CS 170 Week 2

Winter 2026

https://sites.cs.ucsb.edu/~rich/class/cs170/

Agenda

1. Lab Logistics (Partners + UCSB GitHub Enterprise)
2. KOS (Lab1) Overview
3. Questions/Debug Time

Lab Logistics

Instructions: https://sites.cs.ucsb.edu/~rich/class/cs170/discussion.html

1. Source code MUST be hosted on UCSB GitHub Enterprise (join here)
2. The repositories MUST be **PRIVATE**
3. Fill out the partner and repository sign-up sheet: https://forms.gle/3tmQXpUhBQD9B1jeA
4. Add the TA's as read-only collaborators (our github handles are in the sign-up sheet)

Lab Logistics

Submission:

- Us TA's are still working out with Rich how to pull the submission data in the best way. Stay tuned.

In general: Make sure all your code is in your UCSB repo. Continue to submit to Gradescope as normal.

The autograder scores **are not your final scores.** We will be using a rubric to grade each submission.

- Does it build? (10pts)
- Implementation quality (30pts)
- Success of final tests (60pts)

Specifics will be posted once we start grading...

KOS

KOS

The goal of Labs 1 to 3 is to build an operating system called KOS (can be pronounced “Chaos”).

Roadmap:

- **Lab 1 ← (We are here)**
- Lab 2
- Lab 3

KOS Overview

KOS is a “real” OS (with a simulator for *practical reasons*).

- MIPS R3000 is simpler than modern x86/x86-64 machines.
- Avoids “low-level” boot process and simplifies exception/interrupt handling (which involves writing/understanding assembly code)
- Allows easy debugging (via gdb, print statements, etc)

KOS-simulator interface can be “easily” modified to run on physical hardware if we have the real machine (MIPS R3000 processor). Treat this as a REAL OS!

