# introduction / processes + system calls

CS 4414-001

# lectures

via Zoom

recordings available afterwards

- attendance not required

if you aren't watching live, I recommend writing down questions...

- you can ask via Piazza, office hours

# course webpage

`https://www.cs.virginia.edu/~cr4bd/4414/S2021/`

linked off Collab

# office hours

via Discord
  voice chat and/or screensharing and/or text chat
  instructions on website
  invite link on Collab

queue website:
  first few slots first-come, first-served
    may be reset manually by TAs, e.g. when long gaps between OHs
  later slots by last time helped

my office hours: I might be splitting time with CS4630

# homeworks

there will be programming assignments

first is due **next week**

...mostly in C or C++; one in Python

one or two weeks

- if two weeks "checkpoint" submission after first week

two week assignments worth more

# xv6

some assignments will use xv6, a teaching operating system

simplified OS based on an old Unix version
- built by some people at MIT
- (though they currently use a RISC V version
- instead of the x86-32 version we'll use)

theoretically actually boots on real 32-bit x86 hardware

...and supports multicore!
- (but we'll run it only single-core, in an emulator)

# quizzes

there will be online quizzes after each week of lecture

…starting this week (due next Tuesday)

same interface as CS 3330, but no time limit

(haven't seen it? we'll talk more on Thursday)

quizzes are open notes, open book, open Internet

# exams

final exam

current plan: take-home, 24 hours, overlapping official final time

(subject to change, will announce later)

probably mix of quiz-like questions, plus some longer answers

might include some programming exercise or similar

# late policy

there is a late policy on the website

# textbook

recommended textbook:
Anderson and Dahlin, *Operating Systems: Principles and Practice*

no required textbook

alt: Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* **(free PDFs!)**

- some topics we'll cover where this may be primary textbook

alternative: Silberchartz (used in previous semesters)

- full version: Operating System Concepts, Ninth Edition

# cheating: homeworks

don't

homeworks are individual

no code from prior semesters (other than your own)

no sharing code, pesudocode, detailed descriptions of code

no using code from Internet/etc., with limited exceptions

- tiny things solving problems that aren't point of assignment
- …*credited* where used in your code
- e.g. code to split string into array for non-text-parsing assignment
- exception: something explicitly referred to by the assignent writeup
- in doubt: ask

# citation

if using small amount of code *clearly not point of assignment*

- e.g. split string into array for non-text-parsing assignment
- e.g. filling arrays of pointers from vectors of strings

not sure what counts? ask

then make sure you cite where you got it in your code

- should not be other student, etc. — no sharing code

if using code clearly part of major objective of assignment

then don't

- e.g. if you find a shell online, don't use it solve the shell assignment

# cheating: quizzes

don't

quizzes: also individual

don't share answers

don't IM people for answers

don't ask on StackOverflow for answers

# waitlisted?

if you want an exception, please explain why not Prof. Lin's section

# getting help

Piazza

TA and my office hours (will be posted soon)

emailing me

# history: computer operator

via National Library of Medicine; computer operators operating an Honeywell 800

# OS definition ambiguity

different exact defintions

‘part of OS’ v. ‘just a program/library’

- example: code to allow moving windows on the screen part of the OS?
- example: code to support printers is part of the OS?

we’ll not sweat the details — give general, common principles

# what is an operating system?

software that:

Anderson-Dahlin manages a computer's resources

Arpaci-Dusseau provides 'virtual machine': more convenient than real machine

# OS roles

Anderson-Dahlin's taxonomy of things OS's do

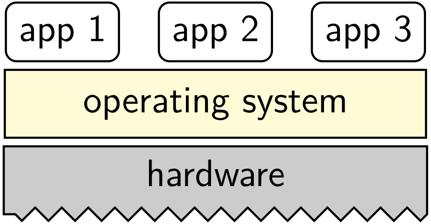
referee — resource sharing, protection, isolation

illusionist — clean, easy abstractions

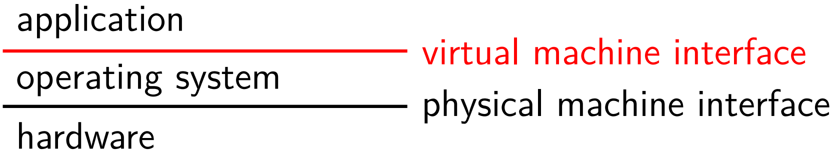
glue — common services

storage, window systems, authorization, networking, ...

