exceptions

Changelog

22 Nov 2022: add back protection and sudo slide to match what was gone over verbally in lecture better

22 Nov 2022: move kernel-mode-only slides to match lecture order better

22 Nov 2022: add solution slide for what-needs-kernel-mode slide

23 Nov 2022: add solution slide for which needs exception/context switch slide

last time

SIMD (single instruction, multiple data)
  - registers containing vectors (fixed sized arrays)
  - instructions typically act on every pairs of values
  - typical implementation: ALUs with many 'lanes'

writing SIMD code
  - expand code to have groups of "parallel" operations
  - sometimes want to broadcast value across vector
  - sometimes need to rearrange vectors

last time (2)

time multiplexing

- two programs running on one core
- they take turns
- hardware help to run OS to change programs

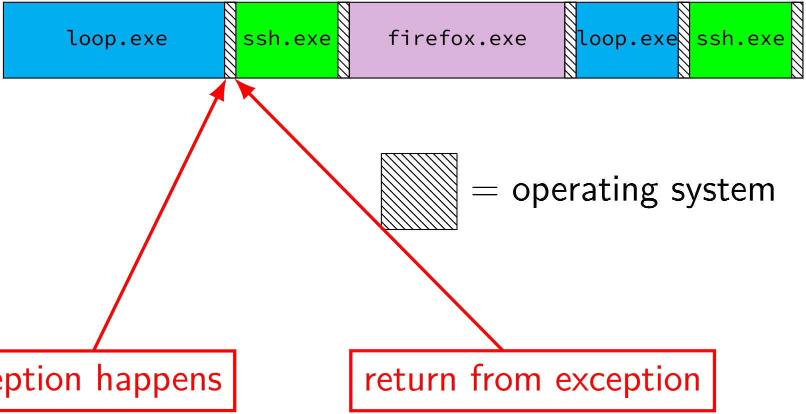
context switches and address spaces

- context = registers + other info “on CPU” about current program
- context switch = save one program’s stuff, restore anothers
- address space = memory accessible to program
- mapping set/changed by OS in context switch

time multiplexing really

▨ = operating system

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

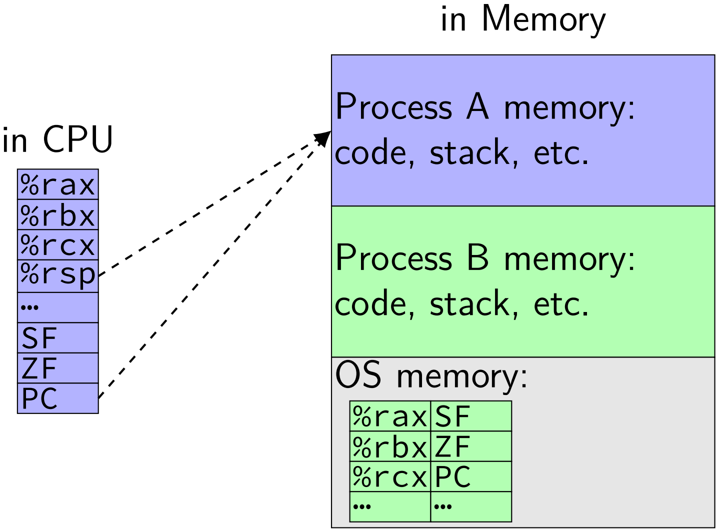
context

- 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 CPU

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

in Memory

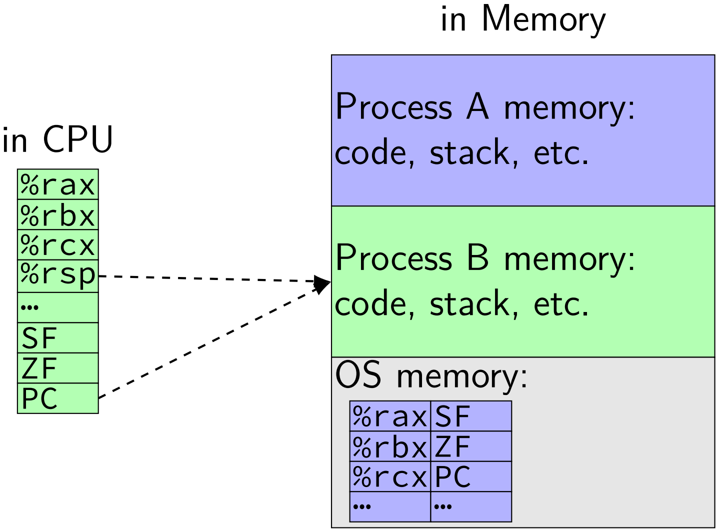
Process A memory:
code, stack, etc.

Process B memory:
code, stack, etc.

OS memory:

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

contexts (B running)



in Memory

Process A memory:
code, stack, etc.

Process B memory:
code, stack, etc.

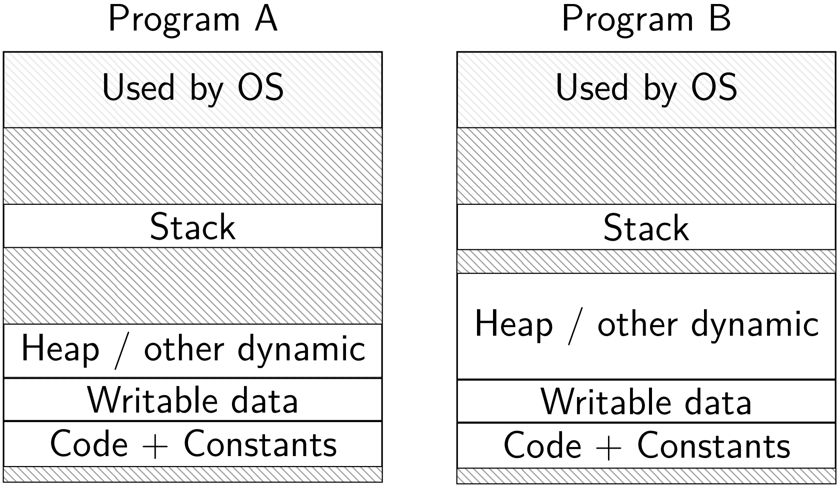
OS memory:

%rax	SF
%rbx	ZF
%rcx	PC
…	…

in CPU

- %rax
- %rbx
- %rcx
- %rsp
- …
- SF
- ZF
- PC

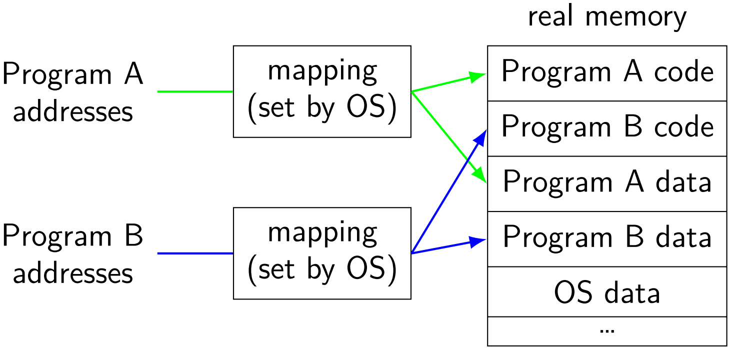
program memory (two programs)



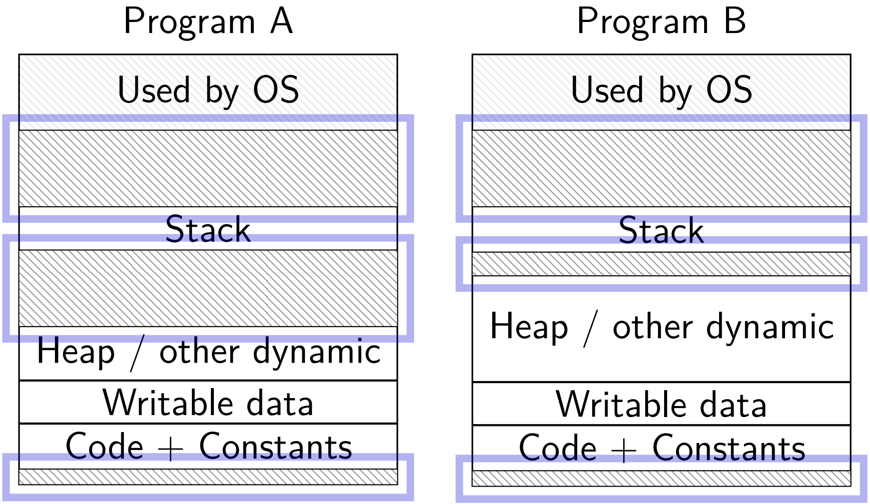
address space

programs have illusion of own memory

called a program's address space



program memory (two programs)

