changelog

1-level exercise (2): fix off-by-one error in looking up 0x22 in table of memory contents

last time

virtual addresses $\rightarrow$ physical addresses

dividing memory into 2^X byte pages

addresses split into page number and X byte offset

table:

- row number = virtual page number
- data in row = valid bit, physical page number

virtual to physical translation with table

- lookup virtual page number to get physical
- check valid bit
- keep page offset the same

table sizes

schedule changes

possibly slightly better at keeping related topics together

better at covering topics before needed in assignments, ...

and having less stuff around Thanksgiving break

but probably some cases where several weeks until lab/assignment on what was in lecture

will see where we are after this lecture re: lab next week

anonymous feedback (1)

"I don't think it's fair to take off points for quiz 3 #4 since the correct answer wasn't even an option. On top of the incorrect answer options, the question itself was poorly worded and confusing. I think everybody should just get the points for that question."

Yes, I wish I had written the question better
Hard for questions with "none of the above — explain" to not have a correct answer as option

(and a lot of those answers should get credit after graders get to them)

think most common misunderstandings are:

not thinking about owner permission at all
thinking the owner of the file is not controllable
thinking users shouldn't be able to run any programs because of this ACL

anonymous feedback (2)

"Could you explain what an offset is and what it does? How does that differ from the number of pages

address = [page number (index)][page offset]
page number $\sim$ which page of a book
number of pages = number of possible page numbers
page offset $\sim$ where to start looking on that page

why —
for setting up programs, we want to work with big pages
...but programs want to work with small bytes

anonymous feedback (3)

"Could you please review this statement from the VM reading in the 2.1 section, "instead, the operating system uses the segments to create hardware-visible page table entries and to react to hardware-generated, page-related faults and potentially convert them into signals to convey to the user process." I understand the process of creating the page table entries but I don't understand anything after that regarding the reaction to the hardware-generated, page related faults. Thank you!

> in lecture, we're covering virtual memory in a different order
> faults = kind of exception
> in particular, exceptions about out-of-bounds memory accesses
> operating system uses its bookkeeping (segments) to determine what to do
> trick we'll learn later: can 'fix' segfault
> edit page table so segfault won't happen + retry

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
	1 D2 D3
	5 D6 D7
	A AB BC
	E 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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

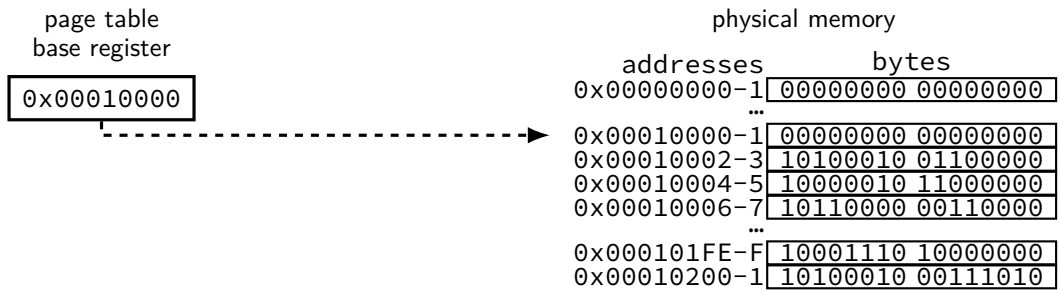
valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)

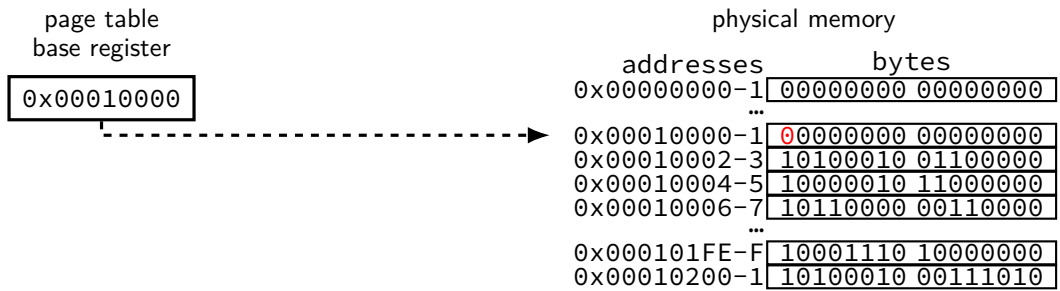


page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)

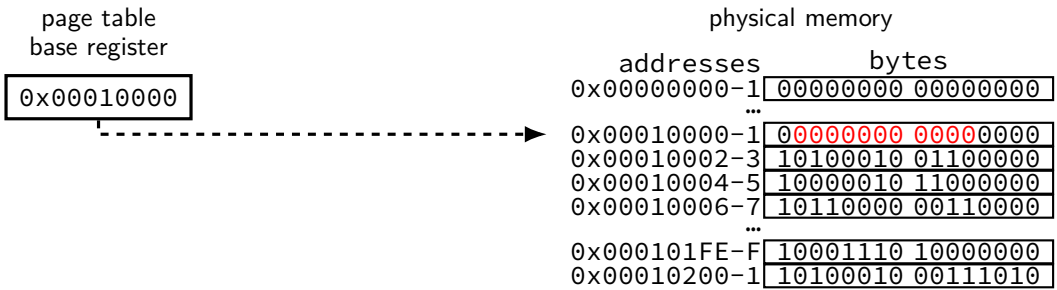


page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)

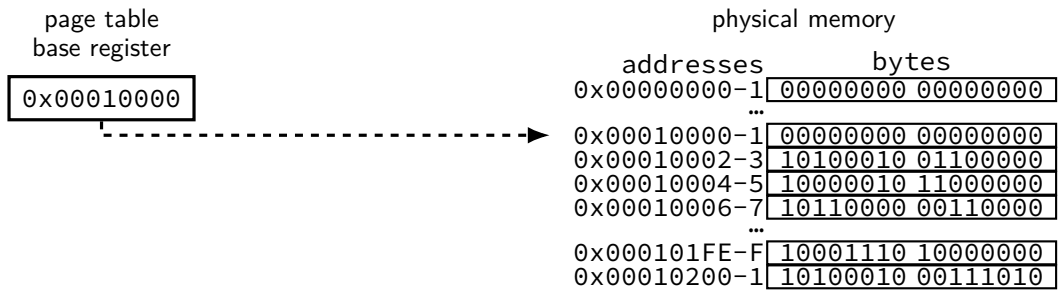


page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)

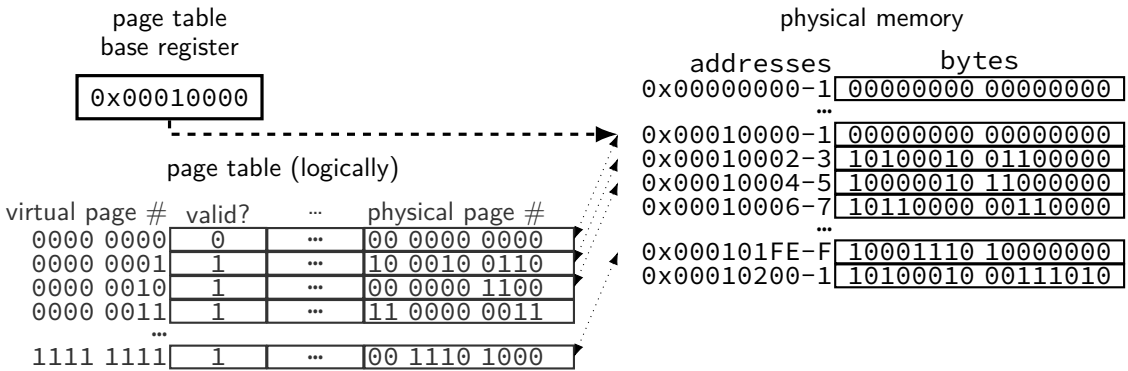


page tables in memory

where can processor store megabytes of page tables? in memory

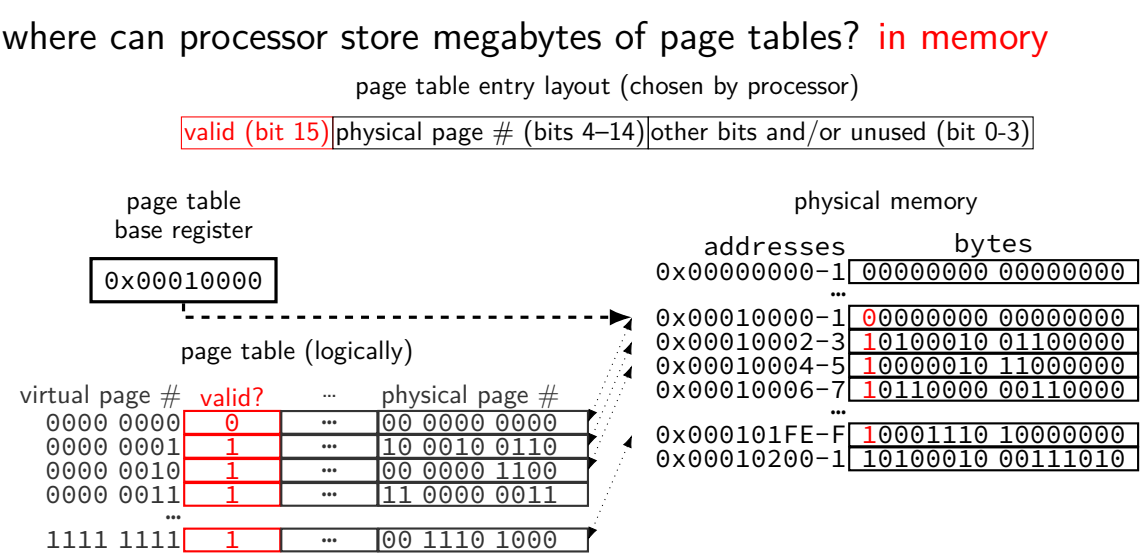
page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)



