last time

SIMD (single instruction multiple data)

hardware idea: wider ALUs and registers

Intel's interface `_mm...`

sharing the CPU: context switching

context = visible CPU state (registers, condition codes, PC, ...)

exceptions = OS gets run *by the processor*

logistics: the final

final exam location: Wilson 402

10 May, 7PM

fill out the conflict form very soon if you can't make it

logistics: lab this week

using SIMD stuff

preview for smooth HW

some optional parts — some students get stuck on earlier parts

- but I expect many of you to have time
- maybe better explanation in lecture??

please try — get comfortable for smooth

a note on smooth

it takes most students conisderably more time than rotate

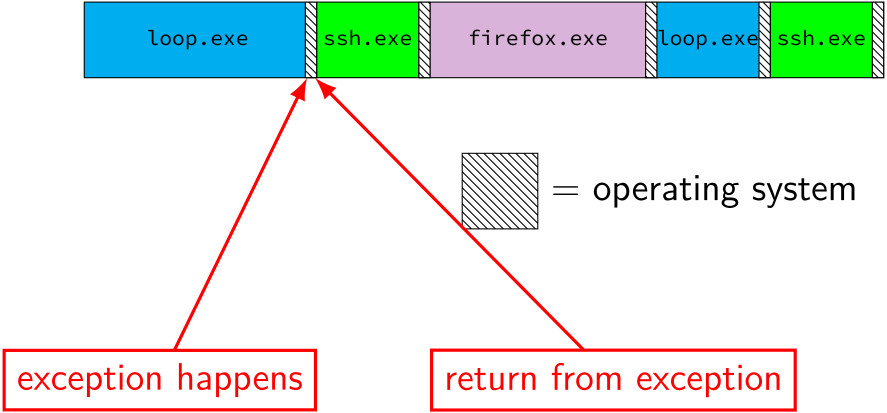
start early

...especially if you have trouble with the lab

time multiplexing really

▨ = operating system

time multiplexing really



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

▨ = 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

context

all registers values

`%rax %rbx`, …, `%rsp`, …

condition codes

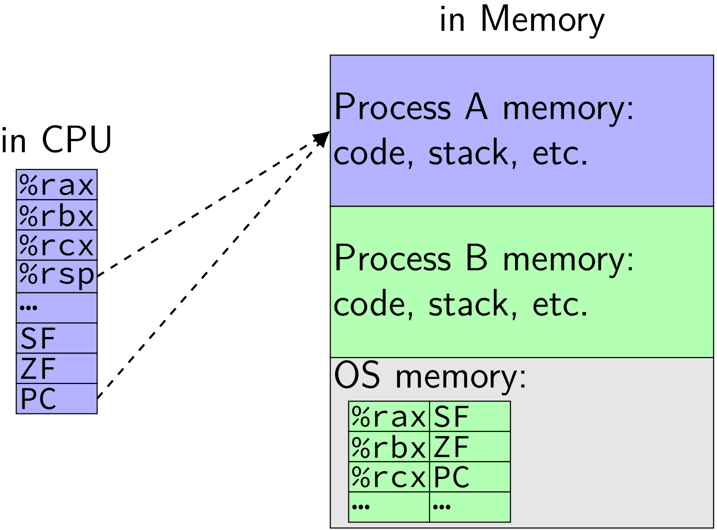
program counter

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)



in Memory

Process A memory:
code, stack, etc.

Process B memory:
code, stack, etc.

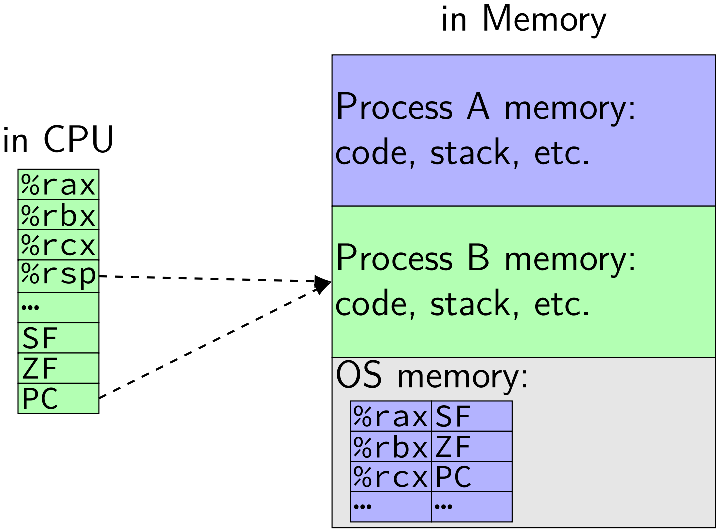
OS memory:

%rax	SF
%rbx	ZF
%rcx	PC
…	…

in CPU

%rax
%rbx
%rcx
%rsp
…
SF
ZF
PC

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

memory protection

reading from another program's memory?

Program A

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

Program B

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

memory protection

reading from another program's memory?

Program A

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

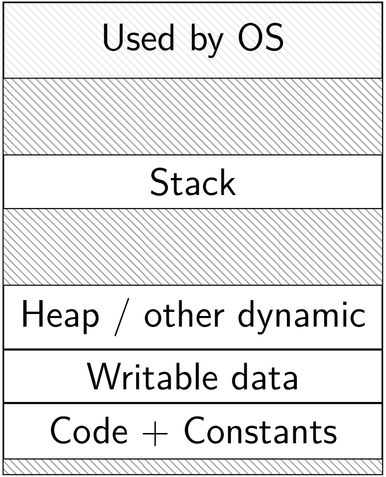
result: `%rax` is 42 (always)

Program B

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

result: might crash

program memory



Used by OS

Stack

Heap / other dynamic

Writable data

Code + Constants

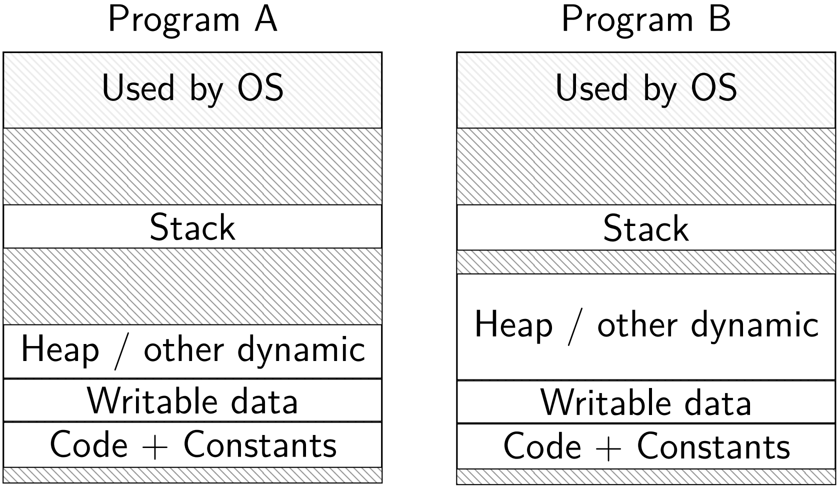
`0xFFFF FFFF FFFF FFFF`

`0xFFFF 8000 0000 0000`

`0x7F…`

`0x0000 0000 0040 0000`

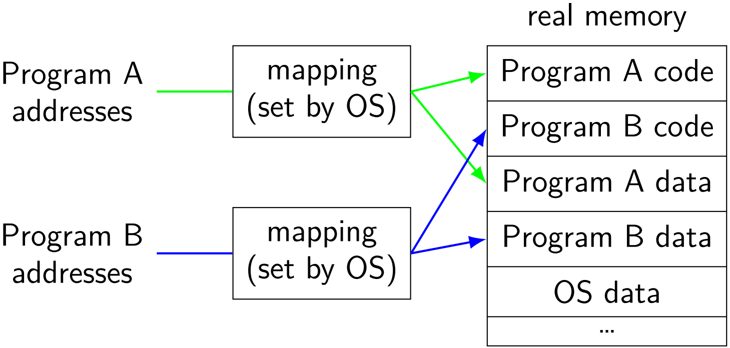
program memory (two programs)



