last time

preview: OS enforces restrictions

user/group IDs

access control list idea
- list of who can access what

chmod rwx for owner/one group/everyone else

more general access control lists

superuser/root = always has permission
- but still goes through OS to do things

set-user-ID: controlled access to special functionality
- mark specific programs to have extra access

reminder: warmup

Q1

problem: existing executable breaks when .so file updated

if .a instead of .so:

executable includes machine code from *old version* of library
because linking process includes .a file

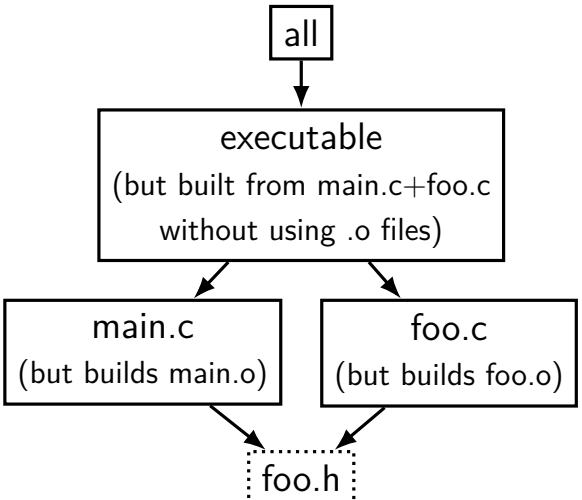
bar.h has both prototypes

would not compile+link
in C (but not C++) — can't have two prototypes for one function
(in C++ — would need to *define* both versions of function)

Q2: erroneous makefile (1)

```
all: executable

~~foo.c~~ (should be foo.o): foo.h
    gcc -Wall -c foo.c

~~main.c~~ (should be main.o): foo.h
    gcc -Wall -c main.c

executable: ~~foo.c main.c~~ (should be foo.o main.o)
    gcc -Wall -o executable ~~foo.c main.c~~ (should be foo.o main.o)

clean:
    rm --force main.o foo.o executable
```

Q2: erroneous makefile (2)

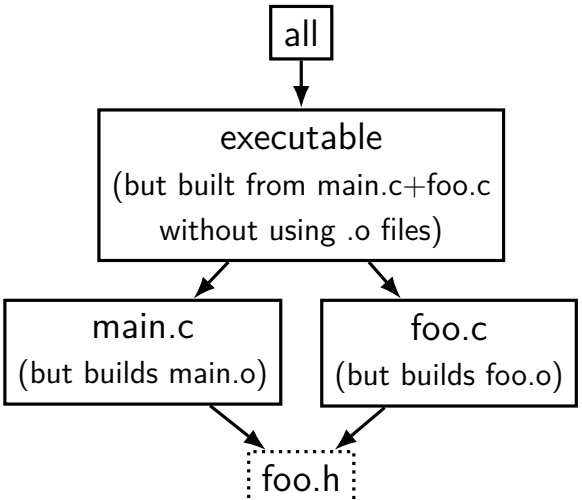


if foo.h more recent than main.c, fo
builds main.o from main.c
(compiles main.c once)
builds foo.o from foo.c (A)
executable need not be updated
(if main.c/foo.c old) (C)

if foo.c or main.c more recent than
builds executable from foo.c, ma
(compile main.c once)

if both true: main.c compiled twice

Q2: erroneous makefile (2)

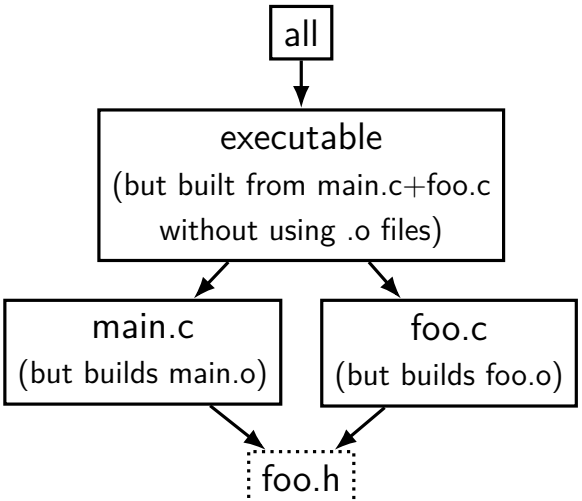


if foo.h more recent than main.c, fo
builds main.o from main.c
(compiles main.c once)
builds foo.o from foo.c (A)
executable need not be updated
(if main.c/foo.c old) (C)

if foo.c or main.c more recent than
builds executable from foo.c, mai
(compile main.c once)

if both true: main.c compiled twice

Q2: erroneous makefile (2)

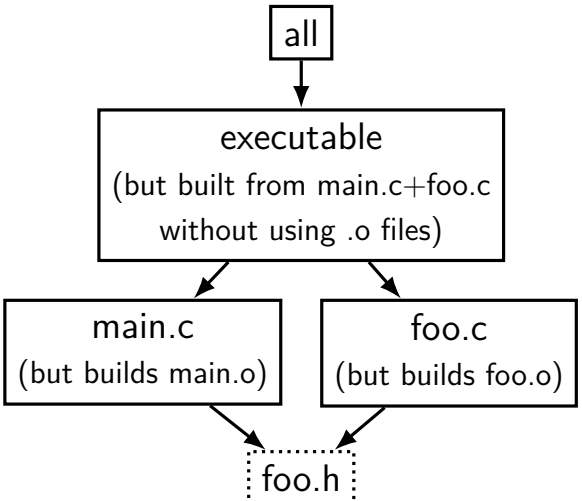


if foo.h more recent than main.c, fo
builds main.o from main.c
(compiles main.c once)
builds foo.o from foo.c (A)
executable need not be updated
(if main.c/foo.c old) (C)

if foo.c or main.c more recent than
builds executable from foo.c, ma
(compile main.c once)

if both true: main.c compiled twice

Q2: erroneous makefile (2)



if foo.h more recent than main.c, fo
builds main.o from main.c
(compiles main.c once)
builds foo.o from foo.c (A)
executable need not be updated
(if main.c/foo.c old) (C)

if foo.c or main.c more recent than
builds executable from foo.c, ma
(compile main.c once)

if both true: main.c compiled twice

Q3

```
logo.h: logo.bmp
```

`xxd` reads `logo.bmp` — is prerequisite

`xxd` writes `logo.h` — is target

Q4

`main.o: logo.h graphics.h`

don't need command because it's in `%.o: %.c` rule (but not error)

don't need main.c because it's in `%.o: %.c` rule (but not error)

do need `logo.h`, `graphics.h`

shouldn't have `stdio.h` since it's not in this directory

could have `/usr/include/stdio.h` or similar

Q6

want two files:

