Virtual Memory

Changelog

Changes made in this version not seen in first lecture:

16 November 2017: shared libraries: clarified that it's multiple read-only virtual copies backed by one physical copy

SIMD notes (1)

sorry for two typos:
`partial_sums = _mm_setzero_si128()`, not *(crazy markdown formatting error)*

...and `_mm_mullo_epi32` truncated to 32-bits (not 16)

lab solutions posted on Collab, resources tab

useful for HW?

SIMD notes (2)

don't try to use array indexing on `__m128i` variables

- compiler doesn't know what size values you put in it

`_mm_extract_epi16(some_vector, 3)`

- reads 3rd 16-bit integer in `some_vector`

or store/load from array on the stack

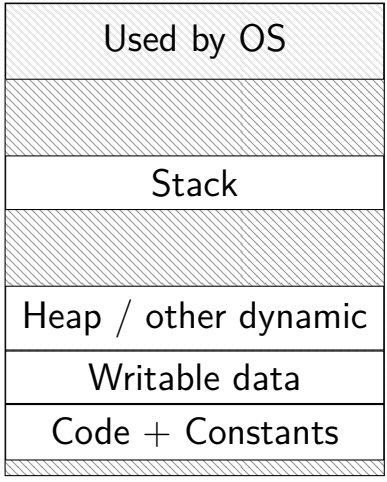
SIMD notes (2)

`_mm_setr_epi16(a[i], a[i+1], a[i+2], ...)` is much slower than `_mm_loadu_si128((__m128i*) &a[i])`

(GCC doesn't figure out that `a[i]` and `a[i+1]` are adjacent... so it makes a temporary array on the stack, element-by-element)

but `_mm_setr_epi16`, etc. are good for constants

program memory



0xFFFF FFFF FFFF FFFF

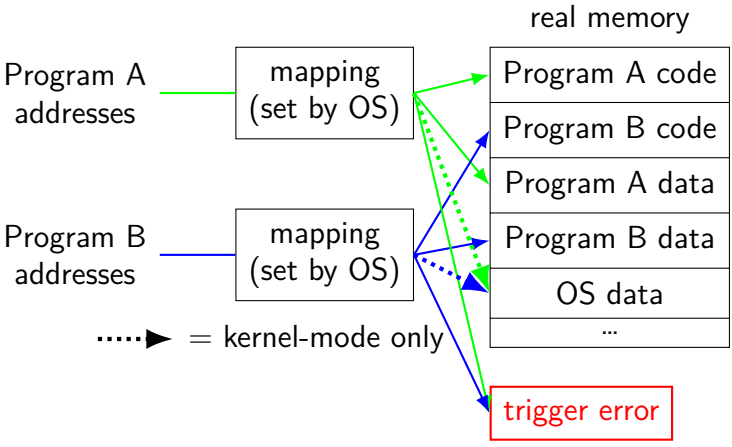
0xFFFF 8000 0000 0000

0x7F…

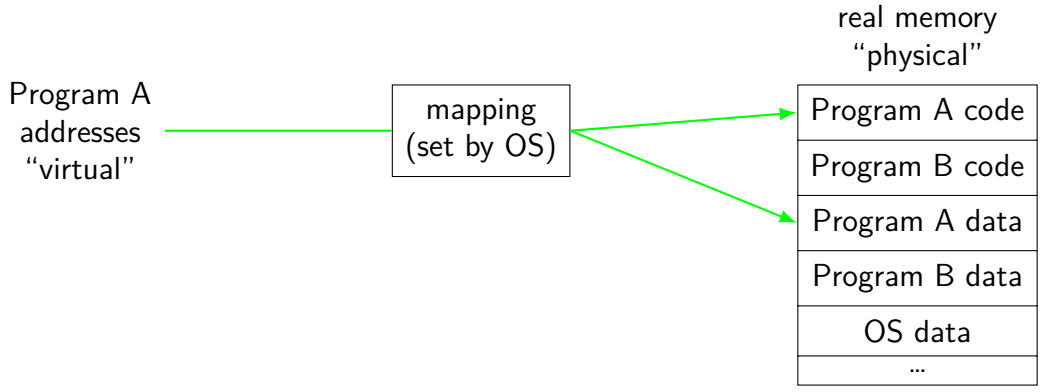
0x0000 0000 0040 0000

Recall: address space

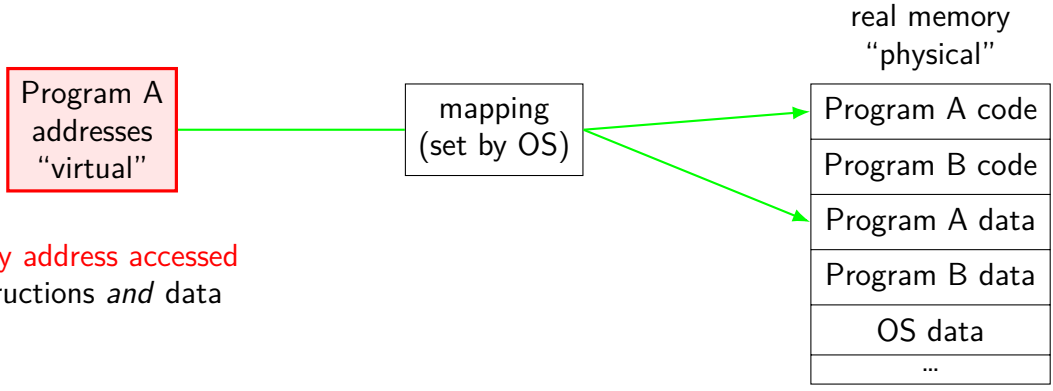
illuision of dedicated memory



address translation

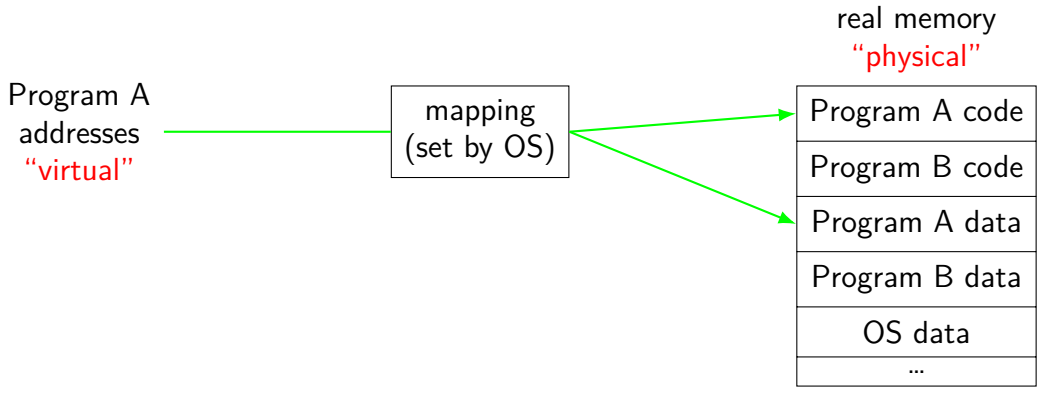


address translation



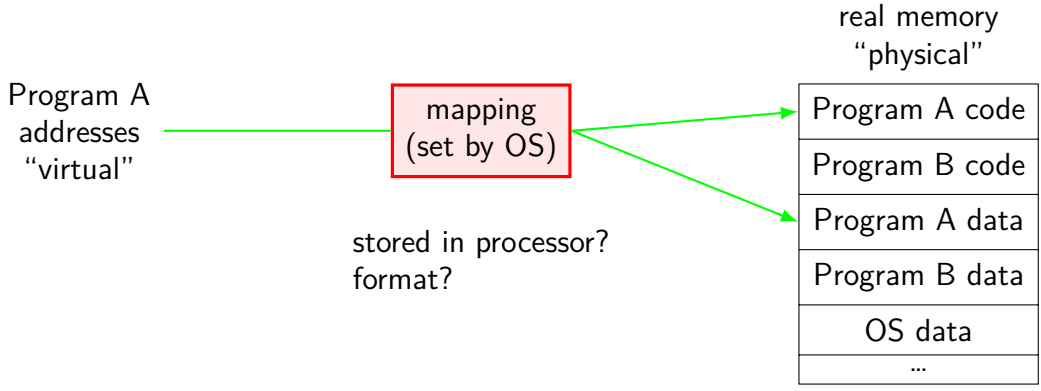
every address accessed
instructions *and* data

address translation



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

address translation



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

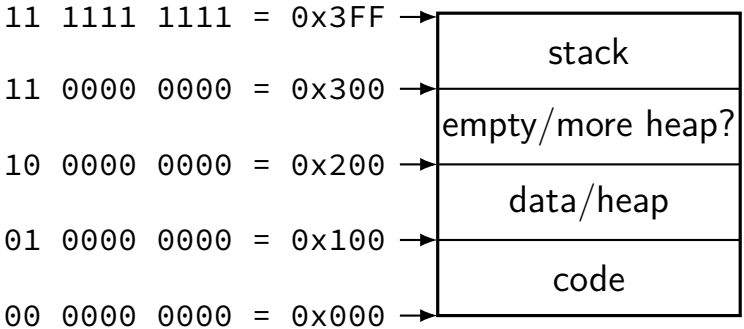
e.g. 32-bit virtual address = 2^{32} byte virtual address space

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

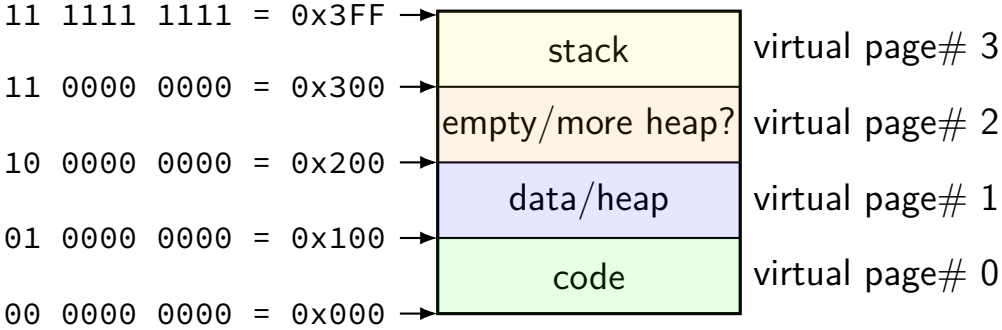
what if my machine has 3GB of memory (not power of two)?