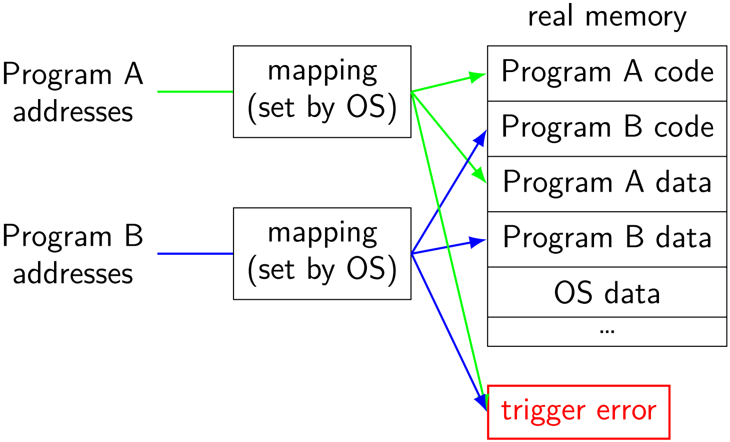

Program A
Used by OS
Stack
Heap / other dynamic
Writable data
Code + Constants

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

topic after exceptions

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

types of exceptions

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

asynchronous
not triggered by
running program

- 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

synchronous
triggered by
current program

types of exceptions

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

aborts — hardware is broken

asynchronous
not triggered by
running program

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

synchronous
triggered by
current program

types of exceptions

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

aborts — hardware is broken

asynchronous
not triggered by running program

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

synchronous
triggered by current program

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, …

aborts — hardware is broken

asynchronous

not triggered by running program

traps — intentionally triggered exceptions

- system calls — ask OS to do something

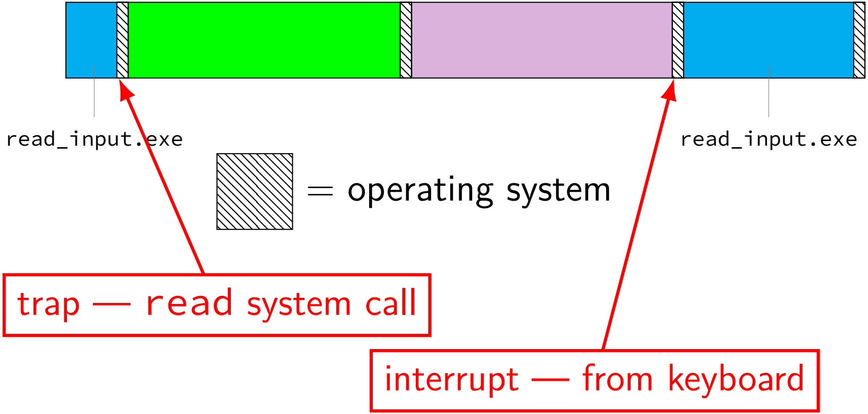
faults — errors/events in programs

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

synchronous

triggered by current program

keyboard input timeline



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, …

aborts — hardware is broken

asynchronous
not triggered by
running program

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

synchronous
triggered by
current program

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

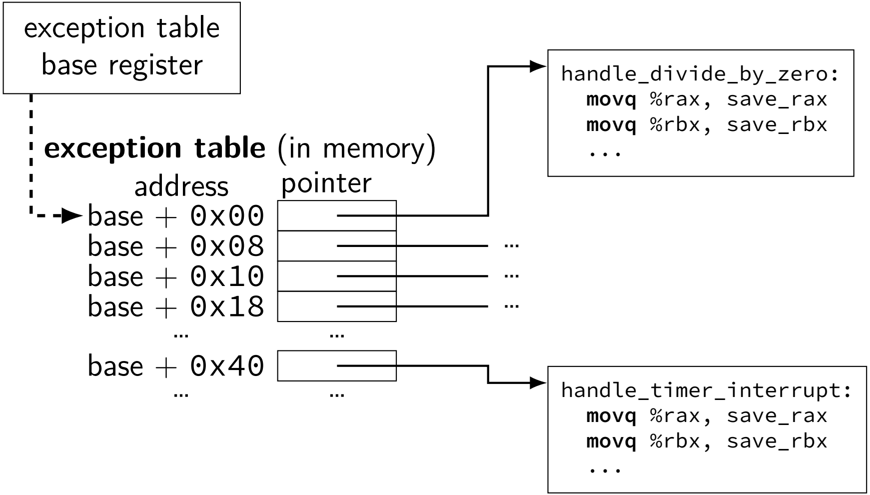
exception implementation: notes

I/textbook describe a simplified version

real x86/x86-64 is a bit more complicated

(mostly for historical reasons)

locating exception handlers



exception table base register

exception table (in memory)

address	pointer
base + `0x00`	
base + `0x08`	
base + `0x10`	
base + `0x18`	
…	…
base + `0x40`	
…	…

```
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

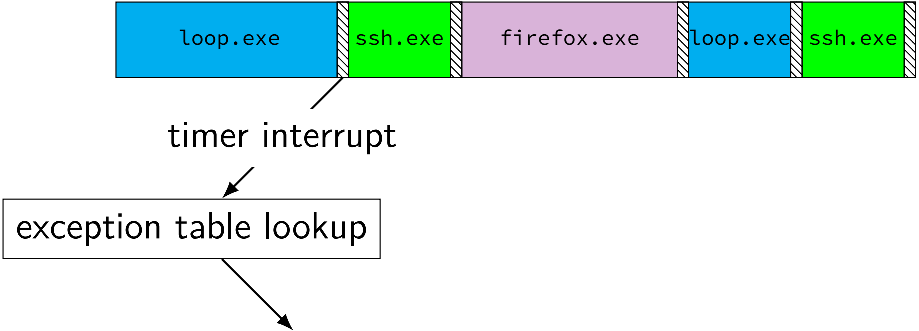
OS code can save anything else it wants to , etc.

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



timer interrupt

exception table lookup

```
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, …

aborts — hardware is broken

asynchronous

not triggered by running program

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

synchronous

triggered by current program

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

editing exception table?

why can't we edit exception table/exception handlers?

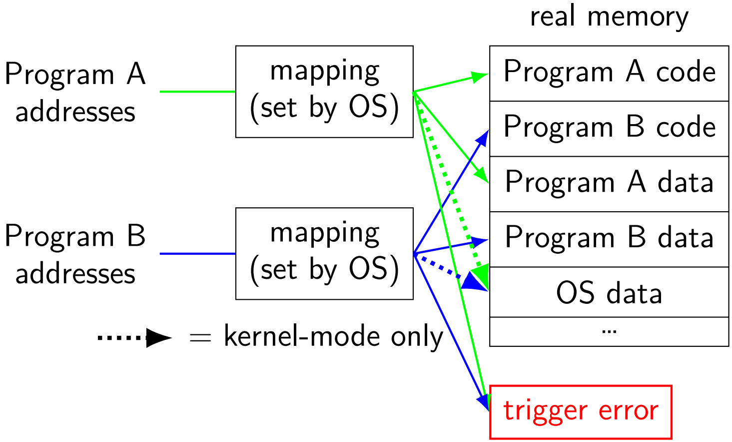
on many processors,
they have to be accessible in memory while processes are running

many OSes: in OS-only region of memory (usually high addresses)
- often same in every process

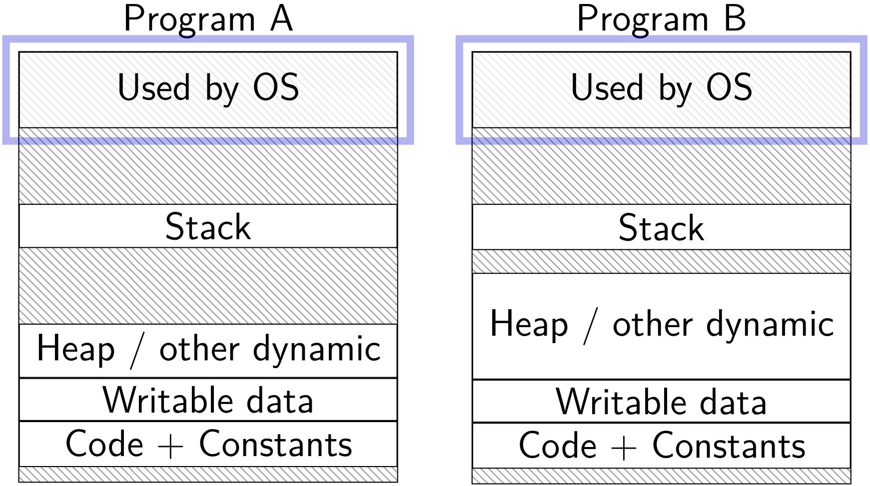
address space

programs have illusion of own memory

called a program's address space



program memory (two programs)



which requires kernel mode?

which operations are likely to fail (trigger an exception to run the OS instead) if attempted in user mode?