- one r/w for A, B only
- one r/w for A, C only

group with A:

- owned by B, user rw, group-with-A rw, other no access
- owner by C, user rw, group-with-A rw, other no access

group with B + group with C:

- owned by A, user rw, group-with-B rw, other no access
- owner by A, user rw, group-with-C rw, other no access

things programs on portal shouldn't do

read other user's files

modify OS's memory

read other user's data in memory

hang the entire system

things programs on portal shouldn't do

read other user's files

modify OS's memory

read other user's data in memory

hang the entire system

privileged operation: problem

how can hardware (HW) plus operating system (OS) allow:

- read your own files from hard drive

but disallow:

- read others files from hard drive

some ideas

OS tells HW ‘okay’ parts of hard drive before running program code

- complex for hardware and for OS

some ideas

OS tells HW ‘okay’ parts of hard drive before running program code

- complex for hardware and for OS

OS verifies your program’s code can’t do bad hard drive access

- no work for HW, but complex for OS
- may require compiling differently to allow analysis

some ideas

OS tells HW 'okay' parts of hard drive before running program code

- complex for hardware and for OS

OS verifies your program's code can't do bad hard drive access

- no work for HW, but complex for OS
- may require compiling differently to allow analysis

OS tells HW to only allow OS-written code to access hard drive

- that code can enforce only 'good' accesses
- requires program code to call OS routines to access hard drive
- relatively simple for hardware

kernel mode

extra one-bit register: "are we in *kernel mode*"

other names: privileged mode, supervisor mode, ...

not in kernel mode = *user mode*

certain operations only allowed in kernel mode

privileged instructions

example: talking to any I/O device

what runs in kernel mode?

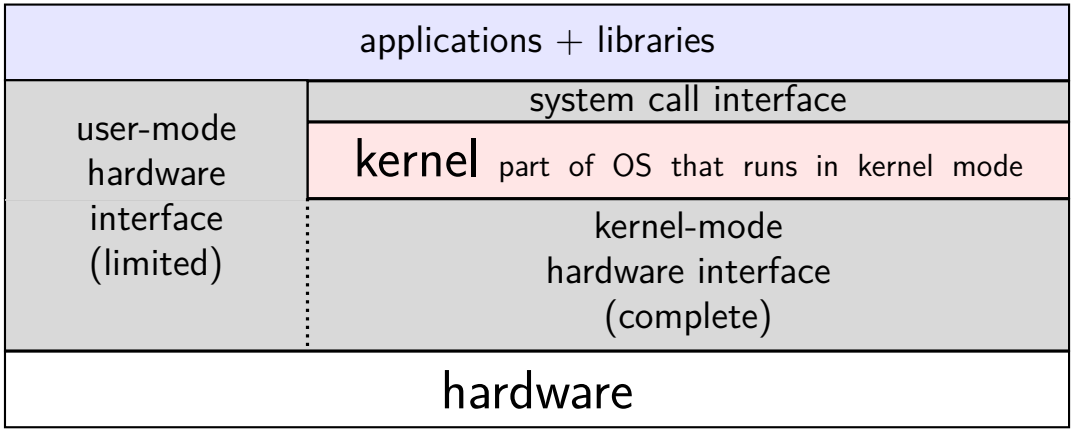
system boots in kernel mode

OS switches to user mode to run program code

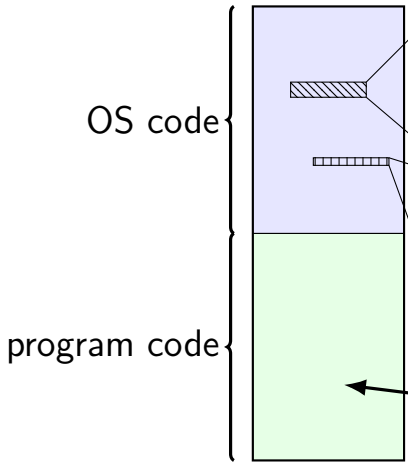
next topic: when does system switch back to kernel mode?

- how does OS tell HW where the (trusted) OS code is?

hardware + system call interface



calling the OS?




```
void readFromDiskInto(int diskLocation, char *dest) {
    ...
    runPrivilegedInstruction(...);
    ...
}
```

```
void readFileSafely(const char *name, char *dest) {
    if (canCurrentProgramCanAccessFile(name)) {
        readFromDiskInto(lookupFile(name), dest)
    }
}
```

how do we let this code run `readFromSafely` in kernel mode but not `readFromDisk`?

controlled entry to kernel mode (1)

special instruction: “make system call”

similar idea as `call` instruction — jump to function elsewhere (and allow that function to return later)

runs OS code in kernel mode at location specified earlier

OS sets up at boot

location can’t be changed without privilieged instrution

controlled entry to kernel mode (2)

OS needs to make specified location:

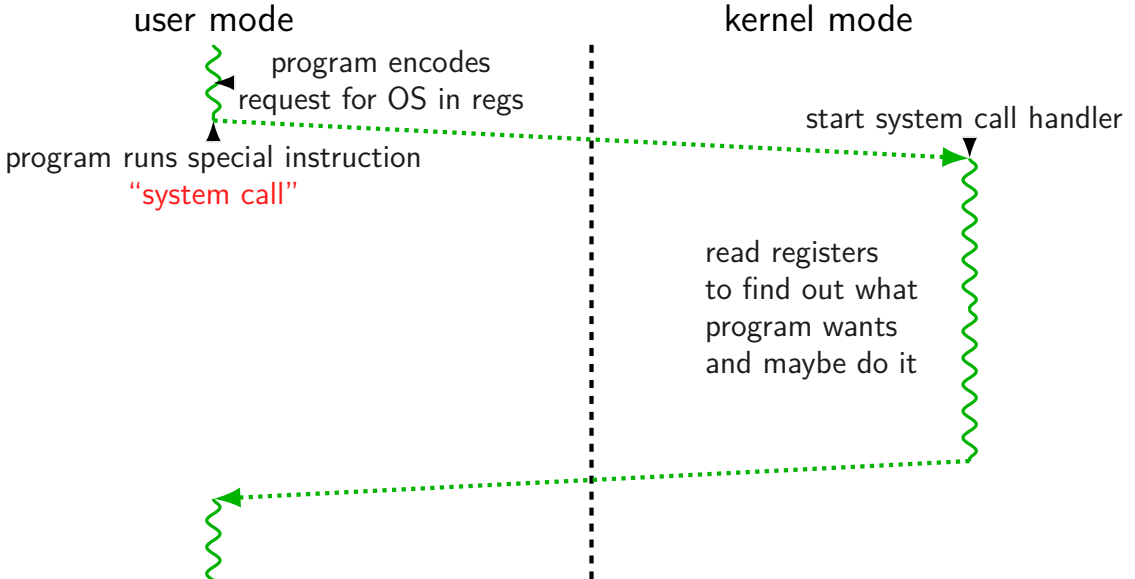
figure out what operation the program wants

- calling convention, similar to function arguments + return value

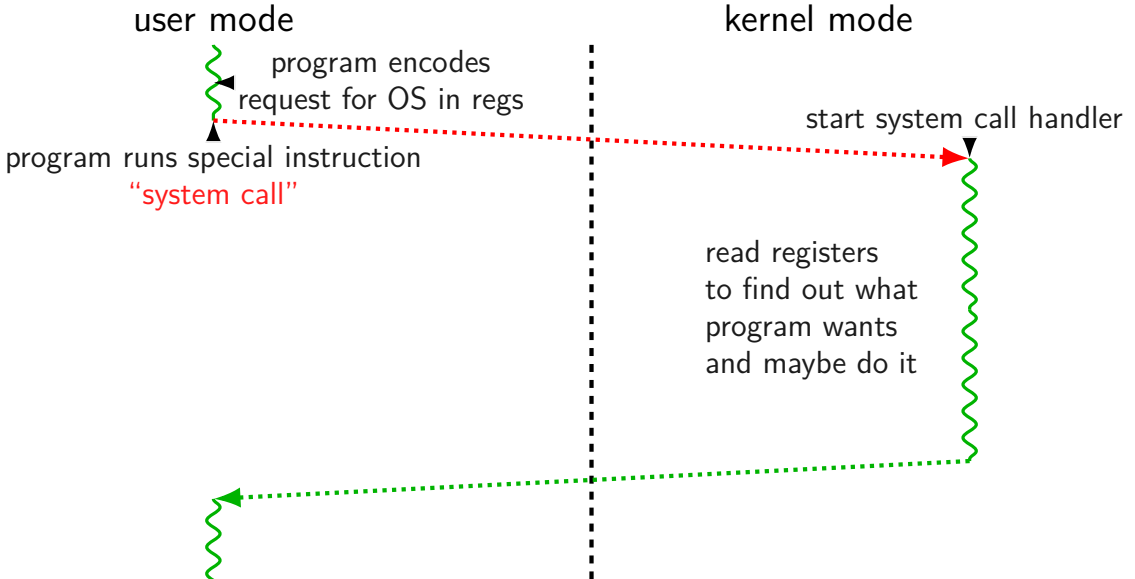
be "safe" — not allow the program to do 'bad' things

- example: checks whether current program is allowed to read file before reading it
- requires exceptional care — program can try weird things

system call process



system call process



system call terminology

some inconsistency:

system call = event of entering kernel mode on request?

system call = whole porcess from beginning to end?

same issue as with ‘function call’

is it just starting the function, or the whole time the function runs?

keeping permissions?

which of the following would still be secure?

A. performing authorization checks in the standard library in addition to system call handlers

B. performing authorization checks in the standard library instead of system call handlers

C. making the user ID a system call argument rather than storing it persistently in the OS's memory

Linux x86-64 system calls

special instruction: `syscall`

runs OS specified code in kernel mode

Linux syscall calling convention

before `syscall`:

`%rax` — system call number

`%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9` — args

after `syscall`:

`%rax` — return value

on error: `%rax` contains -1 times “error number”

almost the same as normal function calls

Linux x86-64 hello world

```
.globl _start
.data
hello_str: .asciz "Hello, World!\n"
.text
_start:
  movq $1, %rax # 1 = "write"
  movq $1, %rdi # file descriptor 1 = stdout
  movq $hello_str, %rsi
  movq $15, %rdx # 15 = strlen("Hello, World!\n")
  syscall

  movq $60, %rax # 60 = exit
  movq $0, %rdi
  syscall
```

approx. system call handler

```
sys_call_table:
    .quad handle_read_syscall
    .quad handle_write_syscall
    // ...

handle_syscall:
    ... // save old PC, etc.
    pushq %rcx // save registers
    pushq %rdi
    ...
    call *sys_call_table(,%rax,8)
    ...
    popq %rdi
    popq %rcx
    return_from_exception
```

