last time (1)

finish vector instructions

- additional hardware support can help with conditionals, loading/storing non-contiguous values
- other vector interfaces exist

profilers: optimize what matters

process idea:

- thread: illusion of dedicated core (via time multiplexing)
- address space: illusion of dedicated memory (via address translation/virtual memory)

last time (2)

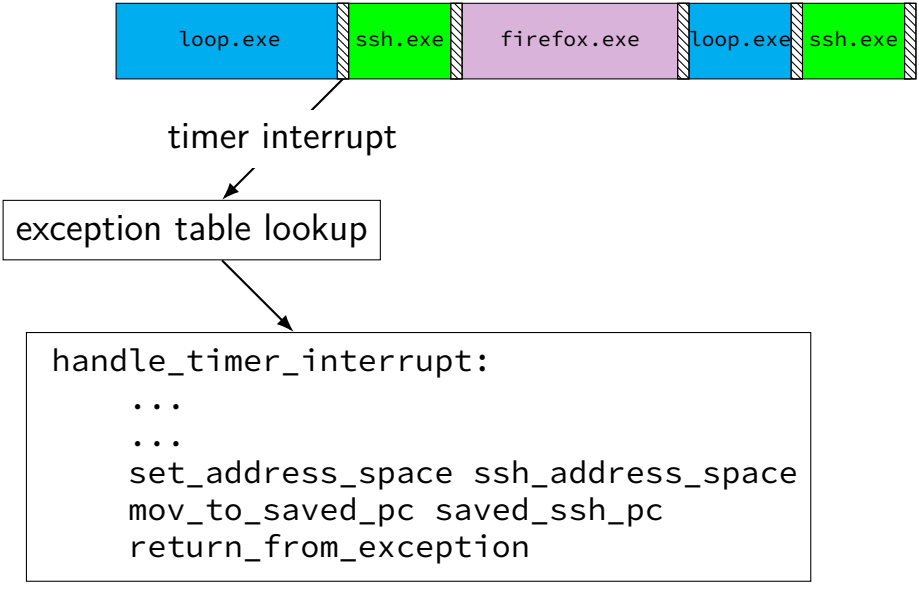
contexts and context switches

(hardware) exceptions: processor gives OS control

- asynchronous: on external events (timer, IO, etc.)
- synchronous: from program events (system call, invalid instruction, out-of-bounds access)

dispatching exceptions: exception tables

exceptions and time slicing



defeating time slices?

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

main:
    set_exception_table_base my_exception_table
loop:
    jmp loop
```

defeating time slices?

wrote a program that tries to set the exception table:

```
my_exception_table:
    ...

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

result: Segmentation fault (exception!)

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

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

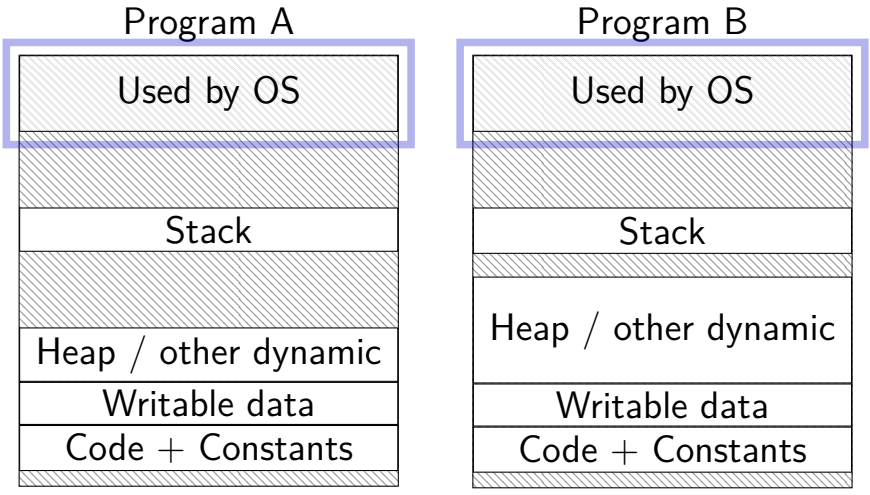
traps — intentionally triggered exceptions
- system calls — ask OS to do something

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

} synchronous
triggered by current program

what about editing exception table?

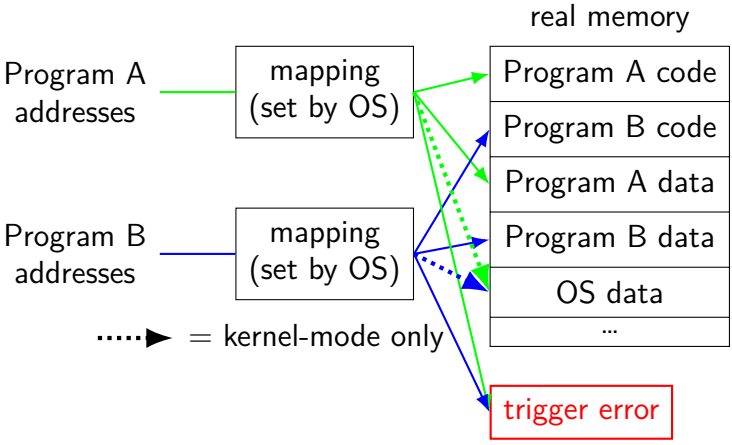
program memory (two programs)



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 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

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

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

can't write C code to generate syscall instruction

solution: call "wrapper" function written in assembly

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

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”

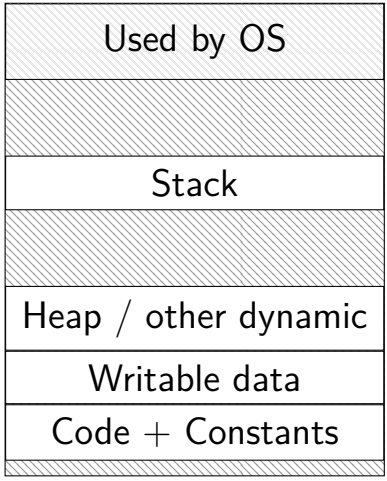
some additional terms:

- supervisor mode
- privileged mode
- ring `0`

some systems have multiple levels of privilege

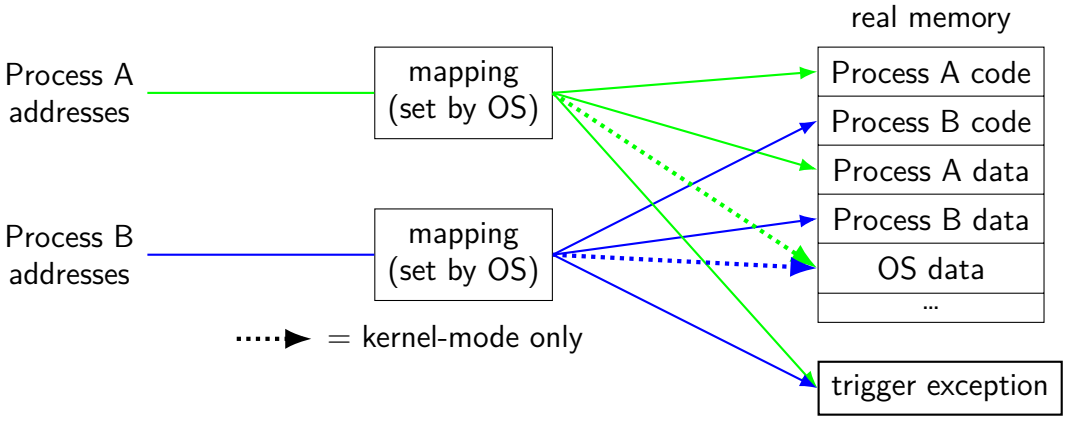
- different sets of priviliged operations work

