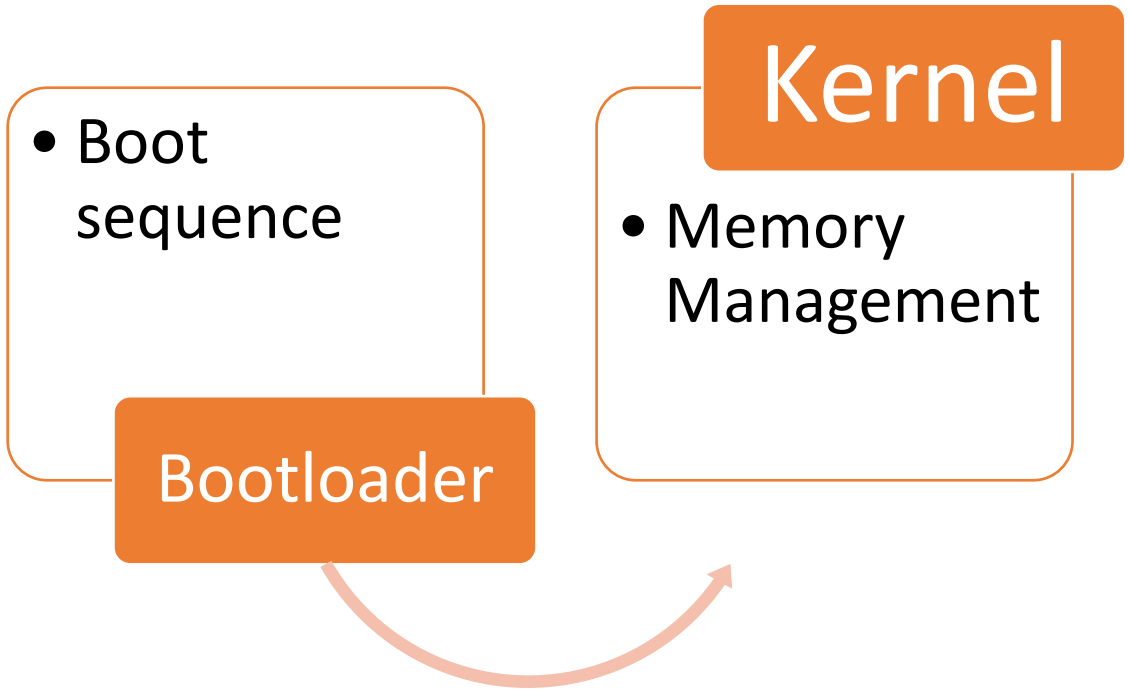
CS444/544 Operating Systems II

Prof. Sibin Mohan

Spring 2022 | Lec7: User and Kernel Spaces

Adapted from content originally created by: Prof. Yeongjin Jang

So far, we've seen...



What is an Operating System [OS]?

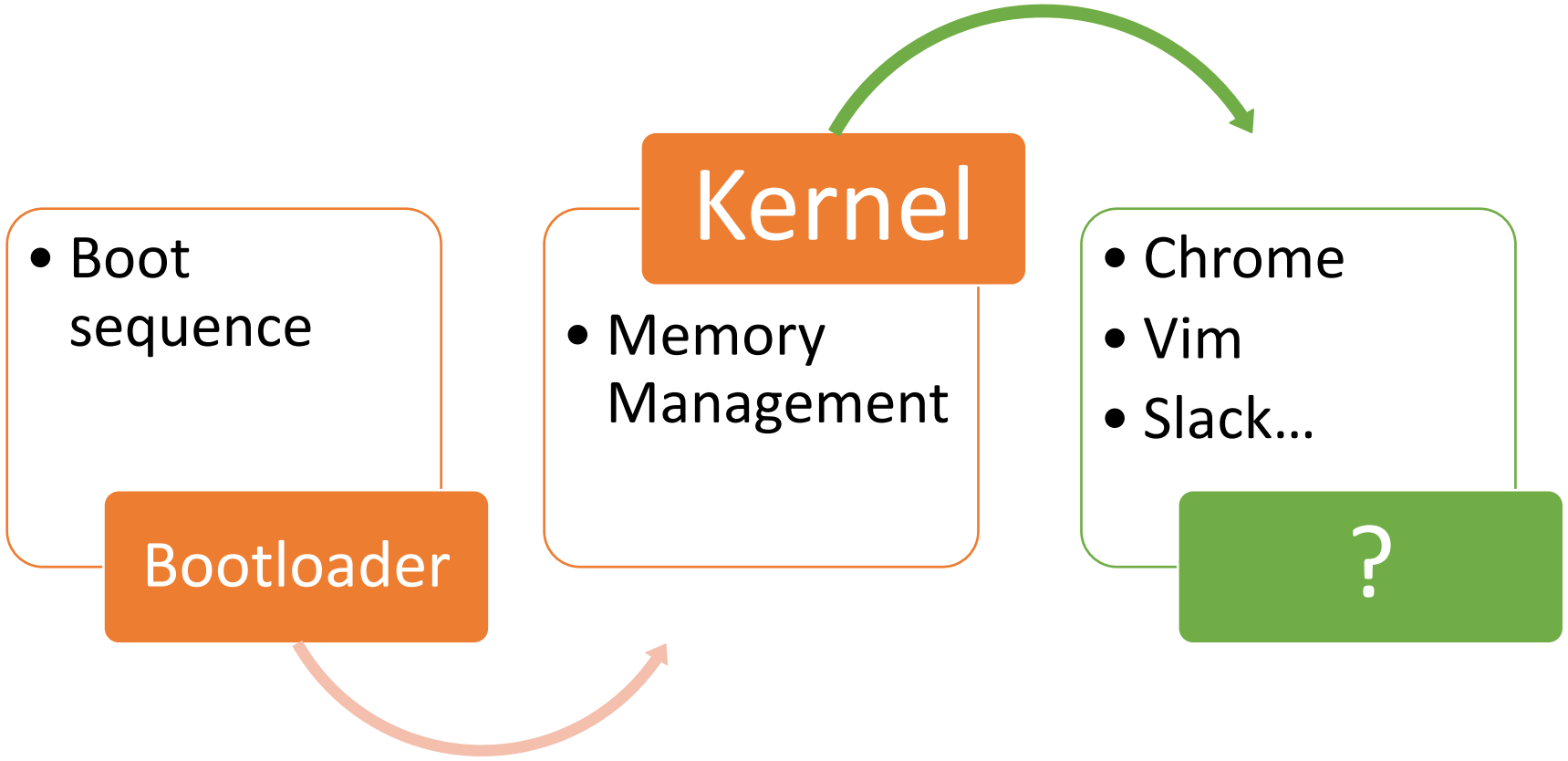
What is an OS?

- **Body of software**
- Allows users (and programs) to use the **low-level hardware**
  - Share memory, enable interactions with devices, etc.
- Manages **sharing** of resources across multiple programs
- Provides additional features like **security**, **isolation**, etc.

- In charge of ensuring system operates **correctly** and **efficiently**

Multiple programs **sharing** hardware resources, **efficiently** and **isolated** from each other

What's next?


• Boot sequence
Bootloader
Kernel
• Memory Management
• Chrome
• Vim
• Slack…
?

Where can these programs run?

- Kernel?

Pros	Cons

User Space

- Boot sequence

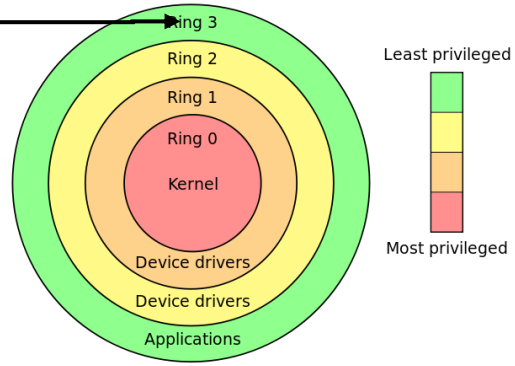
Bootloader

Kernel

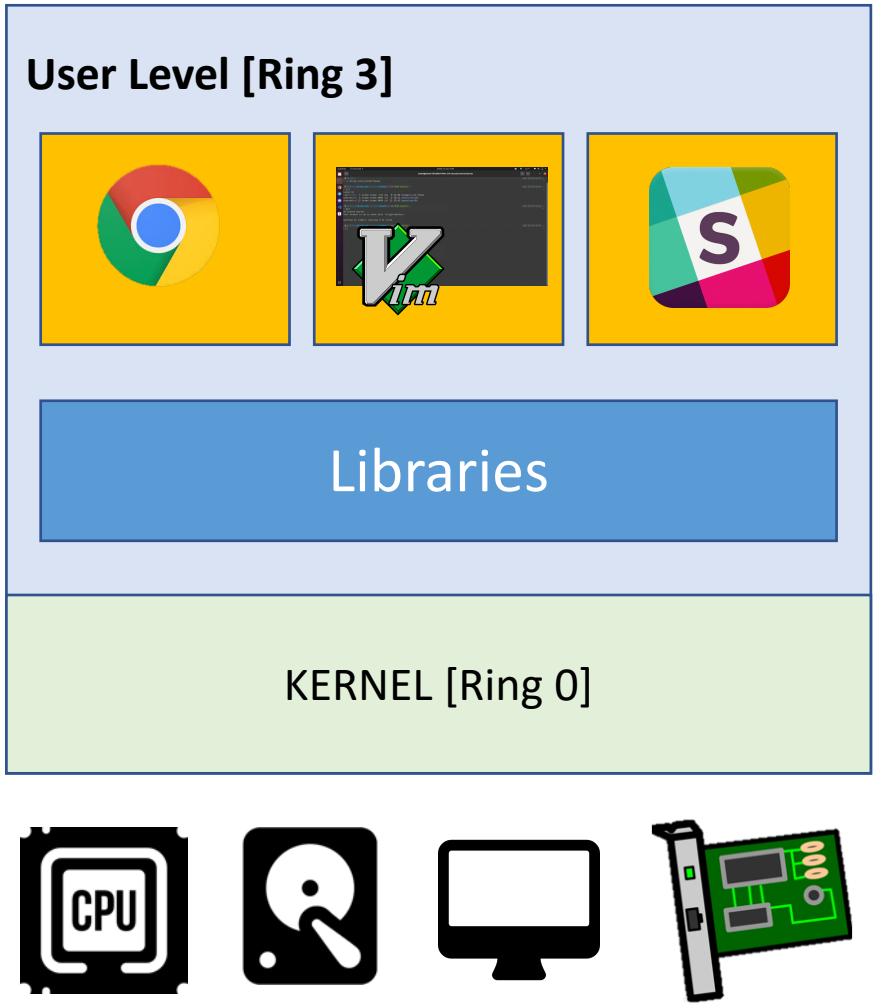
- Memory Management

- Chrome
- Vim
- Slack…

User Space



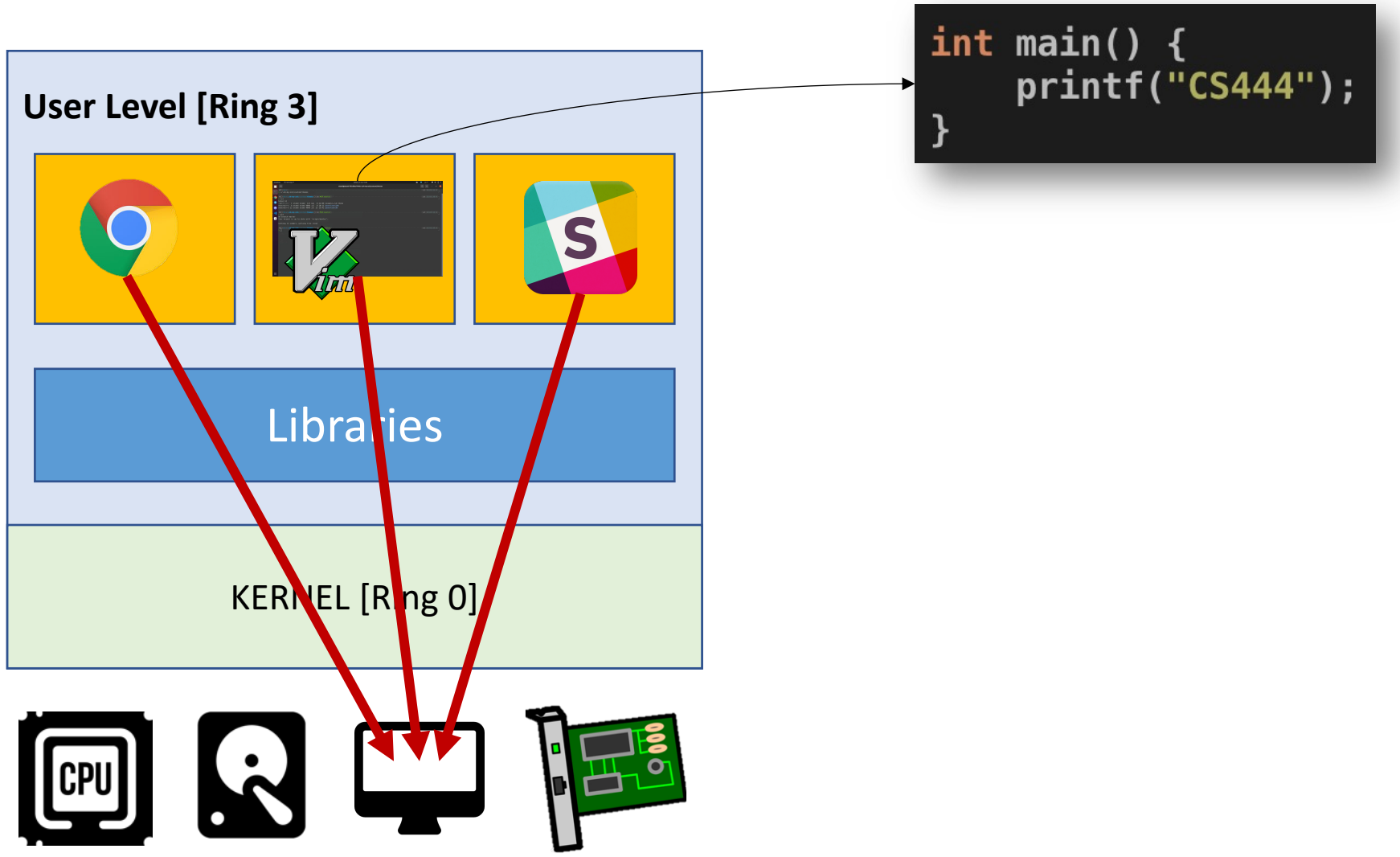
User Space [**Ring 3**]



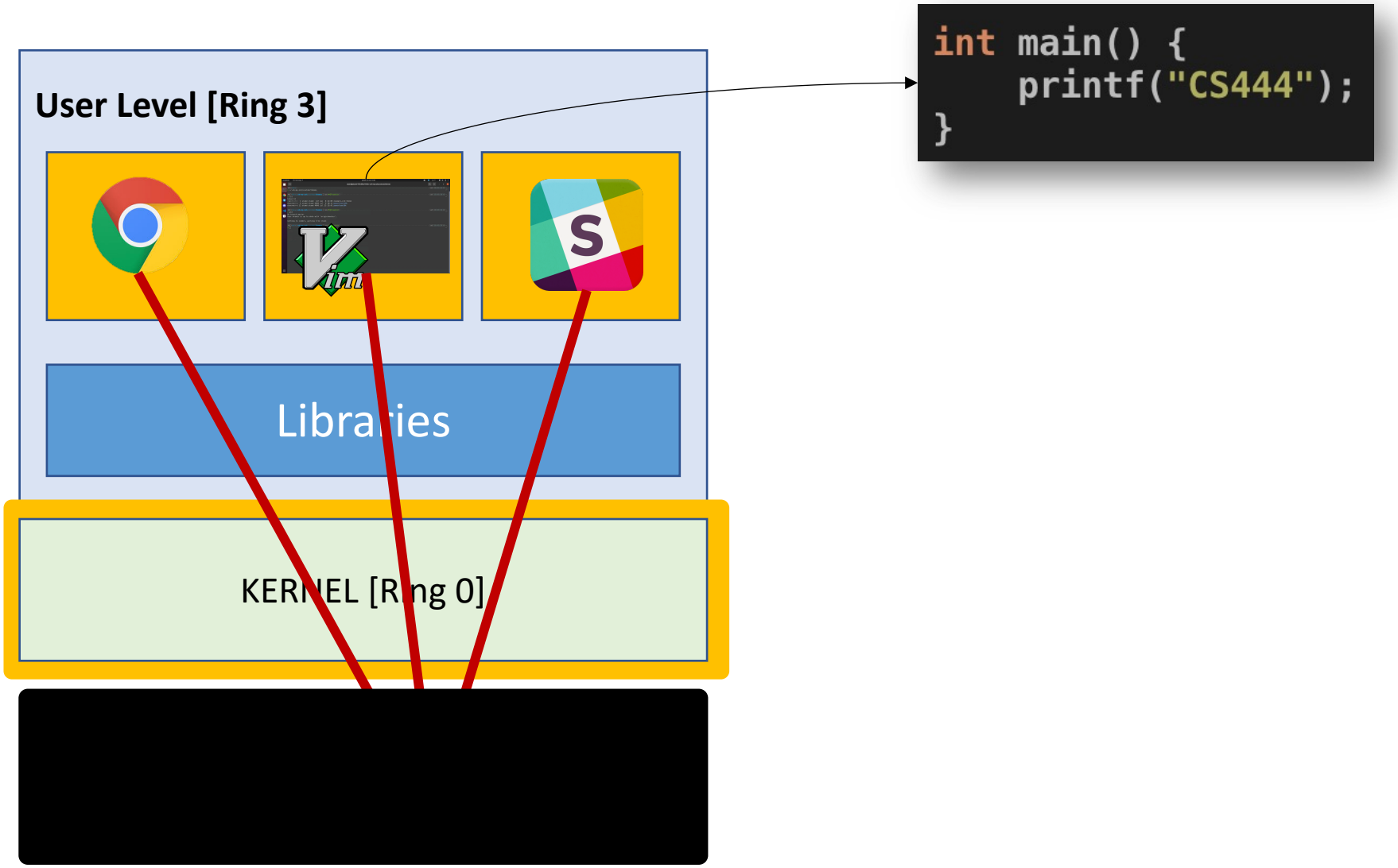
Issues that need to be resolved

1. How do we manage **multiple programs**?
2. How can user programs **access hardware**?
3. Can a ring 3 program use kernel **services**?
4. **Switching** between kernel/user spaces?
5. How does the **kernel regain control**?

