last time

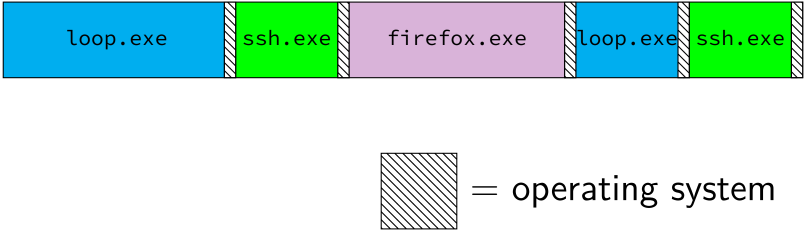
vector instructions / SIMD

profilers

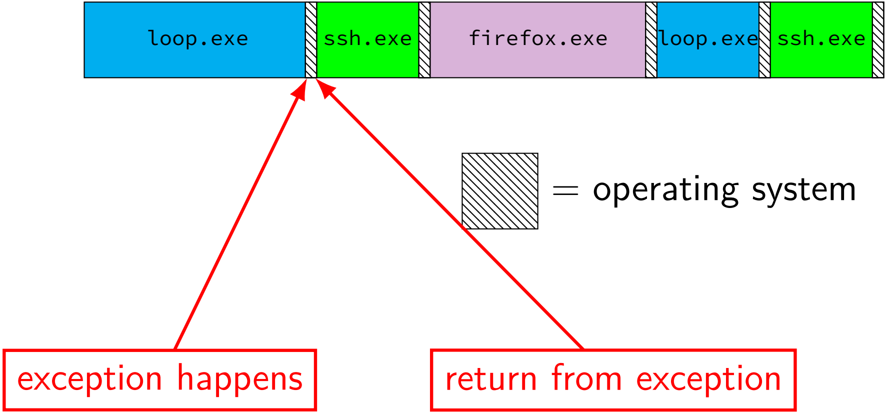
time multiplexing/context switching

address space ideaS

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

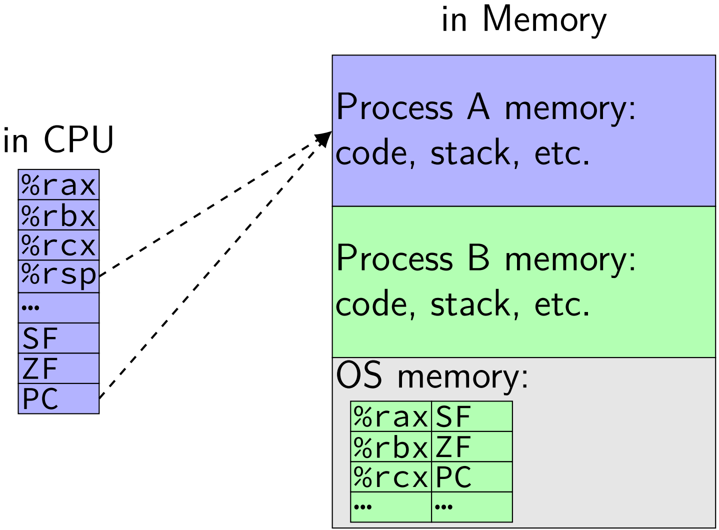
program counter

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

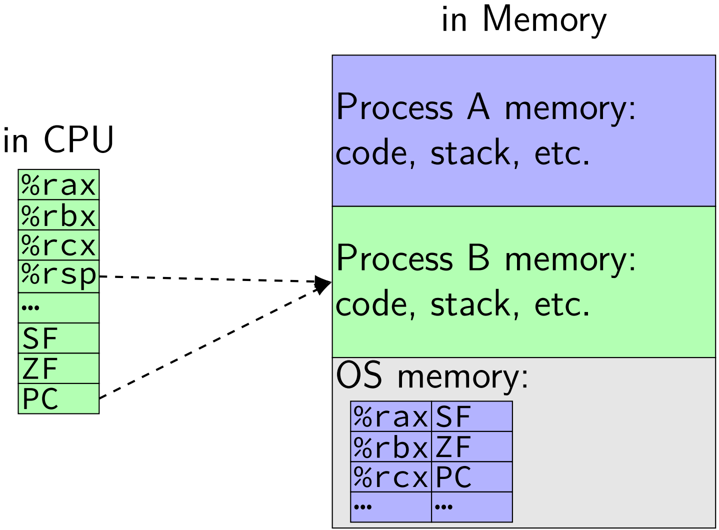
context switch pseudocode

```
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

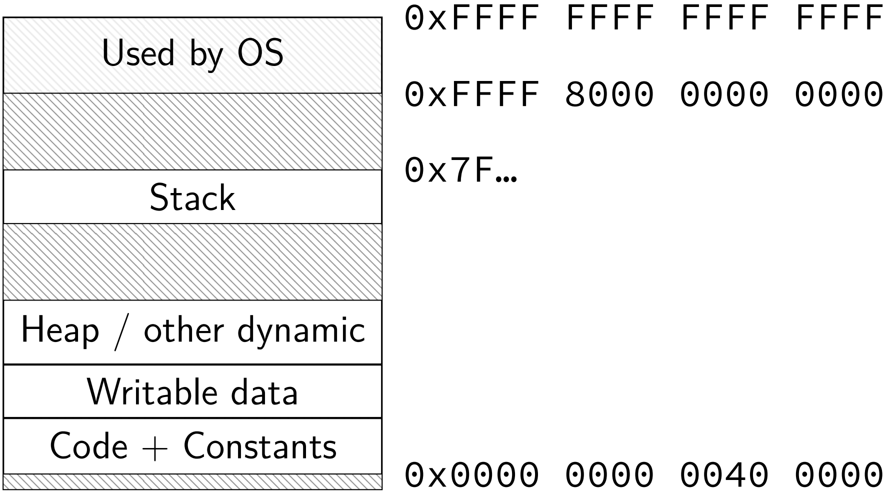
reading from another program's memory?

