last time

address space counting

- address space size = number of unique addresses
- virtual address space size not always same as pointer size
- physical address space size not same as accessible storage

permission bits in page table entries

- checked like valid bit
- what happens depends on type of access
  - in kernel mode?
  - is the processor trying to write?
  - ...

storing page tables in memory

- array of integers
- choose way of representing page table entry fields in an integer
- special register ("page table base register") with starting location

1-level example

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 bits;

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	F4 F5 F6 F7
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 example

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 bits;

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	F4 F5 F6 F7
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` + 6 × 1 = `0x26`

PTE value:

`0xF6` = `1111 0110`

PPN `111`, valid 1

M[`111 001`] = **M[**`0x39`**]**

→ `0x0C`

1-level example

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 bits;

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	F4 F5 F6 F7
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` + 6 ×1 = `0x26`

PTE value:

`0xF6` = `1111 0110`

PPN `111`, valid 1

M[`111 001`] = **M[**`0x39`]

→ `0x0C`

1-level example

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 bits;

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	F4 F5 F6 F7
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 + 6 ×1 = 0x26

PTE value:

0xF6 = 1111 0110

PPN 111, valid 1

M[111 001] = **M[**0x39]

→ 0x0C

1-level example

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 bits;

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	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
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 example

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 bits;

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	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
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:

0xD2 = 1101 0010

PPN 110, valid 1

M[110 010] = **M[**0x32]

$\rightarrow$ 0xBA

1-level example

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 bits;

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	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
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:

`0xD2` = `1101 0010`

PPN `110`, valid 1

M[`110 010`] = **M[**`0x32`]

→ `0xBA`

1-level example

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 bits;

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	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
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:

`0xD2` = `1101 0010`

PPN `110`, valid 1

M[`110 010`] = **M[**`0x32`]

→ `0xBA`

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??)

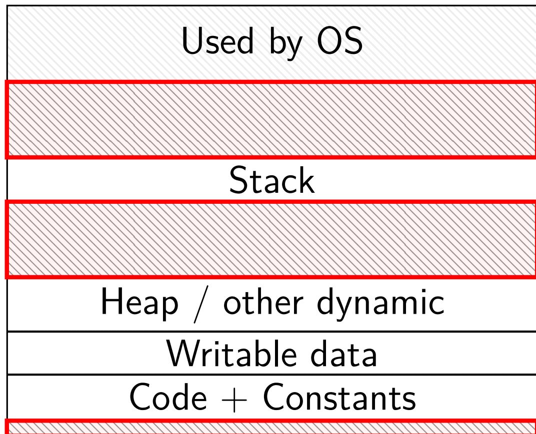
huge page tables

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

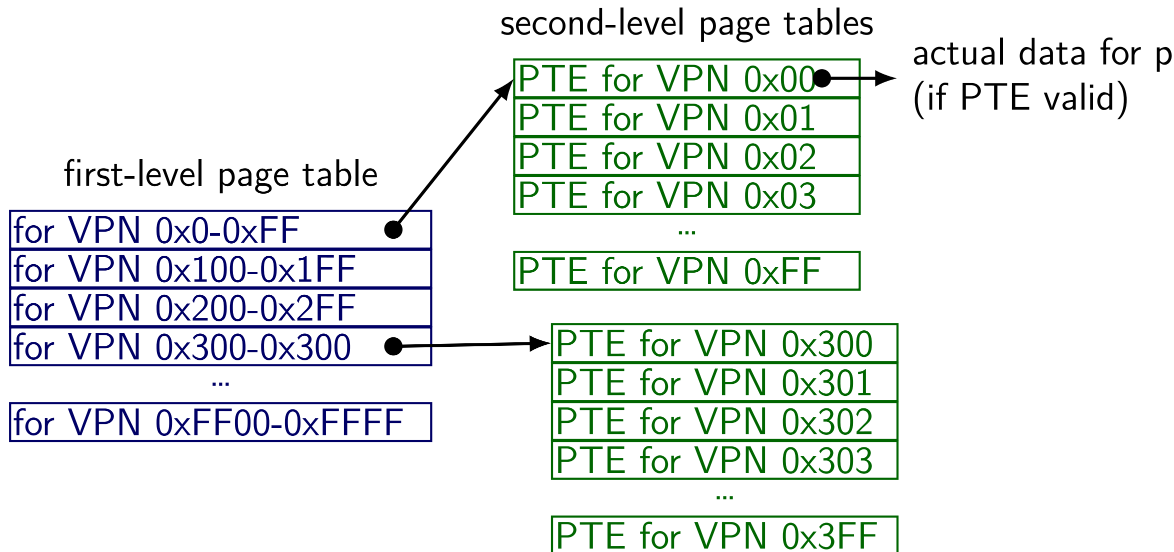
- lookup should be simple
- solution: lookup splits up address bits (no complex calculations)

lookup should not involve many memory accesses

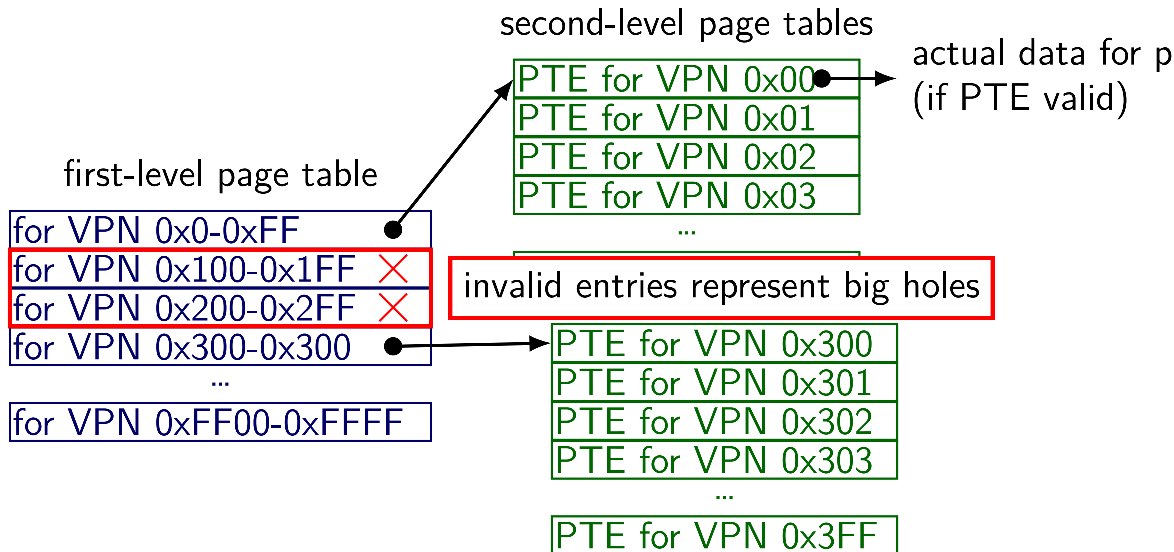
- doing two memory accesses is already very slow
- solution: tree with many children from each node
  - (far from binary tree's left/right child)

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

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

first-level pag

for VPN 0x0-0xF

for VPN 0x100-0

for VPN 0x200-0

for VPN 0x300-0

...

for VPN 0xFF00

first-level page table

VPN range	valid	kernel	write	physical page # (of next page table)
0x0000-0x00FF	1	0	1	0x22343
0x0100-0x01FF	0	0	1	0x00000
0x0200-0x02FF	0	0	0	0x00000
0x0300-0x03FF	1	1	0	0x33454
0x0400-0x04FF	1	1	0	0xFF043
...	...	...	...	...
0xFF00-0xFFFF	1	1	0	0xFF045

PTE for VPN 0x303

...