program memory



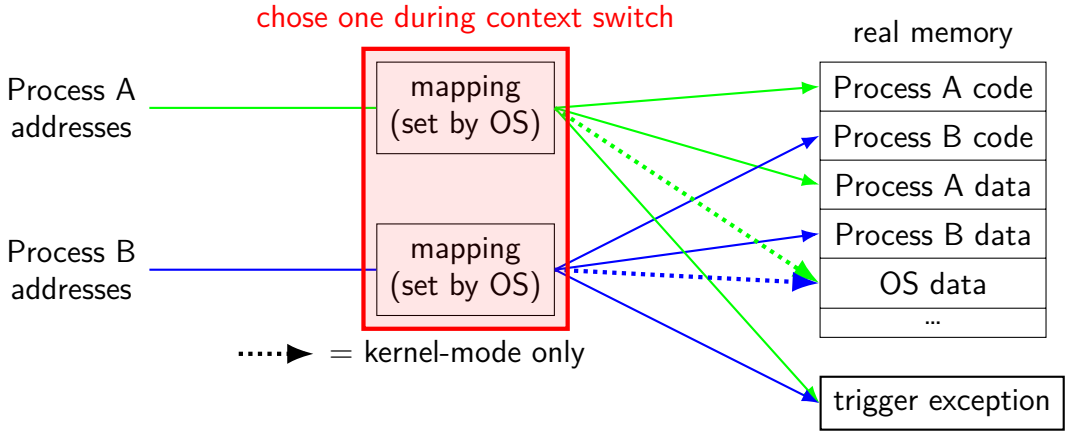
address spaces

illuision of dedicated memory

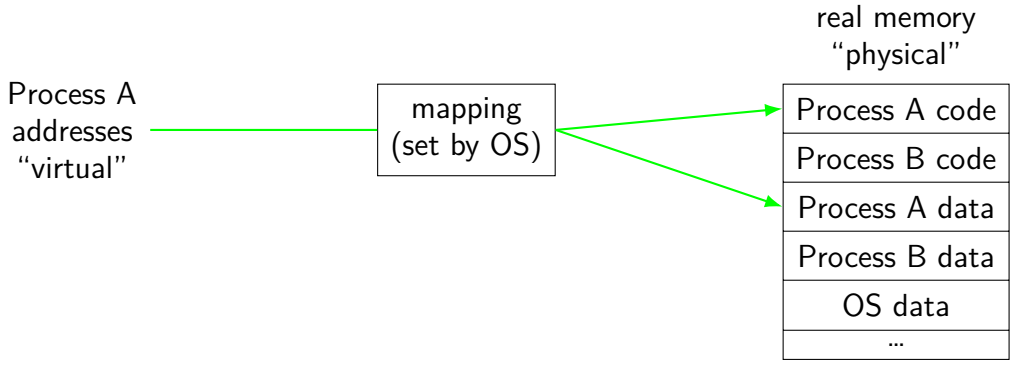


address spaces

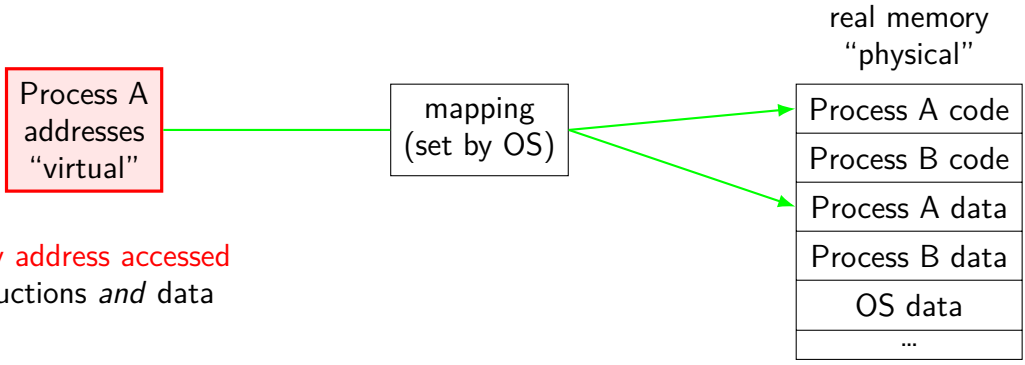
illuision of dedicated memory



address translation

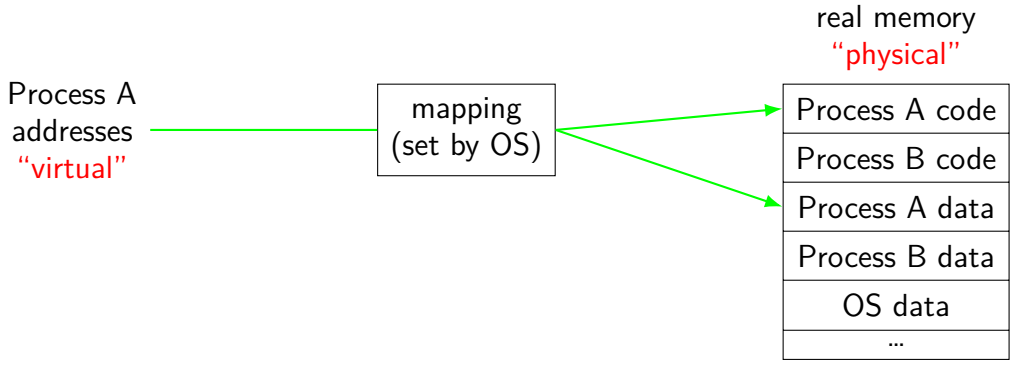


address translation



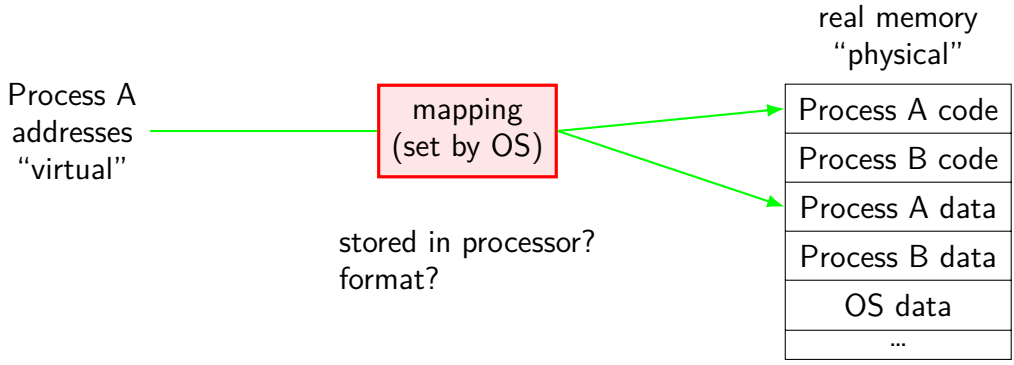
every address accessed
instructions *and* data