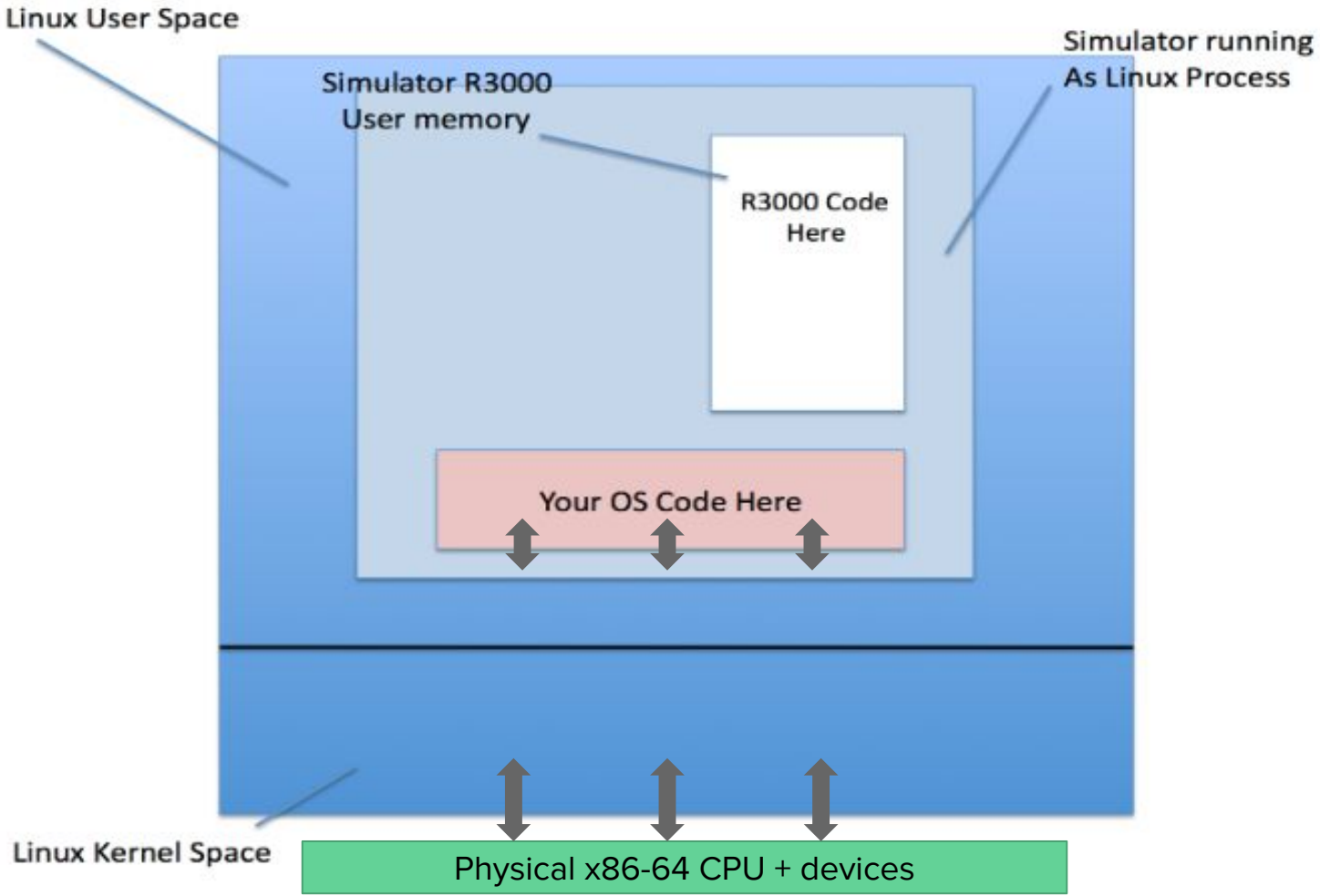


Image: https://sites.cs.ucsb.edu/~rich/class/cs170/notes/IntroKOS/index.html

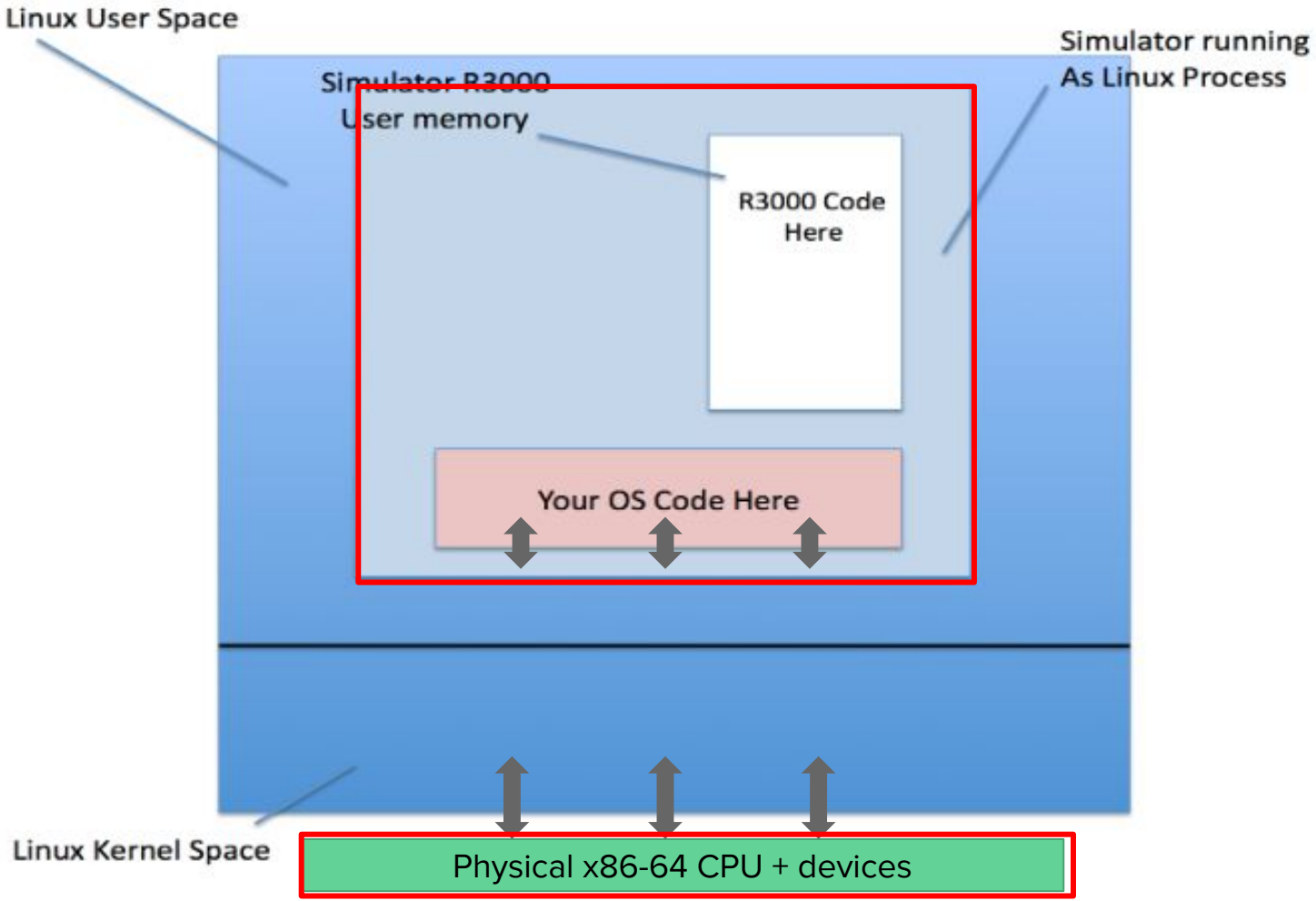


Image: https://sites.cs.ucsb.edu/~rich/class/cs170/notes/IntroKOS/index.html

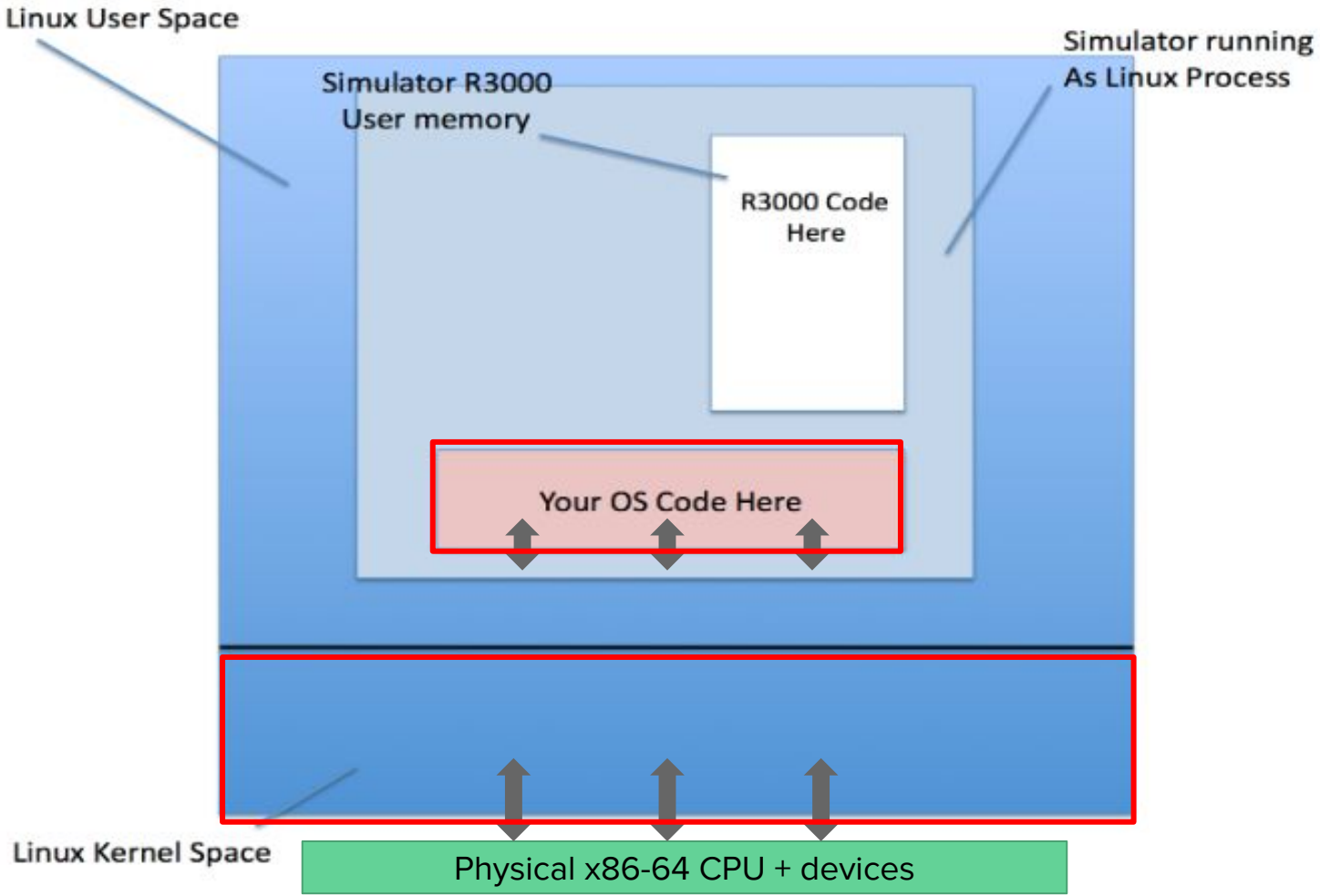


Image: https://sites.cs.ucsb.edu/~rich/class/cs170/notes/IntroKOS/index.html

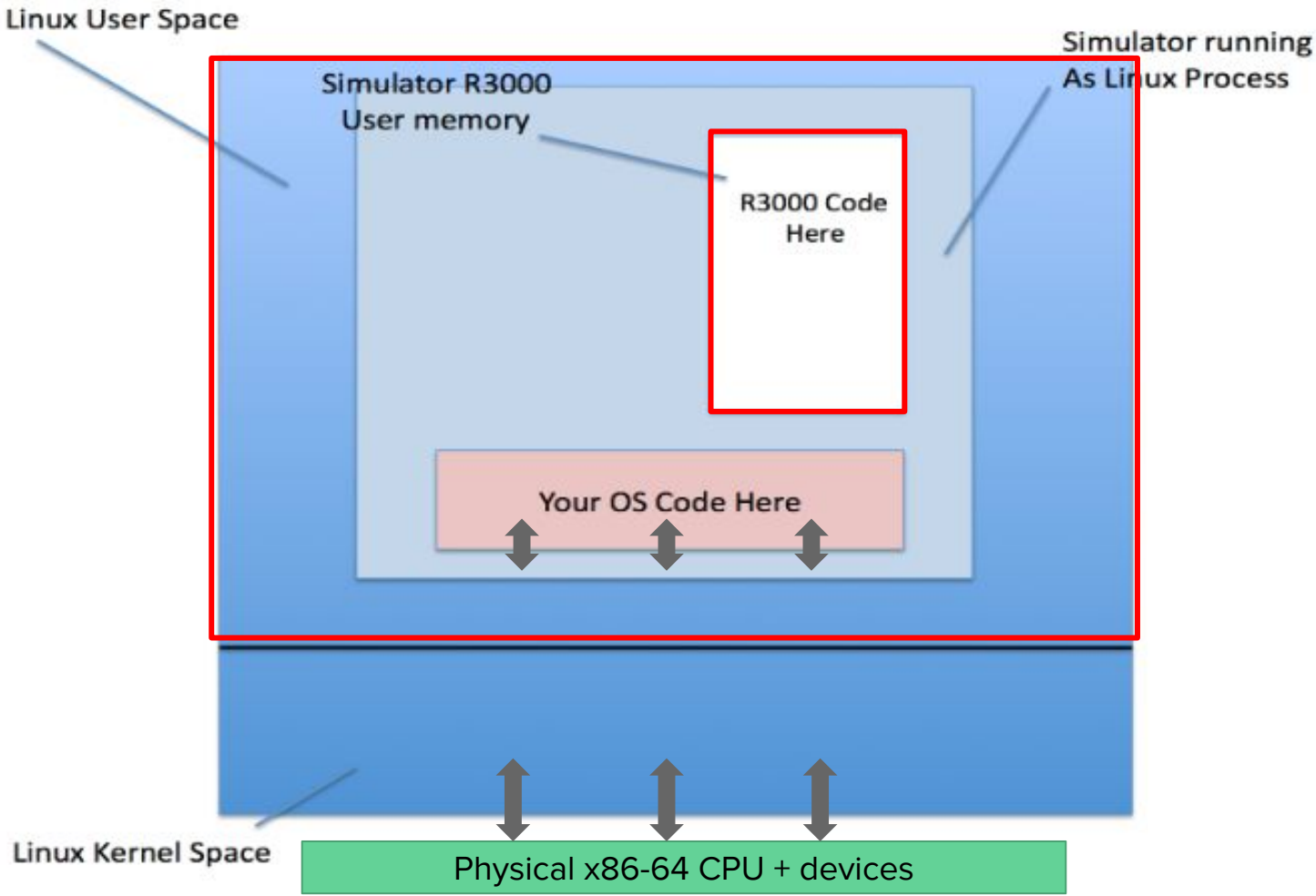


Image: https://sites.cs.ucsb.edu/~rich/class/cs170/notes/IntroKOS/index.html

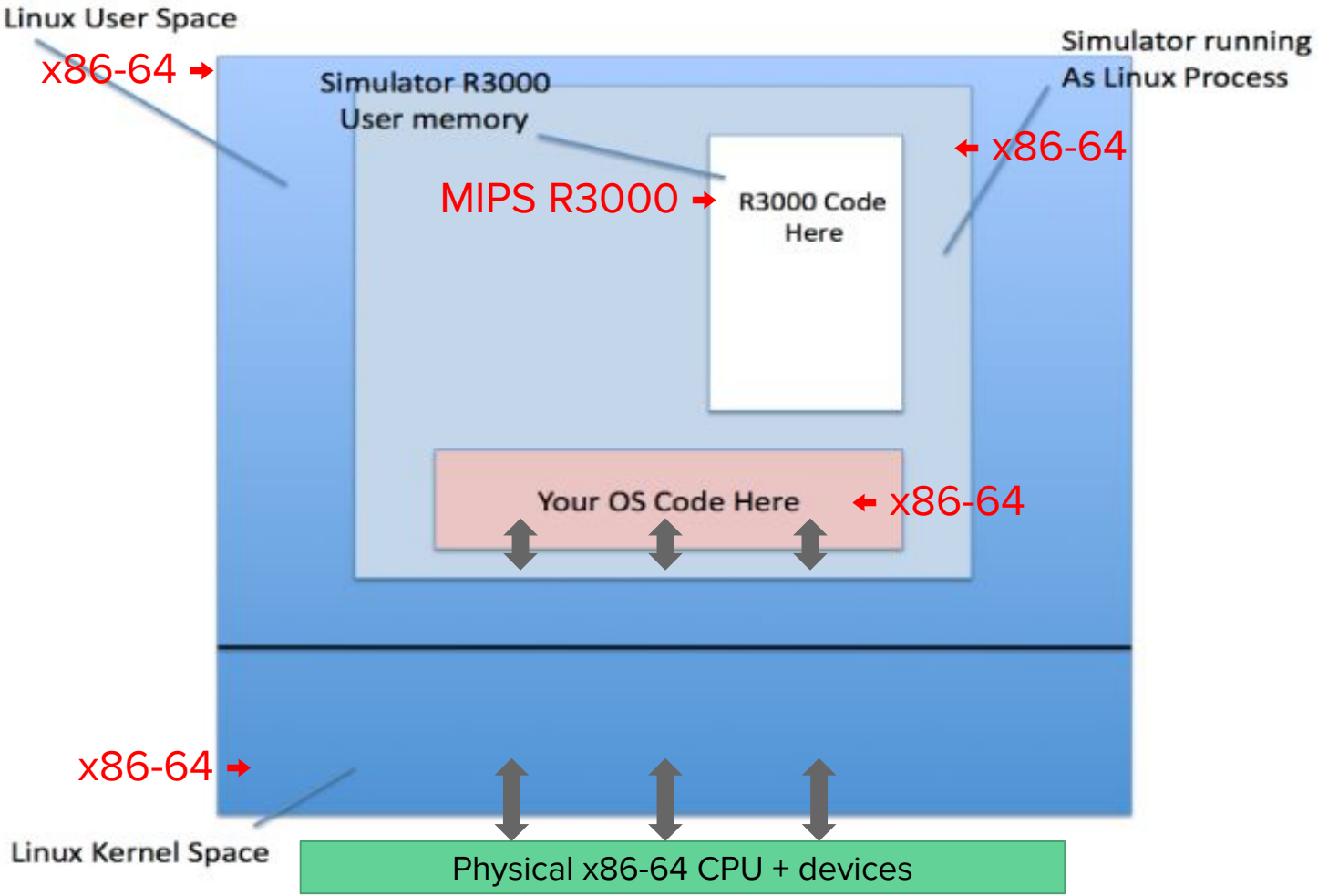


