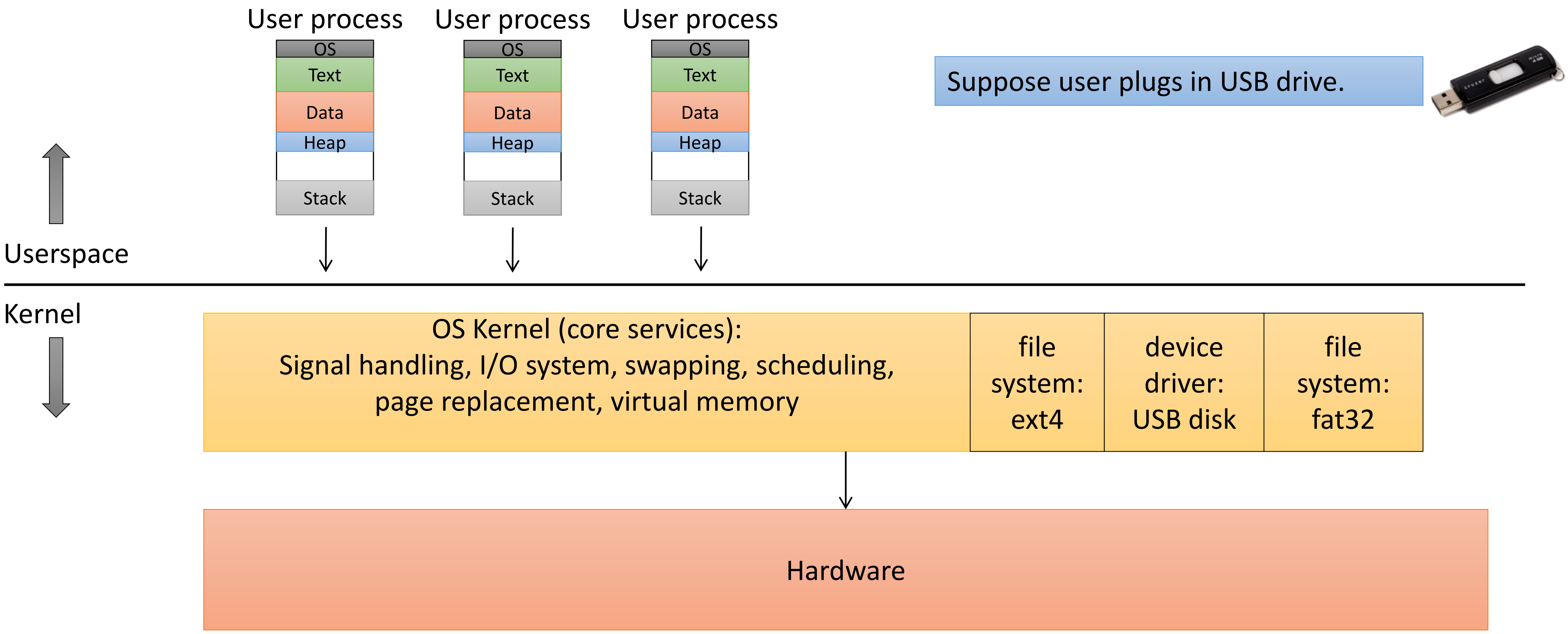
But…where should the important stuff go?

Monolithic – without modules



Problem: How many devices are you planning to support…?

