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?

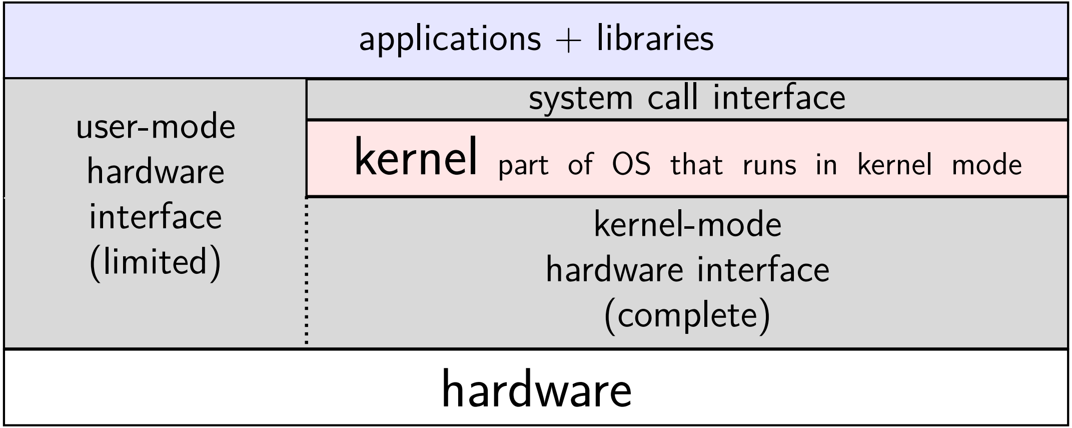
system boots in kernel mode

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)

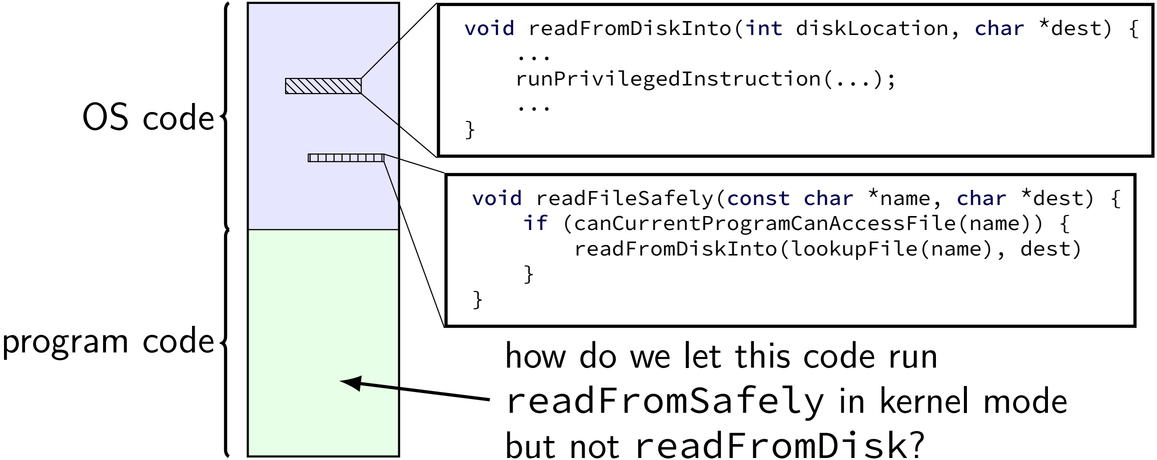
system call interface

kernel part of OS that runs in kernel mode

kernel-mode hardware interface (complete)

hardware

calling the OS?