Program A	Program B

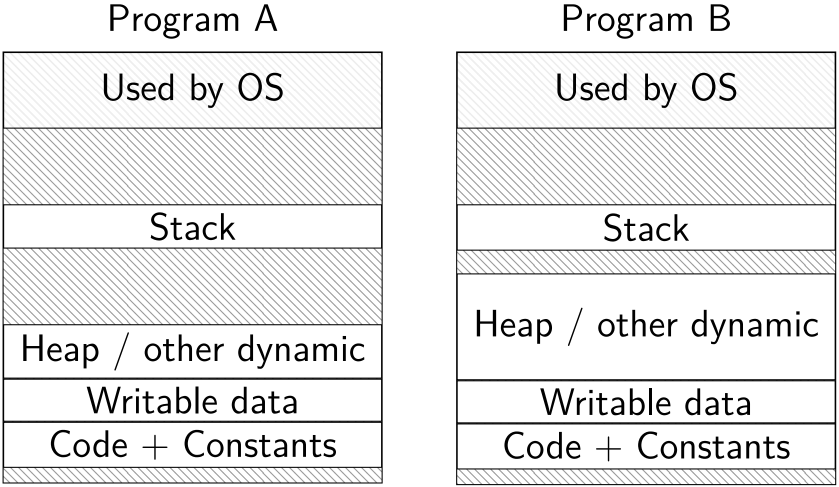
`0x10000: .word 42`
`    // ...`
`    // do work`
`    // ...`
`    movq 0x10000, %rax`
	`// while A is working:`
	`movq $99, %rax`
	`movq %rax, 0x10000`
	`...`

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

program memory



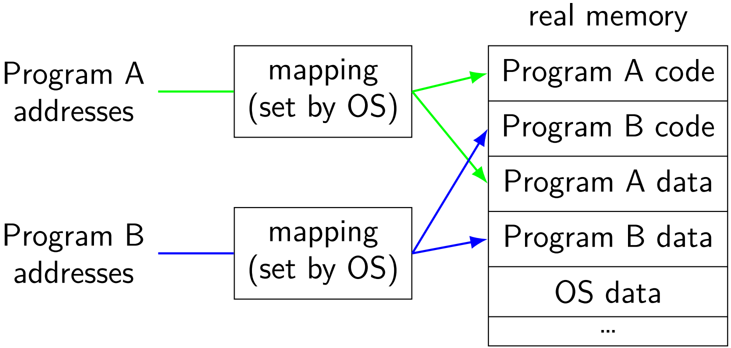
program memory (two programs)



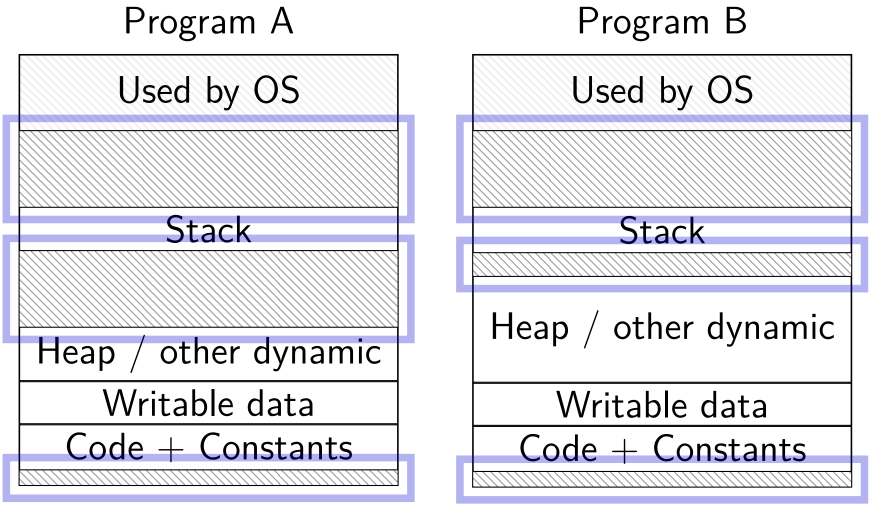
address space

programs have illusion of own memory

called a program's address space



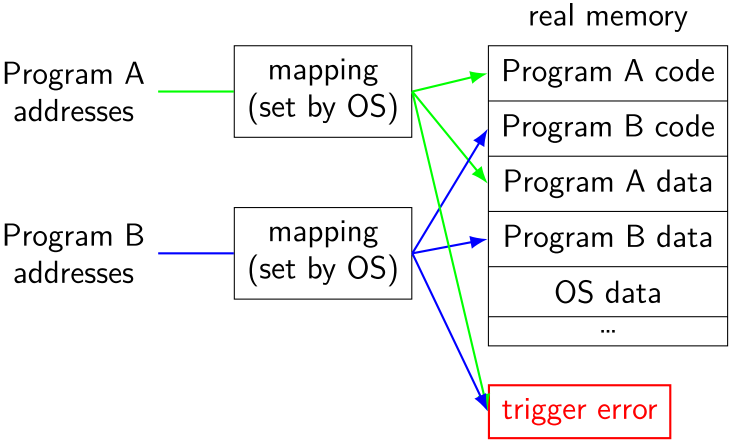
program memory (two programs)



address space

programs have illusion of own memory

called a program's address space



address space mechanisms

next 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

exceptions: OS gets control — two kinds of ways

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, ...

traps — intentionally triggered exceptions

- system calls — ask OS to do something

faults — errors/events in programs

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

aborts

types of exceptions

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

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

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

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, ...

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

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

aborts

types of exceptions

interrupts — externally-triggered

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

traps — intentionally triggered exceptions

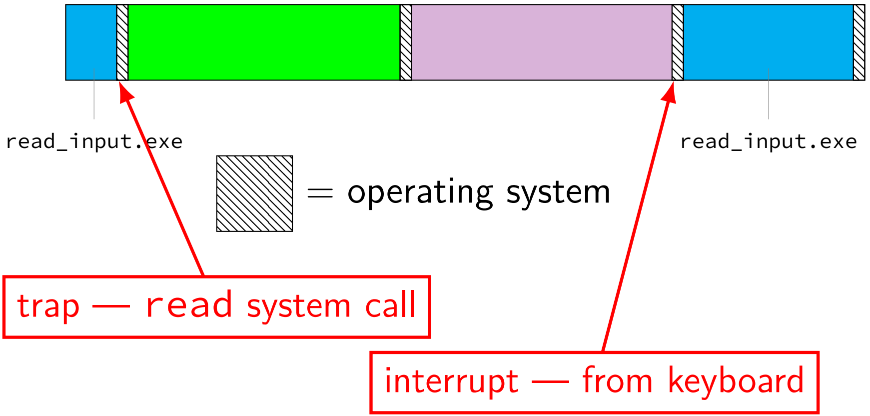
- system calls — ask OS to do something

faults — errors/events in programs

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

aborts

keyboard input timeline



types of exceptions

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

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

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

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

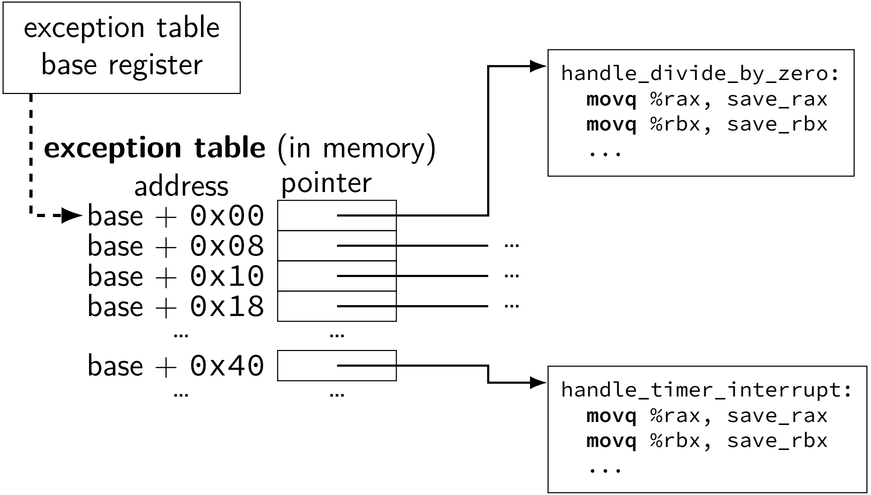
exception implementation: notes

I/textbook describe a simplified version

real x86/x86-64 is a bit more complicated

(mostly for historical reasons)

locating exception handlers



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

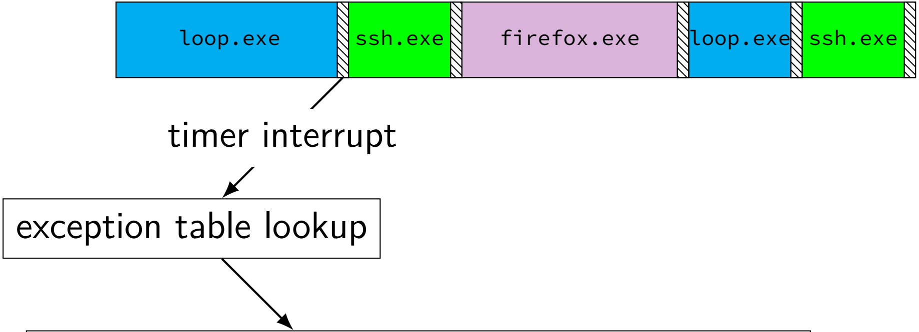
new instruction: return from exception
- i.e. jump to saved PC

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!)

