

# CS614: Linux Kernel Programming

## Logistics, Introduction

Debadatta Mishra, CSE, IIT Kanpur

# Course and instructors

## \$whereis cs614

Mon, Wed 5PM - 6.15PM at KD-102

<https://www.cse.iitk.ac.in/users/deba/cs614/>

Piazza link: <https://piazza.com/iitk.ac.in/secondsemester2024/cs614>

Canvas: <https://canvas.cse.iitk.ac.in/login>

## \$whereis deba

KD 212, [deba@cse.iitk.ac.in](mailto:deba@cse.iitk.ac.in), Meeting hours: 3PM - 5PM Thursday

## \$ ls TAs

Shiv ([shivbt@cse.iitk.ac.in](mailto:shivbt@cse.iitk.ac.in)), Nibir ([nibir@cse.iitk.ac.in](mailto:nibir@cse.iitk.ac.in))

# Course policy

## Add/Drop

Course drop:- Ideally before 24th Jan 2024

## Class guidelines

- All communications will be in Piazza
- Bring your laptop with sufficient power backup to class
- Keep your mobile phones switched off / silent
- Ask questions and interact (both in class and in Piazza)
- Be on time!

# Evaluation

1. In class exercises/quizzes (40%)
2. Project (40%)
3. Mid-semester (20%)

# Evaluation

1. In class exercises/quizzes (40%)
2. Project (40%)
3. Mid-semester (20%)

## In class exercise/quizzes

- Will be done in a group of 2-3 students
- Hands-on exercise will be solved by each group
- Evaluation will be done in the class itself
- Adequate assistance will be provided
- If you are stuck, do not hesitate to contact

# Evaluation

1. In class exercises/quizzes (40%)
2. Project (40%)
3. Mid-semester (20%)

## Project

- Will be done in a group of 4-5 students – form your groups by 25th Jan
- Project topics will be provided, you can also suggest!
- Evaluated in multiple phases
  - Interim report – tentative timeline before mid-sem break
- Final presentation and report due during the end-sem timeline

# Evaluation

1. In class exercises/quizzes (40%)
2. Project (40%)
3. Mid-semester (20%)

## References

Operating Systems: Three Easy Pieces. Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau.

Understanding the Linux Kernel, Daniel P. Bovet, Marco Cesati.

Linux Kernel Development, 3rd Edition, Robert Love.

Linux Device Drivers, 3rd Edition, By Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini.

Linux kernel documentation, Research papers

# Evaluation

1. In class exercises/quizzes (40%)
2. Project (40%)
3. Mid-semester (20%)

“Take pride in honest hard work”

“Cheating implies accepting defeat”

“If you are here to learn, never defeat the purpose by cheating”

<https://www.cse.iitk.ac.in/pages/AntiCheatingPolicy.html>



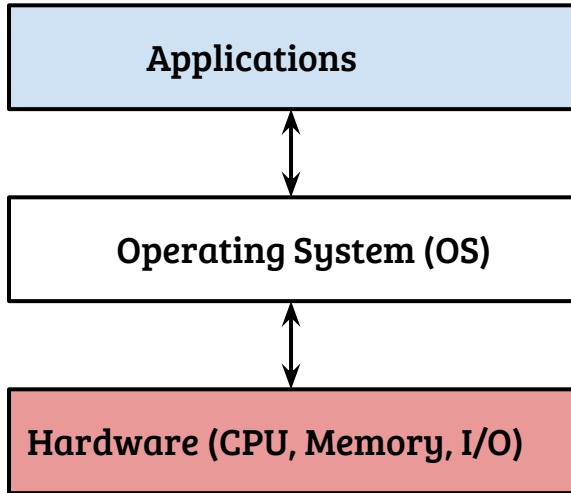
# Homework -1

- Personal laptops with decent backup desirable
- **HW1: Setup a Virtual machine for the course (Due before next lecture)**
  - Create a Linux VM (Ubuntu Linux recommended) (KVM is preferred)
  - Download the Linux kernel version - linux-6.1.4
  - Compile and boot the latest kernel

# OS refresher: True/False

1. Superuser (e.g., root user in UNIX) in a multi-tasking OS can execute all instructions provided by the hardware instruction set architecture (ISA).
2. Every process in a computer system is guaranteed to be in running state at least once during its life time.
3. A critical section consisting of a single instruction may require mutual exclusion.
4. A user process interrupted by a device interrupt is always scheduled immediately after the interrupt is serviced.
5. A page fault can be handled without changing any page table entry.

# Role of the OS



- OS bridges the *semantic gap* between the notions of application execution and real execution
  - OS loads an executable from disk to memory, allocates/frees memory dynamically
  - OS initializes the CPU state i.e., the PC and other registers
  - OS provides interfaces to access I/O devices
- OS facilitates hardware resource sharing and management (How?)

# Virtual view of resources

- Process
  - Each running process thinks that it owns the CPU

# Virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space

# Virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space
- File system tree
  - The user feels like operating on the files directly

# Virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space
- File system tree
  - The user feels like operating on the files directly
- What are the OS responsibilities in providing the above virtual notions?

# Virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space
- File system tree
  - The user feels like operating on the files directly
- What are the OS responsibilities in providing the above virtual notions?
  - The OS performs multiplexing of physical resources efficiently
  - Maintains mapping of virtual view to physical resource



# Virtualization: Efficiency/performance

- Resource virtualization should not add *excessive* overheads
- Efficient when programs use the resources directly, infrequent OS mediation
  - Example: when a process is scheduled on CPU, it should execute without OS intervention
- What is the catch?

# Virtualization: Efficiency/performance

- Resource virtualization should not add *excessive* overheads
- Efficient when programs use the resources directly, infrequent OS mediation
  - Example: when a process is scheduled on CPU, it should execute without OS intervention
- What is the catch?
  - Loss of control e.g., process running an infinite loop on a CPU
  - Isolation issues e.g., process accessing/changing OS data structures

# Virtualization: Efficiency/performance

- Resource virtualization should not add *excessive* overheads
- Efficient when programs use the resources directly, infrequent OS mediation
  - Example: when a process is scheduled on CPU, it should execute without OS intervention
- What is the catch?
  - Loss of control e.g., process running an infinite loop on a CPU
  - Isolation issues e.g., process accessing/changing OS data structures

Conclusion: Some limits to direct access must be enforced.

# Limited direct execution

- Can the OS enforce limits to an executing process by itself?