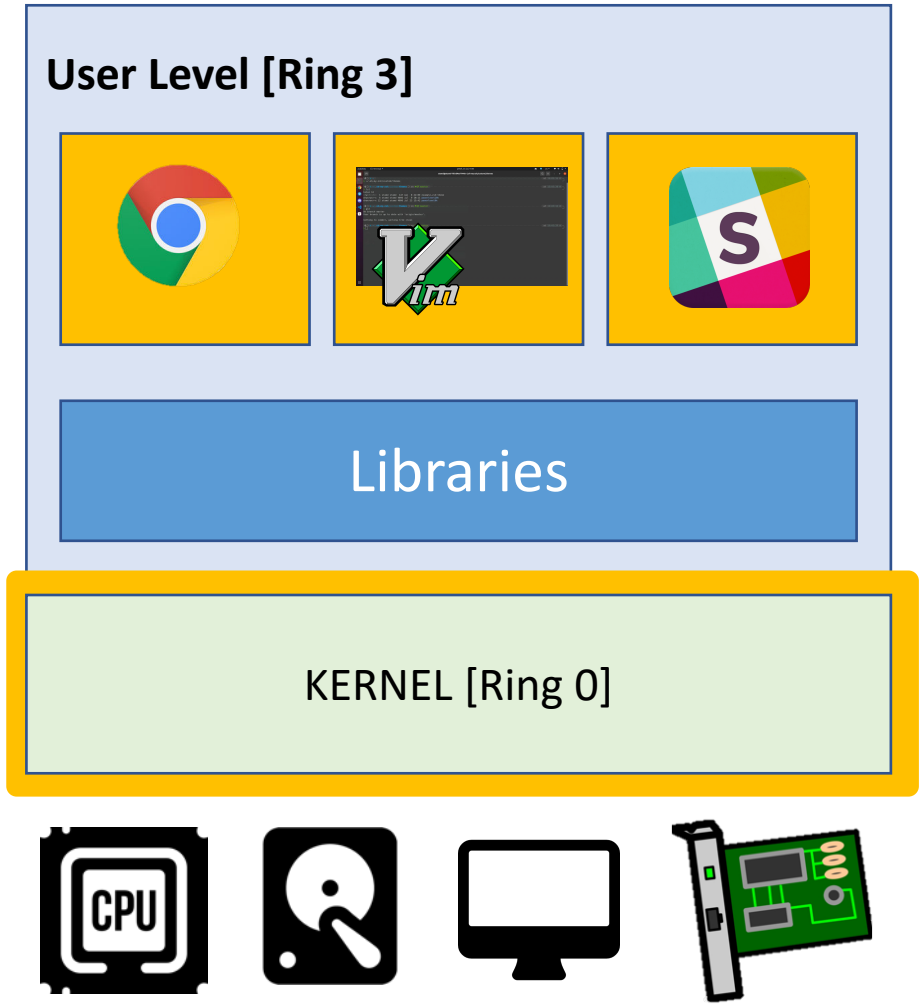
How does a User Space Program Work?



How does a User Space Program Work?

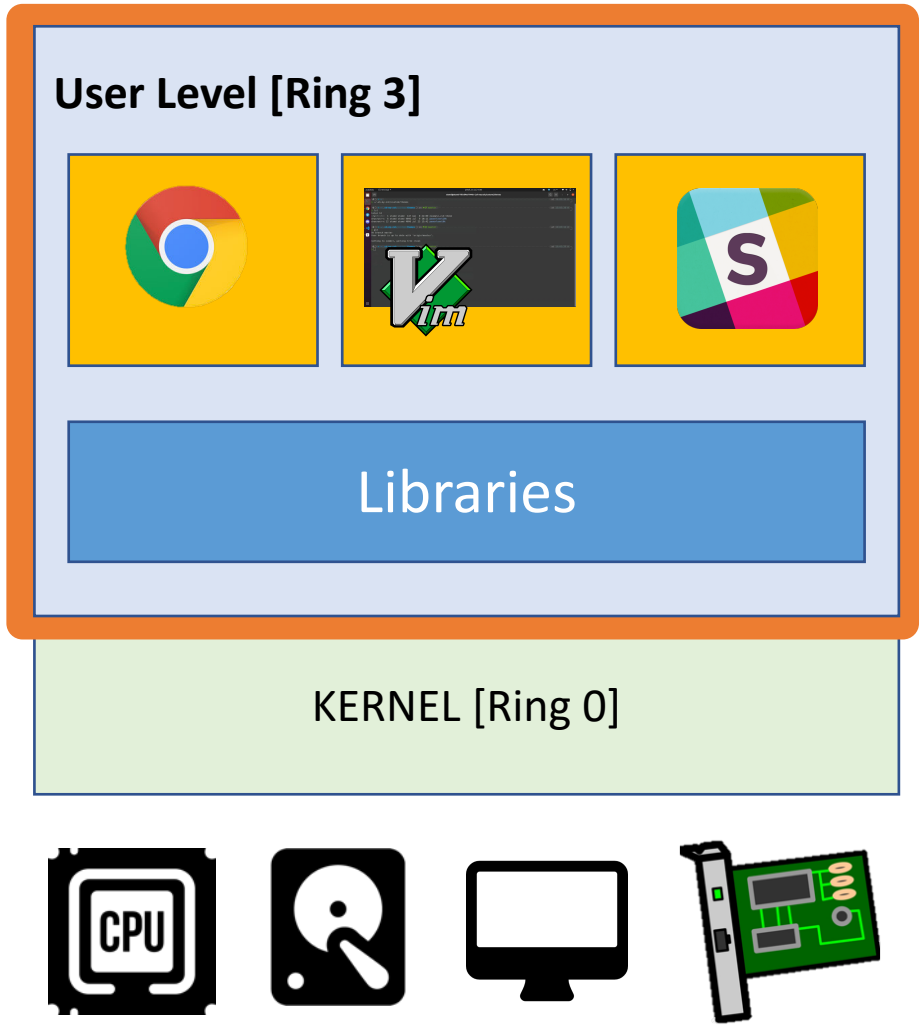


Kernel [Ring 0]



- Executes with **highest privilege level** (Ring 0)
- Configures system (devices, memory, etc.)
- Manages hardware resources
  - Disk, memory, network, video, keyboard, etc.
- Manages **other jobs**
  - Processes and threads
- Serves as **trusted computing base** (TCB)
  - Sets privilege
  - Restrict other jobs from doing something bad

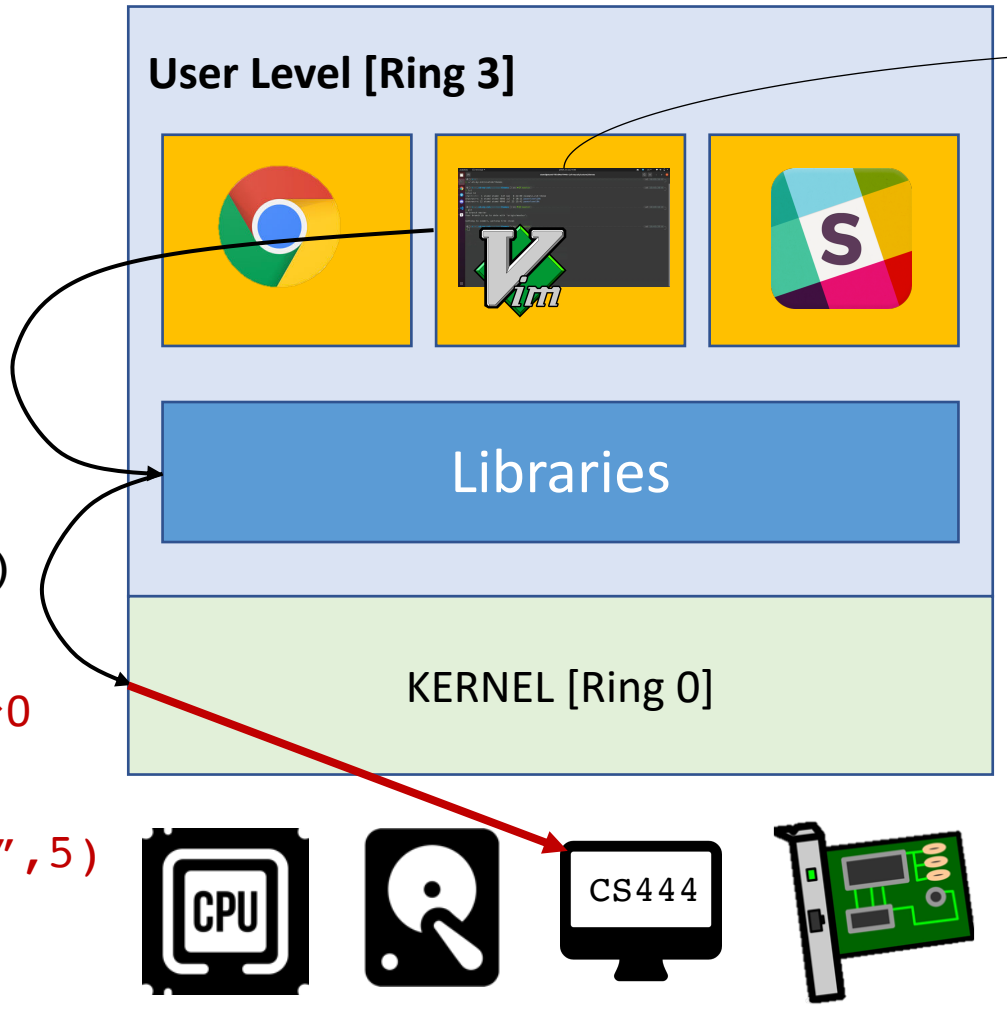
User [**Ring 3**]



- Runs with a **restricted** privilege [**Ring 3**]
  - The privilege level for running an application
- Most regular applications run at this level
- **Cannot** access kernel memory
  - Can only access pages set with **PTE_U**
- **Cannot talk directly to hardware devices**
  - Kernel must mediate the access

So, what happens with that `printf()`?

```
int main() {
    printf("CS444");
}
```



`printf``("CS444")`

library call in **ring 3**

`sys_write``(1,"CS444",5)`

system call from **ring 3**

Interrupt! Switch from ring 3 →0

`do_sys_write``(1,"CS444",5)`

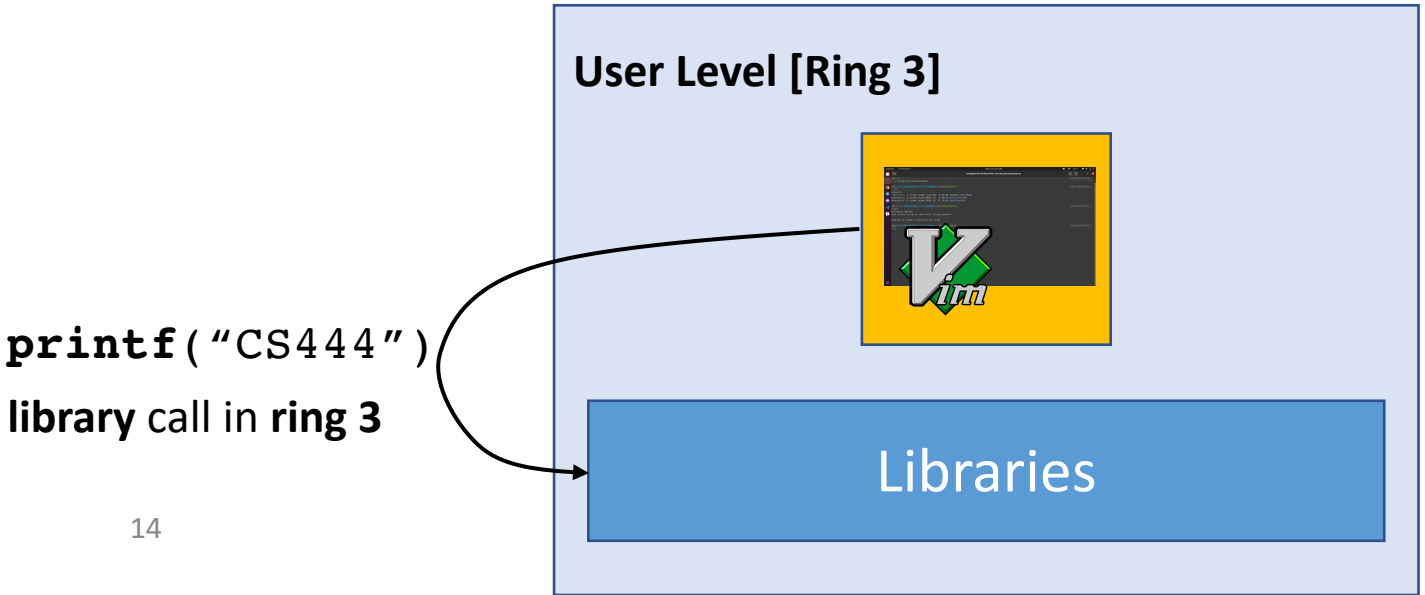
kernel function in **ring 0**

Library Call

- A function call **within the application's memory space**

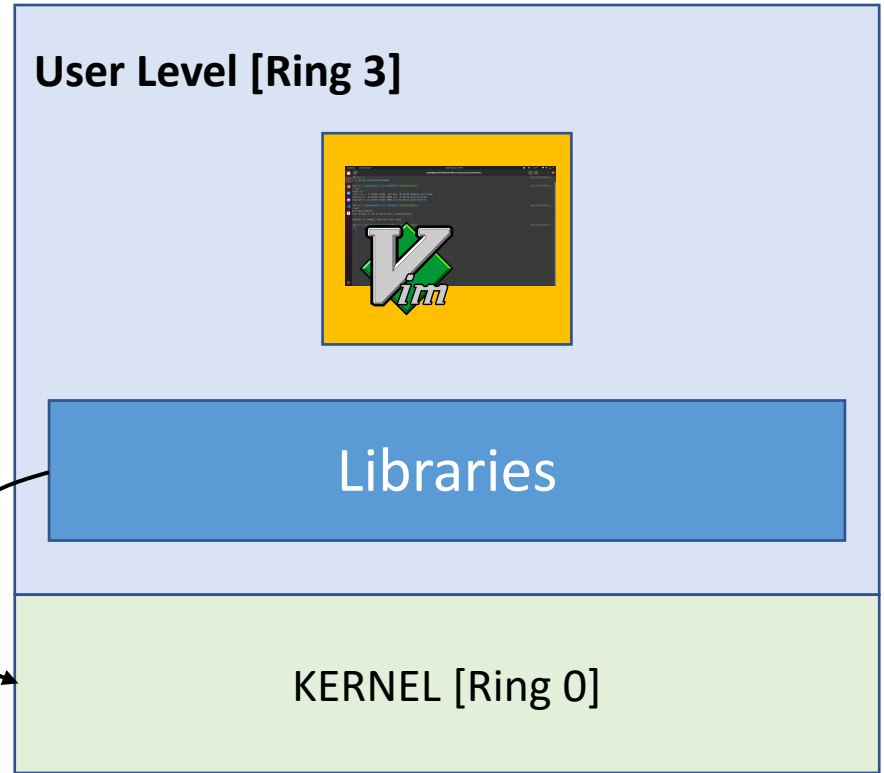
- All regular C/C++ API calls are library calls
  - `fwrite(), printf(), time(), srand(),` etc.
  - Calls that you did not implement but prepared by others [in ring 3]