A. reading data on disk by running special instructions that communicate with the hard disk device

B. changing a program's address space to allocate it more memory

C. returning from a standard library function

D. incrementing the stack pointer

which requires kernel mode? [answers] (1)

A. reading data on disk by running special instructions that communicate with the hard disk device

- yes: generally I/O is reserved for OS
- yes: otherwise programs could read/write files they aren't allowed to (e.g. on shared system like portal) unless some way to restrict what's sent to/from hard disk

B. changing a program's address space to allocate it more memory

- yes: changing address space; have to restrict how that's done to prevent program from accessing other program/user's memory or messing up OS memory

which requires kernel mode? [answers] (2)

C. returning from a standard library function

no: some standard libraries may do things that require system calls, but not all; and the actual returning part should be in user mode since it's just like a normal function

D. incrementing the stack pointer

no: just changing a pointer value; changes what memory we consider allocated/deallocated, but actual changes to the mapping set by the OS would have to be triggered by some other event

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

system call pattern

basically a function call, but…

calling convention for arguments, return value

special instruction to trigger excpetion
- can't specify address of function to call, so…
- typically set *system call number* in register
- e.g. on x86-64 Linux, %rax = 0 for read, 1 for write, 2 for open, …

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

can't write C code to generate syscall instruction

solution: call "wrapper" function written in assembly

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

- allow more system calls than usual

which of these require exceptions? context switches?

A. program calls a function in the standard library

B. program writes a file to disk

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

D. program exits

E. program returns from one function to another function

F. program pops a value from the stack

which require exceptions [answers] (1)

A. program calls a function in the standard library

no (same as other functions in program; some standard library functions might make system calls, but if so, that'll be part of what happens after they're called and before they return)

B. program writes a file to disk

yes (requires kernel mode only operations)

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

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

which require exceptions [answer] (2)

D. program exits

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

E. program returns from one function to another function

no

F. program pops a value from the stack

no

which require context switches [answer]

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

no: B. program writes a file to disk

(but might be done if program needs to wait for disk and other things could be run while it does)

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

yes: D. program exits

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

no: F. program pops a value from the stack

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"

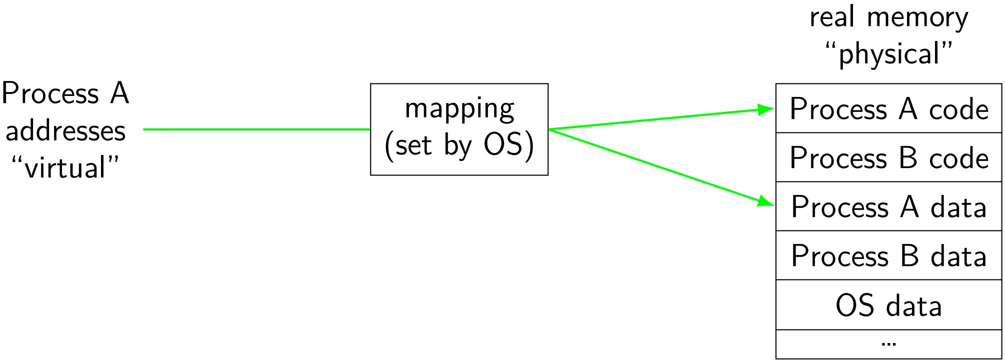
some additional terms:

- supervisor mode
- privileged mode
- ring `0`

some systems have multiple levels of privilege

- different sets of priviliged operations work

address translation



real memory
“physical”

Process A
addresses
“virtual”

mapping
(set by OS)

Process A code

Process B code

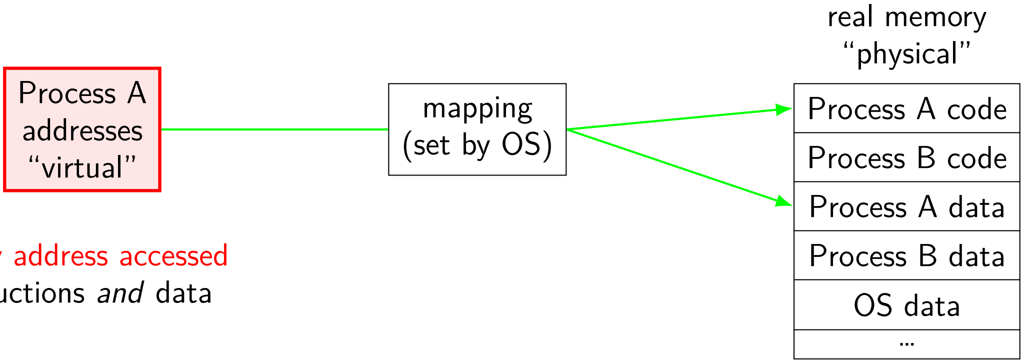
Process A data

Process B data

OS data

...

address translation



real memory
"physical"

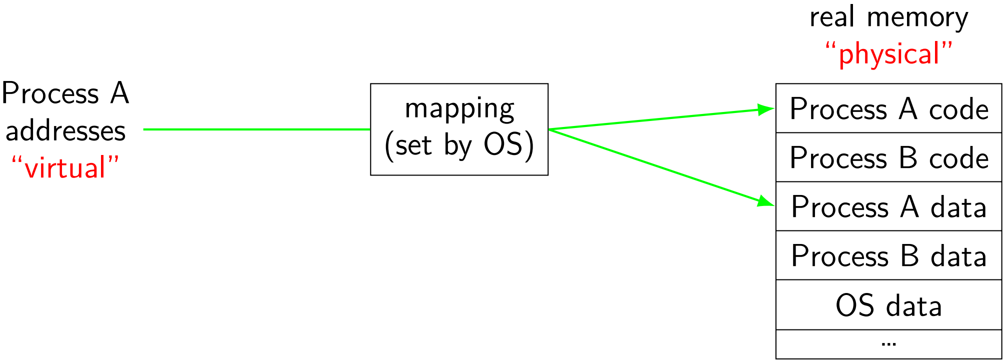
Process A
addresses
"virtual"

mapping
(set by OS)

Process A code
Process B code
Process A data
Process B data
OS data
...

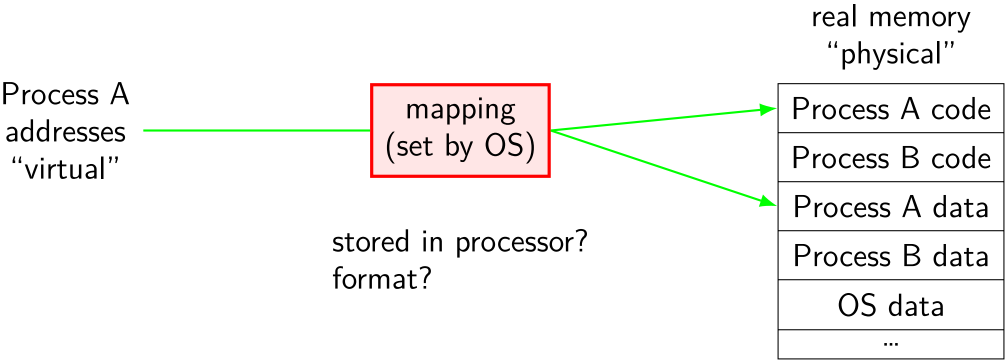
every address accessed
instructions *and* data

address translation



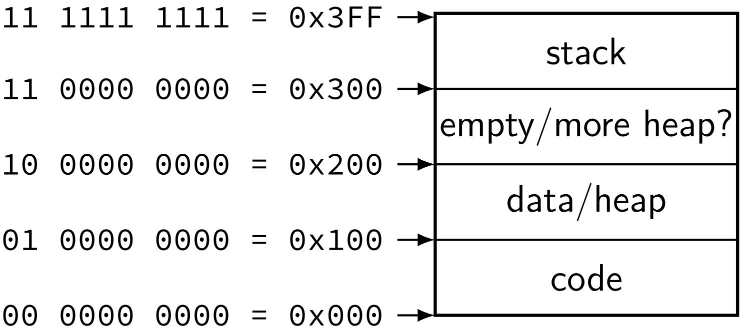
program addresses are 'virtual'
real addresses are 'physical'
can be different sizes!

address translation

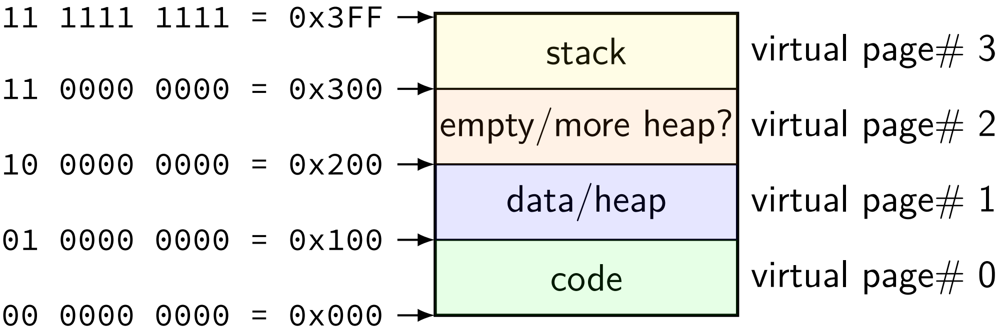


stored in processor?
format?

