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
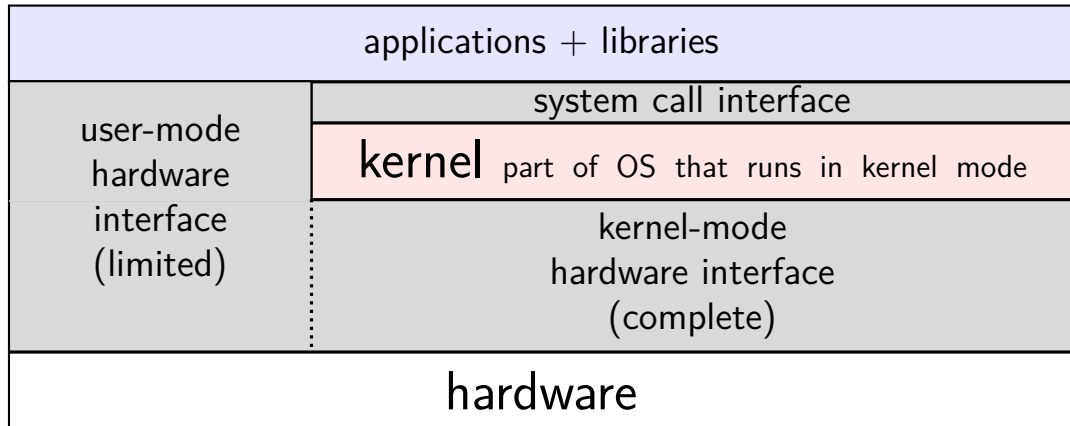
what runs in kernel mode?

system boots in kernel mode

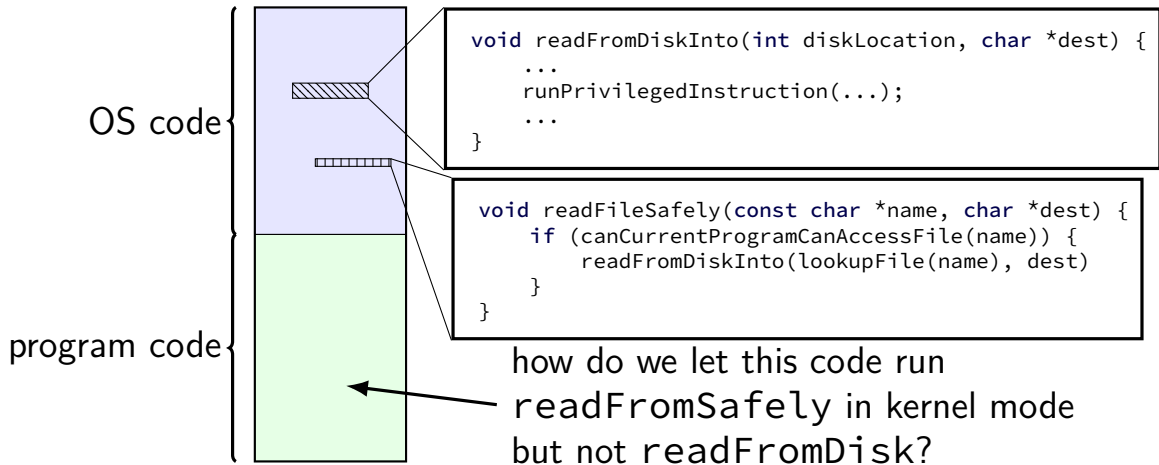OS switches to user mode to run program code

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

applications + libraries		
user-mode hardware interface (limited)	system call interface	
	kernel part of OS that runs in kernel mode	
	kernel-mode hardware interface (complete)	
hardware		

calling the OS?


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

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

OS code

program code

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:

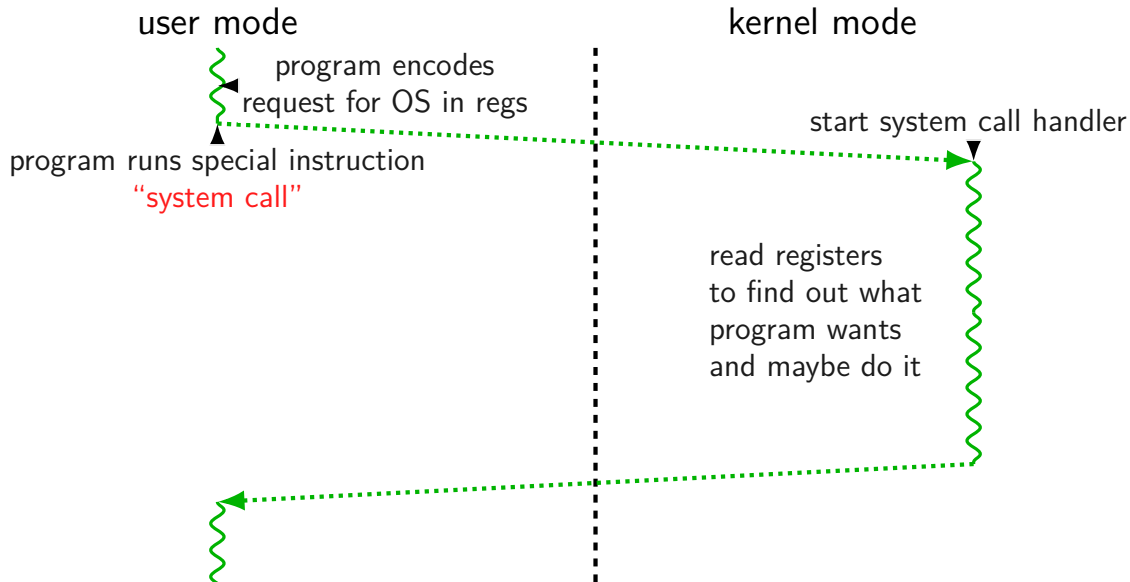figure out what operation the program wants
> calling convention, similar to function arguments + return value

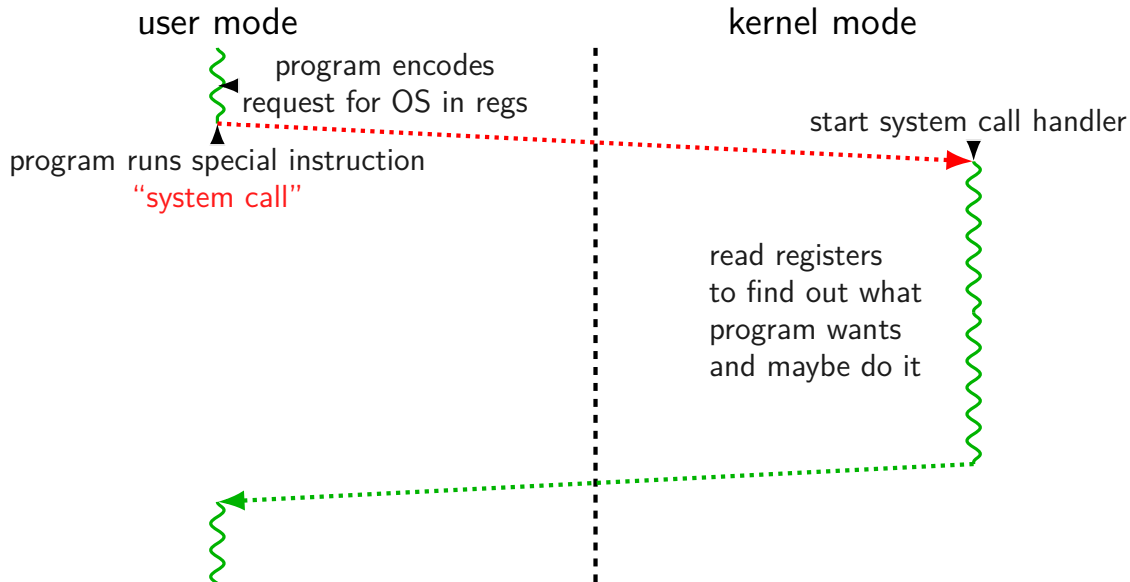be "safe" — not allow the program to do 'bad' things
> example: checks whether current program is allowed to read file before reading it
> requires exceptional care — program can try weird things