types of exceptions

interrupts — externally-triggered

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

traps — intentionally triggered exceptions

- system calls — ask OS to do something

faults — errors/events in programs

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

aborts

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

types of exceptions

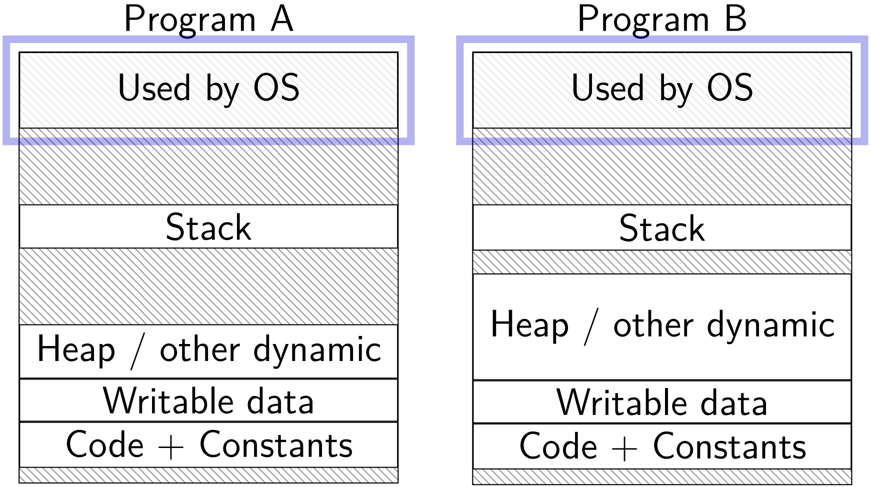
interrupts — externally-triggered
- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, …

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

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

aborts

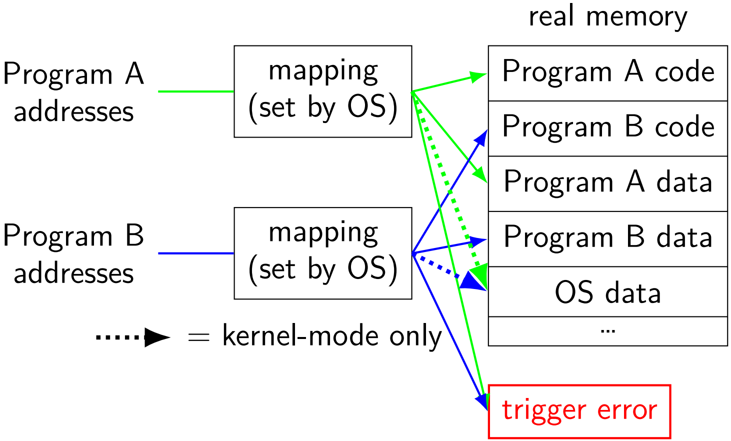
program memory (two programs)



address space

programs have illusion of own memory

called a program's address space



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, ...

traps — intentionally triggered exceptions