PTE for VPN 0x3FF

two-level page tables

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

first-level pag

for VPN 0x0-0xF
for VPN 0x100-0
for VPN 0x200-0
for VPN 0x300-0
...
for VPN 0xFF00

first-level page table

VPN range	valid	kernel	write	physical page # (of next page table)
0x0000-0x00FF	1	0	1	0x22343
0x0100-0x01FF	0	0	1	0x00000
0x0200-0x02FF	0	0	0	0x00000
0x0300-0x03FF	1	1	0	0x33454
0x0400-0x04FF	1	1	0	0xFF043
...	...	...	...	...
0xFF00-0xFFFF	1	1	0	0xFF045

PTE for VPN 0x303

...

PTE for VPN 0x3FF

two-level page tables

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

first-level pag

for VPN 0x0-0xF

for VPN 0x100-0

for VPN 0x200-0

for VPN 0x300-0

...

for VPN 0xFF00

first-level page table

VPN range	valid	kernel	write	physical page # (of next page table)
0x0000-0x00FF	1	0	1	0x22343
0x0100-0x01FF	0	0	1	0x00000
0x0200-0x02FF	0	0	0	0x00000
0x0300-0x03FF	1	1	0	0x33454
0x0400-0x04FF	1	1	0	0xFF043
...	...	...	...	...
0xFF00-0xFFFF	1	1	0	0xFF045

PTE for VPN 0x303

...

PTE for VPN 0x3FF

two-level page tables

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

first-level page table

first-level page table
for VPN 0x0-0xFF
for VPN 0x100-0x1FF
for VPN 0x200-0x2FF
for VPN 0x300-0x300
...
for VPN 0xFF00-0xFFFF

a second-level page table

VPN	valid	kernel	write	physical page # (of data)
0x300	1	1	0	0x42443
0x301	1	1	0	0x4A9DE
0x302	1	1	0	0x5C001
0x303	0	0	0	0x00000
0x304	1	1	0	0x6C223
...	...	...	...	...
0x3FF	0	0	0	0x00000

PTE for VPN 0x303

...

PTE for VPN 0x3FF

two-level page tables

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

first-level page table

- for VPN 0x0-0xFF ●
- for VPN 0x100-0x1FF ✕
- for VPN 0x200-0x2FF ✕
- for VPN 0x300-0x300 ●
- ...
- for VPN 0xFF00-0xFFFF

a second-level page table

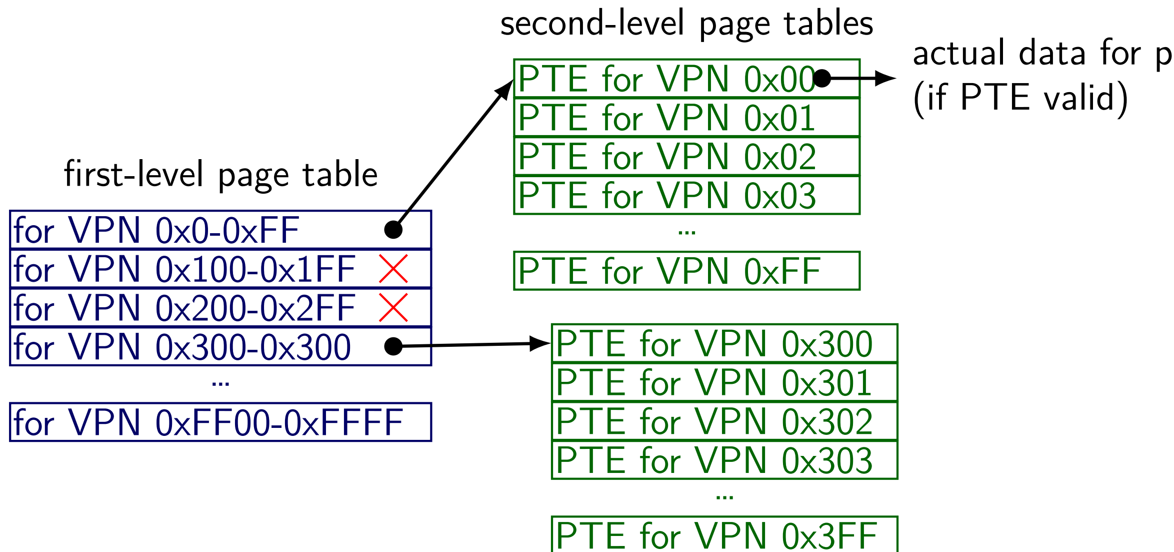
VPN	valid	kernel	write	physical page # (of data)
0x300	1	1	0	0x42443
0x301	1	1	0	0x4A9DE
0x302	1	1	0	0x5C001
0x303	0	0	0	0x00000
0x304	1	1	0	0x6C223
...	...	...	...	...
0x3FF	0	0	0	0x00000

PTE for VPN 0x303

...