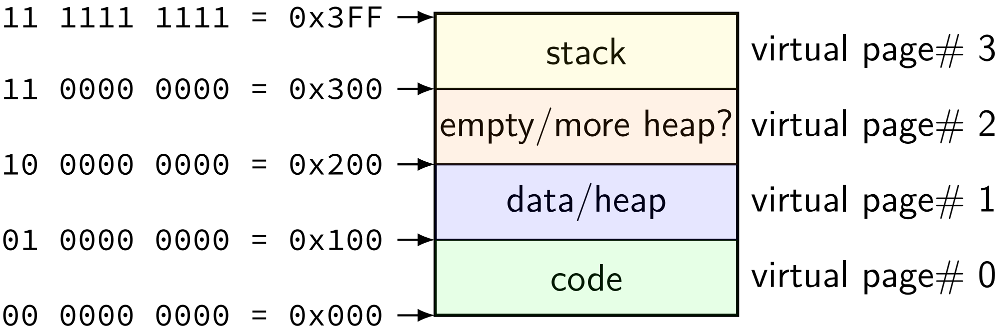
toy program memory



toy program memory

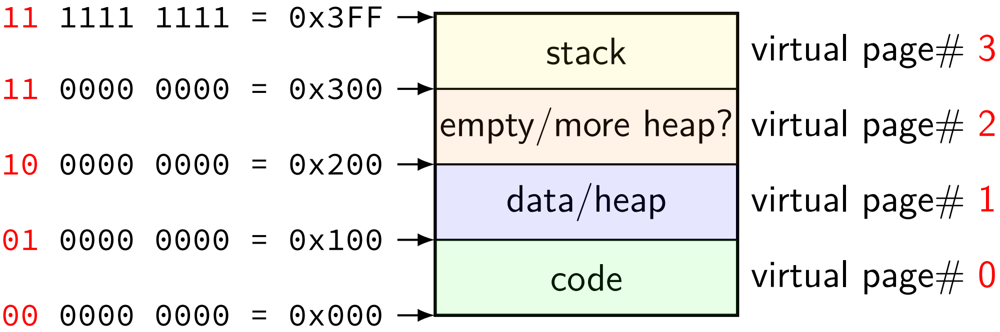


toy program memory



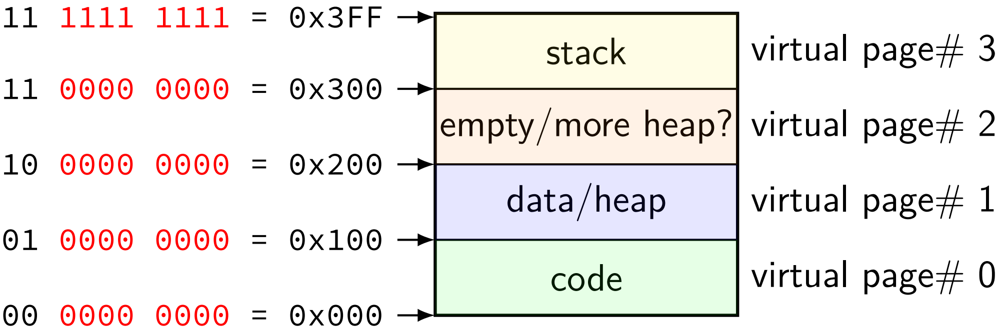
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

toy program memory



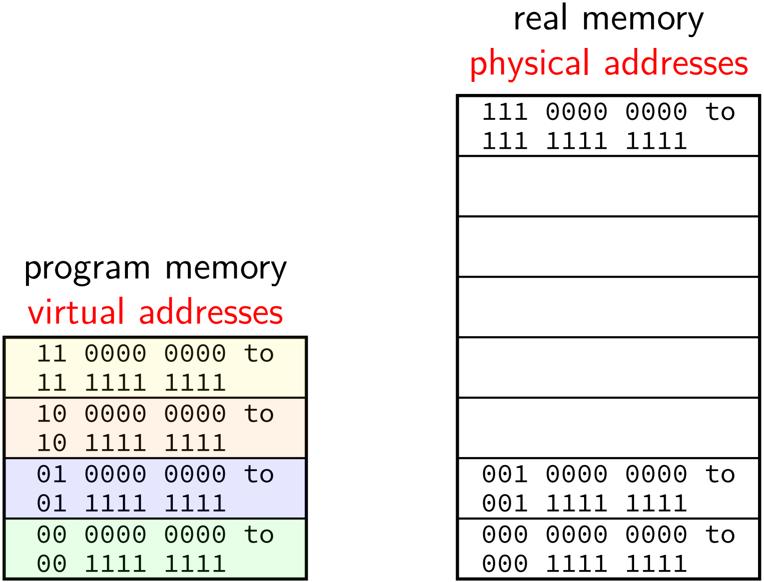
page number is upper bits of address
(because page size is power of two)

toy program memory



rest of address is called page offset

toy physical memory



program memory

virtual addresses

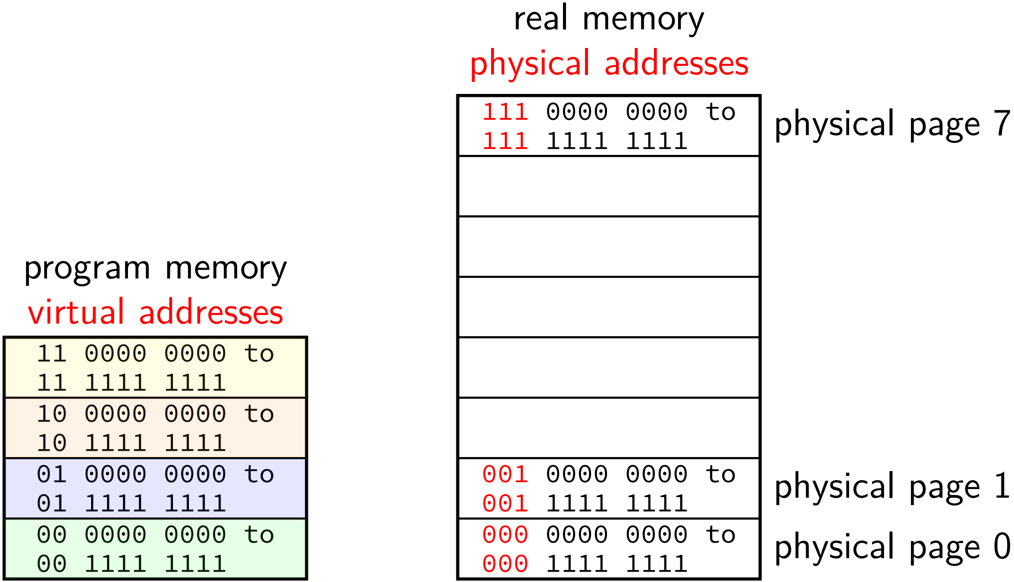
```
11 0000 0000 to
11 1111 1111
10 0000 0000 to
10 1111 1111
01 0000 0000 to
01 1111 1111
00 0000 0000 to
00 1111 1111
```