- system calls — ask OS to do something

faults — errors/events in programs

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

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

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”

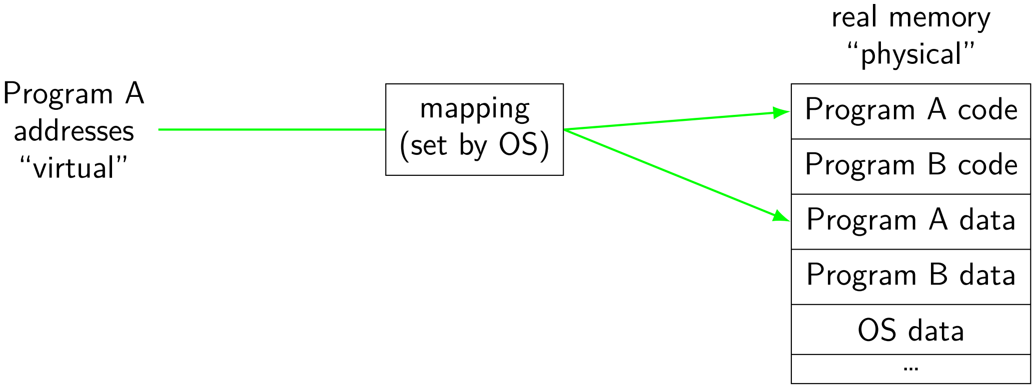
some additional terms:

- supervisor mode
- privileged mode
- ring `0`

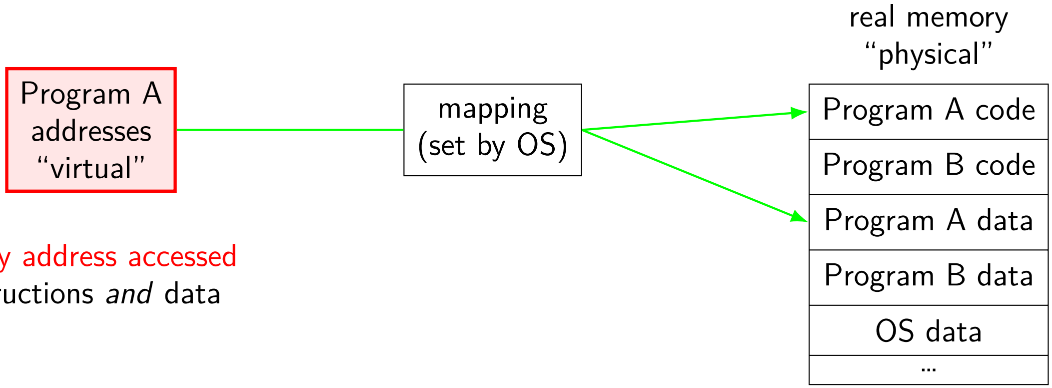
some systems have multiple levels of privilege

- different sets of priviliged operations work

address translation

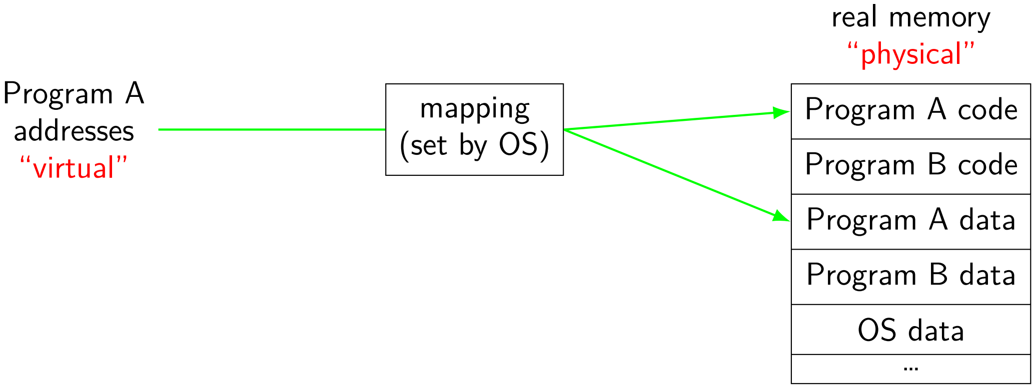


address translation



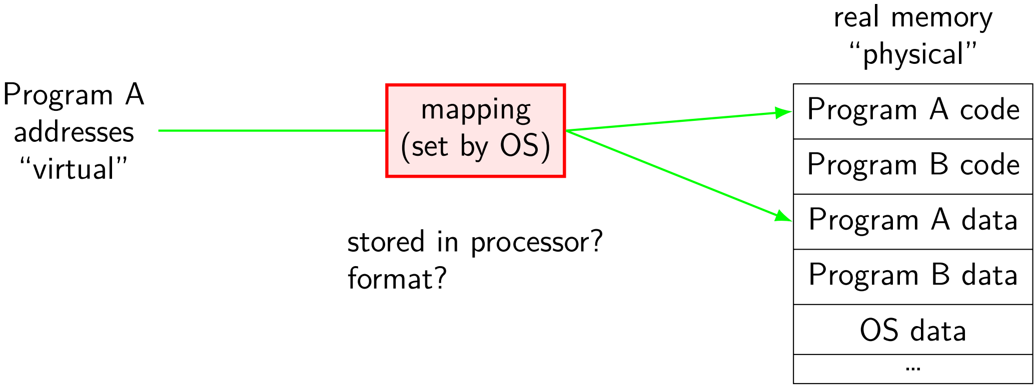
every address accessed
instructions *and* data

address translation

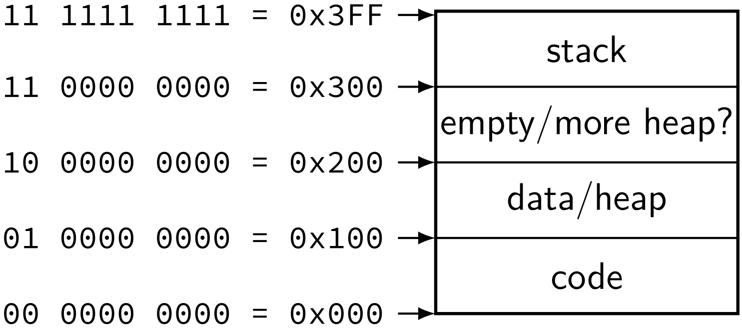


program addresses are 'virtual'
real addresses are 'physical'
can be different sizes!

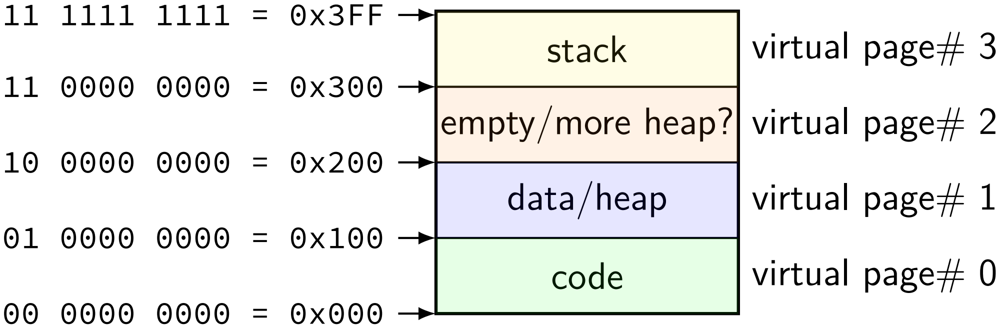
address translation