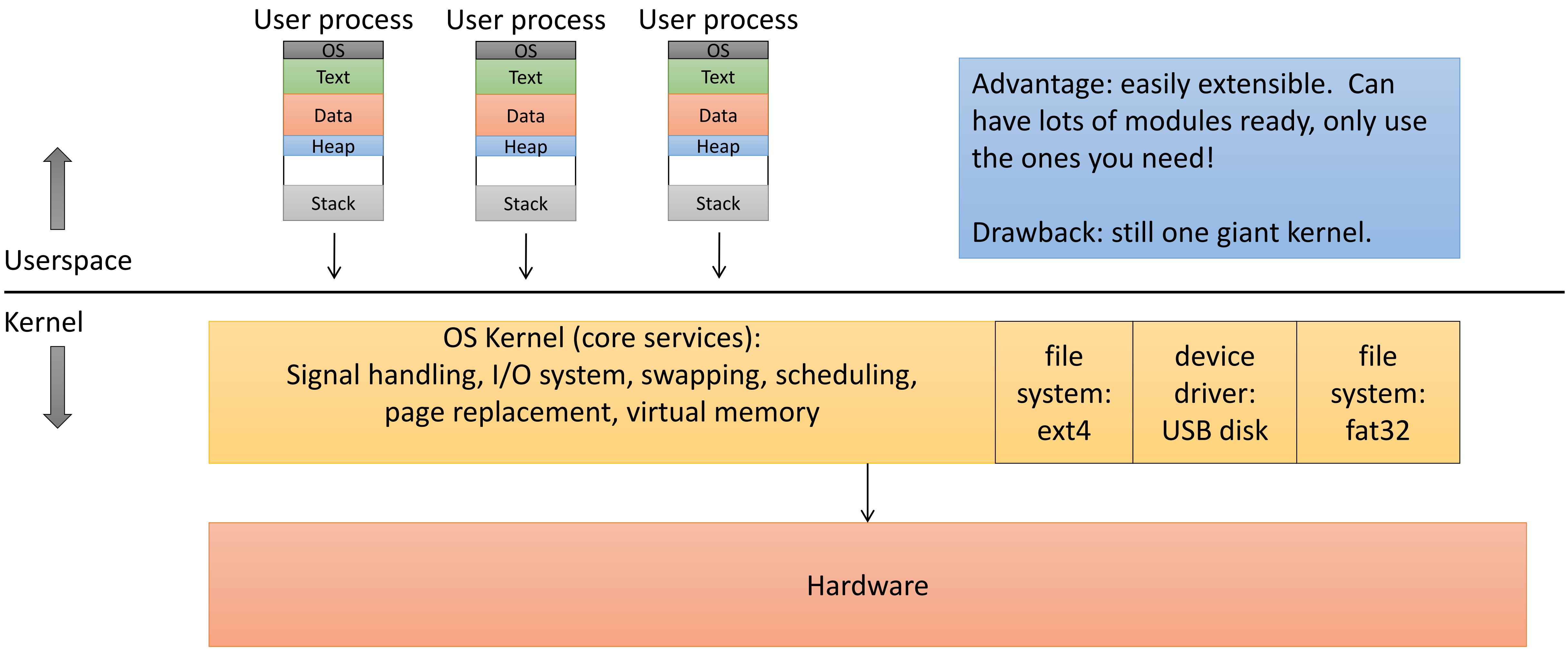
Modular Monolithic (Linux)



Modular Monolithic (Linux)

Advantage: easily extensible. Can have lots of modules ready, only use the ones you need!

Drawback: still one giant kernel.



What's problematic about the modular monolithic model?

