Duke Systems

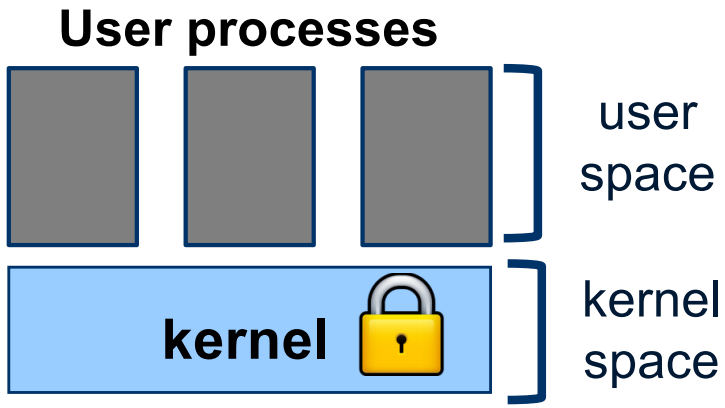
CPS 310
Processes, Threads, and the Kernel

Jeff Chase

Duke University

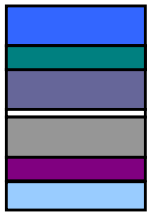
The kernel

- The operating system kernel!
  - What is it?
  - Where is it?
  - How do we get there?
  - How is it protected?
  - How does it control resources?
  - How does it control access to data?
  - How does it keep control?



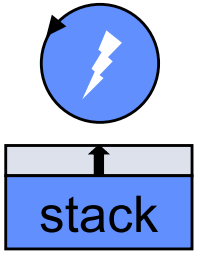
A process and its threads

virtual address space



+

main thread



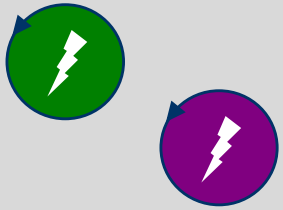
+...

Each process has a virtual address space (VAS): a private name space for the virtual memory it uses.

Each process has a **main thread** bound to the VAS, with a stack.

The **scheduler** (in kernel) can suspend/resume threads wherever and whenever it wants.

other threads (optional)



On modern systems they are all visible to the kernel.

They can all make system calls and enter the kernel independently.

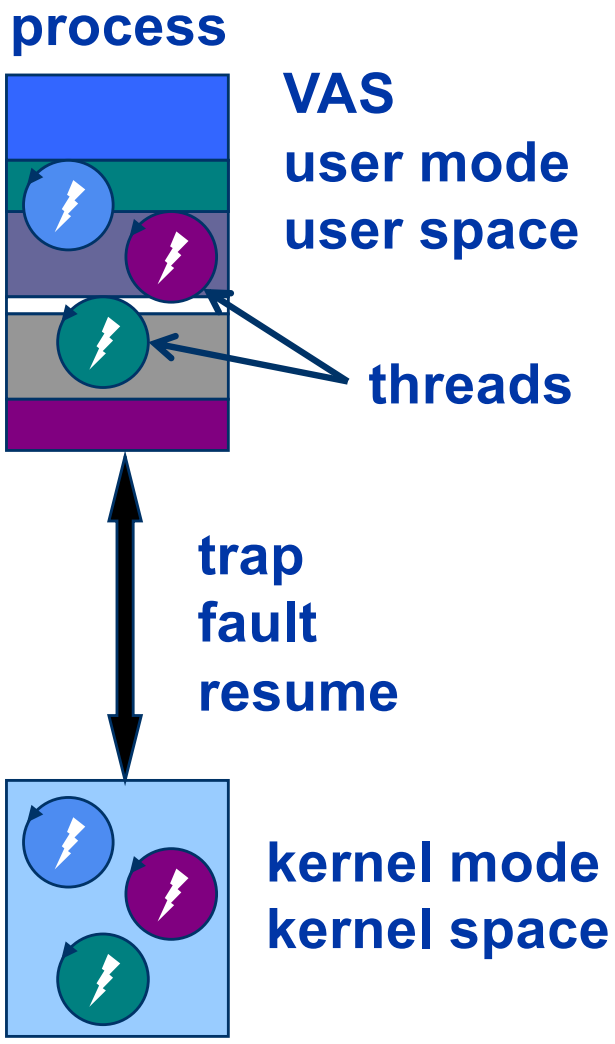
Process vs. thread: what's the difference?

- In classical Unix, each process (each VAS) has exactly one thread—it predates any distinction.
- Linux people say that threads are "just processes" that happen to share an address space, file descriptors, and a program—they like to blur the distinction.
- Sometimes it is useful to speak loosely when we emphasize what is common (e.g., context switch).
- I want you to think in terms of the fundamental kernel-supported abstractions: threads and address spaces.
- If we say a process does something, we really mean a thread within the process does it.