Linux system call examples

`mmap`, `brk` — allocate memory

`fork` — create new process

`execve` — run a program in the current process

`_exit` — terminate a process

`open`, `read`, `write` — access files

`socket`, `accept`, `getpeername` — socket-related

Linux system call examples

`mmap`, `brk` — allocate memory

`fork` — create new process

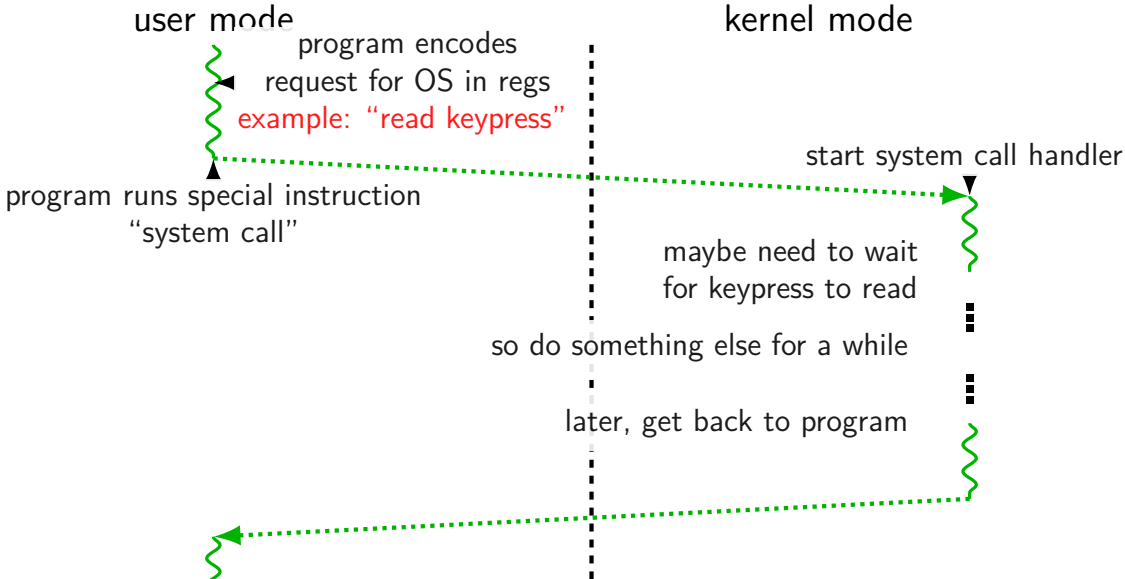
`execve` — run a program in the current process

`_exit` — terminate a process

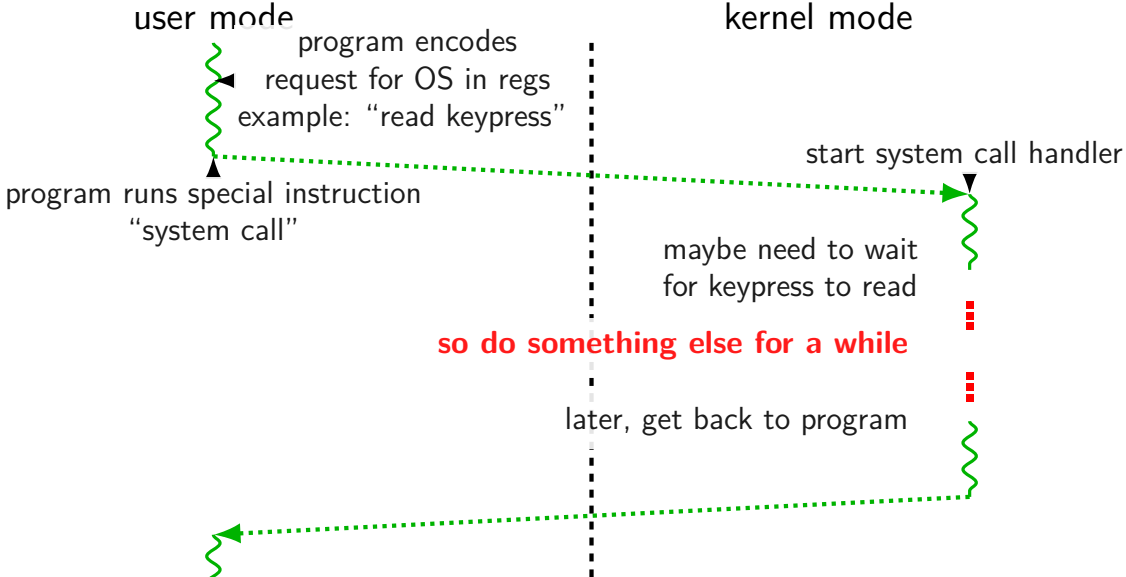
`open`, `read`, `write` — access files

`socket`, `accept`, `getpeername` — socket-related

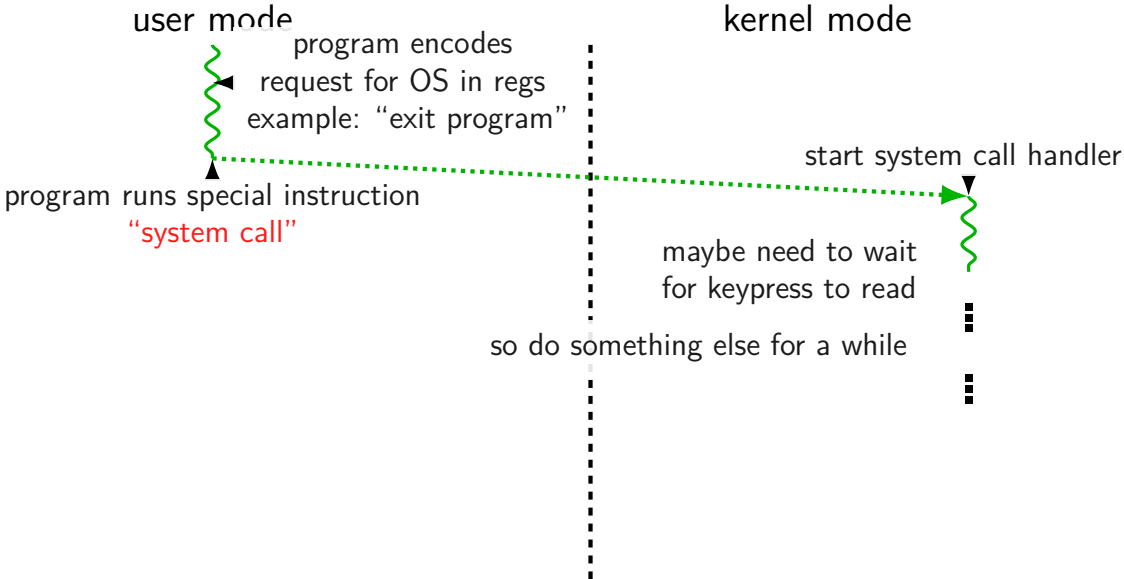
system call handled slowly?