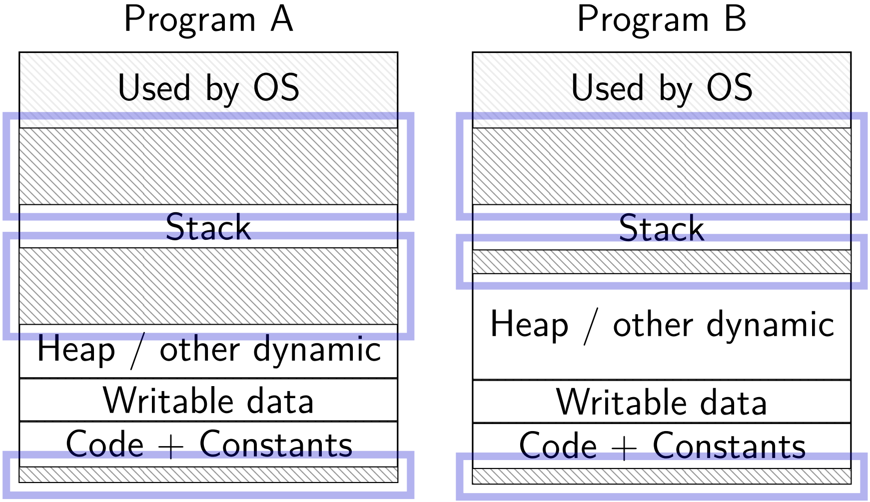
address space

programs have illusion of own memory

called a program's address space



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

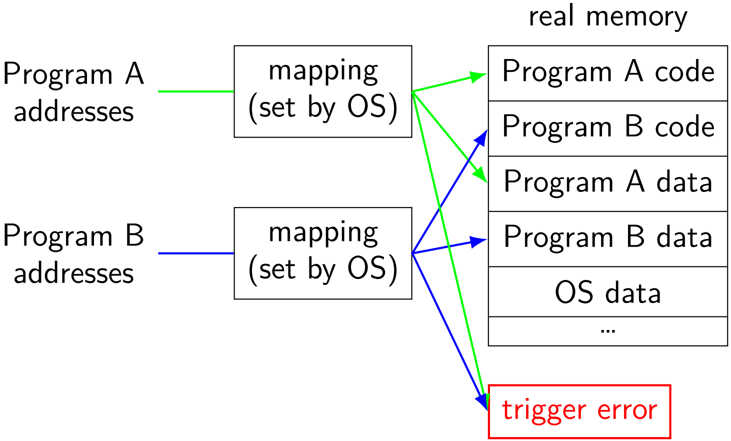
Writable data

Code + Constants

address space

programs have illusion of own memory

called a program's address space



address space mechanisms

next 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

synchronous versus asynchronous

synchronous — triggered by a particular instruction

- traps and faults

asynchronous — comes from outside the program

- interrupts and aborts
- timer event
- keypress, other input event

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

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

    (usually on same chip as processor)

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

types of exceptions

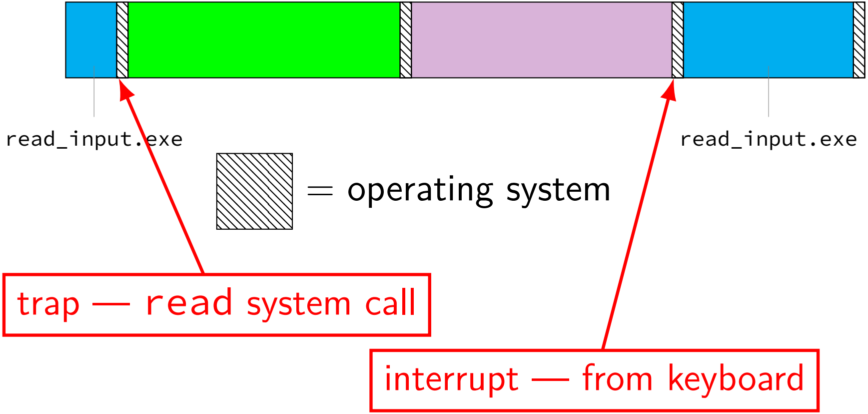
interrupts — externally-triggered
  timer — keep program from hogging CPU
  I/O devices — key presses, hard drives, networks, …

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

keyboard input timeline



`read_input.exe`

`read_input.exe`

▨ = operating system

trap — read system call

interrupt — from keyboard

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

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

exception implementation

detect condition (program error or external event)

save current value of PC somewhere

jump to exception handler (part of OS)

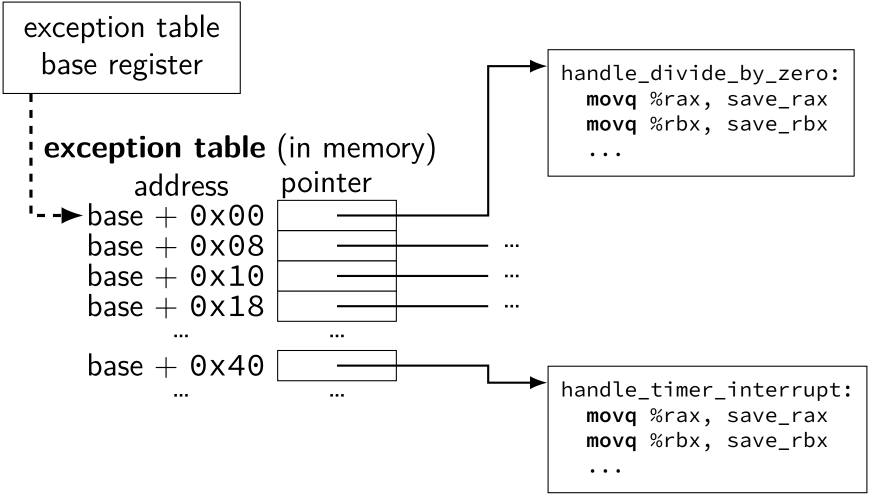
- jump done without program instruction to do so

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



```
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.

added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

new logic: save the old PC (and maybe more)

- 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 (and maybe more)
- 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 (and maybe more)
- 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 (and maybe more)
- to special register or to memory

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

why return from exception?

reasons related to protection (later)

not just ret — can't modify process's stack

- would break the illusion of dedicated CPU/memory
- program could use stack in weird way

```
movq $100, −8(%rsp)
...
movq −8(%rsp), %rax
```

(even though this wouldn't be following calling conventions)

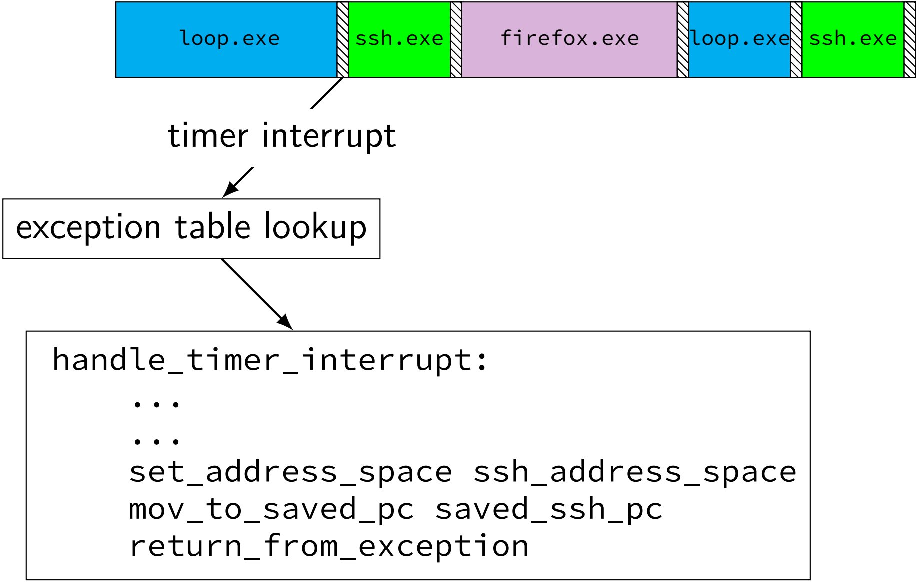
- need to restart program undetectably!

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
```

exceptions and time slicing



```
handle_timer_interrupt:
    ...
    ...
    set_address_space ssh_address_space
    mov_to_saved_pc saved_ssh_pc
    return_from_exception
```

