CS 4414 — introduction

Changelog

Changes made in this version not seen in first lecture:

27 Aug 2019: remove mention of department login server being alternative for xv6, though it may be useful for other assignments

course webpage

`https://www.cs.virginia.edu/~cr4bd/4414/F2019/`

linked off Collab

homeworks

there will be programming assignments

...mostly in C or C++

- possibly one assignment in Python

one or two weeks

- if two weeks "checkpoint" submission after first week

two week assignments worth more

schedule is aggressive...

xv6

some assignments will use xv6, a teaching operating system

simplified OS based on an old Unix version

- built by some people at MIT

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

midterm and final

let us know soon if you can't make the midterm

textbook

recommended textbook:
Operating Systems: Principles and Practice

no required textbook

alternative: 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

no sharing code, pesudocode, detailed descriptions of code

no 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

cheating: quizzes

don't

quizzes: also individual

don't share answers

don't IM people for answers

don't ask on StackOverflow for answers

getting help

Piazza

office hours (will be posted soon)

emailing me

what is an operating system? (1)

several overalpping definitions

abstraction layer over hardware?

alternative to hardware interface?

several distinct jobs relating to sharing/accessing resources?

what is an operating system? (2)

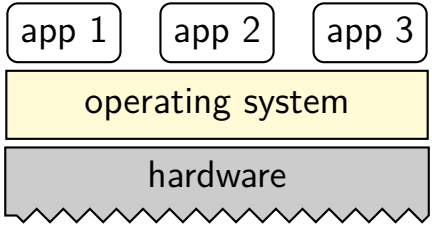
layer of software to provide access to HW

abstraction of complex hardware

protected access to shared resources

communication

security



history: computer operator

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

what is an operating system? (3)

software providing a more convenient/featureful machine interface

what is an operating system? (4)

software performing certain jobs for computer system:

textbook's roles

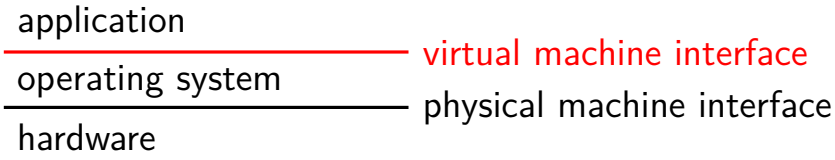
referee — resource sharing, protection, isolation

illusionist — clean, easy abstractions

glue — common services

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

the virtual machine interface



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 $\approx$ hardware interface (but maybe not the real hardware)

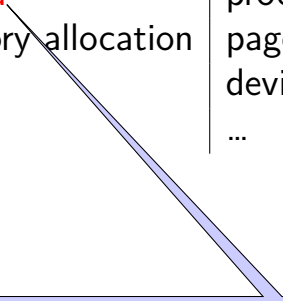
- aid reusing existing raw hardware-targeted code
- different "application programmer"

process virtual machine

process VM	real hardware
thread	processors
memory allocation	page tables
files	devices
...	...

process virtual machine

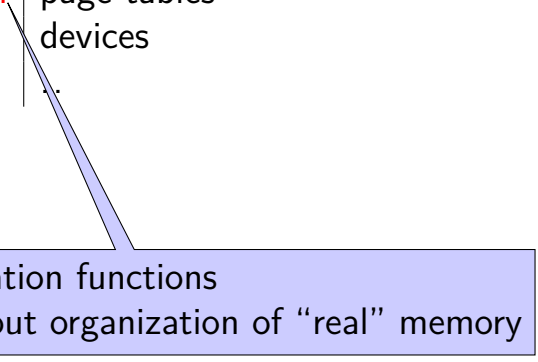
process VM	real hardware
thread	processors
memory allocation	page tables
files	devices
...	...



(virtually) infinite "threads" ($\sim$ virtual CPus)
no matter number of CPUs

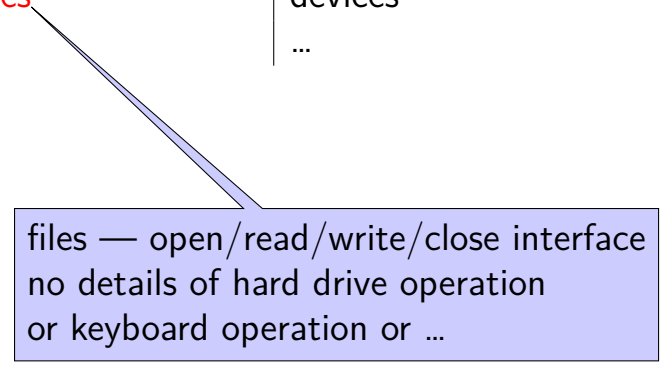
process virtual machine

process VM	real hardware
thread	processors
memory allocation	page tables
files	devices
...	...