# OS as abstraction layer



# the virtual machine interface



application

virtual machine interface

operating system

physical machine interface

hardware

*system virtual machine*

(VirtualBox, VMWare, Hyper-V, …)

*process virtual machine*

(typical operating systems)

imitate physical interface

(of some real hardware)

chosen for convenience

(of applications)

# system virtual machines

run entire operating systems

- for OS development, portability

interface ≈ hardware interface (but maybe not the real hardware)

- aid reusing existing raw hardware-targeted code
- different “application programmer”

# process virtual machine

| process VM | real hardware |
|---|---|
| thread | processors |
| memory allocation | page tables |
| files | devices |
| … | … |

# process virtual machine

| process VM | real hardware |
|---|---|
| thread | processors |
| memory allocation | page tables |
| files | devices |
| … | … |

(virtually) infinite “threads” ($\sim$ virtual CPUs)
no matter number of CPUs

# process virtual machine

| process VM | real hardware |
|---|---|
| thread | processors |
| memory allocation | page tables |
| files | devices |
| … | … |

memory allocation functions
no worries about organization of “real” memory

# process virtual machine

| process VM | real hardware |
|---|---|
| thread | processors |
| memory allocation | page tables |
| files | devices |
| … | … |

files — open/read/write/close interface
no details of hard drive operation
or keyboard operation or …

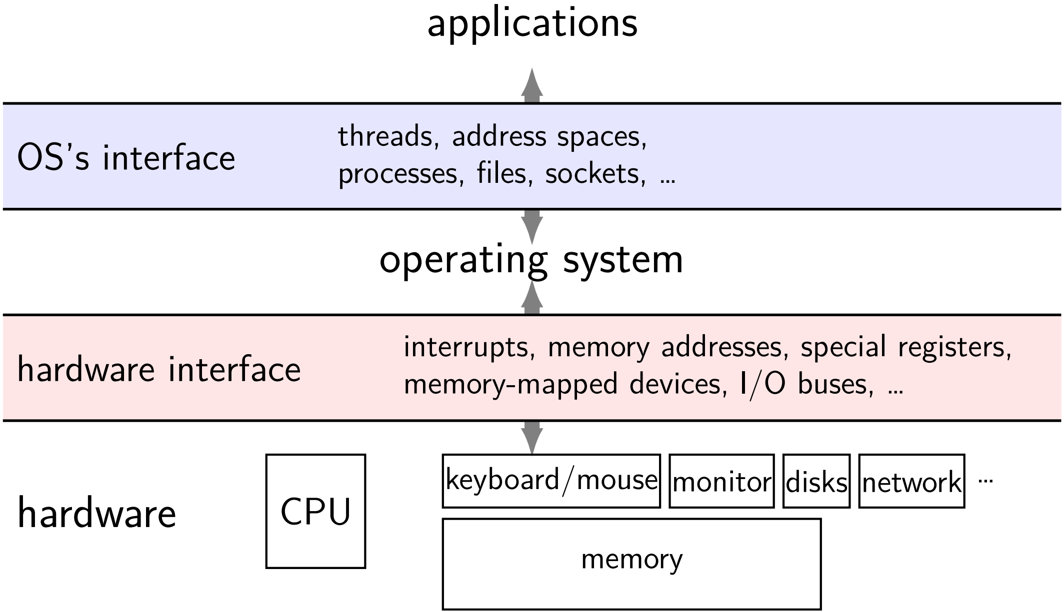
# The Process

process = thread(s) + address space + ...

illusion of dedicated machine:

- thread = illusion of own CPU
- address space = illusion of own memory

# the abstract virtual machine



applications

OS's interface

threads, address spaces,
processes, files, sockets, …

operating system

hardware interface

interrupts, memory addresses, special registers,
memory-mapped devices, I/O buses, …

hardware

CPU

keyboard/mouse

monitor

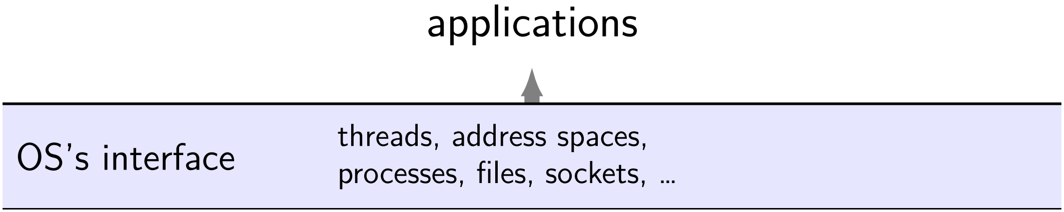
disks

network

…

memory

# abstract VM: application view



applications

OS's interface

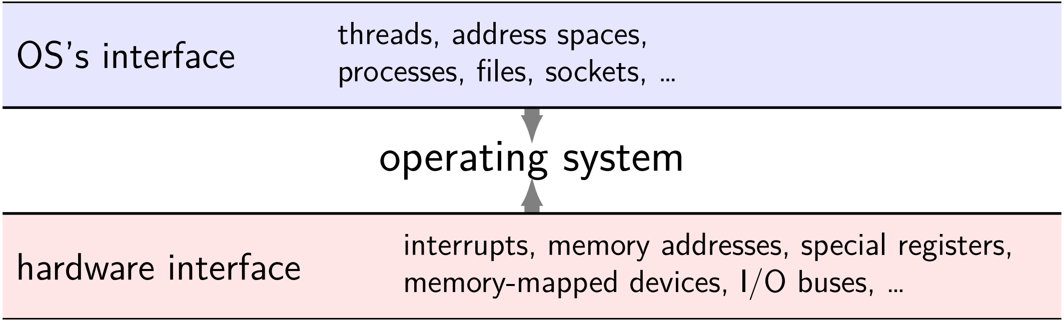
threads, address spaces,
processes, files, sockets, …

the application's "machine" is the operating system

no hardware I/O details visible — future-proof

more featureful interfaces than real hardware

# abstract VM: OS view



OS's interface

threads, address spaces,
processes, files, sockets, ...