Threads and the kernel

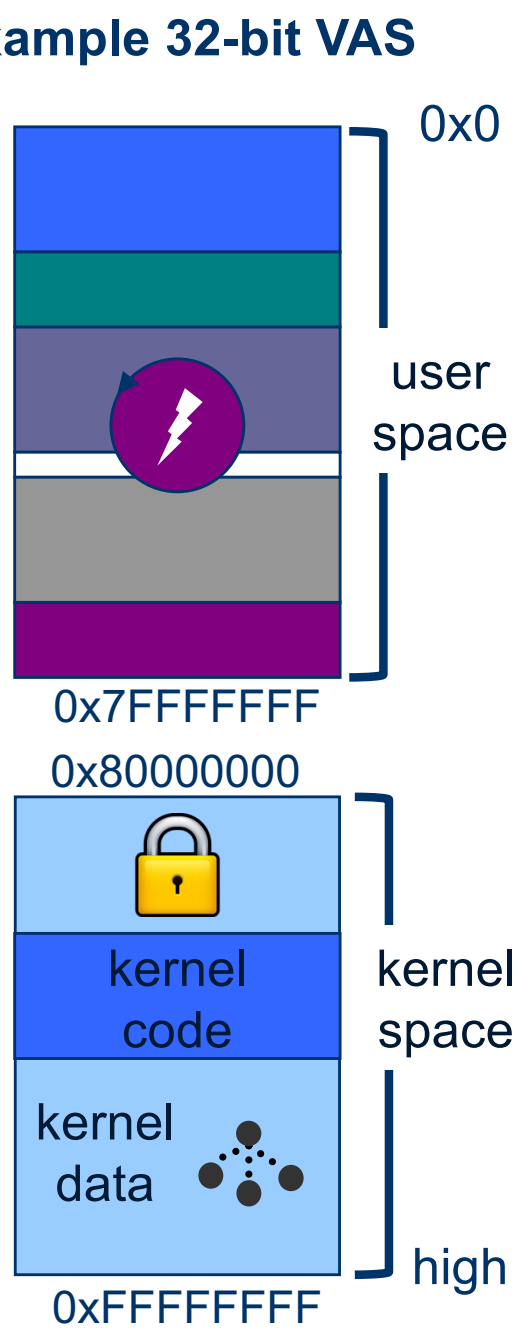
- **Modern operating systems have multi-threaded processes.**
- Kernel API has syscalls to create threads
  - e.g., Linux **clone**, used by pthreads library
- Threads may enter the kernel (e.g., for syscalls) and run concurrently in the kernel.
- This model (1x1) applies to Linux, MacOS, Windows, Android, and pthreads or Java on those systems.

In labs we implement Nx1 threads (**user-level** or **green** threads) in which the main thread calls library routines for switch/swap contexts. No true parallelism! We use it to emulate the kernel environment.

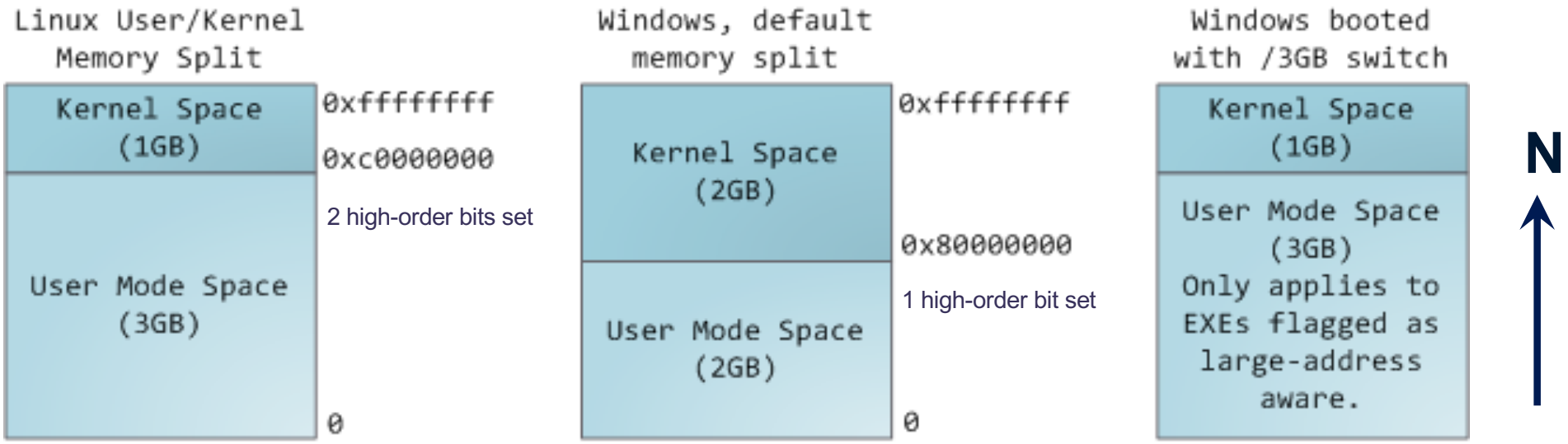


The kernel

- The kernel is just a program: a collection of modules and their state.
- E.g., it may be written in C and compiled/linked a little differently.
  - E.g., linked with –static option: no dynamic libs
- At runtime, kernel code and data reside in a protected range of virtual addresses.
  - VPN->PFN translations for kernel space are **global**.
    - (Details vary by machine and OS configuration)
  - Access to kernel space is denied for user programs.
  - Portions of kernel space may be non-pageable and/or direct-mapped to machine memory.



Where is kernel space?