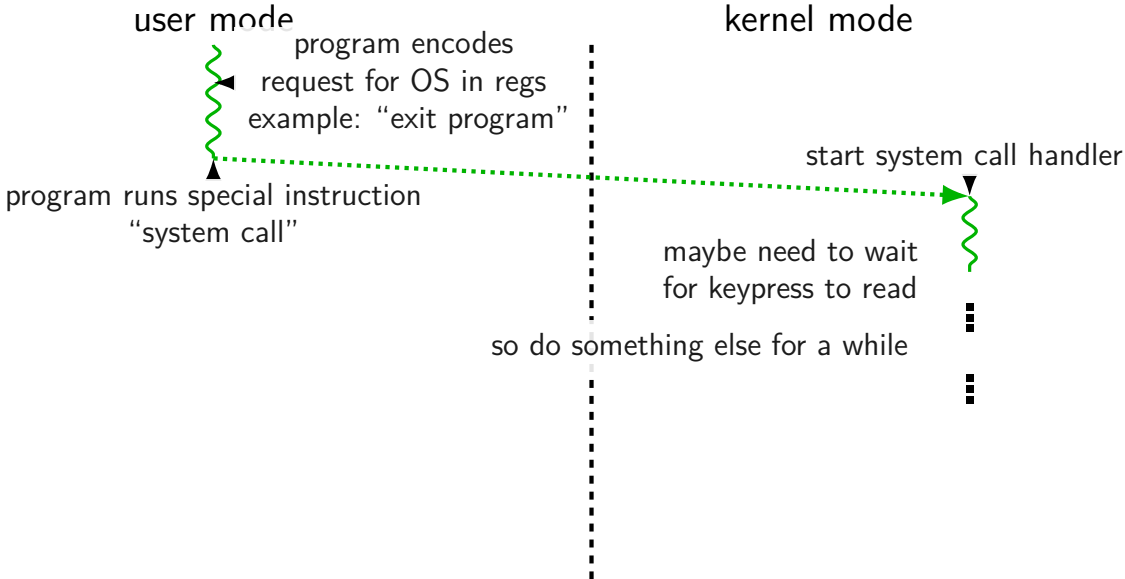
system call handled slowly?



system call handled slowly?



system call handled slowly?



system call wrappers

library functions to not write assembly:

```
open:
    movq $2, %rax // 2 = sys_open
    // 2 arguments happen to use same registers
    syscall
    // return value in %eax
    cmp $0, %rax
    jl has_error
    ret
has_error:
    neg %rax
    movq %rax, errno
    movq $-1, %rax
    ret
```

system call wrappers

library functions to not write assembly:

```
open:
    movq $2, %rax // 2 = sys_open
    // 2 arguments happen to use same registers
    syscall
    // return value in %eax
    cmp $0, %rax
    jl has_error
    ret
has_error:
    neg %rax
    movq %rax, errno
    movq $-1, %rax
    ret
```

system call wrapper: usage

```
/* unistd.h contains definitions of:
    O_RDONLY (integer constant), open() */
#include <unistd.h>
int main(void) {
  int file_descriptor;
  file_descriptor = open("input.txt", O_RDONLY);
  if (file_descriptor < 0) {
      printf("error: %s\n", strerror(errno));
      exit(1);
  }
  ...
  result = read(file_descriptor, ...);
  ...
}
```

system call wrapper: usage

```
/* unistd.h contains definitions of:
    O_RDONLY (integer constant), open() */
#include <unistd.h>
int main(void) {
  int file_descriptor;
  file_descriptor = open("input.txt", O_RDONLY);
  if (file_descriptor < 0) {
      printf("error: %s\n", strerror(errno));
      exit(1);
  }
  ...
  result = read(file_descriptor, ...);
  ...
}
```

strace hello_world (1)

`strace` — Linux tool to trace system calls

run on assembly program we saw earlier:

```
$ strace -o trace.txt ./hello_world
$ cat trace.txt
execve("./hello_world", ["./hello_world"],
        0x7ffeedafd0a0 /* 28 vars */) = 0
write(1, "Hello, World!\n\0", 14)       = 14
exit(0)                                 = ?
+++ exited with 0 +++
```

strace hello_world (2)

```
#include <stdio.h>
int main() { puts("Hello, World!"); }
```

when statically linked:

```
execve("./hello_world", ["./hello_world"], 0x7ffeb4127f70 /* 28 vars */)
                                        = 0
brk(NULL)                               = 0x22f8000
brk(0x22f91c0)                          = 0x22f91c0
arch_prctl(ARCH_SET_FS, 0x22f8880)      = 0
uname({sysname="Linux", nodename="reiss-t3620", ...}) = 0
readlink("/proc/self/exe", "/u/cr4bd/spring2023/cs3130/slide"..., 4096)
                                        = 57
brk(0x231a1c0)                          = 0x231a1c0
brk(0x231b000)                          = 0x231b000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
                                                     directory)
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
write(1, "Hello, World!\n", 14)         = 14
exit_group(0)                           = ?
```

aside: what are those syscalls?

execve: run program

brk: allocate heap space

arch_prctl(ARCH_SET_FS, ...): thread local storage pointer

may make more sense when we cover concurrency/parallelism later

uname: get system information

readlink of /proc/self/exe: get name of this program

access: can we access this file [in this case, a config file]?