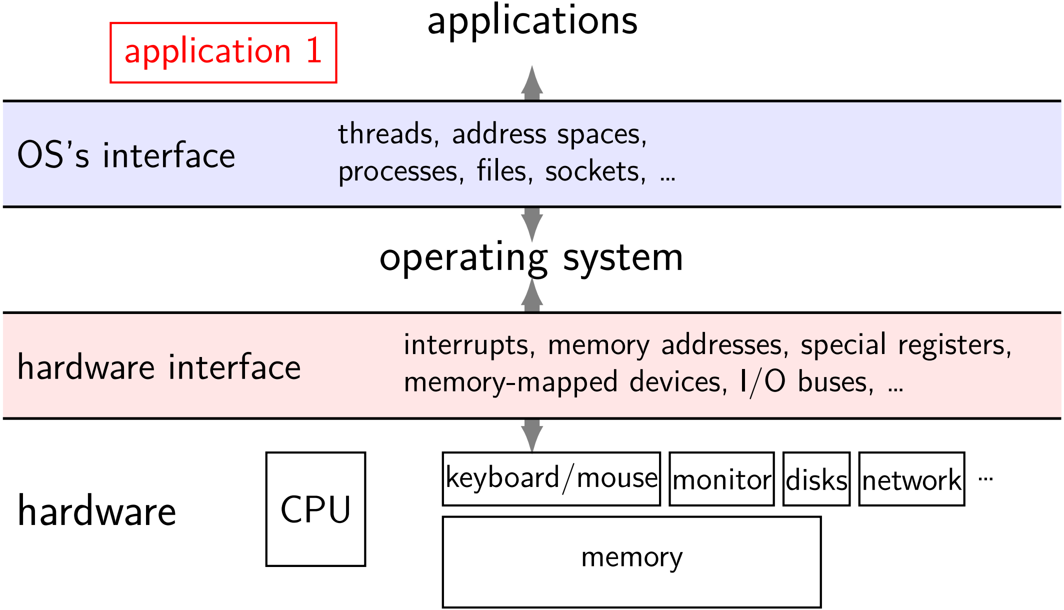
operating system

hardware interface

interrupts, memory addresses, special registers,
memory-mapped devices, I/O buses, ...

operating system's job: translate one interface to another

# program → process → CPU and memory



applications

application 1

OS's interface

threads, address spaces,
processes, files, sockets, ...

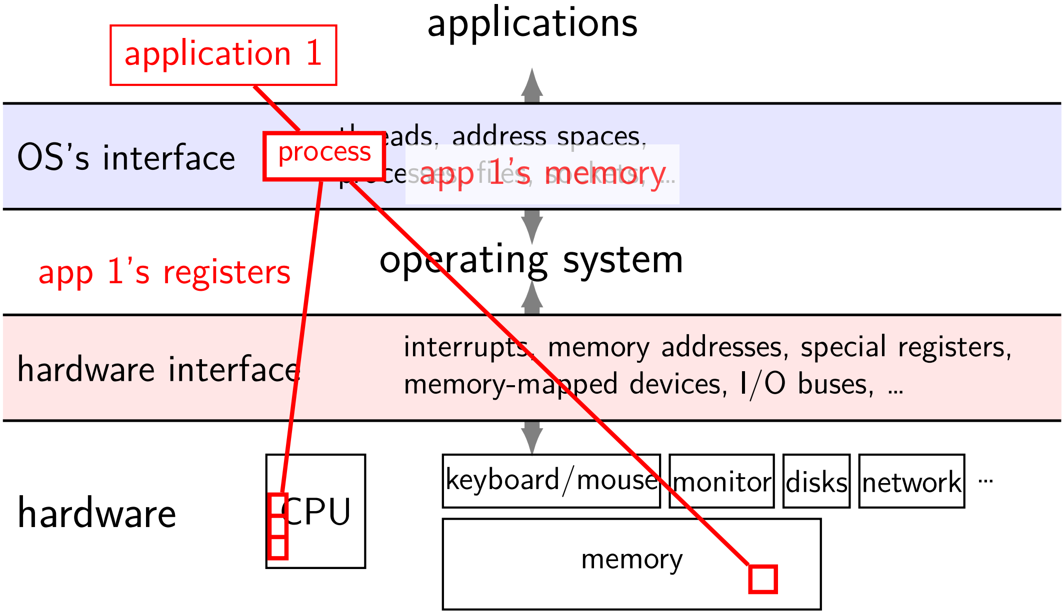
operating system

hardware interface

interrupts, memory addresses, special registers,
memory-mapped devices, I/O buses, ...

hardware

# program → process → CPU and memory



applications

application 1

OS's interface

process

threads, address spaces,
processes, files, sockets, ...