- Classically, the kernel is mapped in every process/VAS.
  - At least on Intel/AMD and most other machines. The details of the mapping are OS-dependent.
- Recent OS unmap the kernel in user mode to defend against the Meltdown vulnerability.

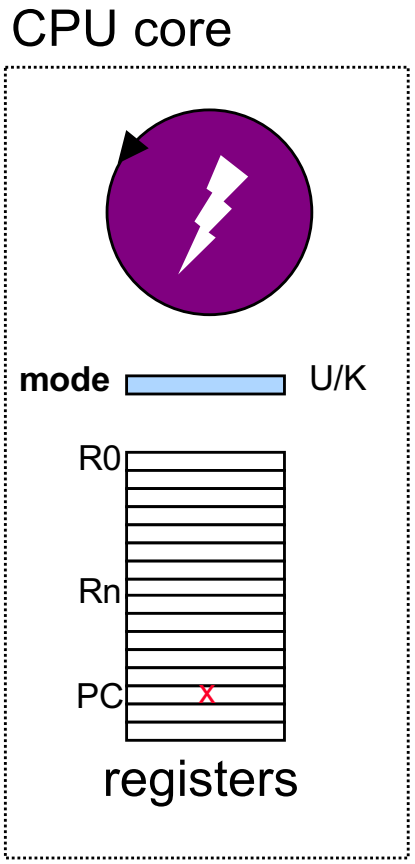
CPU mode: user and kernel

The current **mode** of a CPU core is represented by a field in a protected register.

We consider only two possible values: **user mode** or **kernel mode** (also called **protected mode** or **supervisor mode**).

If the core is in **protected mode** then it can:

- access kernel space
- access certain **control registers**
- execute certain special instructions



If software attempts to do any of these things when the core is in user mode, then the core raises a CPU exception (a **fault**).

Kernel space example
x86-64 Intel Core i7 PTE

Protection bits

XD: execute disable: 1 -> no instruction fetch from this page

R/W: read-write privilege? or read-only?

U/S: accessible from user code? Or only by OS (supervisor)?

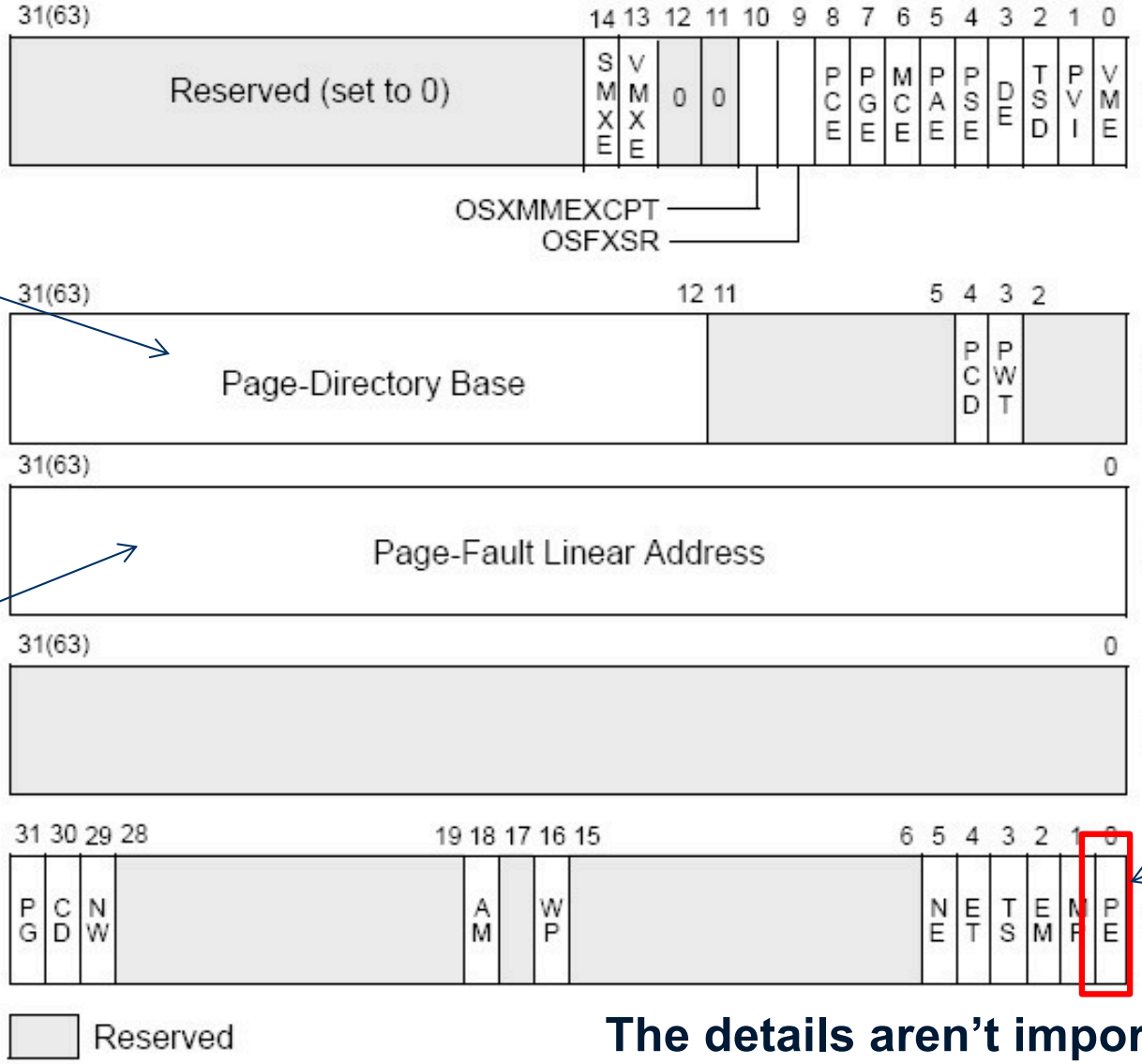
This is an **example** of how to map portions of the VAS as accessible only in protected kernel (supervisor) mode. These portions constitute **kernel space**. If an instruction addresses kernel space while the core is in user mode, then it raises a protection fault.