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

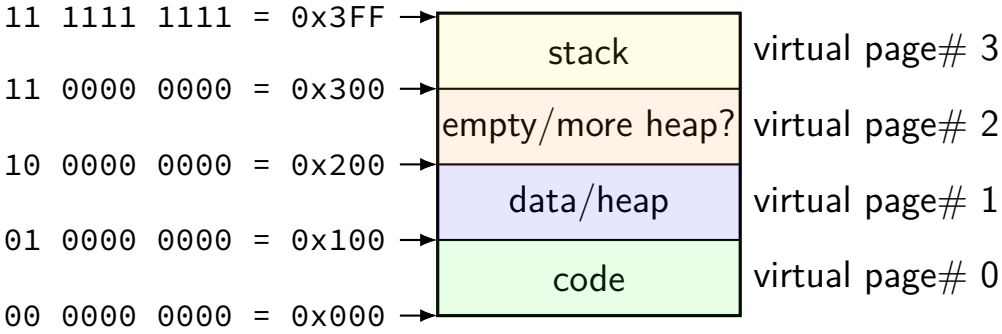
toy program memory



toy program memory

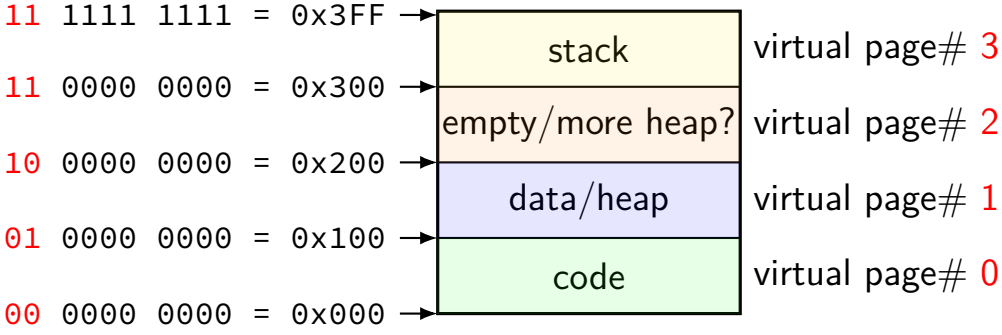


toy program memory



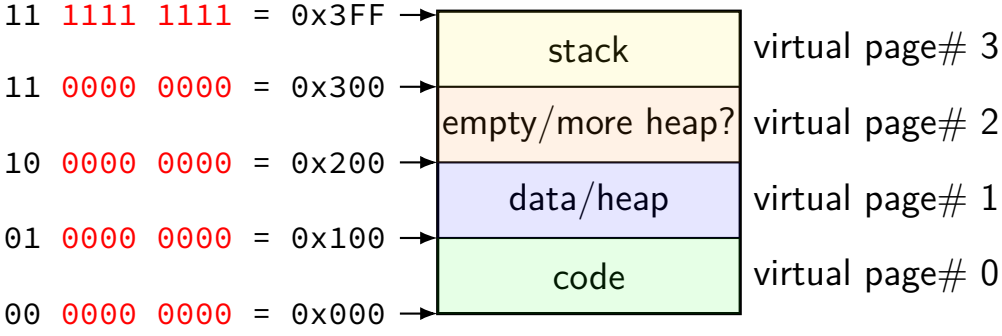
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

toy program memory



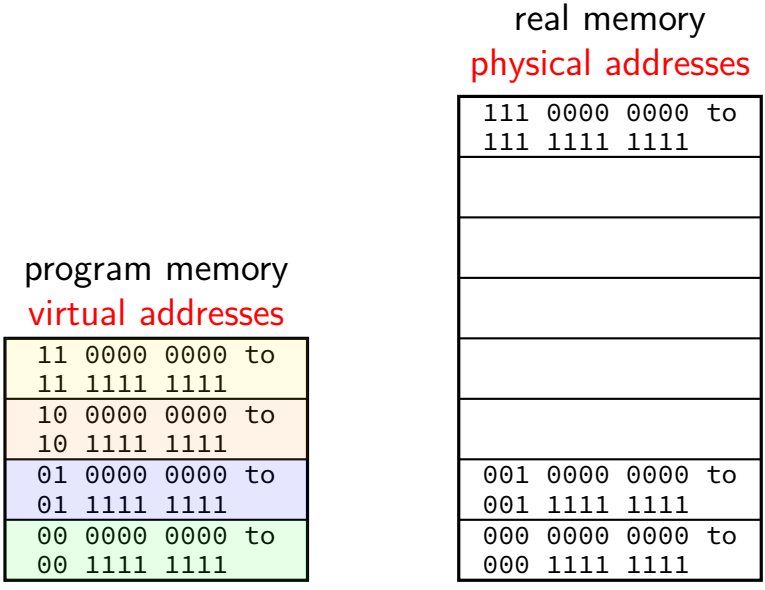
page number is upper bits of address
(because page size is power of two)

toy program memory

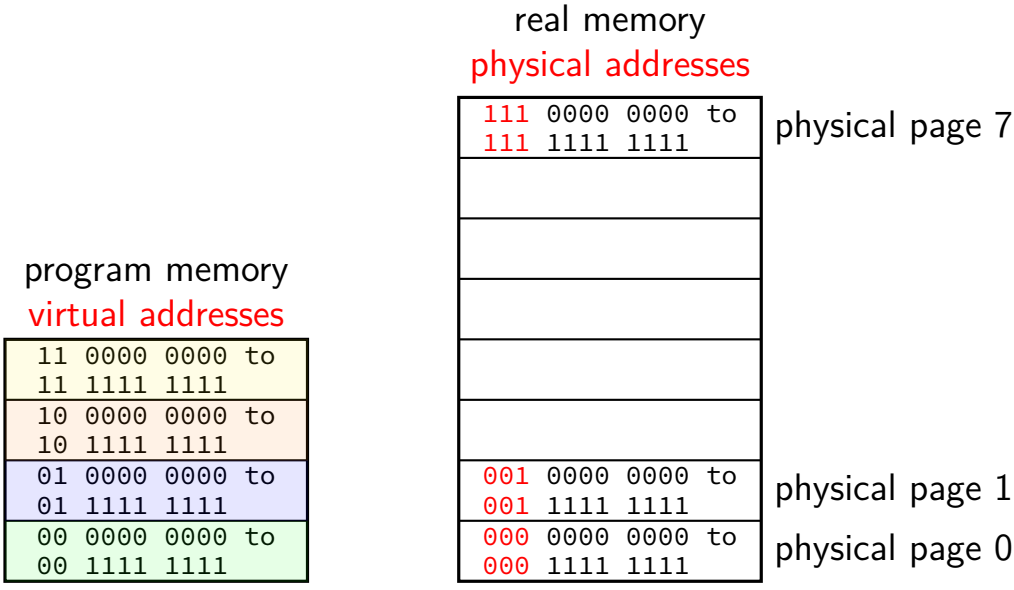


rest of address is called page offset

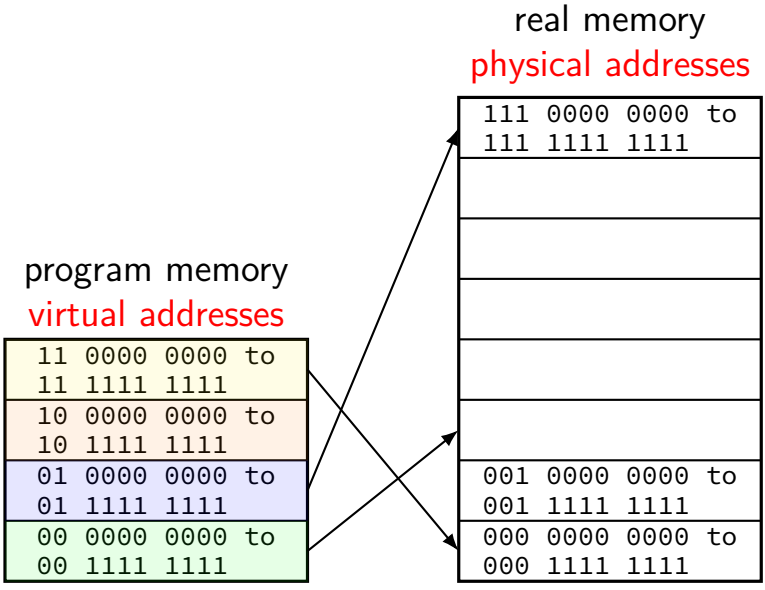
toy physical memory



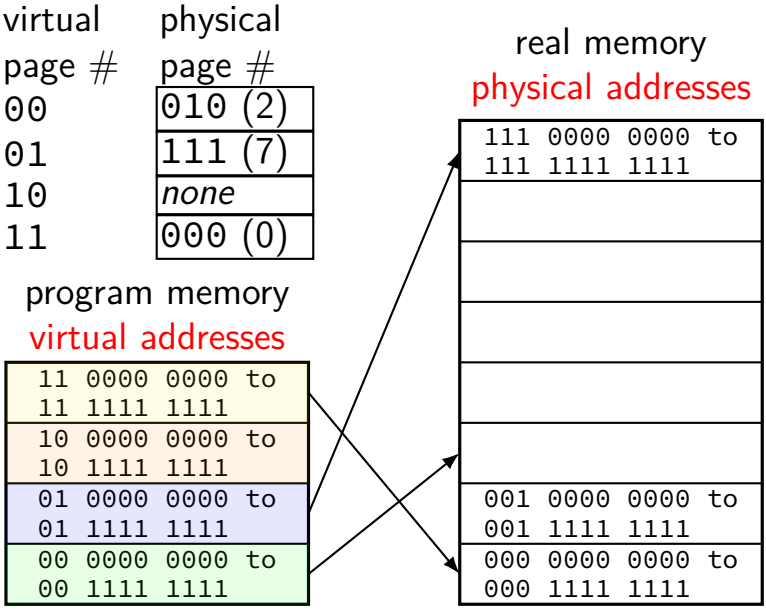
toy physical memory



toy physical memory



toy physical memory



toy physical memory

page table!