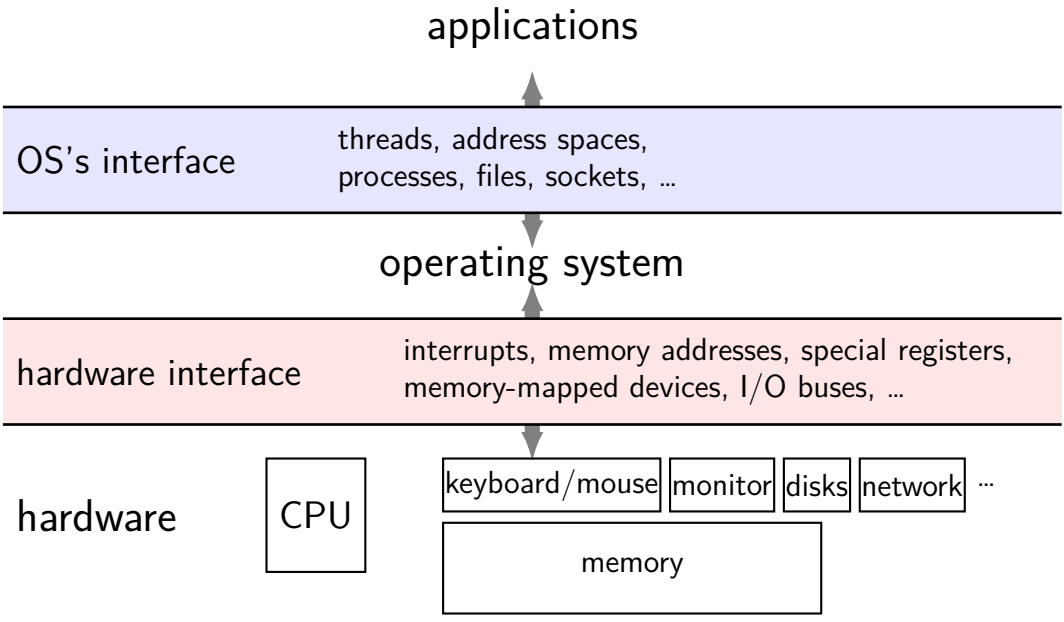


process virtual machine

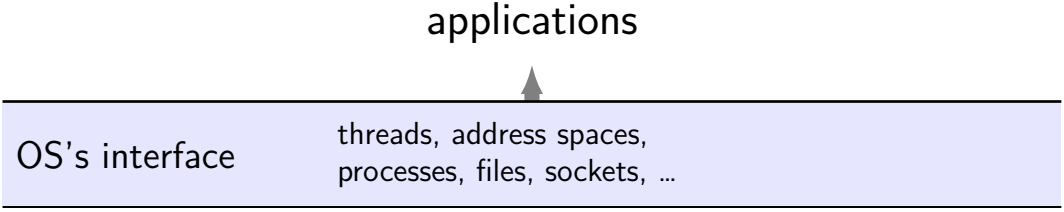
process VM	real hardware
thread	processors
memory allocation	page tables
files	devices
...	...



the abstract virtual machine



abstract VM: application view

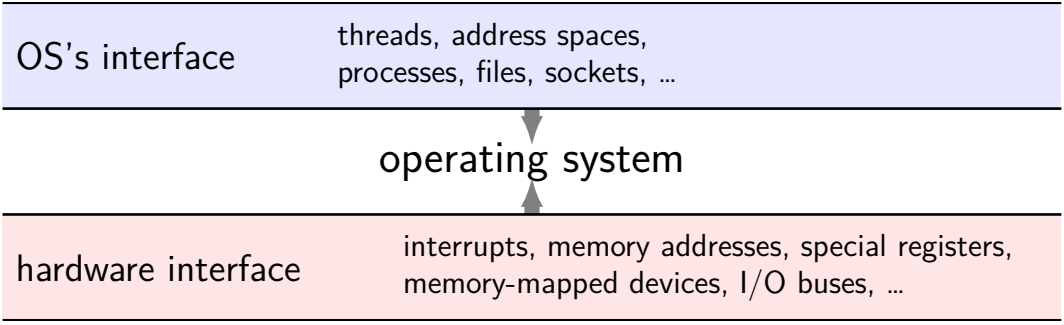


the application's "machine" is the operating system

no hardware I/O details visible — future-proof

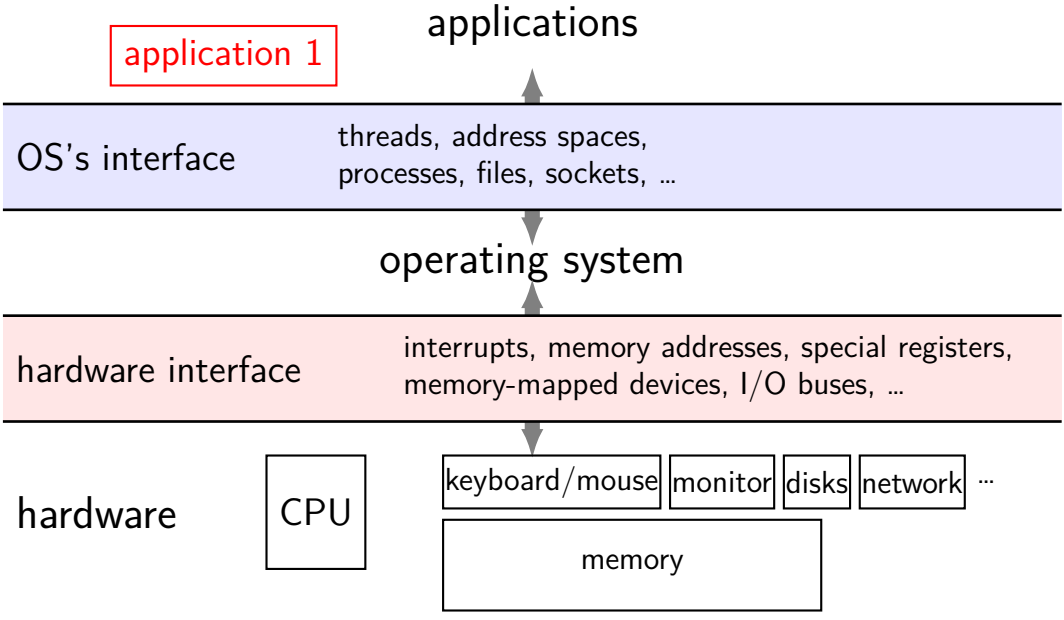
more featureful interfaces than real hardware