app 1's memory

operating system

app 1's registers

hardware interface

interrupts, memory addresses, special registers,
memory-mapped devices, I/O buses, ...

hardware

CPU

keyboard/mouse

monitor

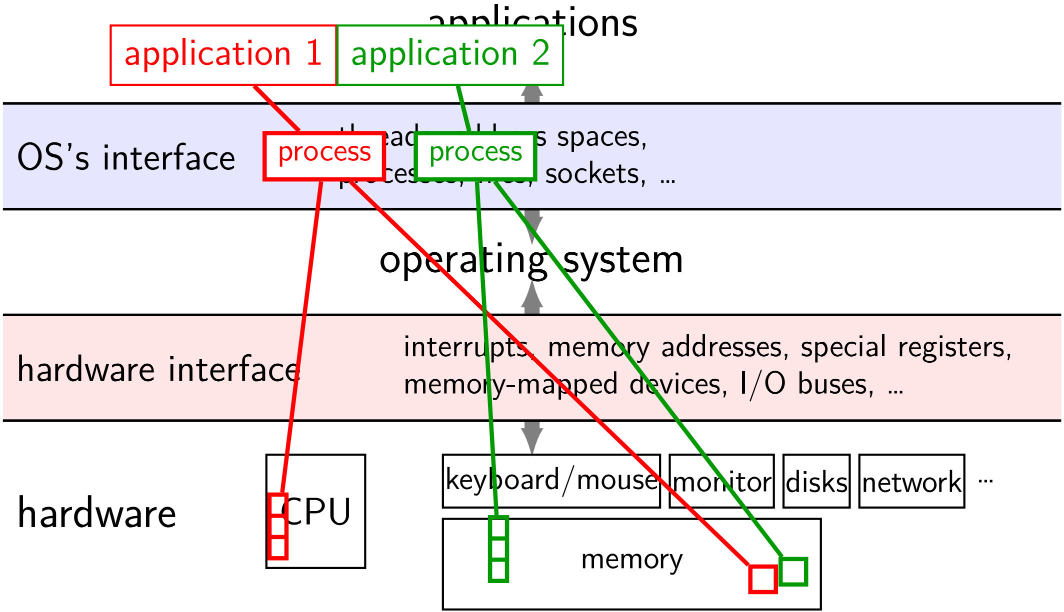
disks

network

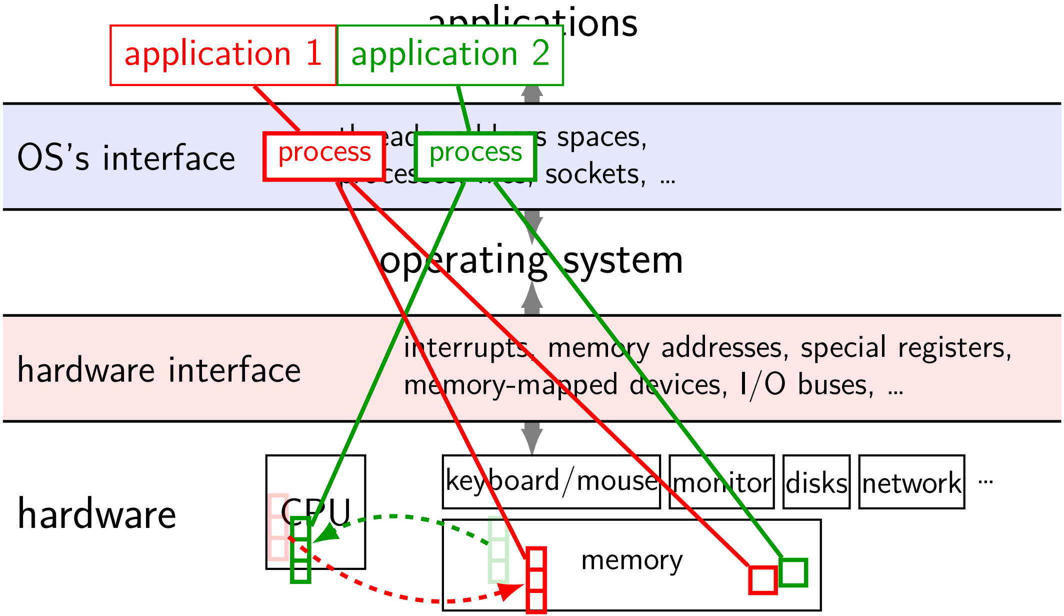
...

memory

# program → process → CPU and memory



# program → process → CPU and memory



applications

application 1

application 2

OS's interface

process

process

threads, address spaces,
processes, files, sockets, ...

operating system

hardware interface

interrupts, memory addresses, special registers,
memory-mapped devices, I/O buses, ...