PTE for VPN 0x3FF

two-level page tables



two-level page table lookup

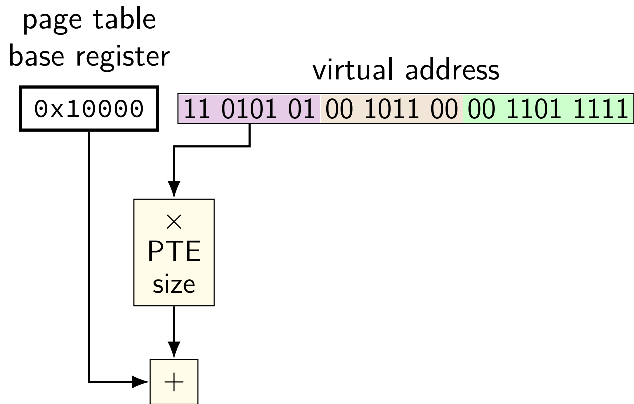
virtual address

11 0101 01 00 1011 00 00 1101 1111

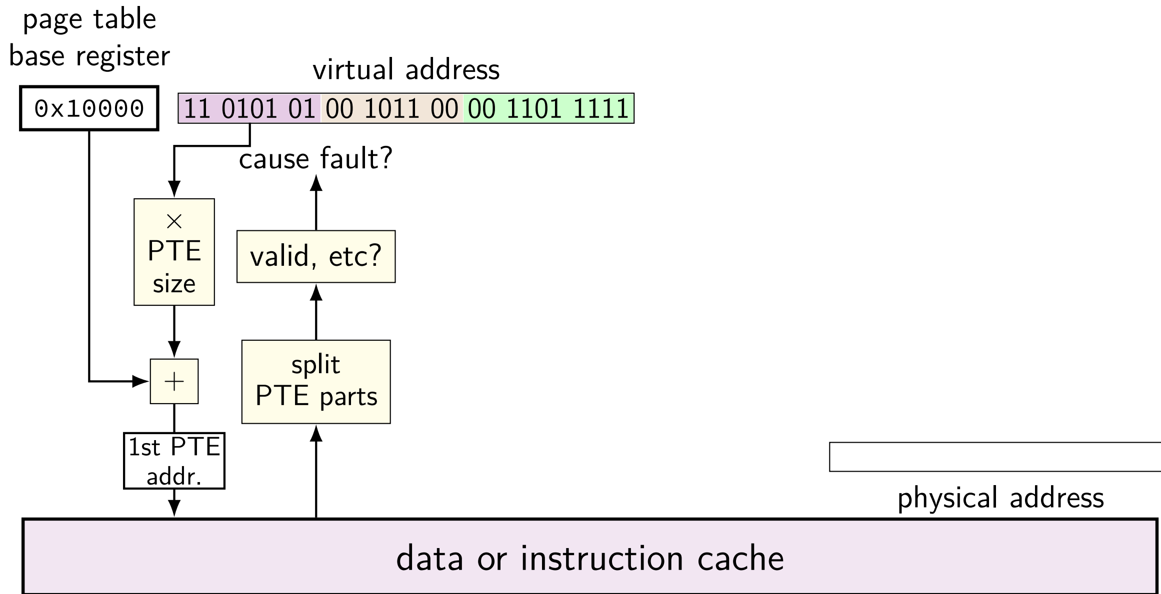
VPN — split into two parts (one per level)