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

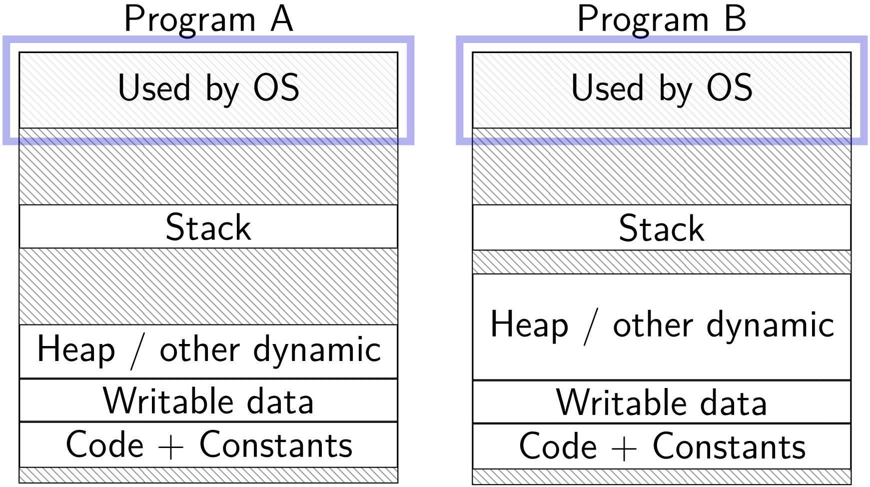
kernel mode

extra one-bit register: “are we in kernel mode”

exceptions enter kernel mode

return from exception instruction leaves kernel mode

program memory (two programs)


Program A

Used by OS

Stack

Heap / other dynamic

Writable data

Code + Constants

Program B

Used by OS

Stack

Heap / other dynamic

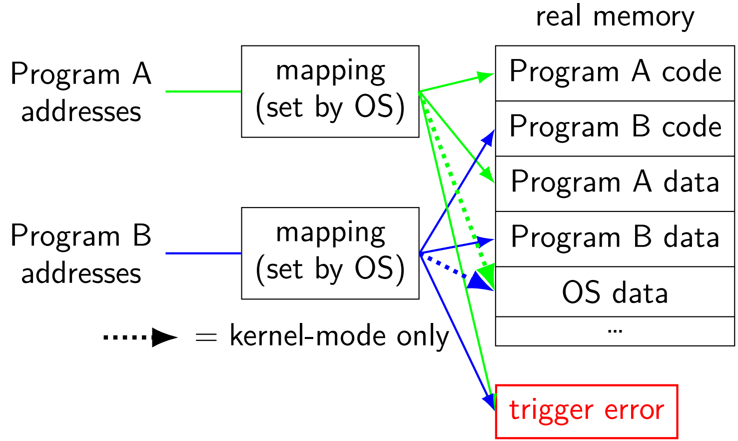
Writable data

Code + Constants

address space

programs have illusion of own memory

called a program's address space



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

when program tries to do other things that are not allowed

e.g. accessing I/O devices directly

e.g. changing exception table base register

OS gets control — can crash the program

- or more interesting things

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

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

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

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

```
/* 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, ...);
  ...
}
```

a note on terminology (1)

real world: inconsistent 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'
- 'hard fault' meaning what we call 'abort'
- 'trap' meaning what we call 'fault'
- … and more

a note on terminology (2)

we use the term “kernel mode”

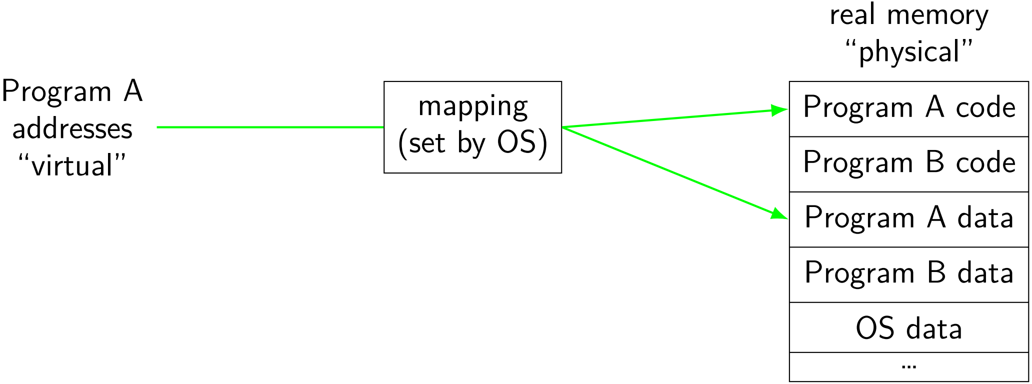
some additional terms:

- supervisor mode
- privileged mode
- ring `0`

some systems have multiple levels of privilege

- different sets of priviliged operations work

address translation



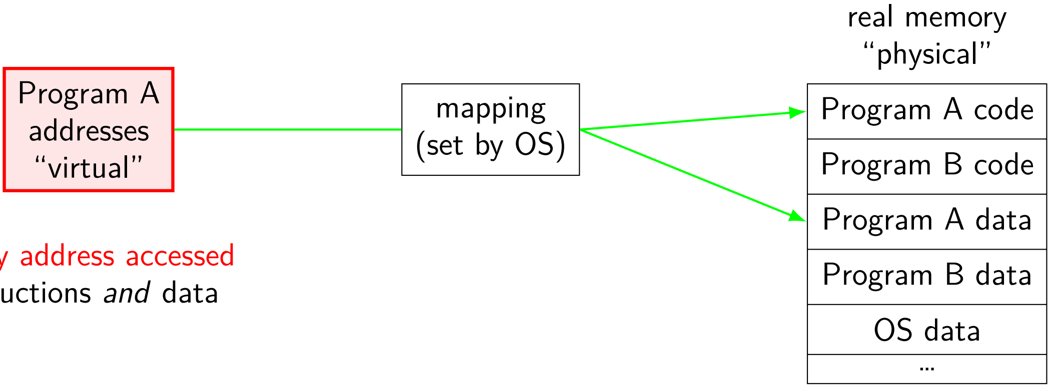
Program A
addresses
“virtual”

mapping
(set by OS)

real memory
“physical”

Program A code
Program B code
Program A data
Program B data
OS data
...

address translation



real memory
"physical"

Program A
addresses
"virtual"

mapping
(set by OS)

Program A code

Program B code

Program A data

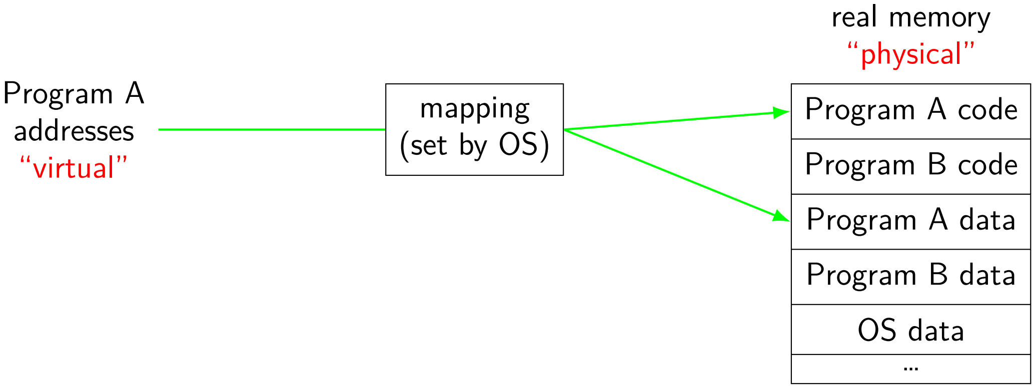
Program B data

OS data

...