hardware

CPU

keyboard/mouse

monitor

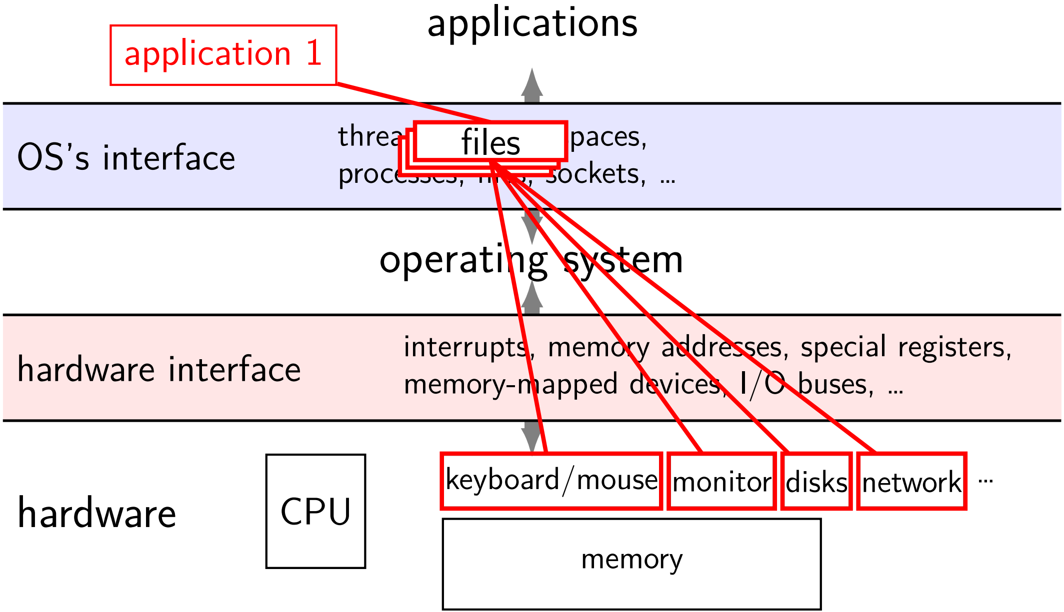
disks

network

...

memory

# files → input/output



applications

application 1

OS's interface

threa paces,
proce sockets, …

files

operating system

hardware interface

interrupts, memory addresses, special registers,
memory-mapped devices, I/O buses, …

hardware

CPU

keyboard/mouse

monitor

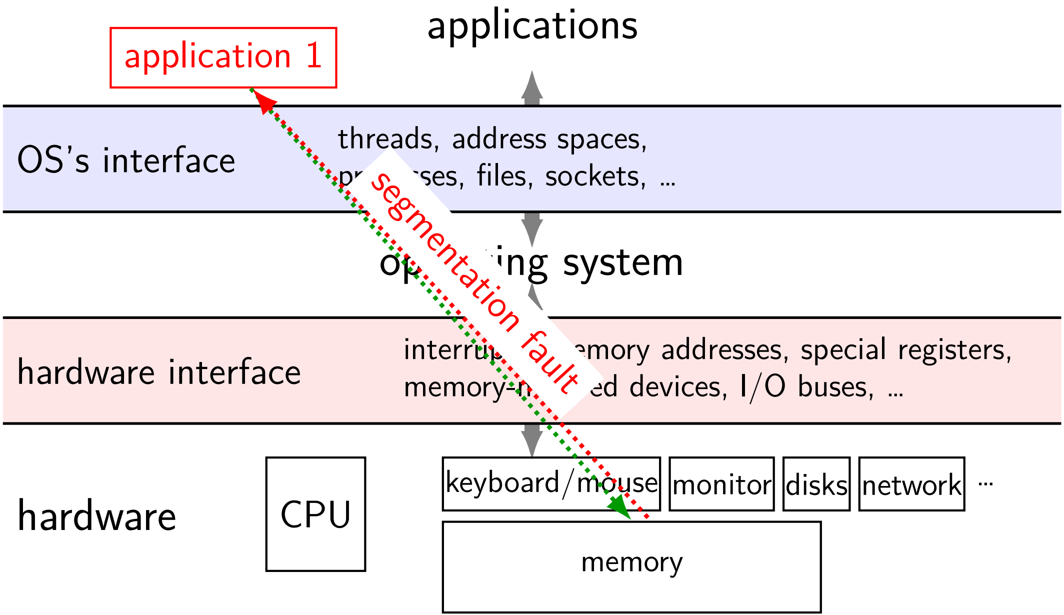
disks

network

…

memory

# security and protection



applications

application 1

OS's interface

threads, address spaces,
pr sses, files, sockets, …

segmentation fault

op ing system

hardware interface

interru emory addresses, special registers,
memory-m ed devices, I/O buses, …

hardware

CPU

keyboard/mouse

monitor

disks

network

…

memory

# goal: protection

run multiple applications, and …

keep them from crashing the OS

keep them from crashing each other

(keep parts of OS from crashing other parts?)