- **Ring 3 → Ring 3**



System Call

- A function call from applications requesting an **OS service**
- **System APIs**
  - I/O access [`read(), write(), send(), recv(),` etc.]
  - Process creation, destruction [`exec(), fork(), kill(),` etc.]
  - Other hardware access
- **Ring 3 → Ring 0**

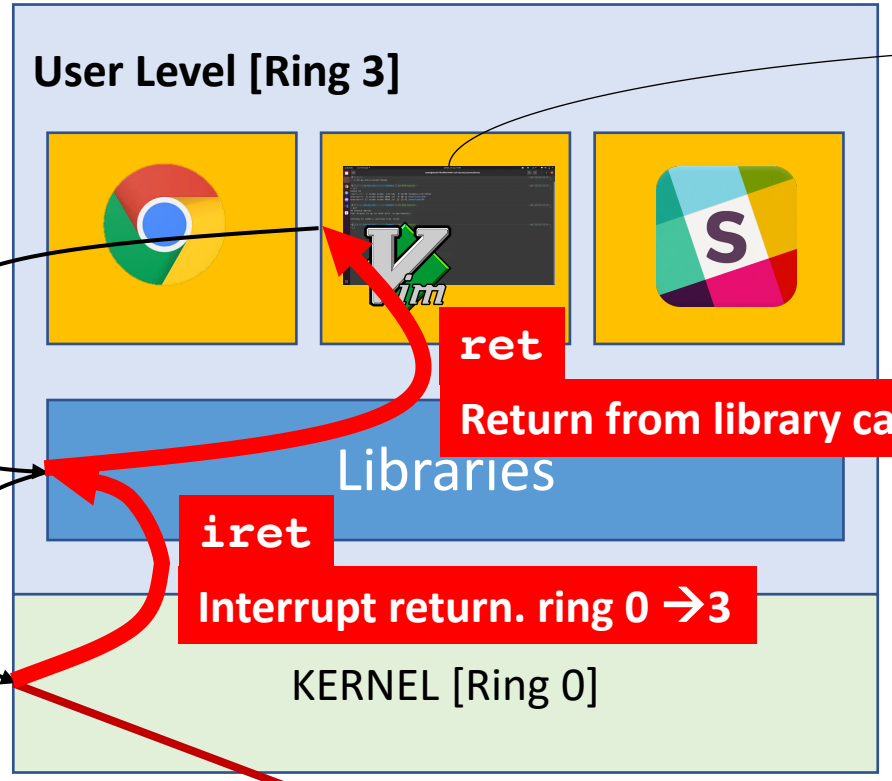


`sys_write``(1,"CS444",5)`

system call from **ring 3**

We're not done with `printf()` though!

```
int main() {
    printf("CS444");
}
```



`printf``("CS444")`
library call in **ring 3**

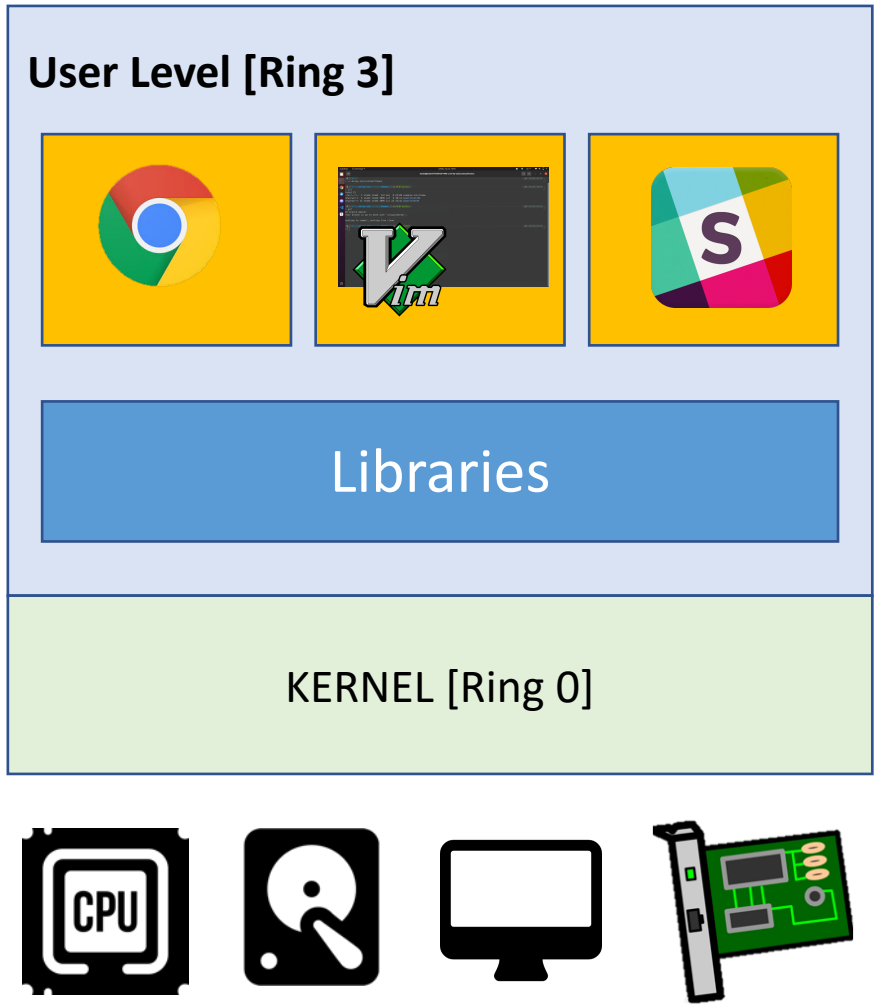
`sys_write``(1,"CS444",5)`
system call from **ring 3**
Interrupt! Switch from ring 3 →0

`do_sys_write``(1,"CS444",5)`
kernel function in **ring 0**

`ret`
Return from library call. ring 3 [no switch]

`iret`
Interrupt return. ring 0 →3

How does the kernel **execute** an application?

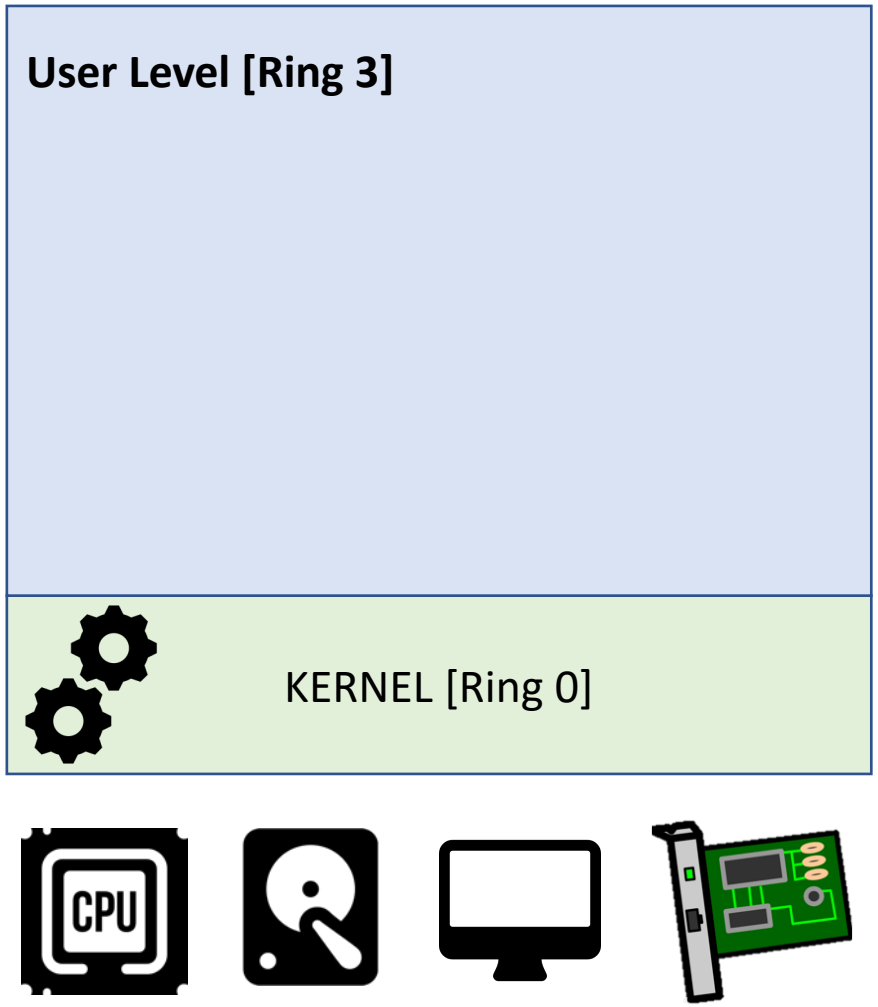

User Level [Ring 3]
Libraries
KERNEL [Ring 0]
CPU

Starting Up

[lab3] **setup user environment**

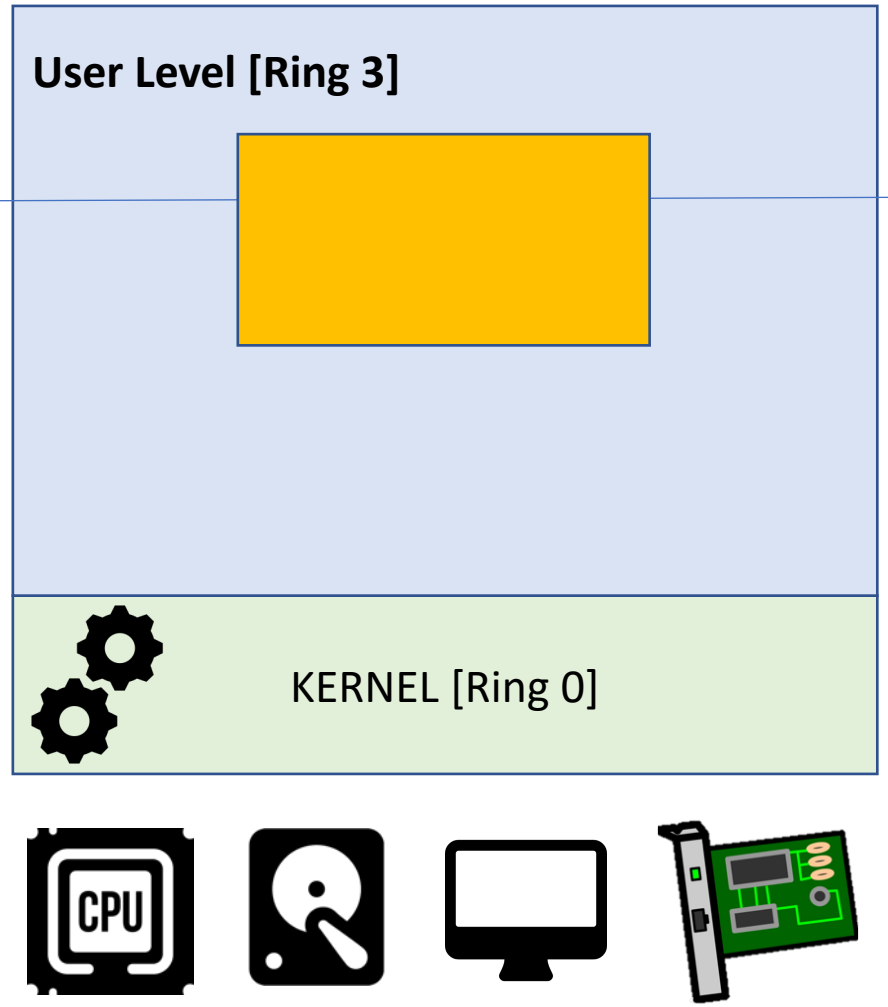
[lab2] memory management

[lab1] boot process



Process Setup

1. Create a **process**
   [environment for running an application]



Assign a separate virtual memory space:

- new page directory
- new page table

[lab3] **setup user environment**

[lab2] memory management

[lab1] boot process

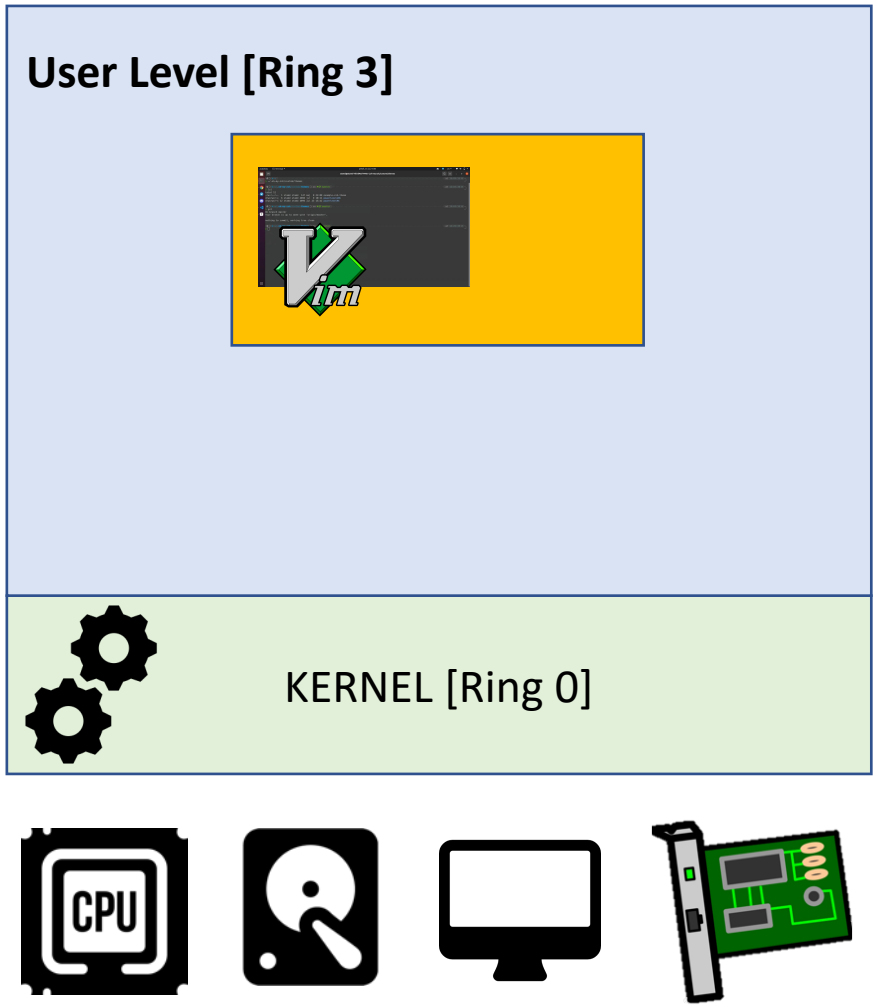
Process Setup

1. Create a **process** [environment for running an application]
2. **Load** application code

[lab3] **setup user environment**

[lab2] memory management

[lab1] boot process



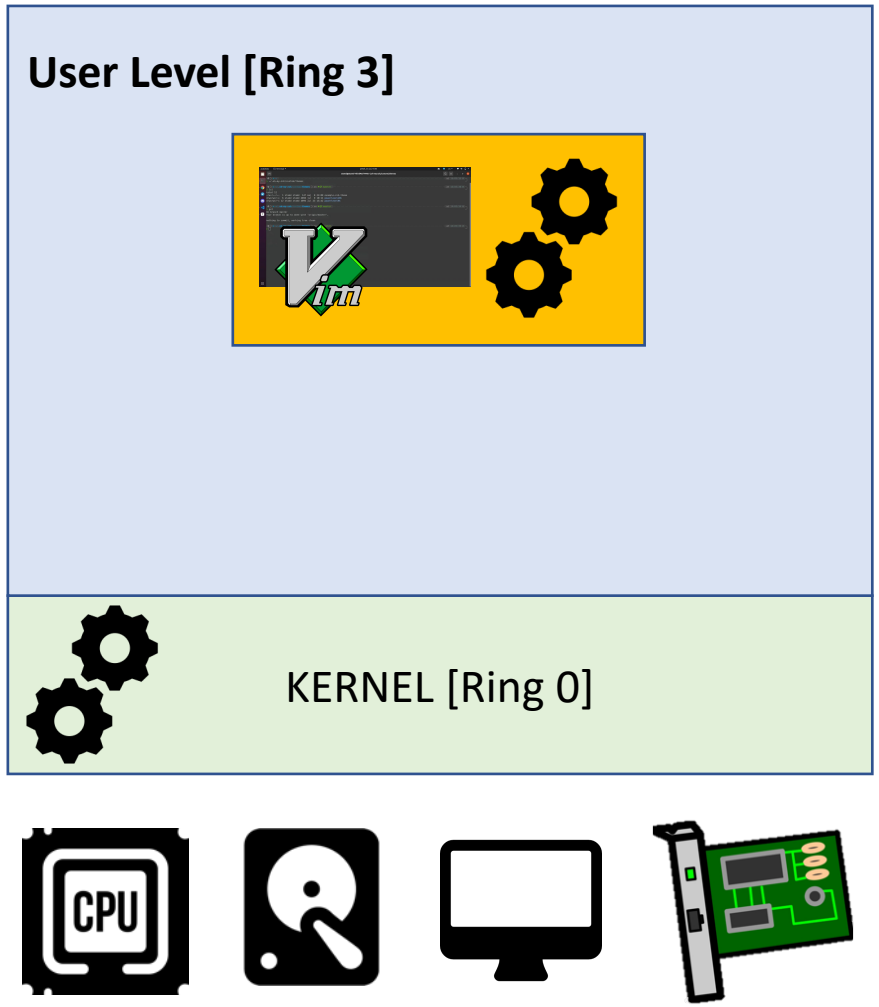
Process Setup

1. Create a **process** [environment for running an application]
2. **Load** application code
3. **Execute!**

[lab3] **setup user environment**

[lab2] memory management

[lab1] boot process



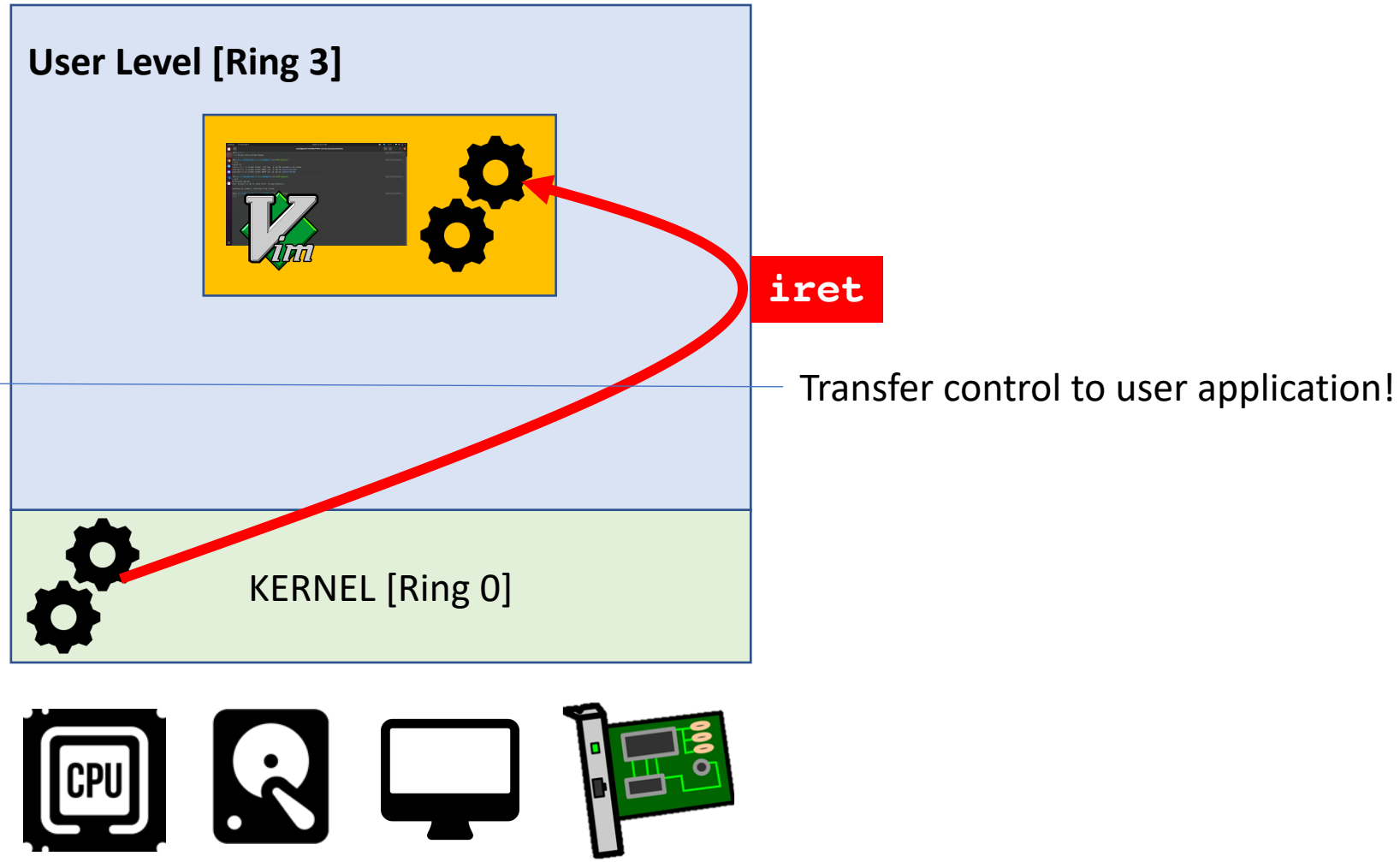
Process Setup

1. Create a **process** [environment for running an application]
2. **Load** application code
3. **Execute!**

[lab3] **setup user environment**

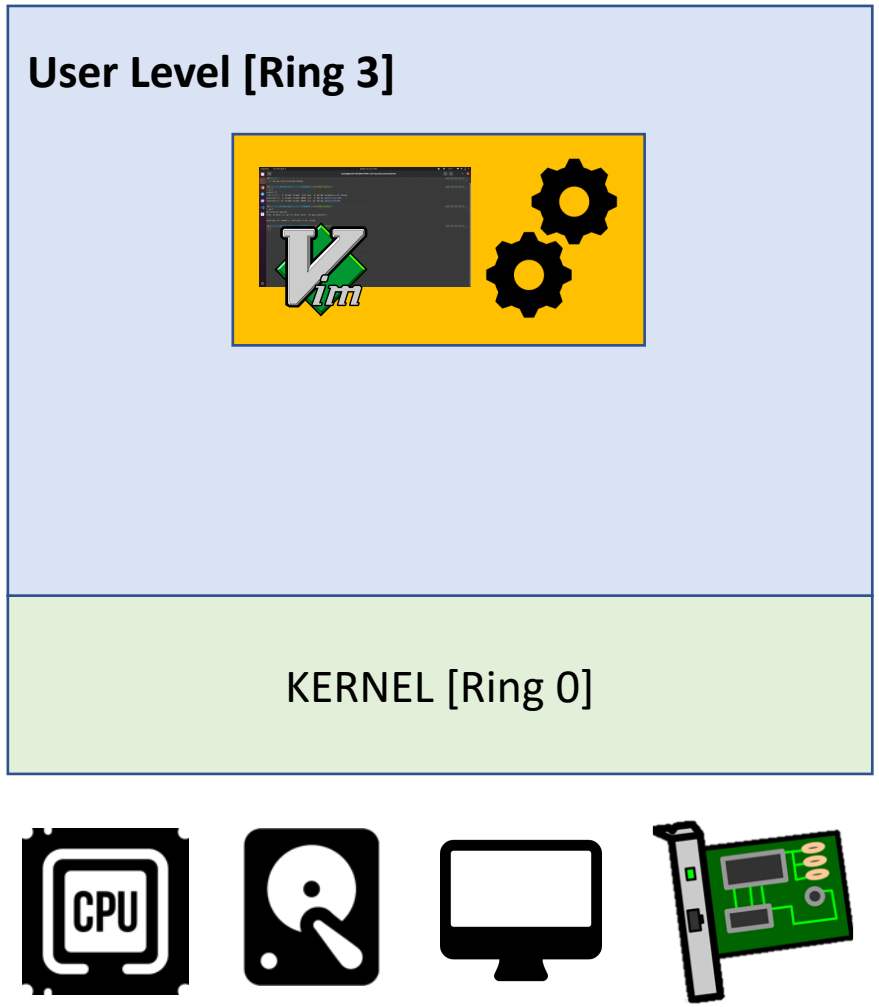
[lab2] memory management

[lab1] boot process



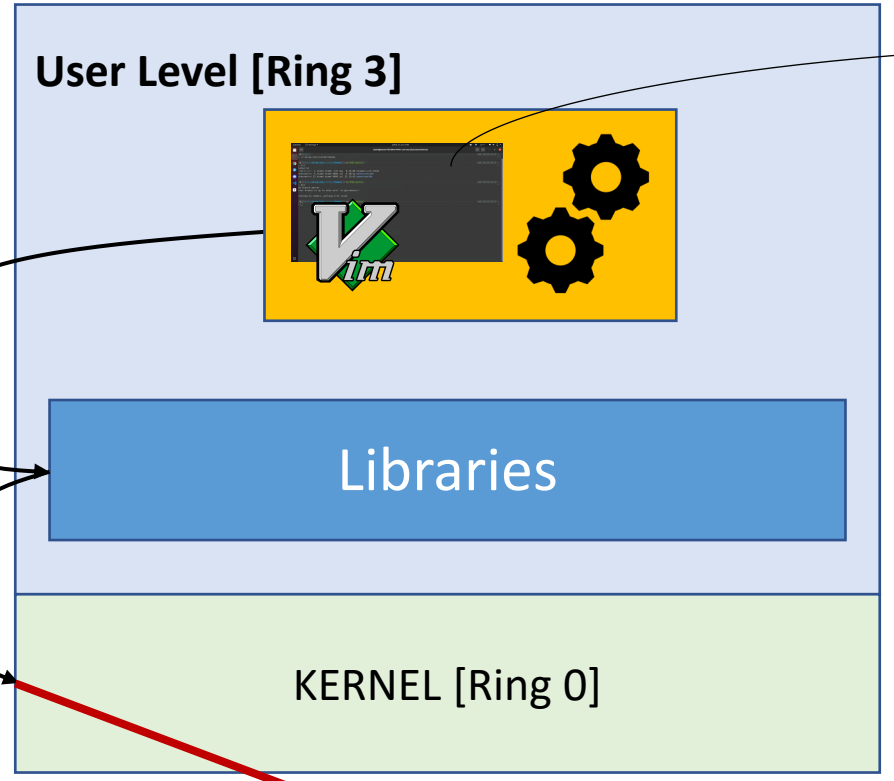
But, how does the kernel get back control?

Current State | Application/Process Executing



Let's revisit `printf()`

```
int main() {
    printf("CS444");
}
```



`printf``("CS444")`

library call in **ring 3**

`sys_write``(1,"CS444",5)`

system call from **ring 3**

Interrupt! Switch from ring 3 →0

`do_sys_write``(1,"CS444",5)`

kernel function in **ring 0**

Is System Call the **ONLY** Way to return execution to the Kernel?

- **No!**

- If that was the case, we would have lots of **problems**
  - E.g., kernel waits until an application executes a system call
  - What if an application never invokes a system call????
    - OS can never get back control

Switch from User to Kernel Space

System call

[ring 3 → ring 0]

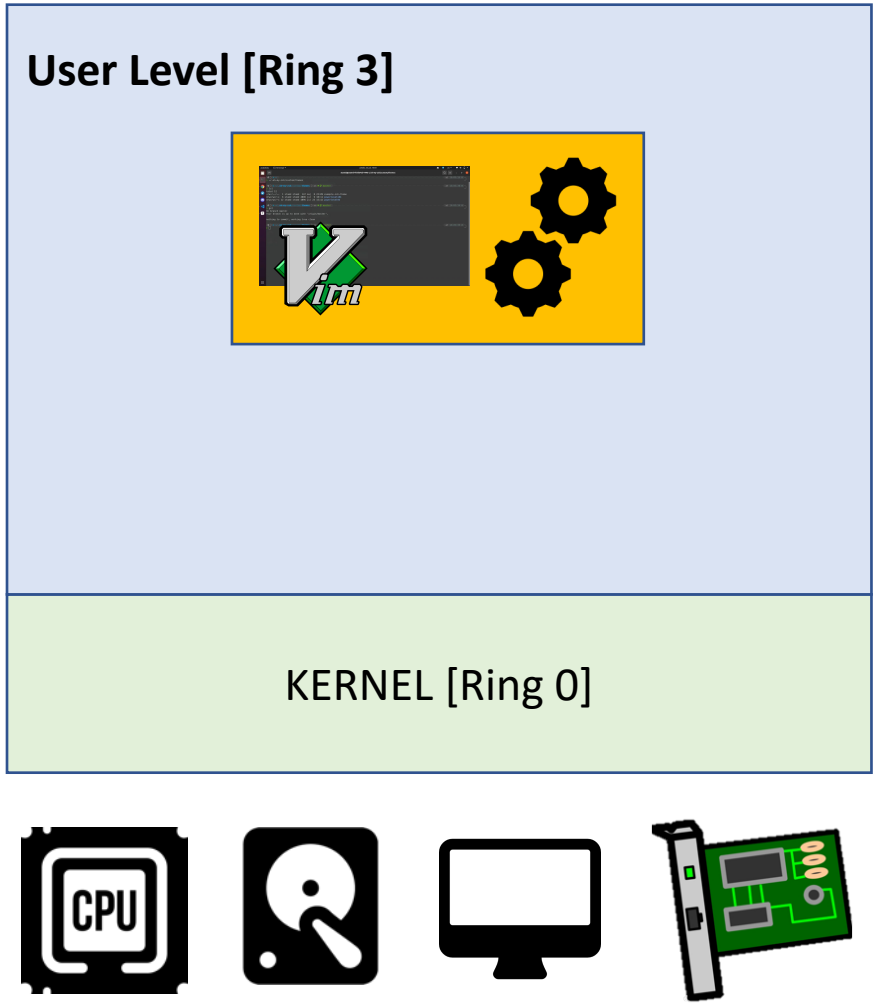
Interrupt

[usually runs in ring 0, sometimes runs in ring 3]

Fault/Exception

[runs in ring 0]

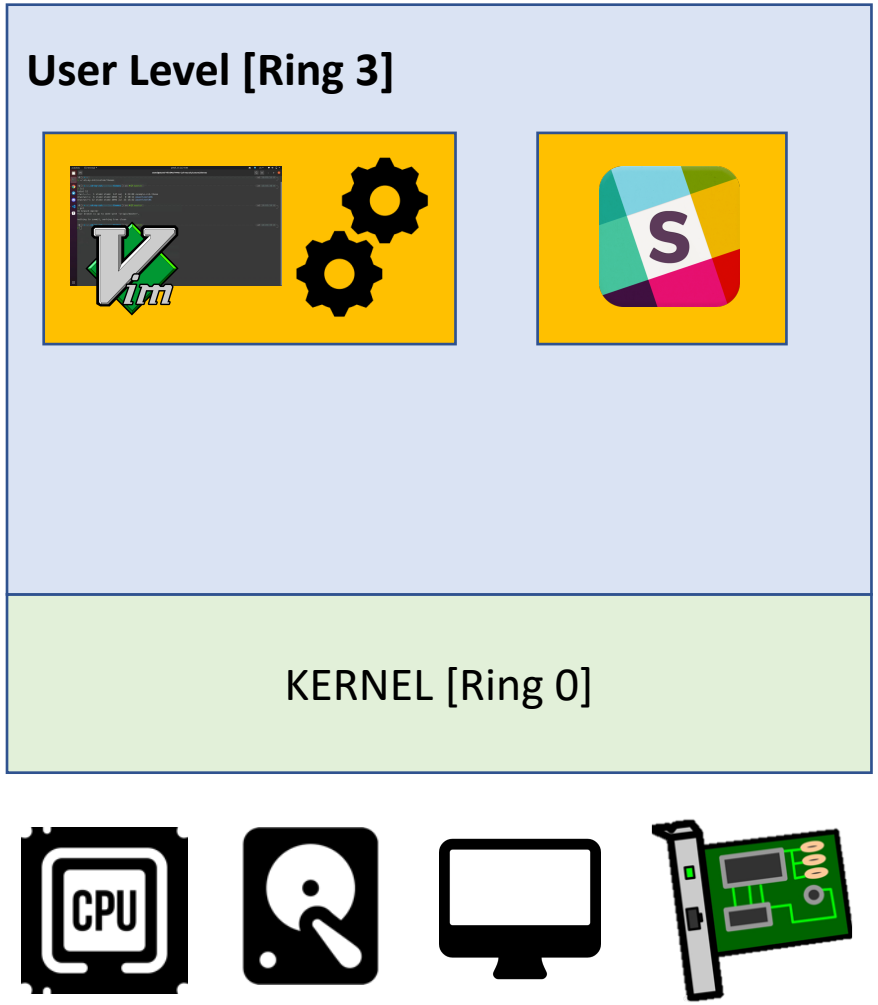
Current State



- Only one application
- Wasted resources
- What if you want **multiple applications**?