OS code

program code

```
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: “system call”

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

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

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)                           = ?
+++ 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

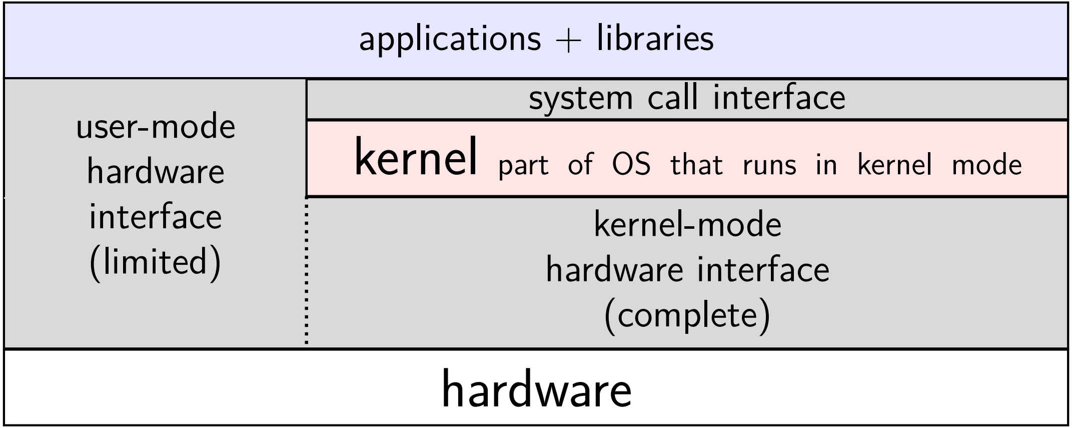
strace hello_world (2)

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

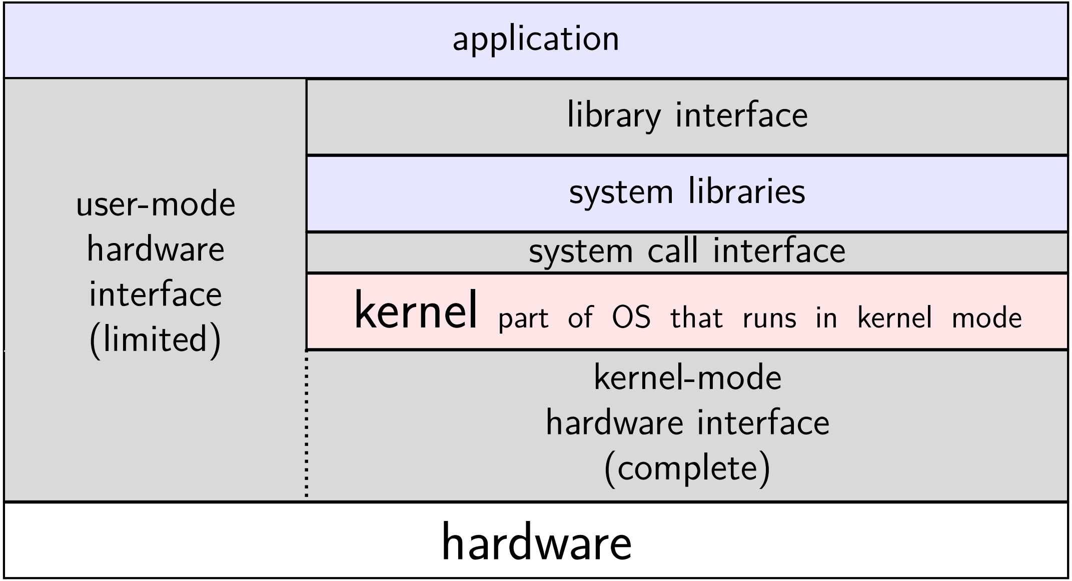
when dynamically linked:

```
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
library interface
system libraries
user-mode hardware interface (limited)
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

```
0x10000: .long 42
     // ...
     // do work
     // ...
     movq 0x10000, %rax
```