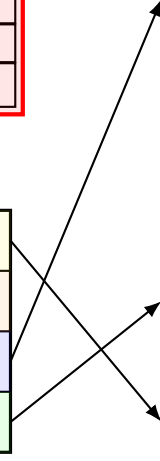
virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
11	000 (0)

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

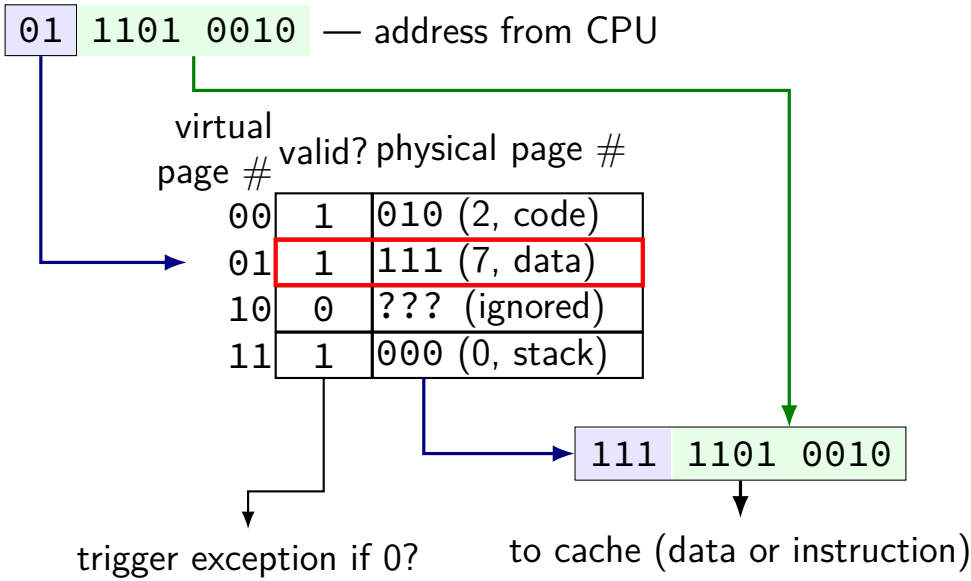
111 0000 0000 to 111 1111 1111
001 0000 0000 to 001 1111 1111
000 0000 0000 to 000 1111 1111



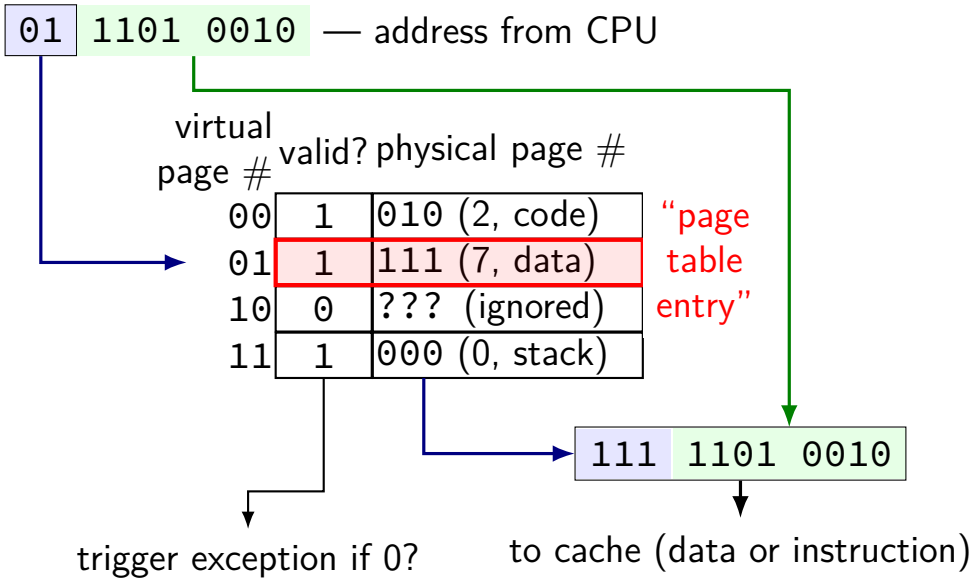
toy page table lookup

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

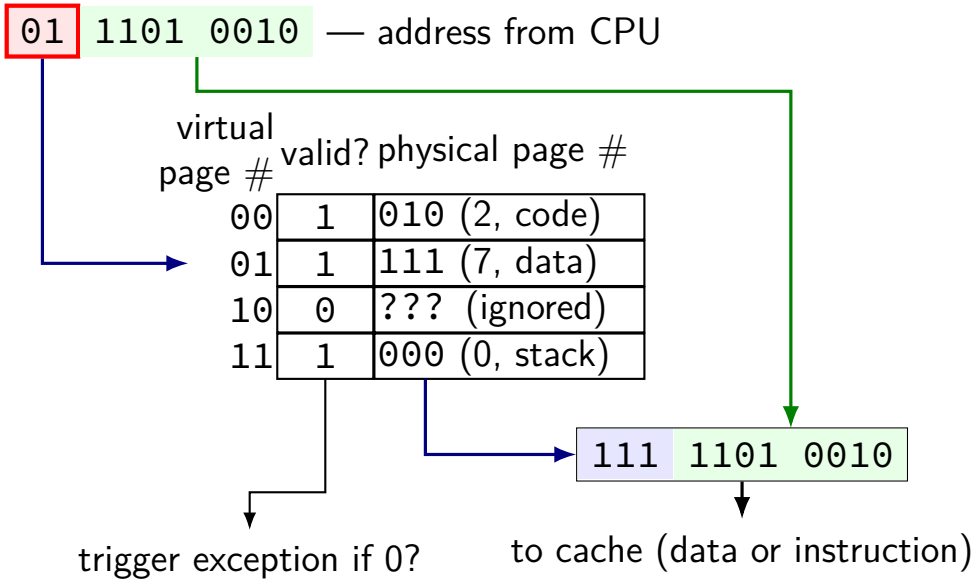


toy page table lookup

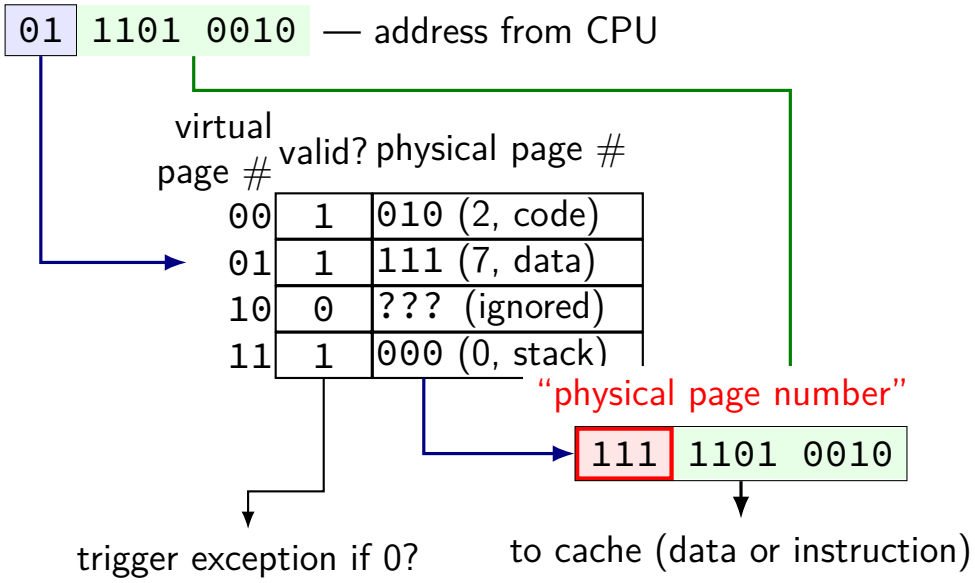


toy page table lookup

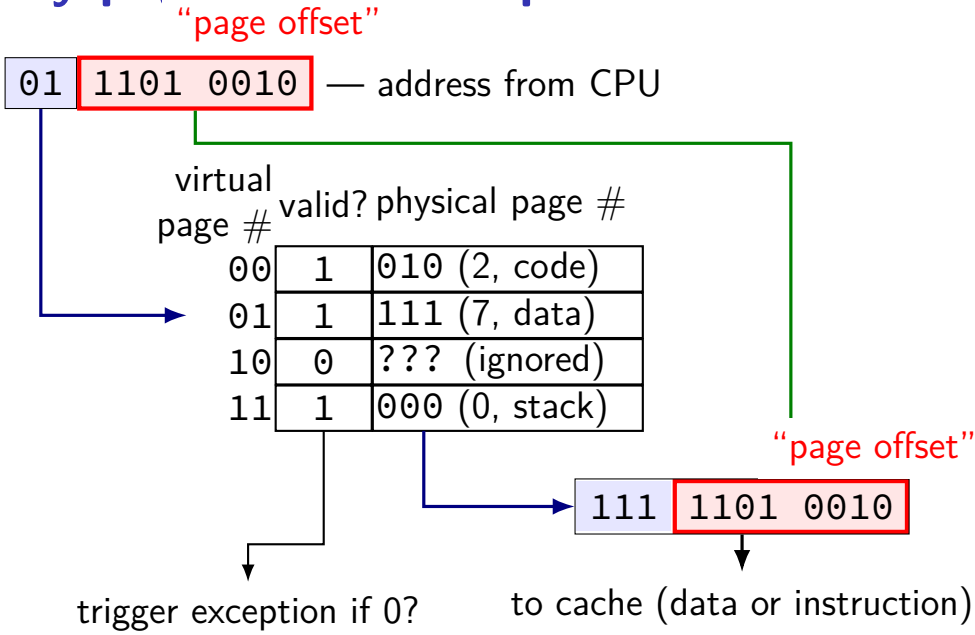
"virtual page number"



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)

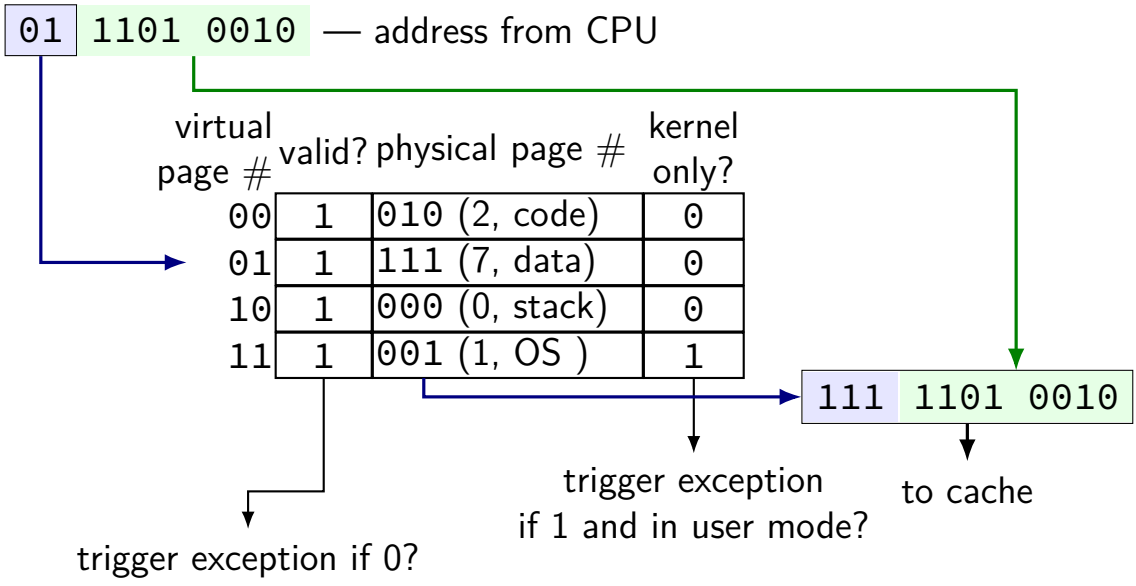
code better not be modified by user program