[from CS:APP3e F9.24: edited and not to scale]

Example: x86 (32-bit/IA32) Privileged control registers

Points to page table (in machine memory) for current VAS active on this core

Faulting VA (if page fault is active)



Protected mode **E**nable

The details aren't important.

See [en.wikipedia.org/wiki/Control_register]

Kernel mode

What turns it on? **Exceptions**

- **Trap**: system call from user code
- **Fault**: CPU needs kernel help
- **Interrupt**: device needs attention

What turns it off?

- Kernel code executes special instruction to enter user mode.
- Kernel decides where to enter.
- E.g., return to saved user context.

Exceptions and interrupts
"trap, fault, interrupt"

	intentional happens every time	**unintentional** contributing factors
synchronous caused by an instruction	**trap: system call** open, close, read, write, fork, exec, exit, wait, kill, etc.	**fault** invalid or protected address or opcode, page fault, overflow, etc.
asynchronous caused by some other event	"software interrupt" software requests an interrupt to be delivered at a later time	**interrupt** caused by an external event: I/O op completed, clock tick, power fail, etc.

Terminology notes: some sources distinguish exceptions and interrupts, and some don't. Some sources (e.g., xv6) use "trap" to refer to both traps and faults. Some sources (e.g., old Intel) use "software interrupt" to refer to a trap.

Hear the fans blow

```
int
main()
{
    while(1);
}
```

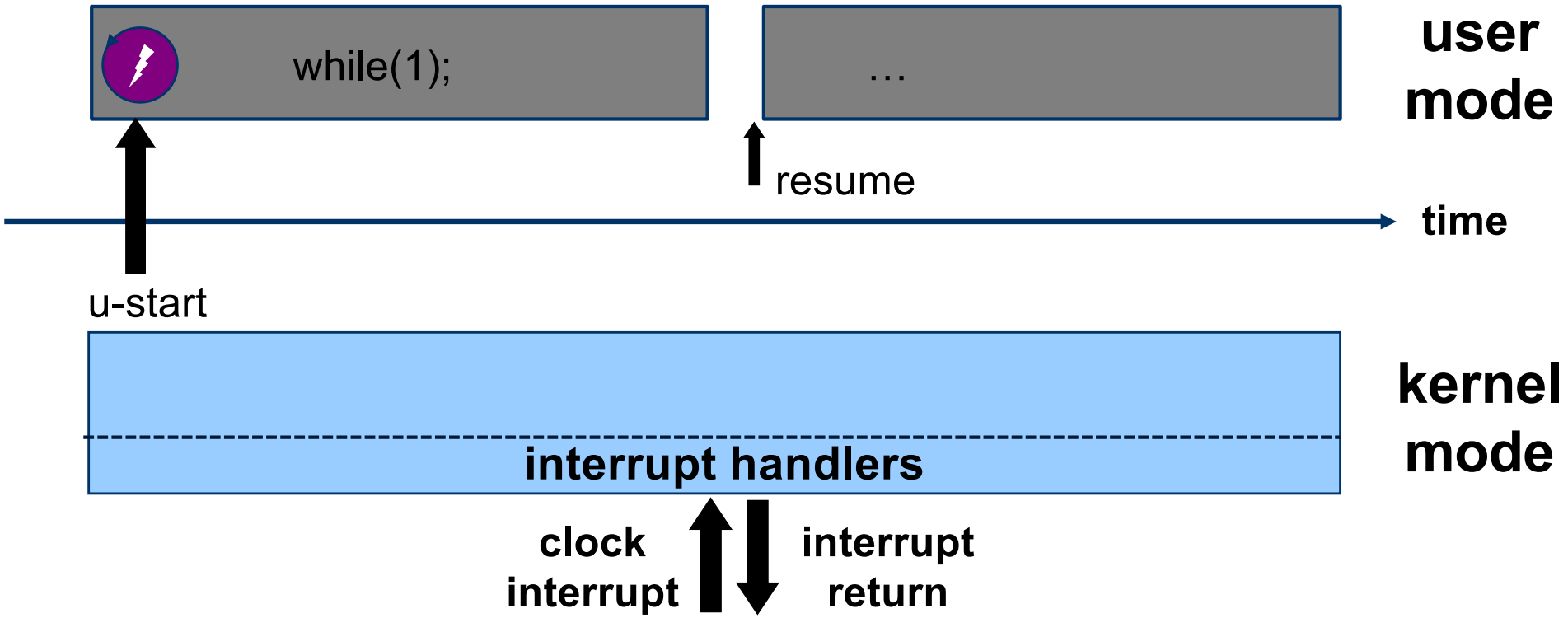
How does the OS regain control of the core from this program?