fstat: get information about open file

exit_group: variant of exit

strace hello_world (2)

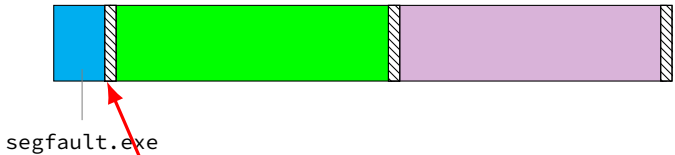
```
#include <stdio.h>
int main() { puts("Hello, World!"); }
```

when dynamically linked:

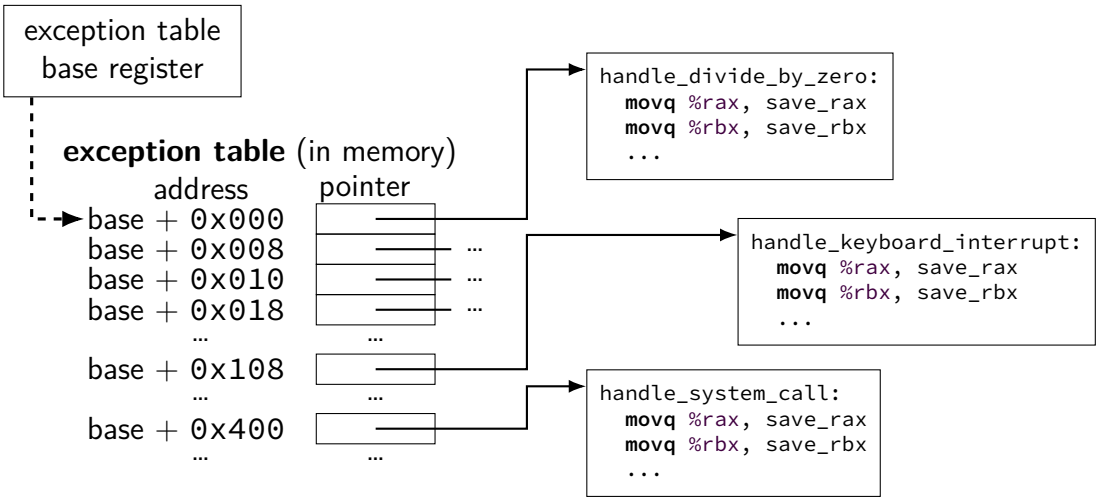
```
execve("./hello_world", ["./hello_world"], 0x7ffcfe91d540 /* 28 vars */)
                                        = 0
brk(NULL)                               = 0x55d6c351b000
...
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=196684, ...}) = 0
mmap(NULL, 196684, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f7a62dd3000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0"..., 832) = 832
...
close(3)                                = 0
write(1, "Hello, World!\n", 14)          = 14
exit_group(0)                           = ?
+++ exited with 0 +++
```

backup slides

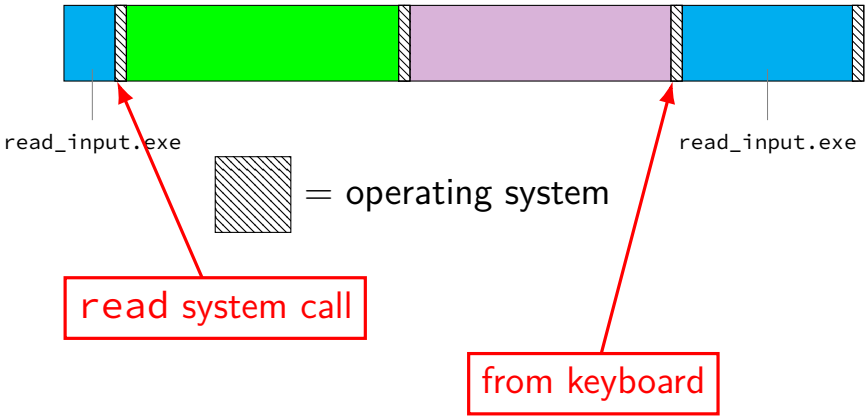
crash timeline timeline



locating exception handlers (one strategy)



keyboard input timeline



exceptions in exceptions

```
handle_timer_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */

  movq %r14, save_r14
  ...
```

exceptions in exceptions

```
handle_timer_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */
  movq %r14, save_r14
  ...
```

```
handle_keyboard_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  movq %r14, save_r14
  movq %r13, save_r13
  ...
```

exceptions in exceptions

```
handle_timer_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */
  movq %r14, save_r14
  ...
```

```
handle_keyboard_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  movq %r14, save_r14
  movq %r13, save_r13
  ...
```

oops, overwrote saved values?

interrupt disabling

CPU supports disabling (most) interrupts

interrupts will wait until it is reenabled

CPU has extra state:

- are interrupts enabled?
- is keyboard interrupt pending?
- is timer interrupt pending?

exceptions in exceptions

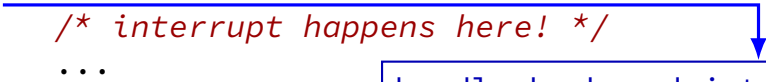
```
handle_timer_interrupt:
  /* interrupts automatically disabled here */
  movq %rsp, save_rsp
  save_old_pc save_pc
  /* key press here */
  jmpIfFromKernelMode skip_exception_stack
  movq current_exception_stack, %rsp
skip_set_kernel_stack:
  pushq save_rsp
  pushq save_pc
  enable_intterupts2
  pushq %r15
  ...

  /* interrupt happens here! */
  ...
```

exceptions in exceptions

```
handle_timer_interrupt:
  /* interrupts automatically disabled here */
  movq %rsp, save_rsp
  save_old_pc save_pc
  /* key press here */
  jmpIfFromKernelMode skip_exception_stack
  movq current_exception_stack, %rsp
skip_set_kernel_stack:
  pushq save_rsp
  pushq save_pc
  enable_intterupts2
  pushq %r15
  ...
  /* interrupt happens here! */
  ...
```

exceptions in exceptions

```
handle_timer_interrupt:
  /* interrupts automatically disabled here */
  movq %rsp, save_rsp
  save_old_pc save_pc
  /* key press here */
  jmpIfFromKernelMode skip_exception_stack
  movq current_exception_stack, %rsp
skip_set_kernel_stack:
  pushq save_rsp
  pushq save_pc
  enable_intterupts2
  pushq %r15
  ...
  /* interrupt happens here! */
  ...
```



disabling interrupts

automatically disabled when exception handler starts

also can be done with privileged instruction:

```
change_keyboard_parameters:
  disable_interrupts
  ...
  /* change things used by
     handle_keyboard_interrupt here */
  ...
  enable_interrupts
```

exception implementation

detect condition (program error or external event)

save current value of PC somewhere

jump to exception handler (part of OS)