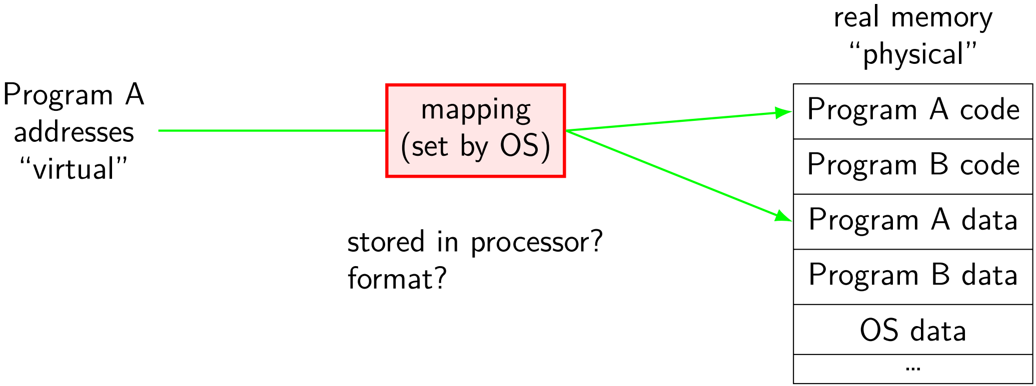
every address accessed
instructions *and* data

address translation



program addresses are ‘virtual’
real addresses are ‘physical’
can be different sizes!

address translation



Program A
addresses
"virtual"

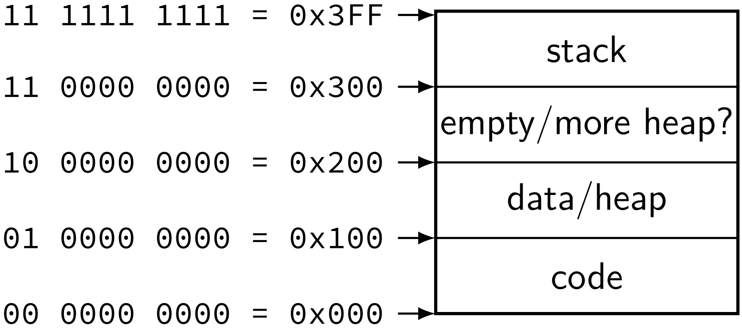
mapping
(set by OS)

real memory
"physical"

Program A code
Program B code
Program A data
Program B data
OS data
...

stored in processor?
format?

toy program memory



11 1111 1111 = 0x3FF

11 0000 0000 = 0x300

10 0000 0000 = 0x200

01 0000 0000 = 0x100

00 0000 0000 = 0x000

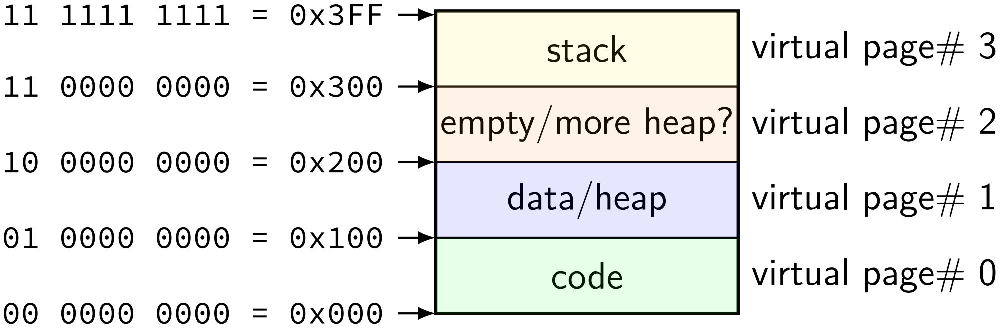
stack

empty/more heap?

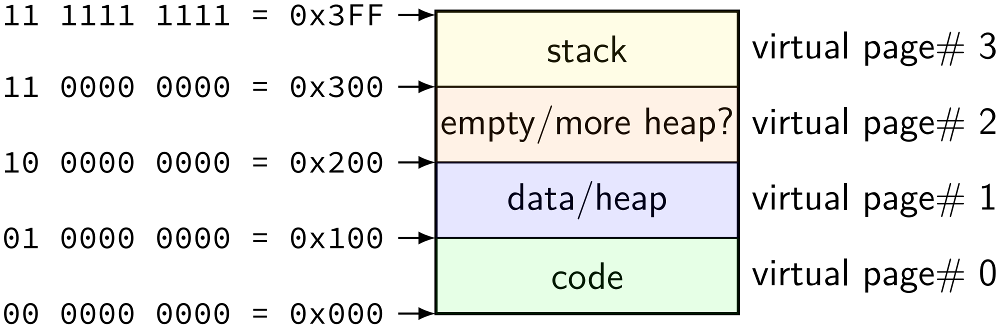
data/heap

code

toy program memory

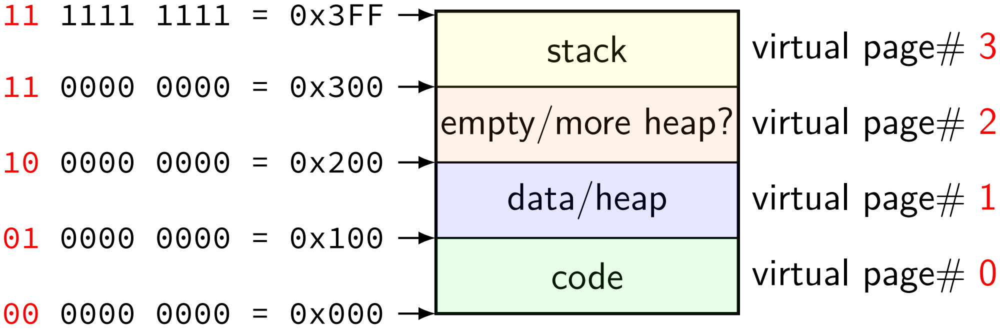


toy program memory



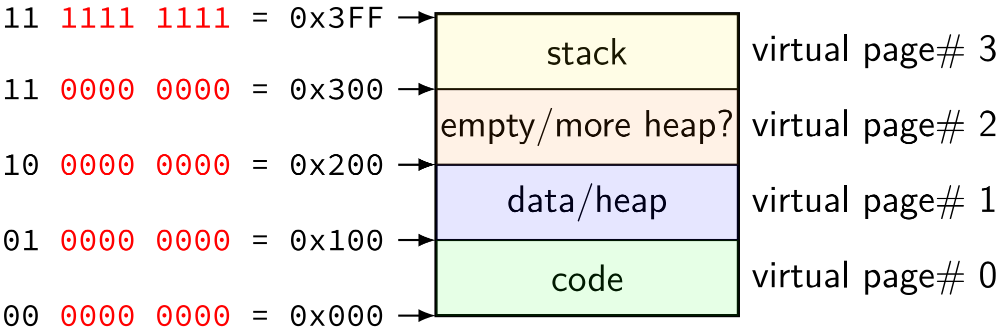
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

toy program memory



page number is upper bits of address
(because page size is power of two)

toy program memory



11 1111 1111 = 0x3FF

11 0000 0000 = 0x300

10 0000 0000 = 0x200

01 0000 0000 = 0x100

00 0000 0000 = 0x000

stack

empty/more heap?