address translation

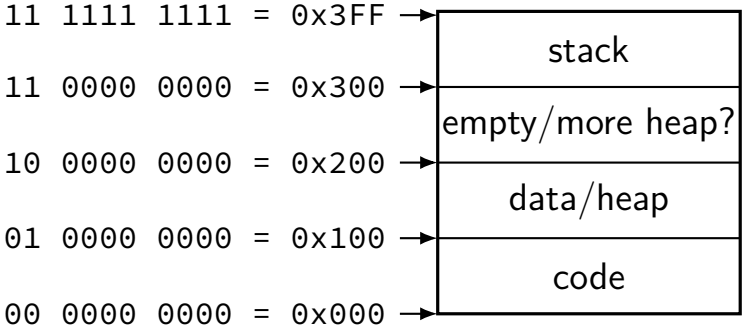


program addresses are 'virtual'
real addresses are 'physical'
can be different sizes!

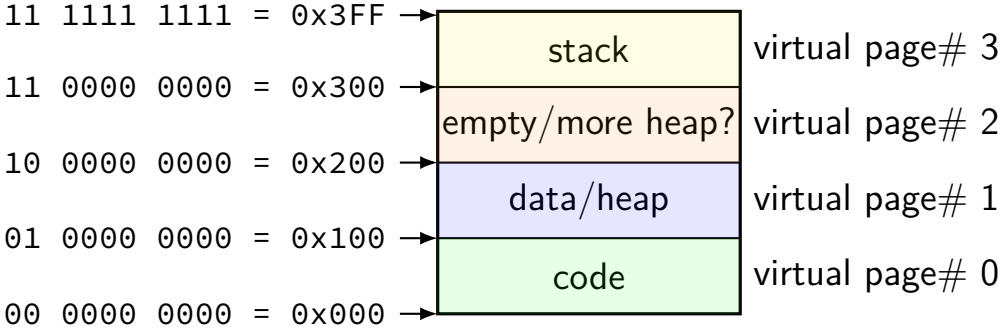
address translation



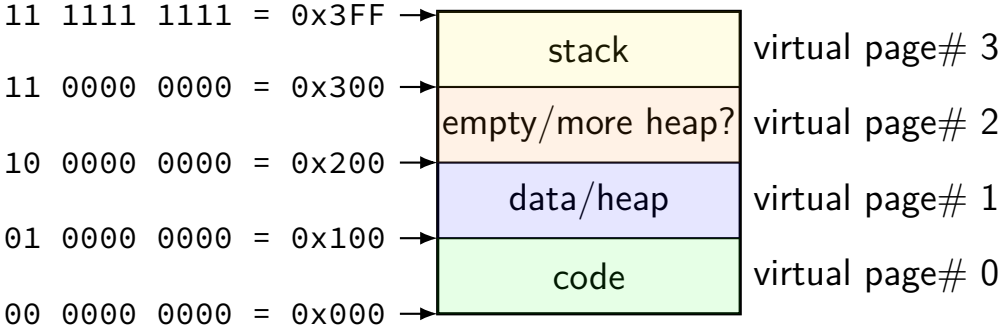
toy program memory



toy program memory

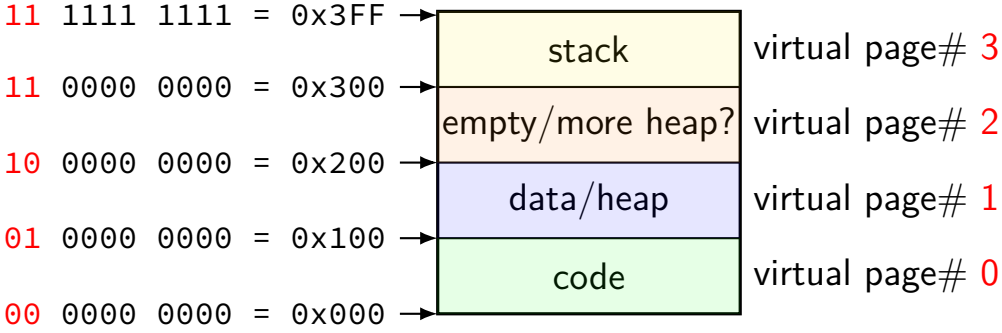


toy program memory



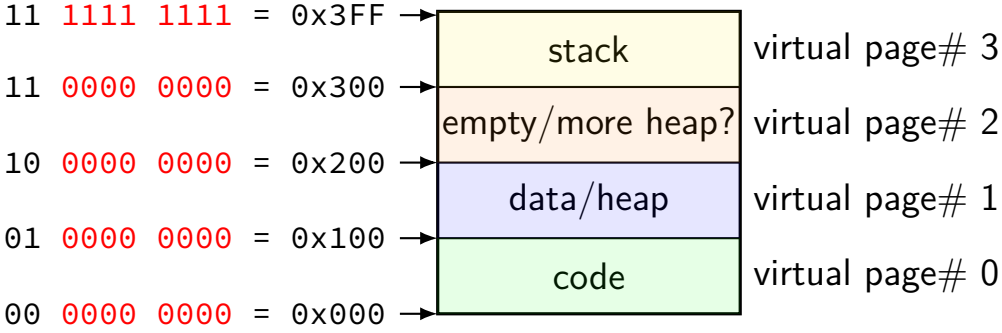
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