Image: https://sites.cs.ucsb.edu/~rich/class/cs170/notes/IntroKOS/index.html

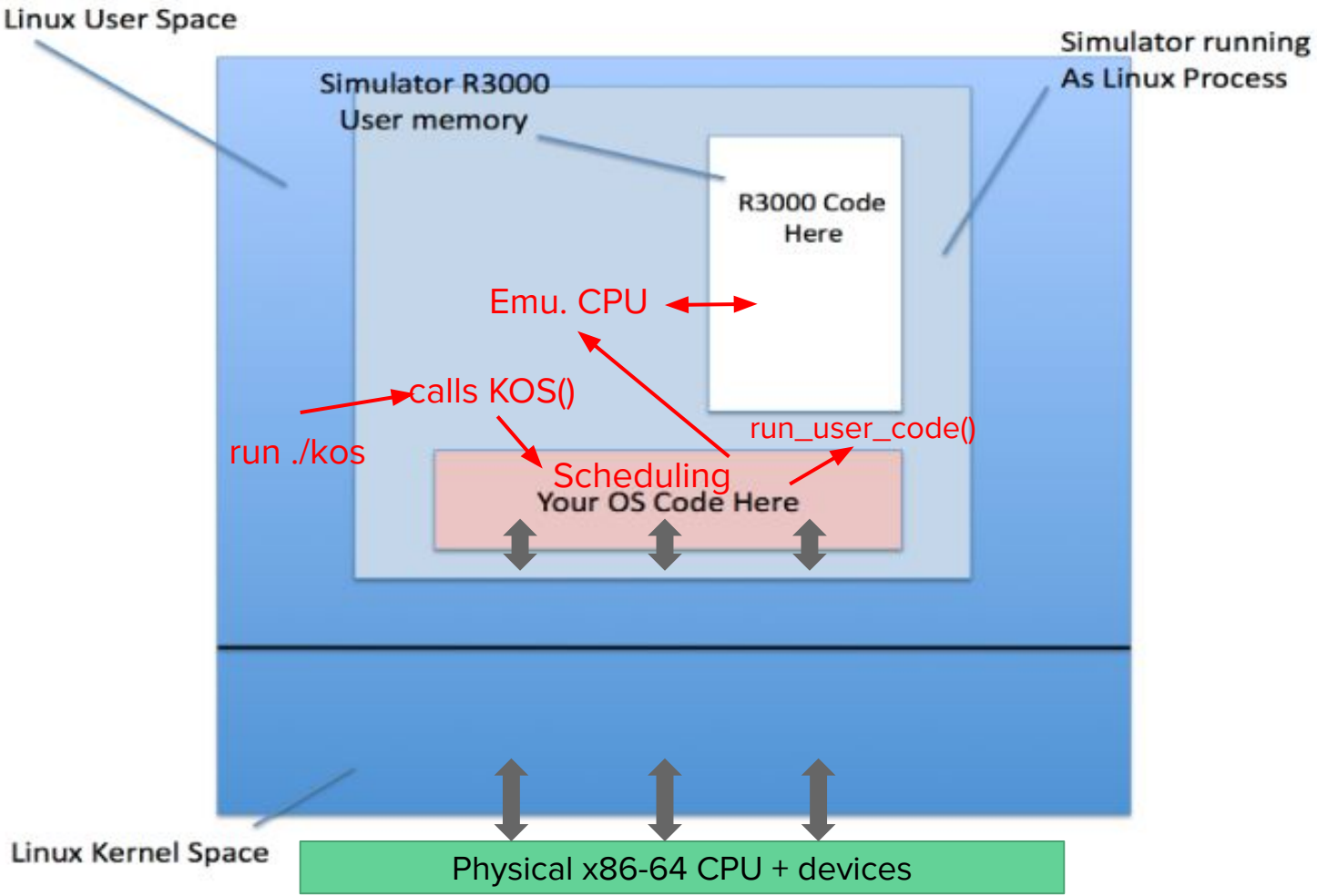


Image: https://sites.cs.ucsb.edu/~rich/class/cs170/notes/IntroKOS/index.html

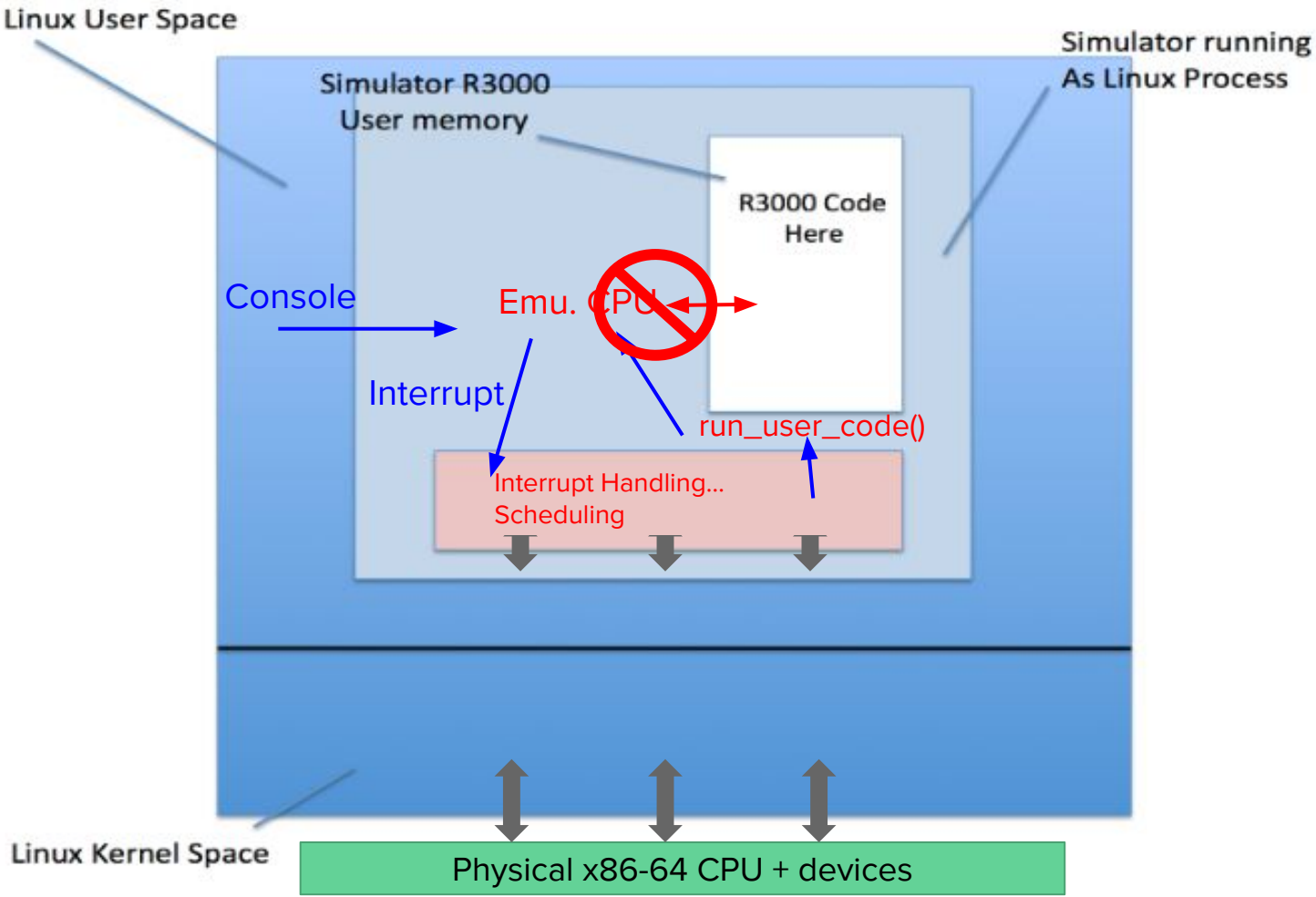


Image: https://sites.cs.ucsb.edu/~rich/class/cs170/notes/IntroKOS/index.html

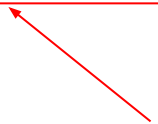
KOS: Lab1 Objectives (Overview)

Lab1 Objectives (Overview)

- KOS initialization (OS entry point)
- Load, initialize, run 1 user process (lab2 will have more than 1 process)
- KOS ➜ User transition via run_user_code