data/heap

code

virtual page# 3

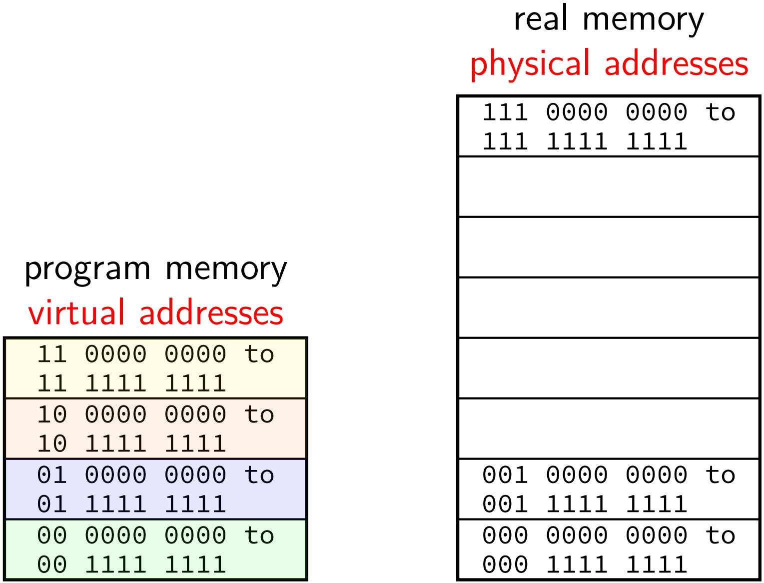
virtual page# 2

virtual page# 1

virtual page# 0

rest of address is called page offset

toy physical memory



real memory

physical addresses

`111 0000 0000 to`
`111 1111 1111`

`001 0000 0000 to`
`001 1111 1111`

`000 0000 0000 to`
`000 1111 1111`

program memory

virtual addresses

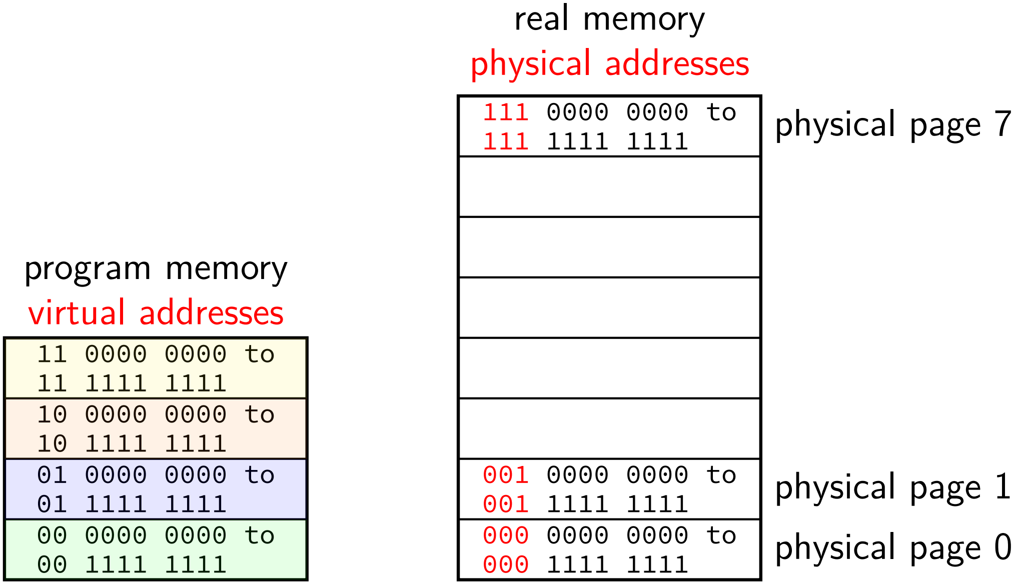
`11 0000 0000 to`
`11 1111 1111`

`10 0000 0000 to`
`10 1111 1111`

`01 0000 0000 to`
`01 1111 1111`

`00 0000 0000 to`
`00 1111 1111`

toy physical memory



real memory

physical addresses

```
111 0000 0000 to
111 1111 1111
```

physical page 7

```
001 0000 0000 to
001 1111 1111
```

physical page 1

```
000 0000 0000 to
000 1111 1111
```

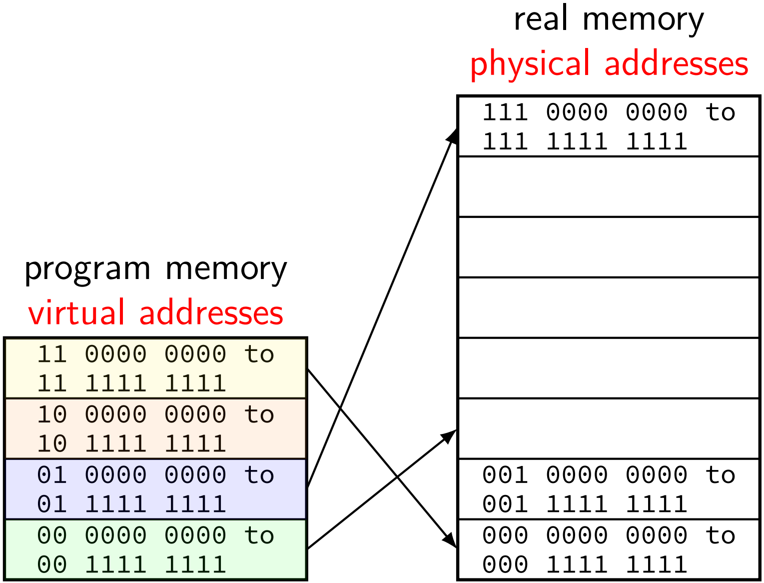
physical page 0

program memory

virtual addresses

```
11 0000 0000 to
11 1111 1111
10 0000 0000 to
10 1111 1111
01 0000 0000 to
01 1111 1111
00 0000 0000 to
00 1111 1111
```

toy physical memory



real memory

physical addresses

```
111 0000 0000 to
111 1111 1111
```

```
001 0000 0000 to
001 1111 1111
```

```
000 0000 0000 to
000 1111 1111
```

program memory

virtual addresses

```
11 0000 0000 to
11 1111 1111
```

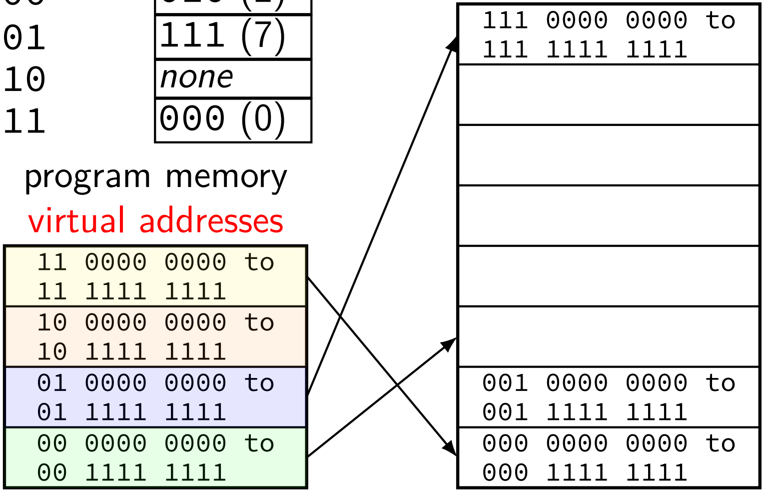
```
10 0000 0000 to
10 1111 1111
```

```
01 0000 0000 to
01 1111 1111
```

```
00 0000 0000 to
00 1111 1111
```

toy physical memory

virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
11	000 (0)