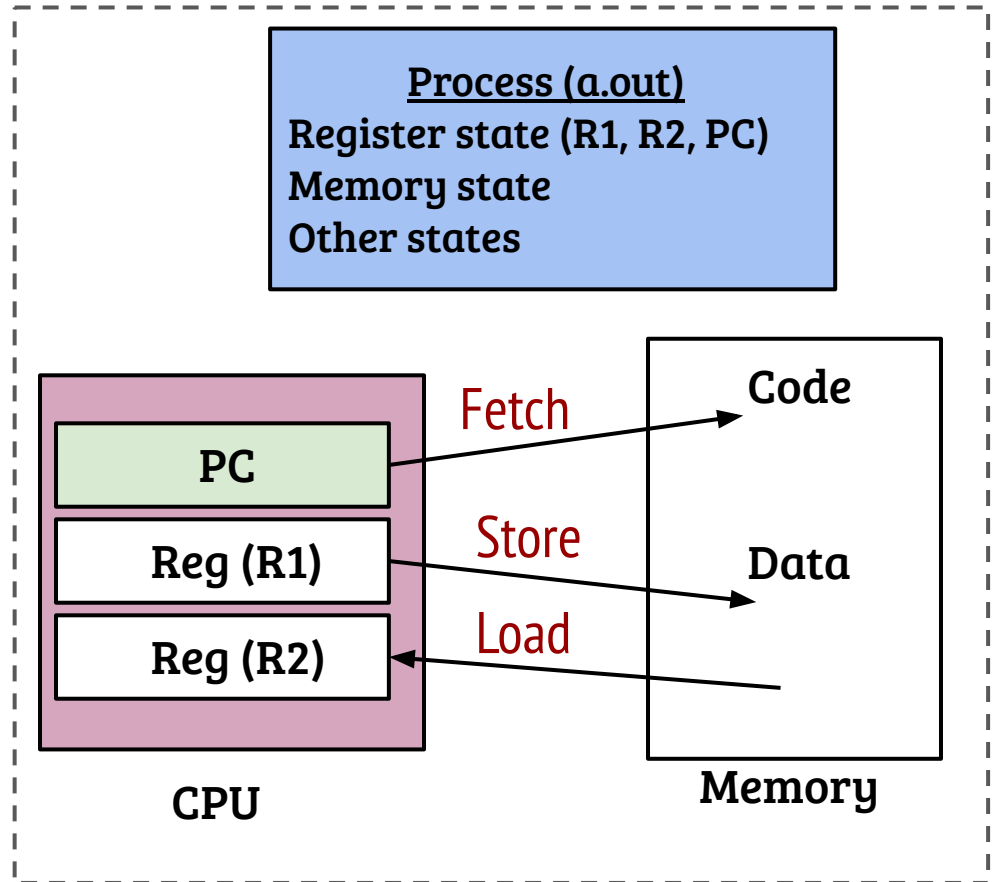
# A process in execution

I want to take control of the CPU from this process which is executing an infinite loop, but how?

OS



I want to restrict this process accessing memory of other processes, but how?  
Monitoring each memory access is not efficient!



# A process in execution

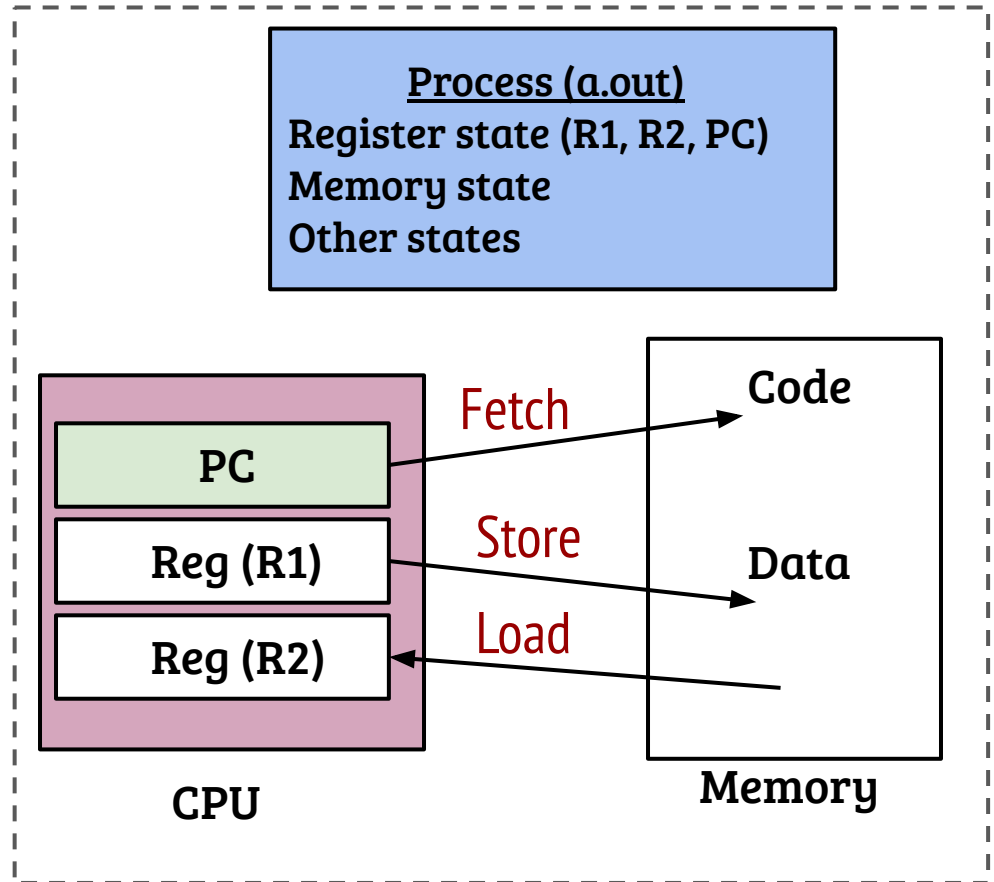
I want to take control of the CPU from this process which is executing an infinite loop, but how?

Help me!

OS



I want to restrict this process accessing memory of other processes, but how?  
Monitoring each memory access is not efficient!



# Limited direct execution

- Can the OS enforce limits to an executing process by itself?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!

# Limited direct execution

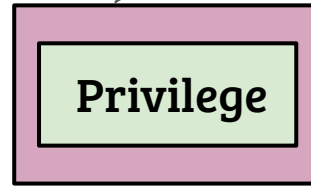
- Can the OS enforce limits to an executing process?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!
- What kind of support is needed from the hardware?