system call process

user mode   kernel mode

program encodes
request for OS in regs

start system call handler

program runs special instruction
"system call"

read registers
to find out what
program wants
and maybe do it

12

system call process

user mode   kernel mode

program encodes
request for OS in regs

                                              start system call handler

program runs special instruction
"system call"

                              read registers
                              to find out what
                              program wants
                              and maybe do it

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?

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
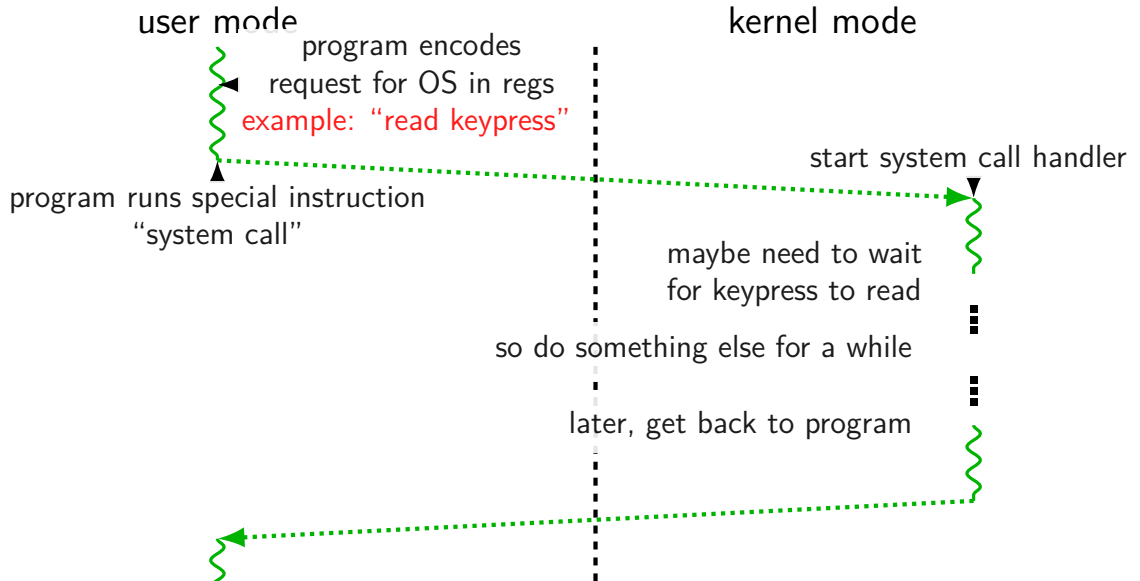
execve — run a program in the current process

_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

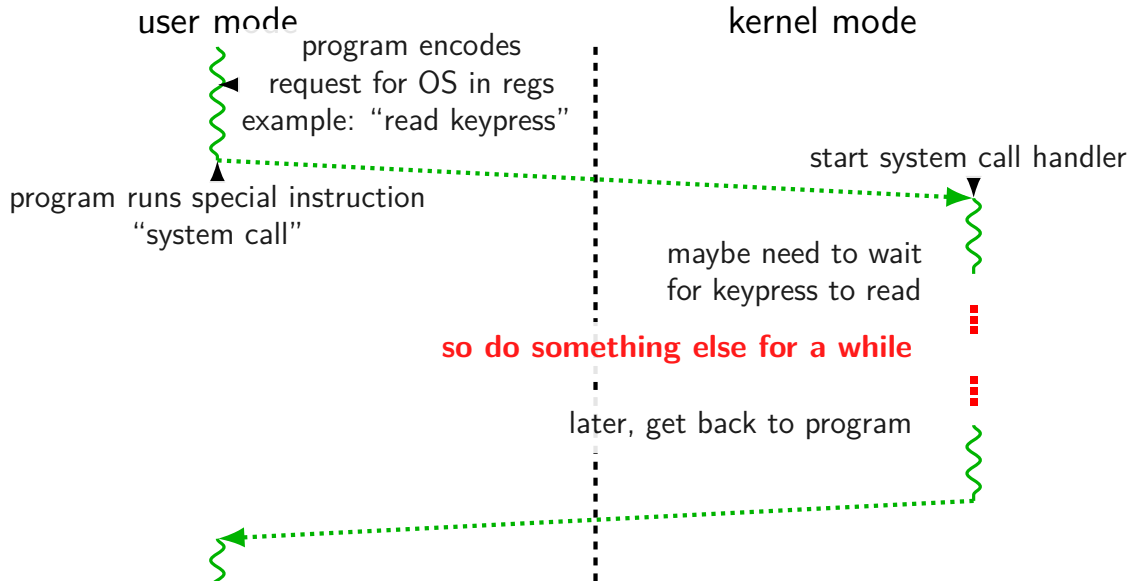execve — run a program in the current process