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

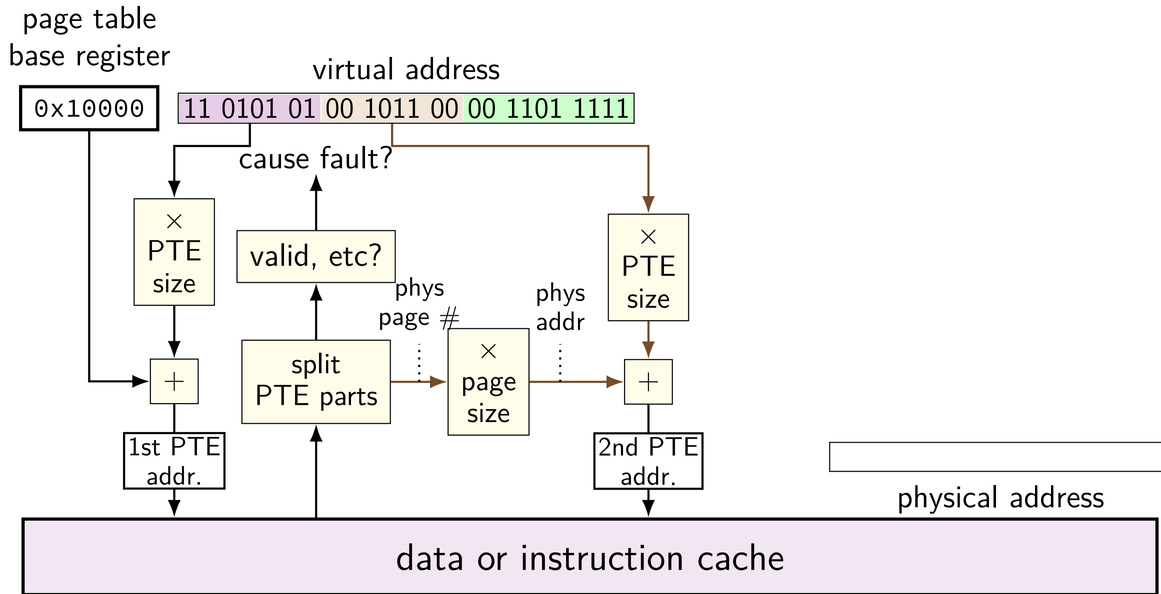
two-level page table lookup



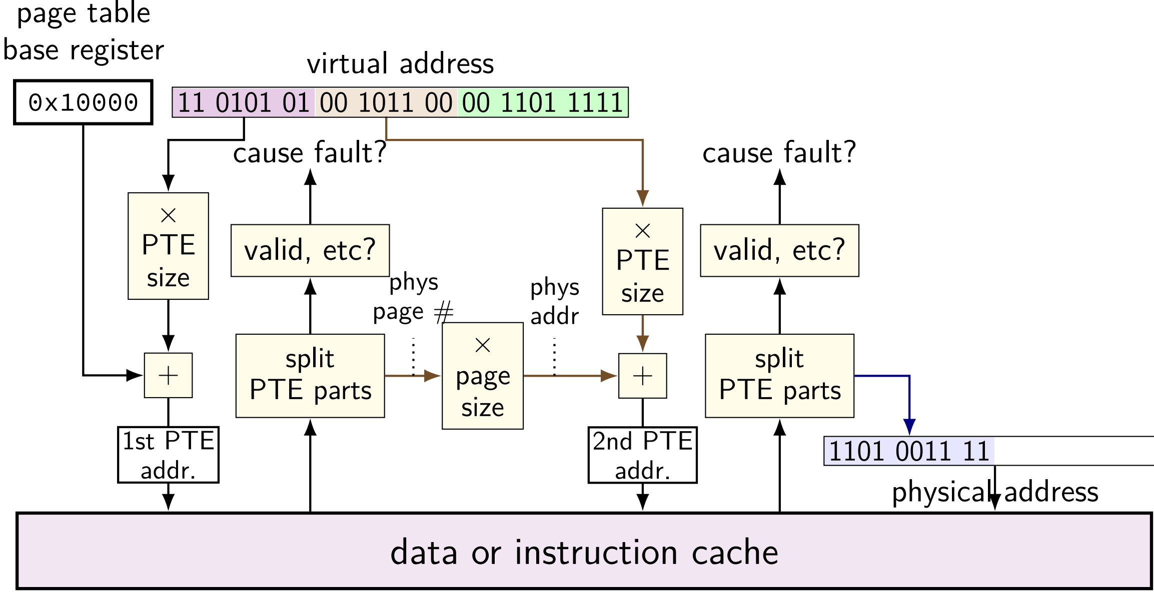
two-level page table lookup



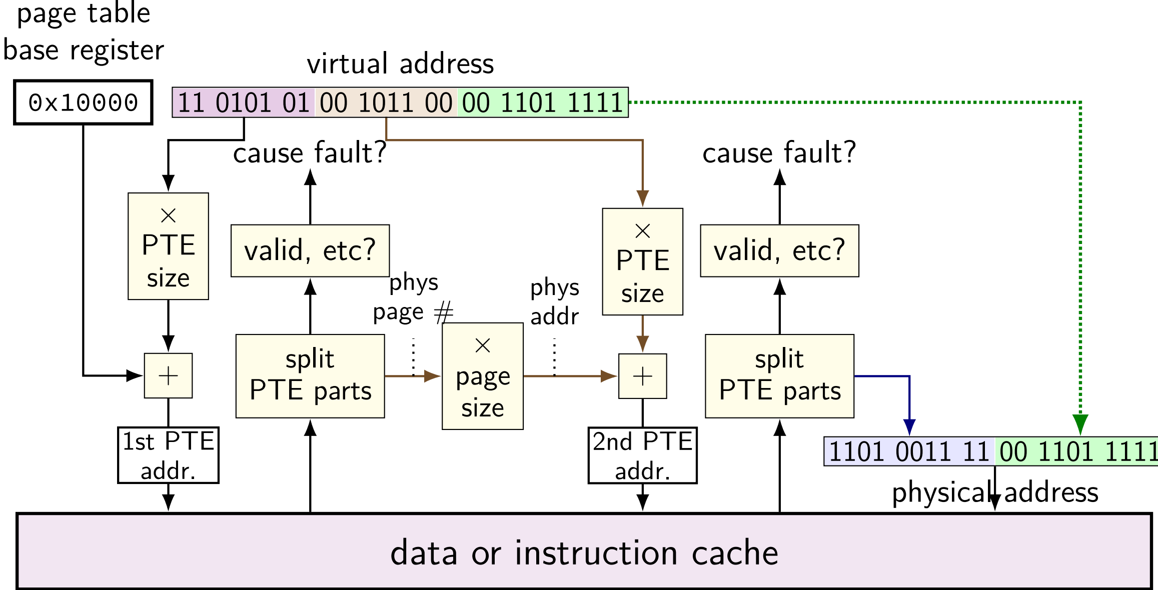
two-level page table lookup



two-level page table lookup

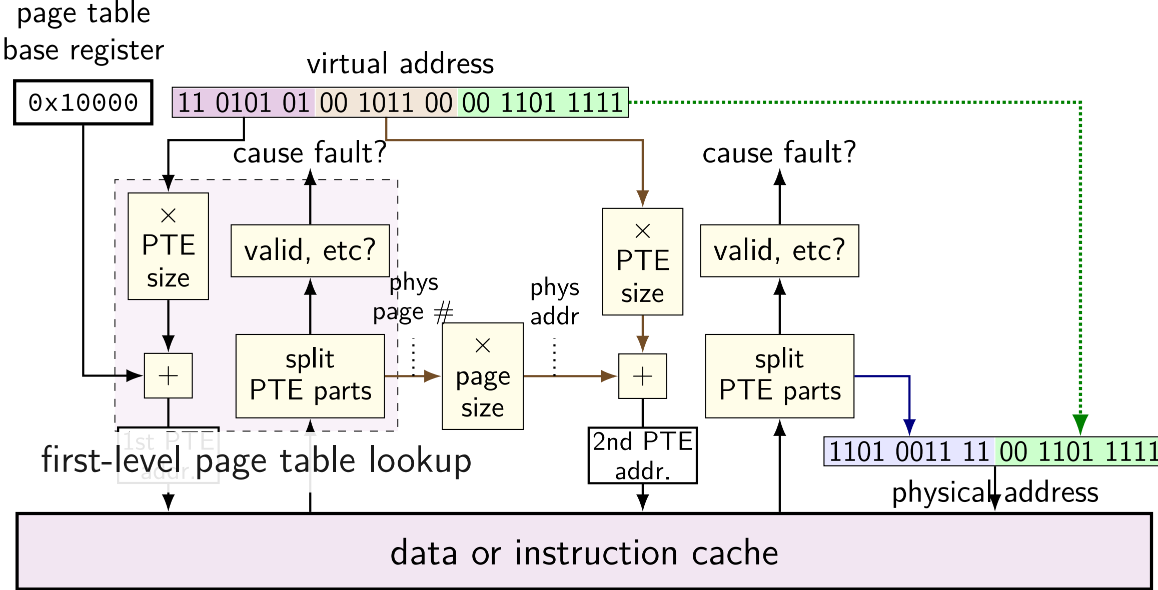


two-level page table lookup

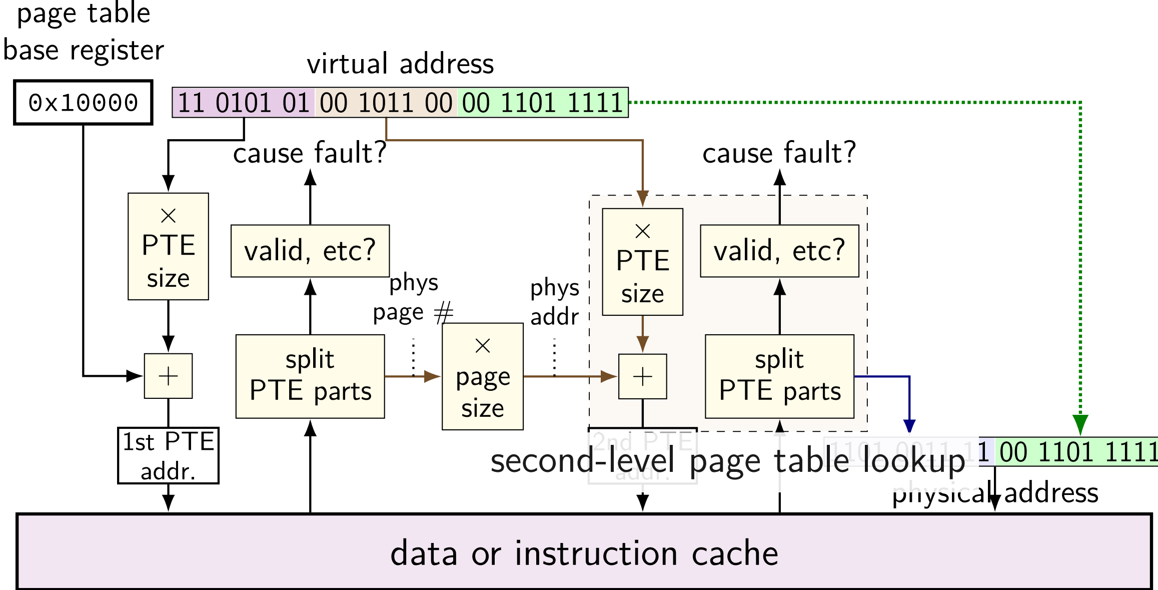


two-level page table lookup

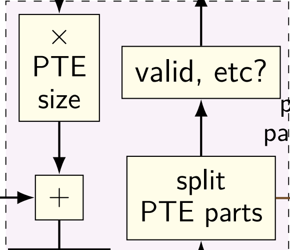
two-level page table lookup

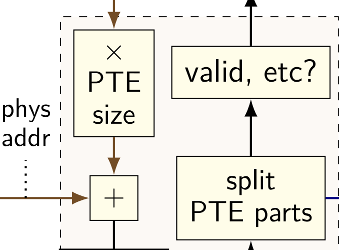


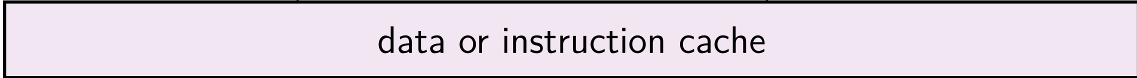
two-level page table lookup



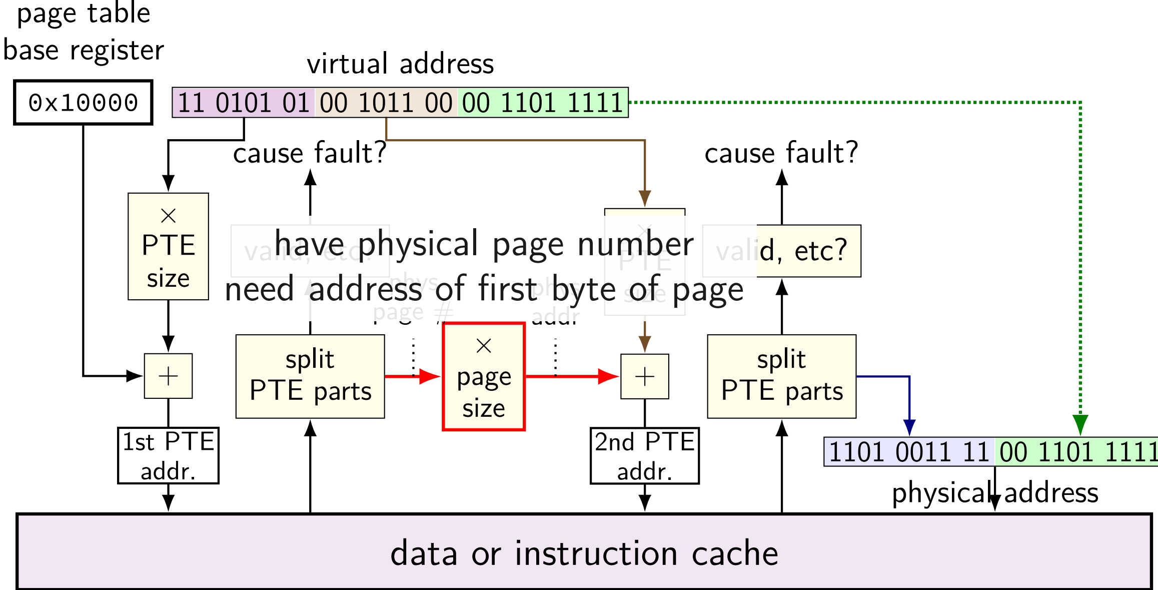
two-level page table lookup



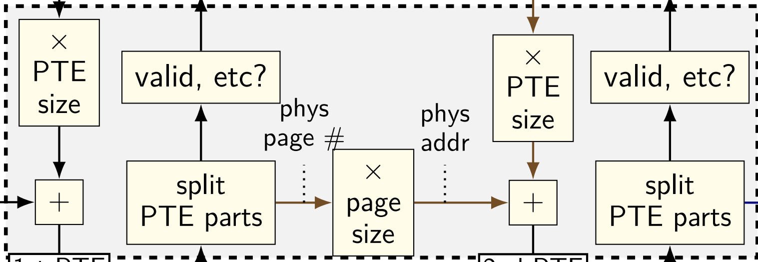




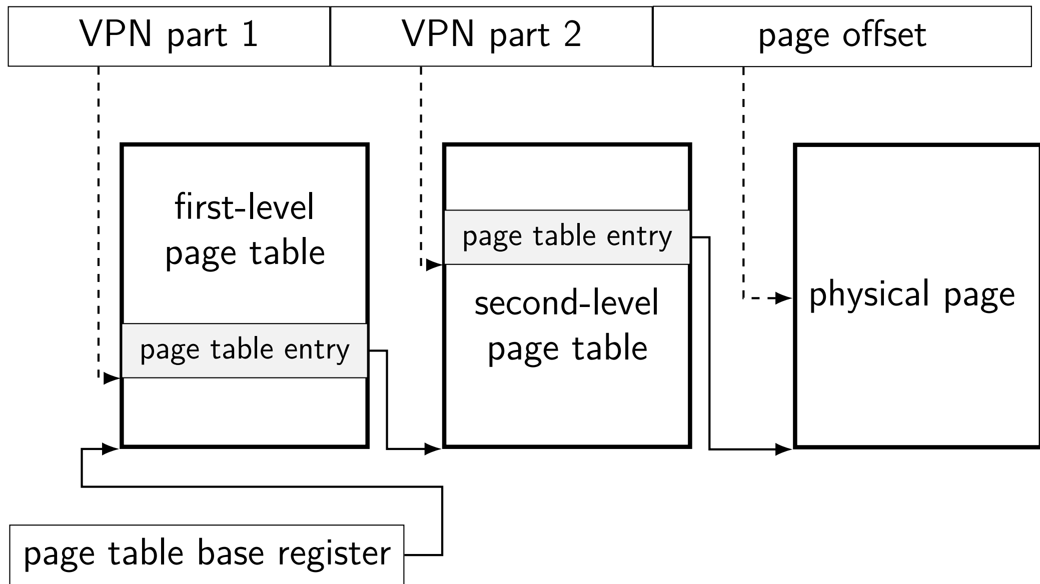
two-level page table lookup



two-level page table lookup



another view



multi-level page tables

VPN split into pieces for each level of page table

top levels: page table entries point to next page table

usually using physical page number of next page table

bottom level: page table entry points to destination page

validity and permission checks at each level

x86-64 page table splitting

48-bit virtual address

12-bit page offset (4KB pages)

36-bit virtual page number, split into four 9-bit parts

page tables at each level: 2^9 entries, 8 bytes/entry

- deliberate choice: each page table is one page

note on VPN splitting

textbook labels it ‘VPN 1’ and ‘VPN 2’ and so on

these are parts of the virtual page number

(there are not multiple VPNs)

2-level example

9-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 unused

page table base register `0x20`; translate virtual address `0x131`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

2-level example

9-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 unused

page table base register `0x20`; translate virtual address `0x131`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x131` $=$ `1 0011 0001`

`0x20` $+ 4 \times 1 =$ `0x24`

PTE 1 value:

`0xD4` $=$ `1101 0100`