program memory

virtual addresses

`11 0000 0000 to`
`11 1111 1111`

`10 0000 0000 to`
`10 1111 1111`

`01 0000 0000 to`
`01 1111 1111`

`00 0000 0000 to`
`00 1111 1111`

real memory

physical addresses

`111 0000 0000 to`
`111 1111 1111`

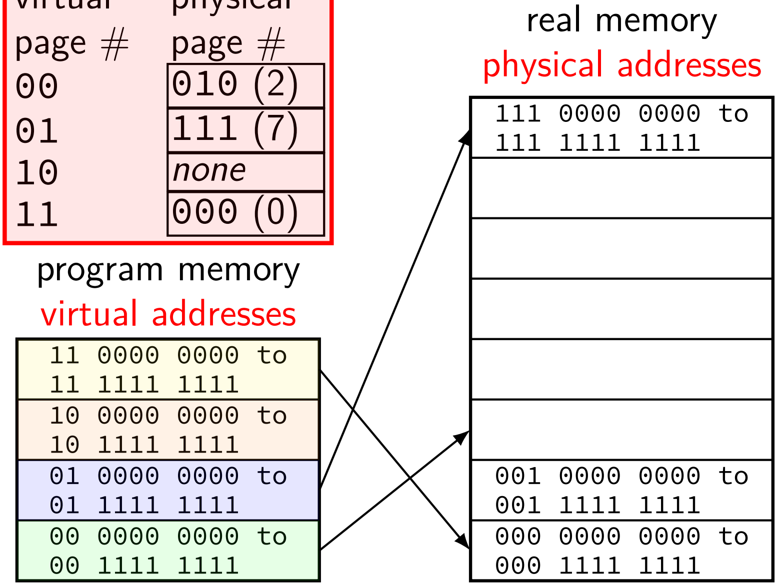
`001 0000 0000 to`
`001 1111 1111`

`000 0000 0000 to`
`000 1111 1111`

toy physical memory

page table!

virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
11	000 (0)



real memory

physical addresses

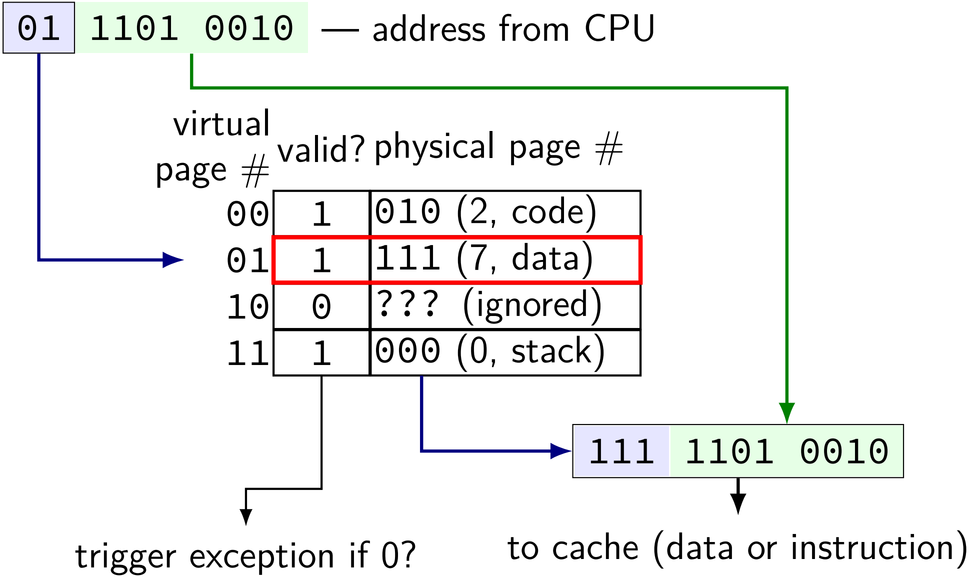
program memory

virtual addresses

toy page table lookup

virtual page #	valid?	physical page #
`00`	`1`	`010` (2, code)
`01`	`1`	`111` (7, data)
`10`	`0`	`???` (ignored)
`11`	`1`	`000` (0, stack)

toy page table lookup



01 1101 0010 — address from CPU

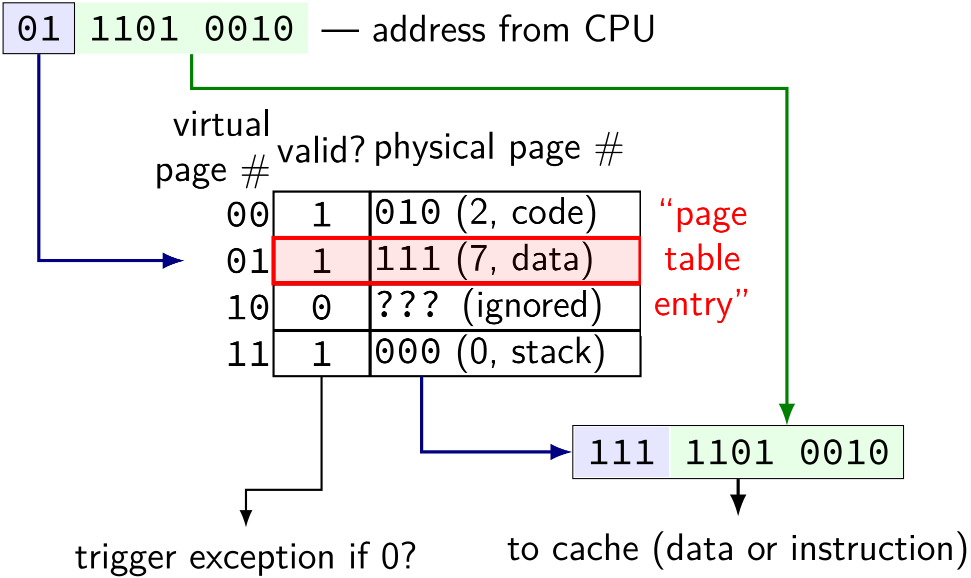
virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

111 1101 0010

trigger exception if 0?

to cache (data or instruction)

toy page table lookup



`01` `1101 0010` — address from CPU

virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

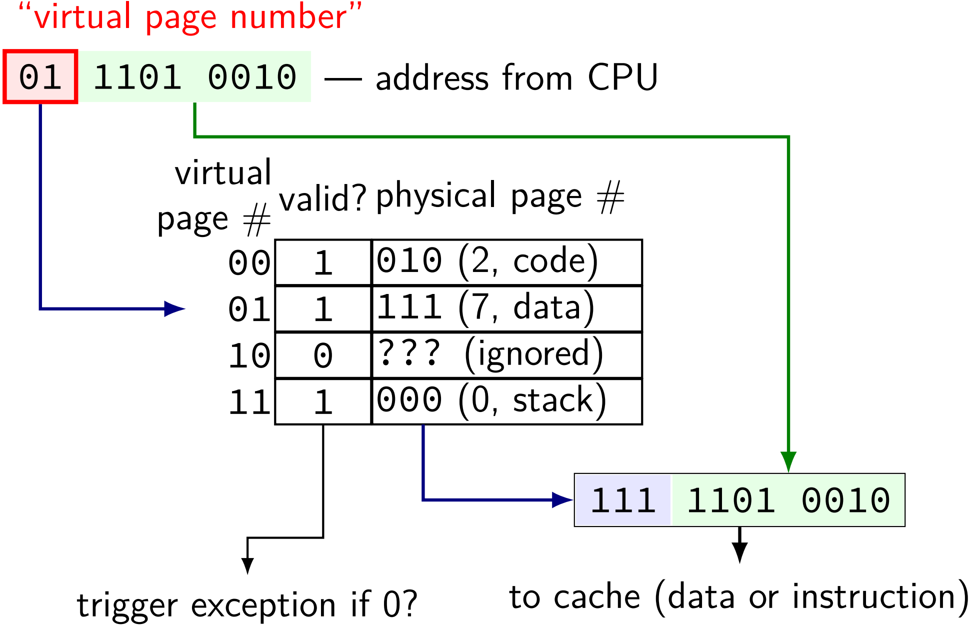
“page table entry”

`111` `1101 0010`

trigger exception if 0?

to cache (data or instruction)

toy page table lookup



“virtual page number”