_exit — terminate a process

open, read, write — access files
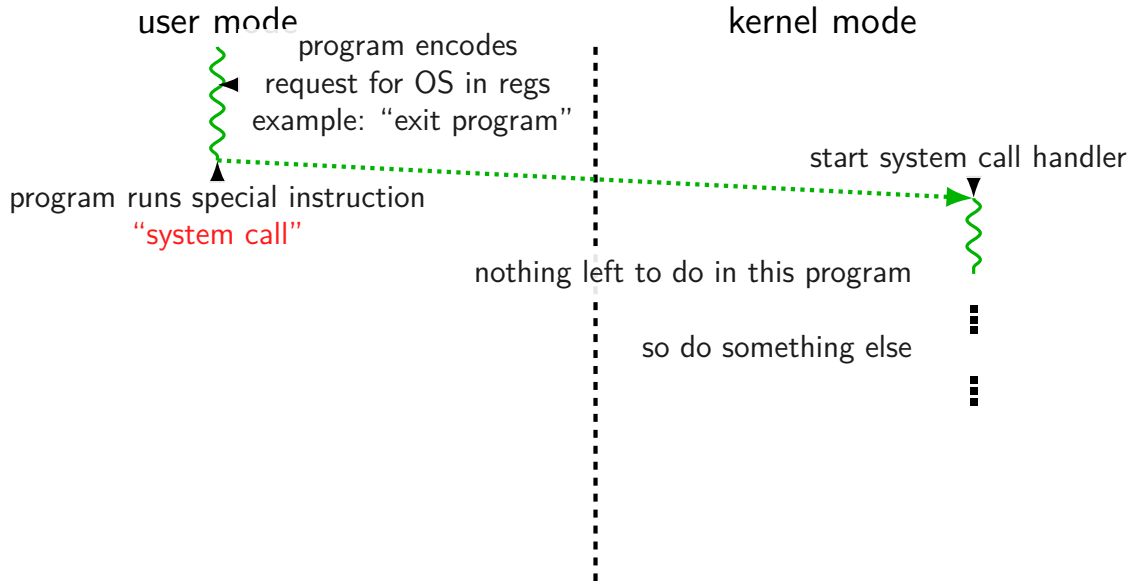
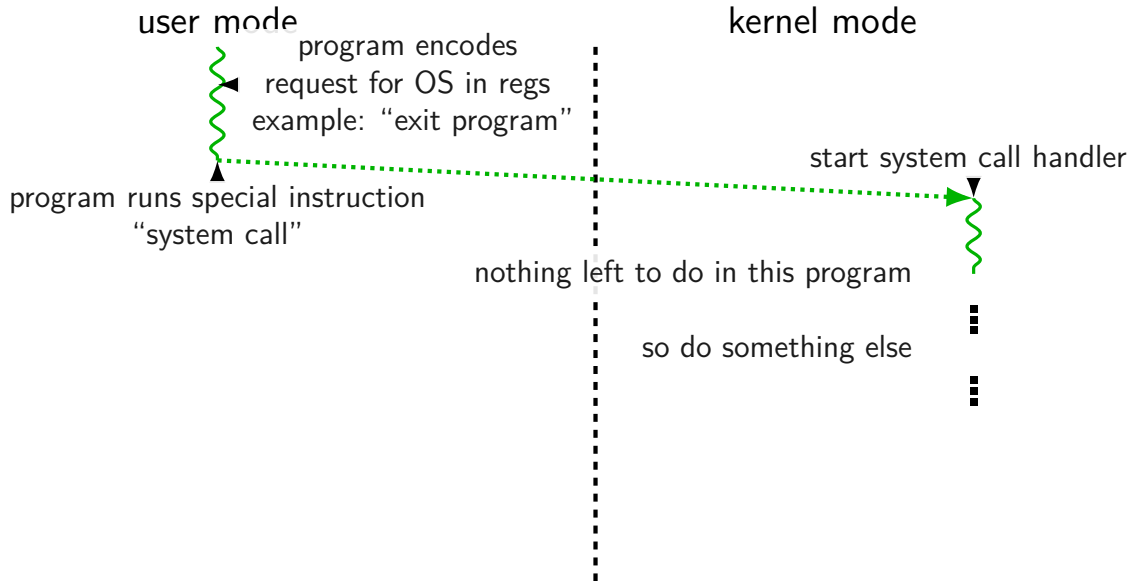socket, accept, getpeername — socket-related

system call handled slowly?

user mode

program encodes
request for OS in regs
example: "read keypress"

program runs special instruction
"system call"

kernel mode

start system call handler

maybe need to wait
for keypress to read

so do something else for a while

later, get back to program

system call handled slowly?

user mode

program encodes
request for OS in regs
example: "read keypress"

program runs special instruction
"system call"

kernel mode

start system call handler

maybe need to wait
for keypress to read

so do something else for a while

later, get back to program

system call handled slowly?

user mode   kernel mode

program encodes
request for OS in regs
example: "exit program"

                                              start system call handler

program runs special instruction
"system call"

            nothing left to do in this program

                    so do something else

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

```c
/* 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

```c
/* 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)                             = ?
+++ exited with 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
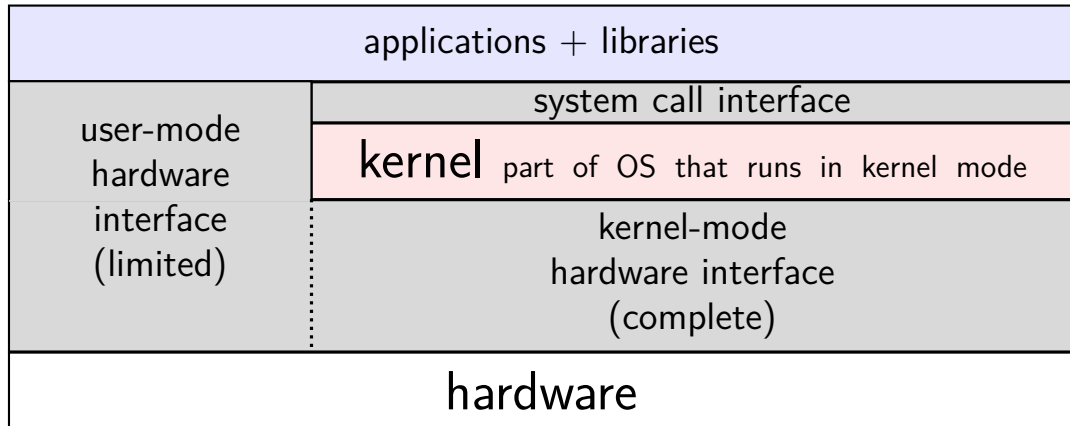
exit_group: variant of exit

strace hello_world (2)

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

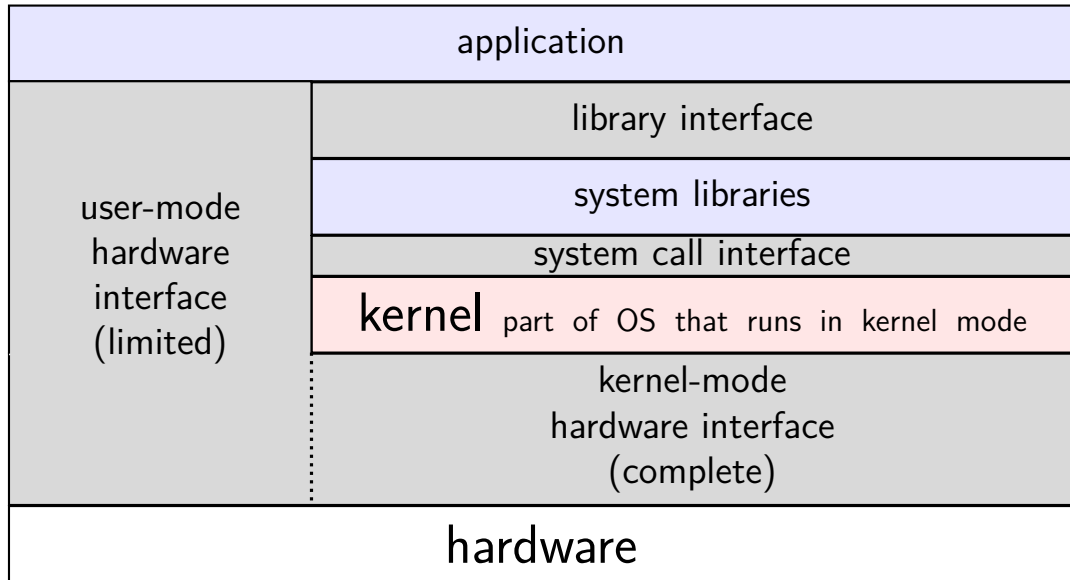when dynamically linked:

```
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 +++
```

hardware + system call interface

applications + libraries		
user-mode hardware interface (limited)	system call interface	
	kernel part of OS that runs in kernel mode	
	kernel-mode hardware interface (complete)	
hardware		

hardware + system call + library interface

application		
user-mode hardware interface (limited)	library interface	
	system libraries	
	system call interface	
	kernel part of OS that runs in kernel mode	
	kernel-mode hardware interface (complete)	
hardware		

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

memory protection

modifying another program's memory?