- How will you do it? What **mechanism**?

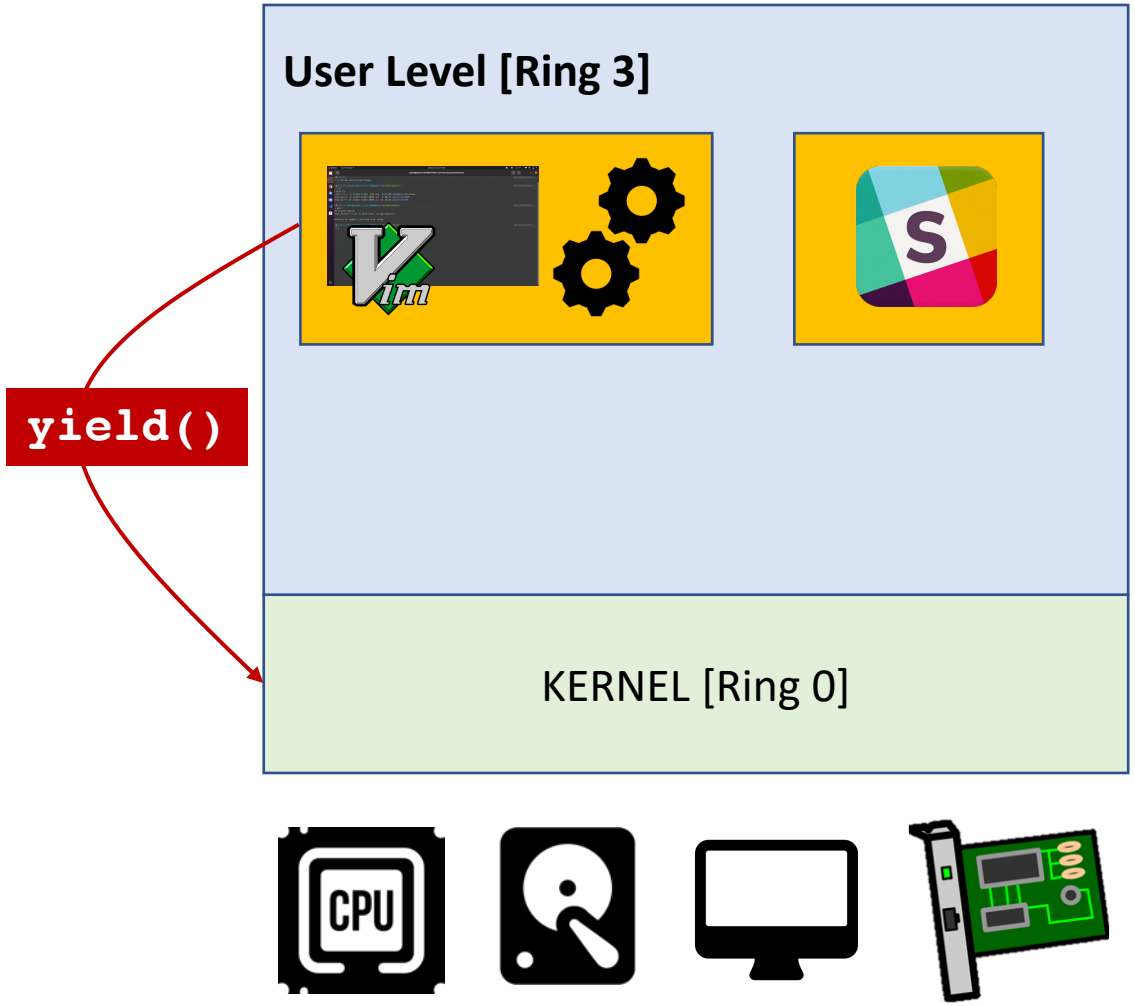
Multiple Applications



- Ways to switch between the two?
- Remember: **CPU runs one at a time**!
  - Vim, Slack or Kernel
- Wait for Vim to invoke a system call
- But what if it **never** invokes one?

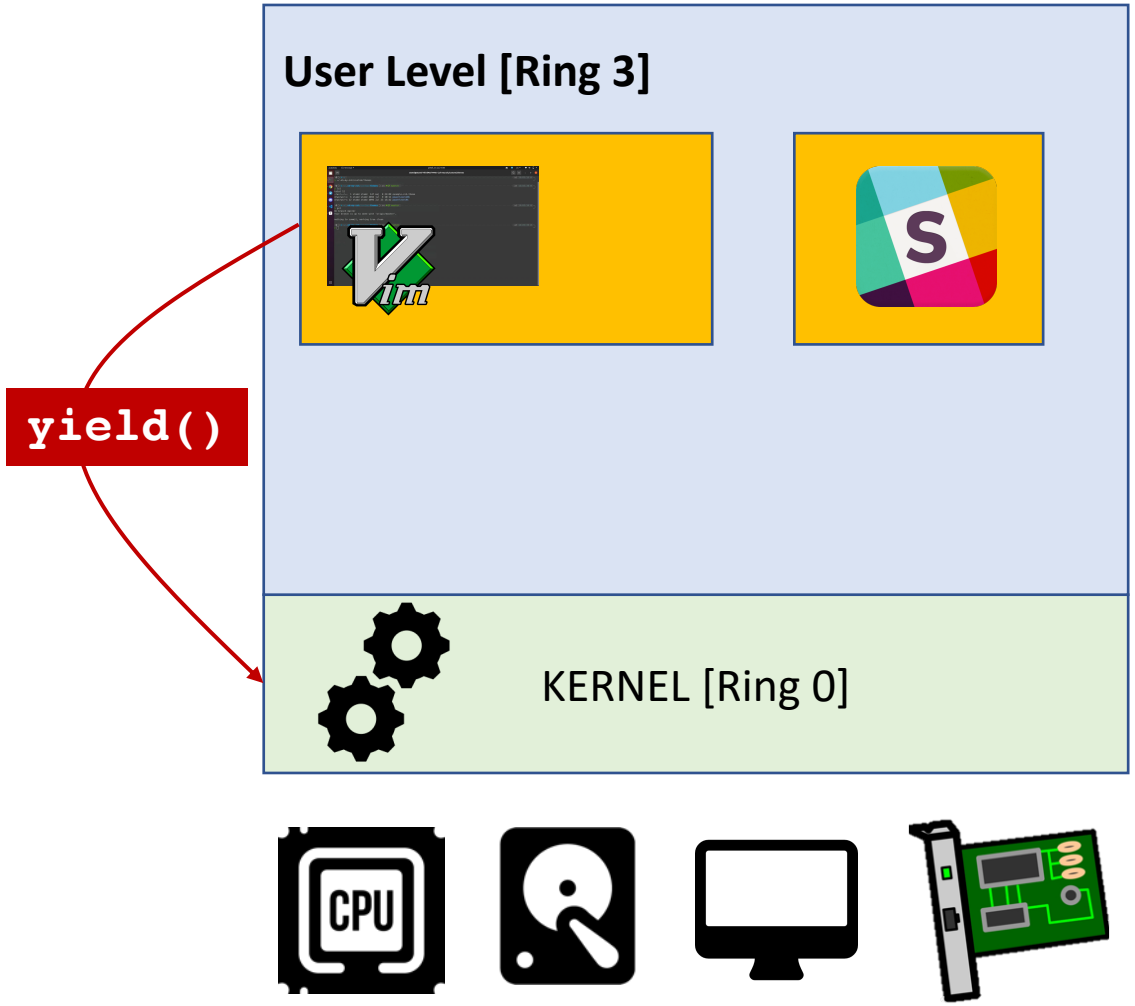
Simplest Method | Cooperative Multitasking

- **Yield** execution when a process completes



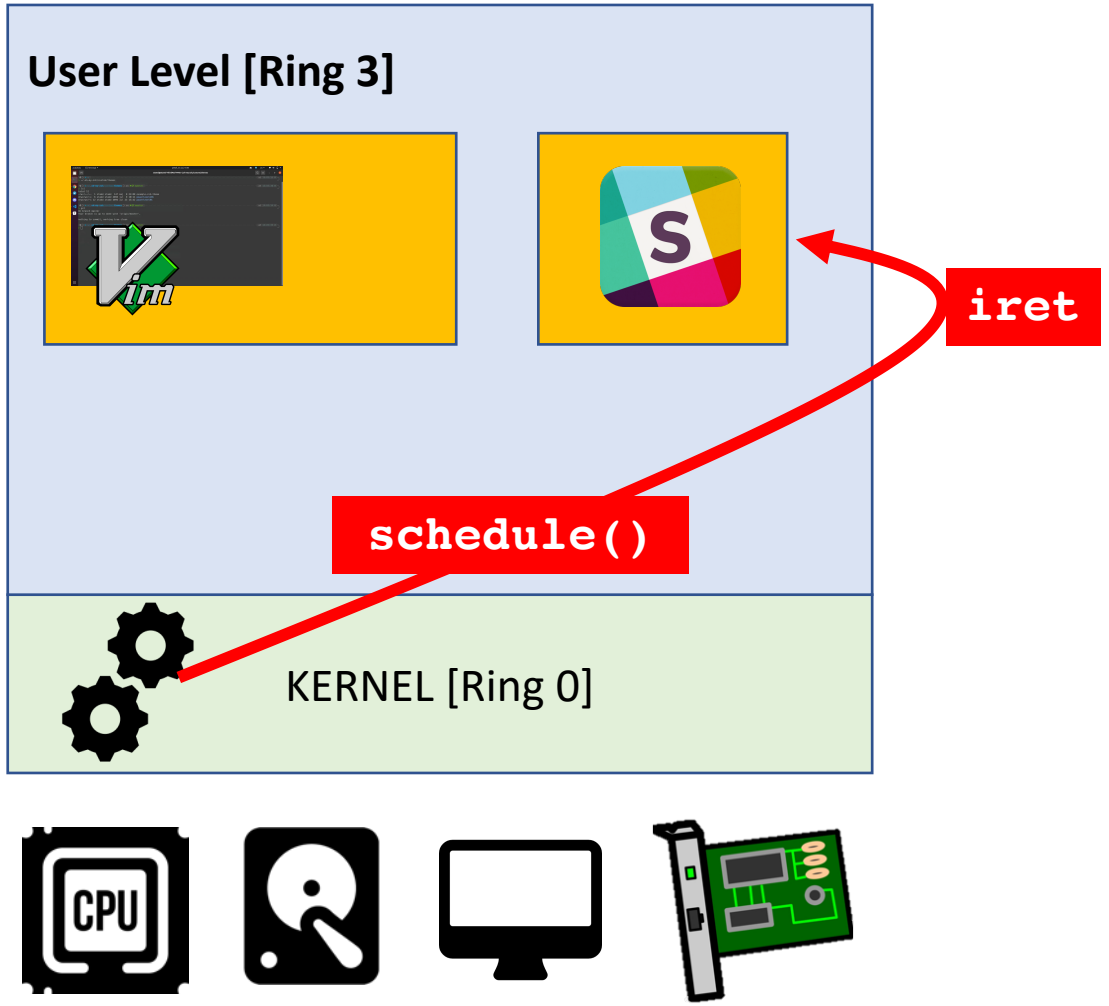
Simplest Method | Cooperative Multitasking

- **Yield** execution when a process completes



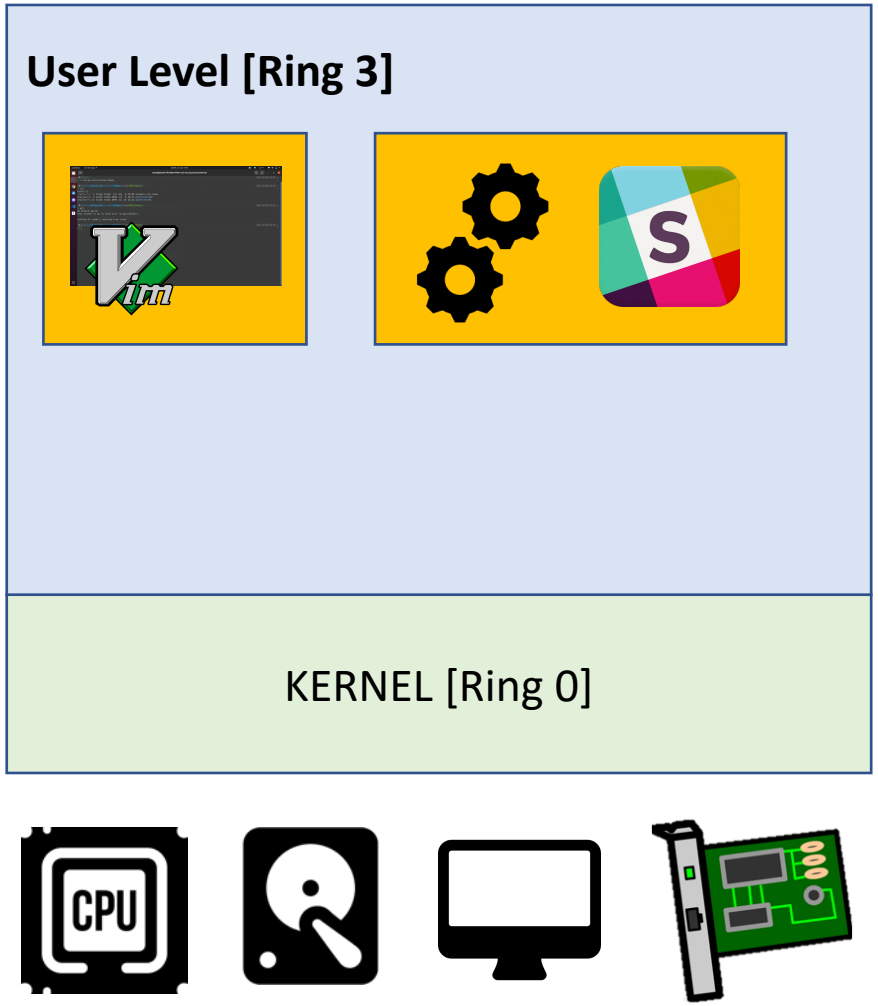
Simplest Method | Cooperative Multitasking