PPN `110`, valid 1

2-level example

9-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 unused

page table base register `0x20`; translate virtual address `0x131`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x131` = `1 0011 0001`

`0x20` + 4 × 1 = `0x24`

PTE 1 value:

`0xD4` = `1101 0100`

PPN `110`, valid `1`

PTE 2 addr:

PPN from 1st × page size +
VPN pt 2 × PTE size

PTE 2 value: `0xDB`

2-level example

9-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 unused

page table base register `0x20`; translate virtual address `0x131`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x131` = `1 0011 0001`

`0x20` + 4 × 1 = `0x24`

PTE 1 value:

`0xD4` = `1101 0100`

PPN `110`, valid `1`

PTE 2 addr:

PPN from 1st × page size +
VPN pt 2 × PTE size

PTE 2 value: `0xDB`

PPN `110`; valid `1`

M[`110 001` (`0x31`)] = `0x0A`

2-level example

9-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 unused

page table base register `0x20`; translate virtual address `0x131`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x131` = `1 0011 0001`

`0x20` + 4 × 1 = `0x24`

PTE 1 value:

`0xD4` = `1101 0100`

PPN `110`, valid `1`

PTE 2 addr:

PPN from 1st × page size +
VPN pt 2 × PTE siz

PTE 2 value: `0xDB`

PPN `110`; valid `1`

M[`110 001` (`0x31`)] = `0x0A`

2-level example

9-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 unused

page table base register `0x20`; translate virtual address `0x131`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x131` = `1 0011 0001`

`0x20` + `4` ×1 = `0x24`

PTE 1 value:

`0xD4` = `1101 0100`

PPN `110`, valid 1

PTE 2 addr:

PPN from 1st × page size +
VPN pt 2 × PTE size

PTE 2 value: `0xDB`

PPN `110`; valid 1

M[`110 001` (`0x31`)] = `0x0A`

2-level splitting

9-bit virtual address

6-bit physical address

8-byte pages $\rightarrow$ 3-bit page offset (bottom bits)

9-bit VA: 6 bit VPN + 3 bit PO

6-bit PA: 3 bit PPN + 3 bit PO

8 entry page tables $\rightarrow$ 3-bit VPN parts

9-bit VA: 3 bit VPN part 1; 3 bit VPN part 2

2-level exercise (1)

