Processes and Exceptions

an infinite loop

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

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

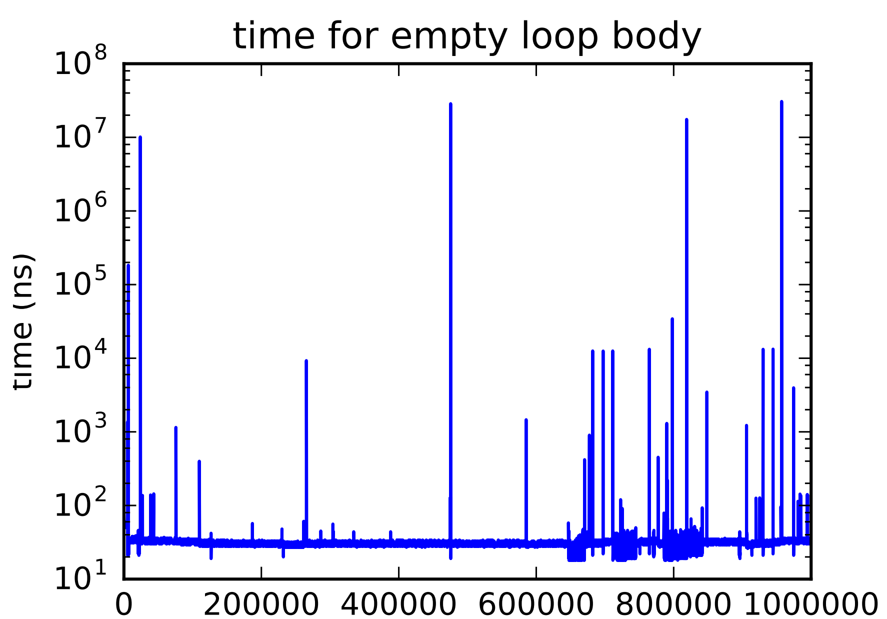
...even if the machine only has one core?

timing nothing

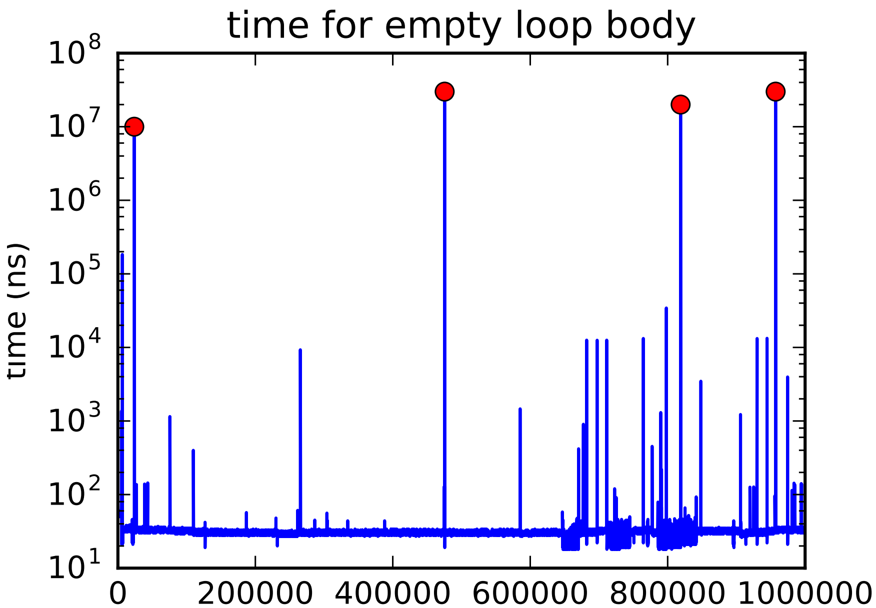
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