No system calls! No faults!

How to give someone else a chance to run?

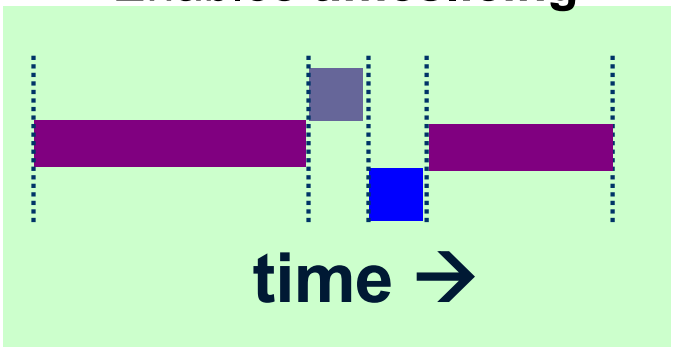
How to “make” processes share machine resources fairly?

Timer interrupts



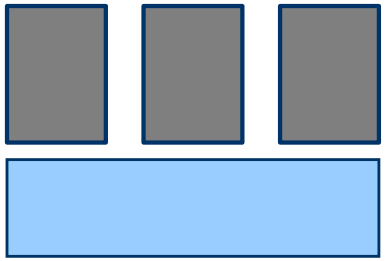
The **system clock** (timer) interrupts periodically, giving control back to the kernel. The kernel can do whatever it wants, e.g., switch threads.

Enables **timeslicing**

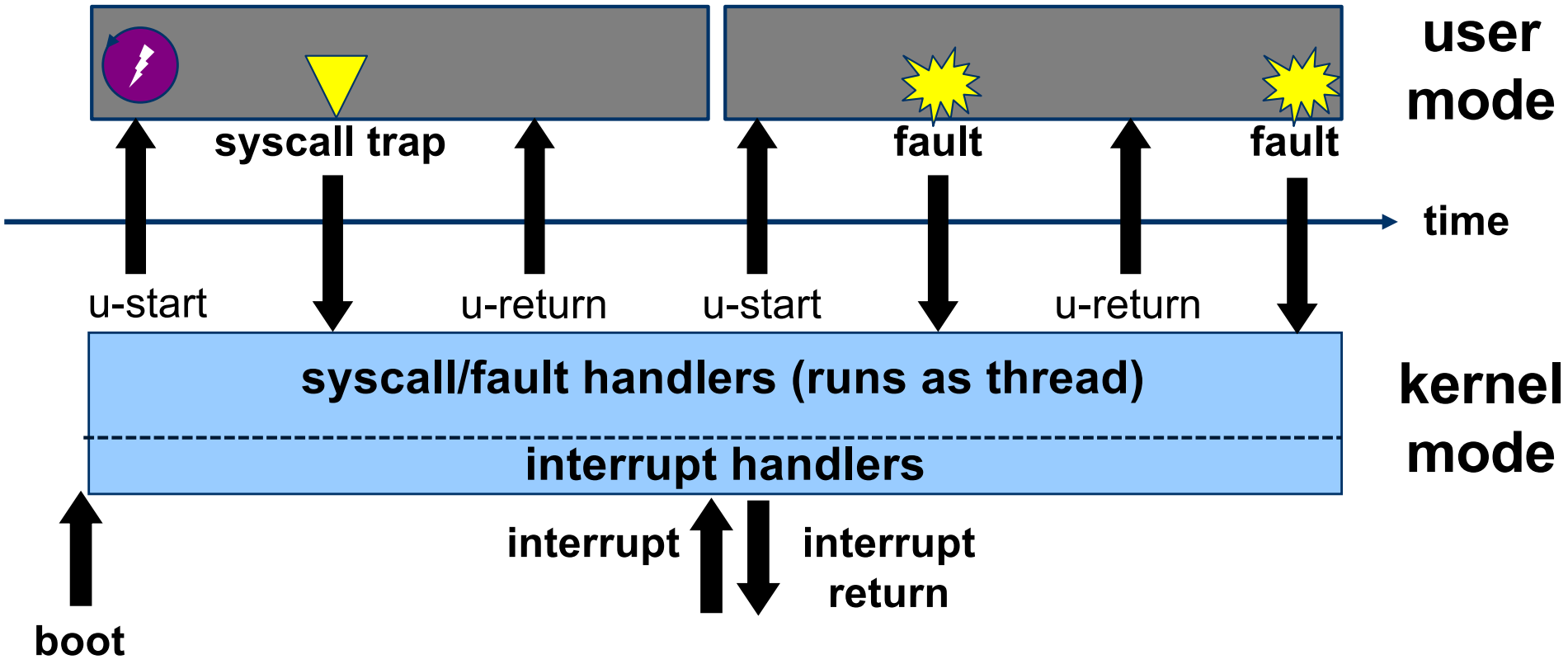


How the kernel gains control

- When processor core is running a user program, the user program/thread controls ("drives") the core.
- The hardware has a **timer** device that **interrupts** the core after a specified interval of time.
- Interrupt transfers control back to the OS kernel, which may switch the core to another thread, or resume.
- Other events also return control to the kernel.
  - Wild pointers
  - Divide by zero
  - Other program actions
  - Page faults



“Limited direct execution”



User code runs on a CPU core in user mode in a user space. If it tries to do anything weird, the core transitions to the kernel, which takes over.