- otherwise: uncontrolled way to "escape" user mode

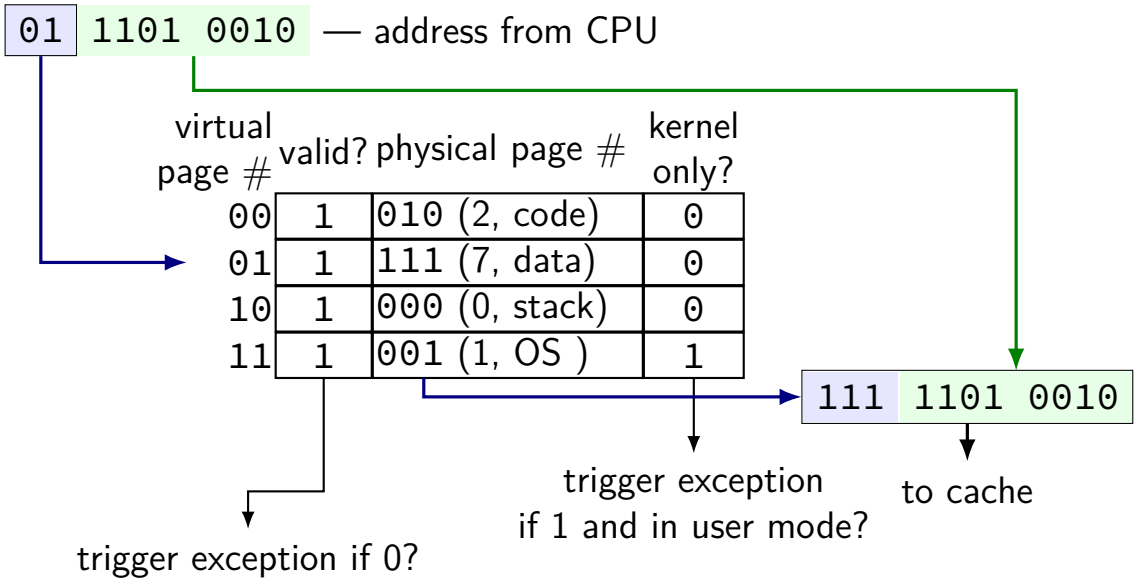
kernel-mode only

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

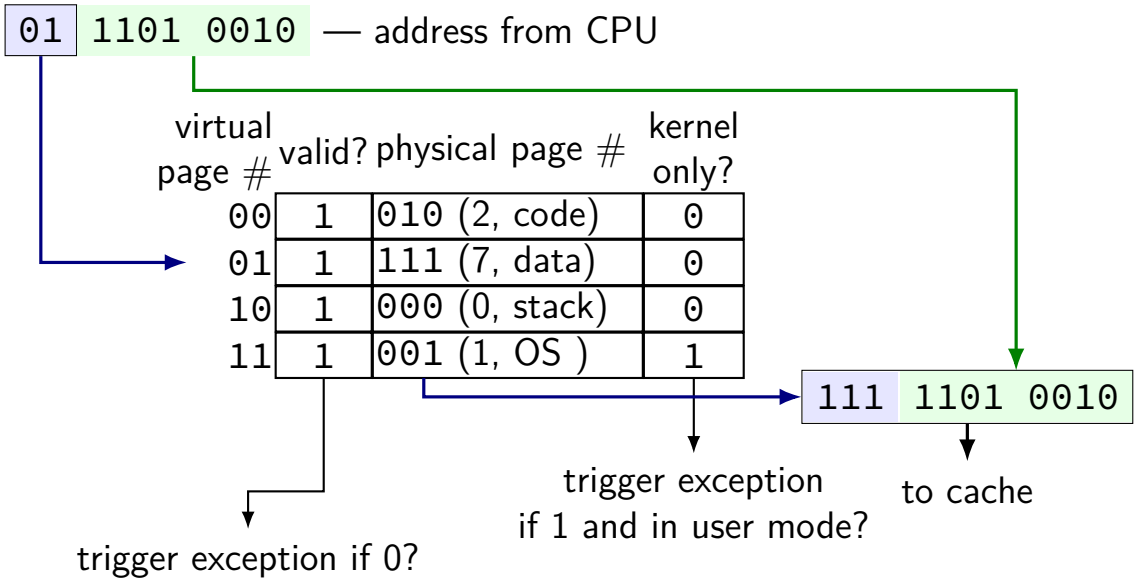
kernel-mode only



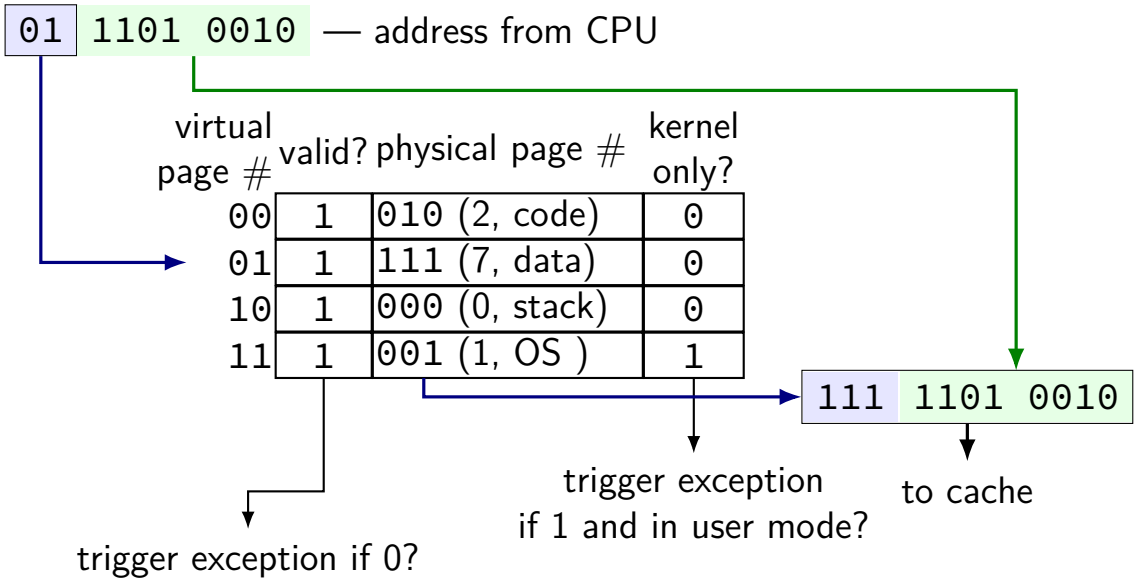
kernel-mode only



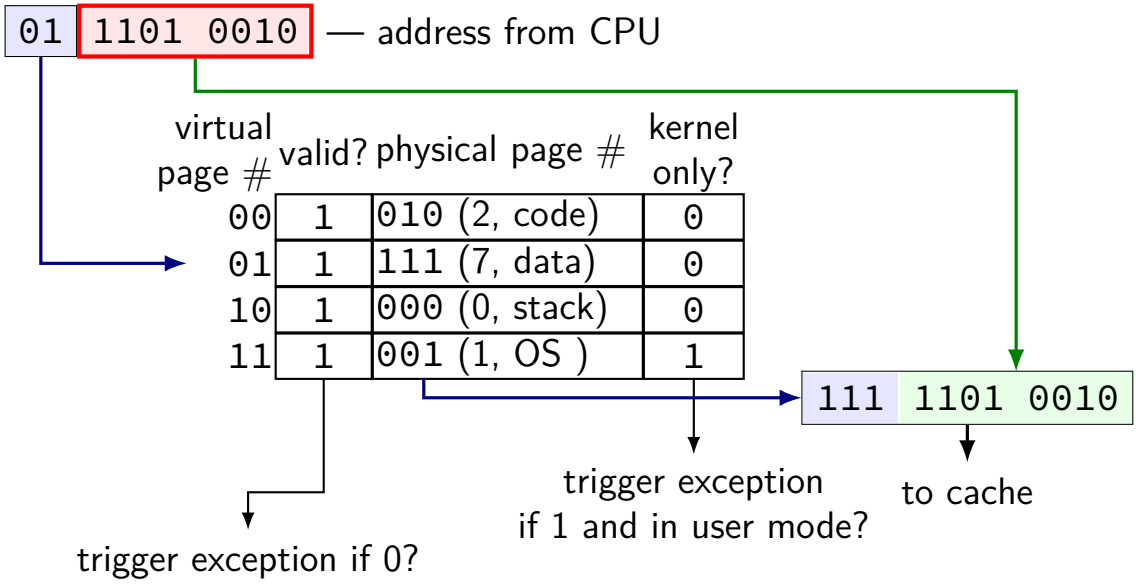
kernel-mode only



kernel-mode only



kernel-mode only



exercise: page counting

suppose 32-bit virtual (program) addresses

and each page is 4096 bytes (2^{12} bytes)

how many virtual pages?

exercise: page counting

suppose 32-bit virtual (program) addresses

and each page is 4096 bytes (2^{12} bytes)

how many virtual pages?

$2^{32}/2^{12} = 2^{20}$

exercise: page table size

suppose 32-bit virtual (program) addresses

suppose 30-bit physical (hardware) addresses

each page is 4096 bytes (2^{12} bytes)

pgae table entries have physical page #, valid bit, kernel-mode bit

how big is the page table (if laid out like ones we've seen)?

exercise: page table size

suppose 32-bit virtual (program) addresses

suppose 30-bit physical (hardware) addresses

each page is 4096 bytes (2^{12} bytes)

pgae table entries have physical page #, valid bit, kernel-mode bit

how big is the page table (if laid out like ones we've seen)?

2^{20} entries $\times(18+2)$ bits per entry

issue: where can we store that?

exercise: address splitting

and each page is 4096 bytes (2^{12} bytes)

split the address `0x12345678` into page number and page offset:

exercise: address splitting

and each page is 4096 bytes (2^{12} bytes)

split the address `0x12345678` into page number and page offset:

page #: `0x12345`; offset: `0x678`

page tables in memory

where can processor store megabytes of page tables? in memory

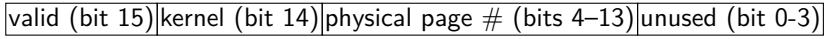
page table entry layout

valid (bit 15)	kernel (bit 14)	physical page # (bits 4–13)	unused (bit 0-3)