9-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 unused;

page table base register `0x08`; translate virtual address `0x0FB`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

2-level exercise (1)

9-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 unused;

page table base register `0x08`; translate virtual address `0x0FB`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x0F3` = `011 111 011`
(PTE 1 addr: `0x08` + PTE size times 011 (3))
PTE 1: `0xBB` at `0x0B`
PTE 1: PPN `101` (5) valid `1`
PTE 2: `0xF0` at `0x2F`
PTE 2: PPN `111` (7) valid `1`
`111 011` = `0x3B` → `0x0C`

2-level exercise (1)

9-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 unused;

page table base register `0x08`; translate virtual address `0x0FB`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x0F3` = `011 111 011`

(PTE 1 addr: `0x08` + PTE size times 011 (3))

PTE 1: `0xBB` at `0x0B`

PTE 1: PPN `101` (5) valid 1

PTE 2: `0xF0` at `0x2F`

PTE 2: PPN `111` (7) valid 1

`111 011` = `0x3B` → `0x0C`

2-level exercise (1)

9-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 unused;

page table base register `0x08`; translate virtual address `0x0FB`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x0F3` = `011 111 011`
(PTE 1 addr: `0x08` + PTE size times 011 (3))
PTE 1: `0xBB` at `0x0B`
PTE 1: PPN `101` (5) valid 1
PTE 2: `0xF0` at `0x2F`
PTE 2: PPN `111` (7) valid 1
`111 011` = `0x3B` → `0x0C`

2-level exercise (1)

9-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 unused;

page table base register `0x08`; translate virtual address `0x0FB`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x0F3` = `011 111 011`

(PTE 1 addr: `0x08` + PTE size times 011 (3))

PTE 1: `0xBB` at `0x0B`

PTE 1: PPN `101` (5) valid 1

PTE 2: `0xF0` at `0x2F`

PTE 2: PPN `111` (7) valid 1

`111 011` = `0x3B` → `0x0C`

2-level exercise (2)

9-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 unused;

page table base register `0x10`; translate virtual address `0x109`

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 5A 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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

2-level exercise (2)

9-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 unused;

page table base register `0x10`; translate virtual address `0x109`

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 5A 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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x109` = `100 011 001`

(PTE 1 at:

0x10 + PTE size times 4 (100))

PTE 1: `0x1B` at `0x14`

PTE 1: PPN `000` (0) valid 1

(second table at:

0 (000) times page size = 0x00)

PTE 2: `0x33` at `0x03`

PTE 2: PPN `001` (1) valid 1

`001 001` = `0x09` → `0x99`

2-level exercise (2)

9-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 unused;

page table base register `0x10`; translate virtual address `0x109`

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 5A 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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x109` = `100 011 001`
(PTE 1 at:
0x10 + PTE size times 4 (100))
PTE 1: `0x1B` at `0x14`
PTE 1: PPN `000` (0) valid `1`
(second table at:
0 (000) times page size = 0x00)
PTE 2: `0x33` at `0x03`
PTE 2: PPN `001` (1) valid `1`
`001 001` = `0x09` → `0x99`

2-level exercise (2)

9-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 unused;

page table base register `0x10`; translate virtual address `0x109`

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 5A 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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x109` = `100 011 001`

(PTE 1 at:

0x10 + PTE size times 4 (100))

PTE 1: `0x1B` at `0x14`

PTE 1: PPN `000` (0) valid 1

(second table at:

0 (000) times page size = 0x00)

PTE 2: `0x33` at `0x03`

PTE 2: PPN `001` (1) valid 1

`001 001` = `0x09` → `0x99`

2-level exercise (2)

9-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 unused;

page table base register `0x10`; translate virtual address `0x109`

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 5A 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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x109` = `100 011 001`
(PTE 1 at:
0x10 + PTE size times 4 (100))
PTE 1: `0x1B` at `0x14`
PTE 1: PPN `000` (0) valid 1
(second table at:
0 (000) times page size = 0x00)
PTE 2: `0x33` at `0x03`
PTE 2: PPN `001` (1) valid 1
`001 001` = `0x09` → `0x99`

2-level exercise (3)

9-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 unused

page table base register `0x08`; translate virtual address `0x00B`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

2-level exercise (3)

9-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 unused

page table base register `0x08`; translate virtual address `0x00B`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x0F3` = `000 001 011`

PTE 1: `0x88` at `0x08`

PTE 1: PPN `100` (5) valid `0`

page fault!

2-level exercise (3)

9-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 unused

page table base register `0x08`; translate virtual address `0x00B`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x0F3` = `000 001 011`

PTE 1: `0x88` at `0x08`

PTE 1: PPN `100` (5) valid `0`

page fault!

2-level exercise (4)

9-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 unused

page table base register `0x08`; translate virtual address `0x1CB`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

2-level exercise (4)

9-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 unused

page table base register `0x08`; translate virtual address `0x1CB`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x1CB` = `111 001 011`

PTE 1: `0xFF` at `0x0F`

PTE 1: PPN `111` (7) valid `1`

PTE 2: `0x0C` at `0x39`

PTE 2: PPN `000` (0) valid `0`

page fault!

2-level exercise (4)

9-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 unused

page table base register `0x08`; translate virtual address `0x1CB`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x1CB` = `111 001 011`

PTE 1: `0xFF` at `0x0F`

PTE 1: PPN `111` (7) valid `1`

PTE 2: `0x0C` at `0x39`

PTE 2: PPN `000` (0) valid `0`

page fault!

2-level exercise (4)

9-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 unused

page table base register `0x08`; translate virtual address `0x1CB`

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	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x1CB` = `111 001 011`

PTE 1: `0xFF` at `0x0F`

PTE 1: PPN `111` (7) valid `1`

PTE 2: `0x0C` at `0x39`

PTE 2: PPN `000` (0) valid `0`

page fault!

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE: 2 bit PPN (MSB of first byte), 1 valid bit, rest unused

page table base register `0x10`; translate virtual address `0x376`

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	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE: 2 bit PPN (MSB of first byte), 1 valid bit, rest unused

page table base register `0x10`; translate virtual address `0x376`

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	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x376` $=$ `110 111 0110`

PTE 1: $0x10 + 6 \times 2 =$ `0x1C`: `AC BC`

PTE 1: PPN `10` valid `1`

PTE 2: $0x20 + 7 \times 2 =$ `0x2E`: `EF F0`

PTE 2: PPN `11` valid `1`

`11 0110` $=$ `0x36` $\rightarrow$ DB

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE: 2 bit PPN (MSB of first byte), 1 valid bit, rest unused

page table base register `0x10`; translate virtual address `0x376`

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	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x376` = `110 111 0110`

PTE 1: 0x10 + 6×2 = `0x1C`: `AC BC`

PTE 1: PPN `10` valid `1`

PTE 2: 0x20 + 7×2 = `0x2E`: `EF F0`

PTE 2: PPN `11` valid `1`

`11 0110` = `0x36` → `DB`

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE: 2 bit PPN (MSB of first byte), 1 valid bit, rest unused

page table base register 0x10; translate virtual address 0x376

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	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

0x376 = 110 111 0110

PTE 1: $0x10 + 6 \times 2 =$ 0x1C: AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 =$ 0x2E: EF F0