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



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

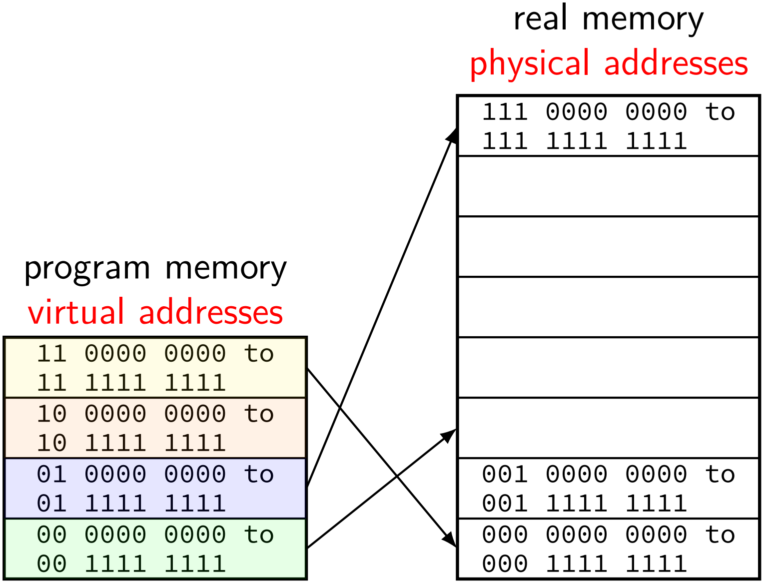
11 0000 0000 to
11 1111 1111

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

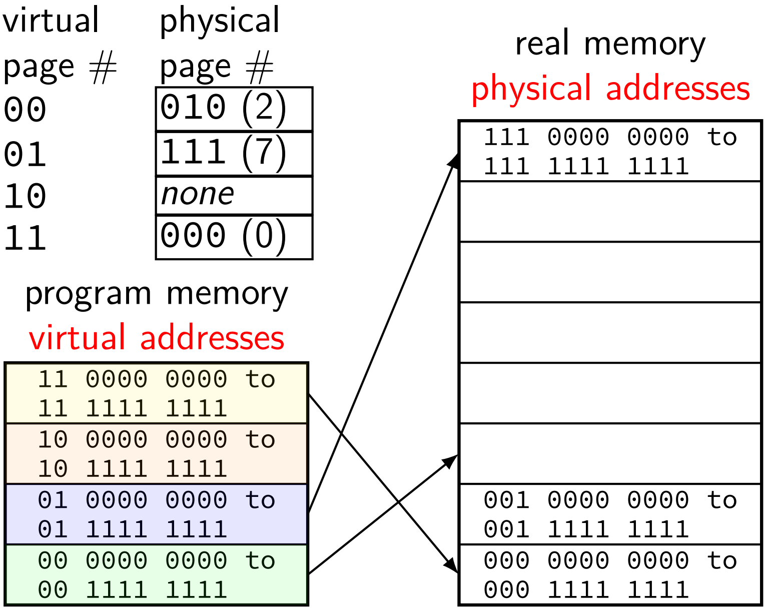
11 0000 0000 to
11 1111 1111

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

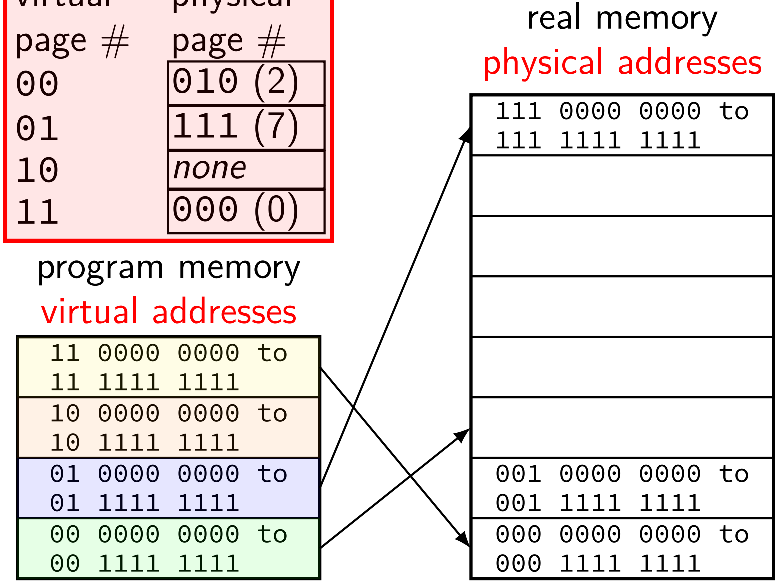
physical addresses

```
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

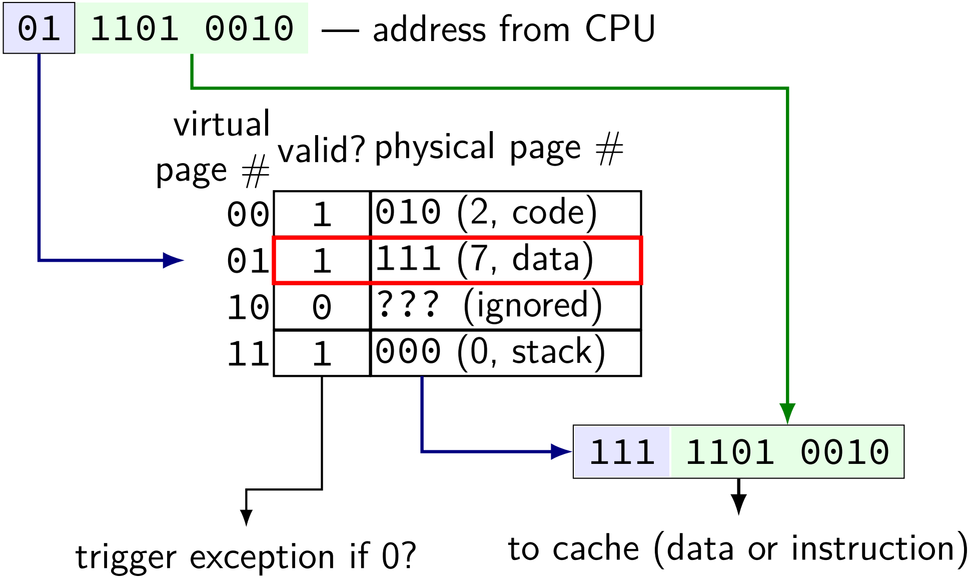
program memory

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

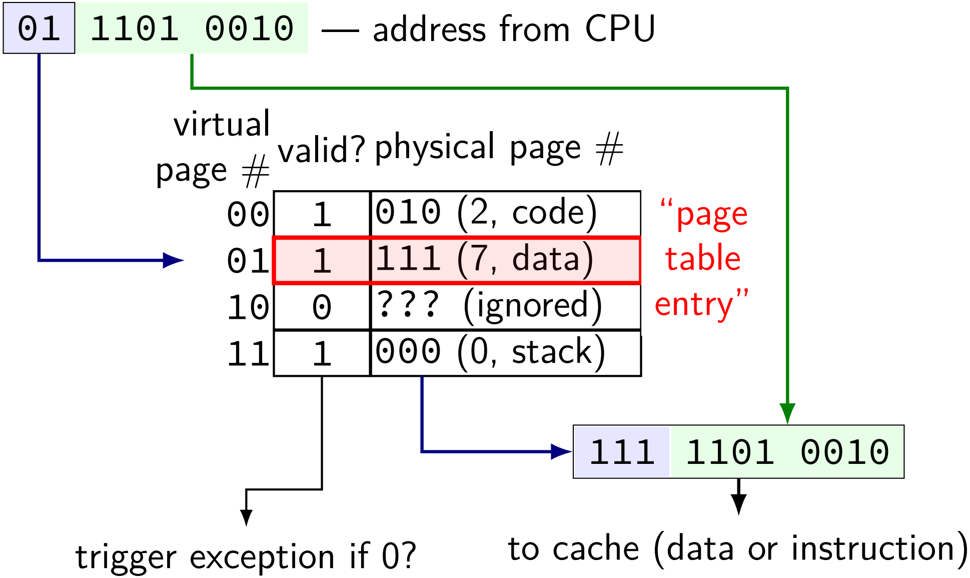
virtual page #	valid?	physical page #
`00`	`1`	`010` (2, code)
`01`	`1`	`111` (7, data)
`10`	`0`	`???` (ignored)
`11`	`1`	`000` (0, stack)

`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)

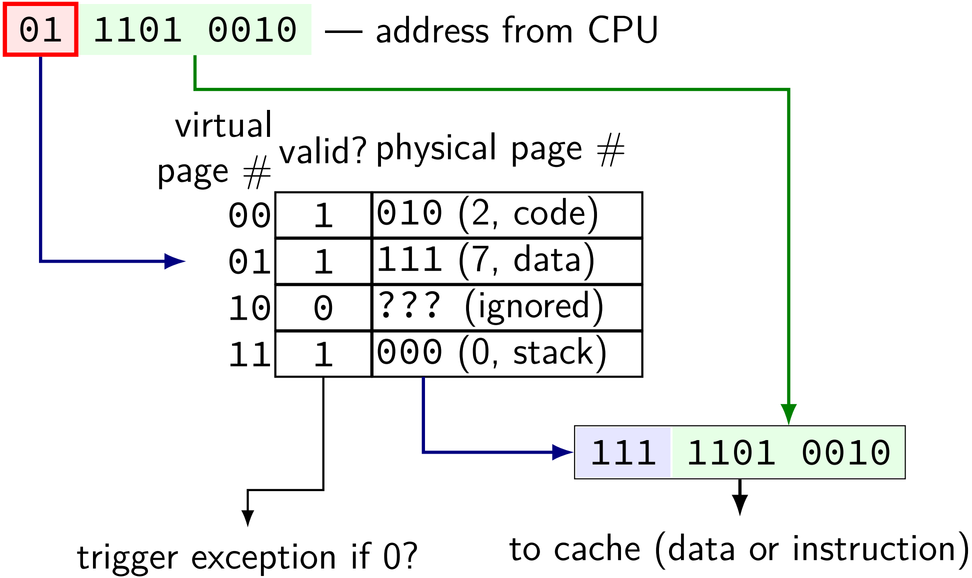
“page table entry”

`111` `1101 0010`

trigger exception if 0?

to cache (data or instruction)

t “virtual page number” lookup