doing nothing on a busy system



doing nothing on a busy system



time multiplexing

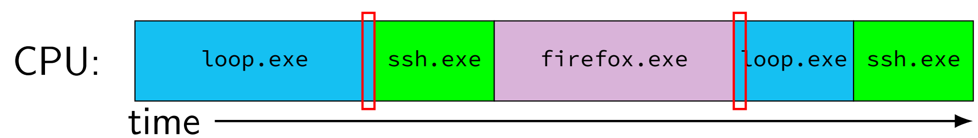
time multiplexing


```
...
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




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

million cycle delay

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

illusion: dedicated processor

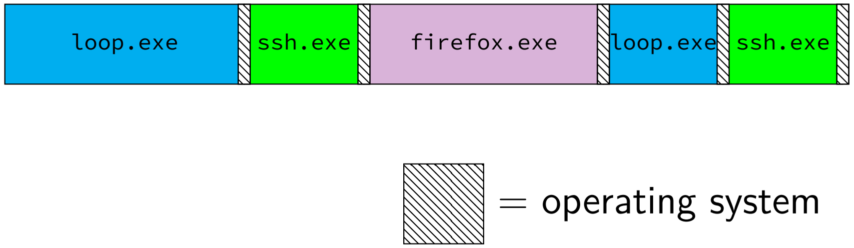
time multiplexing: illusion of dedicated processor

including dedicated registers

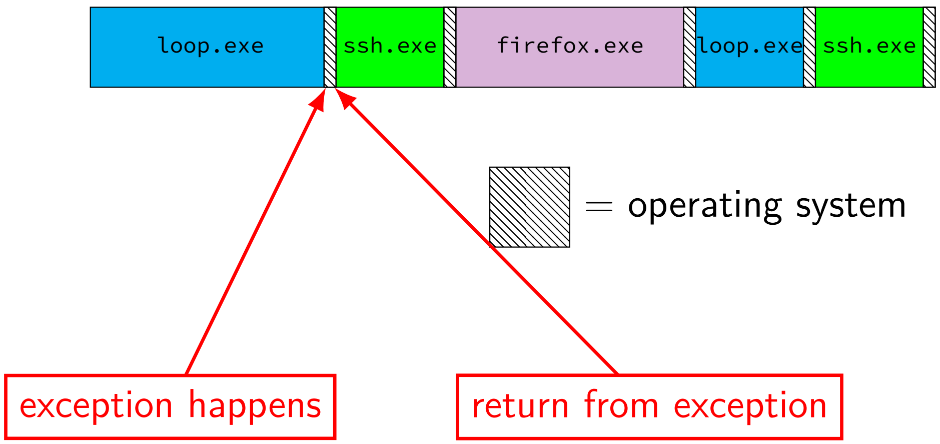
sometimes called a thread

illusion is perfect — except for performance

time multiplexing really



time multiplexing really



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

context

all registers values

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

condition codes

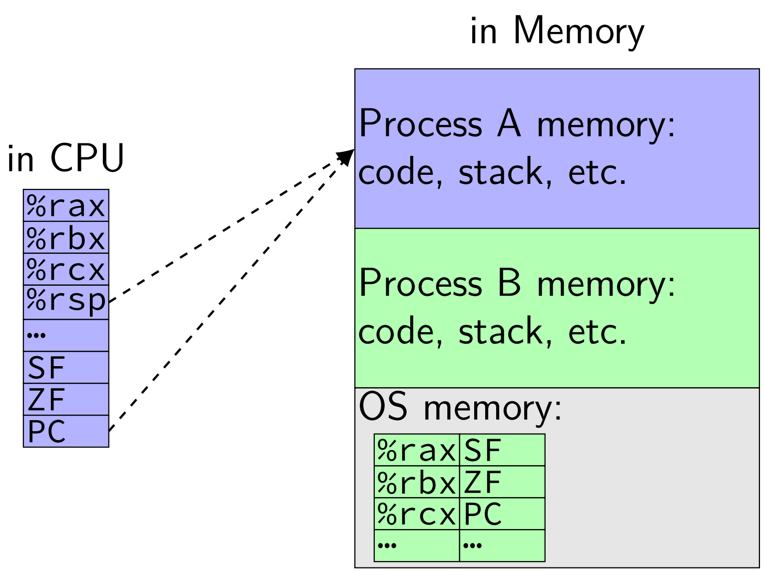
program counter

i.e. all visible state in your CPU except memory

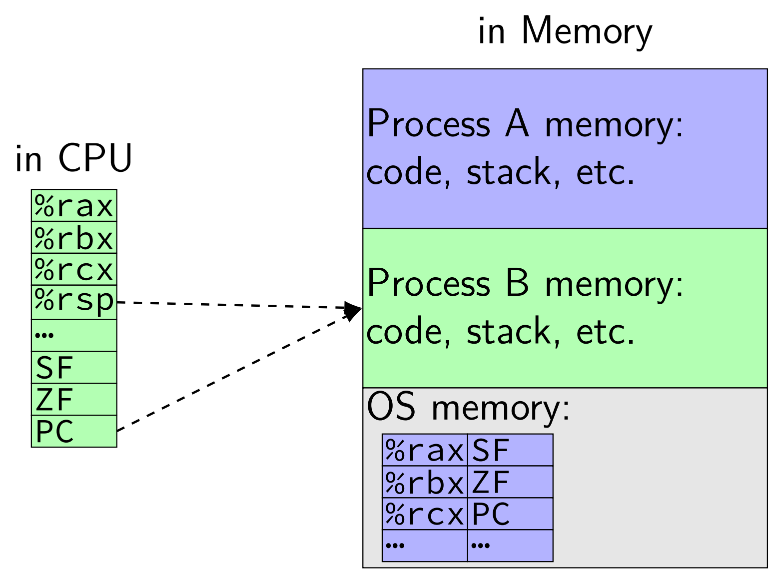
context switch pseudocode