- jump done without program instruction to do so

exception implementation: notes

I describe a simplified version

real x86/x86-64 is a bit more complicated
(mostly for historical reasons)

context

all registers values

`%rax` `%rbx`, ..., `%rsp`, ...

condition codes

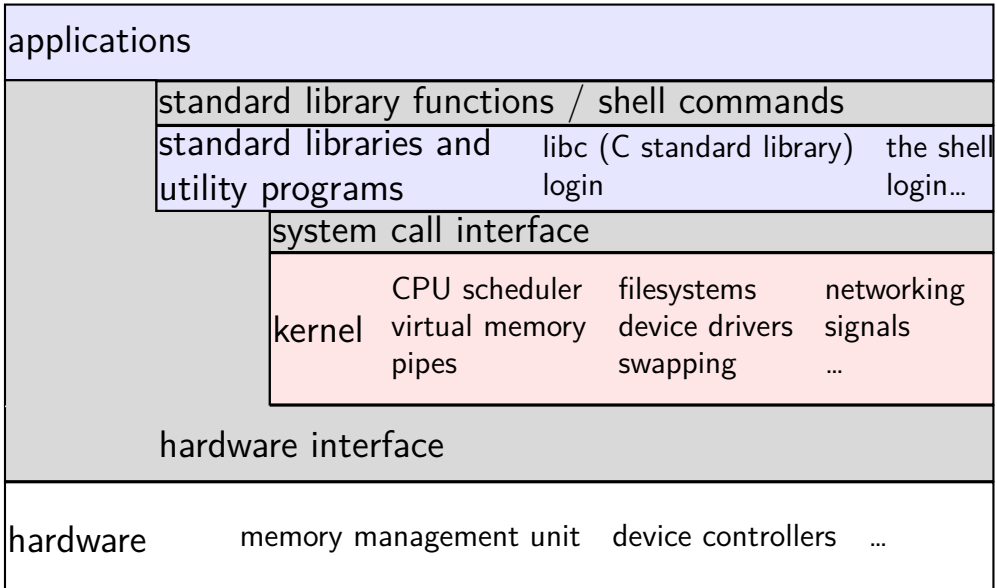
program counter

address space (map from program to real addresses)

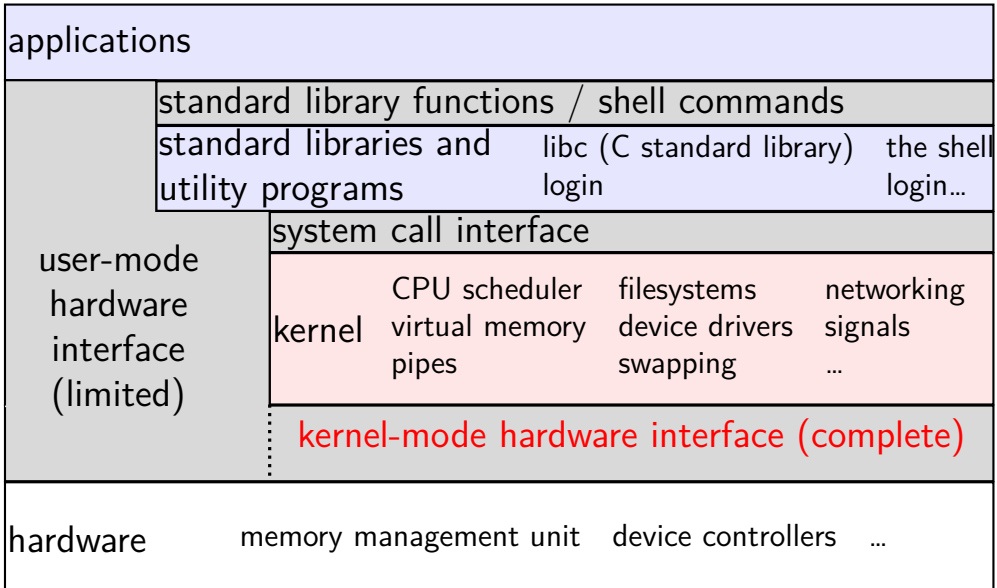
context switch pseudocode

```
context_switch(last, next):
  copy_preexception_pc last->pc
  mov rax,last->rax
  mov rcx, last->rcx
  mov rdx, last->rdx
  ...
  mov next->rdx, rdx
  mov next->rcx, rcx
  mov next->rax, rax
  jmp next->pc
```