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 table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)

page table base register

0x00010000

page table (logically)

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

physical memory

addresses	bytes
0x00000000-1	00000000 00000000
...	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
...	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)

page table base register

0x00010000

page table (logically)

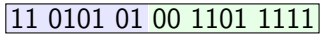
virtual page #	valid?	...	physical page #
0000 0000	0	...	00 0000 0000
0000 0001	1	...	10 0010 0110
0000 0010	1	...	00 0000 1100
0000 0011	1	...	11 0000 0011
...			
1111 1111	1	...	00 1110 1000

physical memory

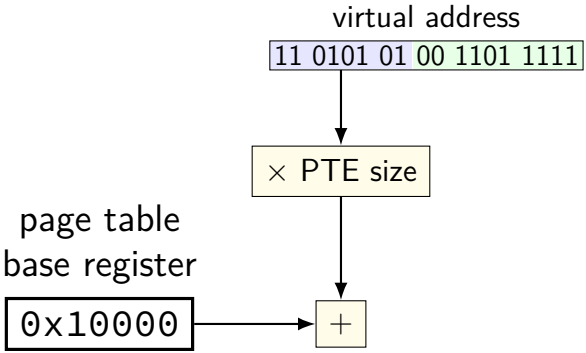
addresses	bytes
0x00000000-1	00000000 00000000
...	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
...	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