PTE 2: PPN 11 valid 1

11 0110 = 0x36 → DB

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE: 2 bit PPN (MSB of first byte), 1 valid bit, rest unused

page table base register `0x10`; translate virtual address `0x376`

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	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x376` $=$ `110 111 0110`

PTE 1: $\text{0x10} + 6 \times 2 =$ `0x1C`:

AC BC

PTE 1: PPN 10 valid 1

PTE 2: $\text{0x20} + 7 \times 2 =$ `0x2E`:

EF F0

PTE 2: PPN 11 valid 1

`11 0110` $=$ `0x36` $\rightarrow$ DB

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE: 2 bit PPN (MSB of first byte), 1 valid bit, rest unused

page table base register `0x10`; translate virtual address `0x376`

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	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

`0x376` $=$ `110 111 0110`

PTE 1: $0x10 + 6 \times 2 =$ `0x1C`: AC BC

PTE 1: PPN `10` valid `1`

PTE 2: $0x20 + 7 \times 2 =$ `0x2E`: EF F0

PTE 2: PPN `11` valid `1`

`11 0110` $=$ `0x36` $\rightarrow$ DB

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE: 2 bit PPN (MSB of first byte), 1 valid bit, rest unused

page table base register `0x10`; translate virtual address `0x376`

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	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

0x376 = 110 111 0110

PTE 1: 0x10 + 6×2 = 0x1C: AC BC

PTE 1: PPN 10 valid 1

PTE 2: 0x20 + 7×2 = 0x2E: EF F0

PTE 2: PPN 11 valid 1

11 0110 = 0x36 → DB

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE: 2 bit PPN (MSB of first byte), 1 valid bit, rest unused

page table base register `0x10`; translate virtual address `0x376`

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	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

0x376 $=$ 110 111 0110

PTE 1: $0x10 + 6 \times 2 =$ 0x1C: AC BC

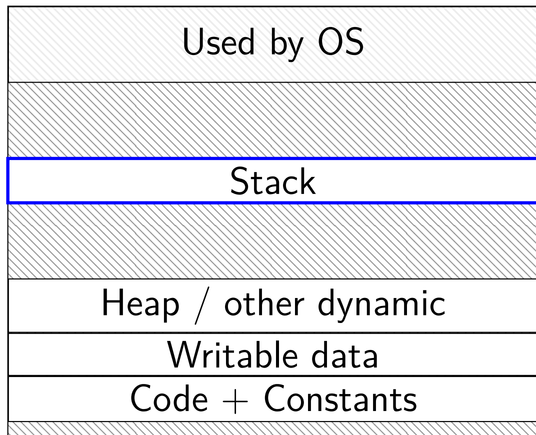
PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 =$ 0x2E: EF F0

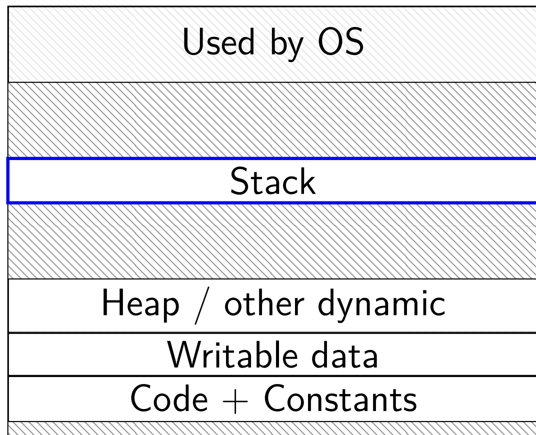
PTE 2: PPN 11 valid 1

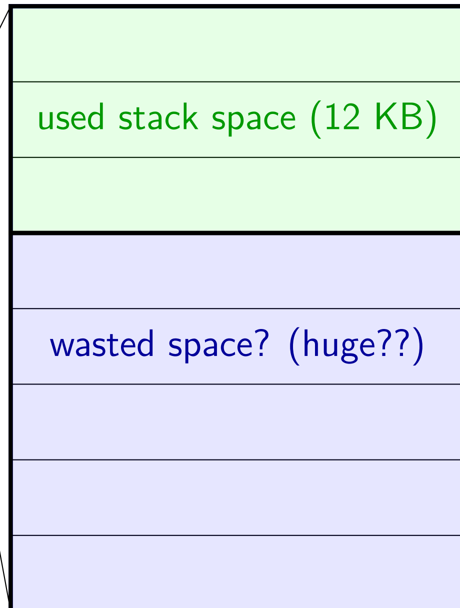
11 0110 $=$ 0x36 $\rightarrow$ DB

space on demand

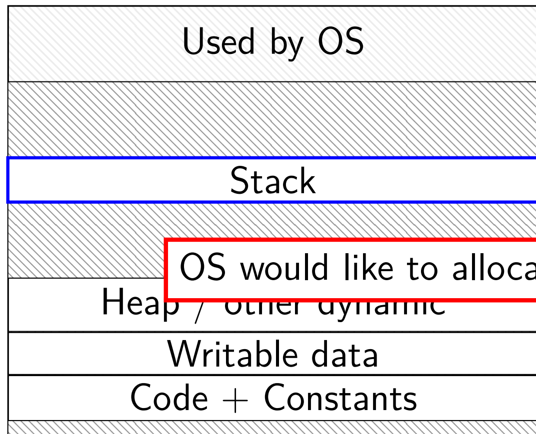


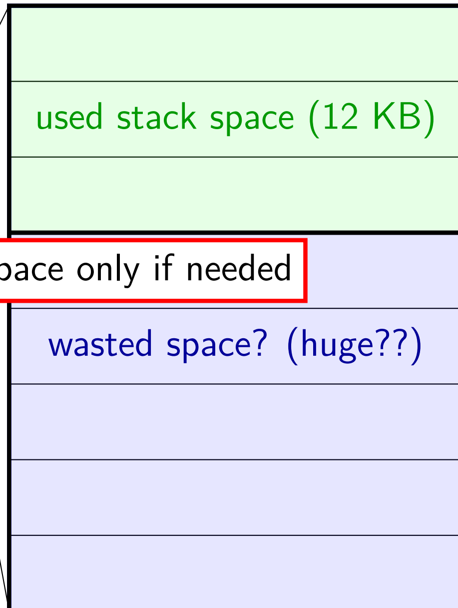
space on demand





space on demand





OS would like to allocate space only if needed

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
...
```

page fault!

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
...
```

restarted

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

swapping

early motivation for virtual memory: swapping

using disk (or SSD, …) as the next level of the memory hierarchy

- how our textbook and many other sources presents virtual memory

OS allocates program space on disk

- own mapping of virtual addresses to location on disk

DRAM is a cache for disk

swapping

early motivation for virtual memory: swapping

using disk (or SSD, ...) as the next level of the memory hierarchy

how our textbook and many other sources presents virtual memory

OS allocates program space on disk

own mapping of virtual addresses to location on disk

DRAM is a cache for disk

swapping versus caching

"cache block" ≈ physical page

fully associative

- every virtual page can be stored in any physical page

replacement/cache misses managed by the OS

normal cache hits happen in hardware

- hardware's page table lookup
- common case that needs to be very fast

swapping components

"swap in" a page — exactly like allocating on demand!