abstract VM: OS view

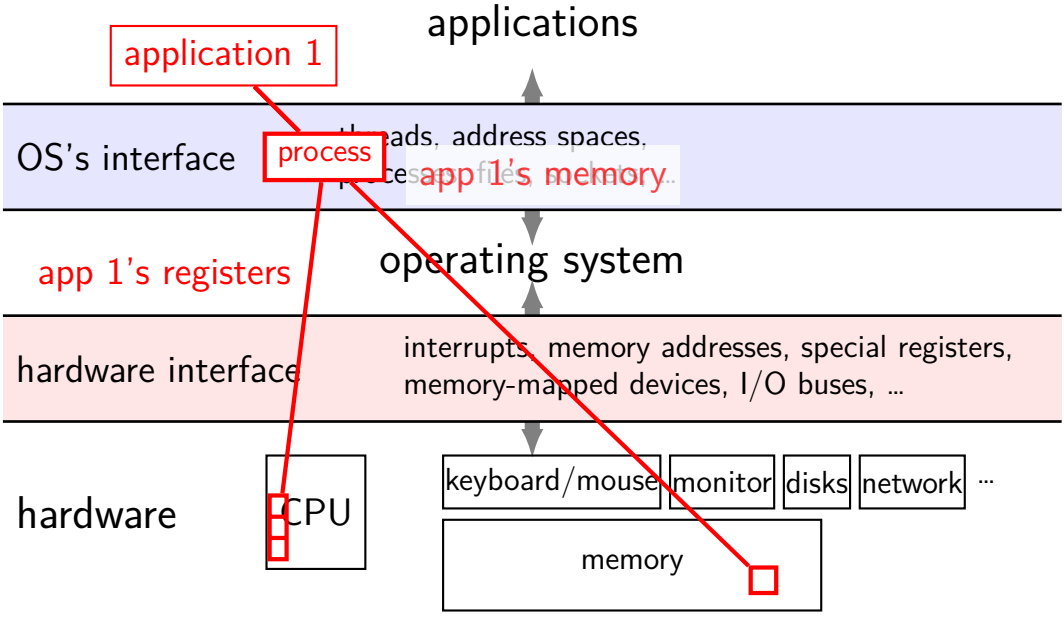


operating system's job: translate one interface to another

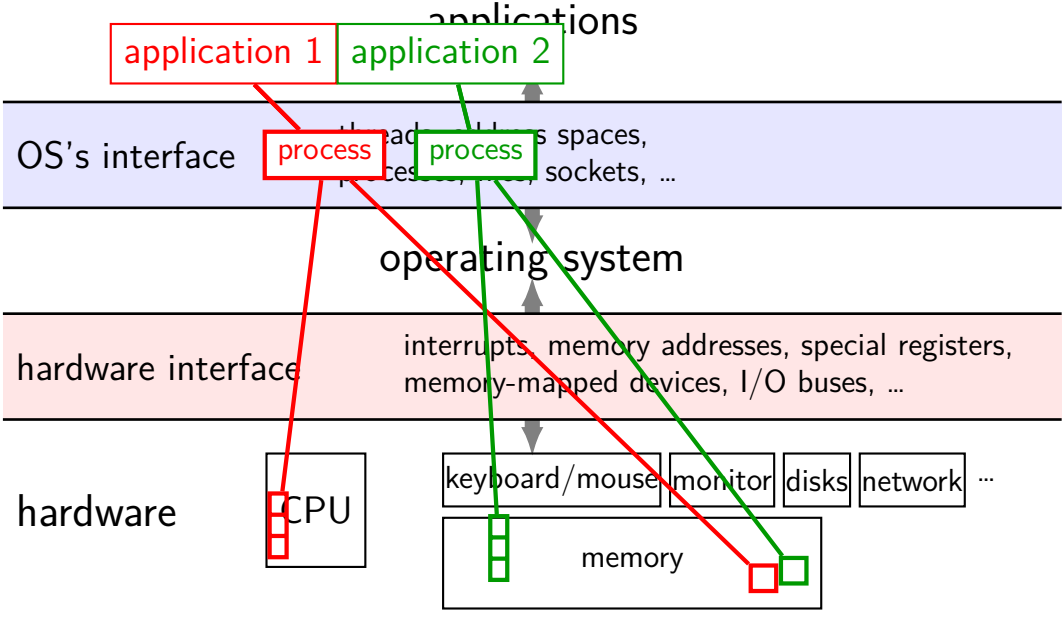
program → process → CPU and memory



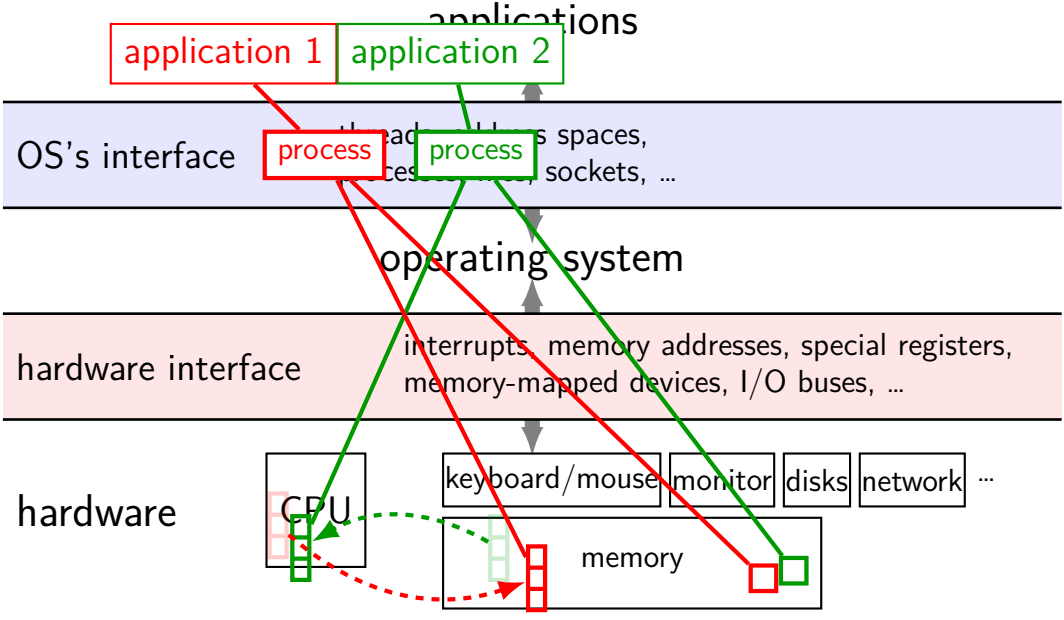
program → process → CPU and memory



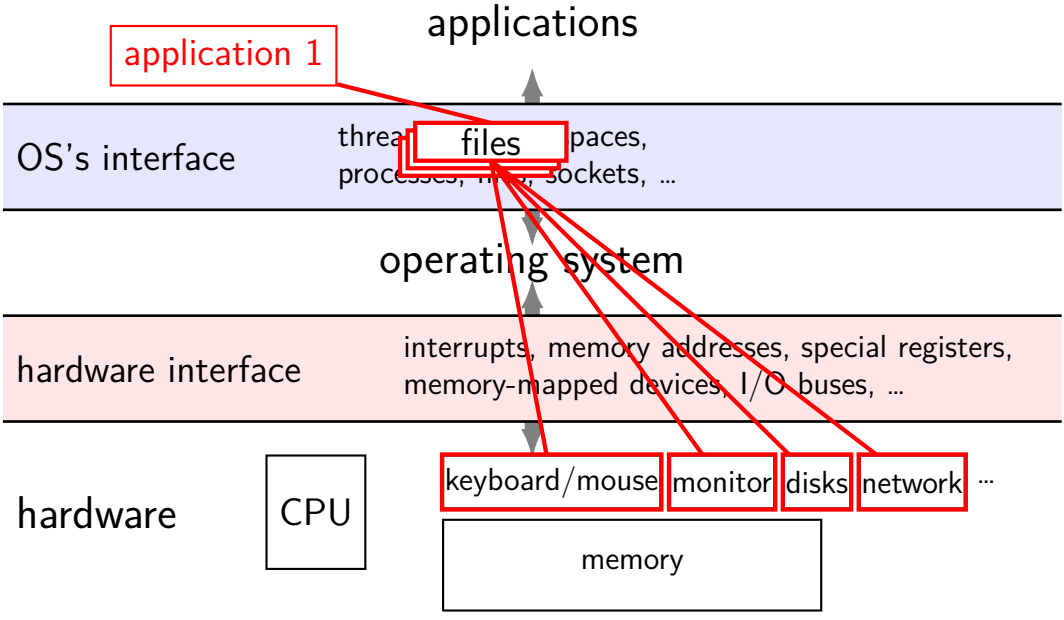
program → process → CPU and memory



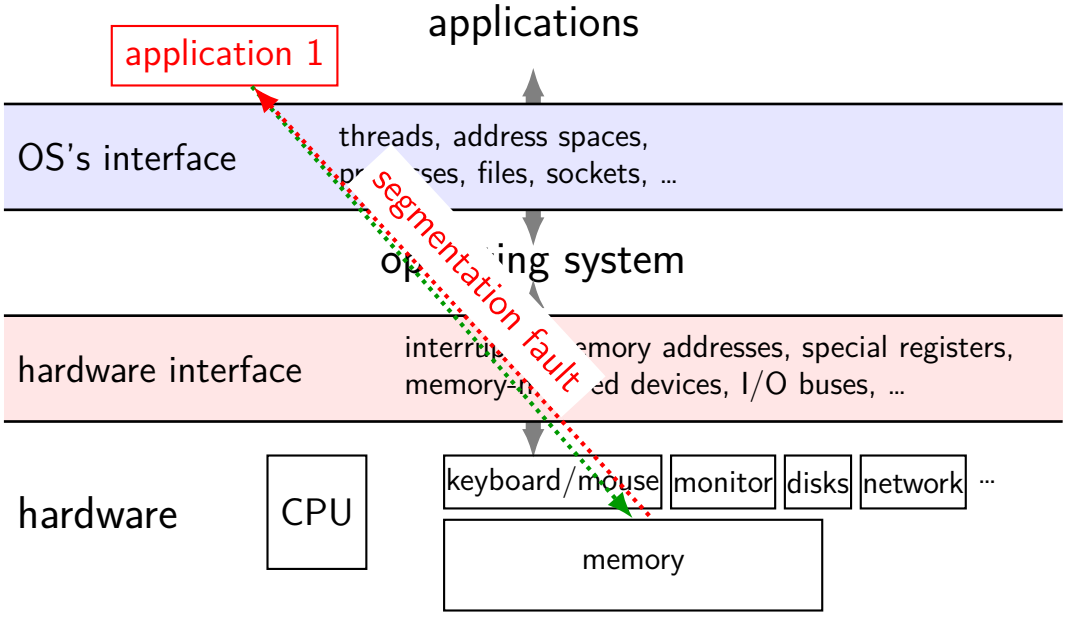
program → process → CPU and memory



files → input/output



security and protection



The Process

process = thread(s) + address space