# Hardware support: Privilege levels



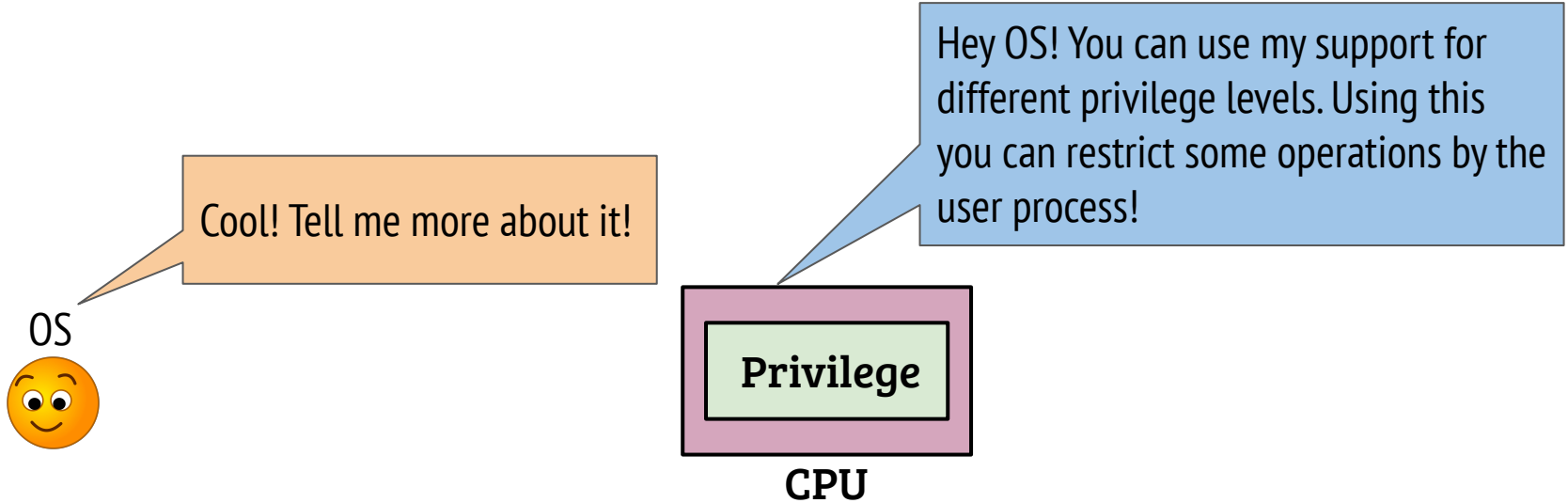
Help me!



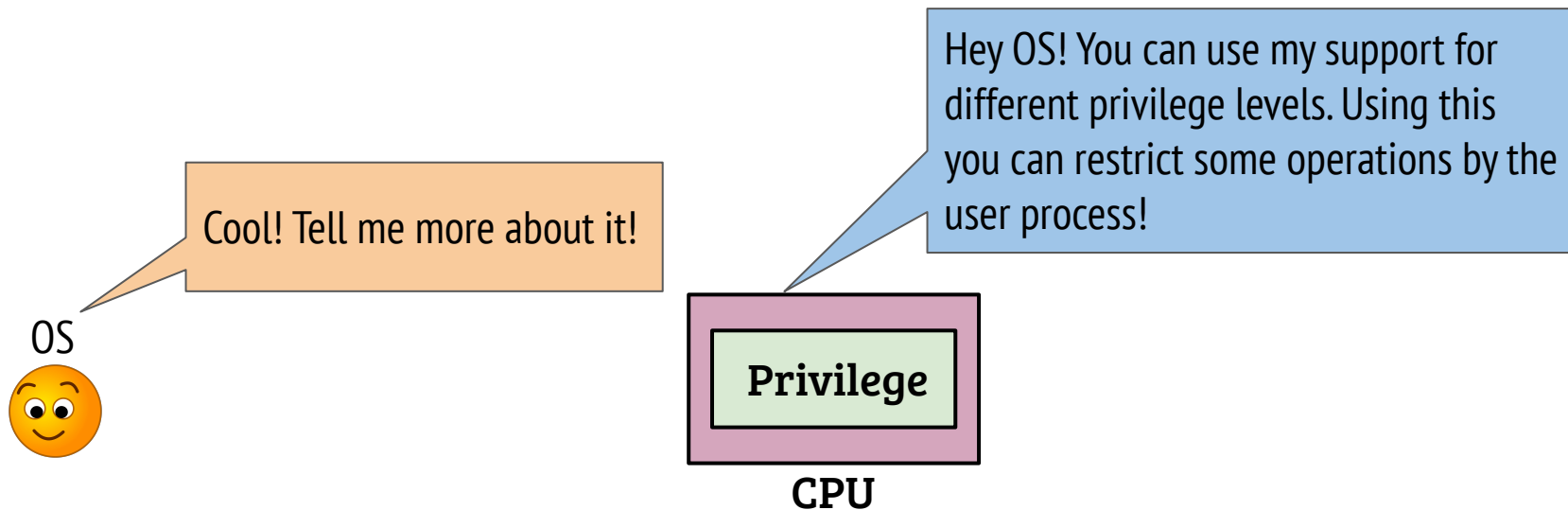
CPU

Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!