memory access with page table

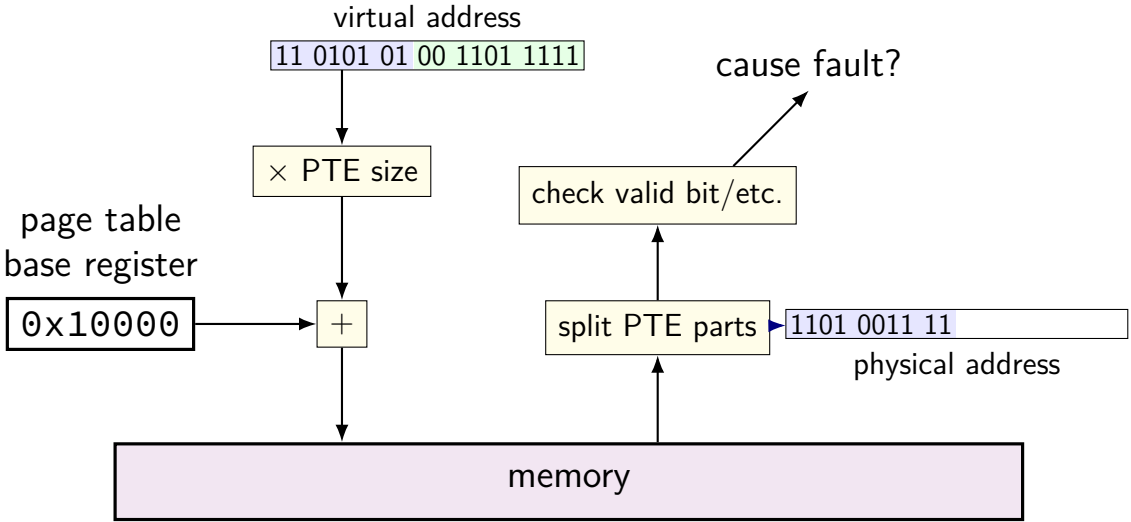
virtual address



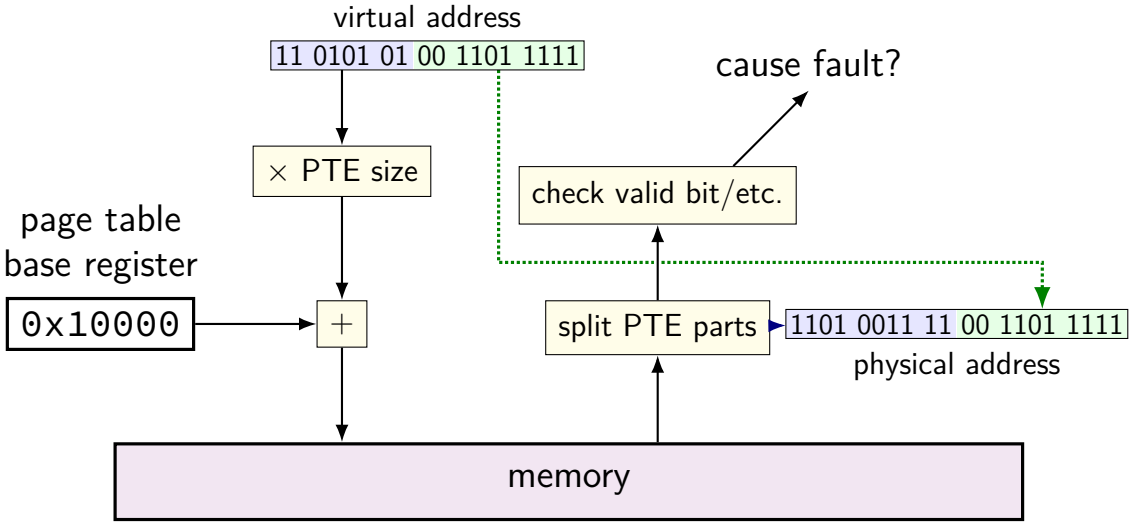
memory access with page table



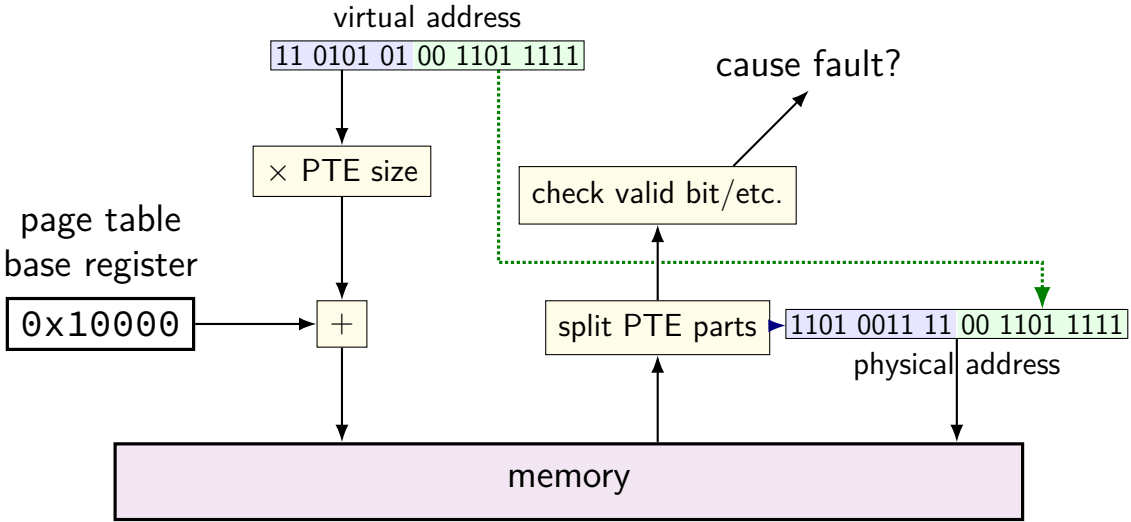
memory access with page table



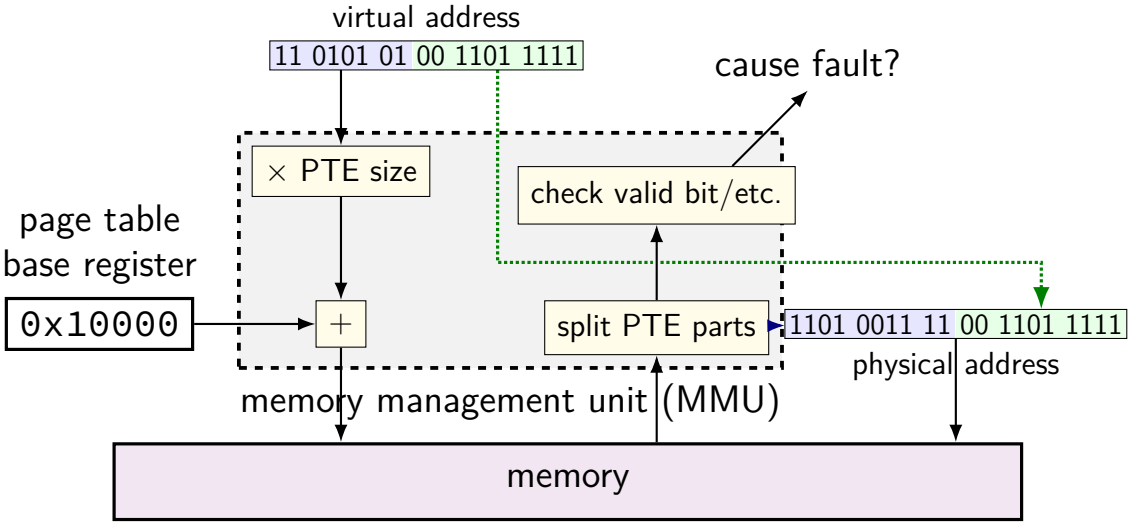
memory access with page table



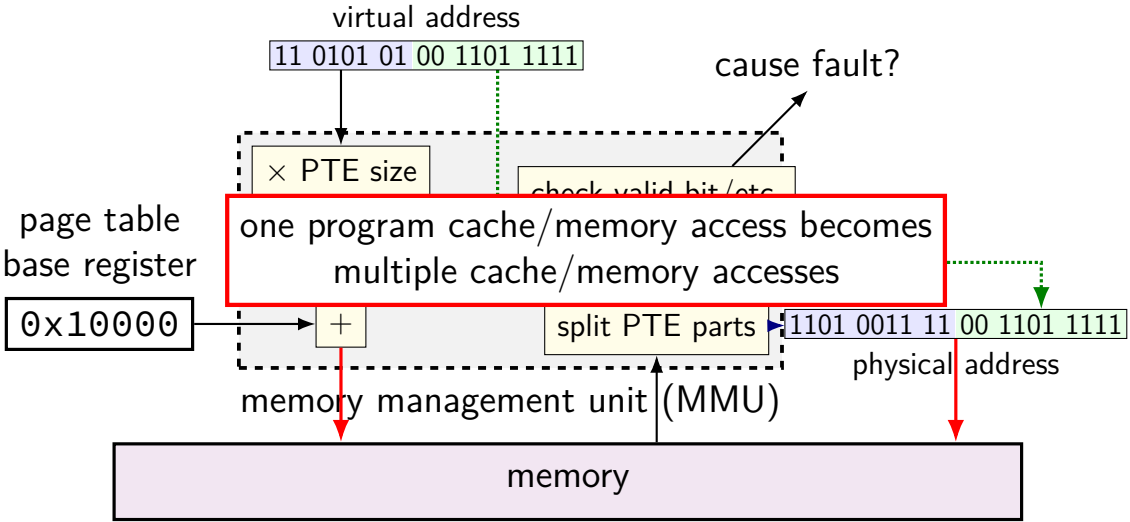
memory access with page table



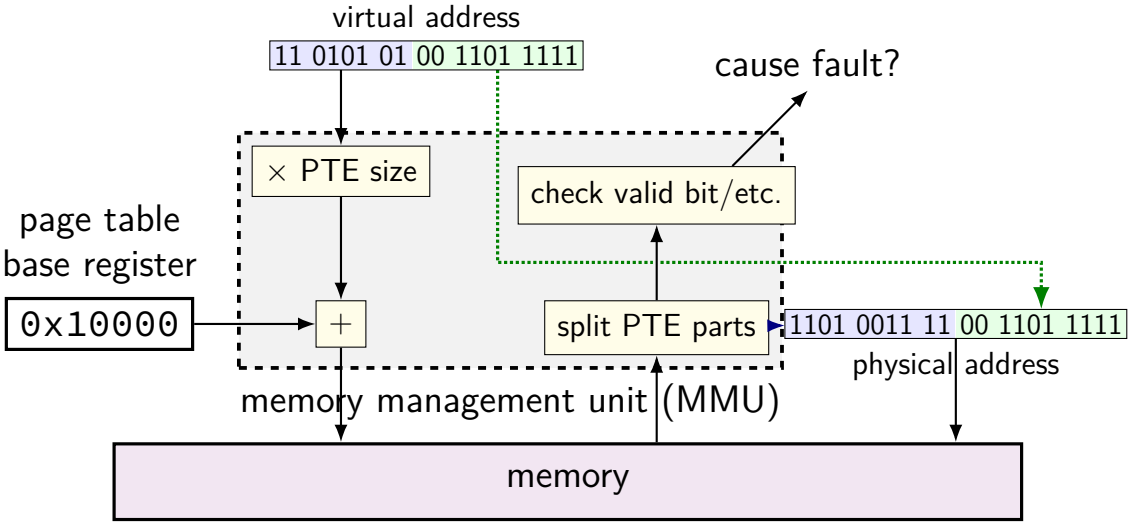
memory access with page table



memory access with page table



memory access with page table



1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;

page table base register `0x20`; translate virtual address `0x31`

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	E4 E5 F6 07
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

1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;

page table base register 0x20; translate virtual address 0x31

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	E4 E5 F6 07
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

0x31 = 11 0001

PTE addr:

0x20 + 110 ×1 = 0x26

PTE value:

0xF6 = 1111 0110

PPN 111, valid 1

M[111 001] = M[0x39]

→ 0x0C

1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;

page table base register 0x20; translate virtual address 0x31

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	E4 E5 F6 07
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

0x31 = 11 0001

PTE addr:

0x20 + 110 ×1 = 0x26

PTE value:

0xF6 = 1111 0110

PPN 111, valid 1

M[111 001] = M[0x39]

→ 0x0C

1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;

page table base register 0x20; translate virtual address 0x31

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	E4 E5 F6 07
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

0x31 = 11 0001

PTE addr:

0x20 + 110 ×1 = 0x26

PTE value:

0xF6 = 1111 0110

PPN 111, valid 1

M[111 001] = M[0x39]

→ 0x0C

1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;

page table base register 0x20; translate virtual address 0x31

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	E4 E5 F6 07
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

0x31 = 11 0001

PTE addr:

0x20 + 110 ×1 = 0x26

PTE value:

0xF6 = 1111 0110

PPN 111, valid 1

M[111 001] = M[0x39]

→ 0x0C

1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other

page table base register `0x20`; translate virtual address `0x12`

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	A0 E2 D1 F3
0x24–7	E4 E5 F6 07
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

1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other

page table base register 0x20; translate virtual address 0x12

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	A0 E2 D1 F3
0x24–7	E4 E5 F6 07
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

0x12 = 01 0010

PTE addr:

0x20 + 2 ×1 = 0x22

PTE value:

0xD1 = 1101 0001

PPN 110, valid 1

M[110 001] = **M[**0x32**]**

→ 0xBA

1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other

page table base register 0x20; translate virtual address 0x12

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	A0 E2 D1 F3
0x24–7	E4 E5 F6 07
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

0x12 = 01 0010

PTE addr:

0x20 + 2 ×1 = 0x22

PTE value:

0xD1 = 1101 0001

PPN 110, valid 1

M[110 001] = **M[**0x32**]**

→ 0xBA

1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other

page table base register 0x20; translate virtual address 0x12

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	A0 E2 D1 F3
0x24–7	E4 E5 F6 07
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

0x12 = 01 0010

PTE addr:

0x20 + 2 ×1 = 0x22

PTE value:

0xD1 = 1101 0001

PPN 110, valid 1

M[110 001] = **M[**0x32**]**

→ 0xBA

1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other

page table base register `0x20`; translate virtual address `0x12`

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	A0 E2 D1 F3
0x24–7	E4 E5 F6 07
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

$0x12 = 01\ 0010$

PTE addr:

$0x20 + 2 \times 1 = 0x22$

PTE value:

$0xD1 = 1101\ 0001$

PPN 110, valid 1

$M[110\ 001] = \mathbf{M[0x32]}$

$\rightarrow$ 0xBA

pagetable assignment

pagetable assignment

simulate page tables (on top of normal program memory)

> alternately: implement another layer of page tables
> on top of the existing system's

in assignment:

virtual address $\sim$ arguments to your functions

physical address $\sim$ your program addresses (normal pointers)

pagetable assignment API

```
/* configuration parameters */
#define POBITS ...
#define LEVELS /* later /
```

```
size_t ptbr; // page table base register
    // points to page table (array of page table entries)

// lookup "virtual" address 'va' in page table ptbr points to
// return (void*) (~0L) if invalid
void *translate(size_t va);

// make it so 'va' is valid, allocating one page for its data
// if it isn't already
void page_allocate(size_t va)
```

translate()

with POBITS=12, LEVELS=1:

ptbr = GetPointerToTable(

VPN	valid?	physical
0	0	—
1	1	0x9999
2	0	—
3	1	0x3333
...	...	...

)

translate(0x0FFF) == (void*) ~0L
translate(0x1000) == (void*) 0x9999000
translate(0x1001) == (void*) 0x9999001
translate(0x2000) == (void*) ~0L
translate(0x2001) == (void*) ~0L
translate(0x3000) == (void*) 0x3333000

translate()

with POBITS=12, LEVELS=1:

ptbr = GetPointerToTable(

VPN	valid?	physical
0	0	—
1	1	0x9999
2	0	—
3	1	0x3333
...	...	...

)

translate(0x0FFF) == (void*) ~0L
translate(0x1000) == (void*) 0x9999000
translate(0x1001) == (void*) 0x9999001
translate(0x2000) == (void*) ~0L
translate(0x2001) == (void*) ~0L
translate(0x3000) == (void*) 0x3333000

page_allocate()

with POBITS=12, LEVELS=1:

ptbr == 0

page_allocate(0x1000) *or* page_allocate(0x1001) *or* ...

page_allocate()

with POBITS=12, LEVELS=1:

ptbr == 0

page_allocate(0x1000) *or* page_allocate(0x1001) *or* ...

ptbr *now* == GetPointerToTable(

VPN	valid?	physical
0	0	—
1	1	(new)
2	0	—
3	1	—
...	...	...

)

allocated with `posix_memalign`

page_allocate()

with POBITS=12, LEVELS=1:

ptbr == 0

page_allocate(0x1000) *or* page_allocate(0x1001) *or* ...

ptbr *now* == GetPointerToTable(

VPN	valid?	physical
0	0	—
1	1	(new)
2	0	—
3	1	—
...	...	...