- **Yield** execution when a process completes



Simplest Method | Cooperative Multitasking

- **Yield** execution when a process completes



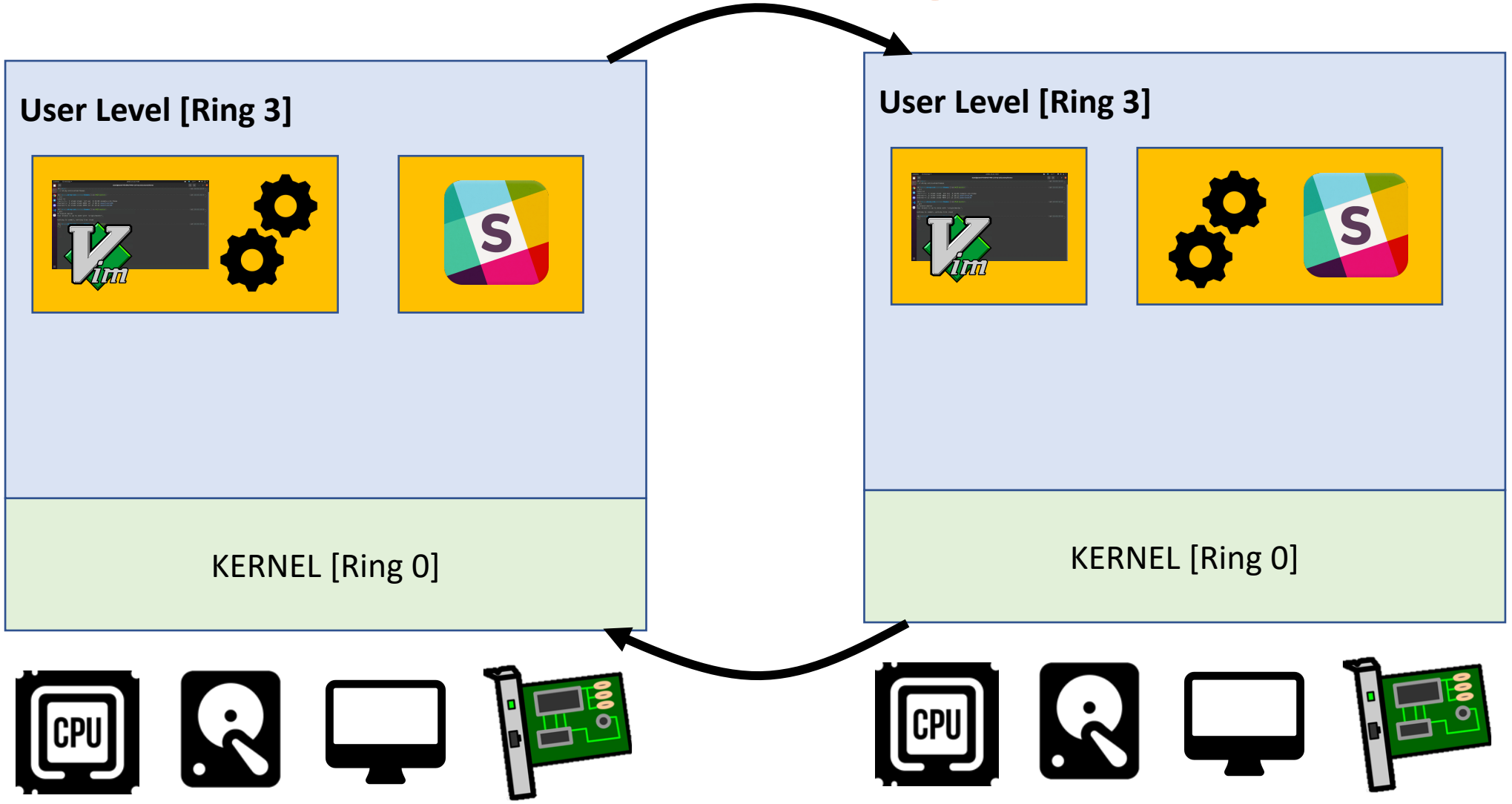
To infinity and beyond…

- What if a user process executes,

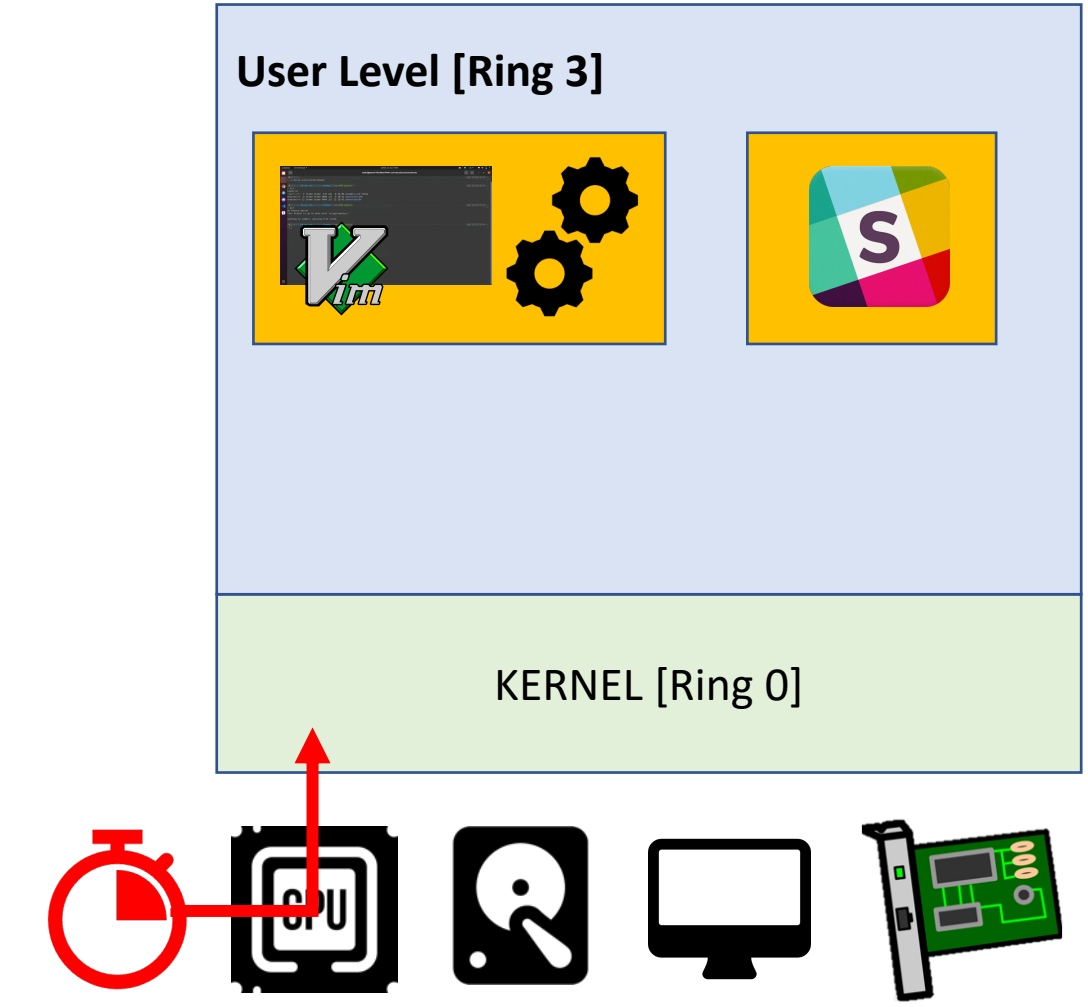
```
int main() {
    while(1);
}
```

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

Preemptive Multitasking

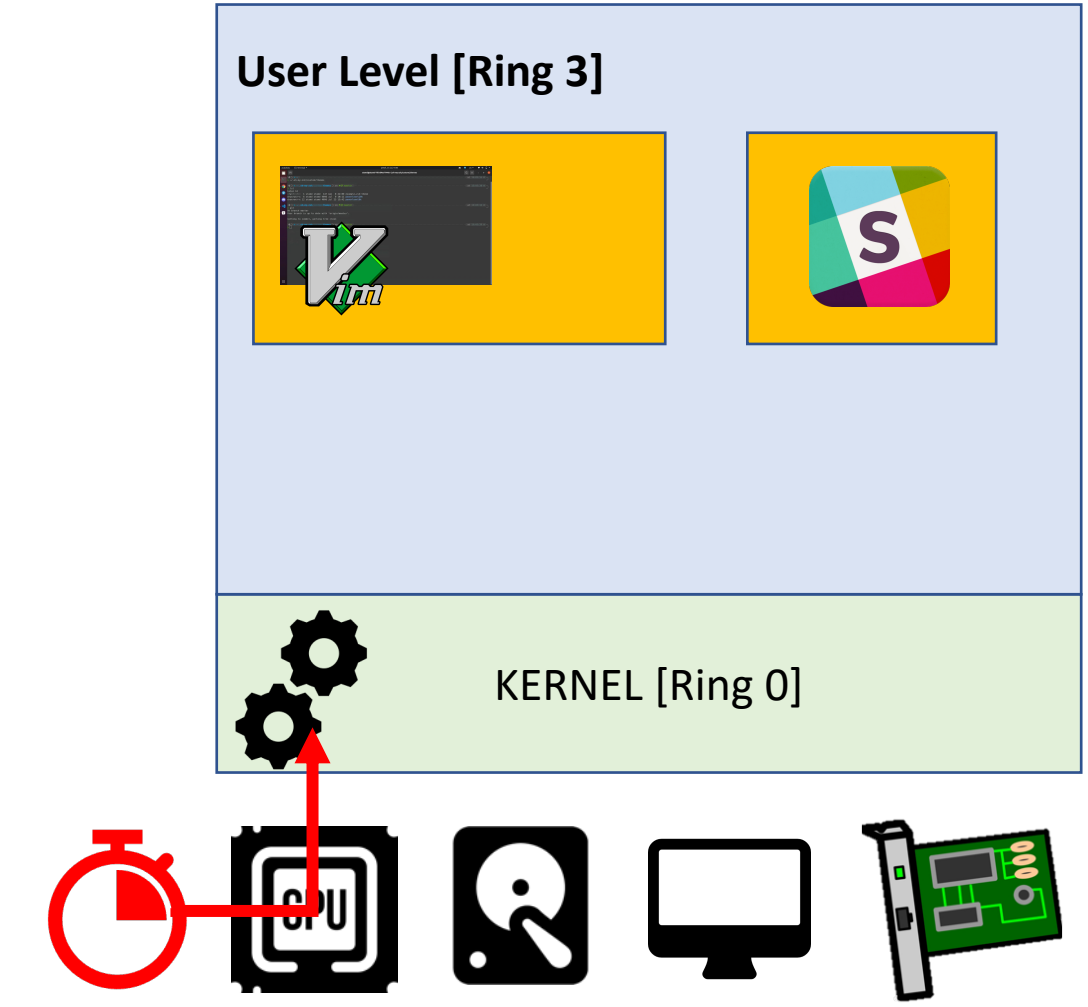


Preemptive Multitasking | Timers



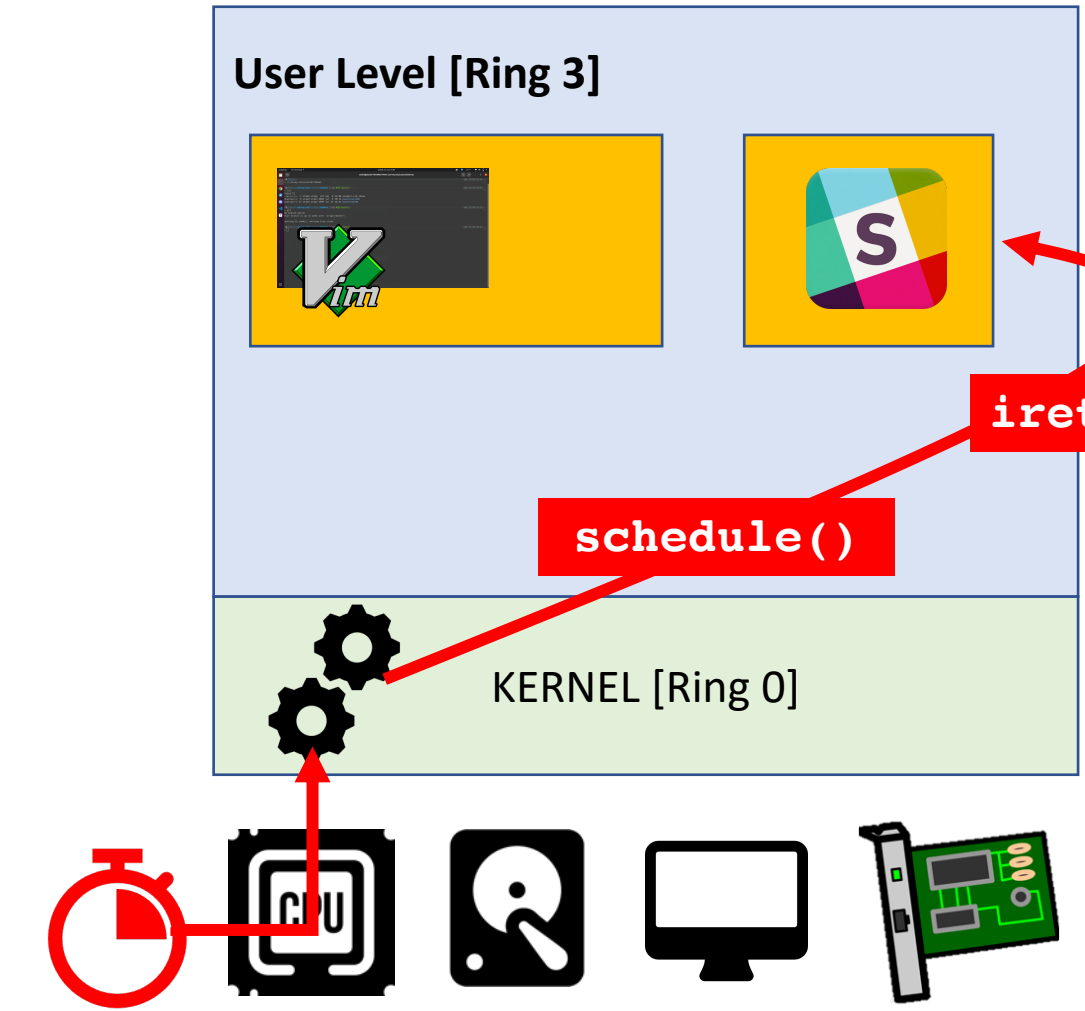
- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**

Preemptive Multitasking | Timers



- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**
- Forces **kernel execution at regular intervals**
- E.g., every 1000 Hz [1 ms]