illusion of dedicated machine:

- thread = illusion of own CPU
- address space = illusion of own 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?)

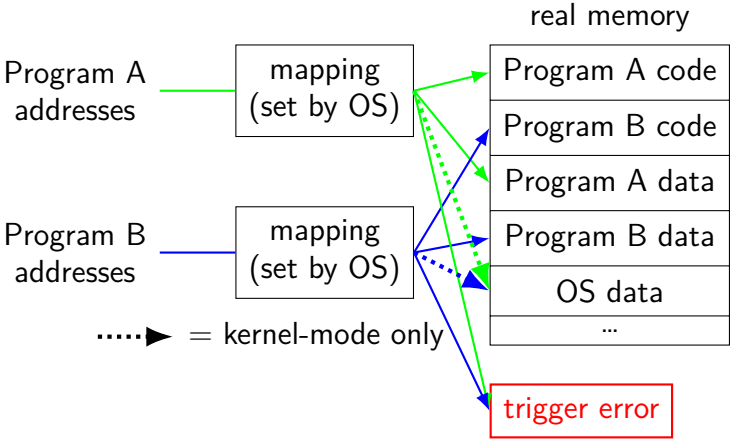
mechanism 1: dual-mode operation

processor has two modes: kernel (privileged) and user

some operations require kernel mode

OS controls what runs in kernel mode

mechanism 2: address translation



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

system call timeline (x86-64 Linux)

in user mode (the standard library)	in kernel mode (the "kernel")