Program A	Program B
| ```
0x10000: .long 42
    // ...
    // do work
    // ...
    movq 0x10000, %rax
``` | ```
// while A is working:
movq $99, %rax
movq %rax, 0x10000
...
``` |

memory protection

modifying another program's memory?

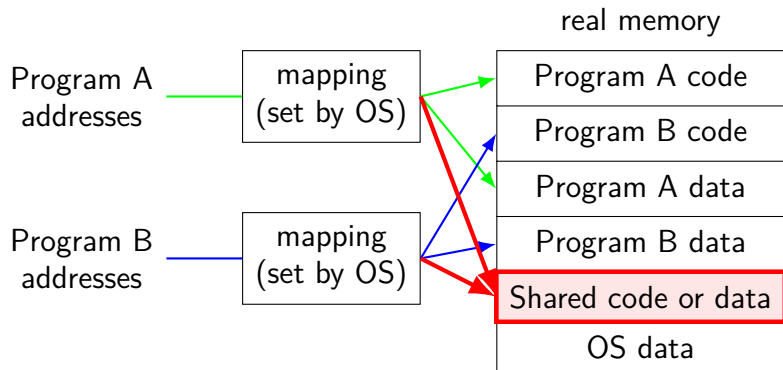| Program A	Program B

```0x10000: .long 42```
    ```// ...```
    ```// do work```
    ```// ...```
    ```movq 0x10000, %rax```
	```// while A is working:```
	```movq $99, %rax```
	```movq %rax, 0x10000```
	```...```

result: %rax (in A) is …	

A. 42   B. 99   C. 0x10000

D. 42 or 99 (depending on timing/program layout/etc)

E. 42 or 99 or program might crash (depending on …)

F. something else

memory protection

modifying another program's memory?

Program A	Program B

`0x10000: .long 42`
    `// ...`
    `// do work`
    `// ...`
    `movq 0x10000, %rax`
	`// while A is working:`
	`movq $99, %rax`
	`movq %rax, 0x10000`
	`...`

result: %rax (in A) is 42
(always with 'normal' multiuser OSes)

A. 42   B. 99   C. 0x10000

D. 42 or 99 (depending on timing/program layout/etc)

E. 42 or 99 or program might crash (depending on …)

F. something else

shared memory

recall: dynamically linked libraries

would be nice not to duplicate code/data...

we can!

real memory

| Program A addresses | mapping (set by OS) | Program A code |
| Program B code |
| Program A data |
| Program B addresses | mapping (set by OS) | Program B data |
| Shared code or data |
| OS data |

one way to set shared memory on Linux

```c
/* regular file, OR: */
int fd = open("/tmp/somefile.dat", O_RDWR);
/* special in-memory file */
int fd = shm_open("/name", O_RDWR);
...
/* make file's data accessible as memory */
void *memory = mmap(NULL, size, PROT_READ | PROT_WRITE,
                    MAP_SHARED, fd, 0);
```

mmap: "map" a file's data into your memory

will discuss a bit more when we talk about virtual memory

part of how Linux loads dynamically linked libraries

memory protection

modifying another program's memory?

Program A	Program B

```0x10000: .long 42```
```   // ...```
```   // do work```
```   // ...```
```   movq 0x10000, %rax```
	```// while A is working:```
	```movq $99, %rax```
	```movq %rax, 0x10000```
	```...```

result: %rax (in A) is 42 (always with 'normal' multiuser OSes)	result: might crash

A. 42   B. 99   C. 0x10000

D. 42 or 99 (depending on timing/program layout/etc)

E. 42 or 99 or program might crash (depending on ...)

F. something else

program crashing?

what happens on processor when program crashes?

other program informed of crash to display message

use processor to run some other program

program crashing?

what happens on processor when program crashes?

other program informed of crash to display message

use processor to run some other program

how does hardware do this?

would be complicated to tell about other programs, etc.

instead: hardware runs designated OS routine

exceptions

recall: system calls — software asks OS for help

also cases where hardware asks OS for help

different triggers than system calls

but same mechanism as system calls:
  switch to kernel mode (if not already)
  call OS-designated function

exceptions

recall: system calls — software asks OS for help

also cases where hardware asks OS for help

different triggers than system calls

but same mechanism as system calls:
    switch to kernel mode (if not already)
    call OS-designated function

types of exceptions

system calls
    intentional — ask OS to do something

errors/events in programs
    memory not in address space ("Segmentation fault")
    privileged instruction
    divide by zero, invalid instruction

    …
(and more we'll talk about later)

types of exceptions

system calls
     intentional — ask OS to do something

errors/events in programs
     memory not in address space ("Segmentation fault")
     privileged instruction
     divide by zero, invalid instruction

     …
(and more we'll talk about later)

types of exceptions

system calls
> intentional — ask OS to do something

errors/events in programs
> memory not in address space ("Segmentation fault")
> privileged instruction
> divide by zero, invalid instruction
>
> …

(and more we'll talk about later)

types of exceptions

system calls
   intentional — ask OS to do something

errors/events in programs
   memory not in address space ("Segmentation fault")
   privileged instruction
   divide by zero, invalid instruction