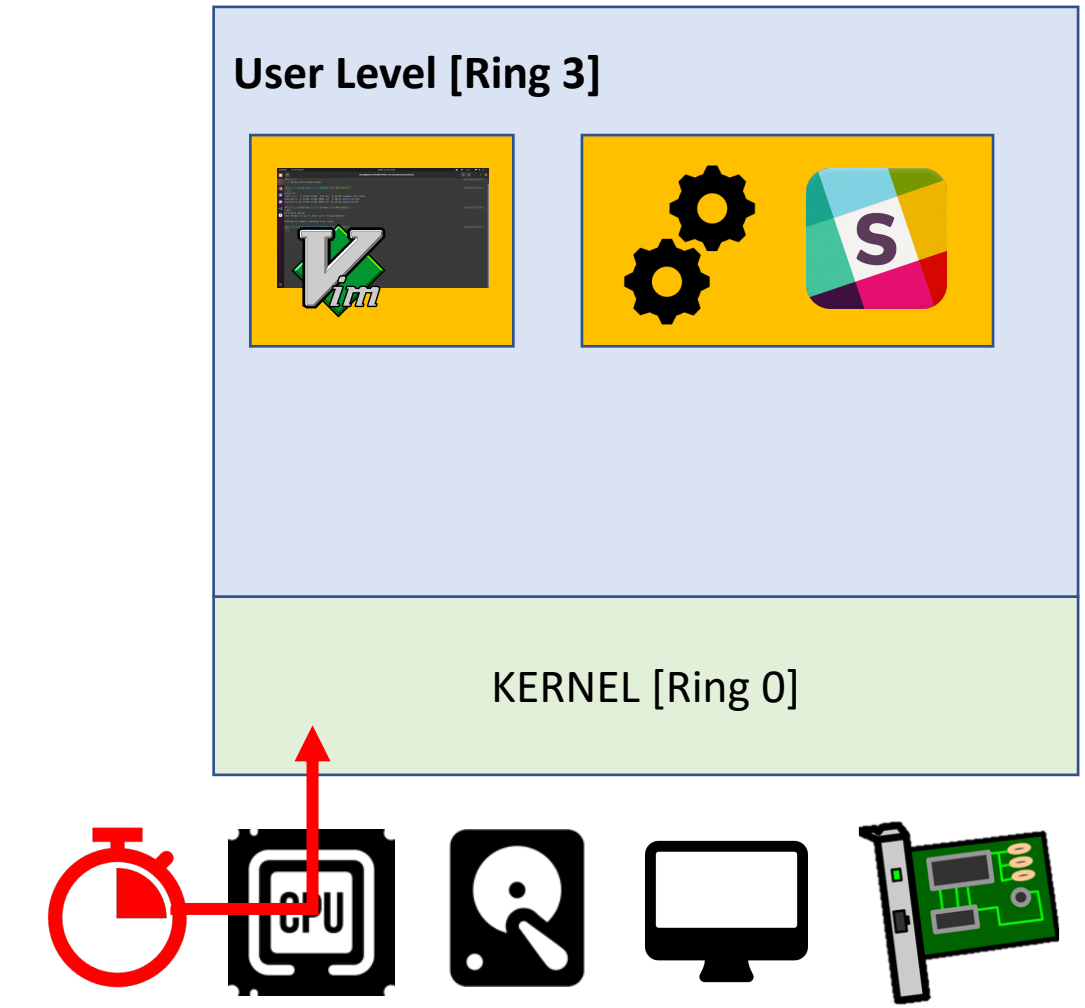
Preemptive Multitasking | Timers



- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**
- Forces **kernel execution at regular intervals**
- E.g., every 1000 Hz [1 ms]

- Kernel then makes **scheduling decisions**
  - and mediates other resources

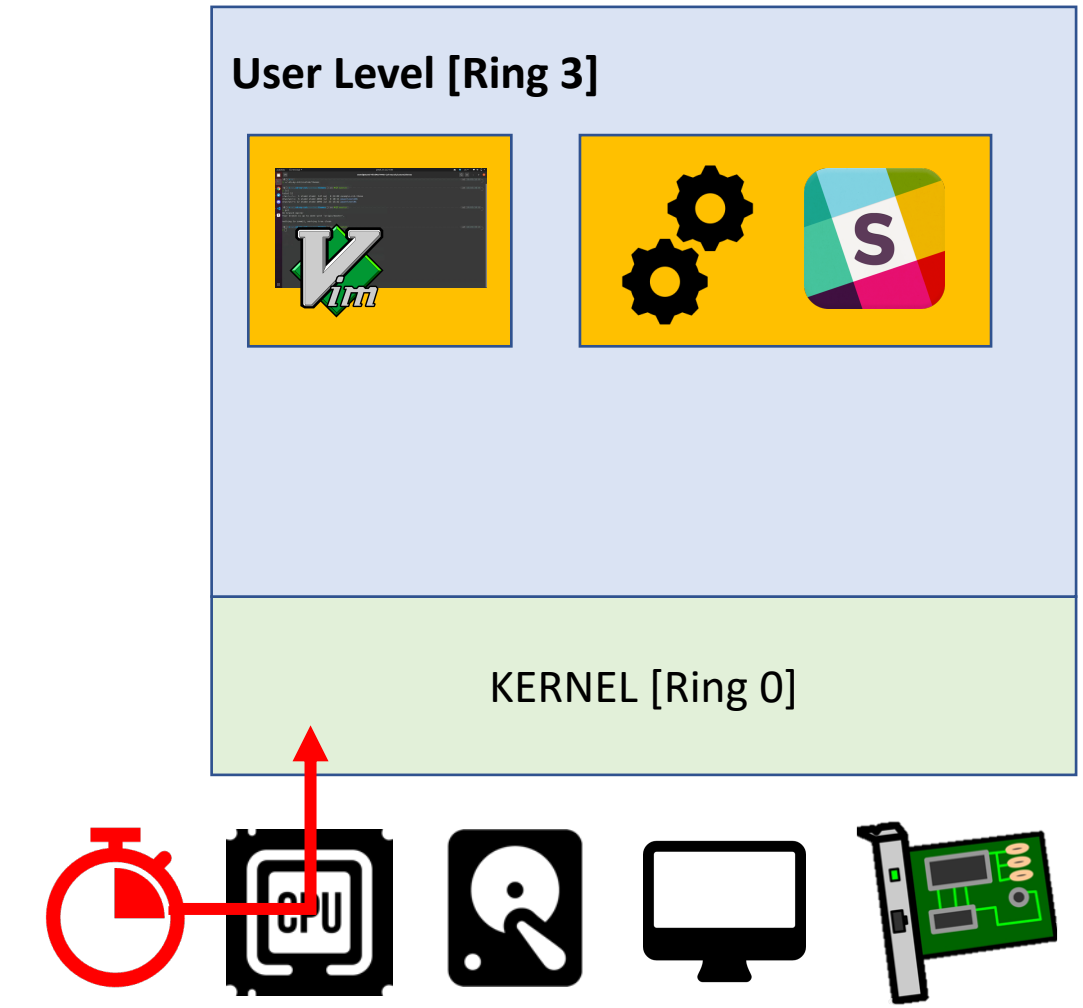
Preemptive Multitasking | Timers



- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**
- Forces **kernel execution at regular intervals**
- E.g., every 1000 Hz [1 ms]

- Kernel then makes **scheduling decisions**
  - and mediates other resources

Preemptive Multitasking | Timers

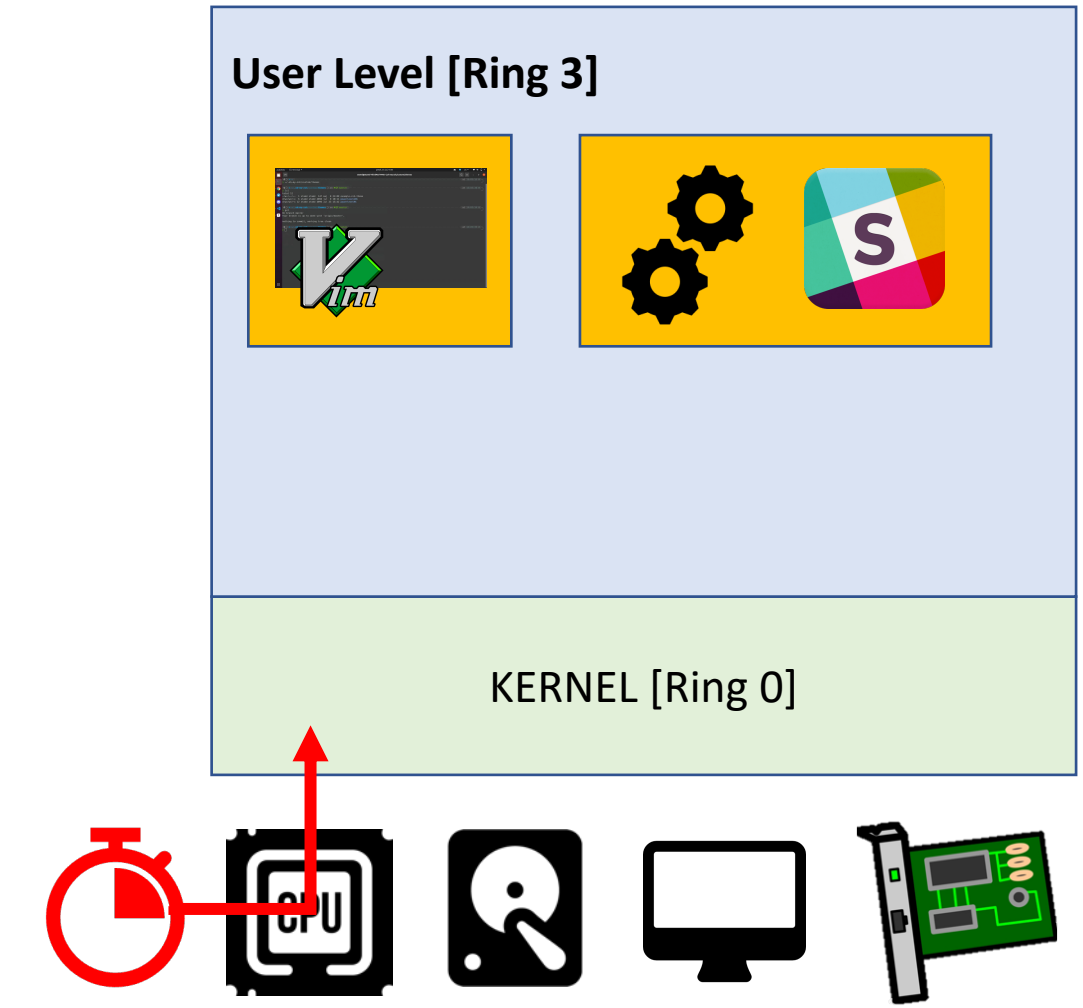


- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**
- Forces **kernel execution at** **regular intervals**
- E.g., every 1000 Hz [1 ms]

- Kernel then makes **scheduling decisions**
  - and mediates other resources

time quantum

Preemptive Multitasking | Timers



- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**
- Forces **kernel execution at regular intervals**
- E.g., every 1000 Hz [1 ms]

- Kernel then makes **scheduling decisions**
  - and mediates other resources

time quantum

- **Timer guarantees execution in kernel**

How are Popular OSes doing?

Operating System	Preemption
Amiga OS	Yes
FreeBSD	Yes
Linux kernel before 2.6.0	Yes
Linux kernel 2.6.0–2.6.23	Yes
Linux kernel after 2.6.23	Yes
classic Mac OS pre-9	None
Mac OS 9	Some
macOS	Yes
NetBSD	Yes
Solaris	Yes
Windows 3.1x	None
Windows 95, 98, Me	Half
Windows NT (including 2000, XP, Vista, 7, and Server)	Yes

From Wikipedia

IT'S A TRAP!

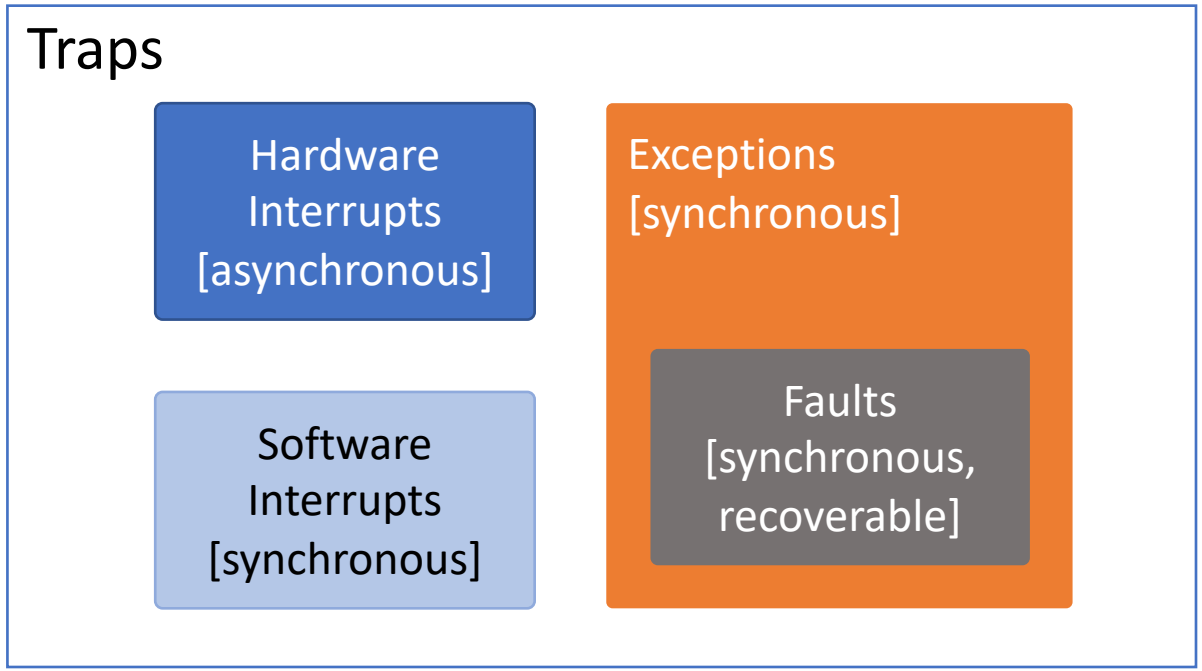
Traps

- Any event that forces CPU to stop and **execute kernel code**
- **trap handler**

Types of Traps

Interrupts	Faults	Exceptions
• **Hardware interrupt** [clock timer, network packet, etc.] • **Software interrupt** [System calls]	• An error that OS **can recover from** and continue execution [e.g., page fault]	• An error that **OS cannot recover from** • must stop the current execution [e.g., divide by zero]

Many others, please refer to the Intel Manual Chapter 6 (https://os.unexploitable.systems/r/ia32/IA32-3A.pdf)


Traps
Hardware Interrupts [asynchronous]
Software Interrupts [synchronous]
Exceptions [synchronous]
Faults [synchronous, recoverable]

Hardware Interrupt

- Method for hardware to **interact** with CPU
- Example: a network device
  - NIC: *"Hey, CPU, I received a new packet, so wake up the OS to handle it"*
  - CPU: calls the **interrupt handler** for network device in ring 0 [set by the OS/driver]

- **Asynchronous** [can happen any time during execution]
  - It's a request from a hardware, so can happen any time

- Read
  - https://en.wikipedia.org/wiki/Intel_8259
  - https://en.wikipedia.org/wiki/Advanced_Programmable_Interrupt_Controller

Software Interrupt

- A piece of **software** mean to run code in ring 0 [e.g., `int $0x30`]
- Tells CPU, "*run the interrupt handler at 0x30*"

- **Synchronous** [caused by running an instruction, e.g., `int $0x30`]

- E.g.
  - System calls [int $0x30 → system call in JOS]
  - Signals in UNIX/Linux [SIGSEGV, SIGKILL, etc.]