)

allocated with `posix_memalign`

posix_memalign

```
void *result;
error_code =
    posix_memalign(&result, alignment, size);
```

allocate `size` bytes

choosing address that is multiple of `alignment`

- can make sure allocation starts at beginning of page

`error_code` indicates if out-of-memory, etc.

fills in `result` (passed via pointer)

posix_memalign

```
void *result;
error_code =
    posix_memalign(&result, alignment, size);
```

allocate `size` bytes

choosing address that is multiple of `alignment`

- can make sure allocation starts at beginning of page

`error_code` indicates if out-of-memory, etc.

fills in `result` (passed via pointer)

posix_memalign

```
void *result;
error_code =
    posix_memalign(&result, alignment, size);
```

allocate `size` bytes

choosing address that is multiple of `alignment`

- can make sure allocation starts at beginning of page

`error_code` indicates if out-of-memory, etc.

fills in `result` (passed via pointer)

parts

part 1 (next week): LEVELS=1, POBITS=12 and
- translate() OR
- page_allocate()

part 2: all LEVELS, both functions
- in preparation for code review
- originally scheduled for lab on the 27th
- will move to lab just after reading day
- (might mean I need to cancel lab one week)

part 3: final submission
- Friday after code review
- most of grade based on this
- will test previous parts again

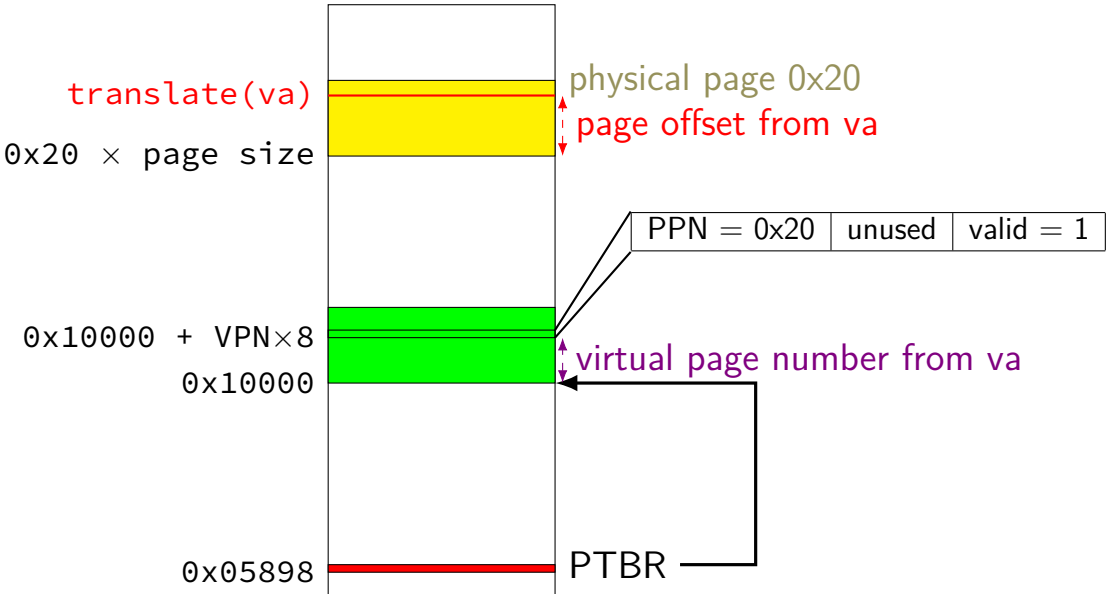
address/page table entry format

(with POBITS=12, LEVELS=1)

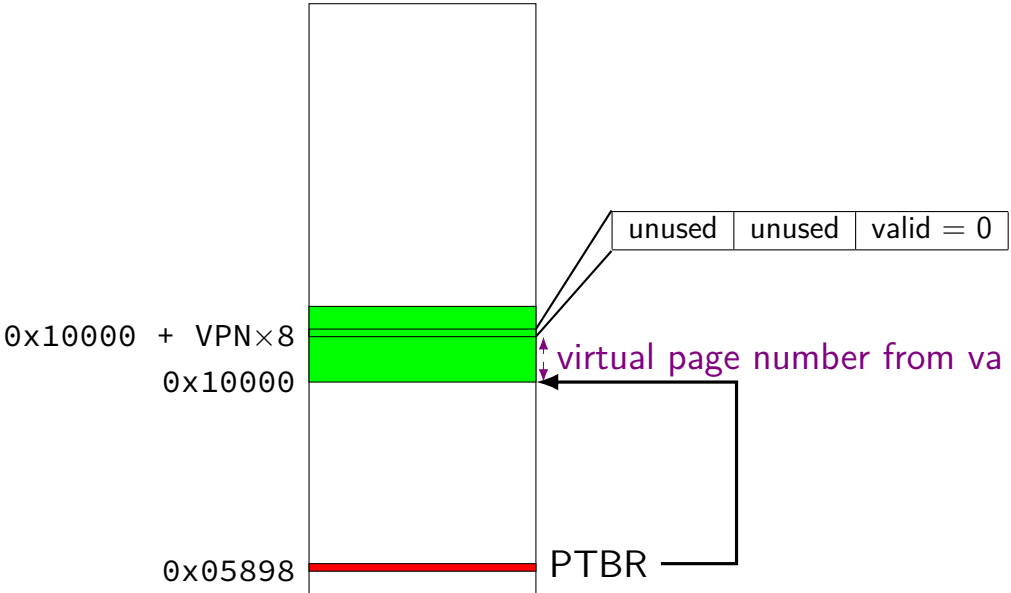
	bits 63–21	bits 20–12	bits 11–1	bit 0
page table entry	physical page number		unused	valid bit
virtual address	unused	virtual page number	page offset	
physical address	physical page number		page offset	

in assignment: value from posix_memalign = physical address

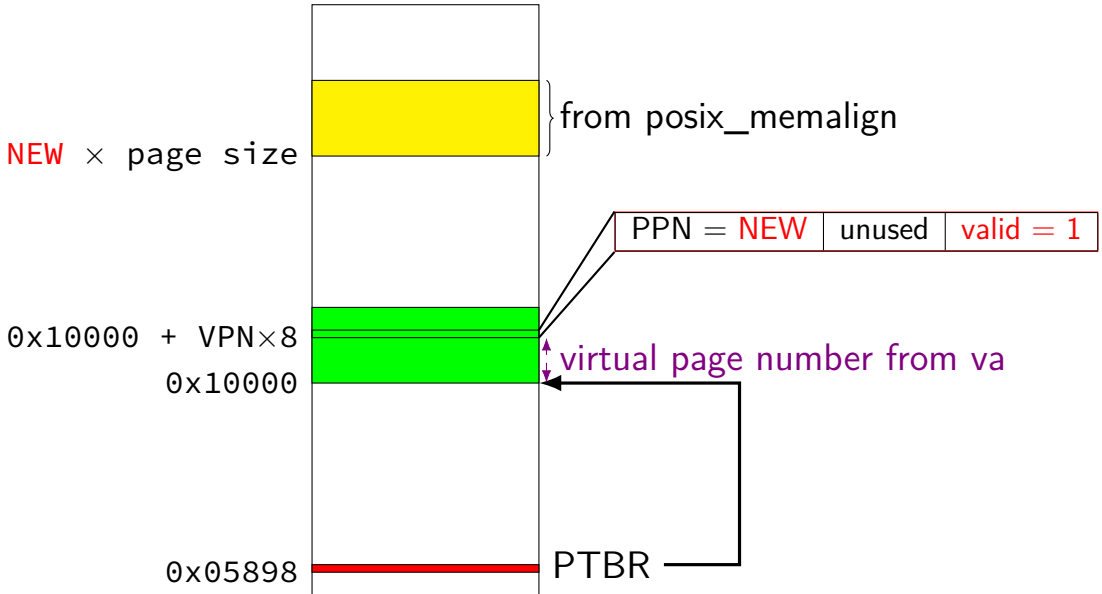
pa = translate(va) [LEVELS=1]



page_allocate(va) [LEVELS=1]



page_allocate(va) [LEVELS=1]



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 64-bit addresses 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? (assuming page table like shown before)

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? (assuming page table like shown before) $2^{48}/2^{12} = 2^{36}$ 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? (assuming page table like shown before) $2^{48}/2^{12} = 2^{36}$ entries

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

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

trick: power of two size makes table lookup faster

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

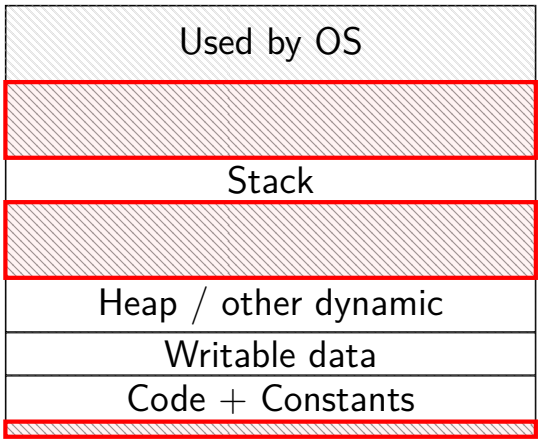
huge page tables

huge virtual address spaces!

impossible to store PTE for every page

how can we save space?

holes



most pages are invalid

saving space

basic idea: don't store (most) invalid page table entries

use a data structure other than a flat array

want a map — lookup key (virtual page number), get value (PTE)

options?

saving space

basic idea: don't store (most) invalid page table entries

use a data structure other than a flat array

- want a map — lookup key (virtual page number), get value (PTE)

options?

hashtable

- actually used by some historical processors
- but never common

saving space

basic idea: don't store (most) invalid page table entries

use a data structure other than a flat array

- want a map — lookup key (virtual page number), get value (PTE)

options?