```
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
```


contexts (A running)



contexts (B running)



memory protection

reading from another program's memory?

Program A	Program B

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

memory protection

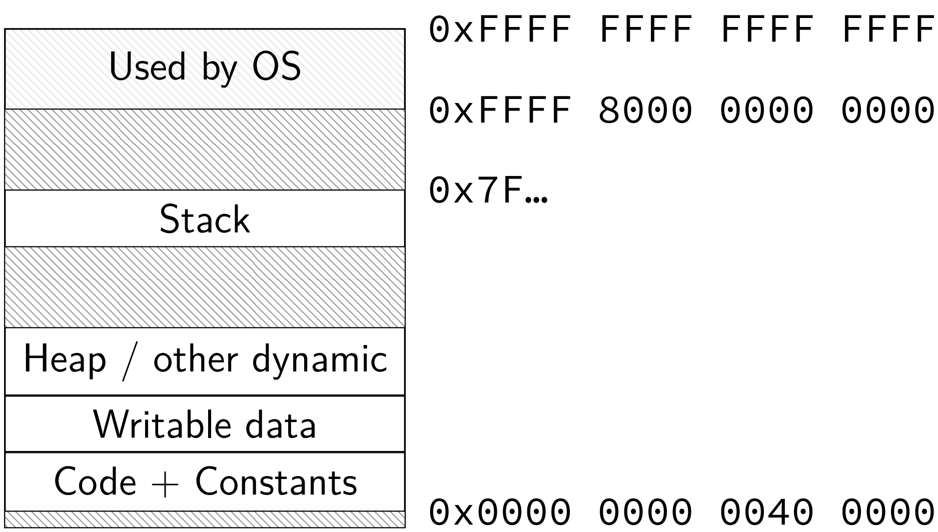
reading from another program's memory?

Program A	Program B

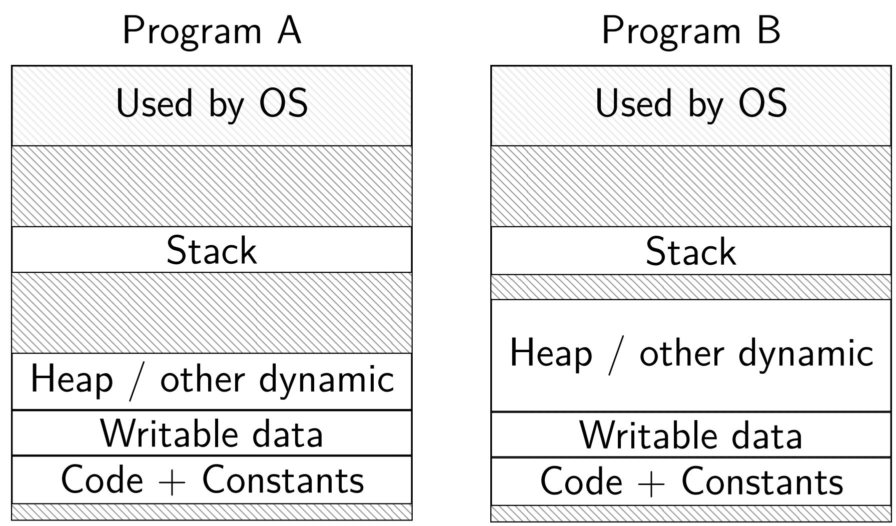
`0x10000: .word 42`
`// ...`
`// do work`
`// ...`
`movq 0x10000, %rax`
	`// while A is working:`
	`movq $99, %rax`
	`movq %rax, 0x10000`
	`...`

| result: `%rax` is 42 (always) | result: might crash |

Recall: program memory



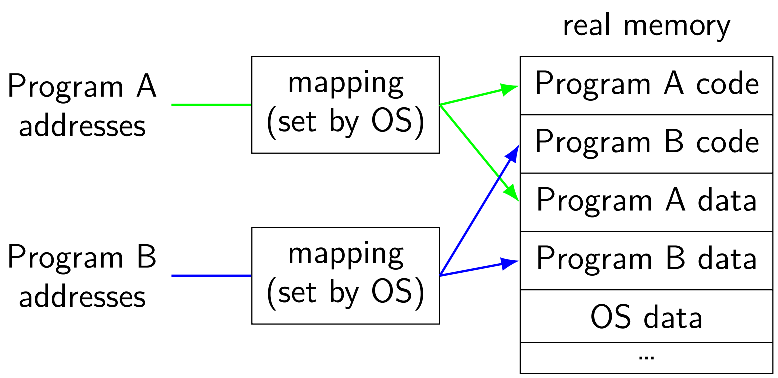
program memory (two programs)



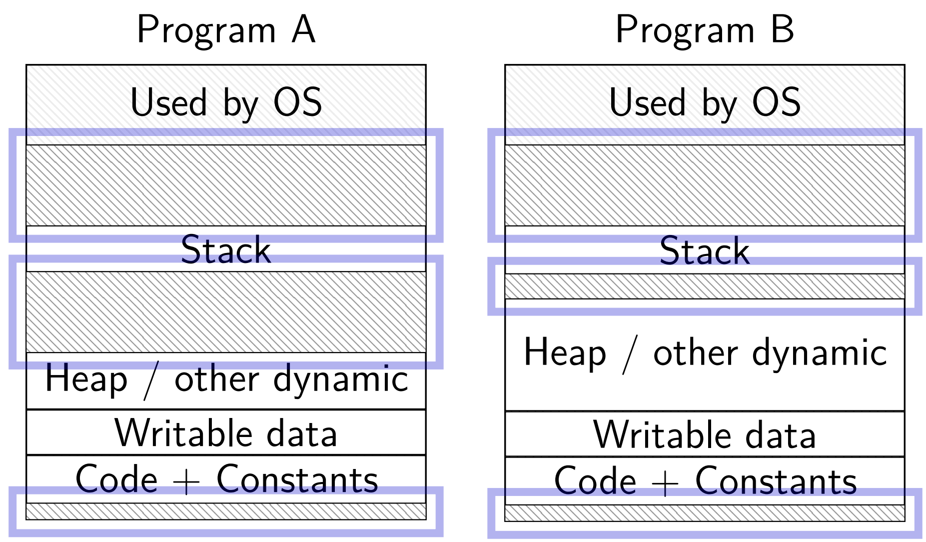
address space

programs have illusion of own memory

called a program's address space



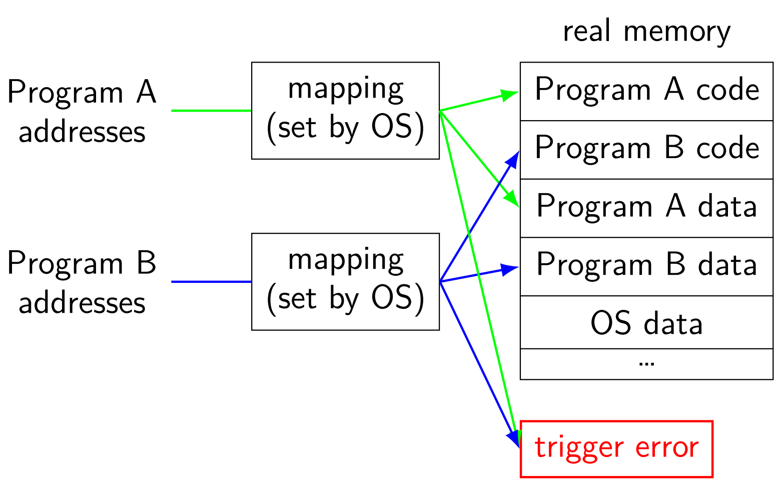
program memory (two programs)



address space

programs have illusion of own memory

called a program's address space



address space mechanisms

next week's topic

called virtual memory

mapping called page tables

mapping part of what is changed in context switch

context

all registers values

%rax %rbx, …, %rsp, …

condition codes

program counter