# Hardware support: Privilege levels

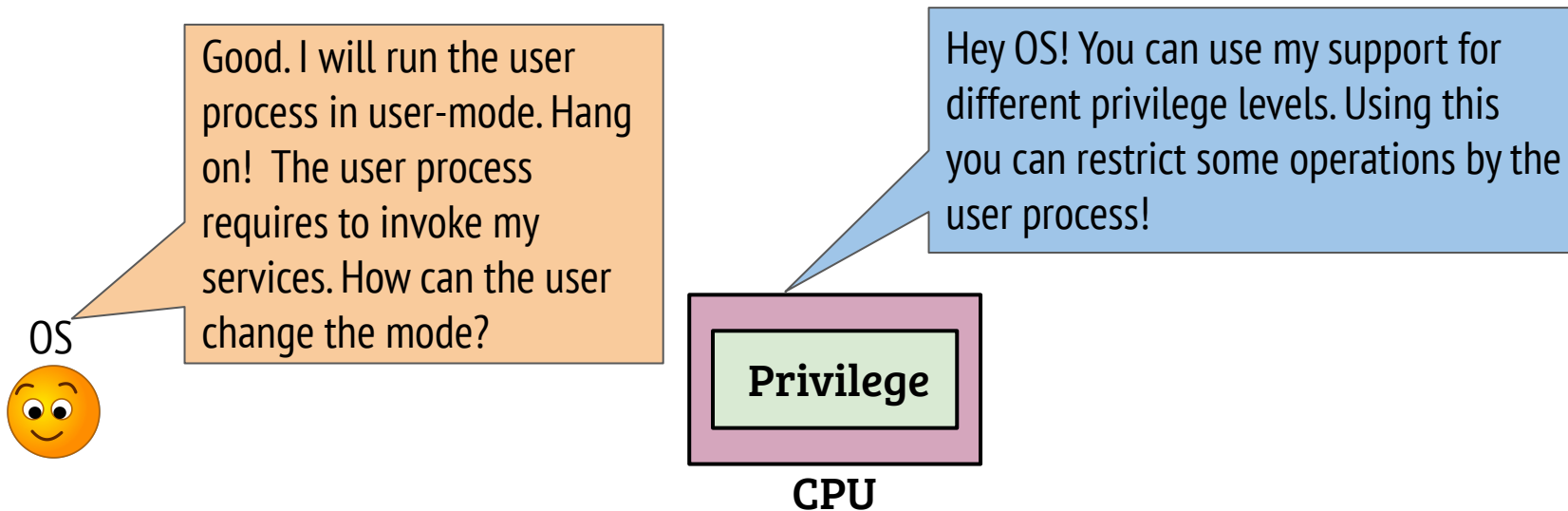


# Hardware support: Privilege levels

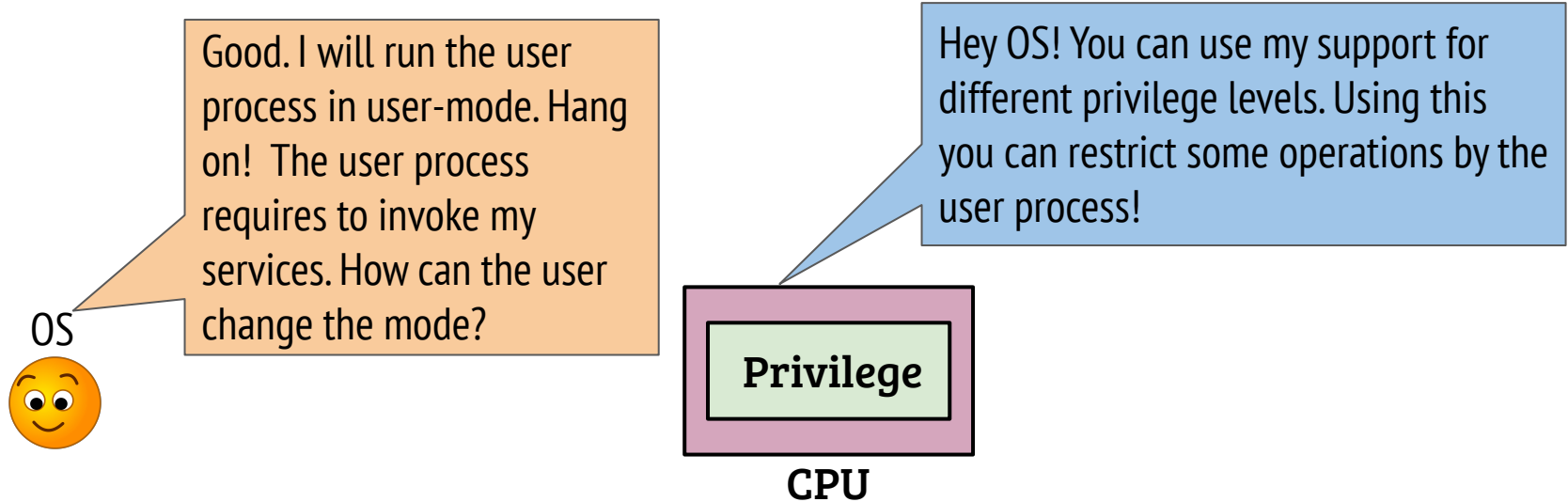


- CPU can execute in two modes: *user-mode* and *kernel-mode*
- Some operations are allowed only from kernel-mode (privileged OPs)
  - If executed from user mode, hardware will notify the OS by raising a fault/trap

# Hardware support: Privilege levels



# Hardware support: Privilege levels



- From user-mode, privilege level of CPU can not be changed directly
- The hardware provides entry instructions from the user-mode which causes a mode switch
- The OS can define the handler for different entry gates

# Hardware support: Privilege levels

OS



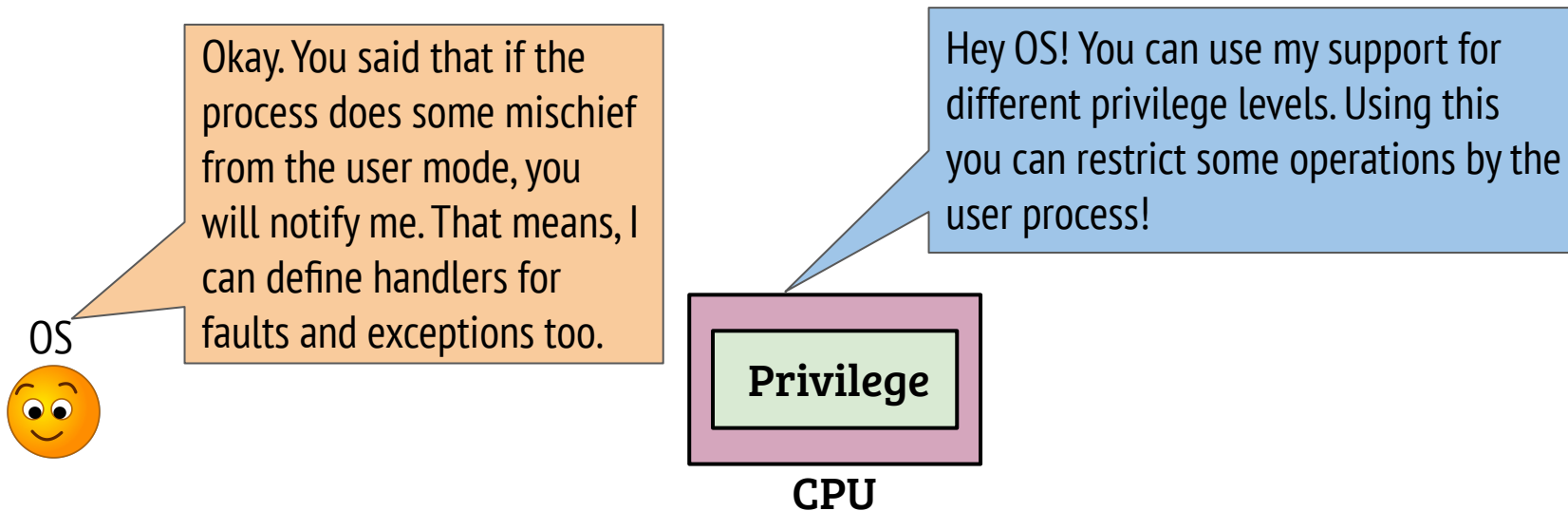
Okay. You said that if the process does some mischief from the user mode, you will notify me. That means, I can define handlers for faults and exceptions too.