The kernel executes a special instruction to **transition to user mode** (labeled as “u-return”), with selected values in CPU registers.

Syscalls/traps

- Programs in C, C++, etc. invoke system calls by linking to a standard library (libc) written in assembly.
  - The library defines a stub or wrapper routine for each syscall.
  - Stub executes a special trap instruction (e.g., chmk or callsys or syscall/sysenter instruction) to change mode to kernel.
  - Syscall arguments/results are passed in registers (or user stack).
  - OS+machine defines Application Binary Interface (ABI).

read() in Unix libc.a Alpha library (**executes in user mode**):

```
#define SYSCALL_READ 27          # op ID for a read system call

        move arg0…argn, a0…an    # syscall args in registers A0..AN
        move SYSCALL_READ, v0    # syscall dispatch index in V0
        callsys                  # kernel trap
        move r1, _errno          # errno = return status
        return
```

Example **read syscall stub** for Alpha CPU ISA (defunct)

MacOS x86-64 syscall example

```
section .data
hello_world     db      "Hello World!", 0x0a

section .text
global start

start:
mov rax, 0x2000004      ; System call write = 4
mov rdi, 1              ; Write to standard out = 1
mov rsi, hello_world    ; The address of hello_world string
mov rdx, 14             ; The size to write
syscall                 ; Invoke the kernel
mov rax, 0x2000001      ; System call number for exit = 1
mov rdi, 0              ; Exit success = 0
syscall                 ; Invoke the kernel
```

Illustration only: this program writes "Hello World!" to standard output (fd == 1), ignores the syscall error return, and exits.

Illustration only: the details aren't important.

http://thexploit.com/secdev/mac-os-x-64-bit-assembly-system-calls/

Linux x64 syscall conventions (ABI)

1. User-level applications use as integer registers for passing the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`. The kernel interface uses `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8` and `%r9`.

2. A system-call is done via the `syscall` instruction. The kernel destroys registers `%rcx` and `%r11`.

3. The number of the syscall has to be passed in register `%rax`.

4. System-calls are limited to six arguments, no argument is passed directly on the stack.

5. Returning from the `syscall`, register `%rax` contains the result of the system-call. A value in the range between -4095 and -1 indicates an error, it is `-errno`.

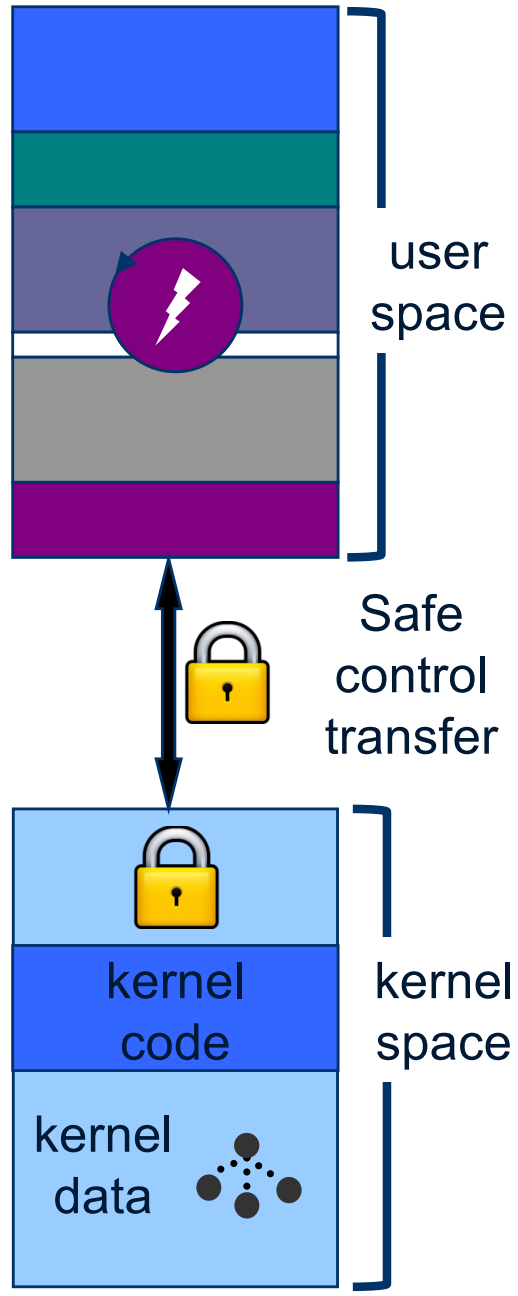
(user buffer addresses)

6. Only values of class INTEGER or class MEMORY are passed to the kernel.

Illustration only: the details aren't important.