~~i.e. all visible state in your CPU except memory~~

address space: map from program to real addresses

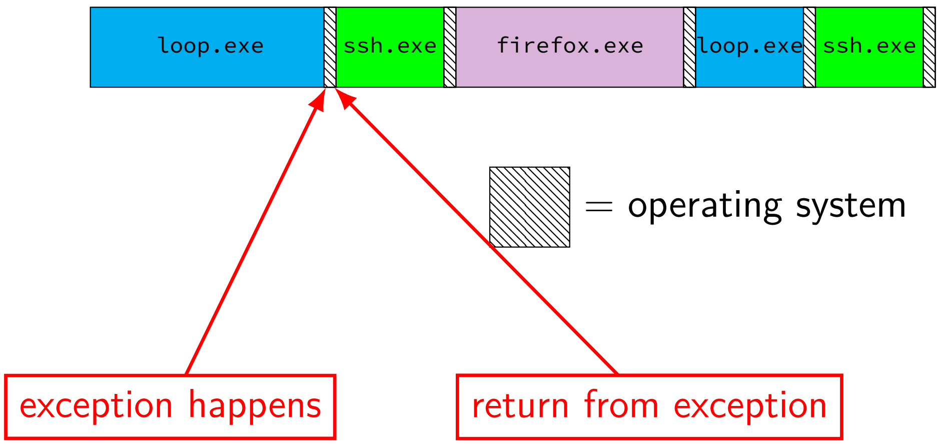
The Process

process = thread(s) + address space

illusion of dedicated machine:

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

time multiplexing really



exceptions

special control transfer

- similar effect to function call
- but often not requested by the program

usually from user programs to the OS

example: from timer expiring

- keeps our infinite loop from running forever

types of exceptions

interrupts — externally-triggered
- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...

faults — errors/events in programs
- memory not in address space ("Segmentation fault")
- divide by zero
- invalid instruction

traps — intentionally triggered exceptions
- system calls — ask OS to do something

aborts

timer interrupt

(conceptually) external timer device

OS configures before starting program

sends signal to CPU after a fixed interval

types of exceptions

interrupts — externally-triggered
- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...

faults — errors/events in programs
- memory not in address space ("Segmentation fault")
- divide by zero
- invalid instruction

