Lecture 1: Course intro

601.418/618 Operating Systems

David Hovemeyer

January 22, 2025

Agenda

- Administrative stuff
- Course overview
- Operating systems: background

Administrative stuff

Staff

- Instructor: David Hovemeyer, daveho@cs.jhu.edu
  - Office: Malone 240A
  - Office hours (via Zoom): T/Th 1–3 pm, or by arrangement (email me)
- Course assistants:
  - Heads: Theo Lee, Max Hahn
  - Regular: TBA
  - Office hours: TBD, will announce soon

Online communication

Courselore will be used for announcements, Q&A

- You should have received an invite (email me if you need one)
- Anonymous posts are ok (although you're not anonymous to course staff)
- Make a private post to course staff if you have an issue that's specific to you
- Please don't post solution code publicly
- Please **do** ask questions and answer other people's questions!

Feel free to email me about any issues/concerns you have, but questions about course content, assignments, etc. should be posted to Courselore.

Online resources

Course website: https://jhuopsys.github.io/spring2025/

- All public information about the course (syllabus, schedule, resources, assignments, lecture slides, etc.) will be posted here

Canvas: lecture recordings will be posted here (click on "Panopto Recordings")

- This is the only thing we will be using Canvas for

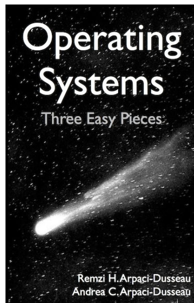
Gradescope: some or all assignments will be submitted here. Assignment grades will *probably* be posted here. Exam grades will be posted here. Instructions about how to log in:

https://jhuopsys.github.io/spring2025/gradescope.html

Make a private post to Courselore if you are having issues accessing Gradescope.

Textbook

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



This is a free online textbook. The readings are linked directly from the Schedule page.

Grading

Grade breakdown for the course is:

- Exams: 40% of grade (3 exams, each is worth $13.\overline{3}\%$)
  - Exams 1 and 2 in class during the semester, Exam 3 will be during the scheduled final exam time
- Assignments: 60% of grade
  - Expect these to be very challenging

Late hours

Everyone gets 120 late hours to use as needed on assignments throughout the semester. (Note that use of late hours will likely be restricted on the last assignment.)

For team assignments, to use late hours, everyone on the team must have late hours to use. (E.g., to use 10 late hours, everyone on the team needs to have at least 10 late hours remaining.)

If you have a concern that you and/or your group won't be able to complete an assignment, please let me know *in advance of the deadline.* In general, I'm willing to provide additional flexibility (within reason) if it means that work will be completed successfully.

Academic ethics

We expect you to follow the JHU Computer Science Academic Integrity Code:
https://www.cs.jhu.edu/academic-programs/academic-integrity-code/

- The work you submit must be your (and your teammates') original effort
- Copying without attribution is a violation of academic ethics
- Copying of significant chunks of code *with* attribution doesn't constitute original effort

Note about use of AI assistants (ChatGPT, Github Copilot, etc.): code written by an AI assistant doesn't consistute original effort. Also, if the tool copies someone else's original code into your program, that violates academic ethics.

Course overview

Goals of the course

By the end of the course, you should

- Deeply understand how an OS kernel provides services to programs
- Have significant hands-on experience implementing significant components of a realistic OS kernel (scheduling, virtual memory, filesystem, etc.)

The nature of OS development

OS kernel programming involves:

1. Direct exposure to hardware. Testing and debugging "bare metal" code is challenging.
2. Asynchrony (such as hardware interrupts) and concurrency (such as thread scheduling and thread synchronization.)

Let's say that issue #1 makes programming N times harder than "regular" programming, and issue #2 makes programming M times harder. So, OS kernel programming is $N \times M$ times harder than regular programming.

If I had to guess: both N and M are *at least* 2, and probably closer to 5.

Warning

OS programming is very challenging.

What you should expect from the course

In prior semesters Prof. Huang's guidance was:

- "This is a TOUGH course"
- "Requires proficiency in systems programming"
- "Requires significant time commitment"

Student feedback from prior semesters:

- "The projects are insanely time consuming"
- "The workload is much much heavier than your average CS course"

I've taught this course once previously (Spring 2024) and I can confirm that the above is true.

The upside

By completing the assignments, you will *really* learn how a modern computer system works.