   …
(and more we'll talk about later)

synchronous
triggered by
current program

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

types of exceptions

system calls
　intentional — ask OS to do something

errors/events in programs
　memory not in address space ("Segmentation fault")
　privileged instruction
　divide by zero, invalid instruction
　…

synchronous
triggered by
current program

external — I/O, etc.
　timer — configured by OS to run OS at certain time
　I/O devices — key presses, hard drives, networks, …
　hardware is broken (e.g. memory parity error)

asynchronous
not triggered by
running program

exceptions [Venn diagram]

system calls

interrupts
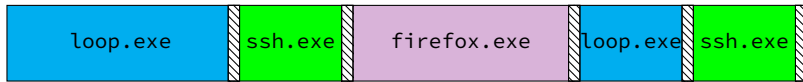(example: I/O)

faults
(example: segfault)
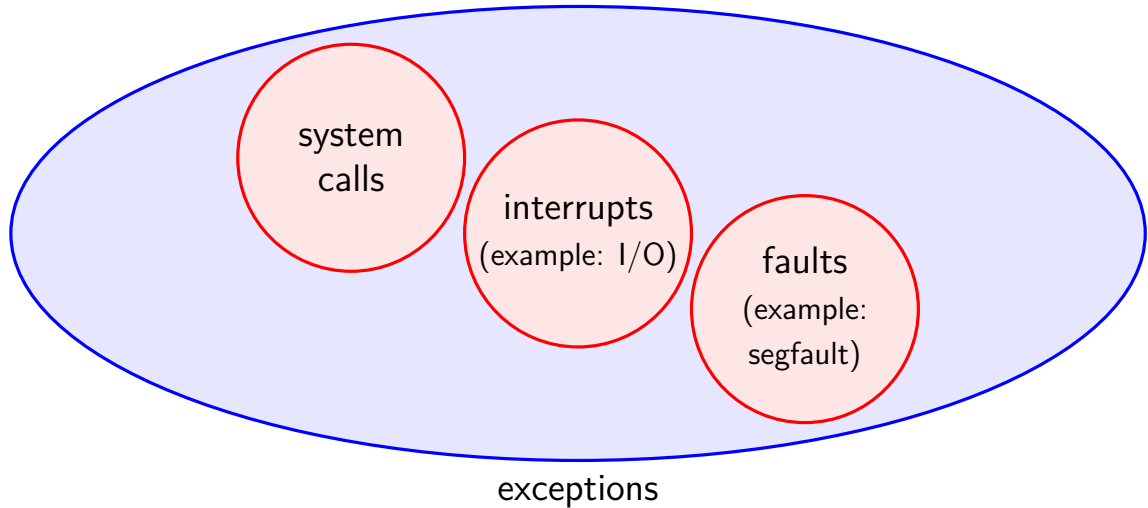
exceptions

general exception process

user mode

kernel mode

something triggers exception
maybe the program did
or maybe something else

start exception handler

OS handles
whatever happened

go back to running
program code
possibly a different
program than before

exit exception handler
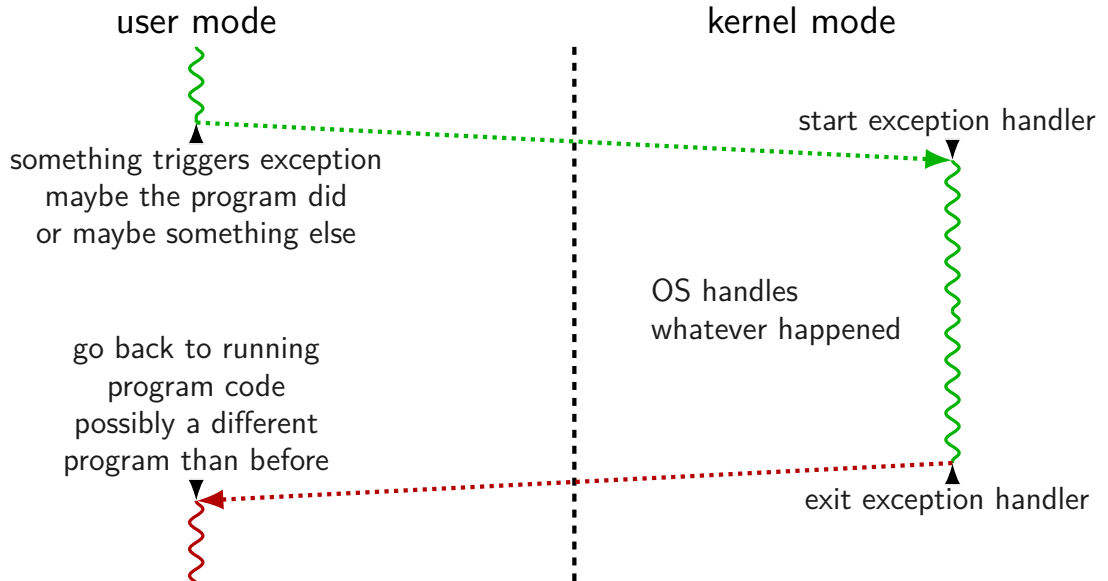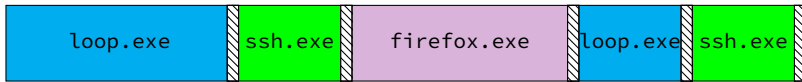
time multiplexing

$\boxed{\text{▨}}$ = operating system

time multiplexing

| loop.exe | ssh.exe | firefox.exe | loop.exe | ssh.exe |

= operating system

exception happens

return from exception

41

switching programs

OS starts running somehow
   some sort of exception

saves old registers + program counter + address mapping
   (optimization: could omit when program crashing/exiting)

sets new registers + address mapping, jumps to new program counter

called context switch
   saved information called context

contexts (A running)

in Memory

in CPU

%rax
%rbx
%rcx
%rsp
...
SF
ZF
PC

Process A memory:
code, stack, etc.

Process B memory:
code, stack, etc.

OS memory:
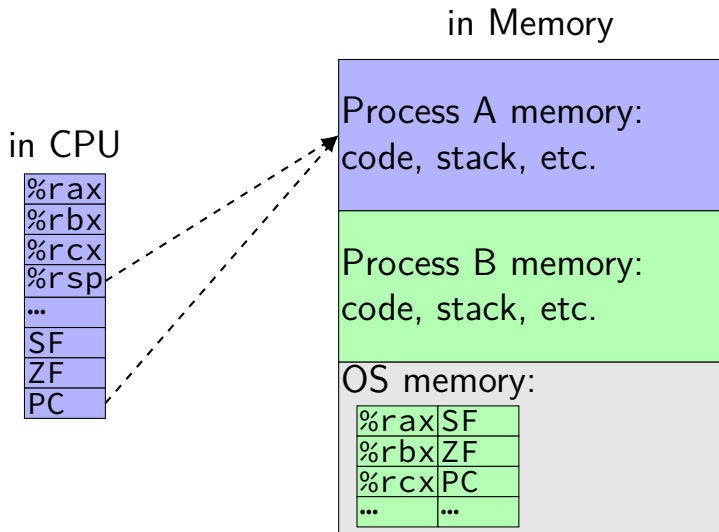
%rax	SF
%rbx	ZF
%rcx	PC
...	...

contexts (B running)

in Memory

in CPU

| %rax |
| %rbx |
| %rcx |
| %rsp |
| ... |
| SF |
| ZF |
| PC |

Process A memory:
code, stack, etc.

Process B memory:
code, stack, etc.