traps — intentionally triggered exceptions
- system calls — ask OS to do something

aborts

protection fault

when program tries to access memory it doesn't own

e.g. trying to write to bad address

OS gets control — can crash the program
- or more interesting things

synchronous versus asynchronous

synchronous — triggered by a particular instruction
- particular `mov` instruction

asynchronous — comes from outside the program
- timer event
- keypress, other input event

exception implementation

detect condition (program error or external event)

save current value of PC somewhere

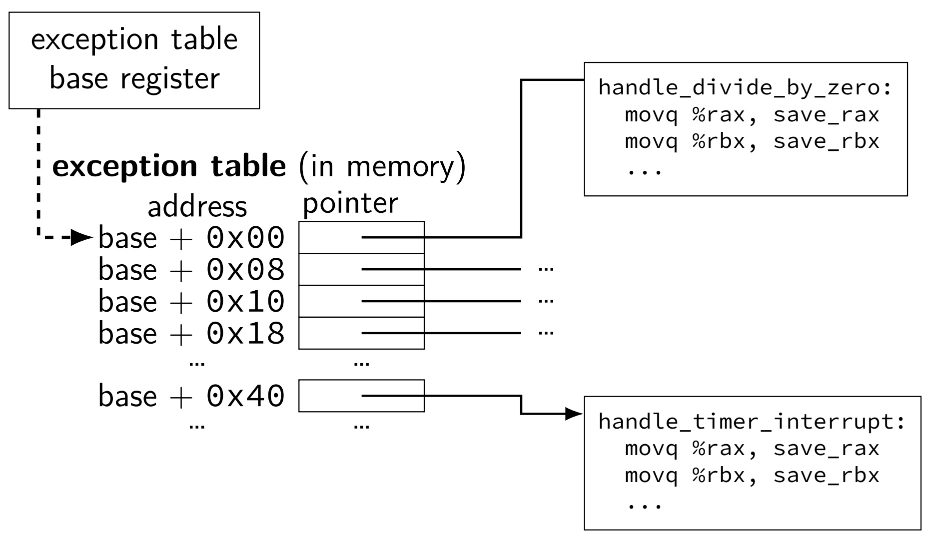
jump to exception handler (part of OS)
- jump done without program instruction to do so

exception implementation: notes

I/textbook describe a simplified version

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

locating exception handlers



running the exception handler

hardware saves the old program counter

identifies location of exception handler via table

then jumps to that location

OS code can save registers, etc., etc.

exception handler structure

1. save process's state somewhere
2. do work to handle exception
3. restore a process's state (maybe a different one)
4. jump back to program