hashtable

- actually used by some historical processors
- but never common

tree data structure

- but not quite a search tree

search tree tradeoffs

lookup usually implemented in hardware

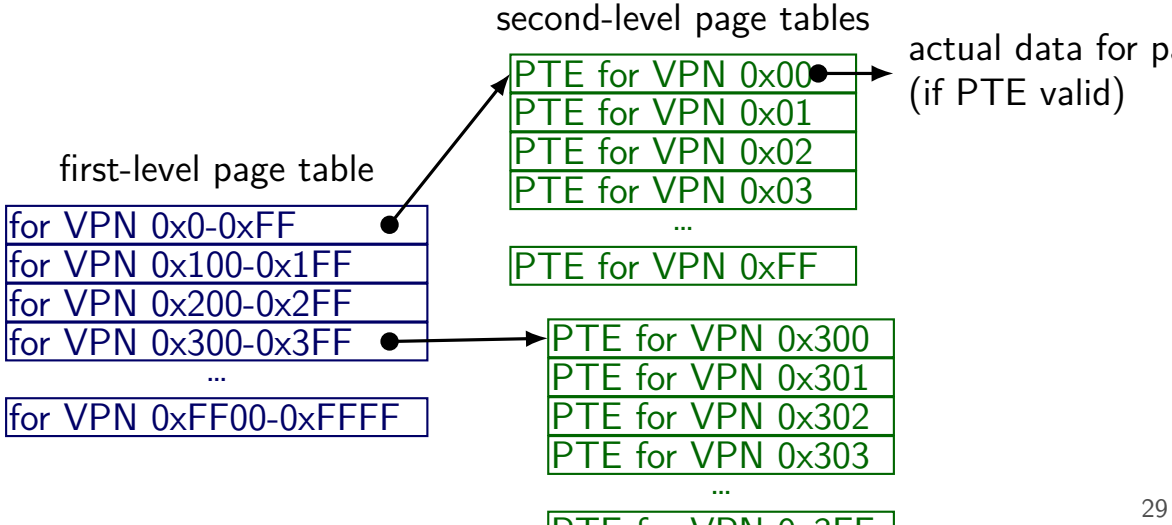
- lookup should be simple
- solution: lookup splits up address bits (no complex calculations)

lookup should not involve many memory accesses

- doing two memory accesses is already very slow
- solution: tree with many children from each node
  - (far from binary tree's left/right child)

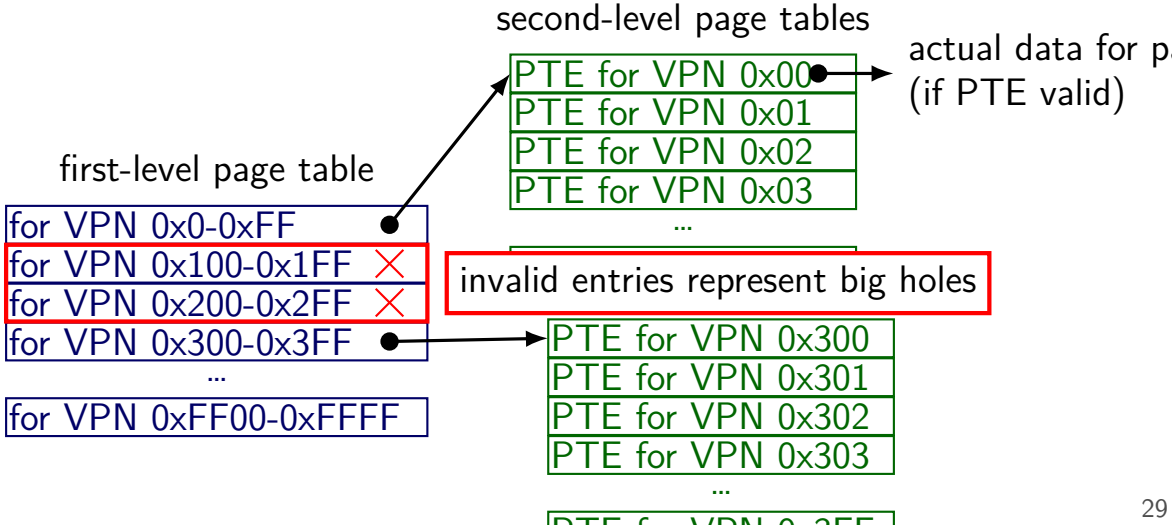
two-level page tables

two-level page tables for 65536 pages (16-bit VPN; 256 entries/table)

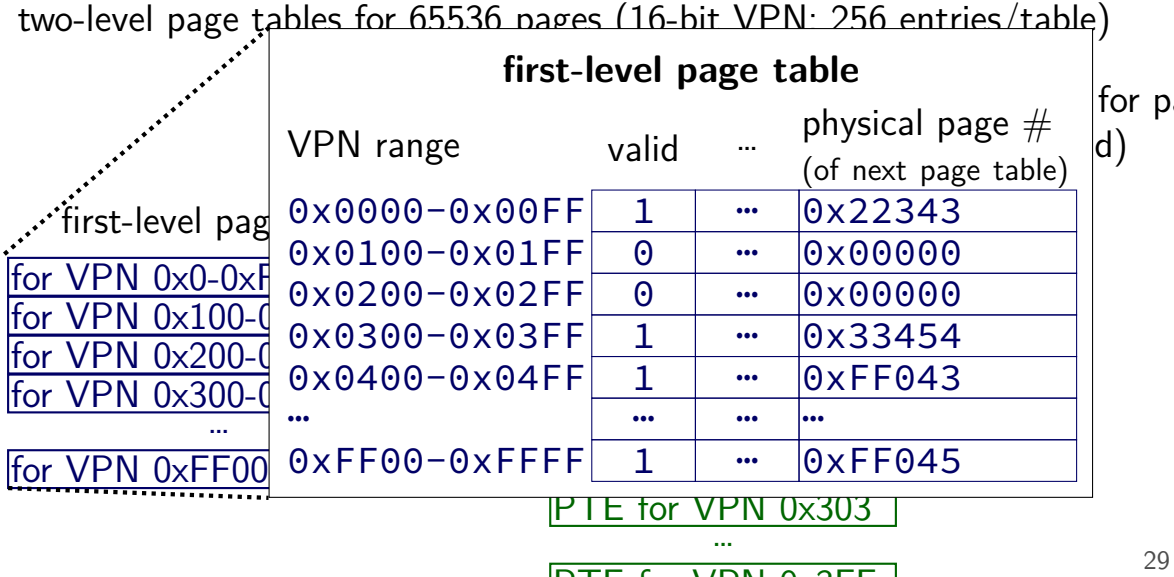


two-level page tables

two-level page tables for 65536 pages (16-bit VPN; 256 entries/table)

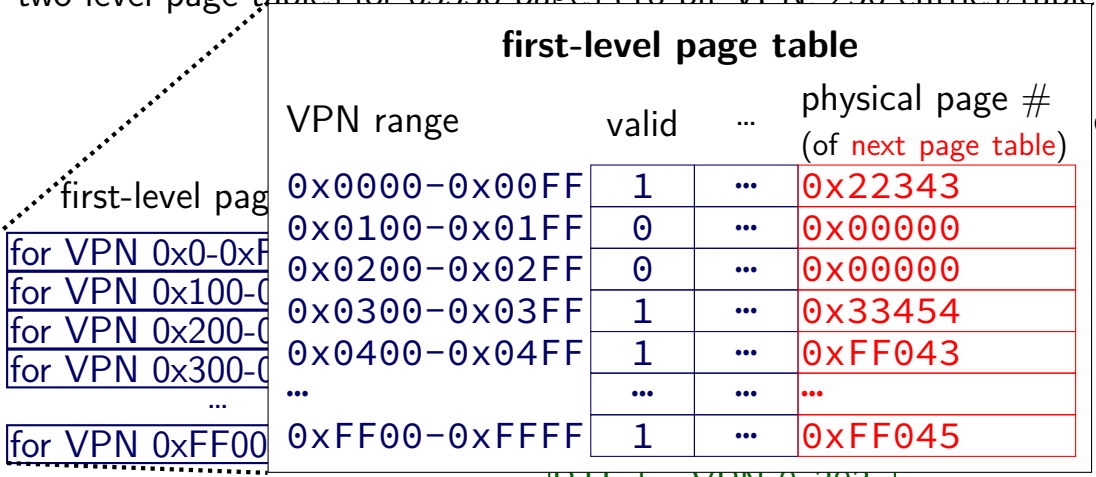


two-level page tables

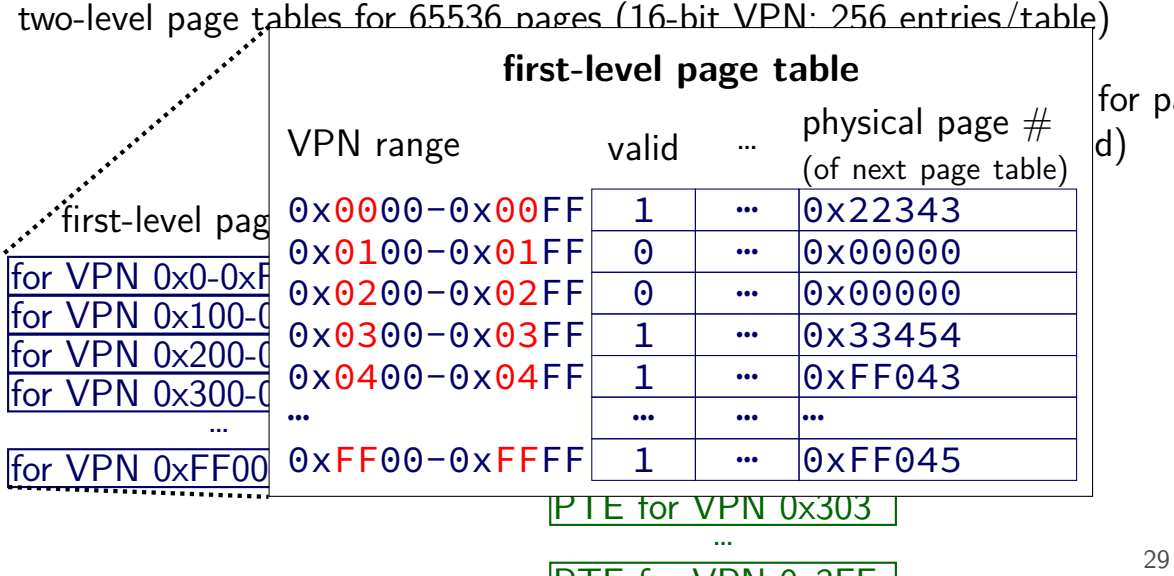


two-level page tables

two-level page tables for 65536 pages (16-bit VPN; 256 entries/table)