- User ➜ KOS transition via exceptions, interrupts (syscalls)
- How to run KOS kernel “tasks” (kthreads)
- Use of semaphores to communicate/sync (ensure correct behaviors)
- Implement read/write syscalls (basic IO via console)

Lab1 Objectives (Overview)

- KOS initialization (OS entry point)
- Load, initialize, run 1 user process (lab2 will have more than 1 process)
- KOS ➔ User transition via run_user_code
- User ➔ KOS transition via exceptions, interrupts (syscalls)
- How to run KOS kernel “tasks” (kthreads)
- Use of semaphores to communicate/sync (ensure correct behaviors)
- Implement read/write syscalls (basic IO via console)



Will make more sense in upcoming weeks.

KOS Initialization + “Init” User Process

KOS entry point is a procedure called “KOS”

- Initialize internal kernel states and prepares the infrastructure (PCB, etc) to run user processes.

KOS Initialization + “Init” User Process

KOS entry point is a procedure called “KOS”

- Initialize internal kernel states and prepares the infrastructure (PCB, etc) to run user processes.

The first user process the kernel runs is called the “init” process.

- In this lab, we only need to load and run the “init” process.
- However, “everything” else should be written in a way that can be easily modified to support multiple processes in lab2.

Understanding Kernel space and User Space

Kernel space: privileged execution mode (OS tasks)
User space: non-privileged execution mode (all user processes)

Understanding Kernel space and User Space

Kernel space: privileged execution mode (OS tasks)
User space: non-privileged execution mode (all user processes)

How to facilitate the transition between Kernel mode and User mode?

- KOS ➜ User (call *run_user_code*)
- User ➜ KOS (exception and interrupt handlers)

Understanding Kernel space and User Space

Kernel space: privileged execution mode (OS tasks)
User space: non-privileged execution mode (all user processes)

How to facilitate the transition between Kernel mode and User mode?

- KOS ➜ User (call *run_user_code*)
- User ➜ KOS (exception and interrupt handlers)

Syscalls: details in upcoming lecture

How to service kernel “tasks”

The kernel will need to handle many “tasks” (i.e. reading/writing to console, handle syscalls, etc).