Program B

```
// 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 ...

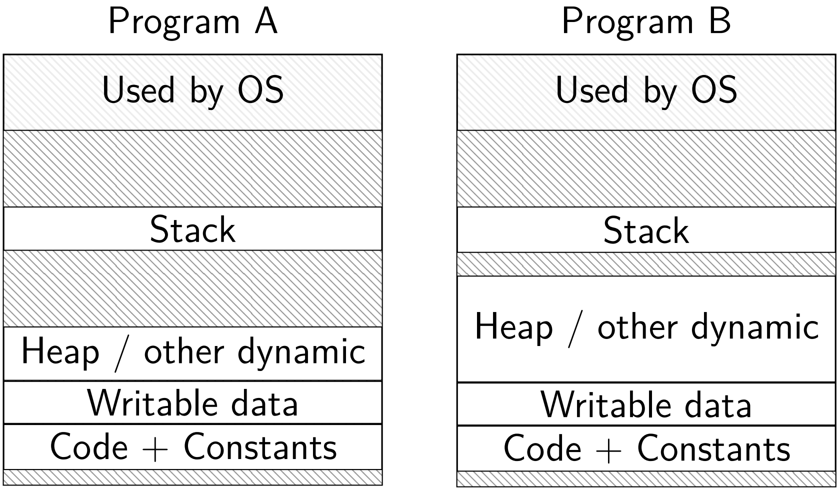
A. 42 B. 99 C. 0x10000

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

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

F. something else

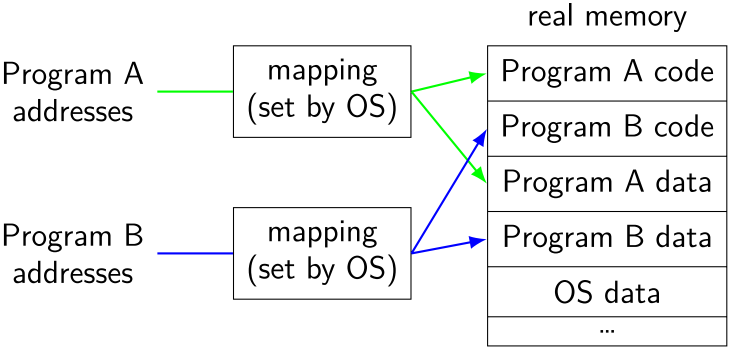
program memory (two programs)



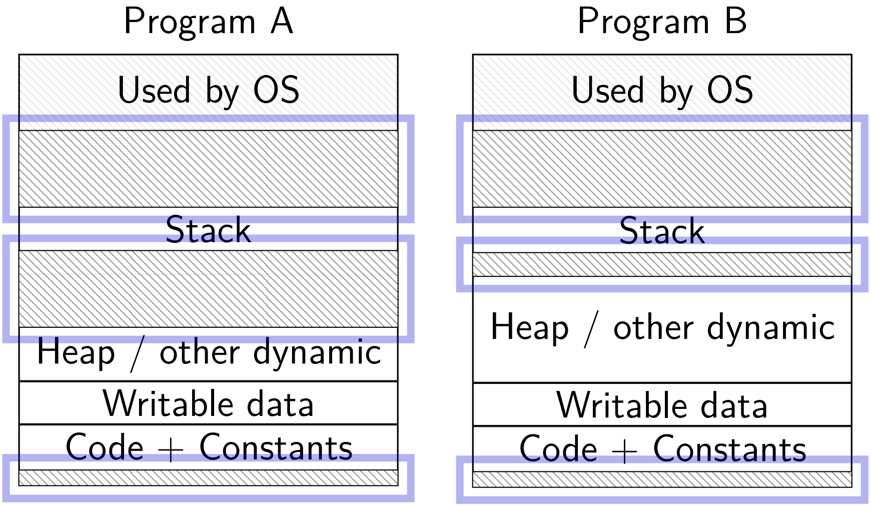
address space

programs have illusion of own memory

called a program's address space



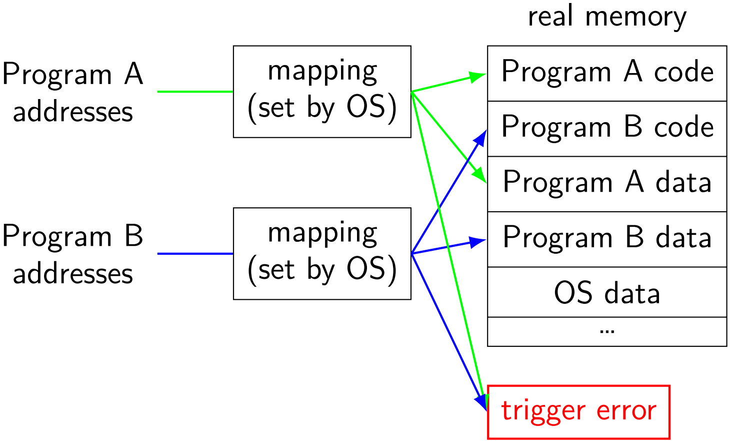
program memory (two programs)



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

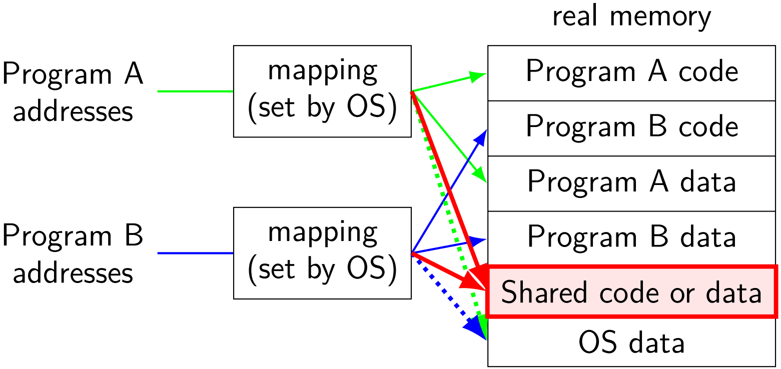
mapping part of what is changed in context switch

shared memory

recall: dynamically linked libraries

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

we can!



one way to set shared memory on Linux