A. Insecure

B. Inefficient

C. Hard to add functionality

D. More than one of the above

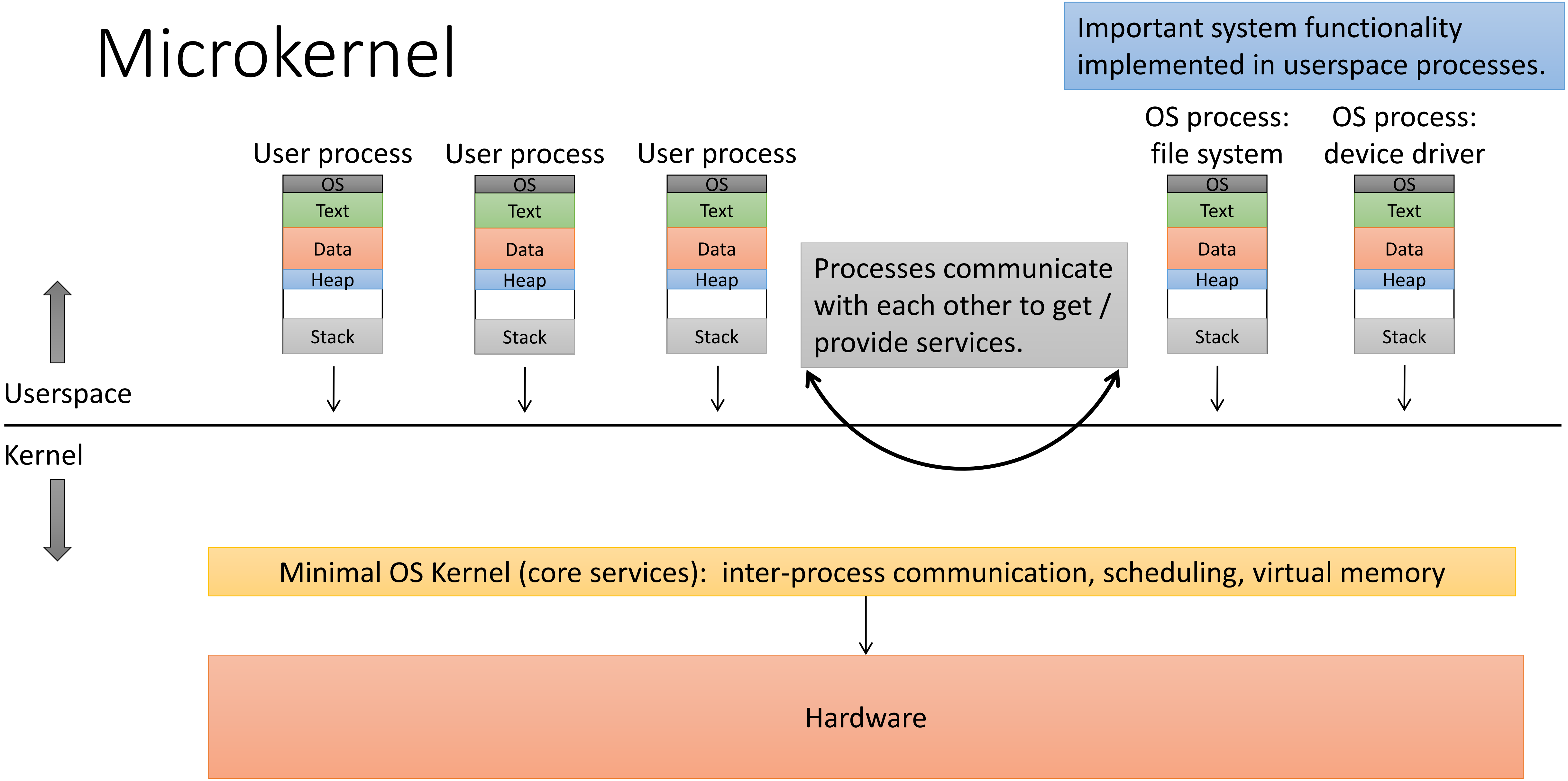
E. Something else

Microkernel

- Kernel supports as little as possible:
  - message-passing (communication between processes)
  - process / “task” management
  - memory allocation

- All other functionality delegated to user level processes

Microkernel



Microkernel

- Kernel supports as little as possible:
  - message-passing (communication between processes)
  - process / “task” management
  - memory allocation

- All other functionality delegated to user level processes

- Benefits: Strong isolation between services, less trusted kernel code.

What's problematic about microkernels?

A. Insecure

B. Inefficient

C. Hard to add functionality

D. More than one of the above

E. Something else

Problem: LOTS of transitioning between userspace and the kernel.

We'll see: not a trivial operation…

Of the choices we've seen so far, which do you like best / would you choose if you built an OS? Why?

A. Simple

See:
https://en.wikipedia.org/wiki/Tanenbaum%E2%80%93Torvalds_debate

B. Monolithic

C. Monolithic + modules

D. Microkernel

E. Something else (?)

Hybrid Kernels

- NT Kernel (Used in modern Windows)
  - Divided into modules
  - Modules communicate via function calls or messaging
  - Almost all modules run in kernel mode
  - Some application system services run in user mode

- Graphics example:
  - Graphics driver moved around a couple of times
  - Initially -> Userspace process for isolation
  - Later -> back to kernel for performance reasons

Hybrid Kernels

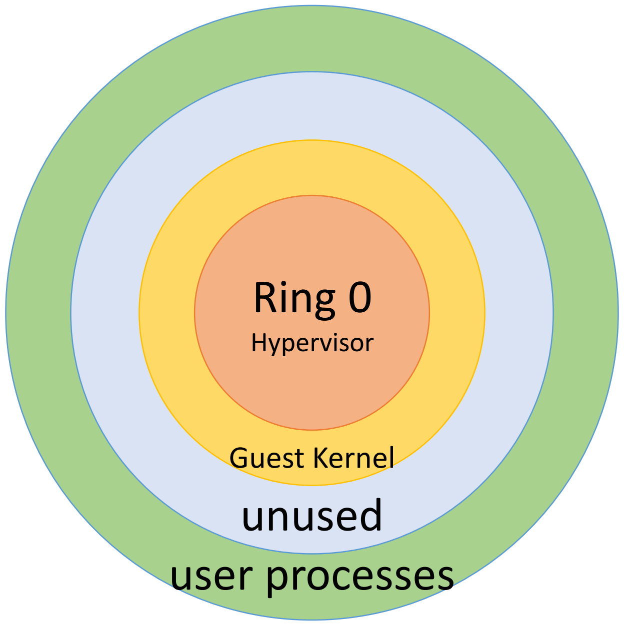
- XNU (OS X)
  - Combines Mach (classic microkernel) with BSD
  - Runs core Mach kernel, but with BSD subsystems and APIs added
  - Mach communicates with BSD via IPC, but everything is running in kernel mode