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

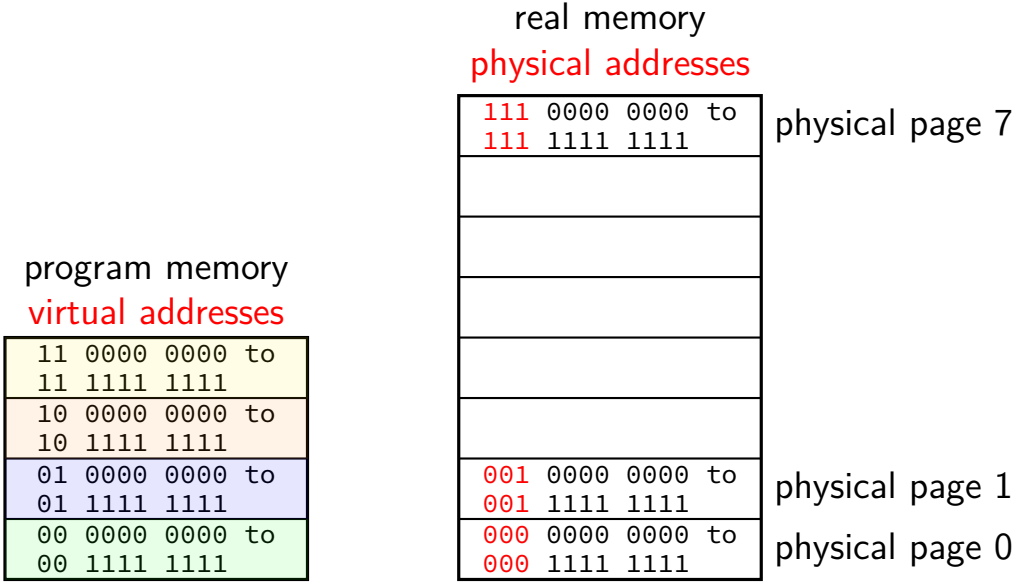
program memory
virtual addresses

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

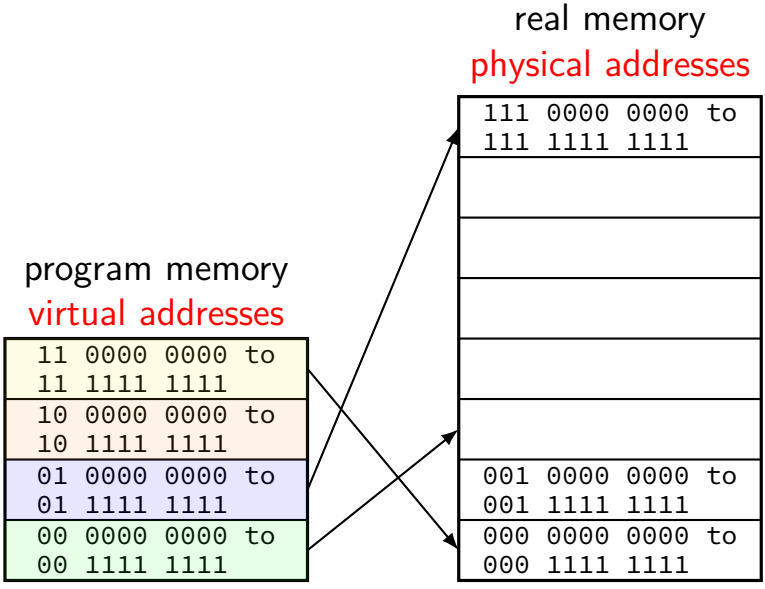
real memory
physical addresses

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

toy physical memory

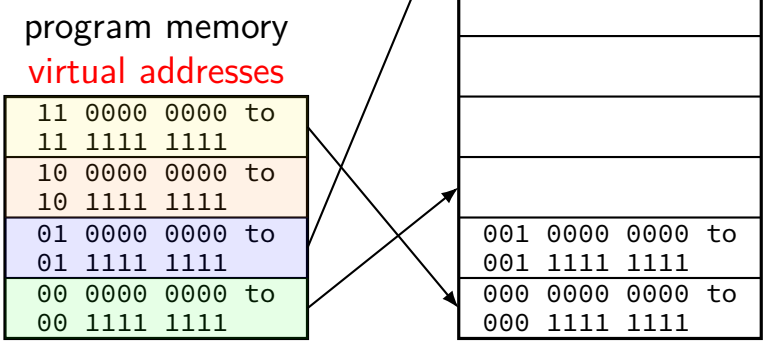


toy physical memory



toy physical memory

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



toy physical memory

page table!