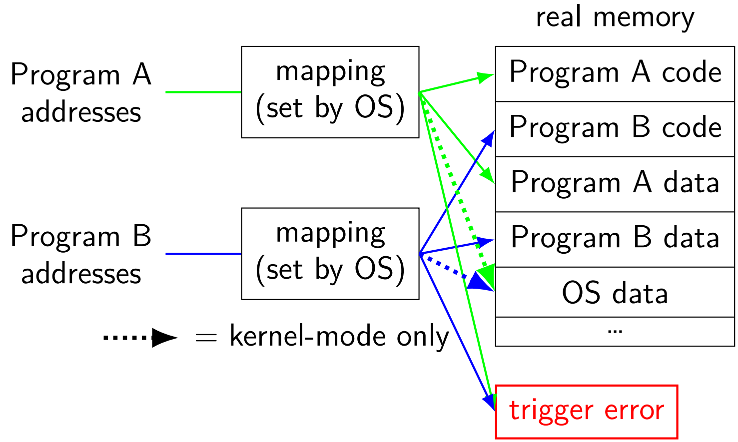
# mechanism 1: dual-mode operation

processor has two modes: kernel (privileged) and user

some operations require kernel mode

OS controls what runs in kernel mode

# mechanism 2: address translation



real memory

Program A addresses

Program B addresses

mapping (set by OS)

mapping (set by OS)

| |
|---|
| Program A code |
| Program B code |
| Program A data |
| Program B data |
| OS data |
| ... |

trigger error

⋯⋯► = kernel-mode only

# aside: alternate mechanisms

dual mode operation and address translation are common today

…so we'll talk about them a lot

not the only ways to implement operating system features

    (plausibly not even the most efficient…)

# problem: OS needs to respond to events

keypress happens?

program using CPU for too long?

…

# problem: OS needs to respond to events

keypress happens?

program using CPU for too long?

…

hardware support for running OS: *exception*

- need hardware support because CPU is running application instructions

# exceptions and dual-mode operation

rule: user code always runs in user mode

rule: only OS code ever runs in kernel mode

on *exception*: changes from user mode to kernel mode

…and is only mechanism for doing so

- how OS controls what runs in kernel mode

# exception terminology

CS 3330 terms:

interrupt: triggered by external event

- timer, keyboard, network, …

fault: triggered by program doing something “bad”

- invalid memory access, divide-by-zero, …

traps: triggered by explicit program action

- system calls

aborts: something in the hardware broke

# xv6 exception terms

everything is a called a trap

or sometimes an interrupt

no real distinction in *name* about kinds

# real world exception terms

it’s all over the place…

context clues

# kernel services

allocating memory? (change address space)

reading/writing to file? (communicate with hard drive)

read input? (communicate with keyboard)

all need privileged instructions!

need to run code in kernel mode

# hardware mechanism: deliberate exceptions

some instructions exist to trigger exceptions

still works like normal exception
- starts executing OS-chosen handler
- …in kernel mode

allows program requests privilieged instructions
- OS handler decides what program can request
- OS handler decides format of requests

# exercise: how many exceptions?

single-core OS with processes A, B, C

running process A

A prompts for input, then

A waits to read a keypress

while A is waiting for the keypress the OS runs B, then C

then keypress happens, and OS switches to A immediately

then A exits

exercise: how many exceptions?

# system call timeline (x86-64 Linux)

in user mode

(the standard library)

```
/* set arguments in registers */
mov $SYS_write, %rax
mov $FILENO_stdout, %rsi
mov $buffer, %rdi
mov $BUFFER_LEN, %r8
/* trigger exception */
syscall  // special instruction
```

```
// now use return value
testq %rax, %rax
```

in kernel mode

(the “kernel”)

```
syscall_handler:
  /* ... save registers and
         actually do read and
         set return value ... */
  /* go back to "user" code */
  iret  // special instruction
```

# system call timeline (x86-64 Linux)

in user mode

(the standard library)

```
/* set arguments in registers */
mov $SYS_write, %rax
mov $FILENO_stdout, %rsi
mov $buffer, %rdi
mov $BUFFER_LEN, %r8
/* trigger exception */
syscall  // special instruction
```

```
// now use return value
testq %rax, %rax
```

in kernel mode

(the “kernel”)

hardware knows to go here

because of pointer set during boot

```
syscall_handler:
  /* ... save registers and
        actually do read and
        set return value ... */
  /* go back to "user" code */
  iret  // special instruction
```

# system call timeline (x86-64 Linux)

in user mode

(the standard library)

```
/* set arguments in registers */
mov $SYS_write, %rax
mov $FILENO_stdout, %rsi
mov $buffer, %rdi
mov $BUFFER_LEN, %r8
/* trigger exception */
syscall  // special instruction
```

'priviliged' operations prohibited

```
// now use return value
testq %rax, %rax
```

in kernel mode

(the "kernel")

```
syscall_handler:
  /* ... save registers and
        actually do read and
        set return value ... */
  /* go back to "user" code */
  iret  // special instruction
```