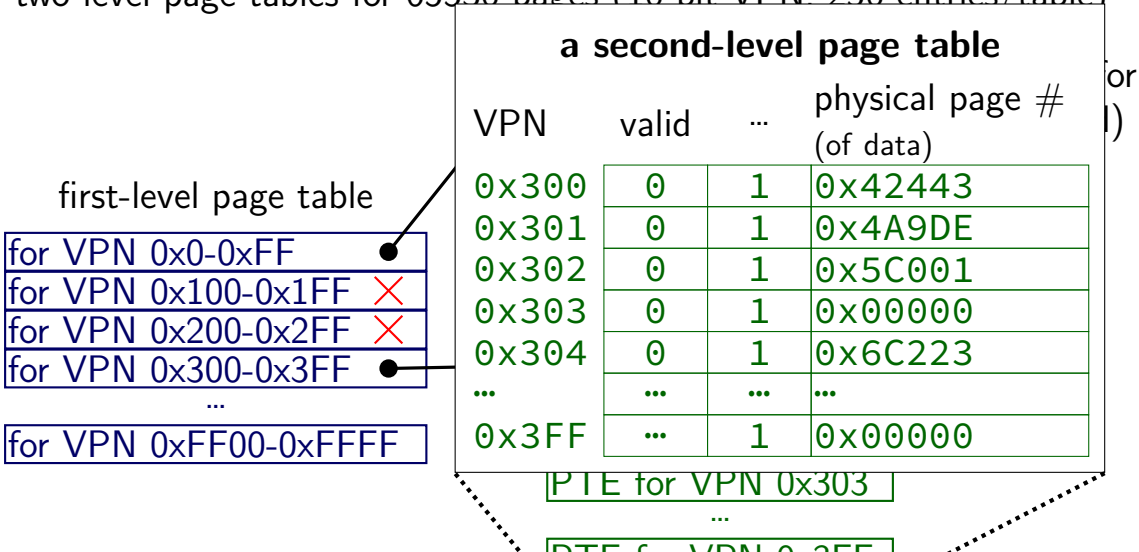


two-level page tables



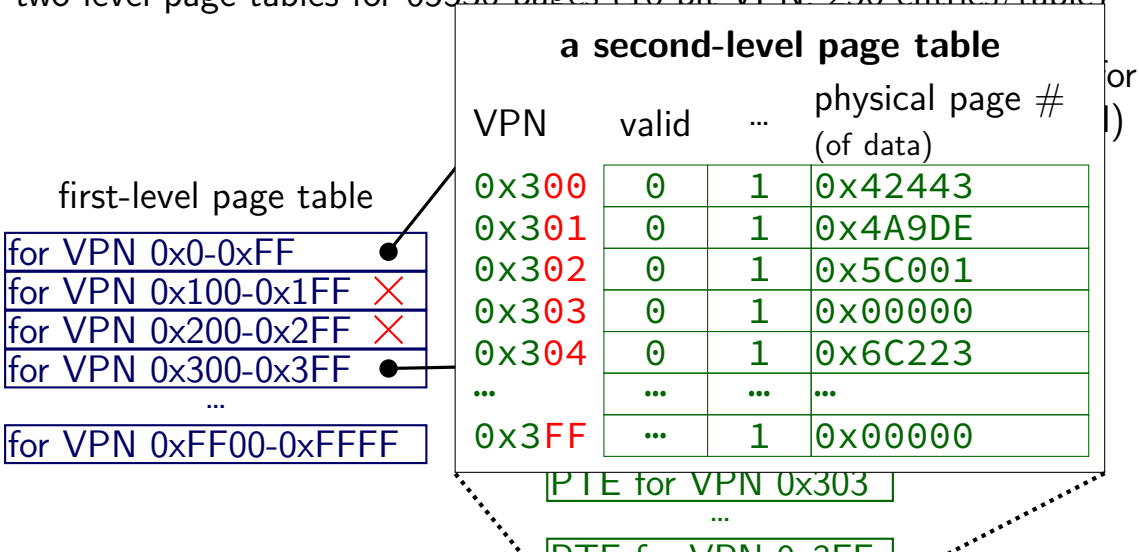
two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table)



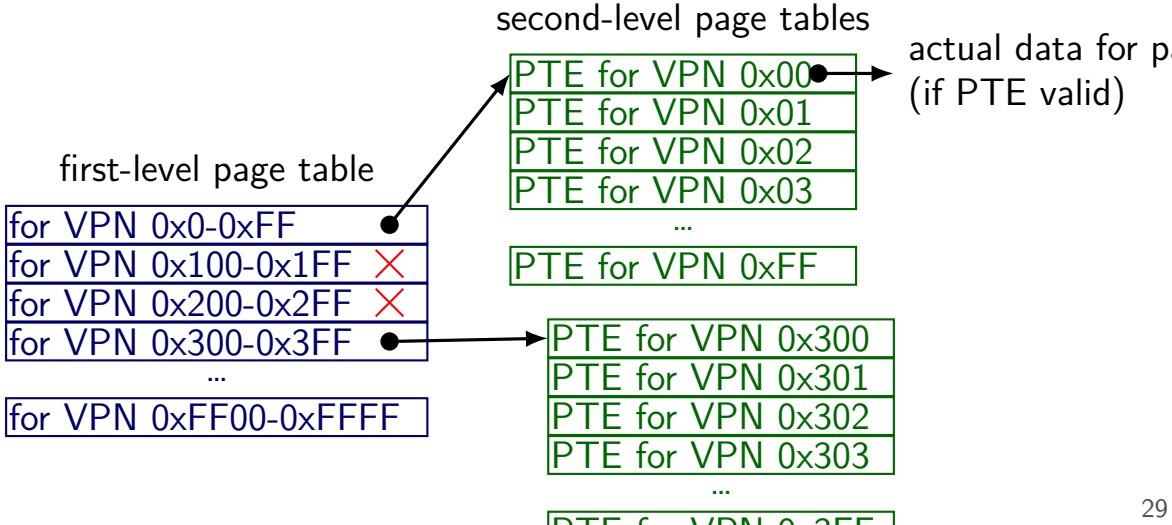
two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table)



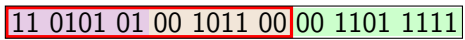
two-level page tables

two-level page tables for 65536 pages (16-bit VPN; 256 entries/table)



two-level page table lookup

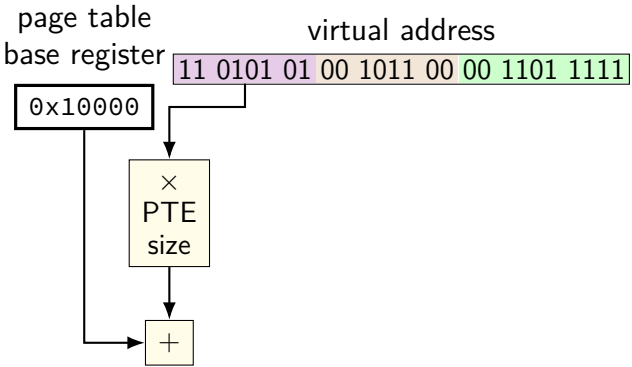
virtual address



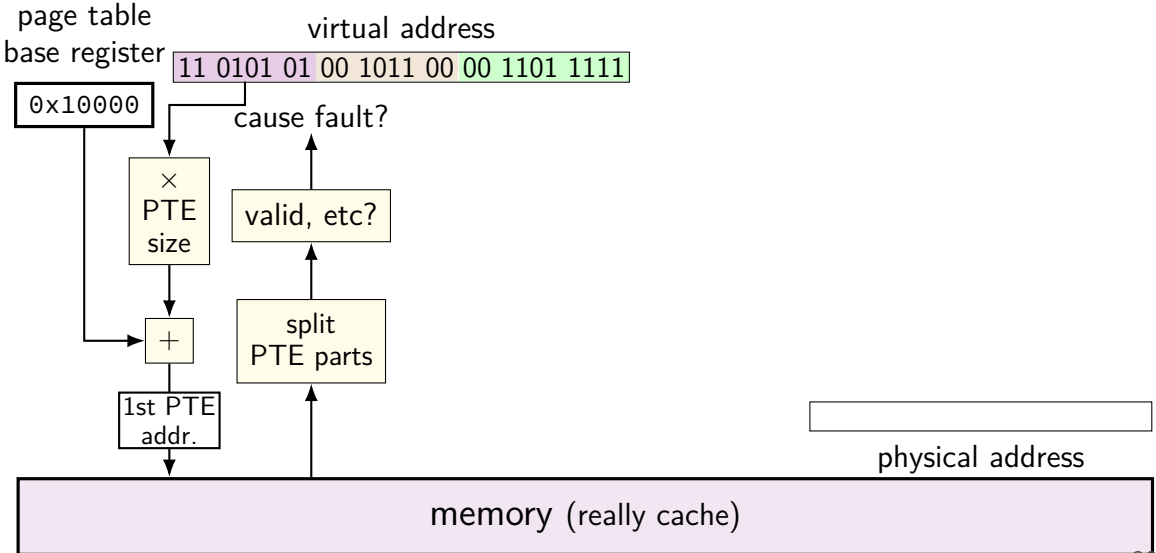
VPN — split into two parts (one per level)