`01` `1101 0010` — address from CPU

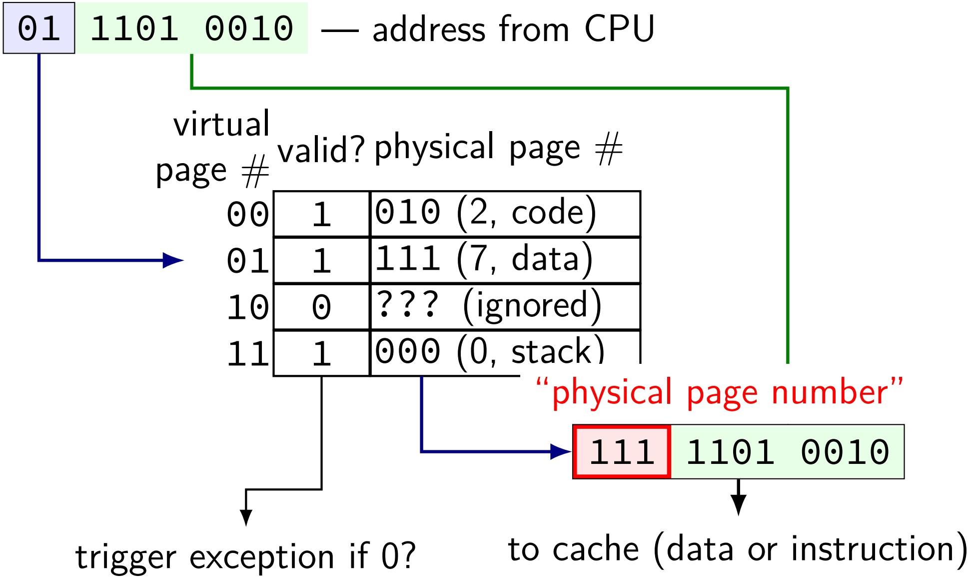
virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

`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)

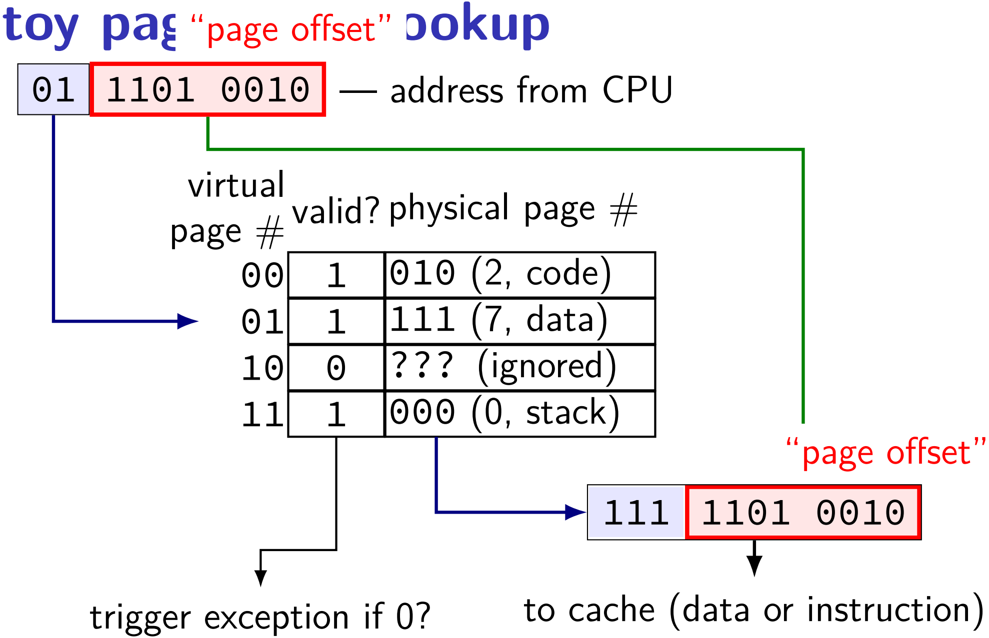
“physical page number”

`111` `1101 0010`

trigger exception if 0?

to cache (data or instruction)

toy pag ookup



`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)

backup slides

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?

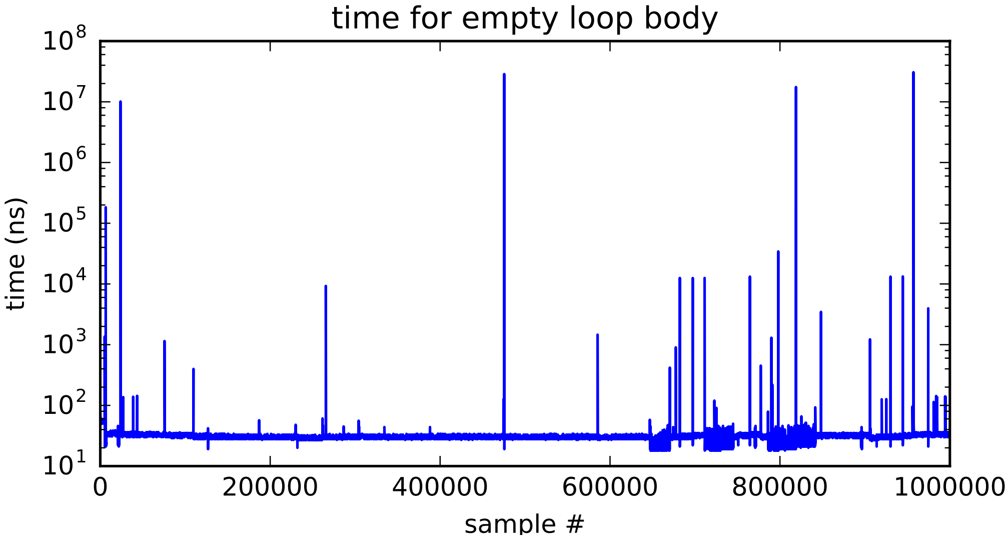
...if the machine only has one core?

timing nothing

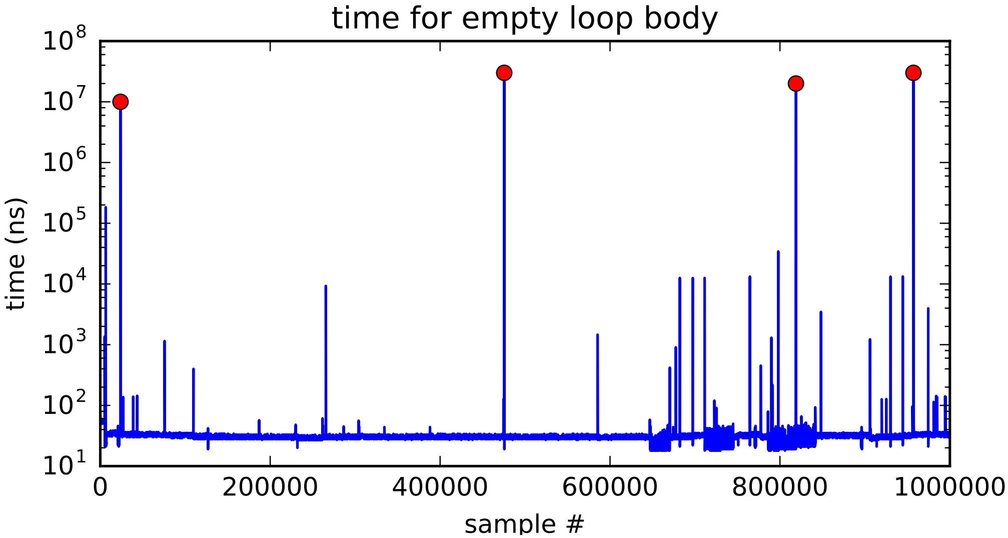
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

doing nothing on a busy system



doing nothing on a busy system



time multiplexing

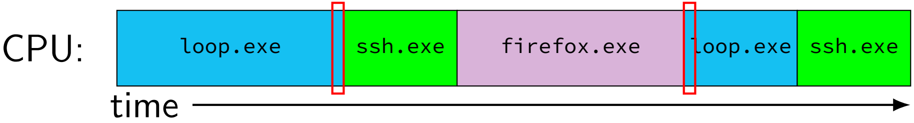
time multiplexing

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

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
```

Program B

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

result: `%rax` is ...