```
/* 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

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

an infinite loop

```
int main(void) {
    while (1) {
        /* waste CPU time */
    }
}
```

If I run this on a shared department machine, can you still use it?

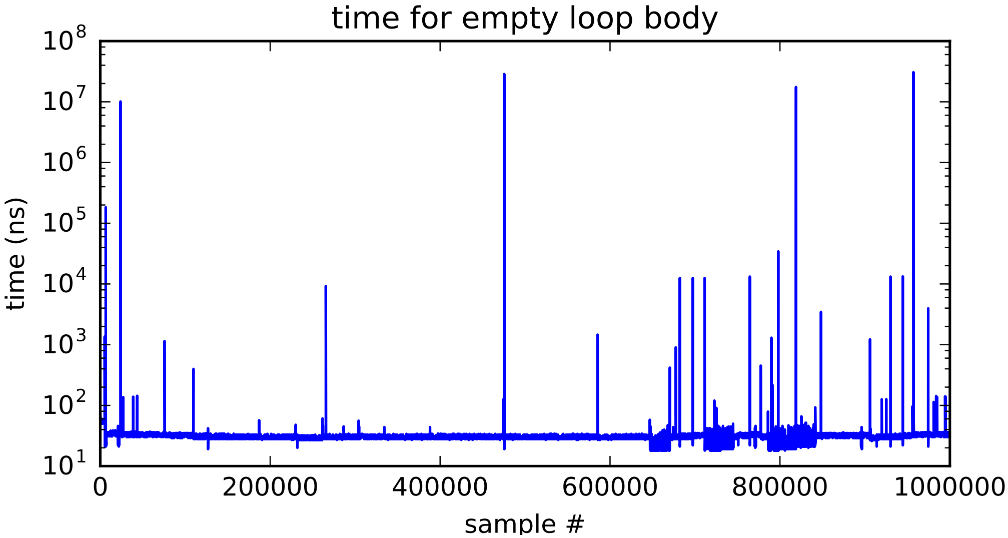
...if the machine only has one core?