OS memory:

%rax	SF
%rbx	ZF
%rcx	PC
...	...

threads

thread = illusion of own processor

own register values

own program counter value

threads

thread = illusion of own processor

own register values

own program counter value

actual implementation:
many threads sharing one processor
    problem: where are register/program counter values
    when thread not active on processor?

types of exceptions

system calls
    intentional — ask OS to do something

errors/events in programs
    memory not in address space ("Segmentation fault")
    privileged instruction
    divide by zero, invalid instruction

    …

} synchronous
triggered by
current program

external — I/O, etc.
    timer — configured by OS to run OS at certain time
    I/O devices — key presses, hard drives, networks, …
    hardware is broken (e.g. memory parity error)

} asynchronous
not triggered by
running program

exception patterns with I/O (1)

input — available now:
>   exception: device says "I have input now"
>   handler: OS stores input for later
>   exception (syscall): program says "I want to read input"
>   handler: OS returns that input

input — not available now:
>   exception (syscall): program says "I want to read input"
>   handler: OS runs other things (context switch)
>   exception: device says "I have input now"
>   handler: OS retrieves input
>   handler: (possibly) OS switches back to program that wanted it

exception patterns with I/O (2)

output — ready now:
> exception (syscall): program says "I want to output this'
> handler: OS sends output to deive

output — not ready now
> exception (syscall): program says "I want to output"
> handler: OS realizes device can't accept output yet
> (other things happen)
> exception: device says "I'm ready for output now"
> handler: OS sends output requested earlier

keyboard input timeline

read_input.exe

read_input.exe

= operating system

read system call

from keyboard
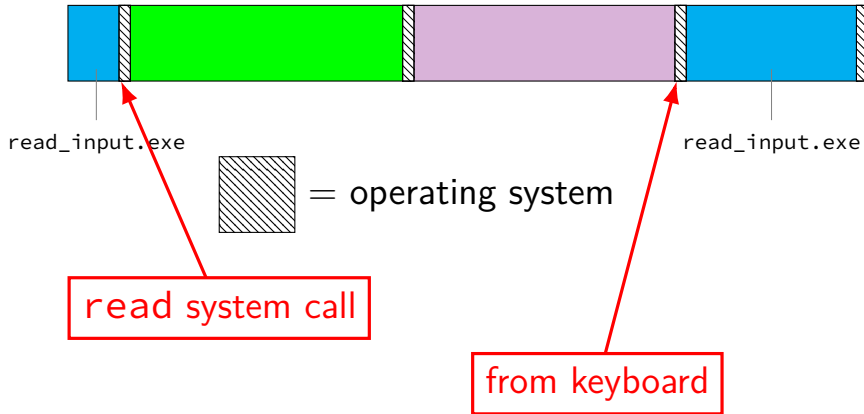
review: definitions

exception: hardware calls OS specified routine
    many possible reasons
    system calls: type of exception

context switch: OS switches to another thread
    by saving old register values + loading new ones
    part of OS routine run by exception

which of these require exceptions? context switches?

A. program calls a function in the standard library

B. program writes a file to disk

C. program A goes to sleep, letting program B run

D. program exits

E. program returns from one function to another function

F. program pops a value from the stack

which require exceptions [answers] (1)

A. program calls a function in the standard library

no (same as other functions in program; many standard library functions make no system calls (and do not otherwise trigger exceptions — for example strlen, pow; also if we consider the calling of a function just the call instruction, then the library functions that do make system calls won't do so until later)

B. program writes a file to disk

yes (requires kernel mode only operations)

C. program A goes to sleep, letting program B run

yes (kernel mode usually required to change the address space to acess program B's memory)

which require exceptions [answer] (2)

D. program exits
   yes (requires switching to another program, which requires accessing OS data + other program's memory)

E. program returns from one function to another function
   no

F. program pops a value from the stack
   no

which require context switches [answer]

no: A. program calls a function in the standard library

no: B. program writes a file to disk
(but might be done if program needs to wait for disk and other things could be run while it does)

yes: C. program A goes to sleep, letting program B run

yes: D. program exits

no: E. program returns from one function to another function

no: F. program pops a value from the stack

terms for exceptions

terms for exceptions aren't standardized

our readings use one set of terms
    interrupts = externally-triggered
    faults = error/event in program
    trap = intentionally triggered

all these terms appear differently elsewhere

The Process

process = thread(s) + address space

illusion of dedicated machine:
     thread = illusion of own CPU
     (process could have multiple threads — with independent registers)
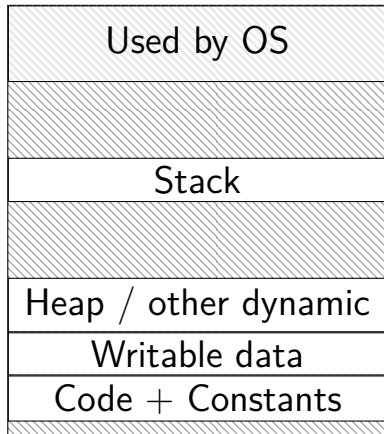   address space = illusion of own memory

backup slides

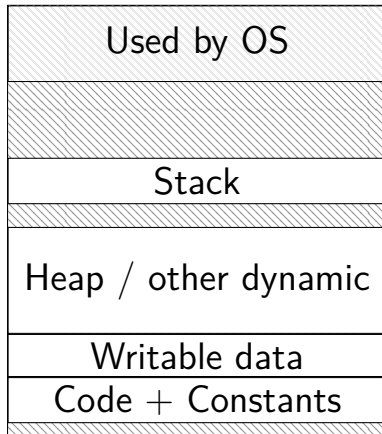# keeping permissions?

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
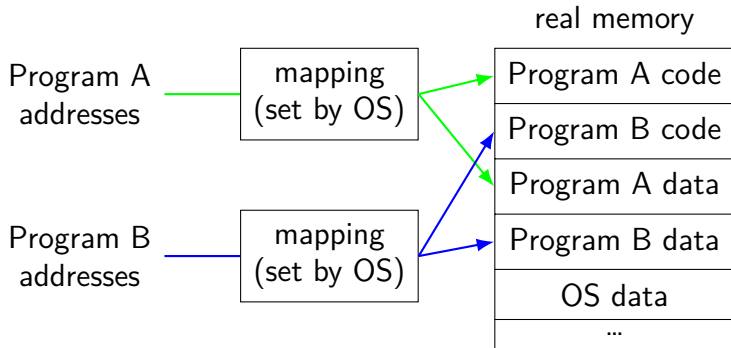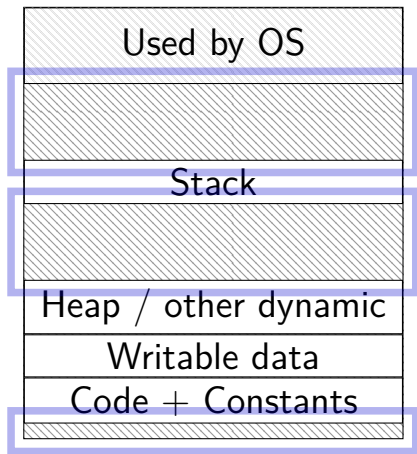
program memory (two programs)

Program A

Used by OS
Stack
Heap / other dynamic
Writable data
Code + Constants

Program B

Used by OS
Stack
Heap / other dynamic
Writable data
Code + Constants

address space

programs have illusion of own memory

called a program's address space

program memory (two programs)