Privilege

CPU

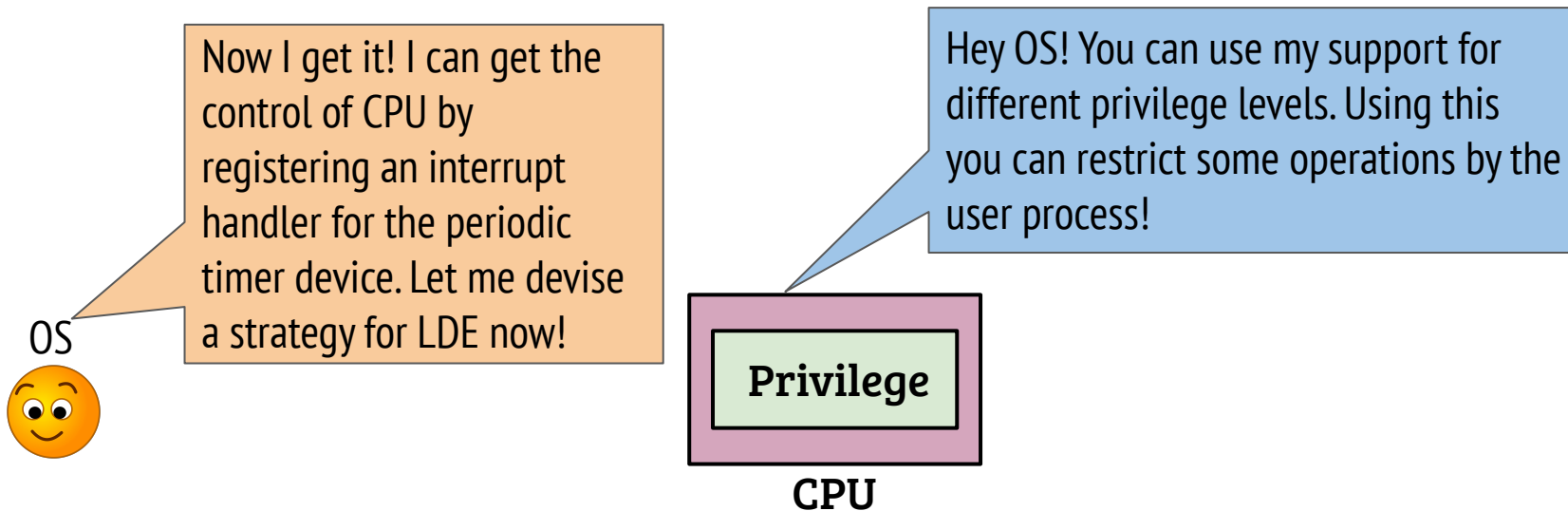
Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!

# Hardware support: Privilege levels



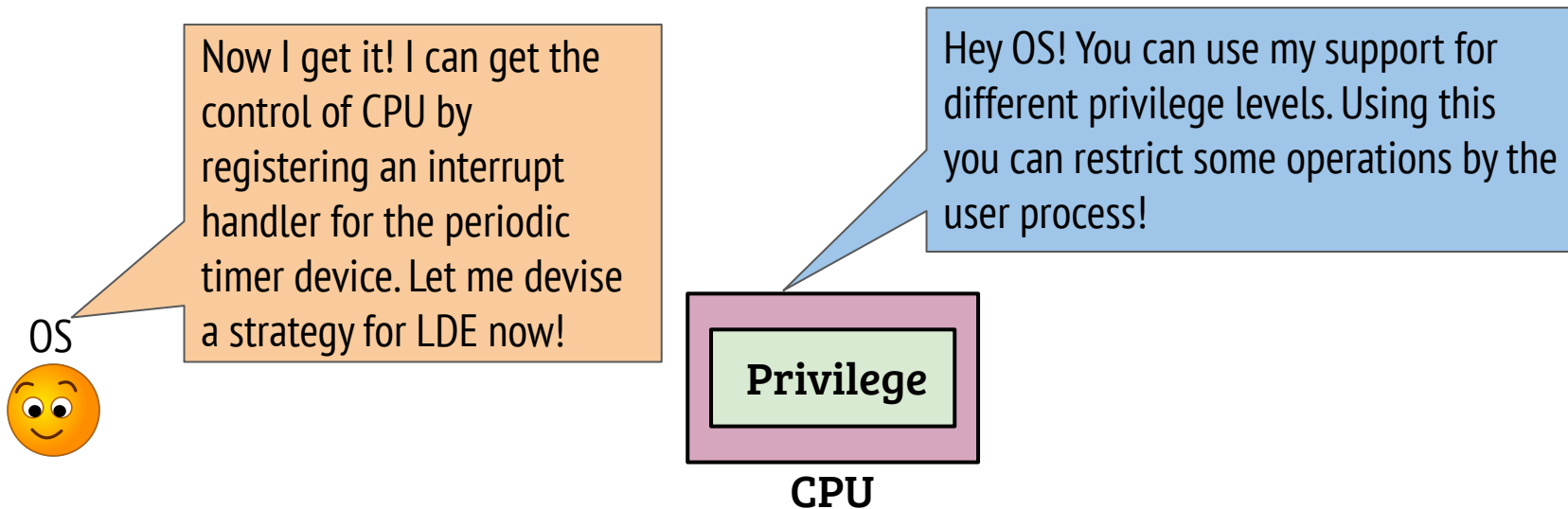
- The OS can register the handlers for faults and exceptions
- The OS can also register handlers for device interrupts
- *Registration of handlers is privileged!*