timing nothing

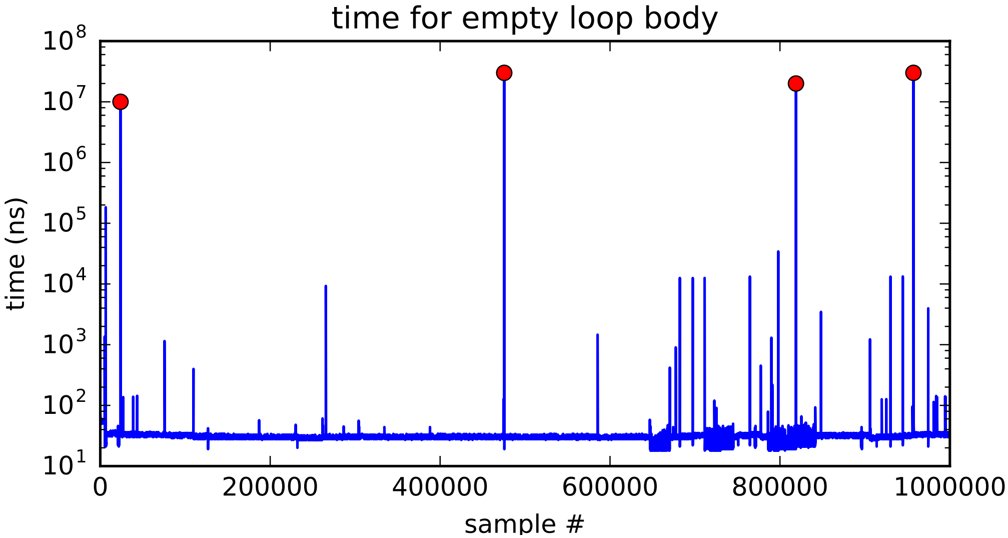
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

doing nothing on a busy system



doing nothing on a busy system



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

external — I/O, etc.

- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, …
- hardware is broken (e.g. memory parity error)

asynchronous
not triggered by running program

system calls

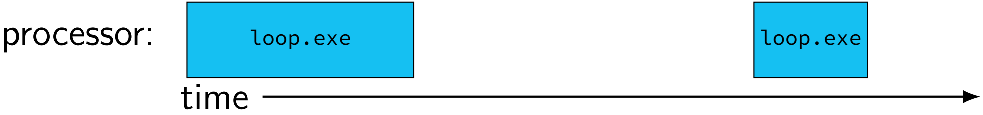
- intentional — ask OS to do something

errors/events in programs

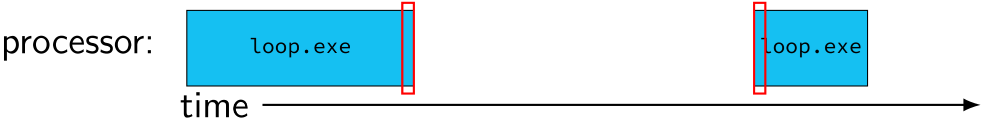
- memory not in address space (“Segmentation fault”)
- privileged instruction
- divide by zero
- invalid instruction

synchronous
triggered by current program

time multiplexing



time multiplexing

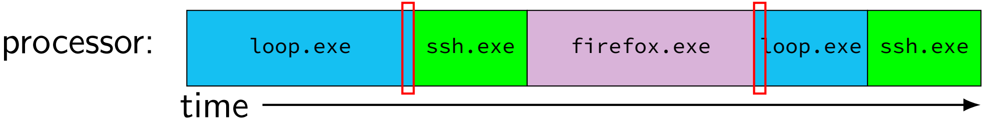


```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

million cycle delay

```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing


processor:

loop.exe

ssh.exe

firefox.exe

loop.exe

ssh.exe

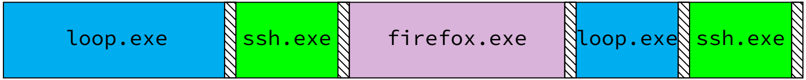
time

```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

million cycle delay

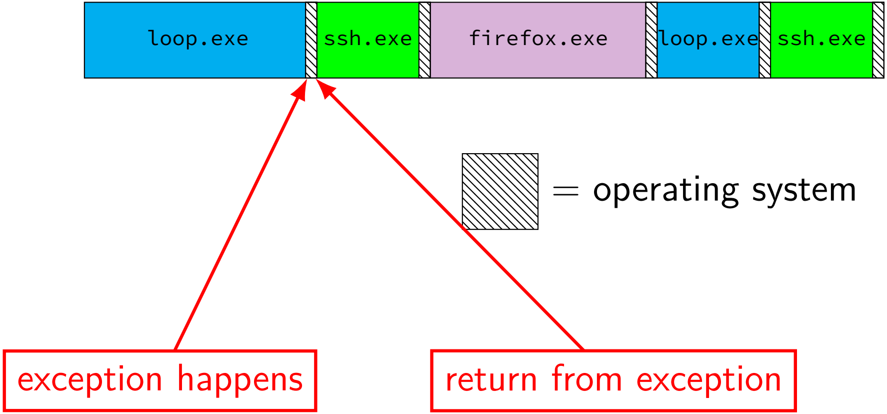
```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing really