It is hugely satisfying to have total control over the machine, and to create the abstractions that user programs require.

In the future, knowing what the OS kernel is doing will help you write programs that take full advantage of system capabilities.

Most importantly: kernel hacking is a lot of fun.

Project sequence

Projects are based on the Pintos educational operating system kernel.

Pintos is a minimal but realistic OS kernel that can boot on 32-bit x86 hardware.

Projects:

- Assignment 0: warm up
- Assignment 1: threads and thread scheduling
- Assignment 2: user processes
- Assignment 3: virtual memory, paging
- Assignment 4: filesystem

When you're done, you have a complete OS!

Assignment structure

In addition to writing code for each assignment, you will also read and trace the provided code.

You will need understand the base Pintos kernel in order to build on top of it. Fortunately, it is well designed and has lots of comments.

Design document, coding style

For each assignment, you will turn in a design document. In the design document, you will answer questions and also document and justify your design for the OS feature you are adding.

When you write code, we expect it to be

- correct (including handling of corner cases and error conditions)
- clean
- well-commented

This should be the best and most beautiful code you've ever written. (Remember: kernel programming is $N \times M$ times harder than normal programming.)

Team projects

Aside from Assigment 0, all assignments may be done in teams of up to 3.

Please start assembling your team! There is a Courselore post where you can advertise your availability and/or see who else is available.

Teamwork

Working in a team can help you learn more and get more done. **However**, the way that the team members work together is critical.

We **strongly** recommend that you do **not** rigidly divide up the work ("I'll work on this part, you work on that part") and then try to integrate your code close to the deadline. This approach will not work out well.

We **do** recommend that you

- Work together synchronously as much as possible
- Keep a task list (shared Google docs are great for this) that has small/modular tasks that anyone can work on
- Make small changes, *test them thoroughly*, and commit/push frequently
- Pull your teammates' changes frequently
- **Communicate** (maybe set up a Slack workspace or Discord for your group?)

Emulation

Although the project sequence is capable of booting and running on real hardware, you will do your development, testing, and debugging in an emulator. In particular, the QEMU emulator allows you to debug kernel code using gdb.

We'll also use the Bochs emulator for interactive and automated testing.

So, you will have some of the conveniences of "normal" development.

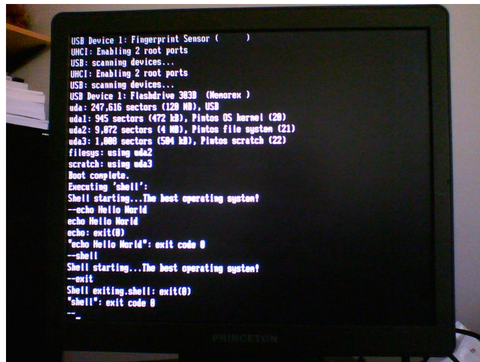
Bochs x86 emulator, http://bochs.sourceforge.net/

```
Bochs VBE Display Adapter enabled

Bochs BIOS - build: 11/11/12
$Revision: 11545 $ $Date: 2012-11-11 09:11:17 +0100 (So, 11. Nov 2012) $
Options: apmbios pcibios pnpbios eltorito rombios32

ata0 master: Generic 1234 ATA-6 Hard-Disk (    0 MBytes)

Press F12 for boot menu.

Booting from Hard Disk...
Pintos hda1
Loading...........
Kernel command line:
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  102,400 loops/s.
Boot complete.
CS318> whoami
Dave
CS318> Hello CS418/618!
invalid command
CS318> _
```

IPS: 2.200M NUM CAPS SCRL HD:0-M

Figure 1: Assignment 0 running on Bochs

Real hardware



In theory, Pintos can boot on real hardware. However, I've had issues getting the Pintos bootloader to work on the one 32-bit x86 system I have access to. If we figure out how to resolve this, we'll let you test your code on real hardware.

Development environment

For development you will need:

- A toolchain (gcc/gas/ld/gdb) targeting 32-bit x86
- Emulators (Bochs and QEMU)
- Various utilities (e.g., Perl)

Options:

- Use CS department ugrad and grad machines
- Your own machine

Instructions here: https://jhuopsys.github.io/spring2025/assign/setup.html

Please get your environment set up ASAP!

Also note: if you choose to try using an IDE (e.g., VS Code), we can't guarantee that we will be able to help you troubleshoot if it doesn't work.