# Hardware support: Privilege levels



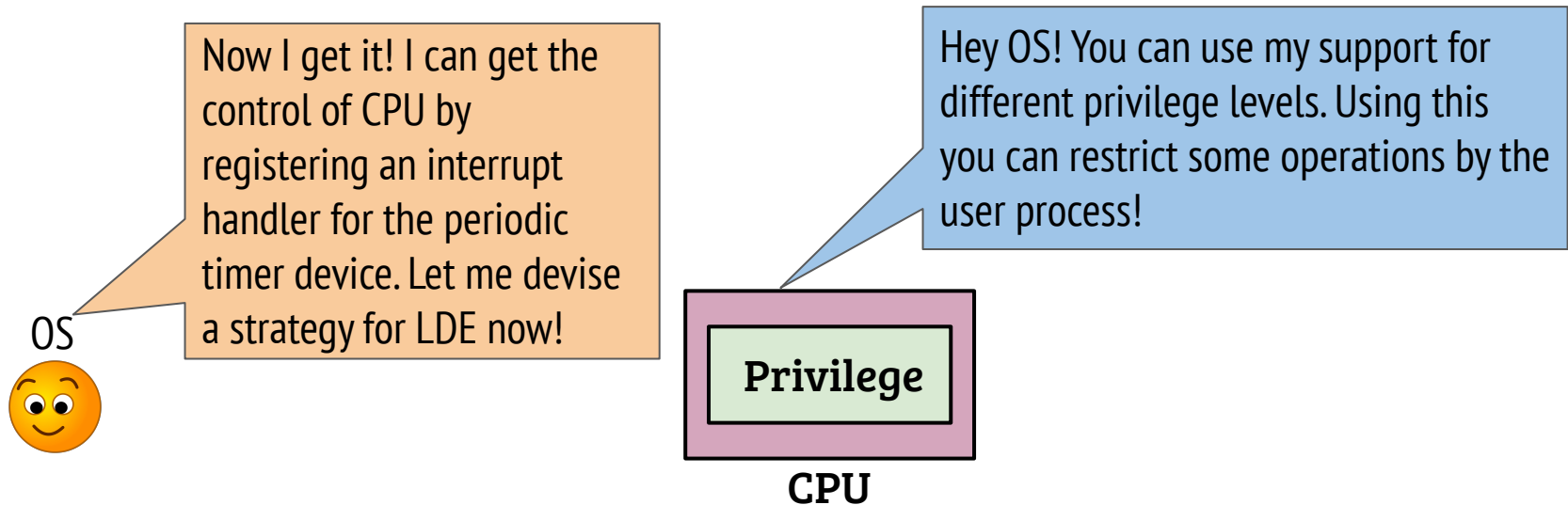


# Hardware support: Privilege levels



- After the boot, the OS needs to configure the handlers for system calls, exceptions/faults and interrupts

# Hardware support: Privilege levels



- After the boot, the OS needs to configure the handlers for system calls, exceptions/faults and interrupts
- The handler code is invoked by the OS when user-mode process invokes a system call or an exception or an external interrupt

# Limited direct execution

- Can the OS enforce limits to an executing process?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!
- What kind of support is needed from the hardware?
- CPU privilege levels: user-mode vs. kernel-mode
- Switching between modes, entry points and handlers

# Evidence based Proof of LDE!

- “Proof” is a term associated with formal world
- “Evidence based proof” is very important for this course
- Proving LDE in the Linux kernel
  - How to prove using a user program executing an infinite loop?

# Evidence based Proof of LDE!

- “Proof” is a term associated with formal world
- “Evidence based proof” is very important for this course
- Proving LDE in the Linux kernel
  - How to prove using a user program executing an infinite loop?
  - CPU usage in user mode should dominate
  - Is this evidence enough?

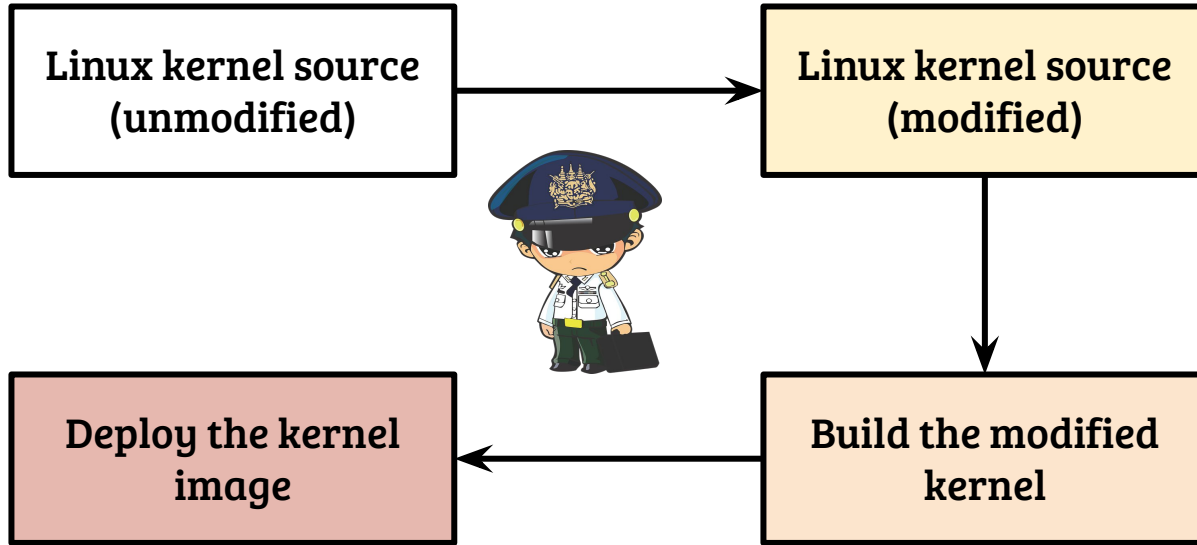
# Evidence based Proof of LDE!

- “Proof” is a term associated with formal world
- “Evidence based proof” is very important for this course
- Proving LDE in the Linux kernel
  - How to prove using a user program executing an infinite loop?
  - CPU usage in user mode should dominate
  - Is this evidence enough? OS intervention yet to be shown!
  - If a program does division by zero infinitely, we can prove OS intervention. How to go about it?

# Evidence based Proof of LDE!

- “Proof” is a term associated with formal world
- “Evidence based proof” is very important for this course
- Proving LDE in the Linux kernel
  - How to prove using a user program executing an infinite loop?
  - CPU usage in user mode should dominate
  - Is this evidence enough? OS intervention yet to be shown!
  - If a program does division by zero infinitely, we can prove OS intervention. How to go about it?
  - Handle the signal and ignore it or something better (modify division-by-zero handler in OS)  $\Rightarrow$  First hand evidence

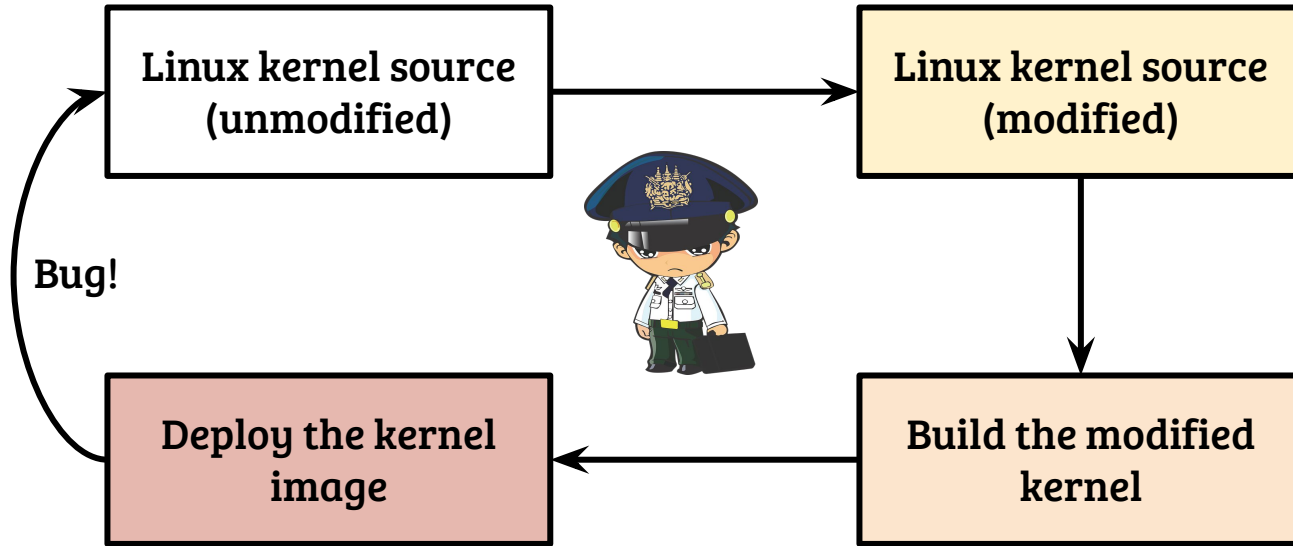
# Modifying the Linux kernel the hard way!