How to service kernel “tasks”

The kernel will need to handle many “tasks” (i.e. reading/writing to console, handle syscalls, etc).

One way is through kthreads (details in upcoming weeks).

How to service kernel “tasks”

The kernel will need to handle many “tasks” (i.e. reading/writing to console, handle syscalls, etc).

One way is through kthreads (details in upcoming weeks).

Why are kthreads useful (as a take-home question)? 🤔

How to service kernel “tasks”

The kernel will need to handle many “tasks” (i.e. reading/writing to console, handle syscalls, etc).

One way is through kthreads (details in upcoming weeks).

Why are kthreads useful (as a take-home question)? 🤔

- kt_fork(): create a new kthread
- kt_joinall(): wait until all kthreads exit or is blocked
- ...

https://sites.cs.ucsb.edu/~rich/class/cs170/notes/Kthreads/index.html

How to service kernel “tasks”

The kernel will need to handle many “tasks” (i.e. reading/writing to console, handle syscalls, etc).

One way is through kthreads (details in upcoming weeks).

Why are kthreads useful (as a take-home question)? 🤔

- kt_fork(): create a new kthread
- kt_joinall(): wait until all kthreads exit or is blocked
- ...

NOTE: kthreads are non-preemptive in our implementation.

https://sites.cs.ucsb.edu/~rich/class/cs170/notes/Kthreads/index.html

Semaphore: A Synchronization Primitive

Recall the “race conditions” lecture about critical section, atomic execution, and mutual exclusion.

https://sites.cs.ucsb.edu/~rich/class/cs170/notes/RaceConditions/index.html

Semaphore: A Synchronization Primitive

Recall the “race conditions” lecture about critical section, atomic execution, and mutual exclusion.

- A semaphore can be used to synchronize and manage critical sections.
- Details on semaphores will be in upcoming lecture

https://sites.cs.ucsb.edu/~rich/class/cs170/notes/RaceConditions/index.html

Semaphore: A Synchronization Primitive

Recall the “race conditions” lecture about critical section, atomic execution, and mutual exclusion.

- A semaphore can be used to synchronize and manage critical sections.
- Details on semaphores will be in upcoming lecture

Take-home question: Do we still need synchronization for non-preemptive (k)threads? 🤔

https://sites.cs.ucsb.edu/~rich/class/cs170/notes/RaceConditions/index.html

Console: read and write syscalls

Console is the way we will be handing basic IO (input-output) in KOS.

Console: read and write syscalls

Console is the way we will be handing basic IO (input-output) in KOS.

It is a shared device so programs can not (and should not) access it directly.

A program asks the OS to **read/write** through syscalls.

- Check the man page for details.

Console: read and write syscalls

Console is the way we will be handing basic IO (input-output) in KOS.

It is a shared device so programs can not (and should not) access it directly.

A program asks the OS to **read/write** through syscalls.

- Check the man page for details.

A key thing to understand is that the console is a *synchronous* device.

- When a char is available to be read, an *interrupt* is generated for the OS to service. Otherwise, the char is lost :(
- Writing a char to console requires “acknowledgment” that the char is written successfully before writing another.

Hints for Lab1

1. Read everything and I mean it!
    - Try to understand how each part interacts with each other.
    - Understand how interrupts, etc work. Execution logic in the OS is inherently non-linear and asynchronous.
    - Read documentation (man pages, libfdr docs, etc)

Hints for Lab1

1. Read everything and I mean it!
    - Try to understand how each part interacts with each other.
    - Understand how interrupts, etc work. Execution logic in the OS is inherently non-linear and asynchronous.
    - Read documentation (man pages, libfdr docs, etc)
2. Start early!

Hints for Lab1

1. Read everything and I mean it!
    - Try to understand how each part interacts with each other.
    - Understand how interrupts, etc work. Execution logic in the OS is inherently non-linear and asynchronous.
    - Read documentation (man pages, libfdr docs, etc)
2. Start early!
3. Write good code: future you (lab2 and lab3) will thank the current you.

Hints for Lab1

1. Read everything and I mean it!
    - Try to understand how each part interacts with each other.
    - Understand how interrupts, etc work. Execution logic in the OS is inherently non-linear and asynchronous.
    - Read documentation (man pages, libfdr docs, etc)
2. Start early!
3. Write good code: future you (lab2 and lab3) will thank the current you.
4. Test your code!
    - Write isolated tests to ensure expected behavior
    - Adopt the mindset of “adversarial testing” to try to produce unexpected behavior

Questions and debug time!