```
handle_timer_interrupt:
  mov_from_saved_pc save_pc_loc
  movq %rax, save_rax_loc
  ... // choose new process to run here
  movq new_rax_loc, %rax
  mov_to_saved_pc new_pc
  return_from_exception
```


added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

new logic: save the old PC
- to special register or to memory

new instruction: return from exception
- i.e. jump to saved PC

added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

new logic: save the old PC
- to special register or to memory

new instruction: return from exception
- i.e. jump to saved PC

added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

new logic: save the old PC
- to special register or to memory

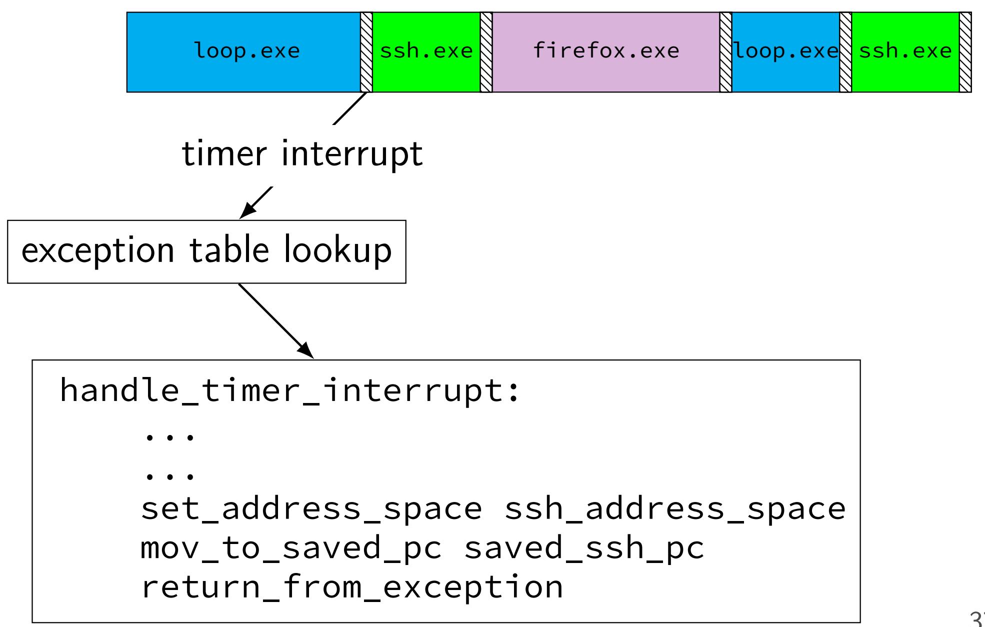
new instruction: return from exception
- i.e. jump to saved PC

why return from exception?

not just ret — can't modify process's stack
- would break the illusion of dedicated CPU

reasons related to address spaces, protection (later)

exceptions and time slicing



defeating time slices?