`01` `1101 0010` — address from CPU

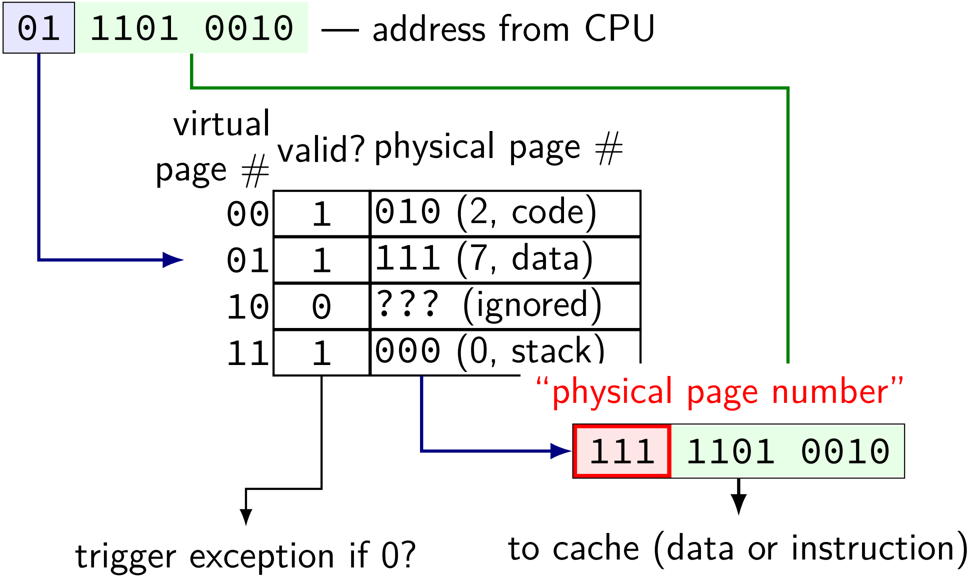
virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

`111` `1101 0010`

trigger exception if 0?

to cache (data or instruction)

toy page table lookup



01 1101 0010 — address from CPU

virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

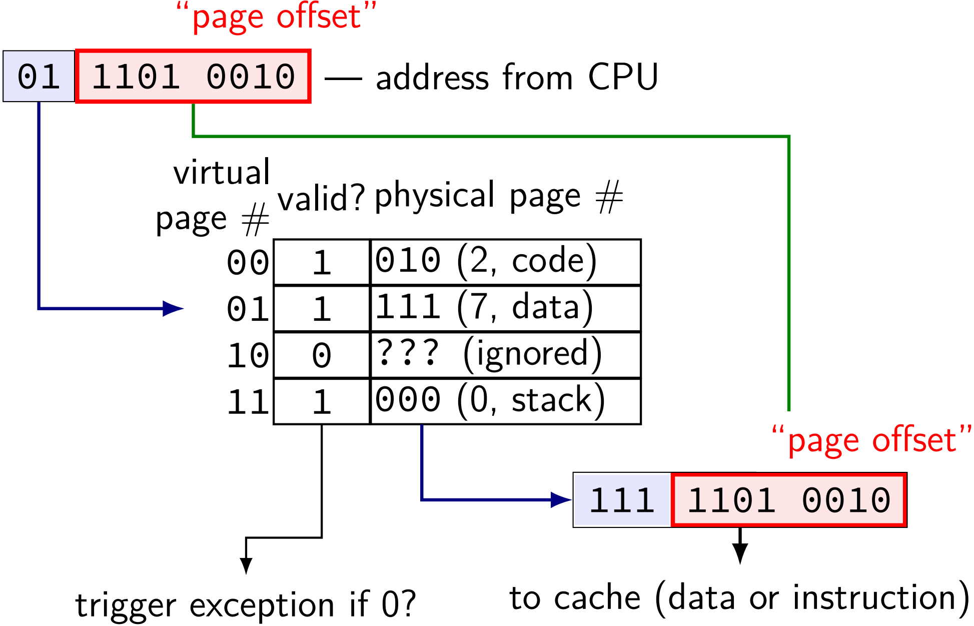
“physical page number”

111 1101 0010

trigger exception if 0?

to cache (data or instruction)

toy page table lookup



“page offset”

`01` `1101 0010` — address from CPU

virtual page #	valid?	physical page #
`00`	1	`010` (2, code)
`01`	1	`111` (7, data)
`10`	0	??? (ignored)
`11`	1	`000` (0, stack)

“page offset”

`111` `1101 0010`

trigger exception if 0?

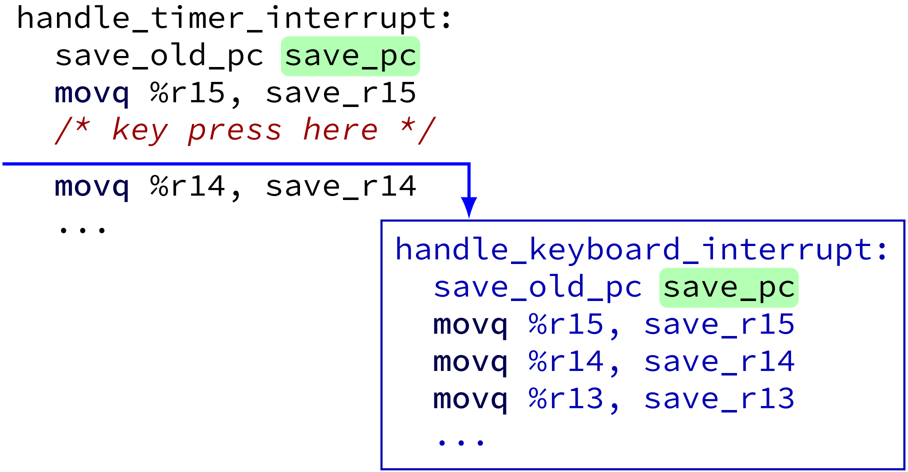
to cache (data or instruction)

backup sldies

exceptions in exceptions

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

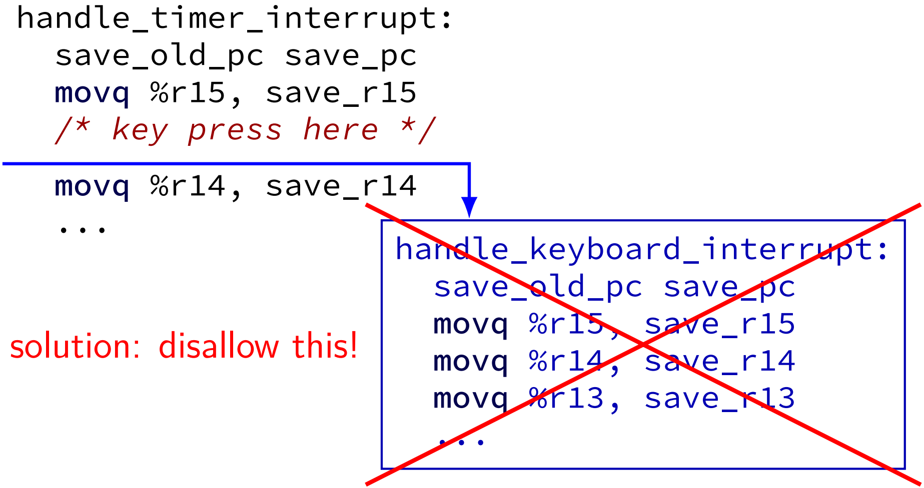
exceptions in exceptions



```
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
  ...
```

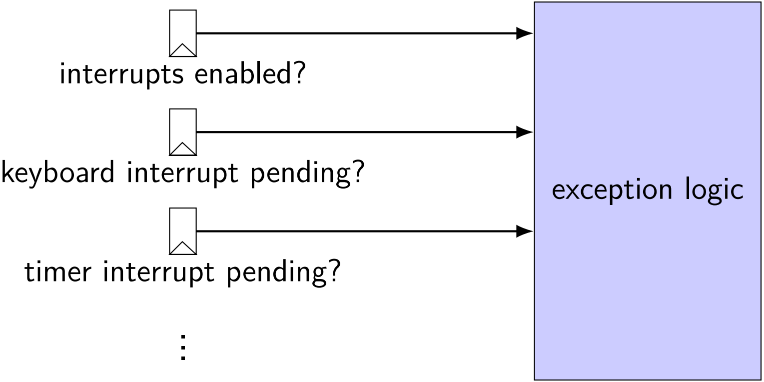
solution: disallow this!