Exams

There will be three exams. Each one will cover roughly 1/3 of the course material. They will include both conceptual/theoretical questions and "practical" questions (which could test what you learned from the assignments.)

Exams 1 and 2 are in class (see the Schedule for the dates.)

Exam 3 is during the scheduled final exam time slot (but it is not a cumulative final exam.)

Homework assignments

We will periodically give you written homework assignments. These don't count towards your grade, but will be very helpful for testing your understanding of the course material and preparing for exams.

We recommend that you take them seriously.

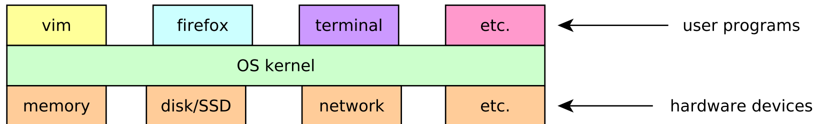
Getting the most out of the course

- Do the reading
- Come to class
- Attend office hours
- Ask questions (and answer questions) on Courselore
- In general, ask for help if you need it
- **Start assignments early and make steady progress**
- Do the written homeworks

Operating systems: background

What is an operating system?

The primary responsibility of an operating system is to allow *user processes* to access the hardware resources they need (CPU time, memory, file storage, network communication, etc.)



The goal is to allow processes to *share* the system resources effectively.

We can say that the OS kernel *virtualizes* system resources, and presents user processes with an *abstraction* of those resources. No process gets direct access to hardware resources.

Why do we need one?

Alternative to using an operating system: we could run programs directly on the "bare hardware".

Problems with this idea:

- How would multiple programs share the CPU?
- How would multiple programs share the SSD?
- How would multiple programs share the network interface?

Another issue: in many cases, the underlying hardware has a complex interface. The OS can present a uniform and simplified interface to the hardware. For example, if you want to write data to an SSD, you just use the `open` and `write` system calls (or a higher-level API.) You don't need to access the device registers, arrange for DMA, handle hardware interrupts, etc.

When might we not need (or want) one?

The abstractions the OS kernel presents do have a cost (performance, memory use.)

For very resource-constrained systems, we might not have enough

- CPU power
- memory
- flash storage, etc.

for a full OS. (Sometimes the case with small embedded systems.)

Another issue: if the OS kernel has control of the hardware, will an application always have the quality of service it needs? E.g., a real time system could miss a deadline to handle an event if it is preempted at the wrong time. (Note: an OS kernel can implement support for real time processes.)

In the beginning...

Early days of computing, direct programming on hardware, maybe batch processing.

(Image from Wikipedia: IBM 704)

Important milestones

Minicomputer era (1960s–1970s): beginnings of support for memory protection via segmentation (e.g., PDP-10, PDP-11)

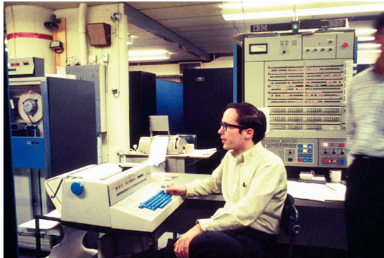
- Some important early multi-user operating systems (CTSS, ITS, Multics, Unix)

(Image from Wikipedia: PDP-11/70)

Important milestones

IBM System/360 Model 67 (1965), DEC VAX (1977): paged virtual memory, 32 bit addresses, designed for time sharing

- Recognizably modern hardware features!
- Still much too expensive for "personal" use



(Images from Wikipedia: IBM System/360 Model 67, VAX 11/780)

Important milestones

Intel 80386 (1985): first x86 with support for paged virtual memory

- Personal computers become powerful enough to run “modern” operating systems

(Image from Wikipedia: Compaq Deskpro 386)

Important milestones

Unix workstations (1980s–early 2000s): These were a thing until the segment was made obsolete by PC hardware catching up. For their time, they had powerful CPUs, large amounts of RAM, and ran recognizably modern operating systems (multiprocess, multiuser, virtual memory, etc.)

(Image from Reddit, u/khooke, Sun Ultra 60)

I used one of these in grad school!

Today's hardware

My current smart phone: Google Pixel 4

- 6 GB of RAM, 8-core CPU (fastest one clocked at 2.8 GHz)

This hardware is **vastly** more powerful than the $20,000 Sun workstation I used in grad school.

By historical standards, nearly every current computing device is a supercomputer!