```
my_exception_table:
    ...
my_handle_timer_interrupt:
    // HA! Keep running me!
    return_from_exception

main:
    set_exception_table_base my_exception_table
loop:
    jmp loop
```

defeating time slices?

wrote a program that tries to set the exception table:

```
my_exception_table:
    ...

main:
    // "Load Interrupt
    //  Descriptor Table"
    // x86 instruction to set exception table
    lidt my_exception_table
    ret
```

result: `Segmentation fault` (exception!)

privileged instructions

can't let any program run some instructions

allows machines to be shared between users (e.g. lab servers)

examples:
- set exception table
- set address space
- talk to I/O device (hard drive, keyboard, display, ...)
- ...

processor has two modes:
- kernel mode — privileged instructions work
- user mode — privileged instructions cause exception instead

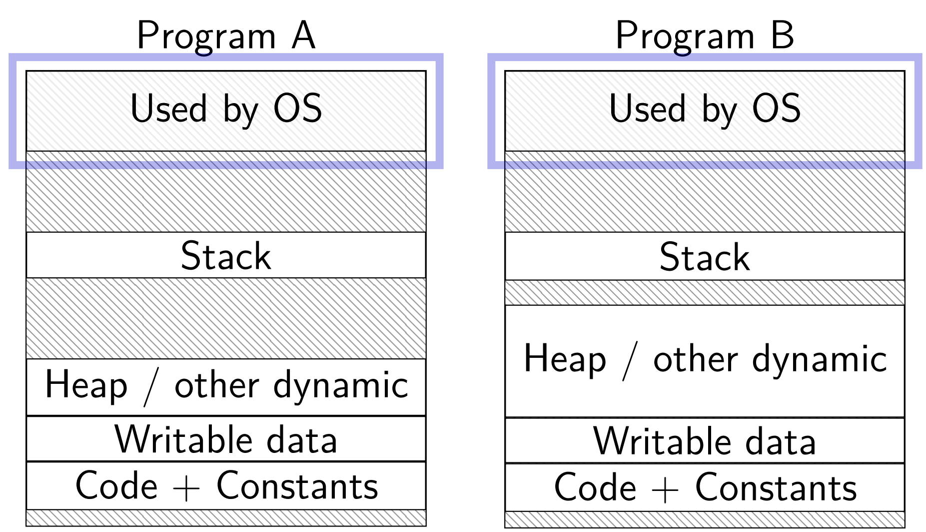
kernel mode

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

exceptions enter kernel mode

return from exception instruction leaves kernel mode

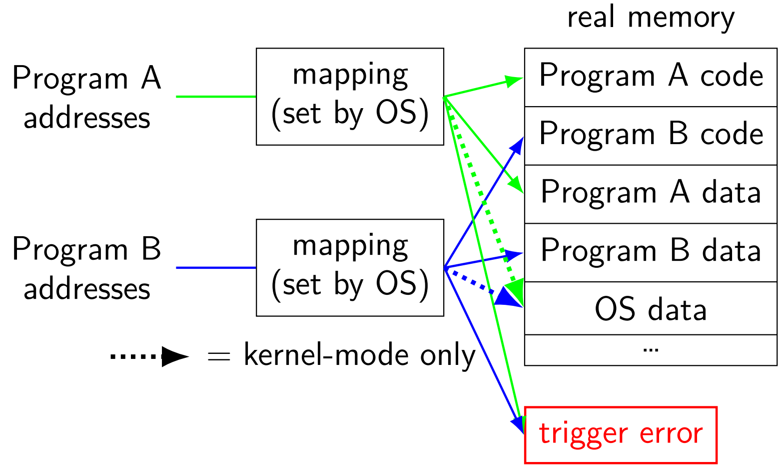
program memory (two programs)



address space

programs have illusion of own memory

called a program's address space



kernel services

allocating memory? (change address space)

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

read input? (communicate with keyborad)

all need privileged instructions!

need to run code in kernel mode

types of exceptions

interrupts — externally-triggered
- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...

faults — errors/events in programs
- memory not in address space ("Segmentation fault")
- divide by zero
- invalid instruction

traps — intentionally triggered exceptions
- system calls — ask OS to do something

aborts

Linux x86-64 system calls

special instruction: `syscall`

triggers trap (deliberate exception)

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

- terminals, etc. count as files, too

system calls and protection

exceptions are only way to access kernel mode

operating system controls what proceses can do

… by writing exception handlers very carefully

careful exception handlers

```
movq $important_os_address, %rsp
```

can't trust user's stack pointer!

need to have own stack in kernel-mode-only memory

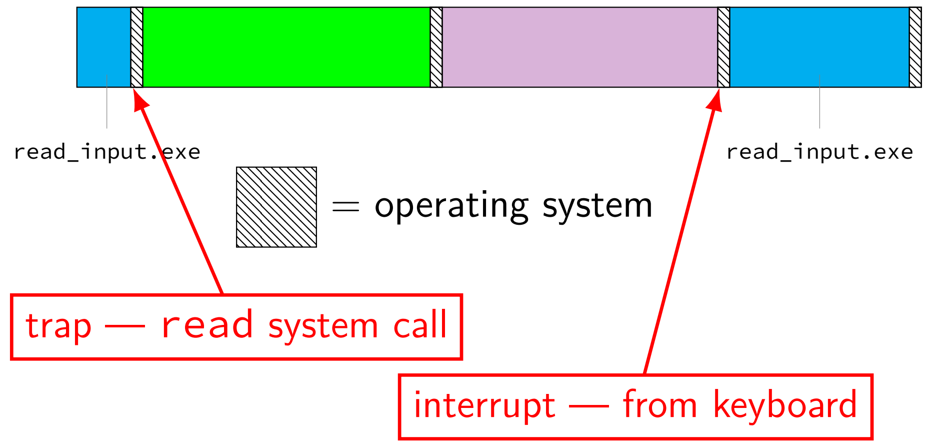
need to check all inputs really carefully

reading keyboard input