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

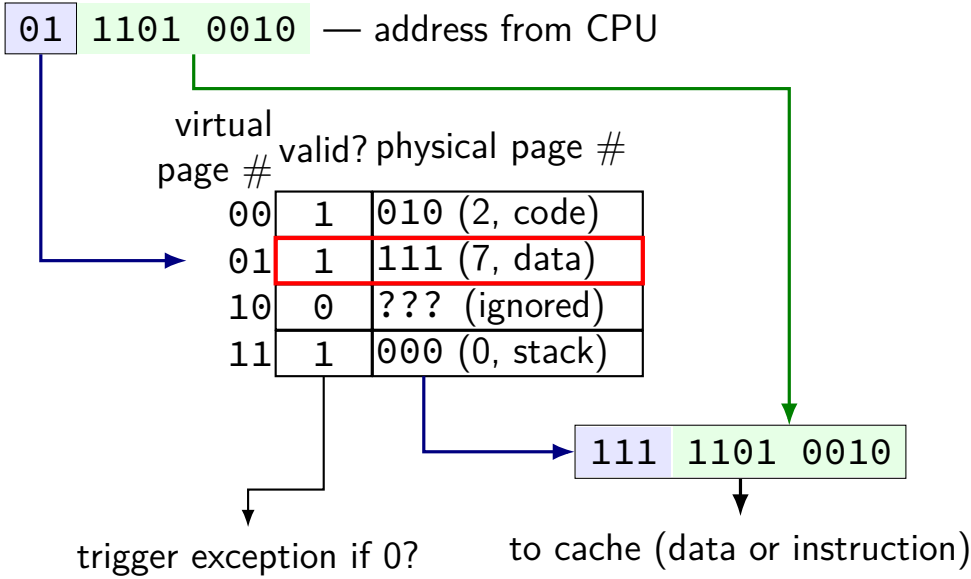
real memory
physical addresses

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

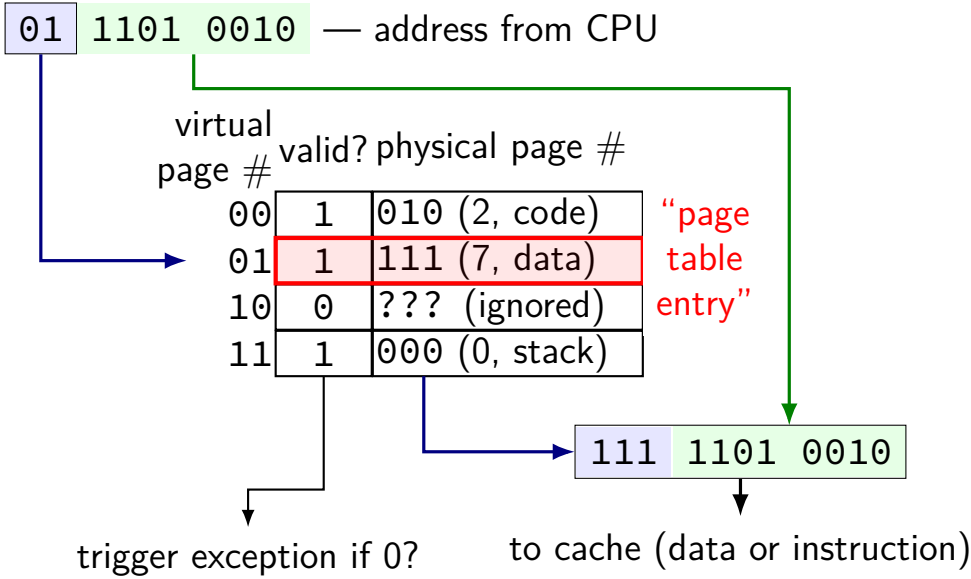
toy page table lookup

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

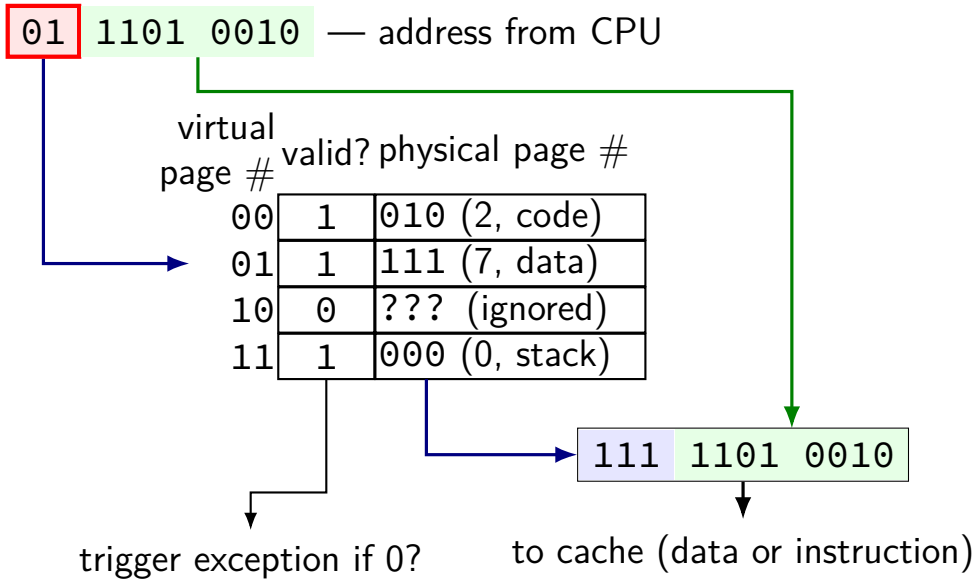
toy page table lookup



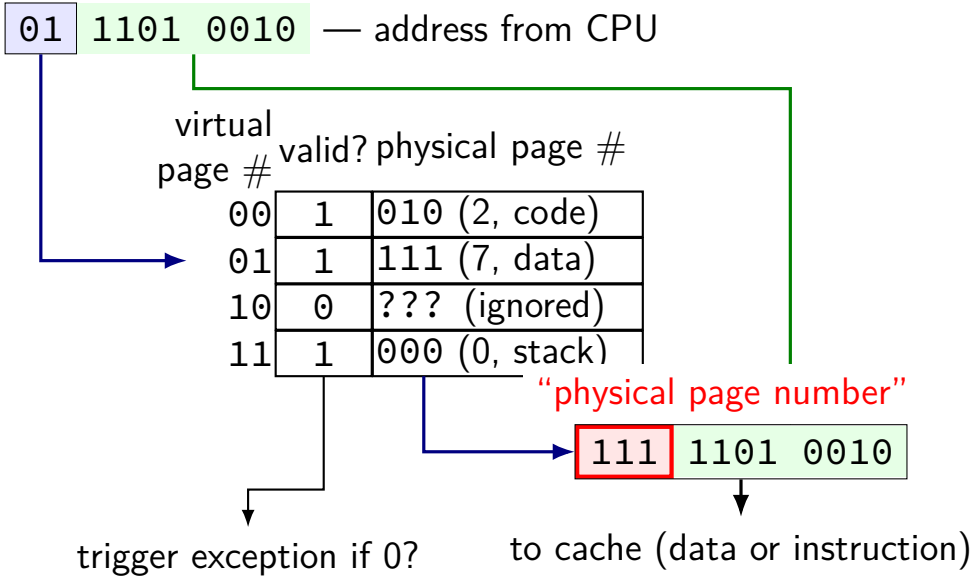
toy page table lookup



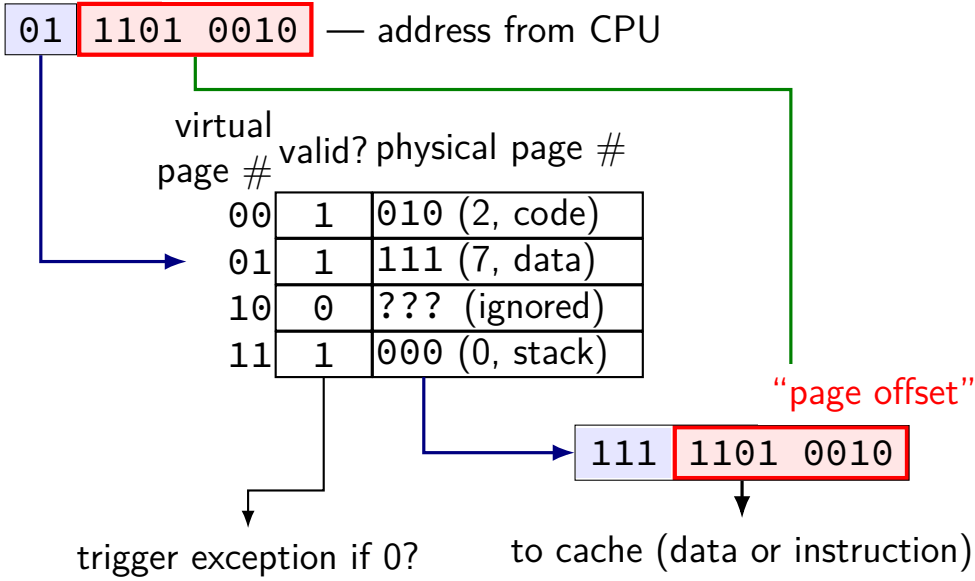
"virtual page number" lookup



toy page table lookup



toy pa "page offset" ookup



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?

- probably have a page table entry pointing to it
- hopefully marked kernel-mode-only

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?

- probably have a page table entry pointing to it
- hopefully marked kernel-mode-only

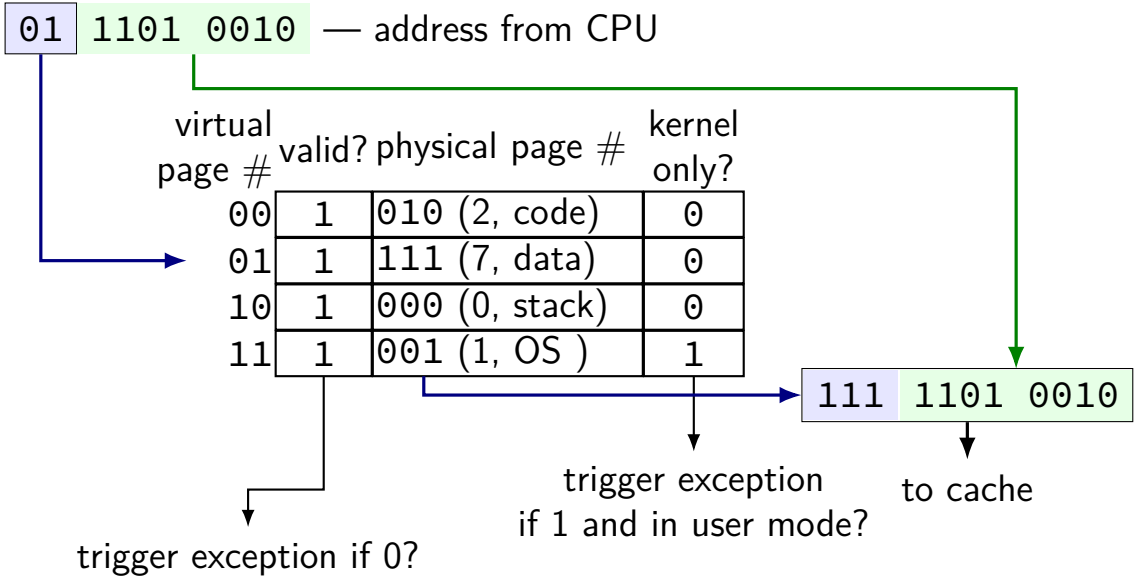
code better not be modified by user program