page tables in memory

where can processor store megabytes of page tables? in memory

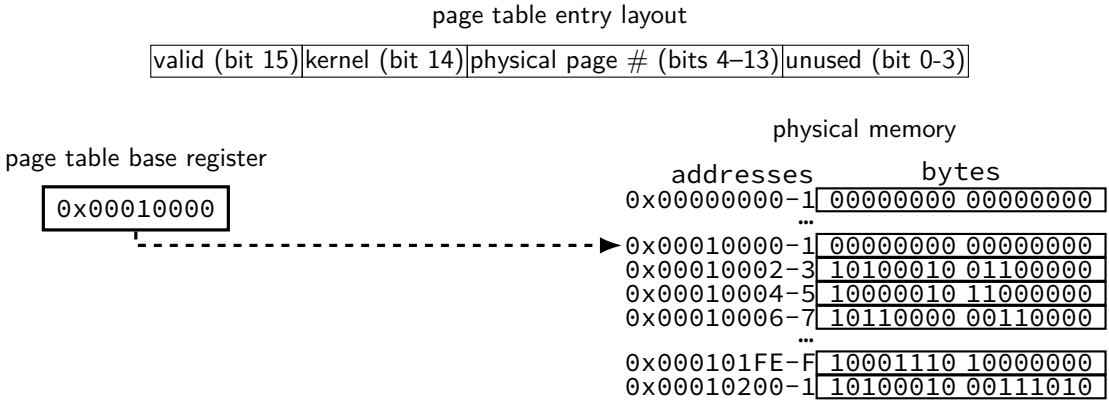
page table entry layout



page table base register

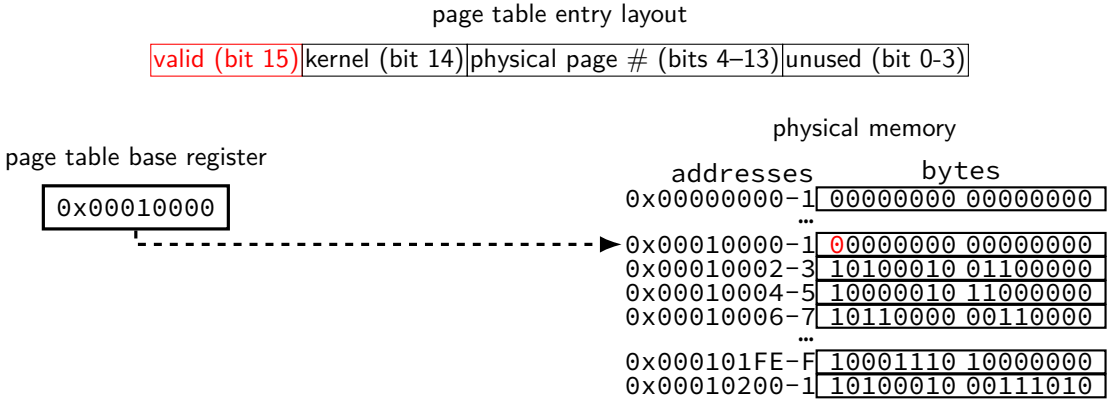
page tables in memory

where can processor store megabytes of page tables? in memory



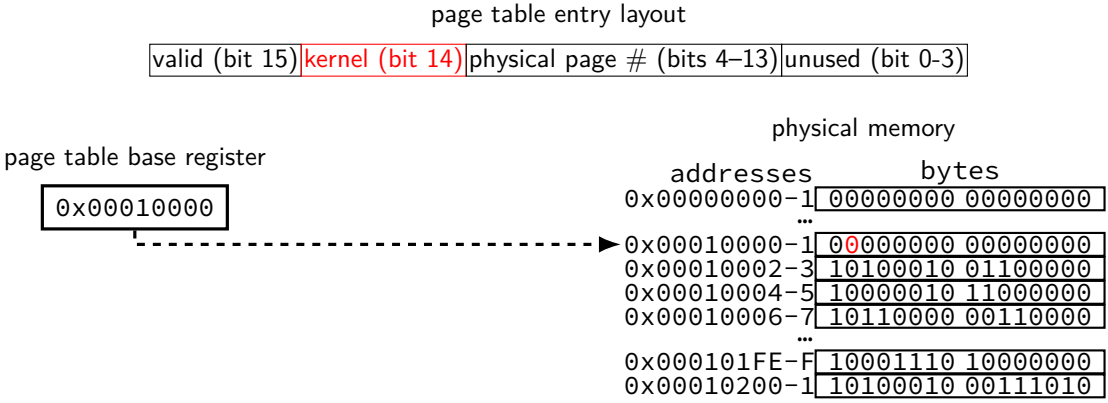
page tables in memory

where can processor store megabytes of page tables? in memory



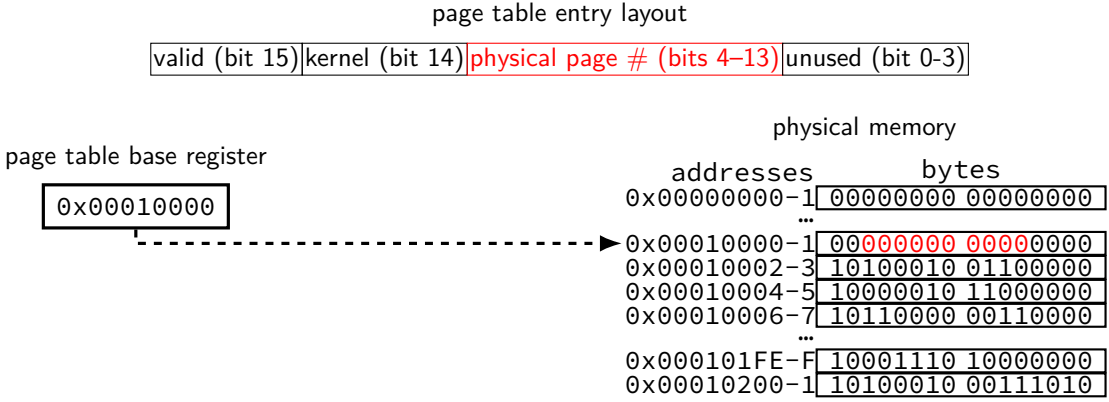
page tables in memory

where can processor store megabytes of page tables? in memory



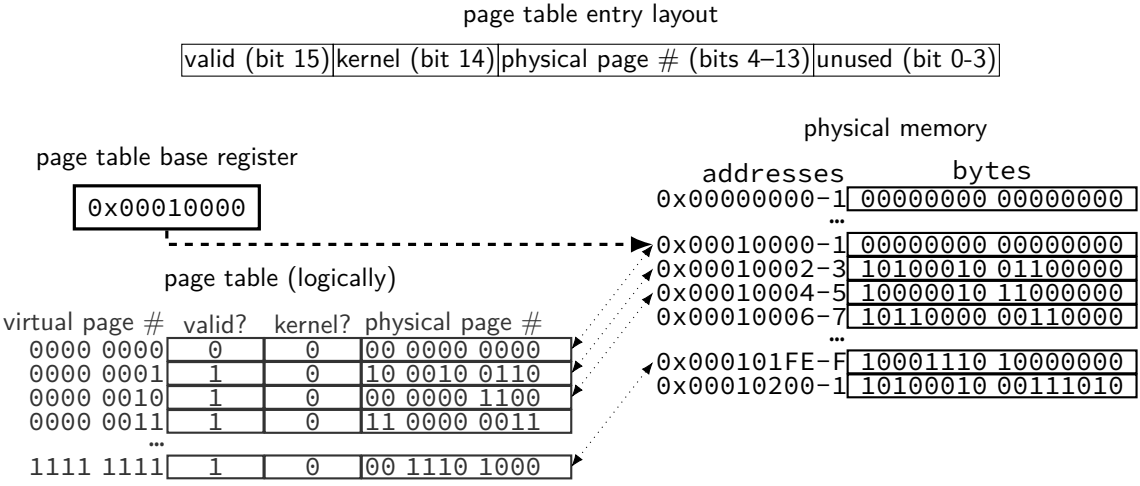
page tables in memory

where can processor store megabytes of page tables? in memory



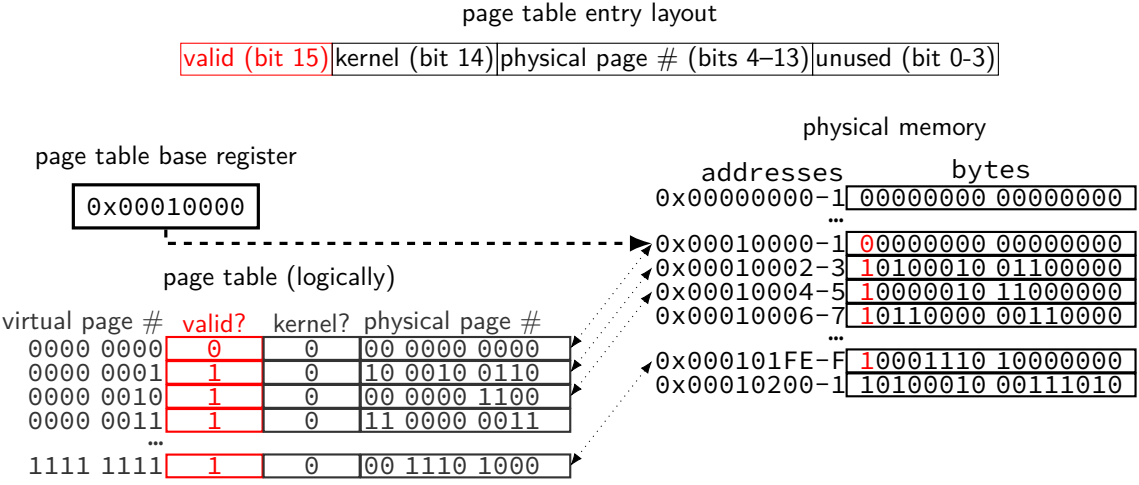
page tables in memory

where can processor store megabytes of page tables? in memory



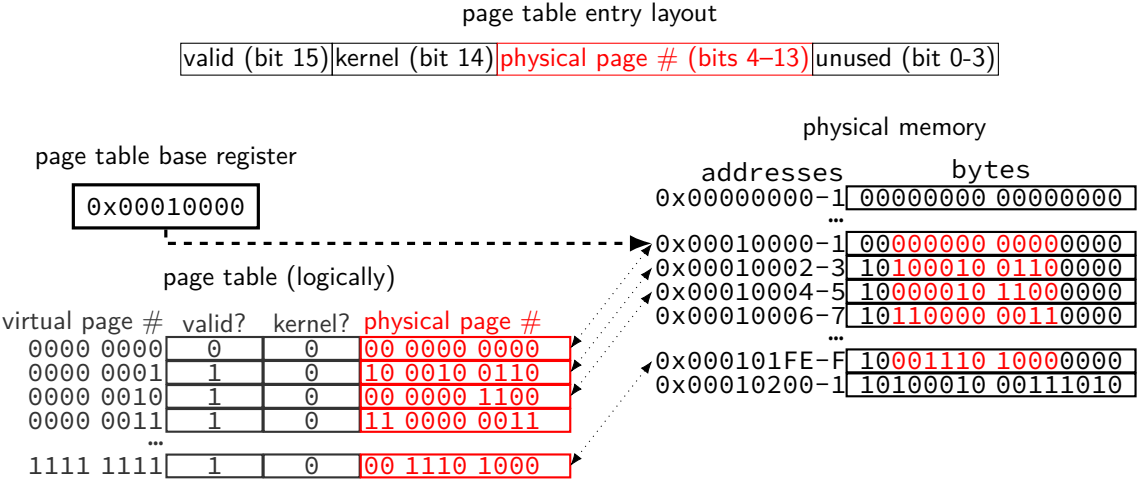
page tables in memory

where can processor store megabytes of page tables? in memory



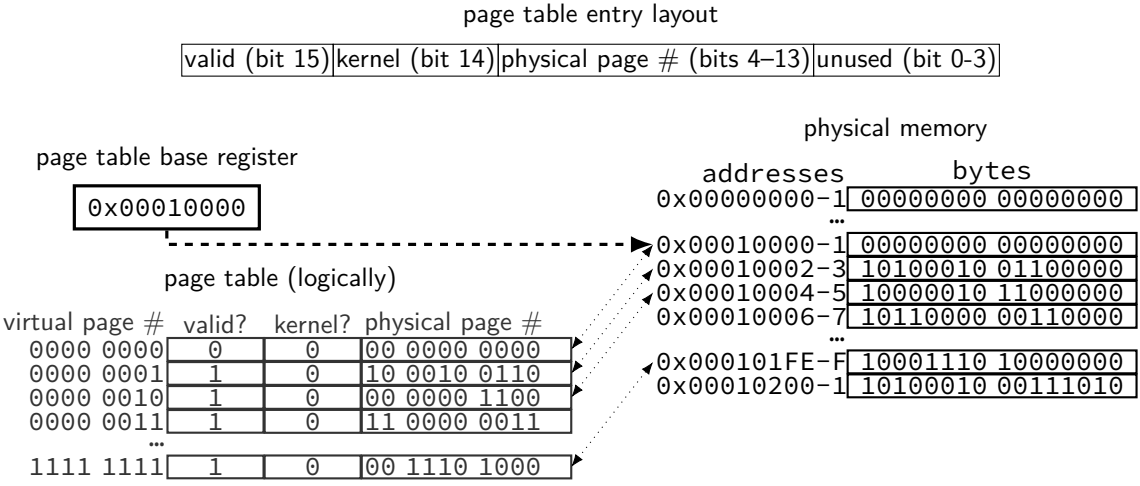
page tables in memory

where can processor store megabytes of page tables? in memory



page tables in memory

where can processor store megabytes of page tables? in memory

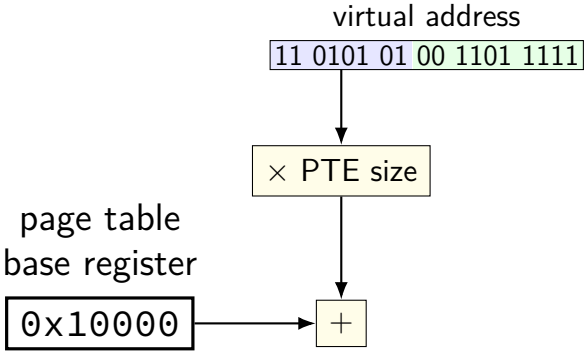


memory access with page table

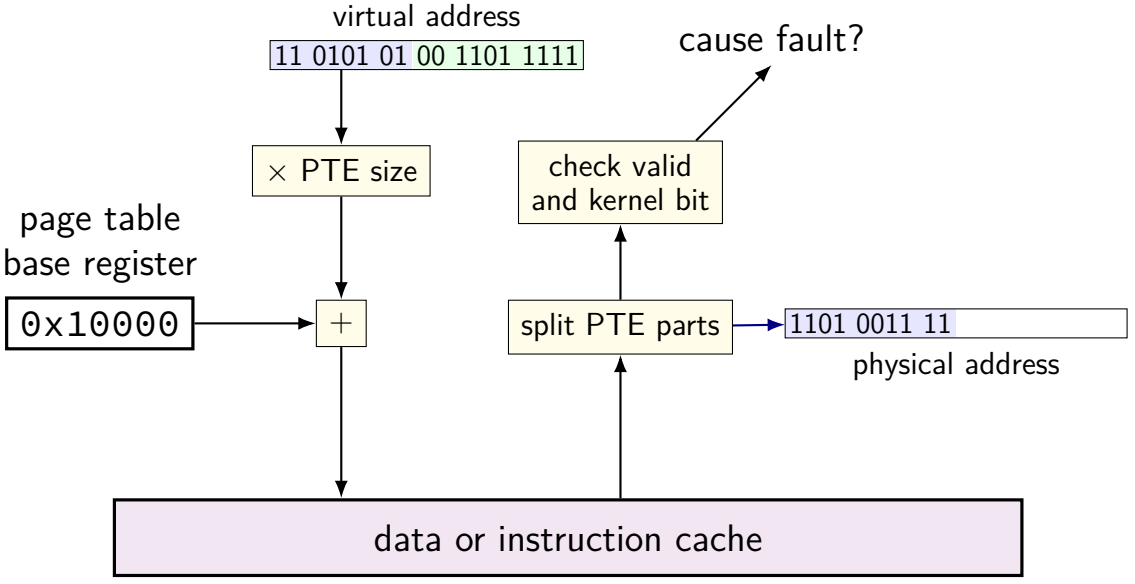
virtual address

11 0101 01 00 1101 1111

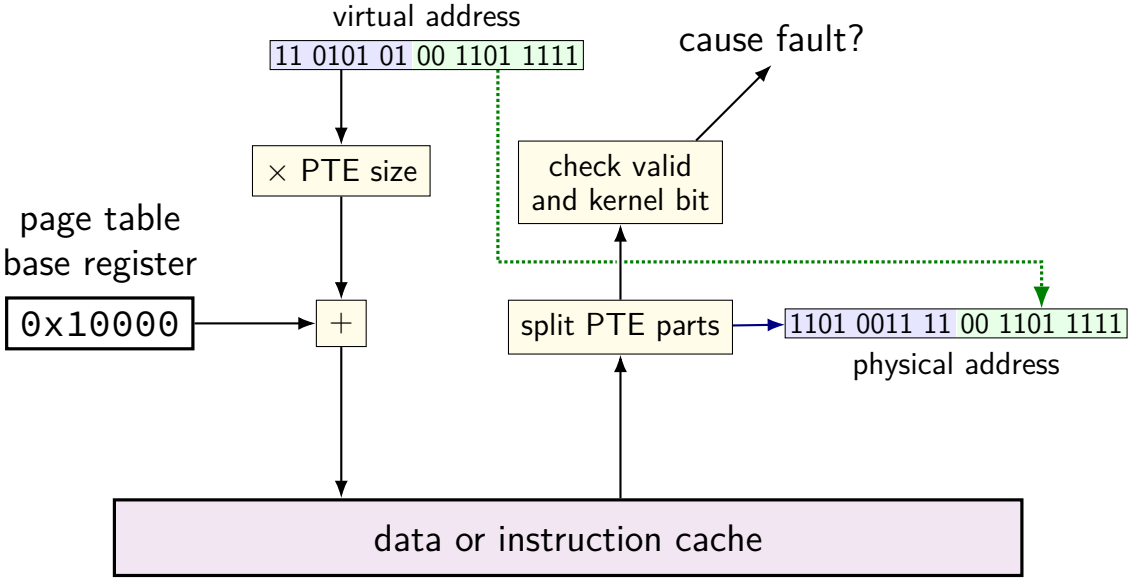
memory access with page table