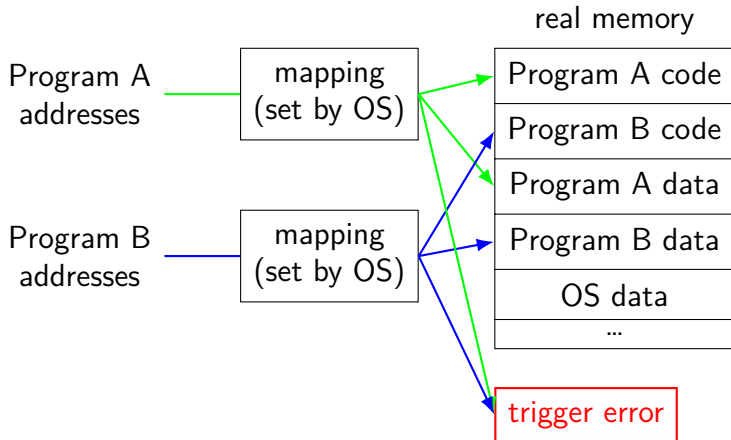| Program A	Program B
Used by OS	Used by OS
Stack	Stack
Heap / other dynamic	Heap / other dynamic
Writable data	Writable data
Code + Constants	Code + Constants
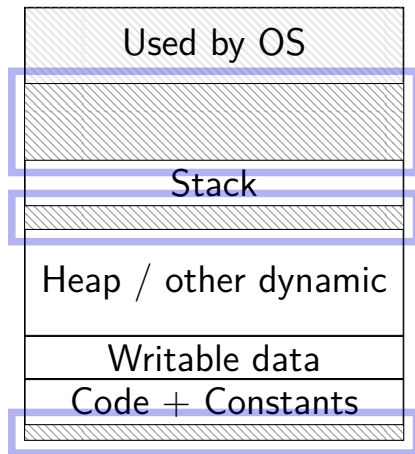
address space

programs have illusion of own memory

called a program's address space

address space mechanisms

topic after exceptions

called virtual memory

mapping called page tables

mapping part of what is changed in context switch

an infinite loop

```c
int main(void) {
    while (1) {
        /* waste CPU time */
    }
}
```
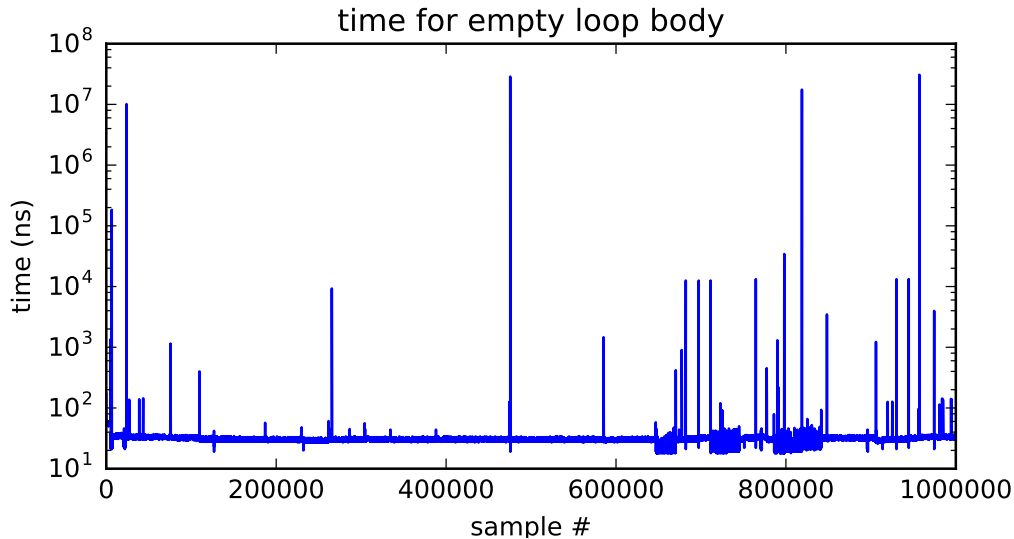
If I run this on a shared department machine, can you still use it?
…if the machine only has one core?

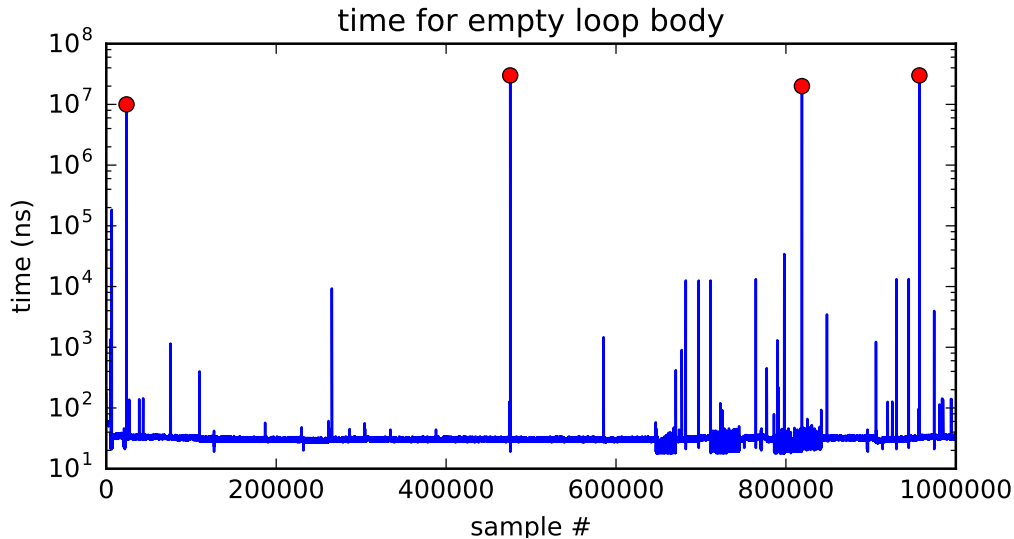# timing nothing

```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

doing nothing on a busy system

time for empty loop body

doing nothing on a busy system

time for empty loop body

time multiplexing

processor:

| loop.exe | | loop.exe |

time ⟶

time multiplexing

processor:

time ⟶

```
    ...
loop: ...
    ...
    jmp loop
loop: ...
    ...
million cycle delay

    ...
    jmp loop
loop: ...
```
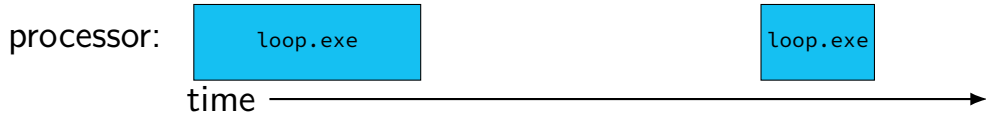
time multiplexing


```
         ...
    loop: ...
         ...
         jmp loop
    loop: ...
         ...
million cycle delay