We are all set. Let us start then!

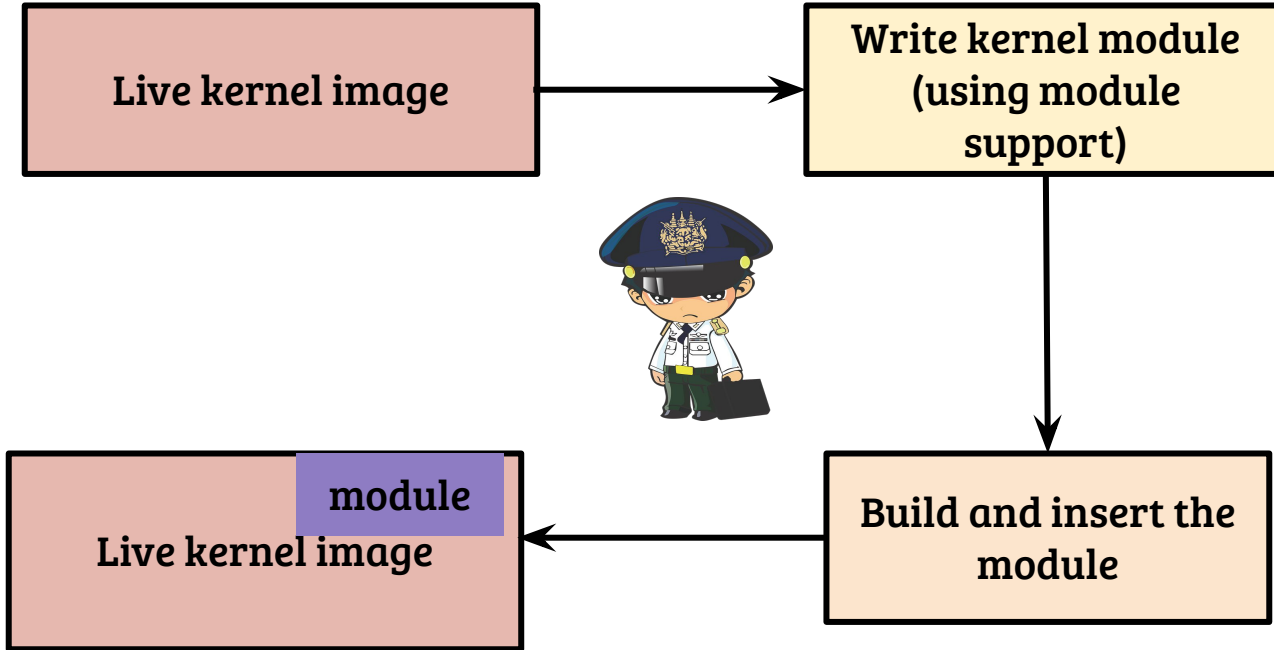


# Modifying the Linux kernel the hard way!



Hang on! What if we find a bug in our implementation?

# Modifying the Linux kernel, little easy!



We are all set then! Not really?

# A self contained module is useless!

Can we invoke existing kernel function?

Can we write buggy code and not crash the system?



Can we access kernel data structures?

How can the user interact with the module?

Can we modify kernel functionality?

# A self sufficient module is useless!

NO :(

Can we write buggy code and not crash the system?

Can we invoke existing kernel function?

Yes, if exposed.

Can we access kernel data structures?

Yes, if exposed.



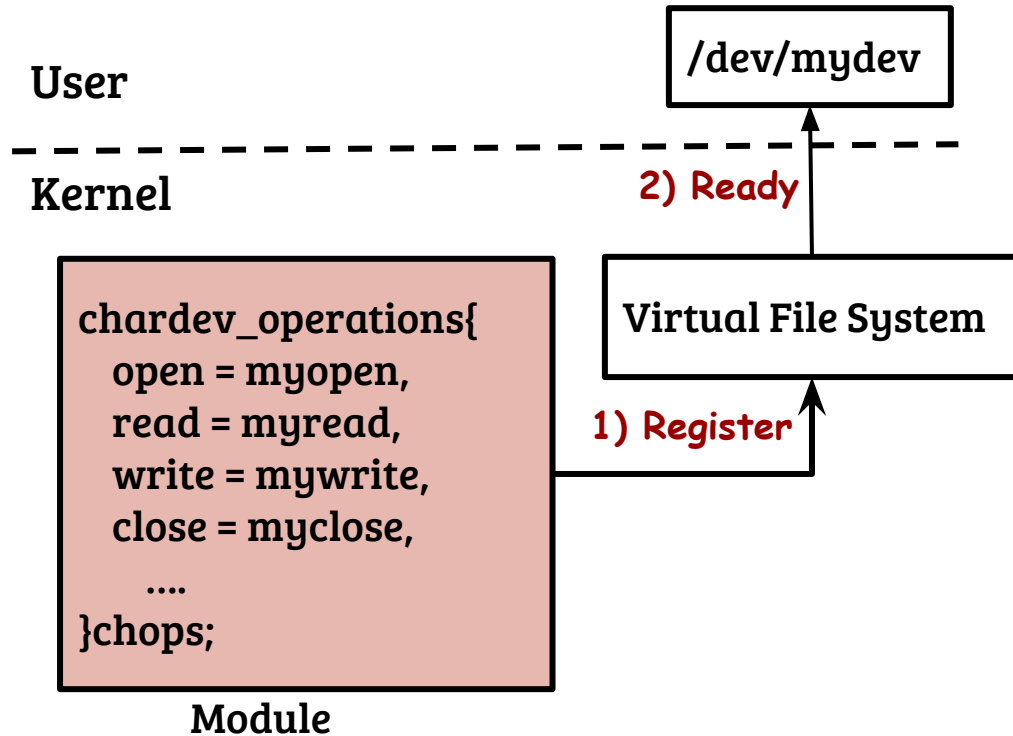
How can the user interact with the module?

Yes, several methods exist

Can we modify kernel functionality?