- otherwise: uncontrolled way to "escape" user mode

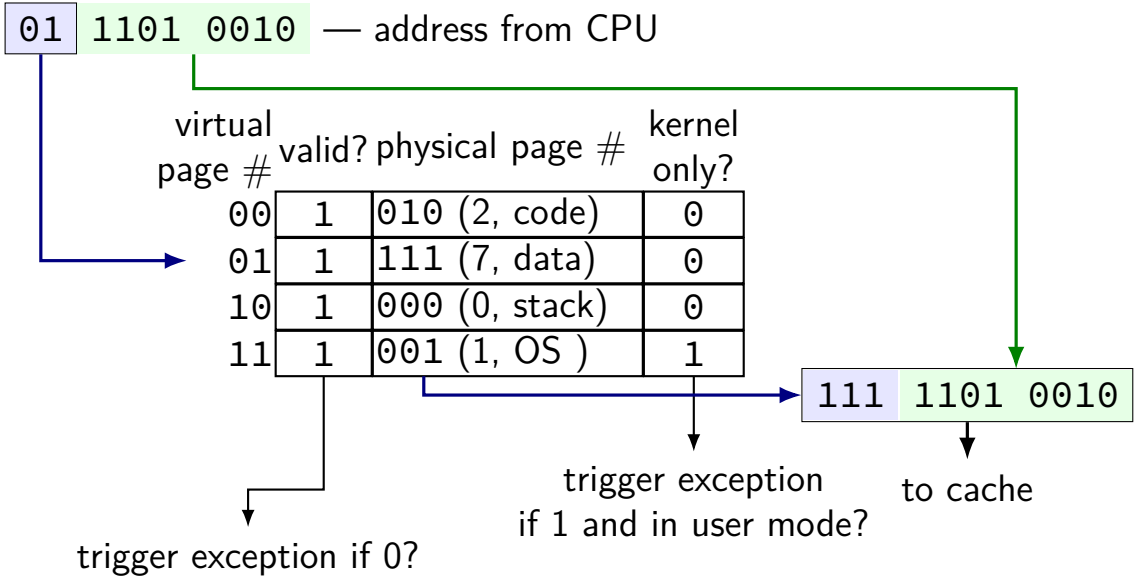
kernel-mode only

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

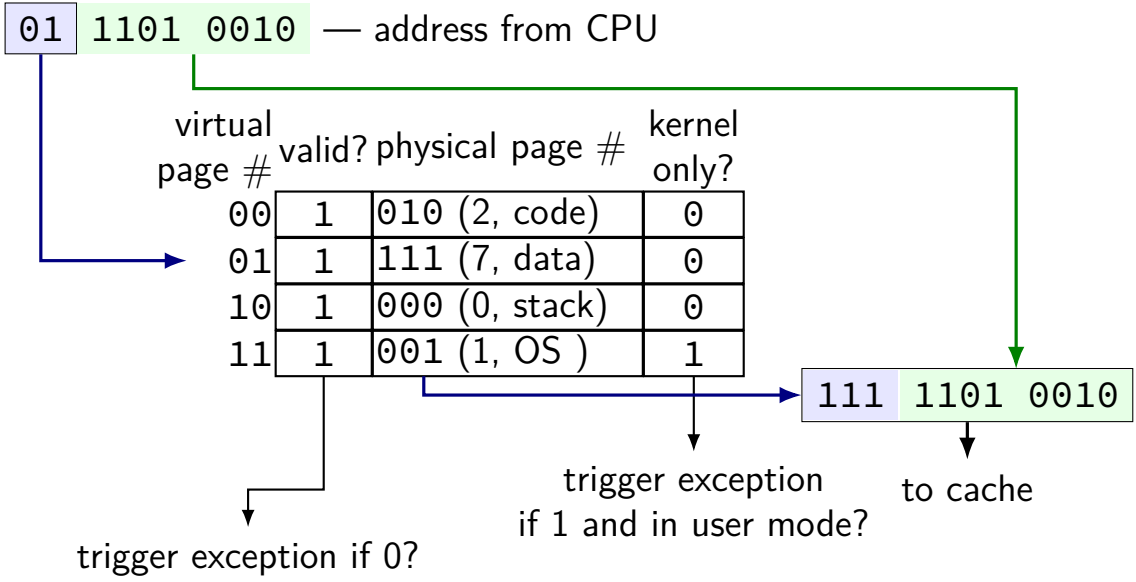
kernel-mode only



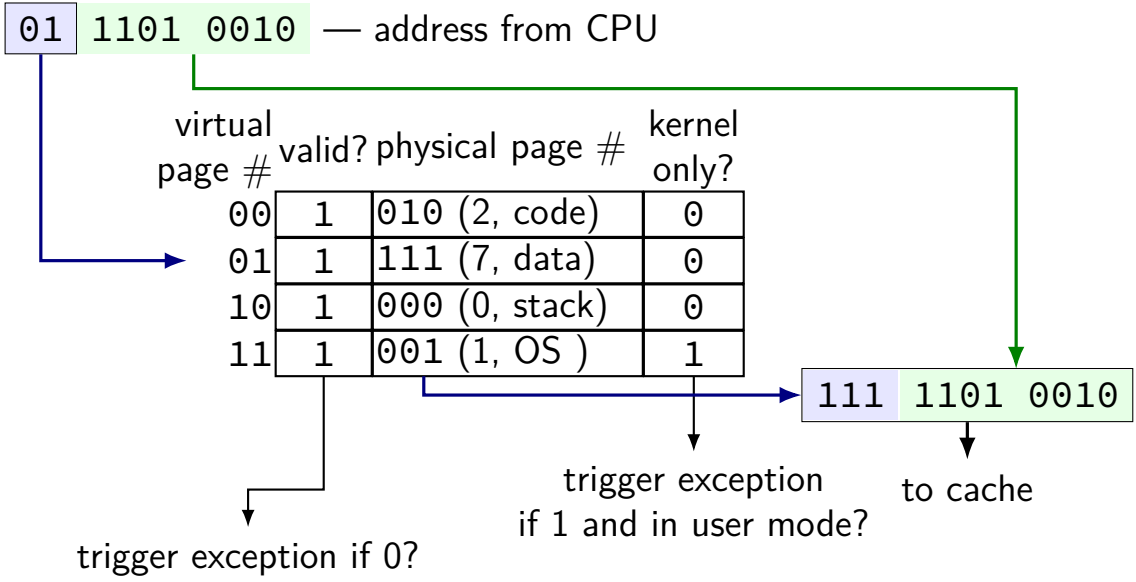
kernel-mode only



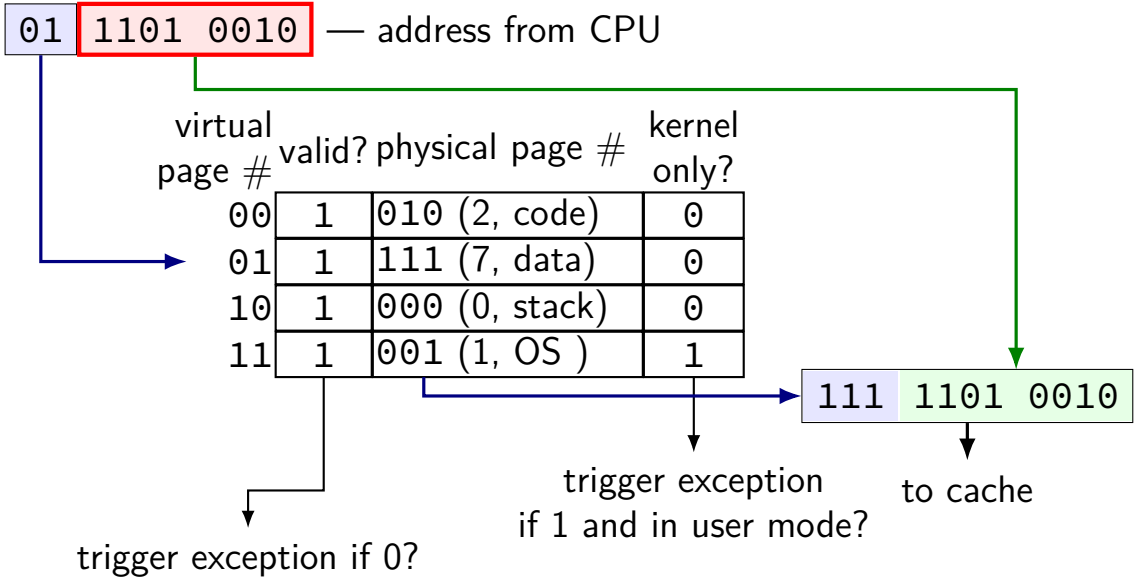
kernel-mode only



kernel-mode only

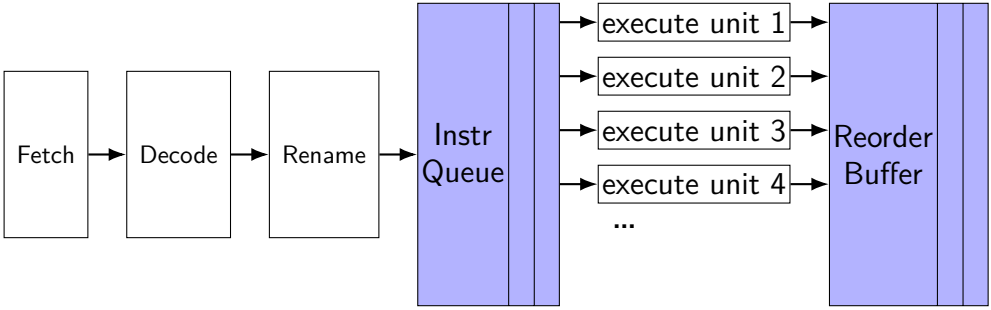


kernel-mode only