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

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

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

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

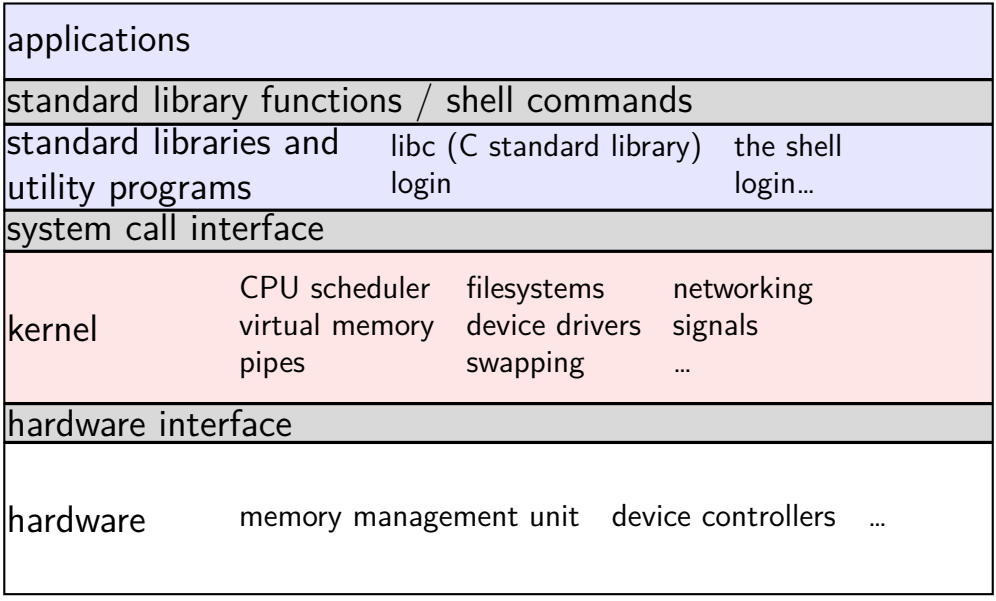
(the "kernel")

'priviliged' operations allowed

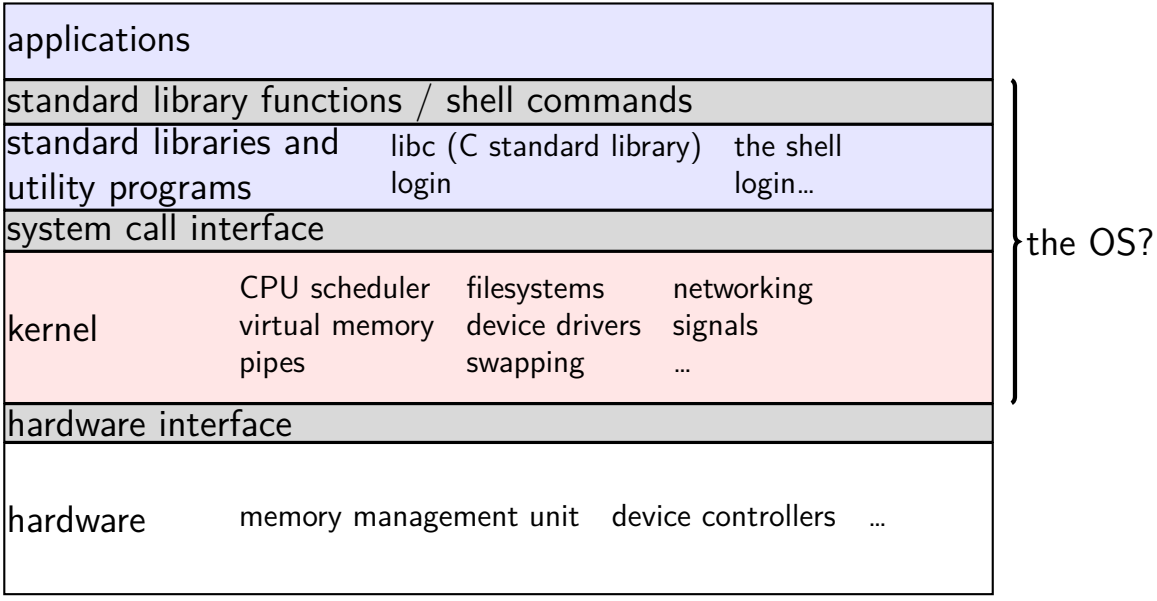
(change memory layout, I/O, exceptions)

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

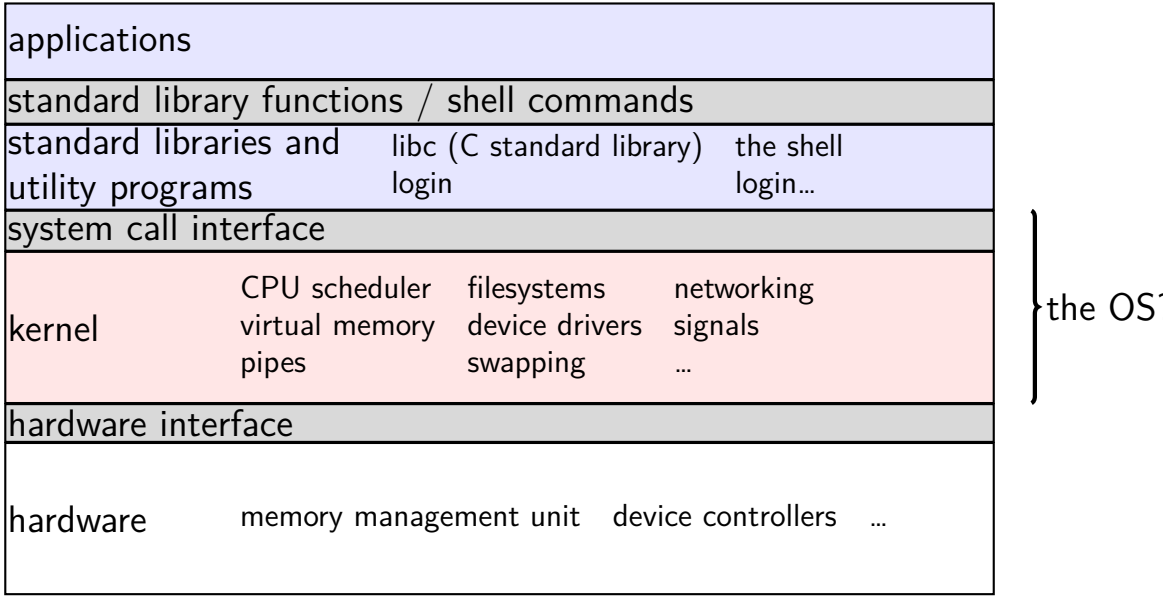
the classic Unix design



the classic Unix design



the classic Unix design



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 simple xv6 system call

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
- should also have department lab accounts (but not usable for xv6)
- (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

syscalls in xv6

fork, exec, exit, wait, kill, getpid — process control

open, read, write, close, fstat, dup — file operations

mknod, unlink, link, chdir — directory operations

...