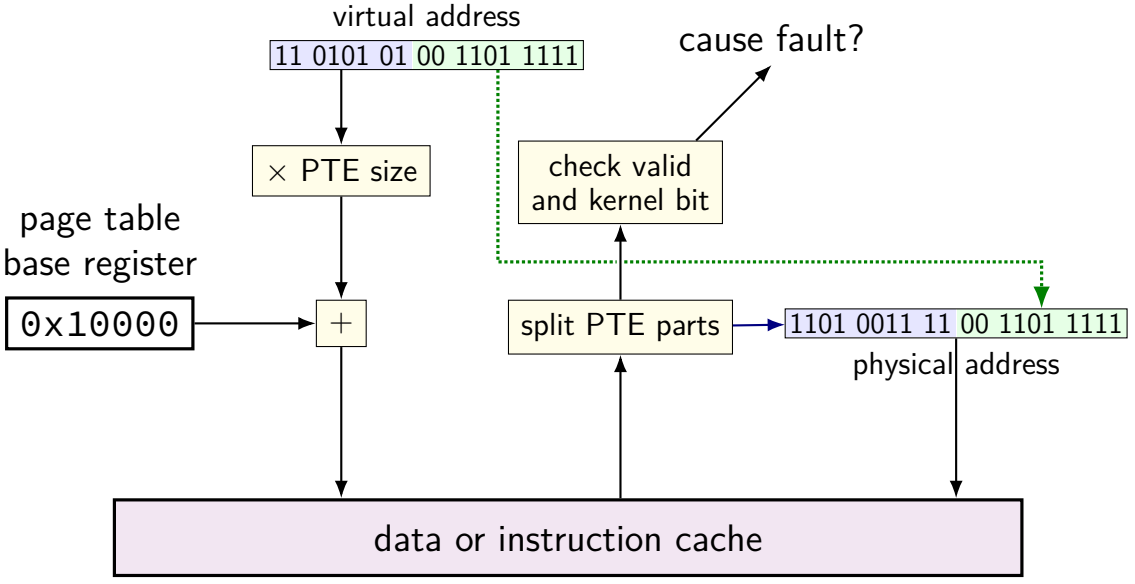
memory access with page table



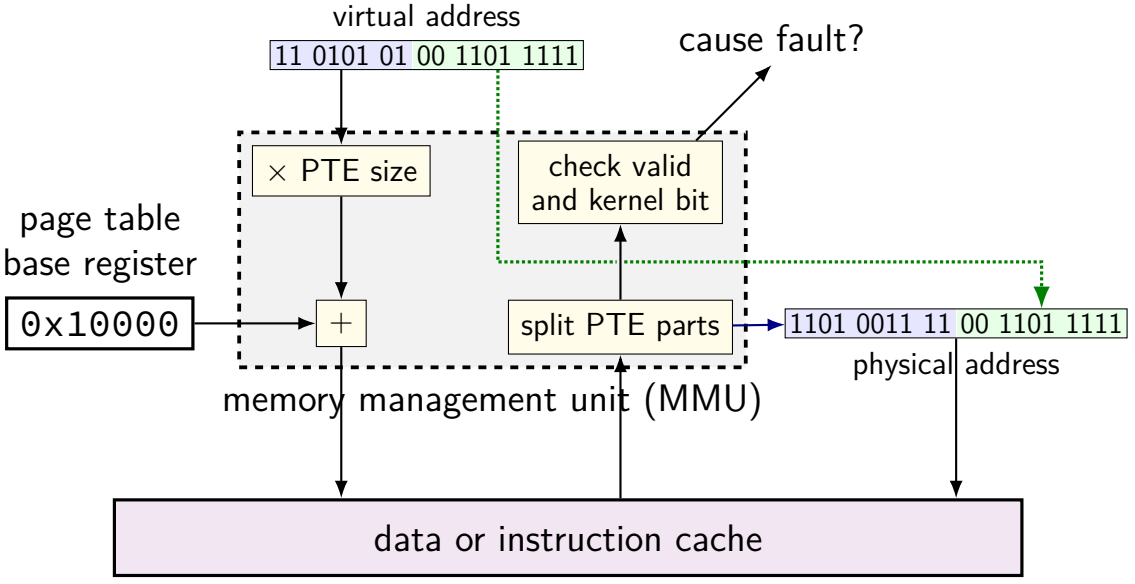
memory access with page table



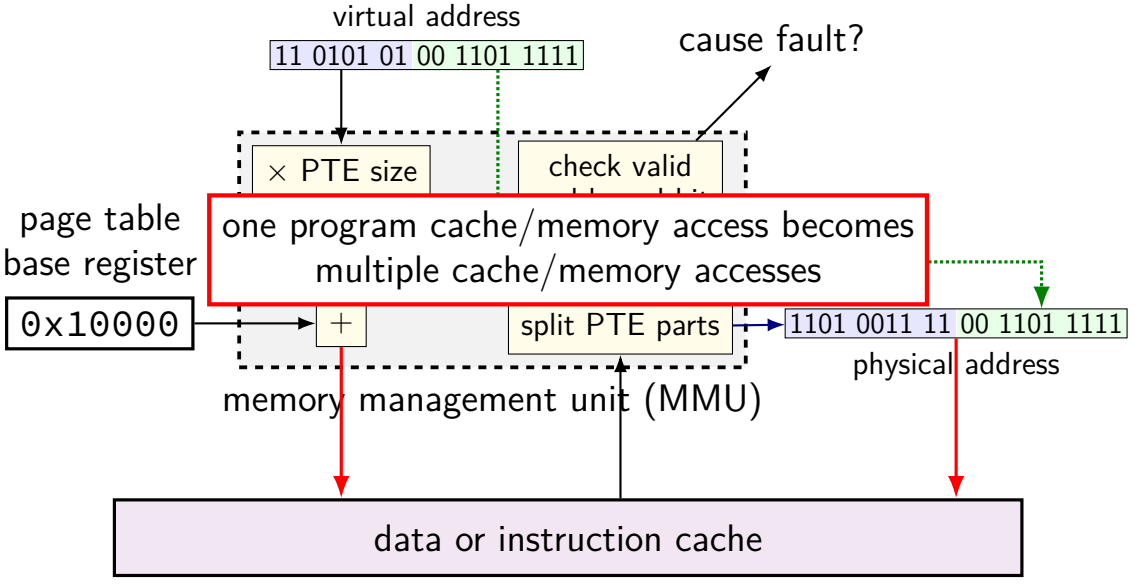
memory access with page table



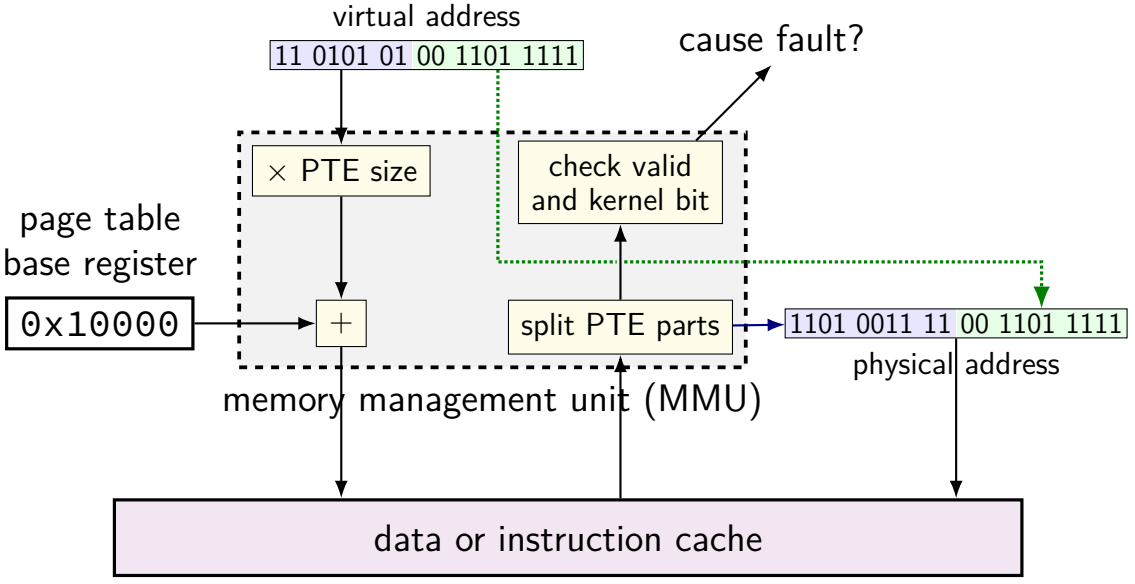
memory access with page table



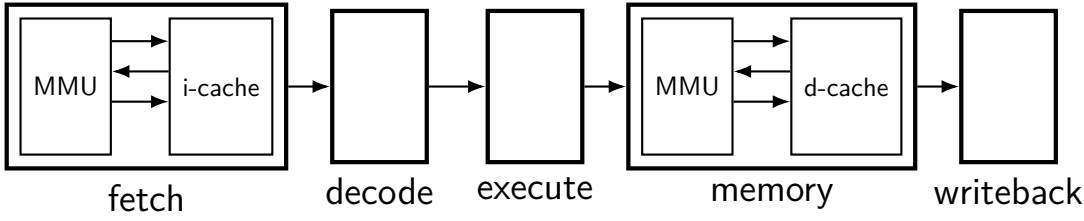
memory access with page table



memory access with page table

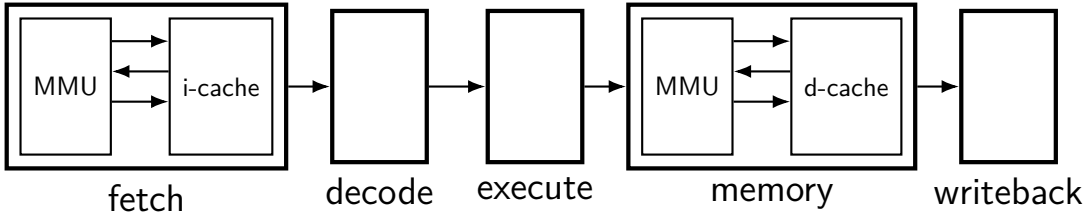


MMUs in the pipeline



up to four memory accesses per instruction

MMUs in the pipeline



up to four memory accesses per instruction

challenging to make this fast (topic for a future date)

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??)

two big problems to solve later

two extra cache accesses seems really slow

solution: more caches (called "TLB", topic for later weeks)

page tables seem really huge

solution: not just a flat table

exercise setup

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	CB 0B CB 0B
0x38–B	DC 0C DC 0C
0x3C–F	EC 0C EC 0C

exercise setup

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

phys. page 0

phys. page 1

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	CB 0B CB 0B
0x38–B	DC 0C DC 0C
0x3C–F	EC 0C EC 0C

exercise

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

(virtual addresses) `0x18` = ???; `0x03` = ???; `0x0A` = ???; `0x13` = ???

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	CB 0B CB 0B