# system call timeline (x86-64 Linux)

in user mode

(the standard library)

```
/* set arguments in registers */
mov $SYS_write, %rax
mov $FILENO_stdout, %rsi
mov $buffer, %rdi
mov $BUFFER_LEN, %r8
/* trigger exception */
syscall  // special instruction




// now use return value
testq %rax, %rax
```

in kernel mode

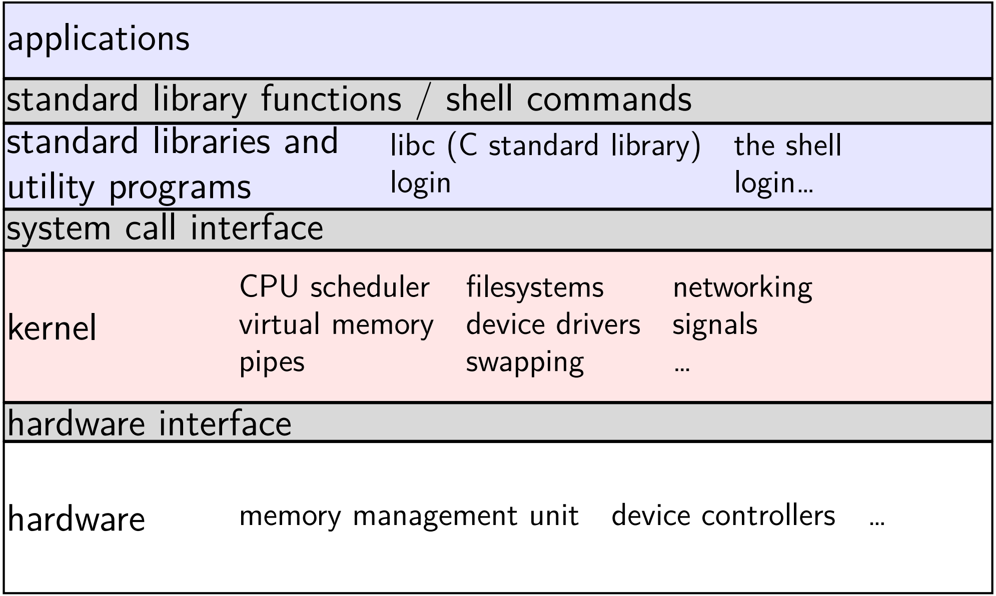
(the “kernel”)

‘priviliged’ operations allowed

(change memory layout, I/O, exceptions)

```
syscall_handler:
  /* ... save registers and
         actually do read and
         set return value ... */
  /* go back to "user" code */
  iret  // special instruction
```

# the classic Unix design



applications

standard library functions / shell commands

standard libraries and utility programs

libc (C standard library)
login

the shell
login...

system call interface

kernel

CPU scheduler
virtual memory
pipes

filesystems
device drivers
swapping

networking
signals
...

hardware interface

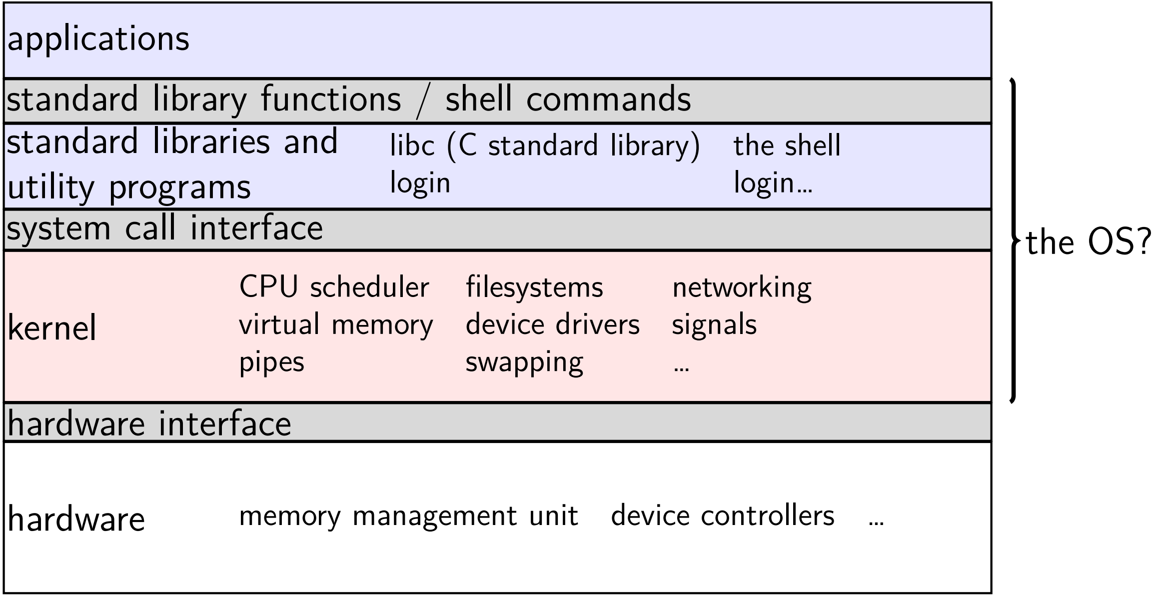
hardware

memory management unit

device controllers

...