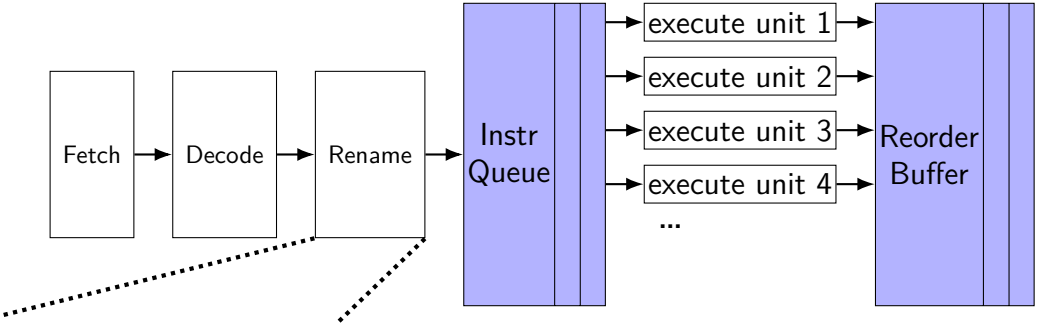


backup slides

exceptions and OOO (one strategy)



exceptions and OOO (one strategy)



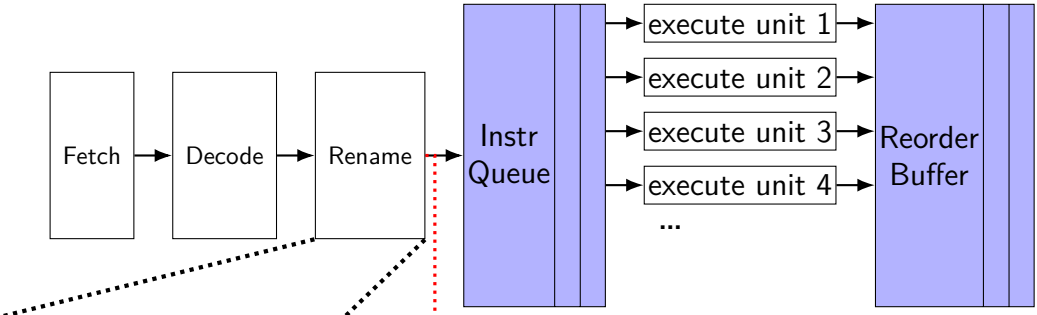
free regs

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

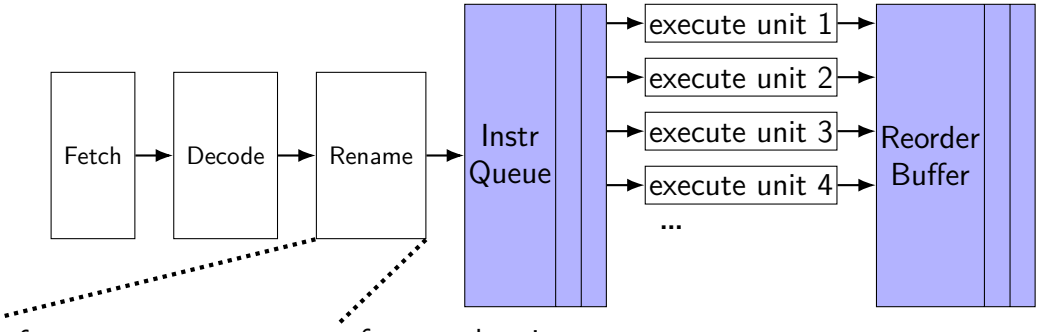
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

done instrs committed in order

new instrs added

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)



free regs

X19
X23
...