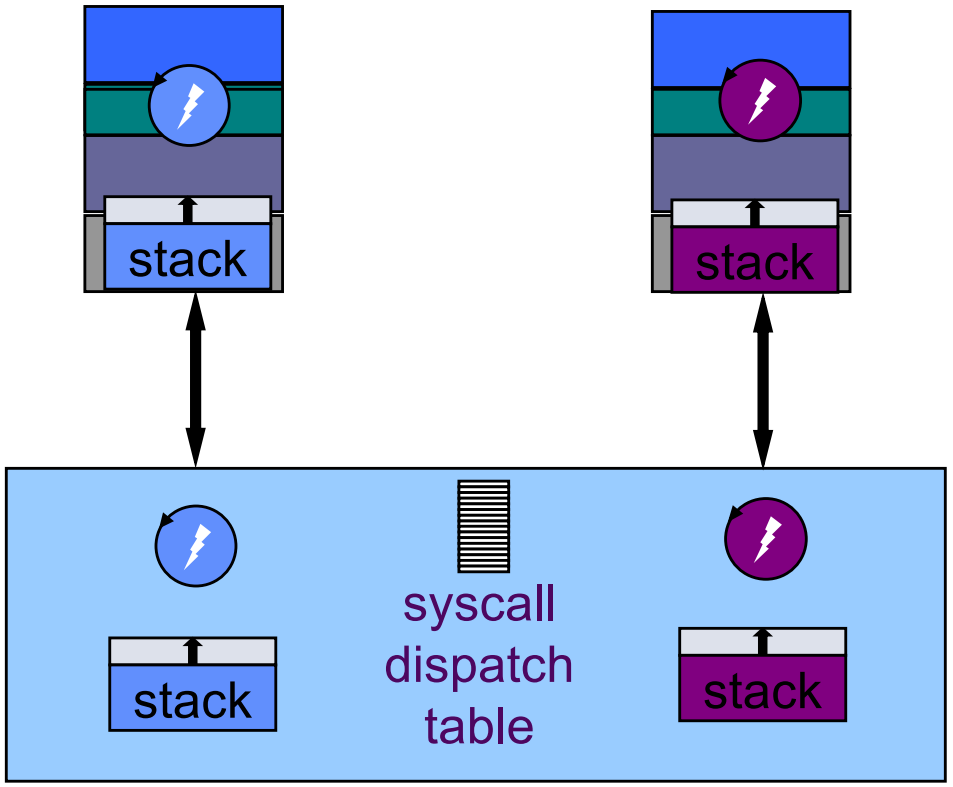
Entering the kernel

- CPU event: trap, fault, interrupt
- Core transitions to kernel mode
  - Map kernel space—if not already mapped.
- Set PC to execute a pre-designated handler routine for that event type.
  - Lookup event type in table or “vector”
- Set SP to pre-designated kernel stack.
  - Per-thread or interrupt stack



Kernel Stacks and Trap/Fault Handling

Threads execute user code on a **user stack** in user space (the process virtual address space).



System calls and faults run in kernel mode on a kernel stack for the current thread.

Each thread has a second **kernel stack** in **kernel space** (VM accessible only in kernel mode).

Kernel code running in P's process context has access to P's virtual memory.

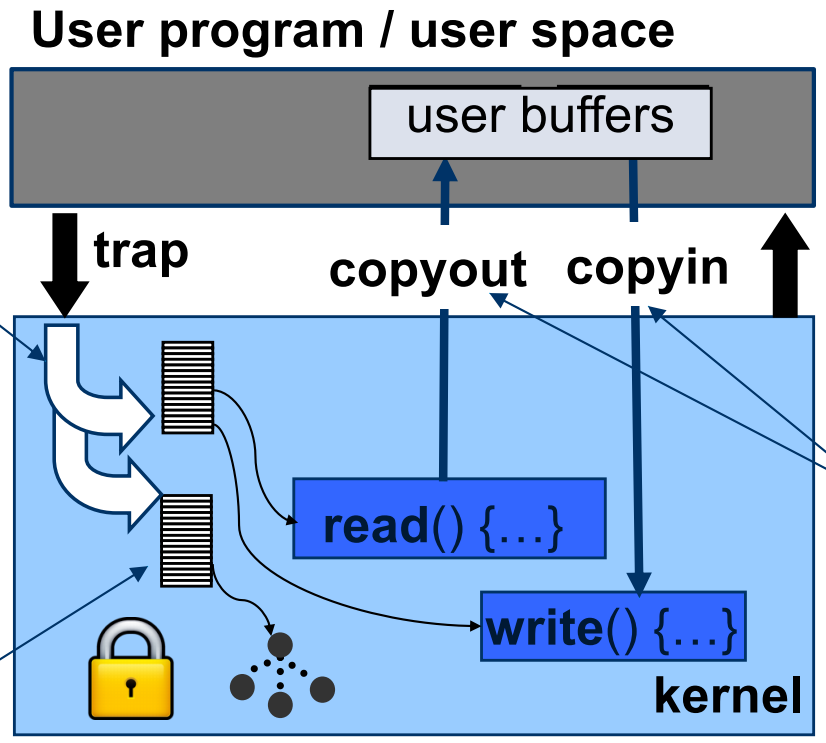
The syscall (trap) handler makes an indirect call through the system call dispatch table to the handler registered for the specific system call.