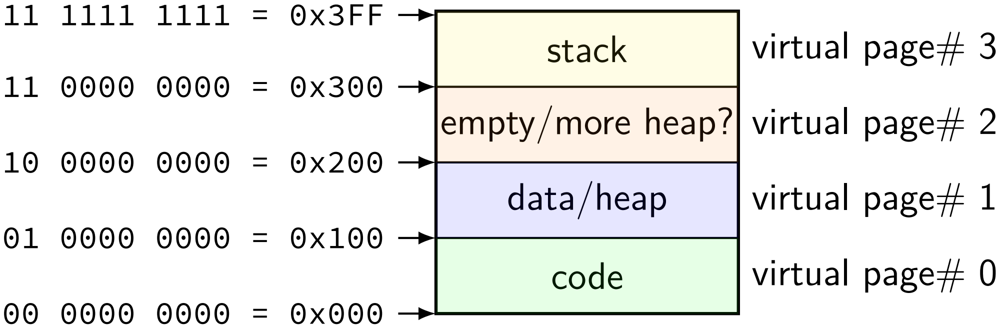
toy program memory



toy program memory

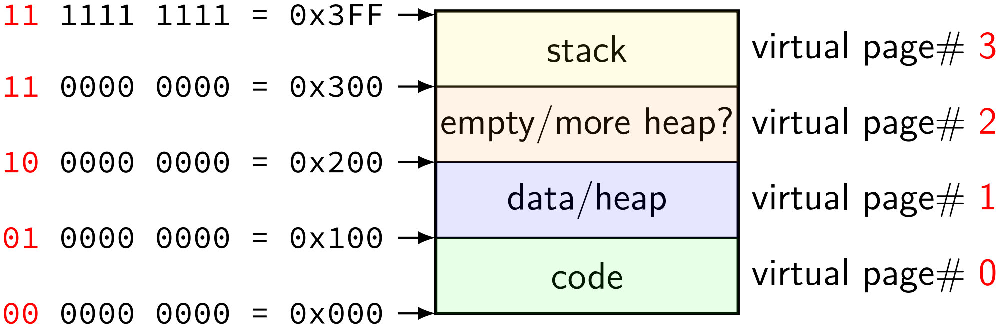


toy program memory



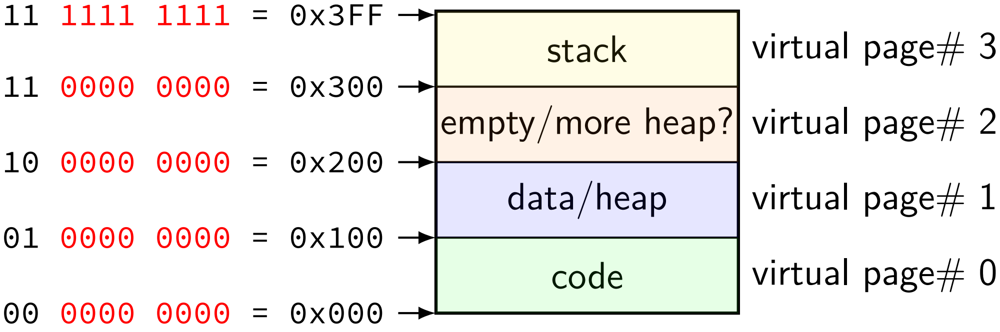
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

toy program memory



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

toy program memory



rest of address is called page offset

toy physical memory

real memory

physical addresses

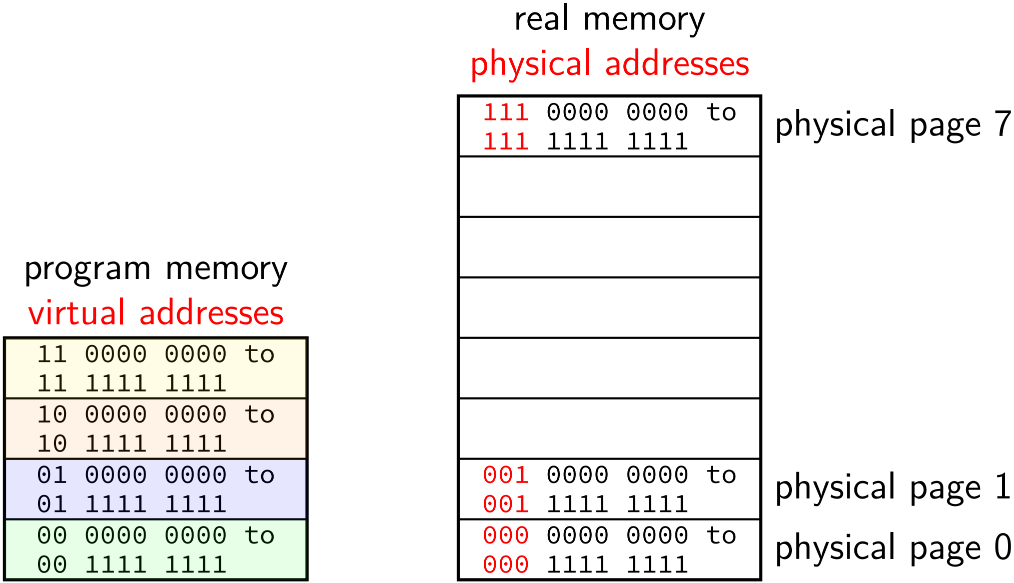
111 0000 0000 to 111 1111 1111
001 0000 0000 to 001 1111 1111
000 0000 0000 to 000 1111 1111

program memory

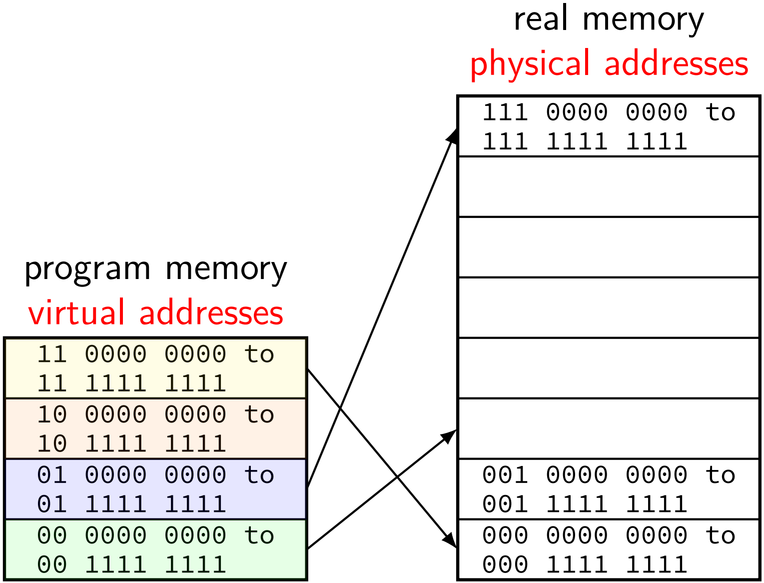
virtual addresses

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

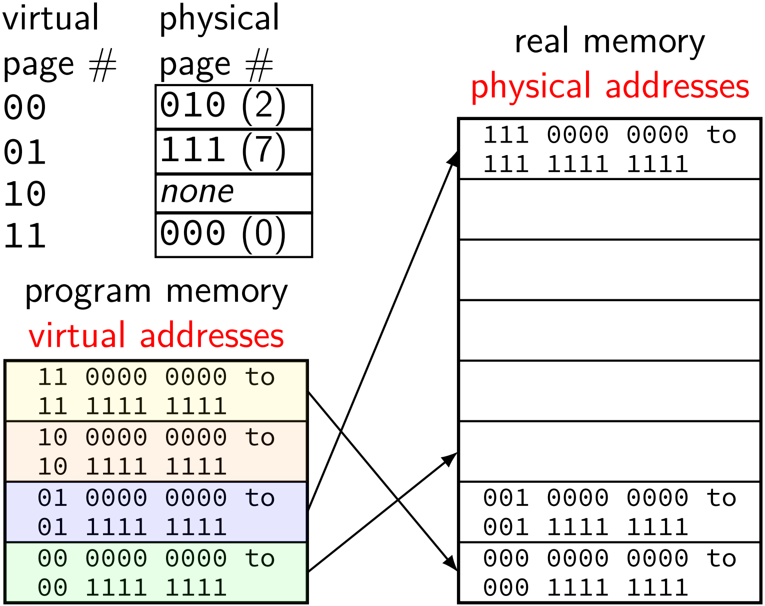
toy physical memory



toy physical memory



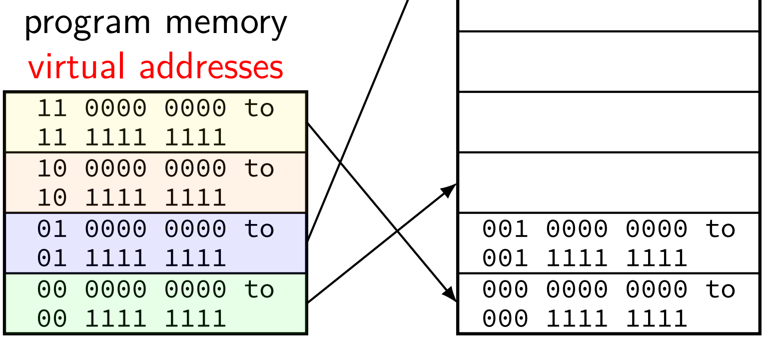
toy physical memory



toy physical memory

page table!