▨ = operating system

time multiplexing really



▨ = operating system

exception happens

return from exception

threads

thread = illusion of own processor

own register values

own program counter value

threads

thread = illusion of own processor

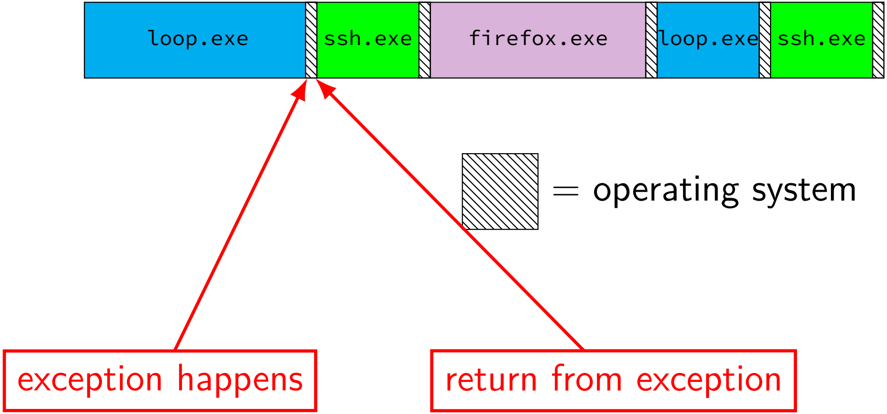
own register values

own program counter value

actual implementation:
many threads sharing one processor

- problem: where are register/program counter values when thread not active on processor?

time multiplexing really



▨ = operating system

exception happens

return from exception

OS and time multiplexing

starts running instead of normal program

- mechanism for this: exceptions (later)

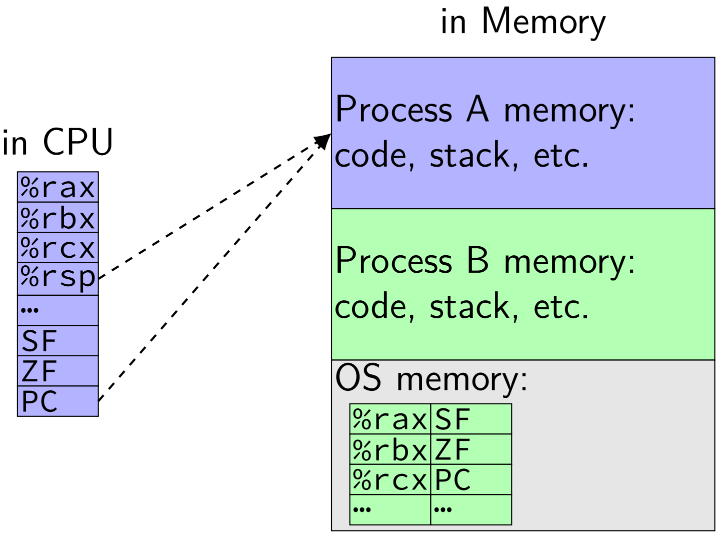
saves old program counter, registers somewhere