interrupt disabling

CPU supports disabling (most) interrupts

interrupts will wait until it is reenabled

CPU has extra state:



exceptions in exceptions

```
handle_timer_interrupt:
  /* interrupts automatically disabled here */
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */
  movq %r14, save_r14
  ...
  call move_saved_state
  enable_interrupts

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

exceptions in exceptions

```
handle_timer_interrupt:
  /* interrupts automatically disabled here */
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */
  movq %r14, save_r14
  ...
  call move_saved_state
  enable_interrupts

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

exceptions in exceptions

```
handle_timer_interrupt:
  /* interrupts automatically disabled here */
  save_old_pc save_pc
  movq %r15, save_r15
  /* key press here */
  movq %r14, save_r14
  ...
  call move_saved_state
  enable_interrupts
  /* interrupt happens here! */
  ...
```

```
handle_keyboard_interrupt:
  save_old_pc save_pc
  ...
  call move_saved_state
```

disabling interrupts

automatically disabled when exception handler starts

also done with privileged instruction:

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

on virtual machines

process can be called a ‘virtual machine’

programmed like a complete computer…

on virtual machines

process can be called a ‘virtual machine’

programmed like a complete computer…

but weird interface for I/O, memory — system calls

can we make that closer to the real machine?

trap-and-emulate

privileged instructions trigger a protection fault

we assume operating system crashes

what if OS pretends the privileged instruction works?

trap-and-emulate: write-to-screen

```
struct Process {
    AddressSpace address_space;
    SavedRegisters registers;
};

void handle_protection_fault(Process *process) {
    // normal: would crash
    if (was_write_to_screen()) {
        do_write_system_call(process);
        process->registers->pc +=
            WRITE_TO_SCREEN_LENGTH;
    } else {
        ...
    }
}
```

trap-and-emulate: write-to-screen

```
struct Process {
    AddressSpace address_space;
    SavedRegisters registers;
};

void handle_protection_fault(Process *process) {
    // normal: would crash
    if (was_write_to_screen()) {
        do_write_system_call(process);
        process->registers->pc +=
            WRITE_TO_SCREEN_LENGTH;
    } else {
        ...
    }
}
```

was_write_to_screen()

how does OS know what caused protection fault?

option 1: hardware “type” register

option 2: check instruction:

```
int opcode = (*process->registers->pc & 0xF0) >> 4;
if (opcode == WRITE_TO_SCREEN_OPCODE)
    ...
```

trap-and-emulate: write-to-screen

```
struct Process {
    AddressSpace address_space;
    SavedRegisters registers;
};

void handle_protection_fault(Process *process) {
    // normal: would crash
    if (was_write_to_screen()) {
        do_write_system_call(process);
        process->registers->pc +=
            WRITE_TO_SCREEN_LENGTH;
    } else {
        ...
    }
}
```

trap-and-emulate: write-to-screen

```
struct Process {
    AddressSpace address_space;
    SavedRegisters registers;
};

void handle_protection_fault(Process *process) {
    // normal: would crash
    if (was_write_to_screen()) {
        do_write_system_call(process);
        process->registers->pc +=
            WRITE_TO_SCREEN_LENGTH;
    } else {
        ...
    }
}
```

system virtual machines

turn faults into system calls

emulate machine that looks more like 'real' machine

what software like VirtualBox, VMWare, etc. does

more complicated than this:

- on x86, some privileged instructions don't cause faults
- dealing with address spaces is a lot of extra work

process VM versus system VM

Linux process feature	real machine feature
files, sockets	I/O devices
threads	CPU cores
mmap/brk (used by malloc)	???
signals	exceptions

setjmp/longjmp

```
jmp_buf env;

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

read_file() {
  ...
  if (open failed) {
      longjmp(env, 1) // like throw
  }
  ...
}
```

implementing setjmp/longjmp

setjmp:

- copy all registers to `jmp_buf`
- … including stack pointer

longjmp

- copy registers from `jmp_buf`
- … but change `%rax` (return value)

setjmp psuedocode

setjmp: looks like first half of context switch

```
setjmp:
  movq %rcx, env->rcx
  movq %rdx, env->rdx
  movq %rsp + 8, env->rsp // +8: skip return value
  ...
  save_condition_codes env->ccs
  movq 0(%rsp), env->pc
  movq $0, %rax // always return 0
  ret
```

longjmp psuedocode

longjmp: looks like second half of context switch

```
longjmp:
  movq %rdi, %rax // return a different value
  movq env->rcx, %rcx
  movq env->rdx, %rdx
  ...
  restore_condition_codes env->ccs
  movq env->rsp, %rsp
  jmp env->pc
```

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 setjmp is slow

on implementing try/catch

could do something like setjmp()/longjmp()

but setjmp is slow

low-overhead try/catch (1)

```
main() {
  printf("about␣to␣read␣file\n");
  try {
    read_file();
  } catch(...) {
    printf("some␣error␣happened\n");
  }
}
read_file() {
  ...
  if (open failed) {
      throw IOException();
  }
  ...
}
```

low-overhead try/catch (2)

```
main:
   ...
   call printf
start_try:
   call read_file
end_try:
   ret
```

```
main_catch:
   movq $str, %rdi
   call printf
   jmp end_try
```

```
read_file:
   pushq %r12
   ...
   call do_throw
   ...
end_read:
   popq %r12
   ret
```

lookup table

program counter range	**action**	**recurse?**
`start_try` to `end_try`	`jmp main_catch`	no
`read_file` to `end_read`	`popq %r12, ret`	yes
anything else	error	—

low-overhead try/catch (2)

```
main:
  ...
  call printf
start_try:
  call read_file
end_try:
  ret
```

```
main_catch:
  movq $str, %rdi
  call printf
  jmp end_try
```

```
read_file:
  pushq %r12
  ...
  call do_throw
  ...
end_read:
  popq %r12
  ret
```

lookup table

program counter range	**action**	**recurse?**
`start_try` to `end_try`	`jmp main_catch`	no
`read_file` to `end_read`	`popq %r12, ret`	yes
anything else	error	—

low-overhead try/catch (2)

```
main:
   ...
   call printf
start_try:
   call read_file
end_try:
   ret
```

```
main_catch:
   movq $str, %rdi
   call printf
   jmp end_try
```

```
read_file:
   pushq %r12
   ...
   call do_throw
   ...
end_read:
   popq %r12
   ret
```

lookup table

program counter range	**action**	**recurse?**
`start_try` to `end_try`	`jmp main_catch`	no
`read_file` to `end_read`	`popq %r12, ret`	yes
anything else	error	—

low-overhead try/catch (2)

```
main:
   ...
   call printf
start_try:
   call read_file
end_try:
   ret
```

```
main_catch:
   movq $str, %rdi
   call printf
   jmp end_try
```

```
read_file:
   pushq %r12
   ...
   call do_throw
   ...
```

not actual x86 code to run
track a “virtual PC” while looking for catch block

lookup table

program counter range	**action**	**recurse?**
`start_try` to `end_try`	`jmp main_catch`	no
`read_file` to `end_read`	`popq %r12, ret`	yes
anything else	error	—

lookup table tradeoffs

no overhead if throw not used

handles local variables on registers/stack, but...

larger executables (probably)

extra complexity for compiler

protection and sudo

programs always run in user mode

extra permissions from OS do not change this

- sudo, superuser, root, SYSTEM, …

operating system may remember extra privileges

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

exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

top 16 bits of address not used for translation

exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: how many page table entries?

exercise: how large are physical page numbers?

exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: how many page table entries? $2^{48}/2^{12} = 2^{36}$ entries

exercise: how large are physical page numbers? $39 - 12 = 27$ bits

exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: how many page table entries? $2^{48}/2^{12} = 2^{36}$ entries

exercise: how large are physical page numbers? $39 - 12 = 27$ bits

page table entries are 8 bytes (room for expansion, metadata)

would take up 2^{39} bytes?? (512GB??)