for new instrs

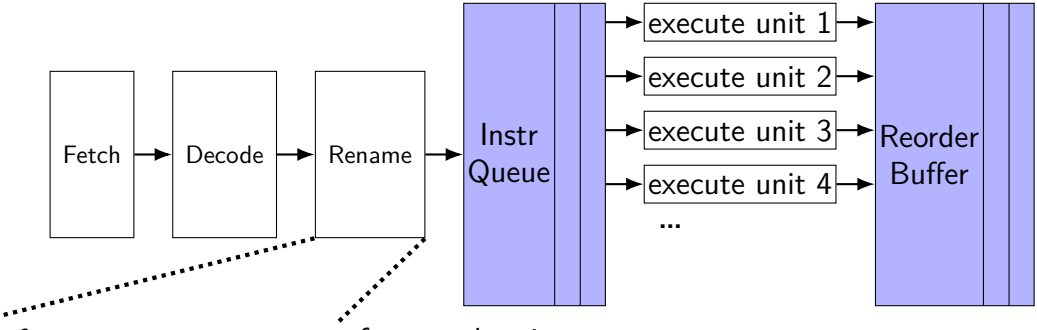
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

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)



free regs

X19
X23
...

for new instrs

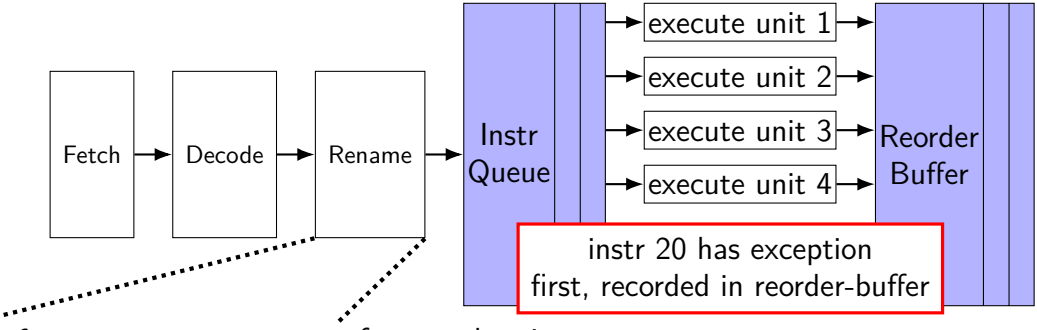
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

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)



free regs

X19
X23
...

for new instrs

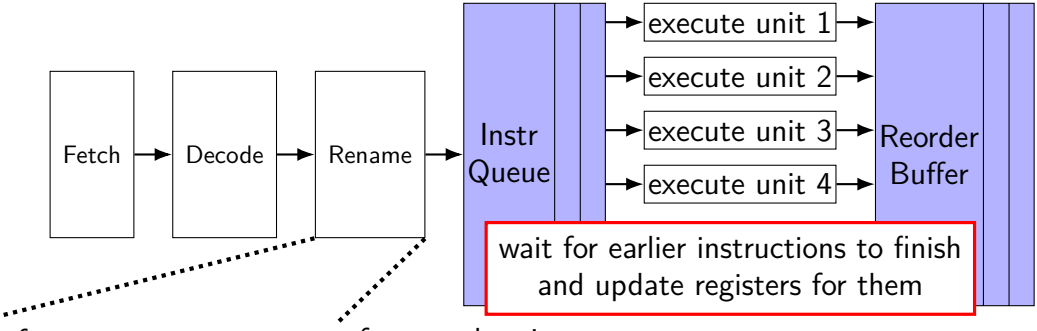
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

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)



free regs

X19
X23
...

for new instrs

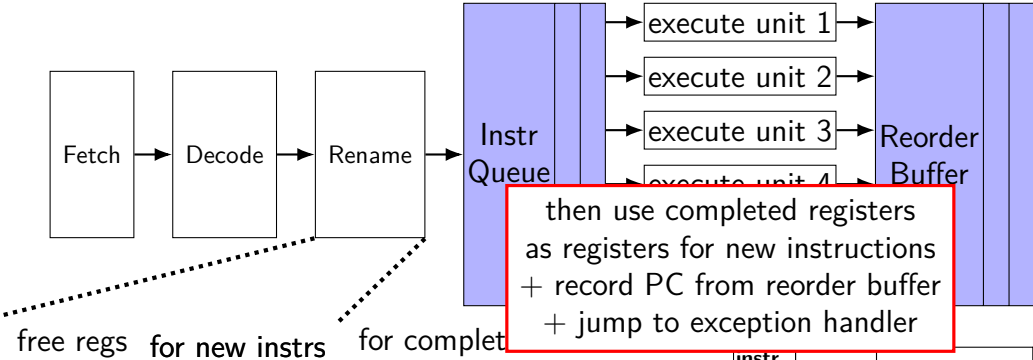
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

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)



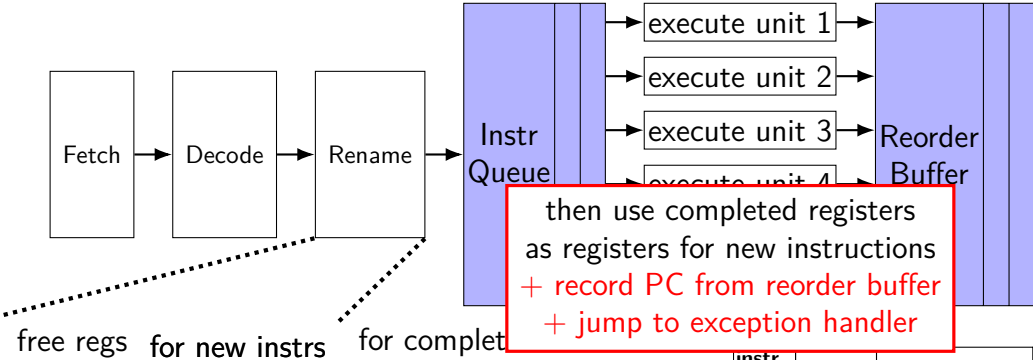
X19
X23
...

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

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)



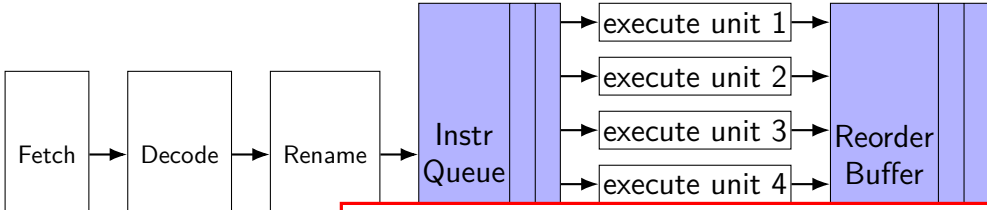
X19
X23
...

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

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

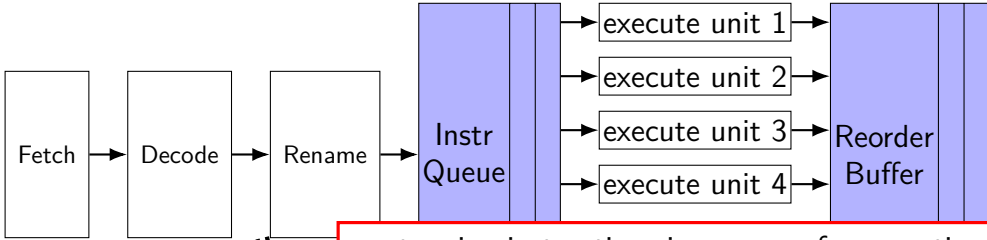
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

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		
...	...	...	...	...