this example: parts equal sized — common, but not required

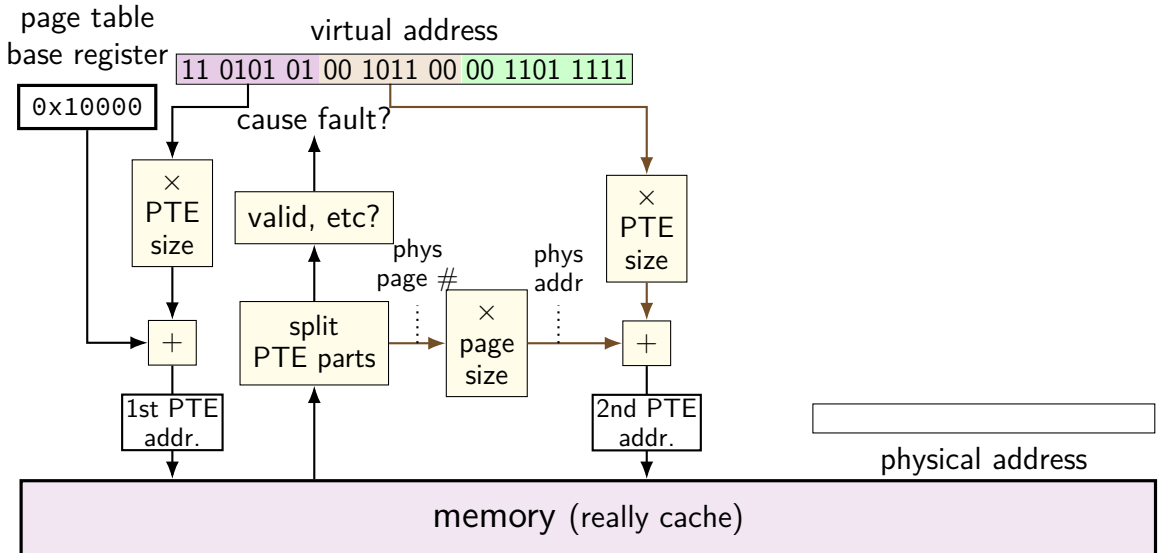
two-level page table lookup



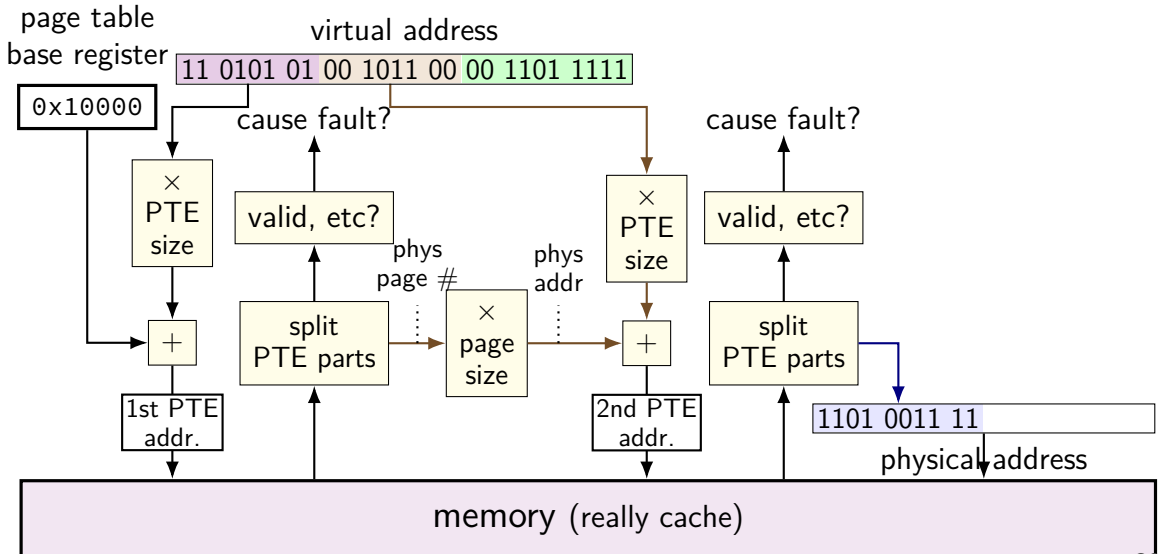
two-level page table lookup



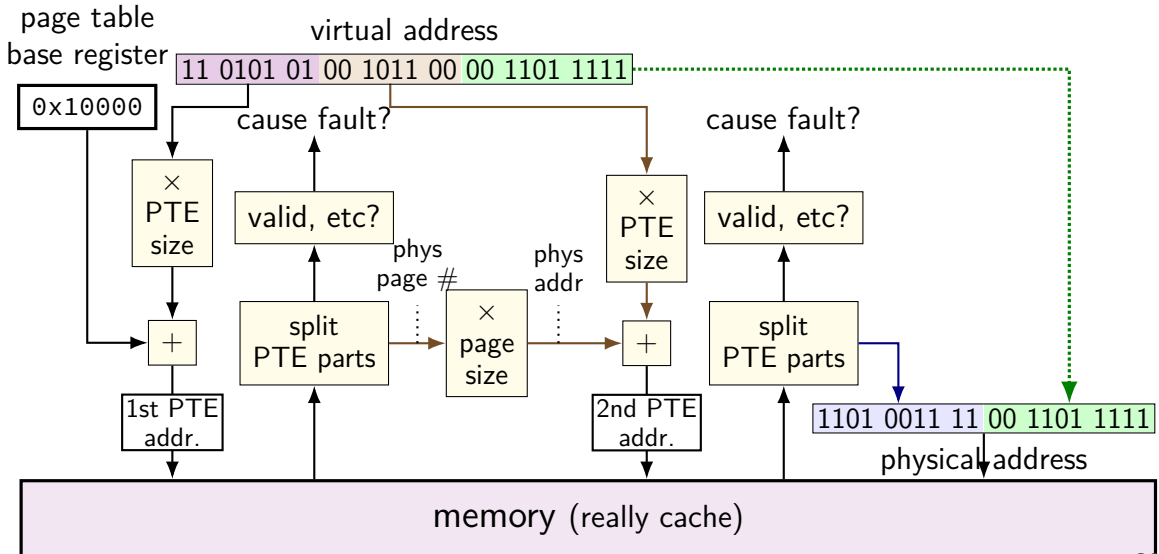
two-level page table lookup



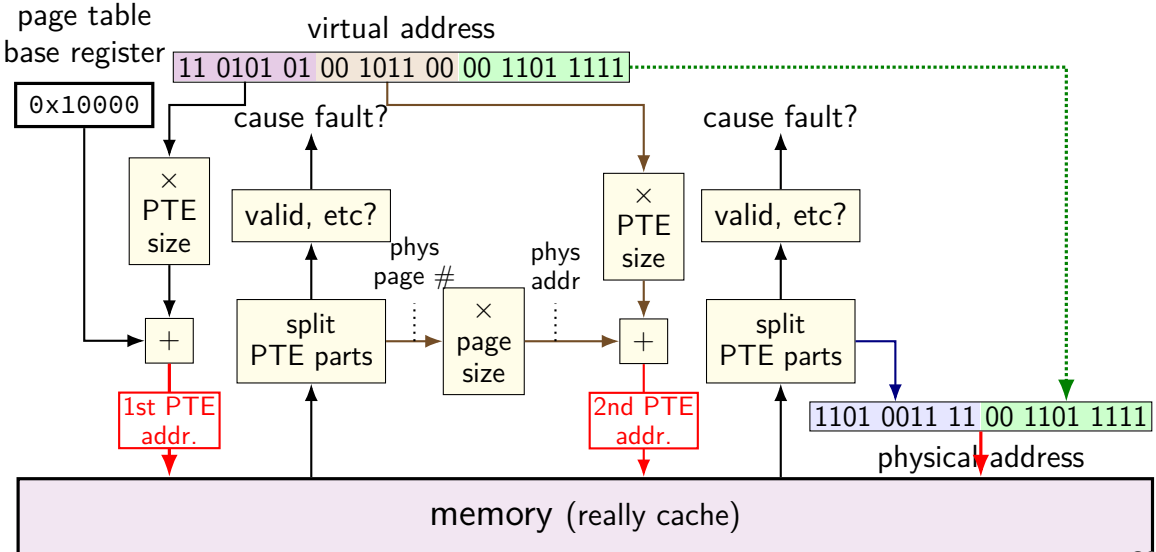
two-level page table lookup



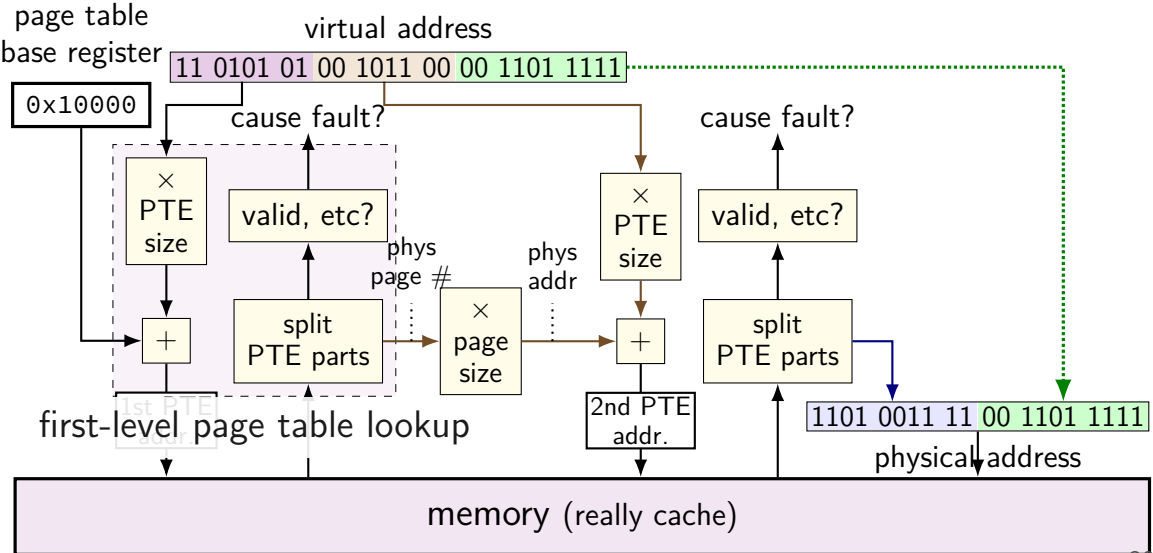