         ...
         jmp loop
    loop: ...
```
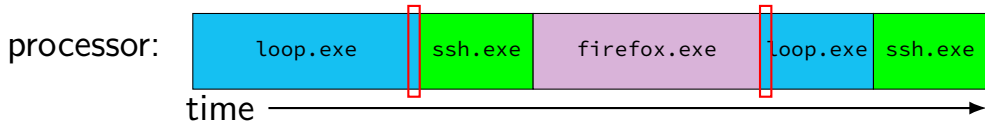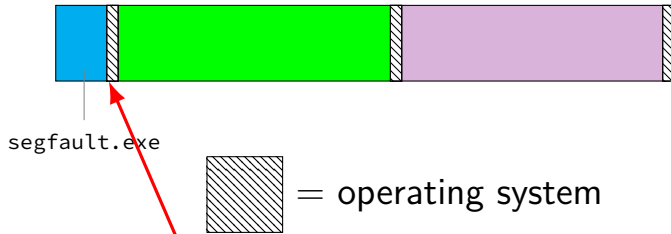
crash timeline timeline

segfault.exe

= operating system

out of bounds memory acecss

locating exception handlers (one strategy)
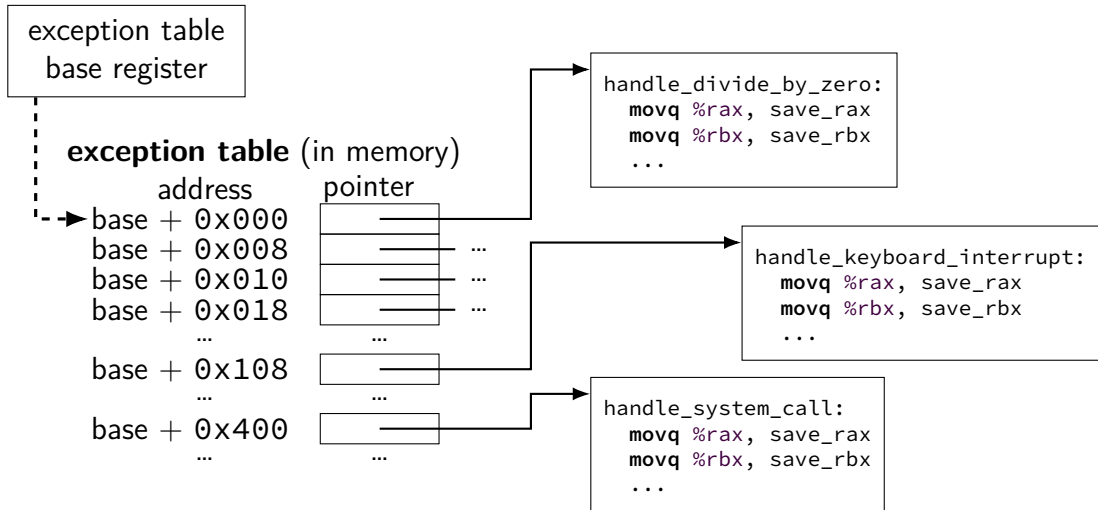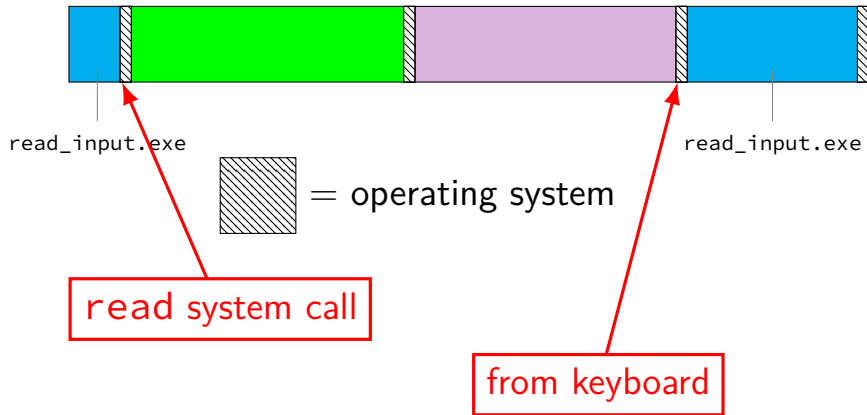
exception table base register

exception table (in memory)

address	pointer
base + 0x000	
base + 0x008	…
base + 0x010	…
base + 0x018	…
…	…
base + 0x108	
…	…
base + 0x400	
…	…

```
handle_divide_by_zero:
  movq %rax, save_rax
  movq %rbx, save_rbx
  ...
```

```
handle_keyboard_interrupt:
  movq %rax, save_rax
  movq %rbx, save_rbx
  ...
```

```
handle_system_call:
  movq %rax, save_rax
  movq %rbx, save_rbx
  ...
```

keyboard input timeline

read_input.exe

read_input.exe

= operating system

read system call

from keyboard

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
  ...
```
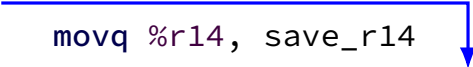
exceptions in exceptions

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

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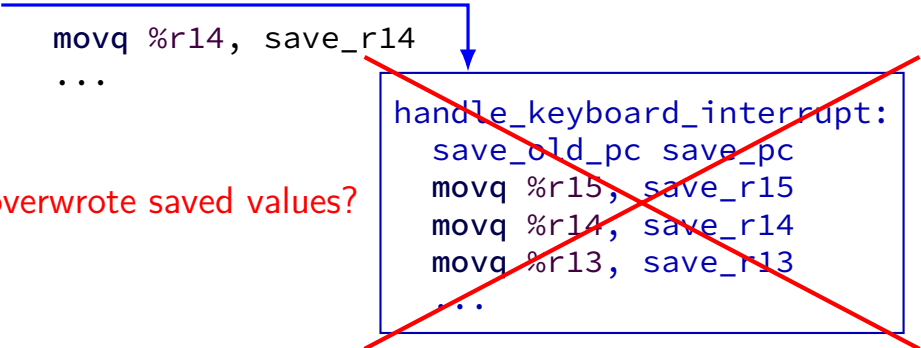
```
handle_keyboard_interrupt:
  save_old_pc save_pc
  movq %r15, save_r15
  movq %r14, save_r14
  movq %r13, save_r13
  ...
```

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

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 ...		

80

the classic Unix design

applications			
standard libraries and utility programs	standard library functions / shell commands		
	libc (C standard library) login		the shell login...
user-mode hardware interface (limited)	system call interface		
	kernel	CPU scheduler virtual memory pipes	filesystems device drivers swapping	networking signals ...
	kernel-mode hardware interface (complete)		
hardware	memory management unit device controllers ...		

the classic Unix design

applications			
	standard library functions / shell commands		
user-mode hardware interface (limited)	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 ...
	kernel-mode hardware interface (complete)		
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			
user-mode hardware interface (limited)	kernel	CPU scheduler virtual memory pipes	filesystems device drivers swapping	networking signals ...
	kernel-mode hardware interface (complete)			
hardware	memory management unit device controllers ...			

the OS?

80

the classic Unix design

applications				
	standard library functions / shell commands			
	standard libraries and utility programs	libc (C standard library) login		the shell login...
user-mode hardware interface (limited)	system call interface			
	kernel	CPU scheduler virtual memory pipes	filesystems device drivers swapping	networking signals ...
	kernel-mode hardware interface (complete)			
hardware	memory management unit device controllers ...			

the OS?

80

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.