Exceptions/Faults

- **Exceptions**
  - Error caused by the current execution [may or may not be recoverable]
  - Examples of non-recoverable exception [**cannot continue the execution**]
    - Triple fault
    - Divide by zero
    - Breakpoint
- **Fault**
  - An error caused by current execution that may be **recoverable** so **execution can continue**
  - Examples
    - Page fault
    - Double fault

- **Synchronous** [an execution of an instruction can generate this]
  - E.g., divide by 0

Handling Interrupt/Exceptions

- **Interrupt Descriptor Table [IDT]**

Interrupt Number	Code address
0 (Divide error)	0xf0130304
1 (Debug)	0xf0153333
2 (NMI, Non-maskable Interrupt)	0xf0183273
3 (Breakpoint)	0xf0223933
4 (Overflow)	0xf0333333
…	
8 (Double Fault)	0xf0222293
…	
14 (Page Fault)	0xf0133390
…	…
0x30 (syscall in JOS)	0xf0222222

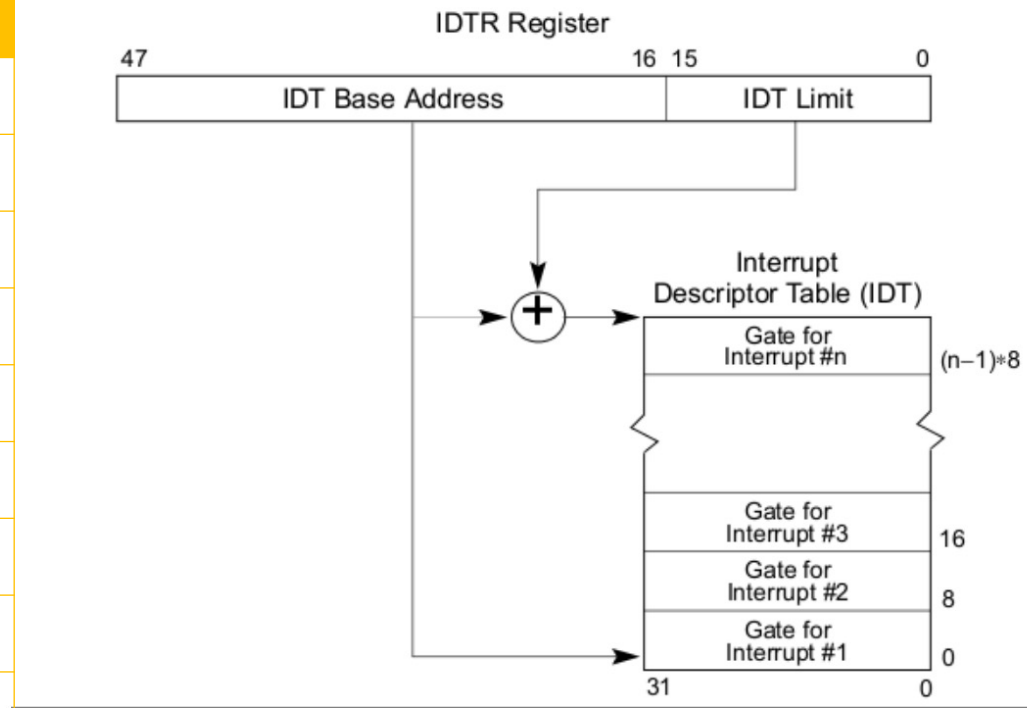
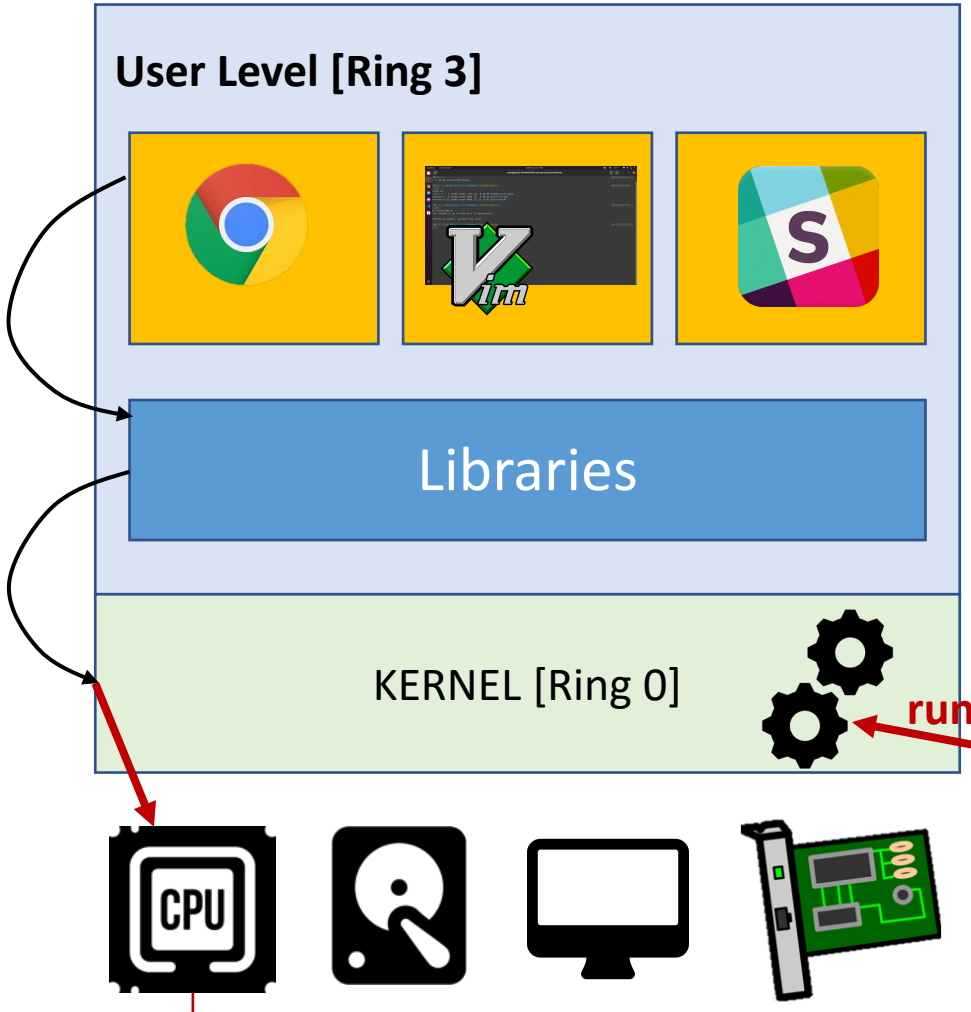


Figure 6-1. Relationship of the IDTR and IDT

Opening a file

`open(filename)`

Interrupt ox30 [syscall in JOS]



Interrupt Number	Code address
0 (Divide error)	0xf0130304
1 (Debug)	0xf0153333
2 (NMI, Non-maskable Interrupt)	0xf0183273
3 (Breakpoint)	0xf0223933
4 (Overflow)	0xf0333333
…	
8 (Double Fault)	0xf0222293
…	
14 (Page Fault)	0xf0133390
…	…
0x30 (syscall in JOS)	0xf0222222

What the kernel does [for open()]

Access **arguments** from Ring 3

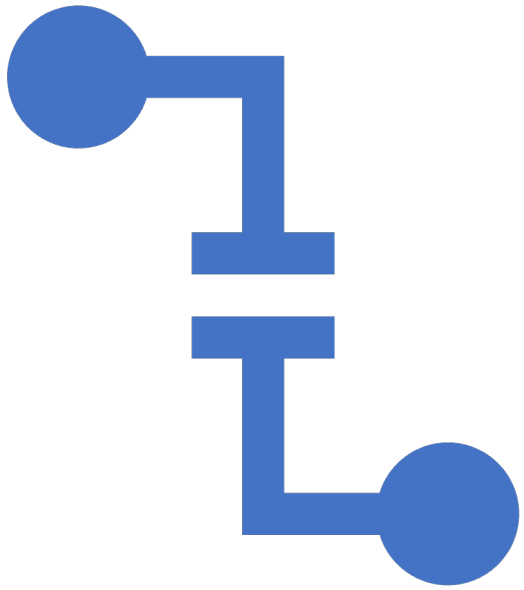
Need to check its **security**

Access disk to open a file

Check **permissions**

Return a file descriptor

`iret`



Summary

- A user program can invoke a system call
  - to ‘request’ OS to run code at a **higher privileged level** [ring 0]
  - System calls [synchronous interrupt]
- A hardware informs the CPU that data is ready for the OS
  - Hardware interrupt [asynchronous interrupt]
- A program generates an unrecoverable error [e.g. a triple fault]
  - A non-recoverable exception, synchronous
- A program generates a page fault
  - Fault [recoverable, synchronous error]
  - (we will learn more about this in coming lectures)

Additional Reading

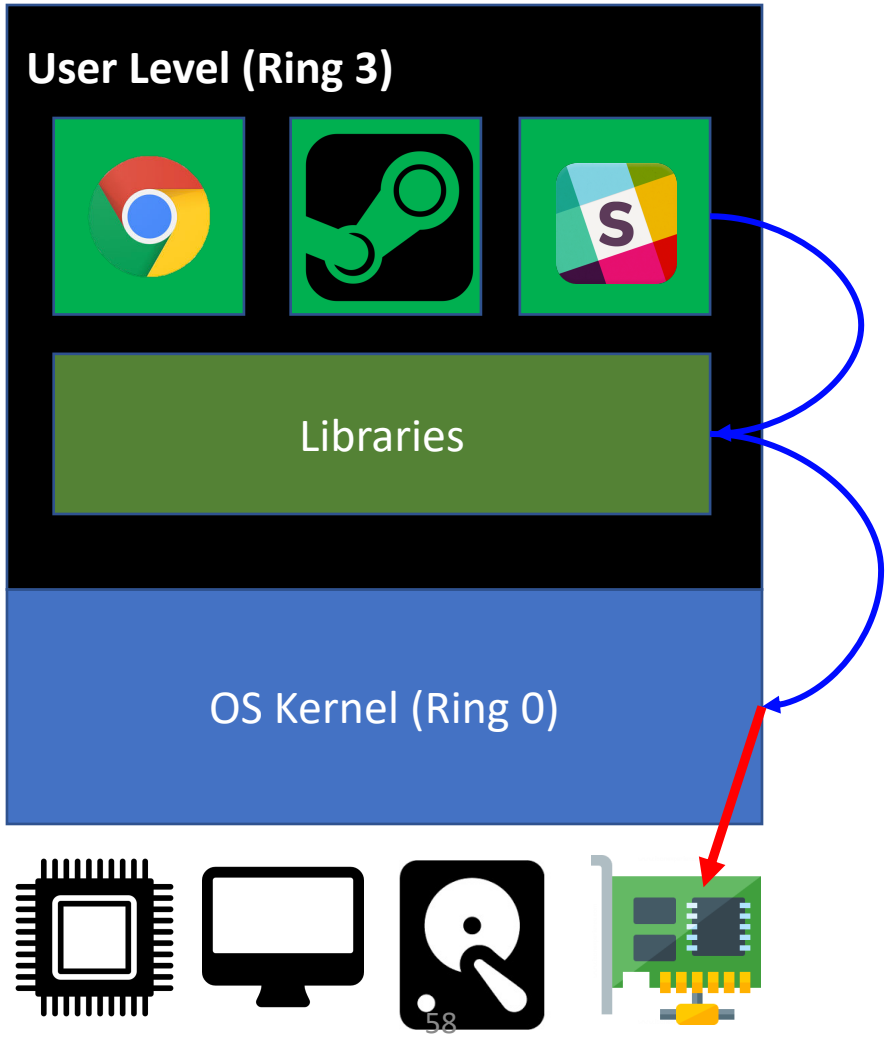
- Types of traps:
  - Intel manual Chapter 6
  - https://os.unexploitable.systems/r/ia32/IA32-3A.pdf

- Hardware Interrupts
  - https://en.wikipedia.org/wiki/Intel_8259
  - https://en.wikipedia.org/wiki/Advanced_Programmable_Interrupt_Controller

Backup Slides

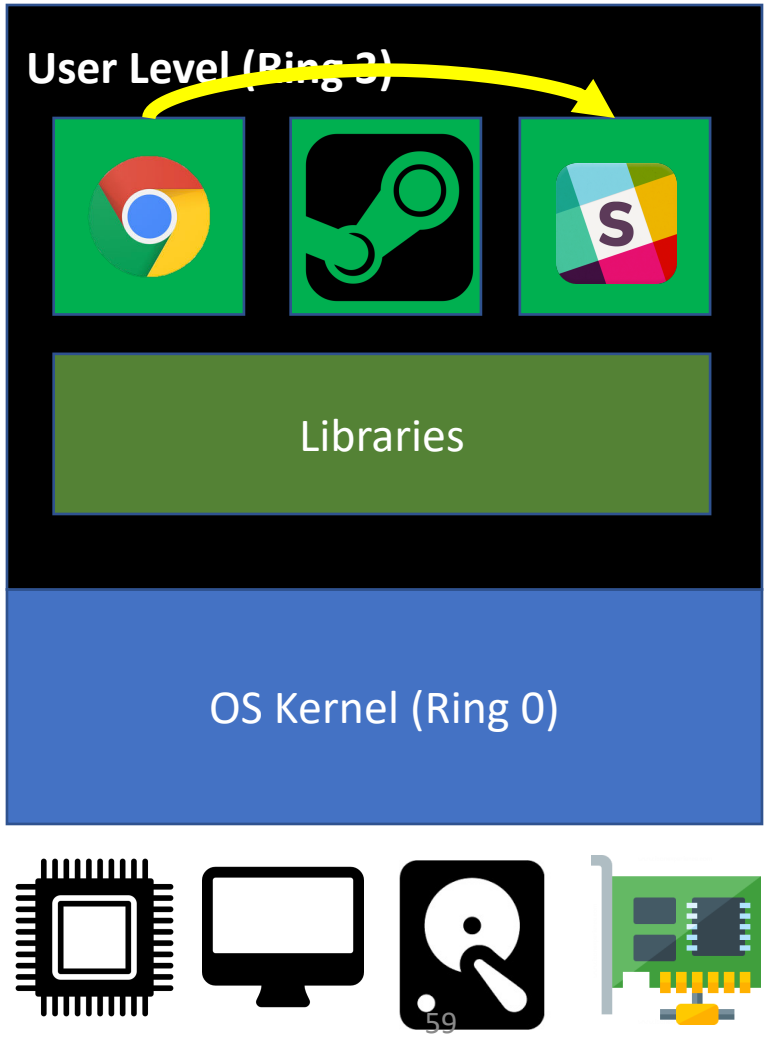
User/Kernel Switch

- User/Kernel Space Switch
  - How the OS kernel run a program in Ring 3 (user level)?
  - How the OS kernel takes back the execution to Ring 0 (kernel)?
- System call
  - How a user level program can let OS do a service for them?



Process Context Switch

- Process Context Switch
  - How our CPU can run multiple applications at the same time?
- 3 design candidates
  - Not switching
  - Co-operative Multitasking
  - Preemptive Multitasking

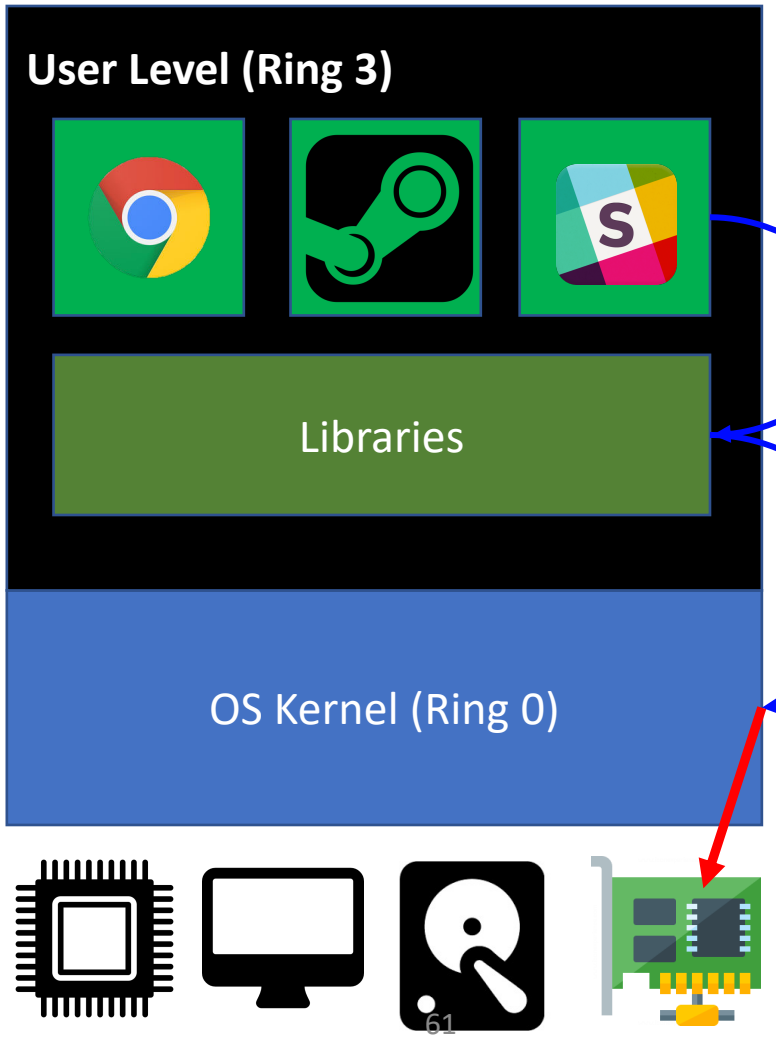


User/Kernel Switch

- Interrupt
- System calls
- Fault / Exceptions

A High-level Overview of User/Kernel Execution

```c
int main() {
    send(4, "I have a question...", 30, 0);
}
```



send() A library call in ring 3

sys_send() A system call,
From ring 3 to ring 0

Interrupt!

do_sys_send()

A High-level Overview of User/Kernel Execution

```
int main() {
    send(4, "I have a question...", 30, 0);
}
```