# the classic Unix design



applications

standard library functions / shell commands

standard libraries and utility programs

libc (C standard library)

login

the shell

login…

system call interface

kernel

CPU scheduler

virtual memory

pipes

filesystems

device drivers

swapping

networking

signals

…

hardware interface

hardware

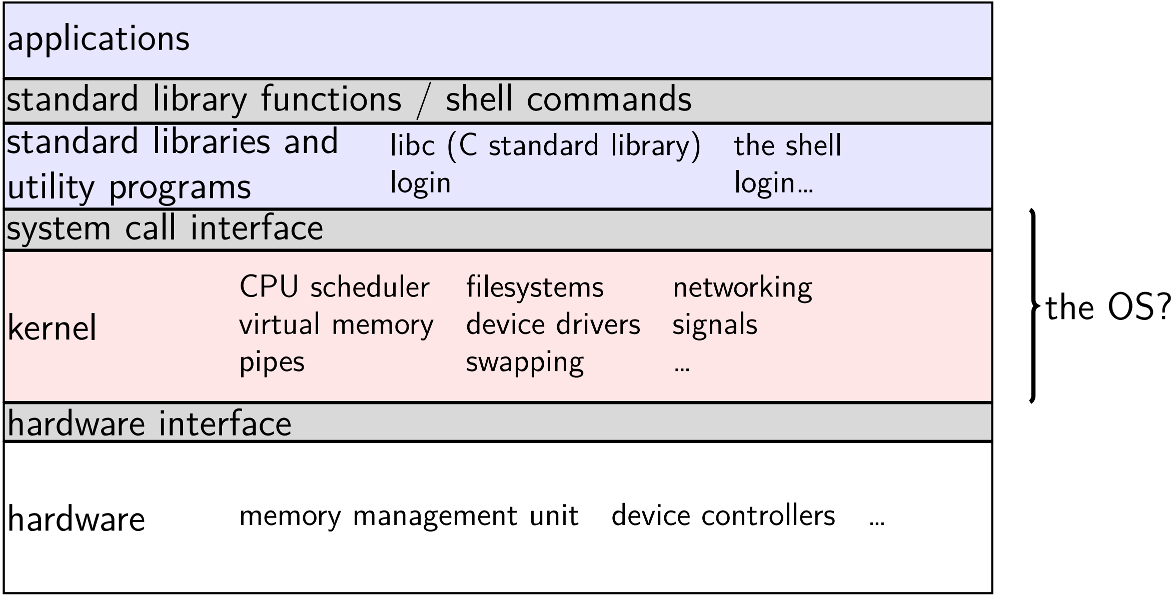
memory management unit

device controllers

…

the OS?

# the classic Unix design



applications

standard library functions / shell commands

standard libraries and utility programs

libc (C standard library)
login

the shell
login…

system call interface

kernel

CPU scheduler
virtual memory
pipes

filesystems
device drivers
swapping

networking
signals
…

hardware interface

hardware

memory management unit

device controllers

…

the OS?

# aside: is the OS the kernel?

OS = stuff that runs in kernel mode?

OS = stuff that runs in kernel mode + libraries to use it?

OS = stuff that runs in kernel mode + libraries + utility programs (e.g. shell, finder)?

OS = everything that comes with machine?

no consensus on where the line is

each piece can be replaced separately…

# xv6

we will be using an teaching OS called “xv6”

based on Sixth Edition Unix

modified to be multicore and use 32-bit x86 (not PDP-11)

# xv6 setup/assignment

first assignment — adding two simple xv6 system calls

includes xv6 download instructions

and link to xv6 book

# xv6 technical requirements

you will need a Linux environment

- we will supply one (VM on website), or get your own
- (it's probably possible to use OS X, but you need a cross-compiler and we don't have instructions)

...with qemu installed

- qemu (for us) = emulator for 32-bit x86 system
- Ubuntu/Debian package qemu-system-i386

# first assignment

get compiled and xv6 working

…toolkit uses an emulator

- could run on real hardware or a standard VM, but a lot of details
- also, emulator lets you use GDB

# xv6: what's included

Unix-like kernel

- very small set of syscalls
- some less featureful (e.g. exit without exit status)

userspace library

- very limited

userspace programs

- command line, ls, mkdir, echo, cat, etc.
- some self-testing programs

# xv6: echo.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
  int i;

  for(i = 1; i < argc; i++)
    printf(1, "%s%s", argv[i], i+1 < argc ? " " : "\n");
  exit();
}
```

# xv6: echo.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
  int i;

  for(i = 1; i < argc; i++)
    printf(1, "%s%s", argv[i], i+1 < argc ? " " : "\n");
  exit();
}
```

# xv6: echo.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
  int i;

  for(i = 1; i < argc; i++)
    printf(1, "%s%s", argv[i], i+1 < argc ? " " : "\n");
  exit();
}
```

# xv6 demo

# backup slides

# common goal: hide complexity

hiding complexity

# common goal: hide complexity

hiding complexity

competing applications — failures, malicious applications
  - text editor shouldn't need to know if browser is running

varying hardware — diverse and changing interfaces
  - different keyboard interfaces, disk interfaces, video interfaces, etc.
  - applications shouldn't change

# common goal: for application programmer

write once for lots of hardware

avoid reimplementing common functionality

don't worry about other programs