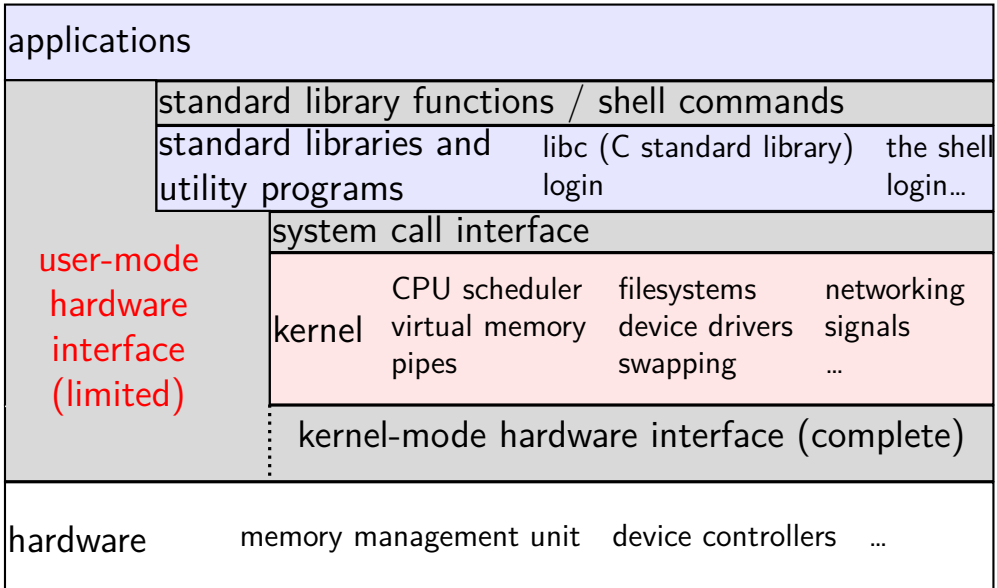
the classic Unix design



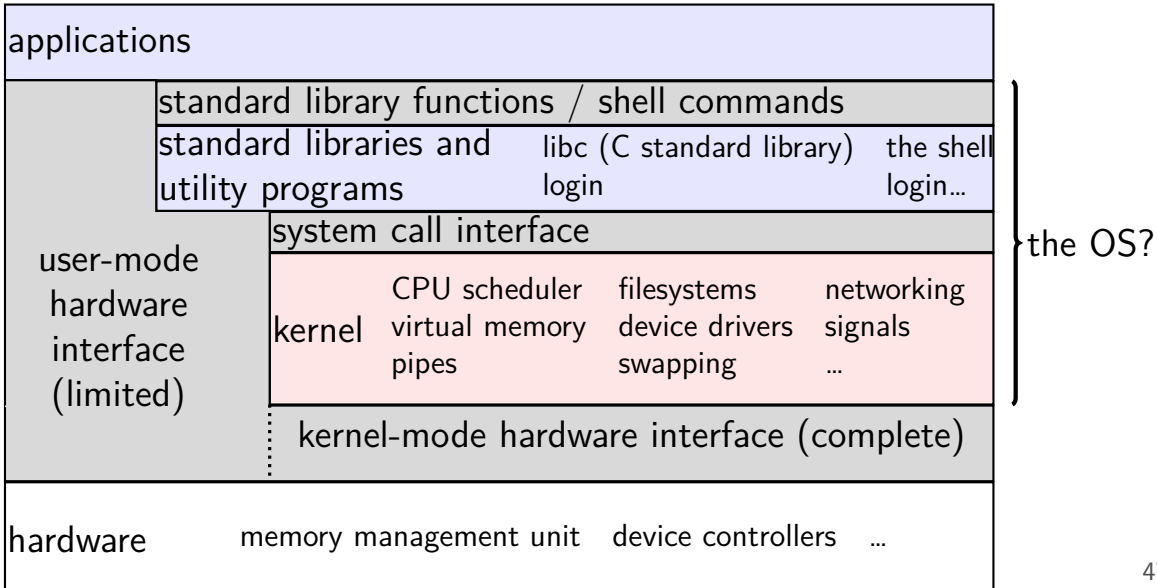
the classic Unix design



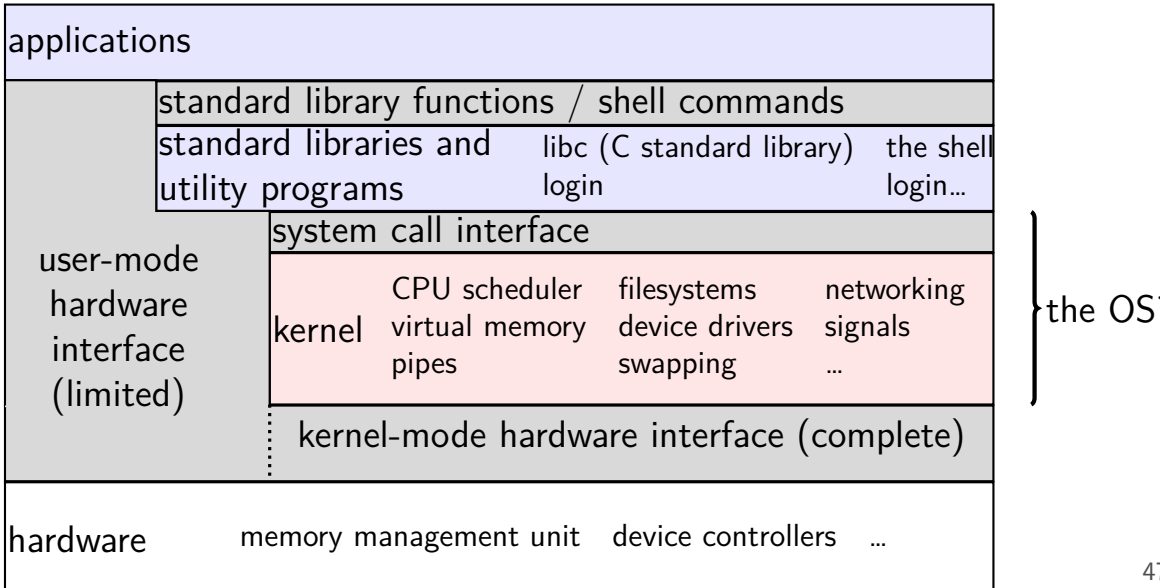
the classic Unix design



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...

exception implementation

detect condition (program error or external event)

save current value of PC somewhere

jump to exception handler (part of OS)

jump done without program instruction to do so

exception implementation: notes

I describe a simplified version

real x86/x86-64 is a bit more complicated
(mostly for historical reasons)

running the exception handler

hardware saves the old program counter (and maybe more)

identifies location of exception handler via table

then jumps to that location

OS code can save anything else it wants to , etc.