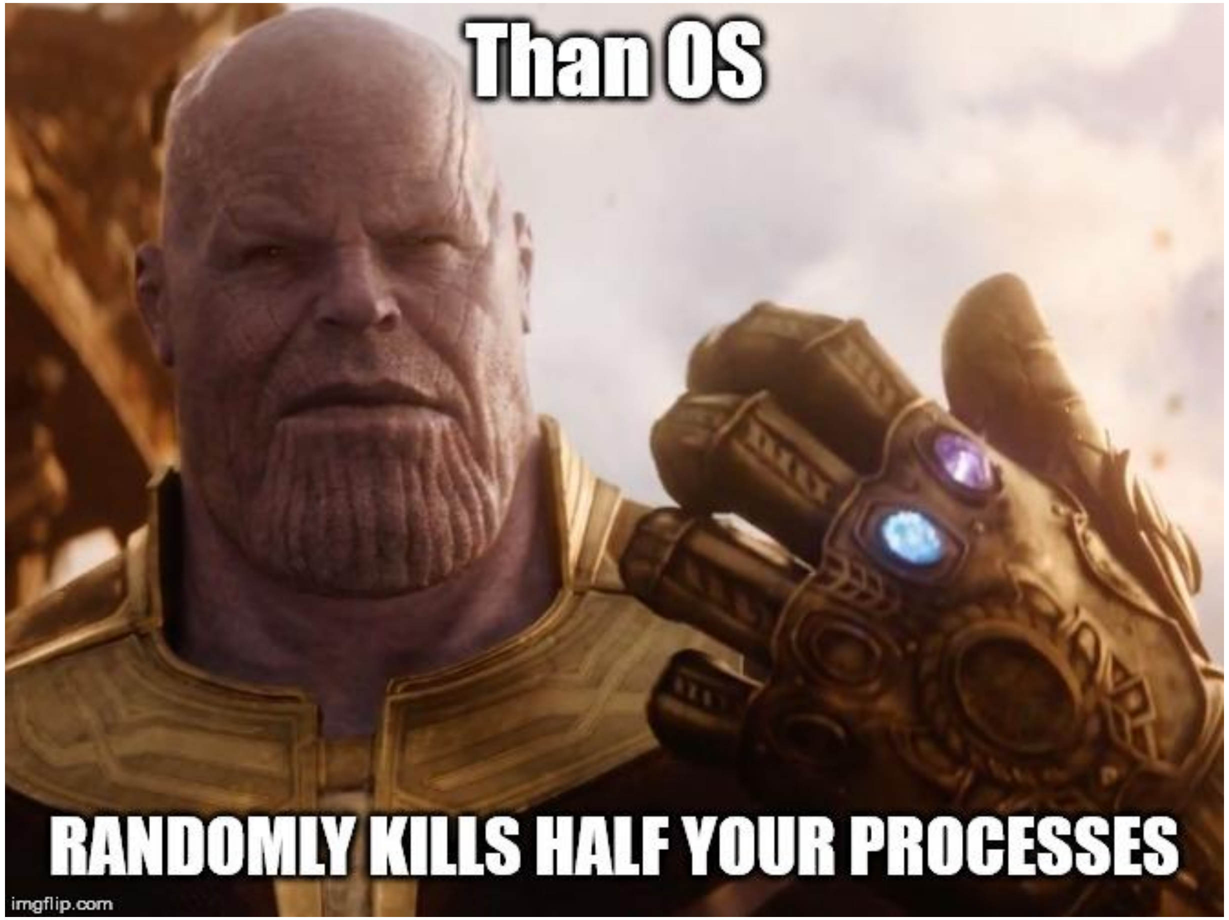
Recent OS Research: Arrakis (2014)

- "Applications have direct access to virtualized I/O devices, allowing most I/O operations to skip the kernel entirely . . ."

- Takes advantage of hardware support for virtualizing I/O devices (meant for performance speedups in virtual machines) in order to guarantee protection

Virtualization, more generally




Than OS
RANDOMLY KILLS HALF YOUR PROCESSES
imgflip.com

Summary

- Important distinction: userspace vs. the OS kernel
- We don't *want* the OS using resources, but it has to when it gets a system call, exception, or hardware interrupt.
- Transition to kernel amplifies power, allows privileged instructions
- Many patterns for structuring a kernel, each has merits and drawbacks
  - monolithic, microkernel, hybrid