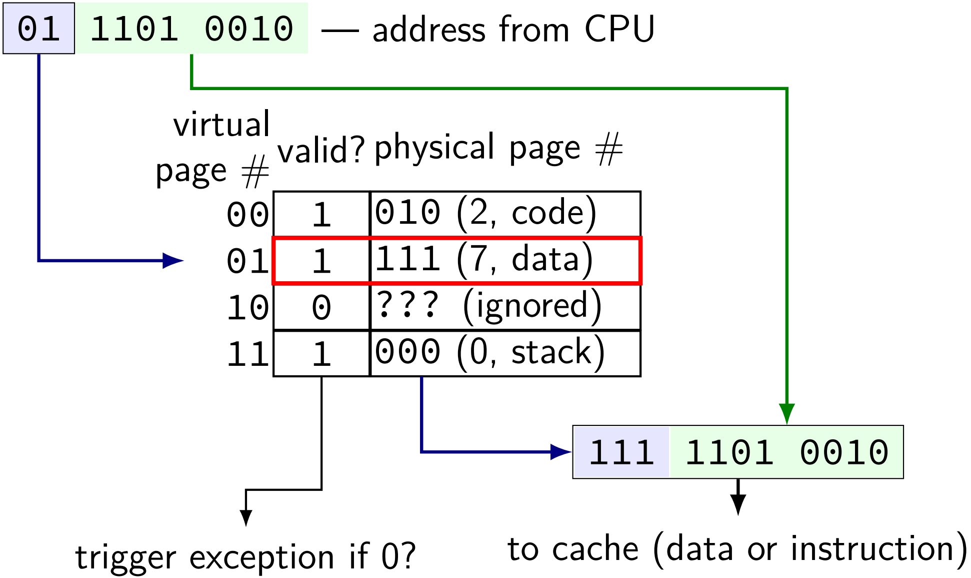
virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
11	000 (0)



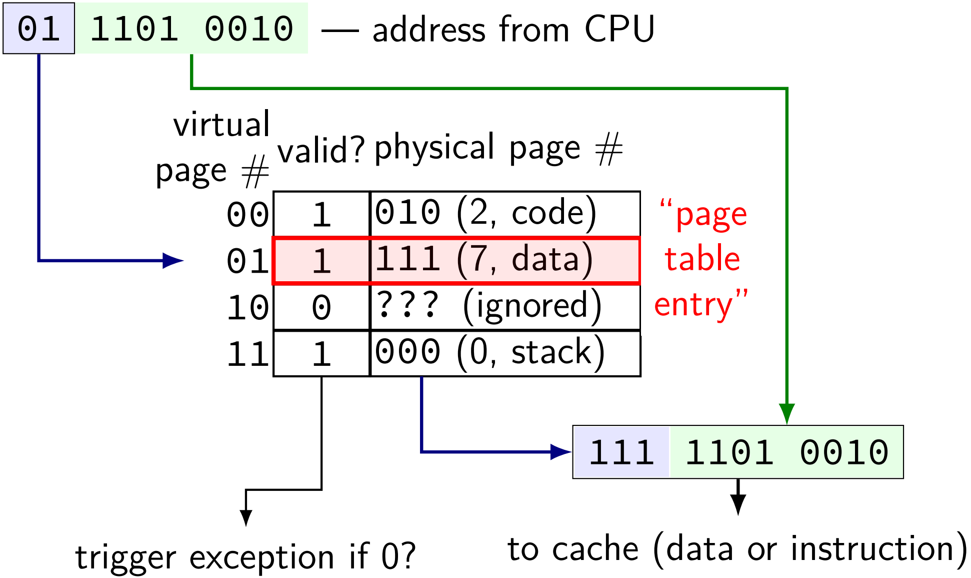
toy page table lookup

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

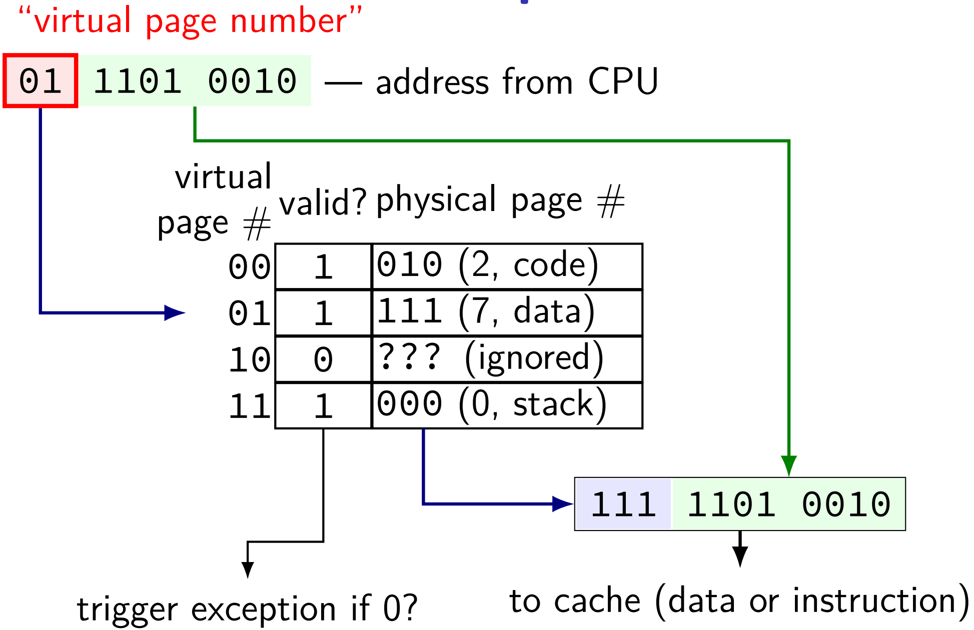
toy page table lookup



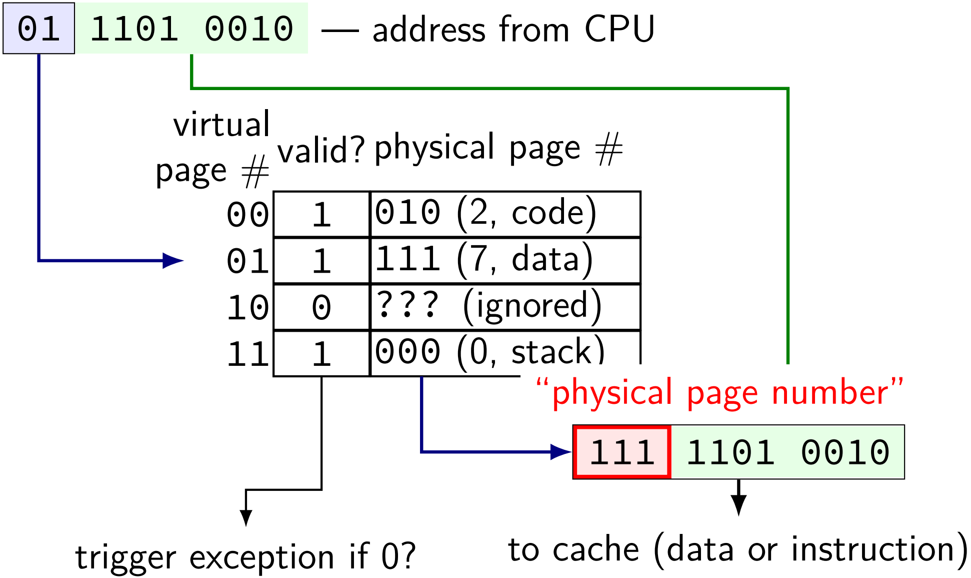
toy page table lookup



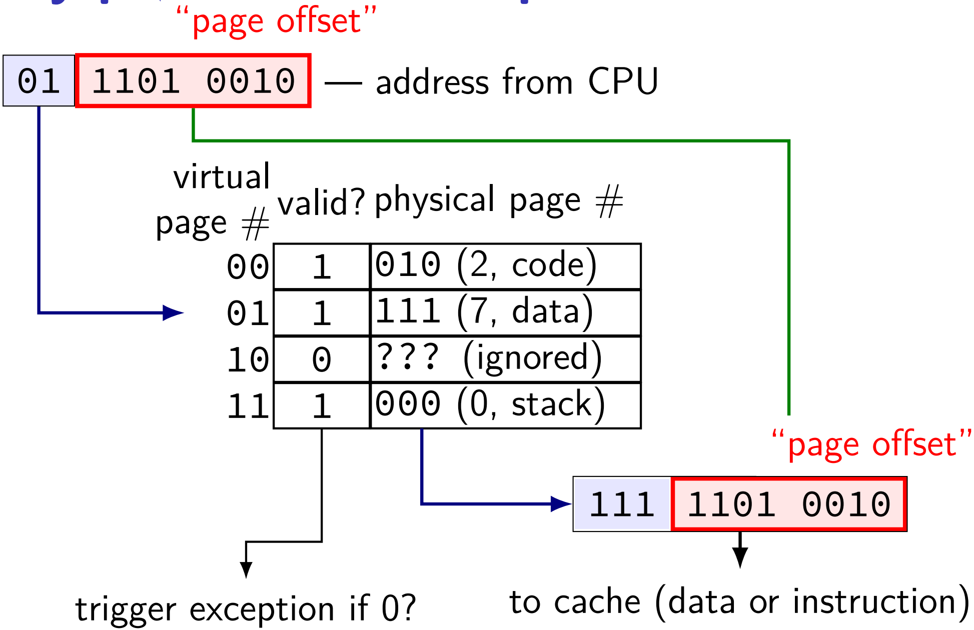
toy page table lookup



toy page table lookup



toy page table lookup



switching page tables

part of context switch is changing the page table

extra privileged instructions

switching page tables

part of context switch is changing the page table

extra privileged instructions

where in memory is the code that does this switching?