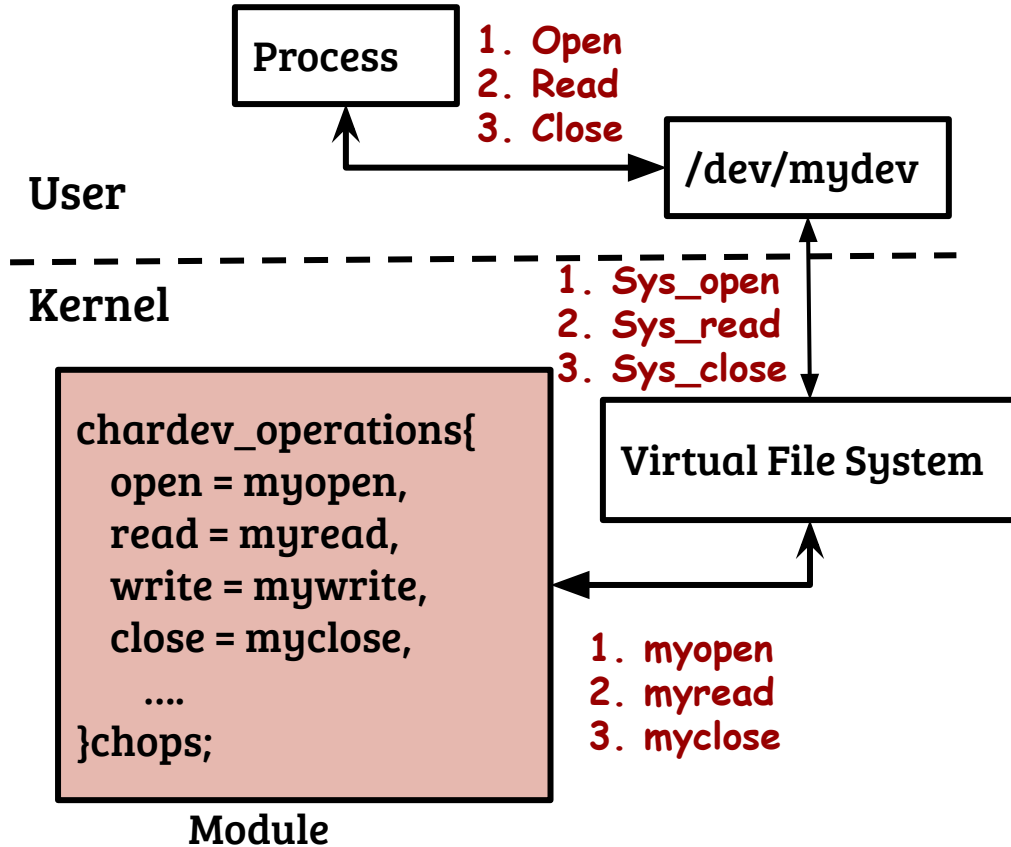
Yes, several methods exist

# User kernel interfacing with chardev



- Module must provide methods to handle operations on the chardev
- Everything is a file in Linux (Unix), Thanks to Richie and Thompson!
- Now the device is ready to be used!  
But how?

# User kernel interfacing with chardev



- User process can use system calls like `open()`, `read()` etc.
- Just like normal file read/write, data can be passed to and from the device
- The module implementation determines the exact behavior of system calls

# User kernel interfacing (with sysfs)

- Kernel creates virtual file systems such as procfs and sysfs
- User can operate on files hosted in sysfs using normal file system calls
- How to communicate between user and kernel using sysfs?

# User kernel interfacing (with sysfs)

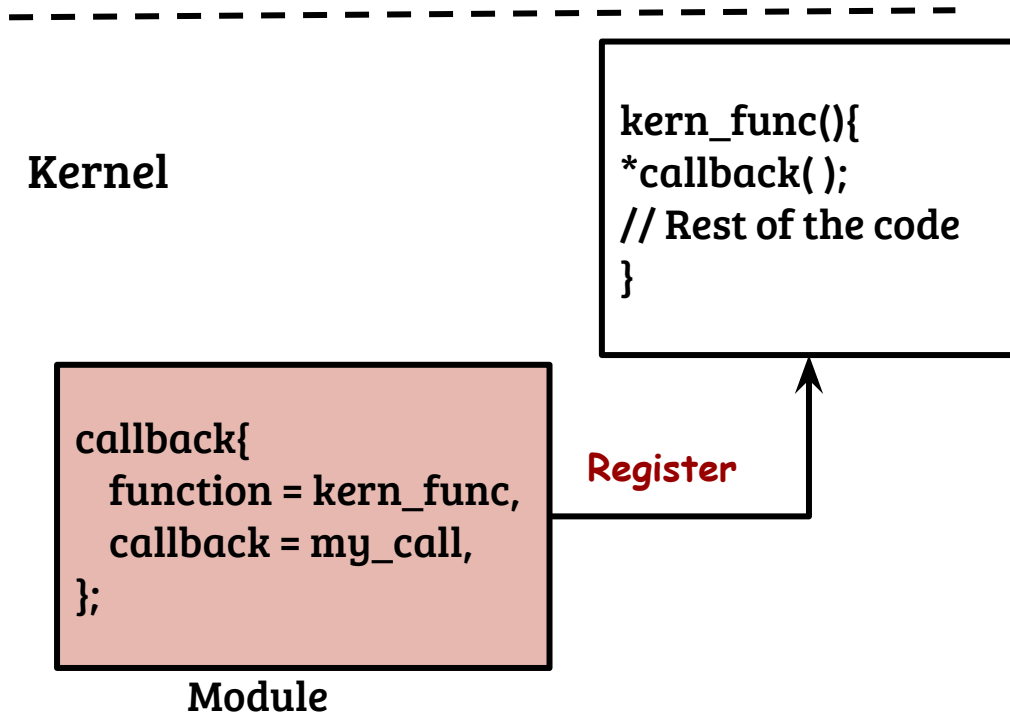
- Kernel creates virtual file systems such as procfs and sysfs
- User can operate on files hosted in sysfs using normal file system calls
- How to communicate between user and kernel using sysfs?
  - Create sysfs entries from the kernel module
  - Provide read/write handlers
  - When user performs read or write to the file, the handlers are invoked in the kernel (similar to chardev)
  - Note: Cleanup sysfs entries on module unload



# Modifying kernel functionality (explicit hooks)

- This strategy requires one-time modification of the kernel
  - Hijack the execution control back to a module
  - Incorporate the modified logic
- Notes
  - Be sure about the parameters you are passing and the return handling in the original kernel implementation
  - Handler registration and de-registration should be done carefully

# Modifying kernel functionality (kernel probe)



- Callback is invoked when `kern_func()` is executed
- After returning from the callback, the rest of the code executes normally
- Documentation/trace/kprobes.rst has further details