write syscall in xv6: user mode

syscall.h

```
...
#define SYS_write   16
...
```

main.c

```
...
write(1,
      "Hello, World!\n",
      14);
...
```

usys.S

(after macro replacement)

```
#include "syscall.h"
// ...
.globl write
write:
    /* 16 = SYS_write */
    movl $16, %eax
    /* 0x40 = T_SYSCALL */
    int $0x40
    ret
```

write syscall in xv6: user mode

syscall.h

```
...
#define SYS_write   16
...
```

main.c

```
...
write(1,
      "Hello, World!\n",
      14);
...
```

usys.S

(after macro replacement)

```
#include "syscall.h"
// ...
.globl write
write:
    /* 16 = SYS_write */
    movl $16, %eax
    /* 0x40 = T_SYSCALL */
    int $0x40
    ret
```

interrupt — trigger an exception similar to a keypress
parameter (0x40 in this case) — type of exception

write syscall in xv6: user mode

syscall.h

```
...
#define SYS_write   16
...
```

main.c

```
...
write(1,
      "Hello, World!\n",
      14);
...
```

usys.S

(after macro replacement)

```
#include "syscall.h"
// ...
.globl write
write:
    /* 16 = SYS_write */
    movl $16, %eax
    /* 0x40 = T_SYSCALL */
    int $0x40
    ret
```

xv6 syscall calling convention:
eax = syscall number
otherwise: same as 32-bit x86 calling convention (arguments on stack)

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

`lidt` —
function (in x86.h) wrapping lidt instruction

sets the *interrupt descriptor table*
table of *handler functions* for each interrupt type

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

(from mmu.h):

```
// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap gate, 0 for an interrupt gate.
//   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d)                \
```

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

set the T_SYSCALL (= 0x40) interrupt to be callable from user mode via **int** instruction

(otherwise: triggers fault like privileged instruction)

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

set it to use the kernel "code segment"
meaning: run in kernel mode
(yes, code segments specifies more than that — nothing we care about)

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

1: do not disable interrupts during syscalls
e.g. keypress handling can interrupt slow syscall
con: makes writing system calls safely more complicated
pro: slow system calls don't stop timers, keypresses, etc. from working

xv6 choice: interrupts *are* disabled during non-syscall exception handling
(e.g. don't worry about keypress being handled while timer being handled

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

`vectors[T_SYSCALL]` — OS function for processor to run
set to pointer to assembly function `vector64`

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

`vectors[T_SYSCALL]` — OS function for processor to run
set to pointer to assembly function `vector64`

hardware jumps here

vectors.S

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
...
```

trapasm.S

```
alltraps:
  ...
  call trap
  ...
  iret
```

trap.c

```
void
trap(struct trapframe *tf)
{
...
```

write syscall in xv6: the trap function

trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

write syscall in xv6: the trap function

trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

struct trapframe — set by assembly
interrupt type, application registers, ...
example: `tf->eax` = old value of eax

write syscall in xv6: the trap function

trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

`myproc()` — pseudo-global variable represents currently running process

much more on this later in semester

write syscall in xv6: the trap function

trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

syscall() — actual implementations uses `myproc()->tf` to determine what operation to do for program

write syscall in xv6: the syscall function

syscall.c

```
static int (*syscalls[])(void) = {
...
[SYS_write]  sys_write,
...
};

...

void
syscall(void)
{
...
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
...
```

write syscall in xv6: the syscall function

syscall.c

```
static int (*syscalls[])(void) = {
...
[SYS_write]   sys_write,
...
};

...

void
syscall(void)
{
...
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
...
```

array of functions — one for syscall

'[number] value': syscalls[number] = value

write syscall in xv6: the syscall function

syscall.c

```
static int (*syscalls[])(void) = {
...
[SYS_write]   sys_write,
...
};

...

void
syscall(void)
{
...
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
...
```

(if system call number in range)
call sys_...function from table
store result in user's eax register

write syscall in xv6: the syscall function

syscall.c

```
static int (*syscalls[])(void) = {
...
[SYS_write]   sys_write,
...
};

...

void
syscall(void)
{
...
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
...
```

result assigned to eax
(assembly code this returns to
copies `tf->eax` into `%eax`)

write syscall in xv6: sys_write

sysfile.c

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

write syscall in xv6: sys_write

sysfile.c

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

utility functions that read arguments from user's stack
returns -1 on error (e.g. stack pointer invalid)
(more on this later)
(note: 32-bit x86 calling convention puts all args on stack)

write syscall in xv6: sys_write

sysfile.c

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

actual internal function that implements writing to a file (the terminal counts as a file)

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

`trap` returns to `alltraps`

alltraps restores registers from `tf`, then returns to user-mode

hardware jumps here

vectors.S

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
...
```