A. 42 B. 99 C. 0x10000

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

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

F. 99 or program might crash (depending on ...)

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

H. something else

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

added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

  may need to cancel partially completed instructions before jumping

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

- may need to cancel partially completed instructions before jumping

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

- may need to cancel partially completed instructions before jumping

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

- may need to cancel partially completed instructions before jumping

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

protection fault

when program tries to access memory it doesn't own

e.g. trying to write to OS 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, …

aborts — hardware is broken

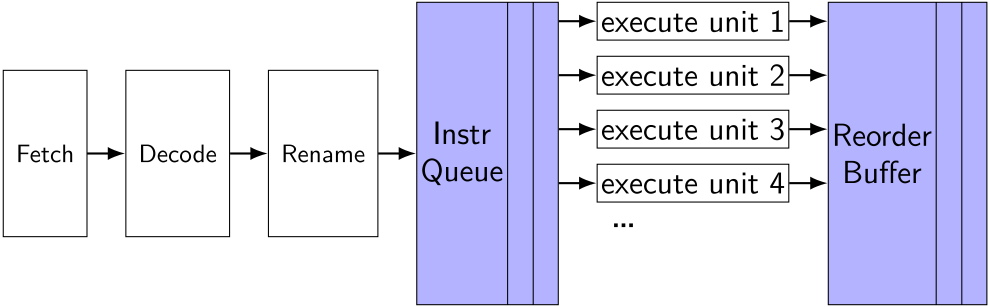
asynchronous
not triggered by
running program

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

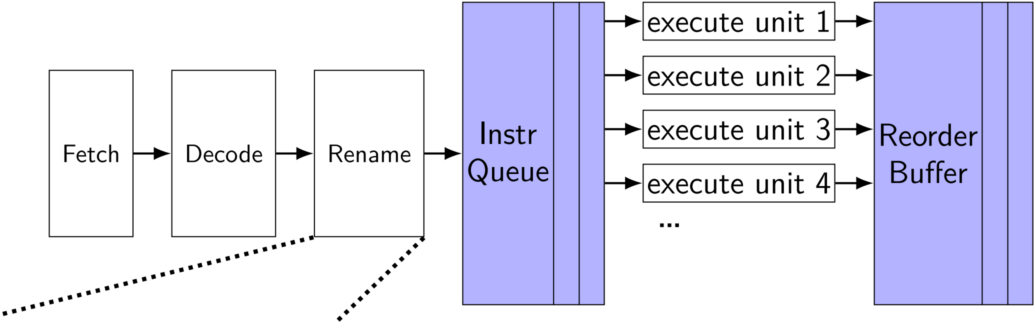
faults — errors/events in programs
- memory not in address space (“Segmentation fault”)
- privileged instruction
- divide by zero
- invalid instruction

synchronous
triggered by
current program

exceptions and OOO (one strategy)



exceptions and OOO (one strategy)



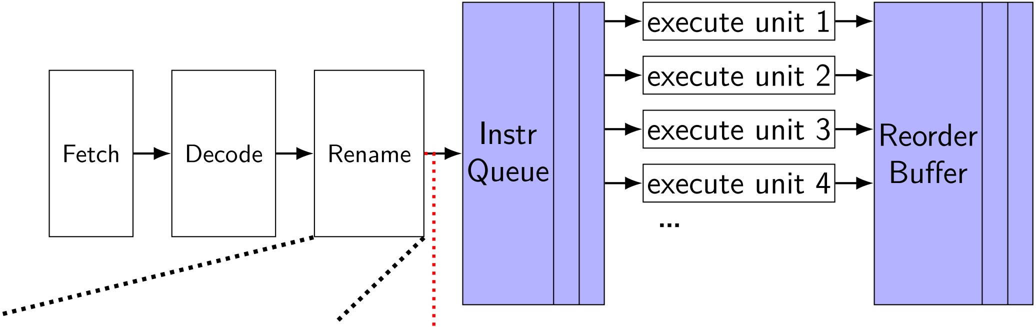
free regs

X19
X23
...

for new instrs

arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

exceptions and OOO (one strategy)



free regs

X19
X23
…

for new instrs

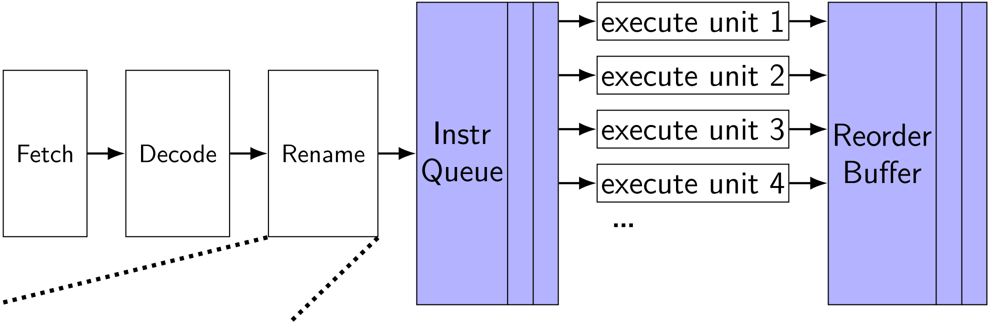
arch. reg	**phys. reg**
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

done instrs committed in order

instr num.	**PC**	**dest. reg**	**done?**	**except?**
…	…	…	…	…
17	0x1244	RCX / X32		
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
…	…	…	…	…

new instrs added

exceptions and OOO (one strategy)



free regs

X19
X23
…

for new instrs

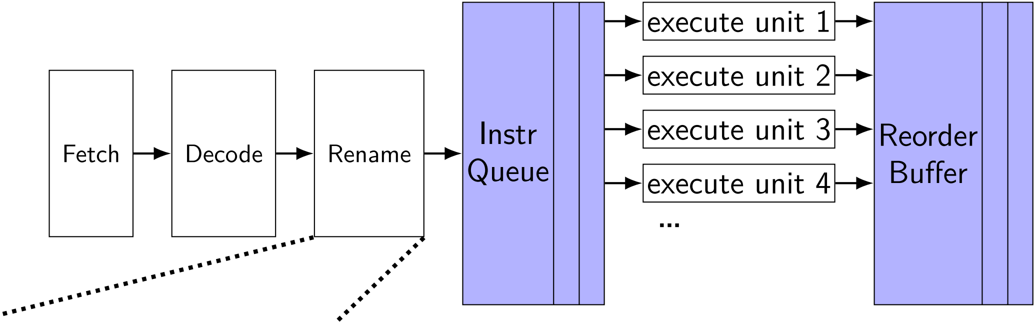
arch. reg	**phys. reg**
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

for complete instrs

arch. reg	**phys. reg**
RAX	X21
RCX	~~X2~~ X32
RBX	X48
RDX	X37
…	…

instr num.	**PC**	**dest. reg**	**done?**	**except?**
…	…	…	…	…
17	0x1244	RCX / X32	✓	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