sets new registers, jumps to new program counter

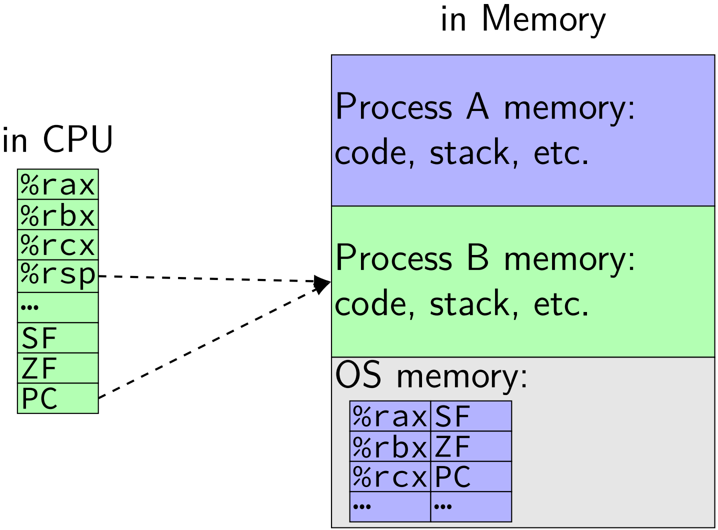
called context switch

- saved information called context

contexts (A running)



contexts (B running)



in Memory

Process A memory:
code, stack, etc.

Process B memory:
code, stack, etc.

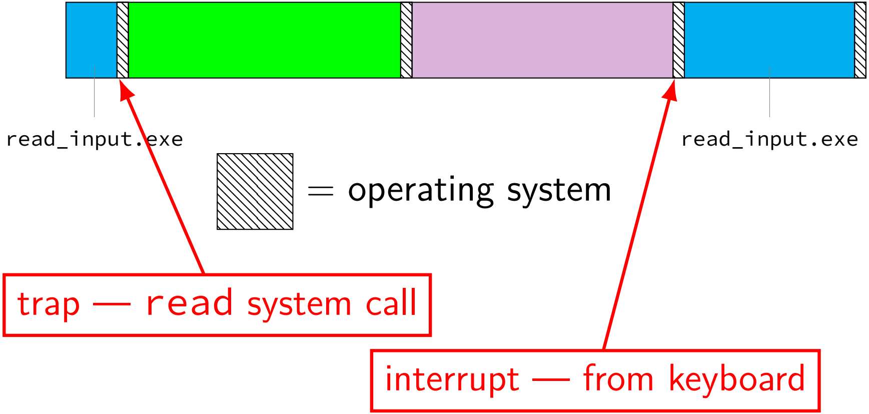
OS memory:

%rax	SF
%rbx	ZF
%rcx	PC
…	…

in CPU

%rax
%rbx
%rcx
%rsp
…
SF
ZF
PC

keyboard input timeline



trap — read system call

interrupt — from keyboard

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

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

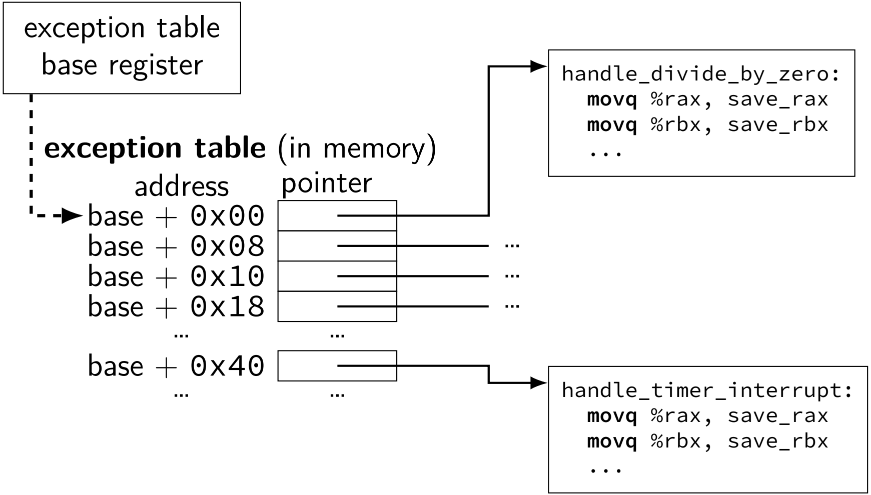
exception implementation: notes

I describe a simplified version

real x86/x86-64 is a bit more complicated

(mostly for historical reasons)

locating exception handlers



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

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

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.

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

The Process

process = thread(s) + address space

illusion of dedicated machine:

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