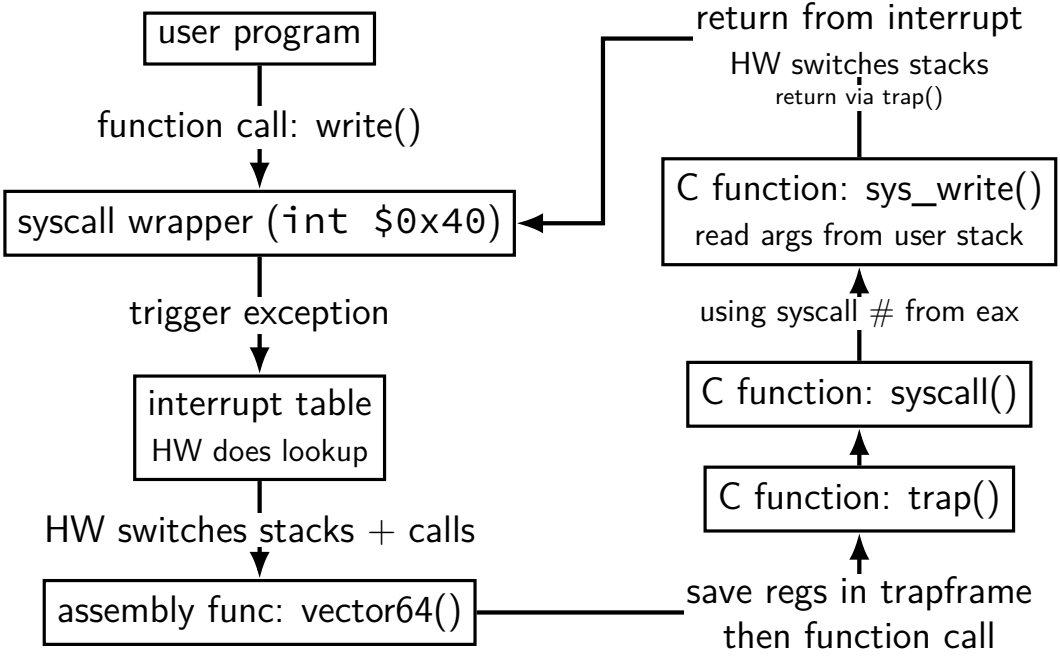
trapasm.S

```
alltraps:
  ...
  call trap
  ...
  iret
```

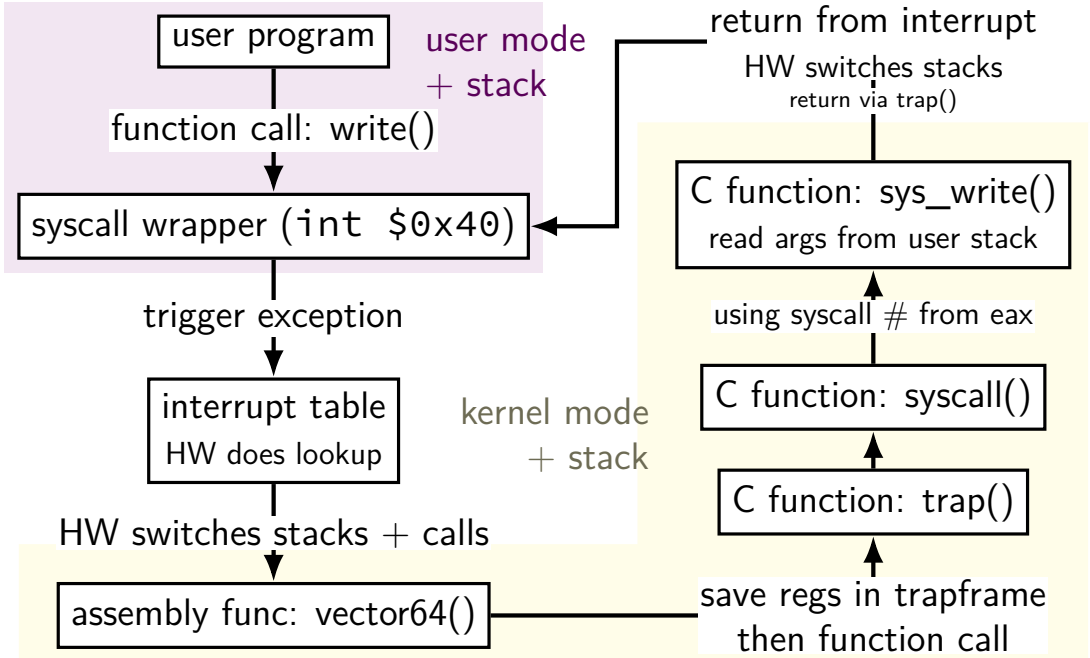
trap.c

```
void
trap(struct trapframe *tf)
{
...
```

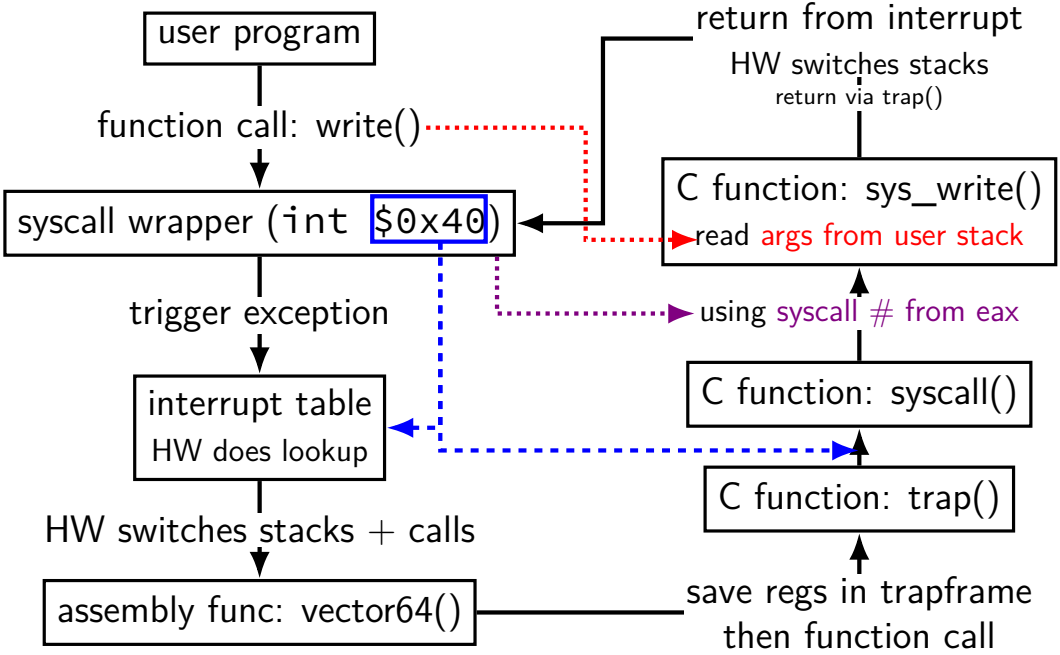
write syscall in xv6



write syscall in xv6



write syscall in xv6



xv6intro homework

get familiar with xv6 OS

add a new system call: `writecount()`

returns total number of times write call happened

homework steps

system call implementation: sys_writecount

- hint in writeup: imitate sys_uptime
- need a counter for number of writes

add writecount to several tables/lists

- (list of handlers, list of library functions to create, etc.)
- recommendation: imitate how other system calls are listed