The kernel must be bulletproof

Secure kernels handle system calls verrry carefully.

Syscalls indirect through **syscall dispatch table** by syscall number. No direct calls to kernel routines from user space!

What about references to kernel data objects passed as syscall arguments (e.g., file to read or write)?



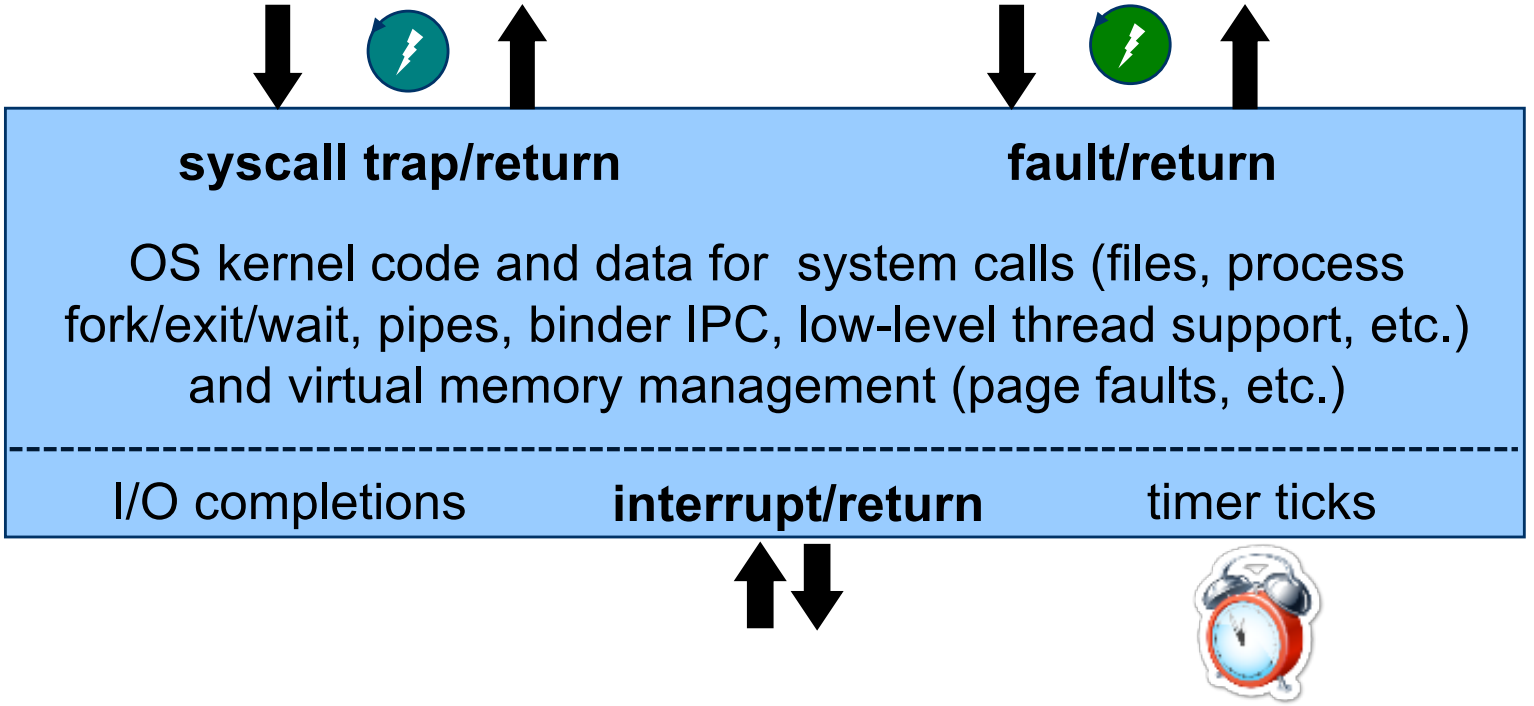
Kernel copies all arguments into kernel space and validates them.

Kernel interprets pointer arguments in context of the user VAS, and copies the data in/out of kernel space (e.g., for **read** and **write** syscalls).

Use an integer index into a kernel table that points at the data object. The value is called a **handle** or **descriptor**. No direct pointers to kernel data from user space!

Entry to the kernel

Every entry to the kernel is the result of a **trap**, **fault**, or **interrupt**. The core switches to kernel mode and transfers control to a handler routine.



The handler accesses the core register context to read the details of the exception (trap, fault, or interrupt). It may call other kernel routines.

Recap: OS protection

Know how a classical OS uses the hardware to protect itself and implement a **limited direct execution** model for untrusted user code.

- **Virtual addressing**. Applications run in sandboxes that prevent them from calling procedures in the kernel or accessing kernel data directly (unless the kernel chooses to allow it).
- **Events**. The OS kernel installs handlers for various machine events when it boots (starts up). These events include machine exceptions (faults), which may be caused by errant code, interrupts from the clock or external devices (e.g., network packet arrives), and deliberate kernel calls (traps) caused by programs requesting service from the kernel through its API.
- **Designated handlers**. All of these machine events make safe control transfers into the kernel handler for the named event. In fact, once the system is booted, these events are the only ways to ever enter the kernel, i.e., to run code in the kernel.