0x38–B	DC 0C DC 0C
0x3C–F	EC 0C EC 0C

exercise

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

(virtual addresses) `0x18` = `00`; `0x03` = ???; `0x0A` = ???; `0x13` = ???

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	CB 0B CB 0B
0x38–B	DC 0C DC 0C
0x3C–F	EC 0C EC 0C

exercise

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

(virtual addresses) `0x18` = `00`; `0x03` = 0x4A; `0x0A` = ???; `0x13` = ???

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	CB 0B CB 0B
0x38–B	DC 0C DC 0C
0x3C–F	EC 0C EC 0C

exercise

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

(virtual addresses) `0x18` = `00`; `0x03` = 0x4A; `0x0A` = `0xDC`; `0x13` = ???

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	CB 0B CB 0B
0x38–B	DC 0C DC 0C
0x3C–F	EC 0C EC 0C

exercise

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

(virtual addresses) `0x18` = `00`; `0x03` = 0x4A; `0x0A` = `0xDC`; `0x13` = fault

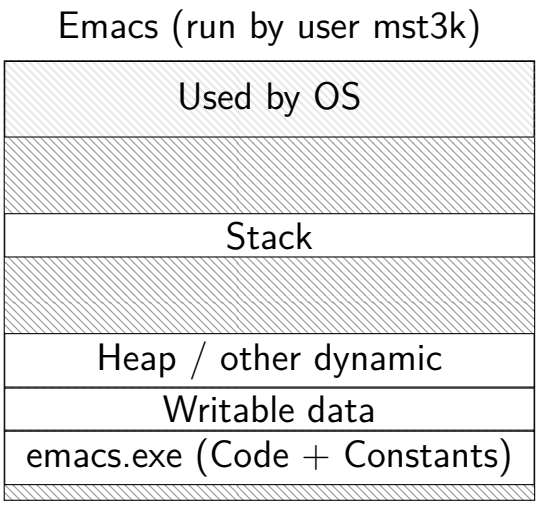
page table

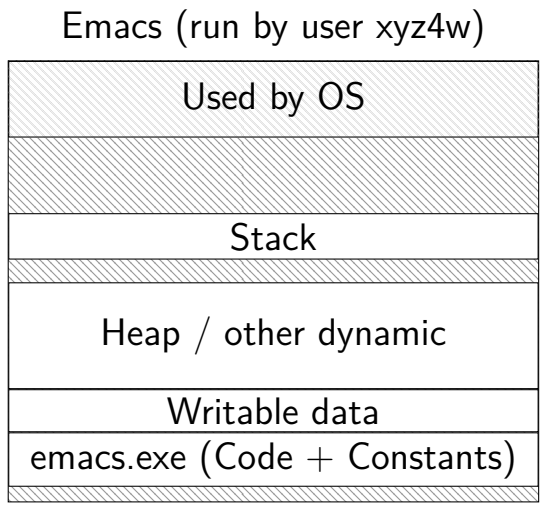
virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

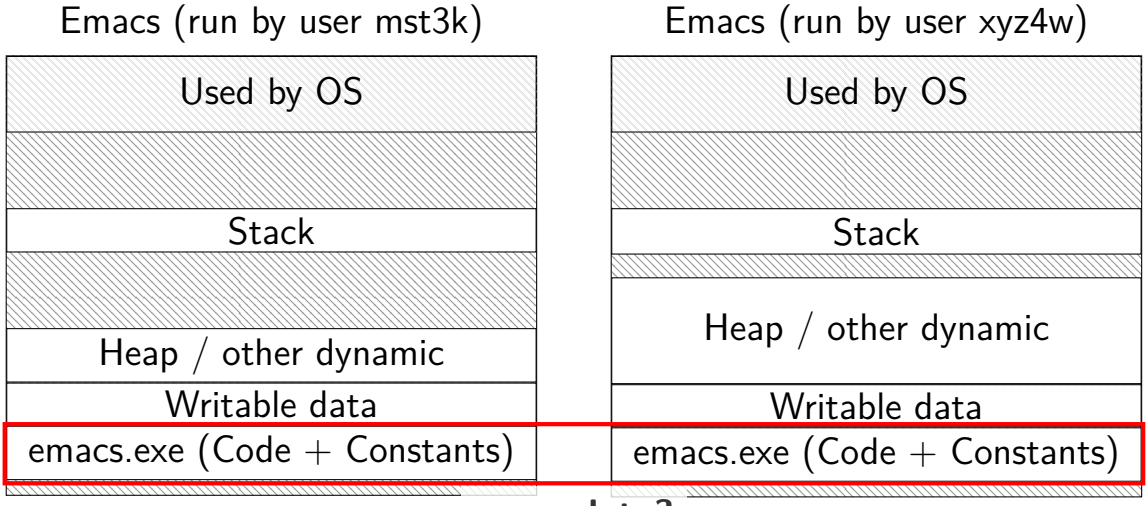
physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	CB 0B CB 0B
0x38–B	DC 0C DC 0C
0x3C–F	EC 0C EC 0C

emacs (two copies)





emacs (two copies)



two copies of program

would like to only have one copy of program

what if `mst3k`'s emacs tries to modify its code?