switching page tables

part of context switch is changing the page table

extra privileged instructions

where in memory is the code that does this switching?

- needs a page table entry pointing to it
- (alternate: HW changes page table when starting exception handler)

switching page tables

part of context switch is changing the page table

extra privileged instructions

where in memory is the code that does this switching?

- needs a page table entry pointing to it
- (alternate: HW changes page table when starting exception handler)

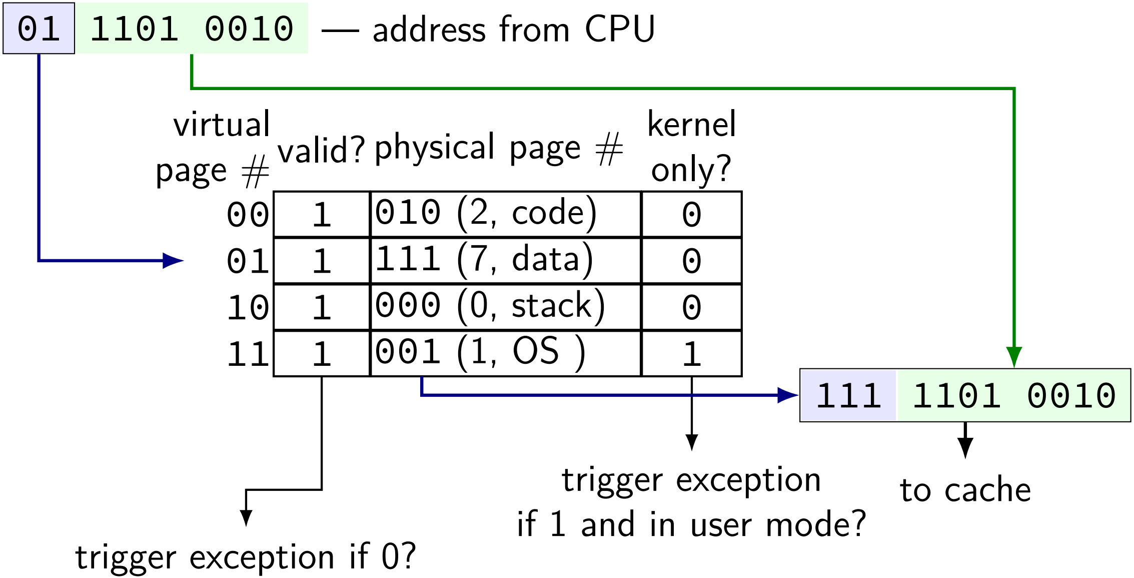
code better not be modified by user program

- otherwise: uncontrolled way to “escape” user mode

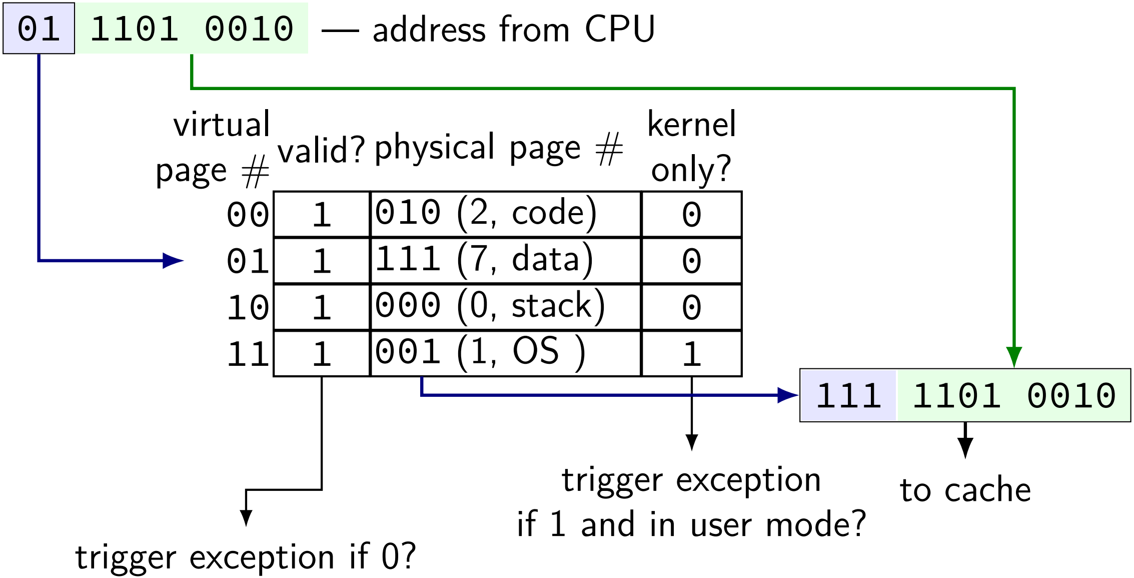
kernel-mode only

virtual page #	valid?	physical page #	kernel only?