two-level page table lookup



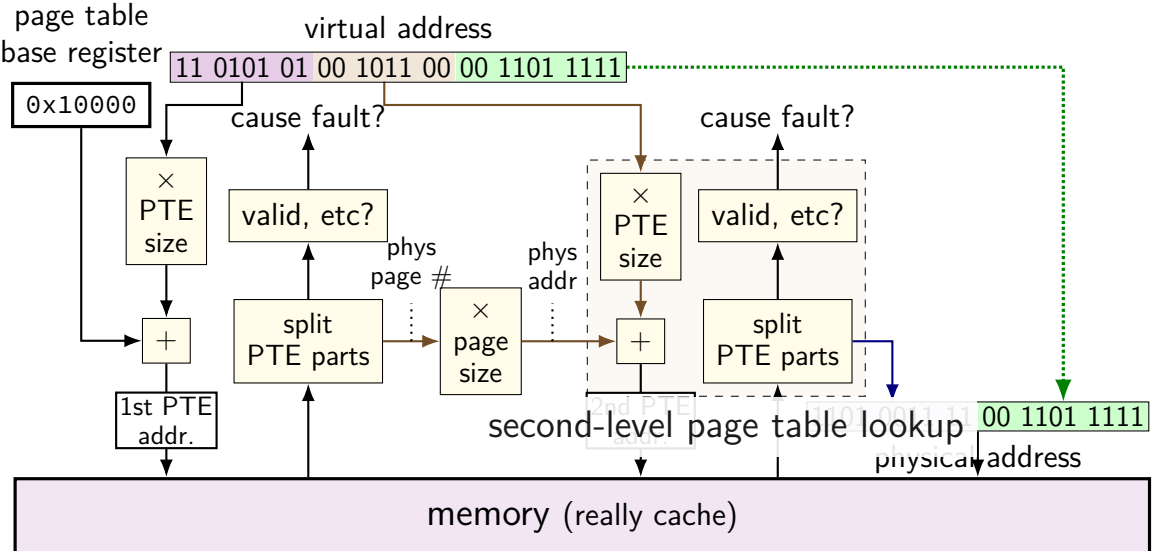
two-level page table lookup



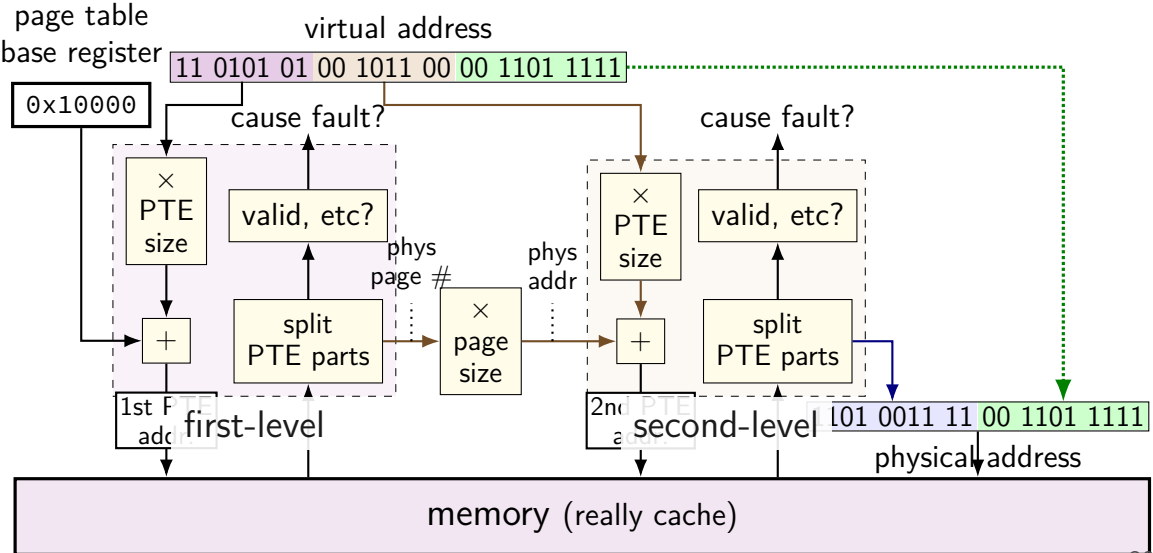
two-level page table lookup



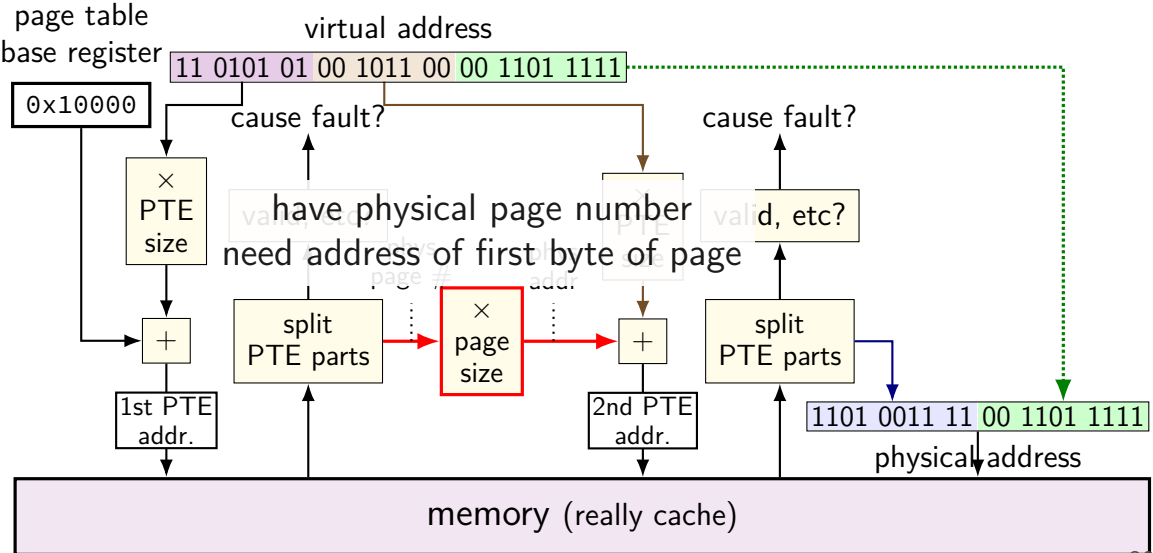
two-level page table lookup



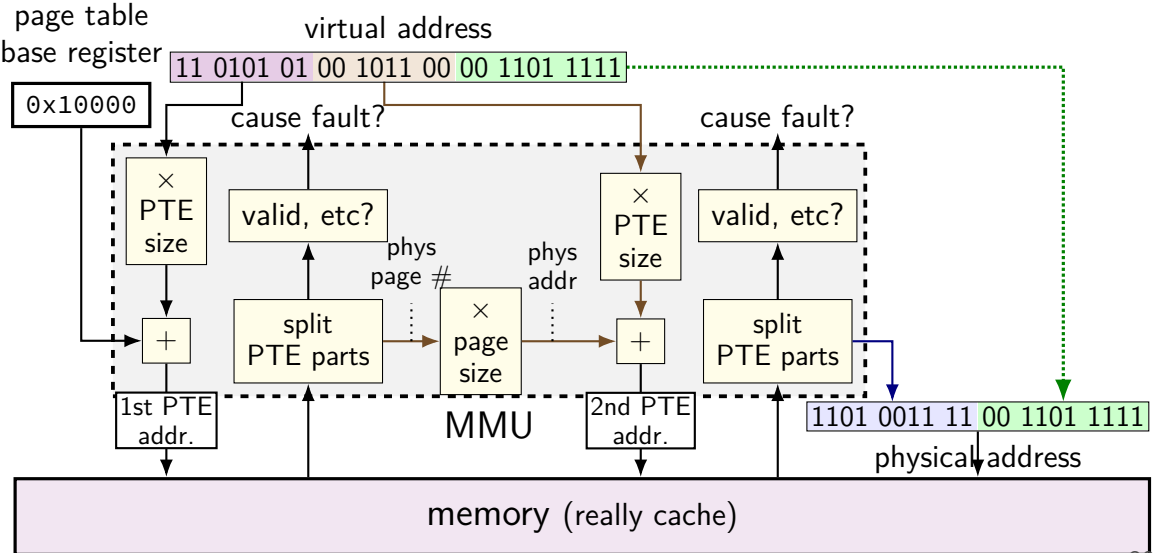
two-level page table lookup



two-level page table lookup



two-level page table lookup



another view