would break process abstraction:

> "illusion of own memory"

typical page table entries

solution: same idea as kernel-only bit

page table entry will have more permissions bits

- can read?
- can write?
- can execute?

checked by MMU like valid/kernel bit

page table (logically)

virtual page #	valid?	kernel?	write?	exec?	physical page #
0000 0000	0	0	0	0	00 0000 0000
0000 0001	1	0	1	0	10 0010 0110
0000 0010	1	0	1	0	00 0000 1100
0000 0011	1	0	0	1	11 0000 0011
...					
1111 1111	1	0	1	0	00 1110 1000

shared libraries

C standard library has lots of common code:

```
printf
fopen
qsort
...
```

would like to not have multiple copies

solution: multiple virtual read-only copies, one physical copy

efficient shared libraries

recall: linking

- placeholders to fill in with addresses
- need to change depending on where library ends up?

different code depending on where library ends up?

one solution: position-independent code

position-independent code: one idea

Y86 encoding for jump: include target address

alternate encoding: include target address - PC

"relative address"

handles jumps within standard library

real shared libraries

Linux tool `ldd` — what shared libraries does this program use:

```
$ ldd /bin/ls
        linux-vdso.so.1 =>  (0x00007ffebbd49000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so
            (0x00007f4fac17c000)
        libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1
            (0x00007f4fabf74000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
            (0x00007f4fabbab000)
        libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3
            (0x00007f4fab96d000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2
            (0x00007f4fab769000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f4fac39f000)
        libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1
            (0x00007f4fab564000)
```

position-independent code: indirection

what about code between libraries?

what about changing the library code?

use indirection:

```
/* instead of: */
    call printf
/* use something like: */
    relative_movq LOOKUP_TABLE + 100, %rax
        // x86-64 syntax: movq LOOKUP_TABLE+100(%rip), %rax
    call *%rax
```

populate lookup table for each function used

- lookup table not shared between programs

add version of move instruction that uses relative address

page tables

program memory = virtual; real memory = physical

each memory divided into fixed-sizes pages

each virtual page has a page table entry:

- has location of physical page (if any)
- has valid bit
- has permission bits

all page table entries stored in page table

permission or validity error triggers exception

page table tricks

page tables let operating systems do ‘magic’ with memory

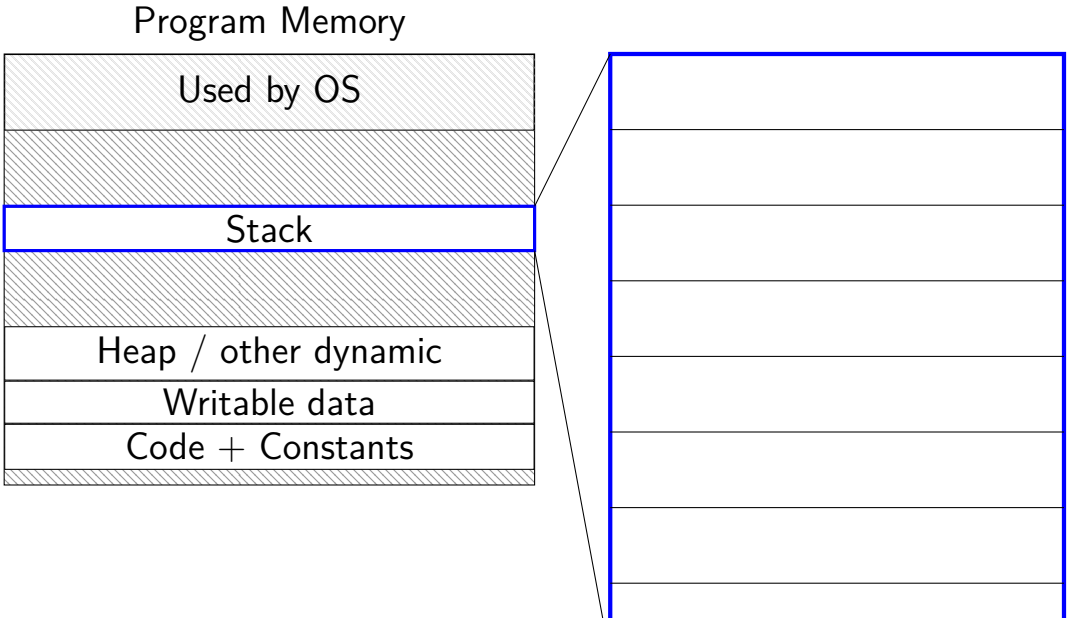
key idea: OS doesn’t need to crash on a fault!

instead: change page tables and rerun memory access

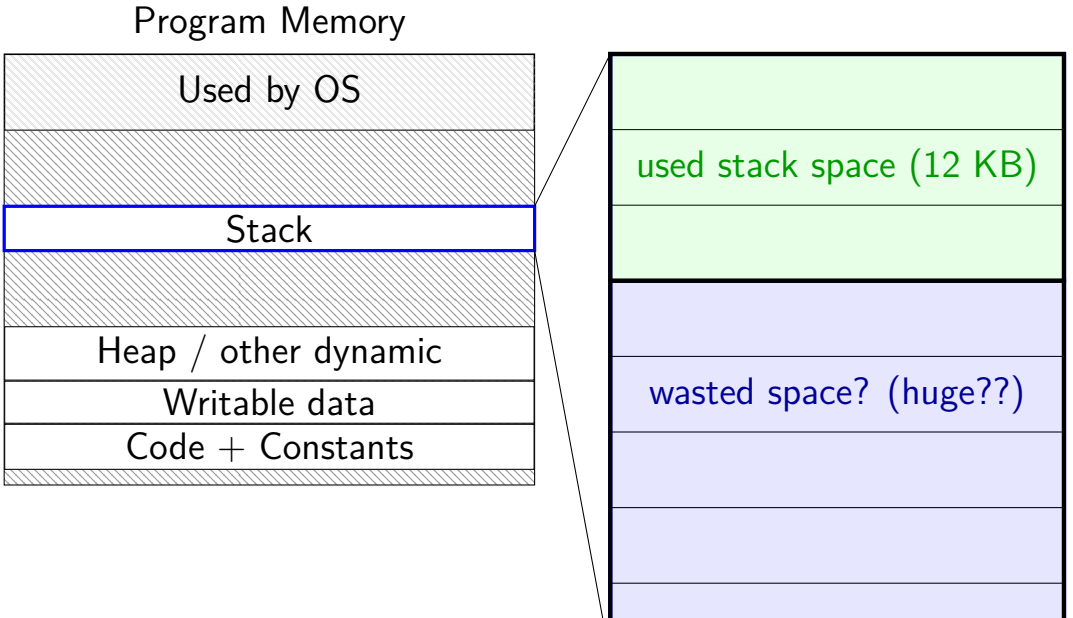
- similar idea to trap-and-emulate

- will talk about these in future weeks
- today: what makes paging useful

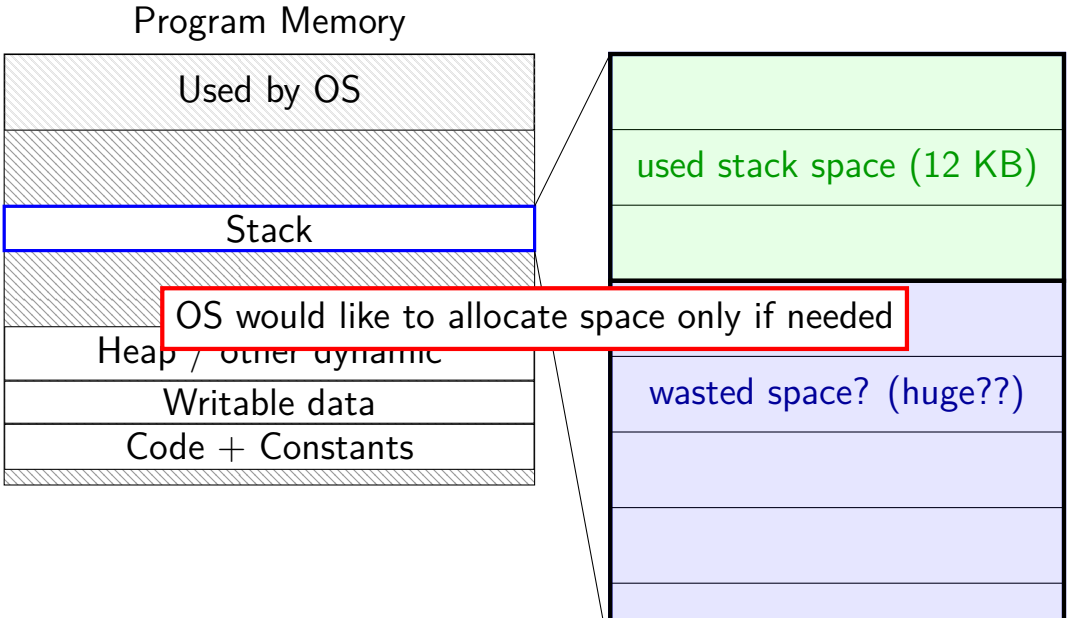
space on demand



space on demand



space on demand



allocating space on demand

`%rsp = 0x7FFFC000`

```
...
// requires more stack space
A: pushq %rbx
B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

VPN	valid?	physical page
…	…	…
0x7FFFB	0	---
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
…	…	…

allocating space on demand

`%rsp = 0x7FFFC000`

```
...
// requires more stack space
A: pushq %rbx
B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

VPN	valid?	physical page
…	…	…
0x7FFFB	0	---
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
…	…	…

pushq triggers exception
hardware says "accessing address `0x7FFFBFF8`"
OS looks up what's should be there — "stack"

allocating space on demand

`%rsp = 0x7FFFC000`

```
...
// requires more stack space
A: pushq %rbx
B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

VPN	valid?	physical page
…	…	…
0x7FFFB	1	0x200D8
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
…	…	…

in exception handler, OS allocates more stack space
OS updates the page table
then returns to retry the instruction

allocating space on demand

note: the space doesn't have to be initially empty

only change: load from file, etc. instead of allocating empty page

loading program can be merely creating empty page table

everything else can be handled in response to page faults

- no time/space spent loading/allocating unneeded space

page tricks generally

deliberately make program trigger page/protection fault

but don't assume page/protection fault is an error

have seperate data structures represent logically allocated memory

> e.g. "addresses `0x7FFF8000` to `0x7FFFFFFF` are the stack"
> might talk about Linux data structures later (book section 9.7)

page table is for the hardware and not the OS

hardware help for page table tricks

information about the address causing the fault

- e.g. special register with memory address accessed
- harder alternative: OS disassembles instruction, look at registers

(by default) rerun faulting instruction when returning from exception

precise exceptions: no side effects from faulting instruction or after

- e.g. `pushq` that caused did not change `%rsp` before fault
- e.g. instructions reordered after faulting instruction not visible