```c
int main(void) {
    char buf[1024];
    /* read a line from stdin ---
       waits for keyboard input */
    if (fgets(buf, sizeof buf, stdin) != NULL) {
        printf("You typed [%s]\n", buf);
    }
}
```

fgets uses `read` system call

keyboard input timeline



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
```

55

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, ...);
  ...
}
```

56

types of exceptions

interrupts — externally-triggered
- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, …

faults — errors/events in programs
- memory not in address space ("Segmentation fault")
- divide by zero
- invalid instruction

traps — intentionally triggered exceptions
- system calls — ask OS to do something

aborts

a note on terminology

the real world does not use consistent terms for exceptions

we will follow textbook's terms in this course

the real world won't

you might see:
- 'interrupt' meaning what we call 'exception' (x86)
- 'exception' meaning what we call 'fault'
- 'fault' meaning what we call 'fault' *or* 'abort' (ARM)
- … and more

signals

Unix-like operating system feature

like interrupts for processes:

can be triggered by external process (instead of device)
- kill command/system call

can be triggered by special events
- pressing control-C

can invoke signal handler

signal API

`sigaction` — register handler for signal

`kill` — send signal to process

`pause` — put process to sleep until signal received

`sigprocmask` — block some signals from being received until ready

… and much more

example signal program

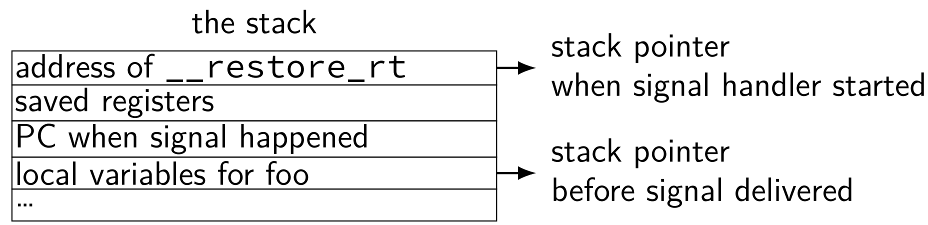
```
#include <signal.h>
#include <unistd.h>

void handle_sigint(int signum) {
    write(1, "Got␣signal!\n", sizeof("Got␣signal!
    _exit(0);
}

int main(void) {
    struct sigaction act;
    act.sa_handler = &handle_sigint;
    sigemptyset(&act.sa_mask);
    sigaction(&act);

    char buf[1024];
    while (fgets(buf, sizeof buf, stdin)) {
```


signal delivery (1)

signal happens while `foo()` is running

OS writes stack from to user stack

OS modifies registers to call signal handler



signal delivery (2)

```
handle_sigint:
    ...
    ret
...
__restore_rt:
    // 15 = "sigreturn" system call
    movq $15, %rax
    syscall
```

`__restore_rt` is return address for signal handler

system call restores pre-signal state, then returns

signal handler unsafety (1)

```
void *malloc(size_t size) {
    ...
    to_return = next_to_return;
    /* SIGNAL HAPPENS HERE */
    next_to_return += size;
    return to_return;
}

void foo() {
    /* This malloc() call interrupted */
    char *p = malloc(1024);
    p[0] = 'x';
}

void handle_sigint() {
    // printf might use malloc()
```

setjmp/longjmp

C flow control

```
jmp_buf env;

main() {
  if (setjmp(env) == 0) { // like try {
    ...
    read_file()
    ...
  } else { // like catch
    printf("some error happened\n");
  }
}

read_file() {
  ...
```


implementing setjmp/lonjmp

setjmp:

copy all registers to `jmp_buf`

... including stack pointer

longjmp

copy registers from `jmp_buf`

... but change `%rax` (return value)

setjmp weirdness — local variables

Undefined behavior:

```
int x = 0;
if (setjmp(env) == 0) {
    ...
    x += 1;
    longjmp(env, 1);
} else {
    printf("%d\n", x);
}
```


setjmp weirdness — fix

Defined behavior:

```
volatile int x = 0;
if (setjmp(env) == 0) {
    ...
    x += 1;
    longjmp(env, 1);
} else {
    printf("%d\n", x);
}
```

on implementing try/catch

could do something like setjmp()/longjmp()

but want try to be really fast!

instead: tables of information indexed by program counters:

- where register values are stored on stack/in other registers
- where old program counters are stored on stack
- where catch blocks are located