create a userspace program that calls writecount

- recommendation: copy from given programs

note on locks

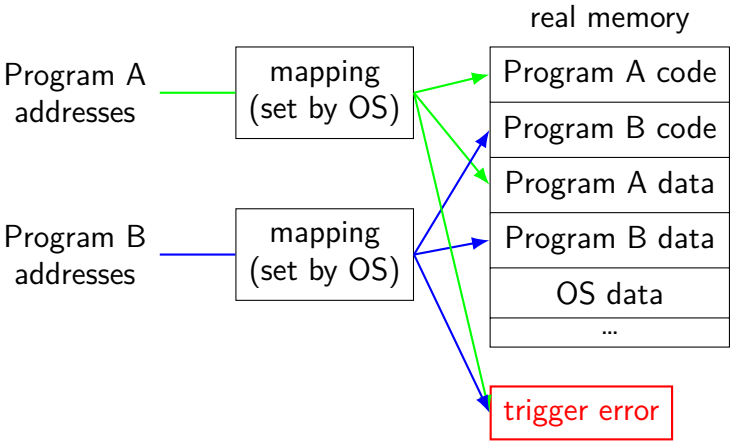
some existing code uses acquire/release

you do not have to do this

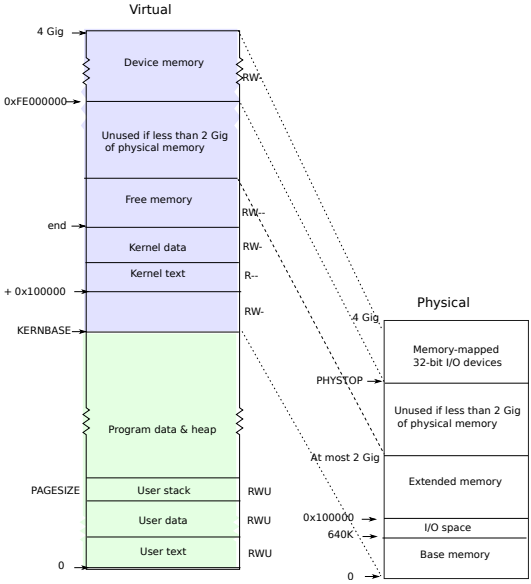
only for multiprocessor support

...but, copying what's done for ticks would be correct

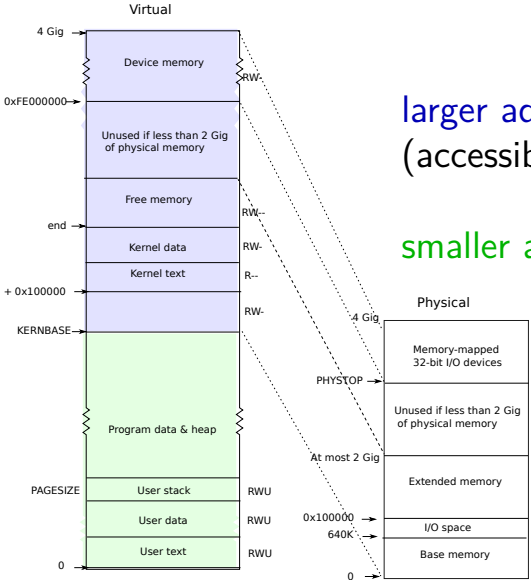
recall: address translation



xv6 memory layout



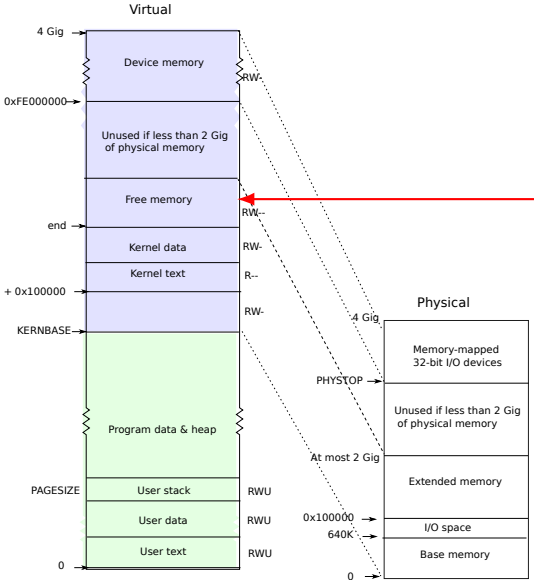
xv6 memory layout



larger addresses are for kernel (accessible in kernel mode *only*)

smaller addresses are for applications

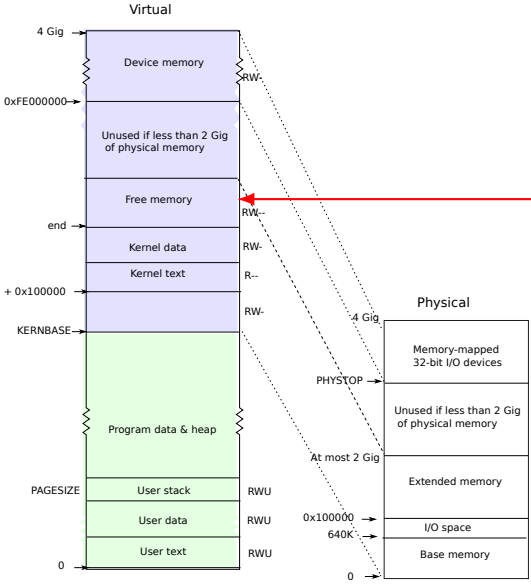
xv6 memory layout



kernel stack allocated here

processor switches stacks
when execption/interrupt/...happens
location of stack stored
in special "task state selector"

xv6 memory layout



kernel stack allocated here

one kernel stack per process

change which one exceptions use as part of switching which processes is active on a processor

aside: nested exceptions

x86 switches to kernel stack on exception...

assuming it's switching to kernel mode

system call or timer interrupt in user mode

- start at top of kernel stack

timer interrupt during system call

- continue using current kernel stack

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