Fetch

Decode

Rename

Instr Queue

execute unit 1

execute unit 2

execute unit 3

execute unit 4

…

Reorder Buffer

free regs

X19
X23
…

for new instrs

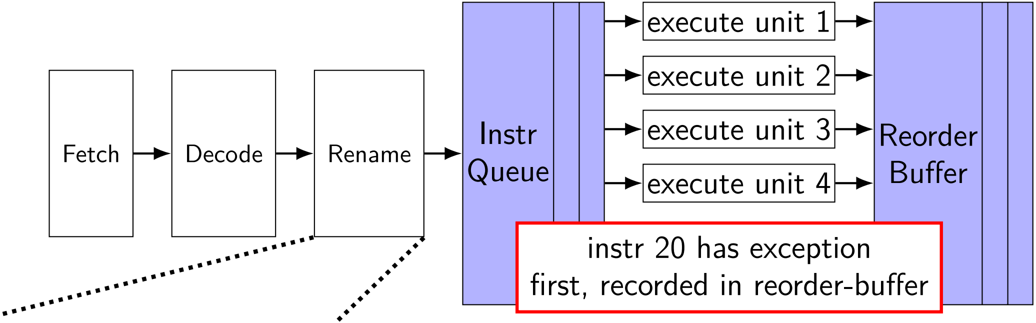
arch. reg	**phys. reg**
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

for complete instrs

arch. reg	**phys. reg**
RAX	X21
RCX	~~X2~~ X32
RBX	X48
RDX	X37
…	…

instr num.	**PC**	**dest. reg**	**done?**	**except?**
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



instr 20 has exception
first, recorded in reorder-buffer

free regs

X19
X23
…

for new instrs

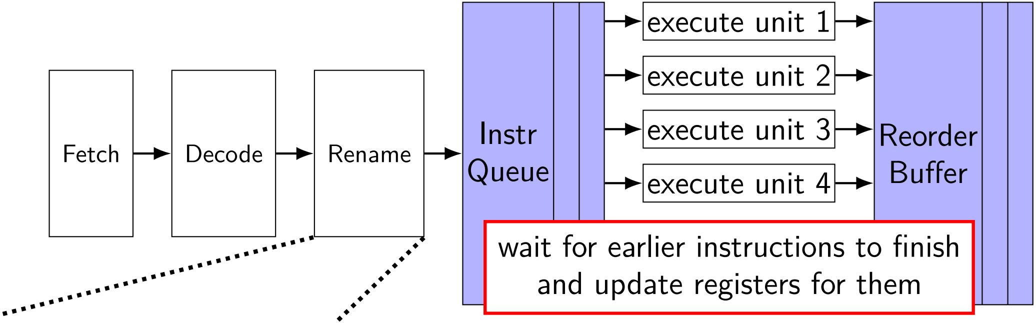
arch. reg	**phys. reg**
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

for complete instrs

arch. reg	**phys. reg**
RAX	X21
RCX	~~X2~~ X32
RBX	X48
RDX	X37
…	…

instr num.	**PC**	**dest. reg**	**done?**	**except?**
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



wait for earlier instructions to finish and update registers for them

free regs

X19
X23
...

for new instrs

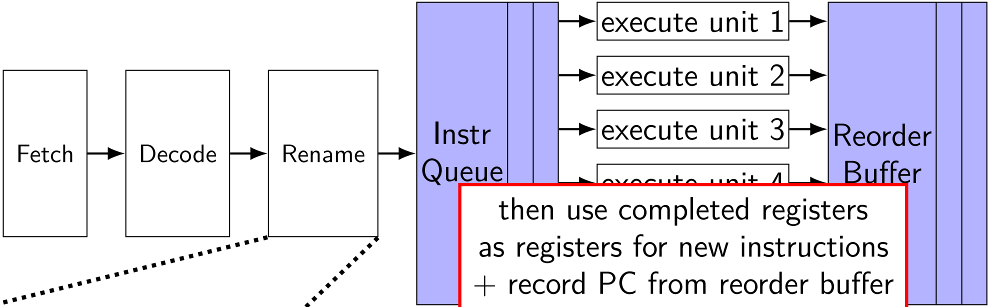
arch. reg	**phys. reg**
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	**phys. reg**
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
...	...

instr num.	**PC**	**dest. reg**	**done?**	**except?**
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



then use completed registers
as registers for new instructions
+ record PC from reorder buffer
+ jump to exception handler

free regs

X19
X23
…

for new instrs

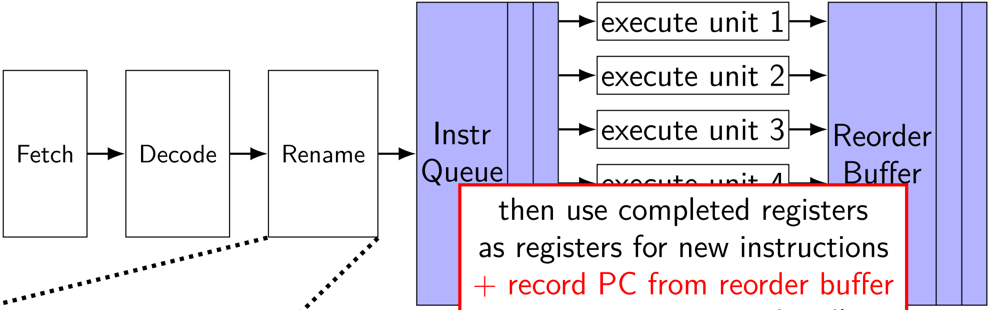
arch. reg	phys. reg
RAX	X38
RCX	X32
RBX	X48
RBX	X34
…	…

for complet

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
~~18~~	~~0x1248~~	~~RDX / X34~~	~~✓~~	
~~19~~	~~0x1249~~	~~RAX / X38~~	~~✓~~	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



then use completed registers
as registers for new instructions
+ record PC from reorder buffer
+ jump to exception handler

free regs

X19
X23
…

for new instrs

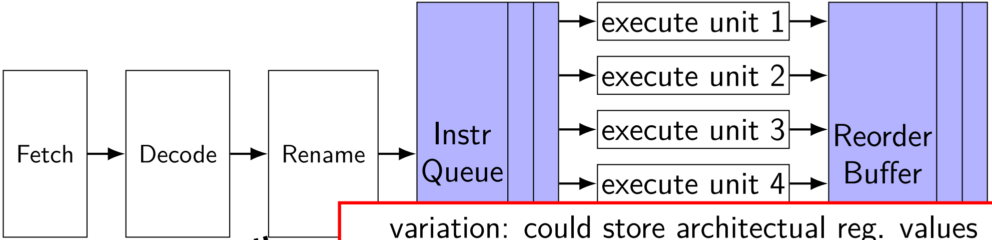
arch. reg	**phys. reg**
RAX	X38
RCX	X32
RBX	X48
RBX	X34
…	…

for complet

arch. reg	**phys. reg**
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
…	…

instr num.	**PC**	**dest. reg**	**done?**	**except?**
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
~~18~~	~~0x1248~~	~~RDX / X34~~	~~✓~~	
~~19~~	~~0x1249~~	~~RAX / X38~~	~~✓~~	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



variation: could store architectual reg. values instead of mapping for completed instrs.
(and copy values instead of mapping on exception)

free regs

X19
X23
…

for new instrs

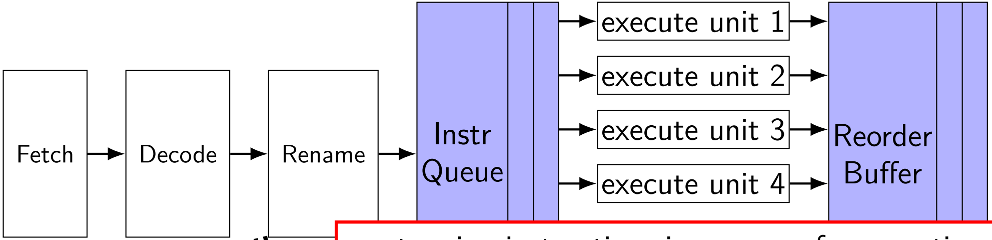
arch. reg	**phys. reg**
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

for complete instrs

arch. reg	**value**
RAX	0x12343
RCX	0x234543
RBX	0x56782
RDX	0xF83A4
…	…

instr num.	**PC**	**dest. reg**	**done?**	**except?**
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	✓	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



stopping instructions in progress for exception
similar to how ‘squashing’ mispredicted instructions

free regs

X19
X23
…

for new instrs

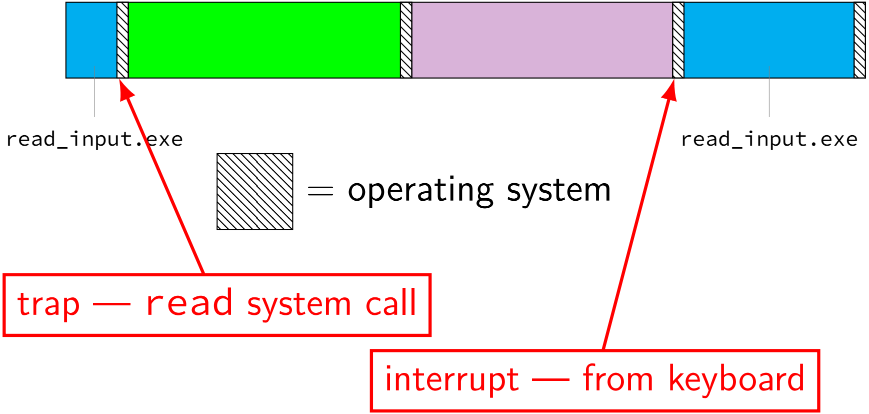
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

for complete instrs

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
…	…	…	…	…

keyboard input timeline



▨ = operating system

trap — read system call

interrupt — from keyboard