00	1	010 (2, code)	0
01	1	111 (7, data)	0
10	1	000 (0, stack)	0
11	1	001 (1, OS)	1

kernel-mode only

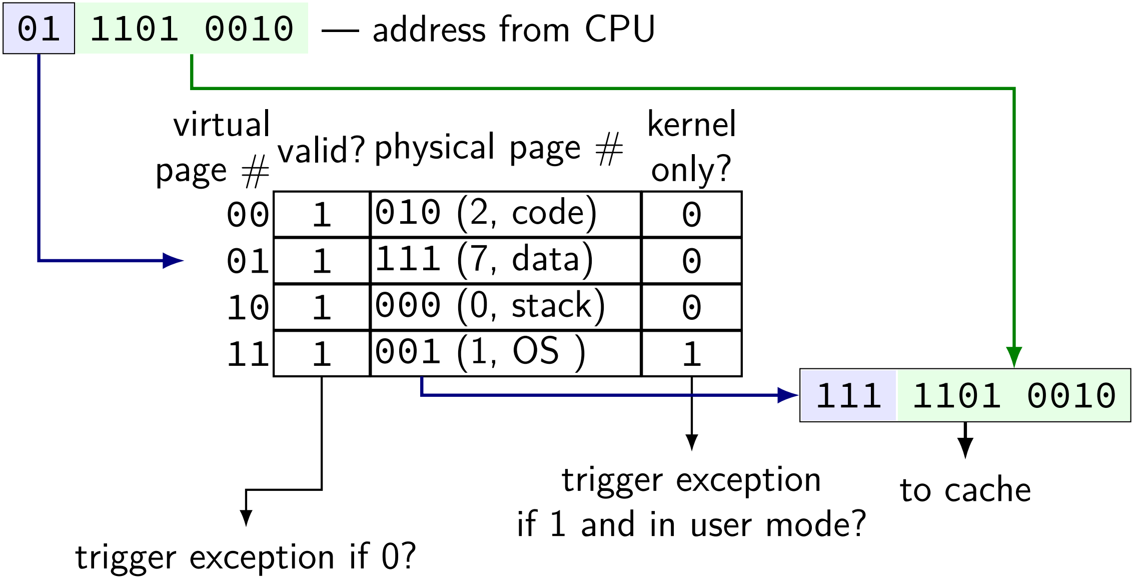


kernel-mode only

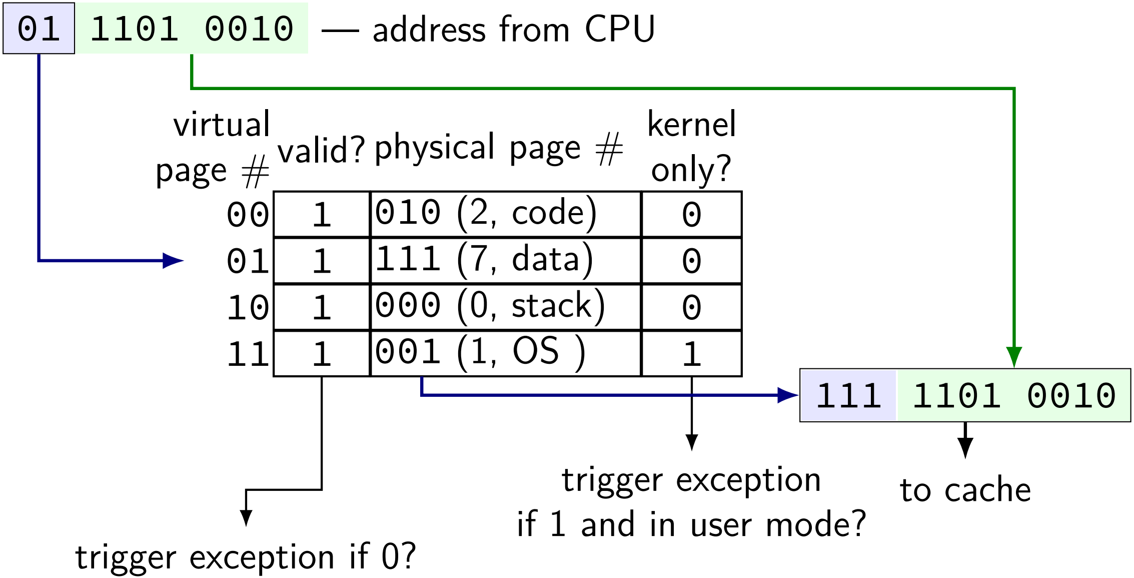


virtual page #	valid?	physical page #	kernel only?
00	1	010 (2, code)	0
01	1	111 (7, data)	0
10	1	000 (0, stack)	0
11	1	001 (1, OS)	1

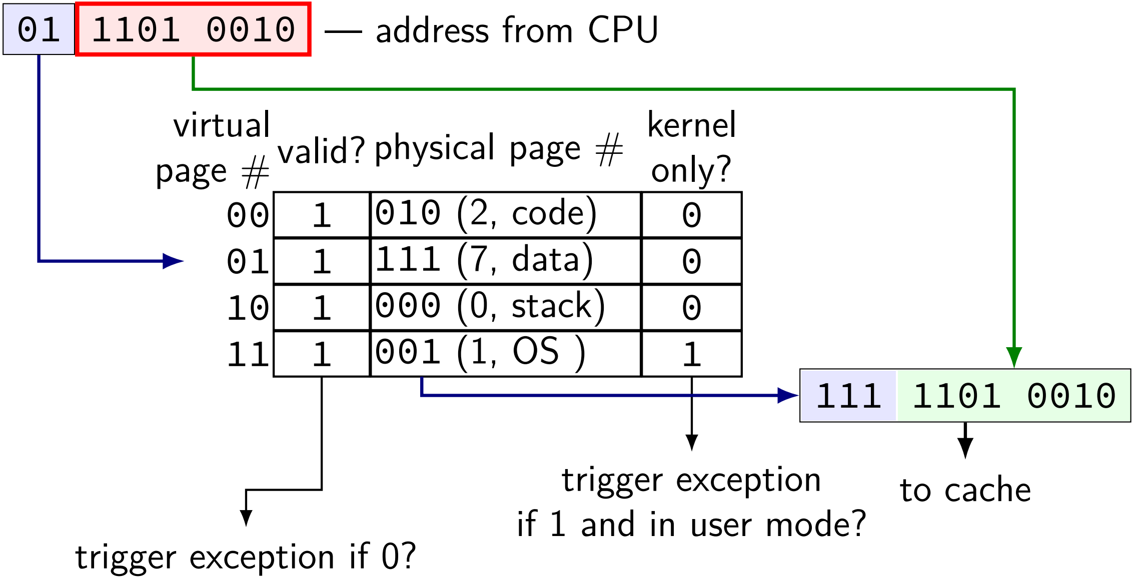
kernel-mode only



kernel-mode only



kernel-mode only



on virtual address sizes

virtual address size = size of pointer?

often, but — sometimes part of pointer not used

example: typical x86-64 only use 48 bits

- rest of bits have fixed value

virtual address size is amount used for mapping

address space sizes

amount of stuff that can be addressed $=$ address space size

based on number of unique addresses

e.g. 32-bit virtual address $= 2^{32}$ byte virtual address space

e.g. 20-bit physical addresss $= 2^{20}$ byte physical address space

address space sizes

amount of stuff that can be addressed = address space size

- based on number of unique addresses

e.g. 32-bit virtual address = 2^{32} byte virtual address space

e.g. 20-bit physical addresss = 2^{20} byte physical address space

what if my machine has 3GB of memory (not power of two)?

- not all addresses in physical address space are useful
- most common situation (since CPUs support having a lot of memory)

time multiplexing

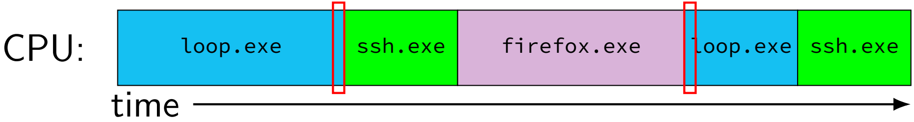
time multiplexing


```
...
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
...
```

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!

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

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