The point

The point here is that modern, fully-featured operating systems are possible and useful on a large subset of all computing devices now.

OS kernel vs. "Operating System"

Lots of things we normally consider to be part of an operating system are outside the kernel.

- Standard libraries (e.g., the C library)
- Window system
- Standard utility programs

System wouldn't be very useful without these things!

Linux terminology: "distribution" = kernel + lots of necessary/useful software

The process abstraction

A *process* is a running program.

Rather than directly accessing system resources, it requests services from the OS kernel.

Issue: for the process abstraction to be robust, the OS kernel must be able to control how system resources are used. Processes must be able to use *only* the resources to which they've been granted access.

User mode vs. kernel mode

To make the process abstraction robust, the CPU must have privilege levels:

- *Kernel mode*: code running in this mode has full access to all hardware resources
- *User mode*: restricted, cannot directly access all hardware resources

Only the OS kernel runs in kernel mode.

Ordinary processes run in user mode.

Restrictions in user mode

Typically, code executing in user mode can only

- Access memory to which the process has been specifically granted access
  - Including executing instructions from code memory
- Invoke *system calls* to request a service from the OS kernel

A process *cannot*

- Access arbitrary memory (either the kernel's memory or that of another process)
- Access a hardware device (the CPU instructions required to access hardware are only allowed in kernel mode)

These restrictions are (in theory) sufficient for the OS kernel to maintain complete control over the system.

Ensuring fair access to the CPU

What about the following program?

```
int main(void) {
  while (true) {
    // do nothing
  }
}
```

Why doesn't this program monopolize the CPU indefinitely?

Hardware interrupts

System hardware devices can request to *interrupt* the CPU. The occurrence of an interrupt

- transitions to kernel mode (if the CPU is not already in kernel mode)
- executes a kernel *interrupt handler* routine to respond to the interrupt

If a hardware interrupt occurs while a process is executing user code, *the OS kernel regains control*.

Interval timer: a hardware device whose only purpose is to periodically interrupt the CPU, to ensure the OS kernel regains control of the CPU at regular intervals.

Later on we'll see how the OS kernel can make *scheduling* decisions to allocate intervals of CPU time to processes fairly. This idea is called *time-slicing*.

System calls

A *system call* allows a user process to request a service from the OS kernel:

- opening a file
- reading data from a file
- writing data from a file
- mapping memory into the address space
- creating a child process
- etc.

A system call is invoked using a special CPU instruction, sometimes referred to as a *trap*. The effect of a trap is very similar to what happens when a hardware interrupt occurs:

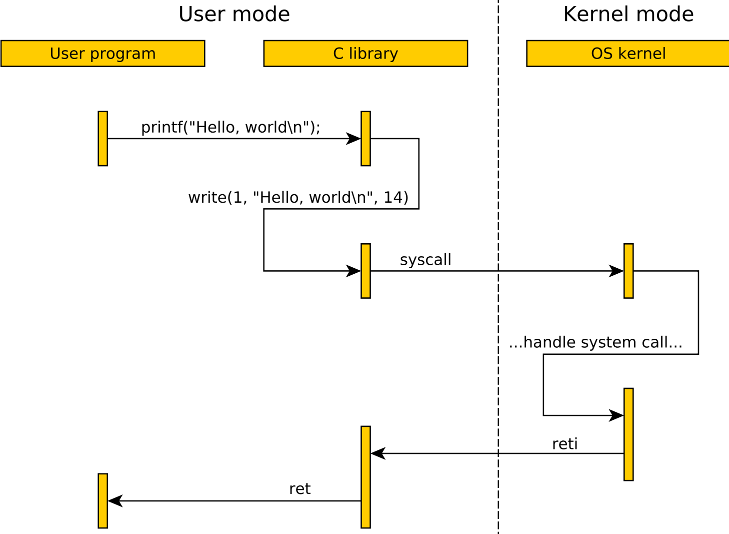
- the CPU switches to kernel mode, and
- the *system call handler* routine in the OS kernel is executed

Traps are sometimes called "software interrupts."

Example program

```c
int main(void) {
  printf("Hello, world\n");
  return 0;
}
```

What happens



For next time

Lots of stuff you can/should do!

- Reading (Chapters 1–2, 6)
- Get your development environment set up
- Form your project team (for Assignments 1 and later)
- Start working on Assignment 0