- OS gets page fault — invalid in page table
- check where page actually is (from virtual address)
- read from disk
- eventually restart process

"swap out" a page

- OS marks as invalid in the page table(s)
- copy to disk (if modified)

backup slides

cache accesses and multi-level PTs

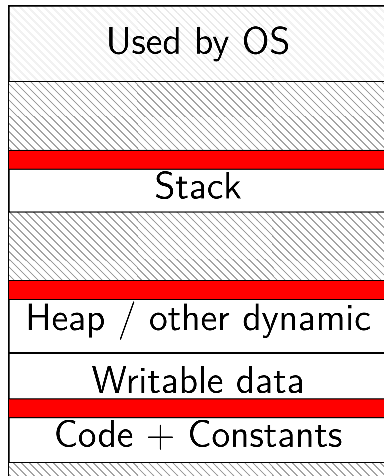
four-level page tables — five cache accesses per program memory access

L1 cache hits — typically a couple cycles each?

so add 8 cycles to each program memory access?

not acceptable

program memory active sets



0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F...

small areas of memory active at a time

one or two pages in each area?

0x0000 0000 0040 0000

page table entries and locality

page table entries have excellent temporal locality

typically one or two pages of the stack active

typically one or two pages of code active

typically one or two pages of heap/globals active

each page contains whole functions, arrays, stack frames, etc.

page table entries and locality

page table entries have excellent temporal locality

typically one or two pages of the stack active

typically one or two pages of code active

typically one or two pages of heap/globals active

each page contains whole functions, arrays, stack frames, etc.

needed page table entries are very small

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

only caches the page table lookup itself
(generally) just entries from the last-level page tables

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

not much spatial locality between page table entries
(they're used for kilobytes of data already)
(and if spatial locality, maybe use larger page size?)

page table entry cache

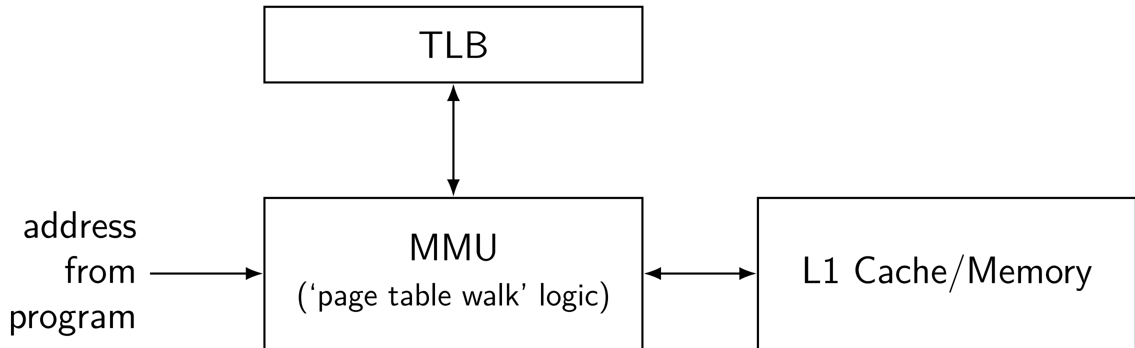
caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

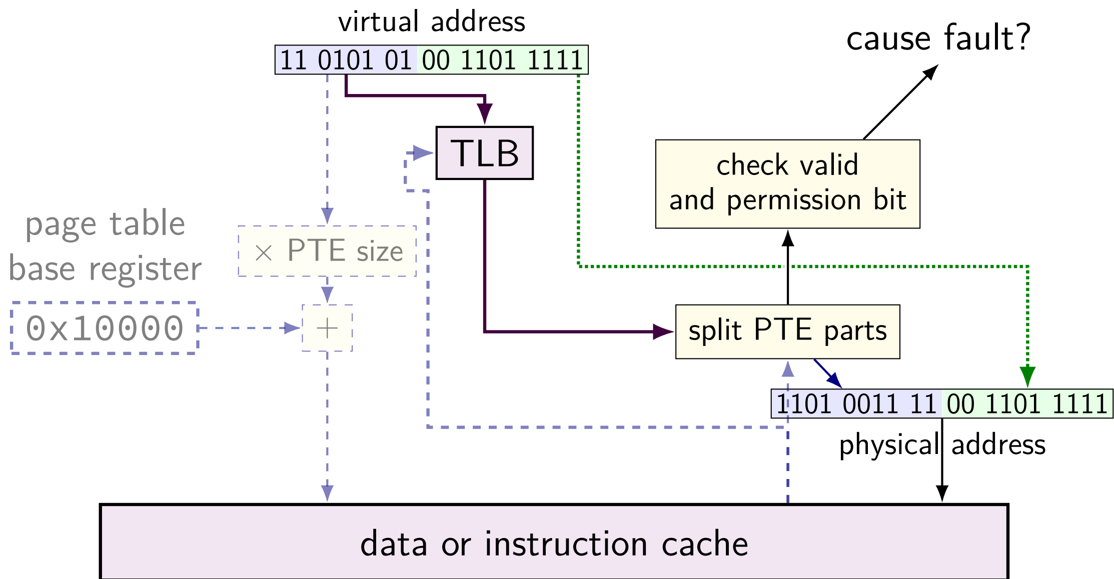
L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

few active page table entries at a time
enables highly associative cache designs

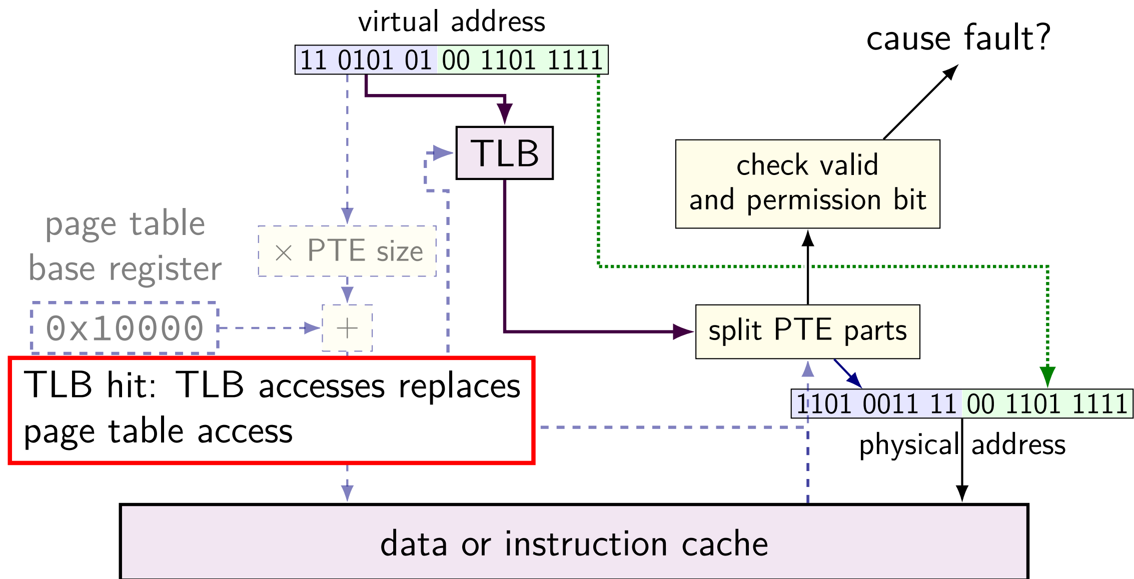
TLB and the MMU (1)



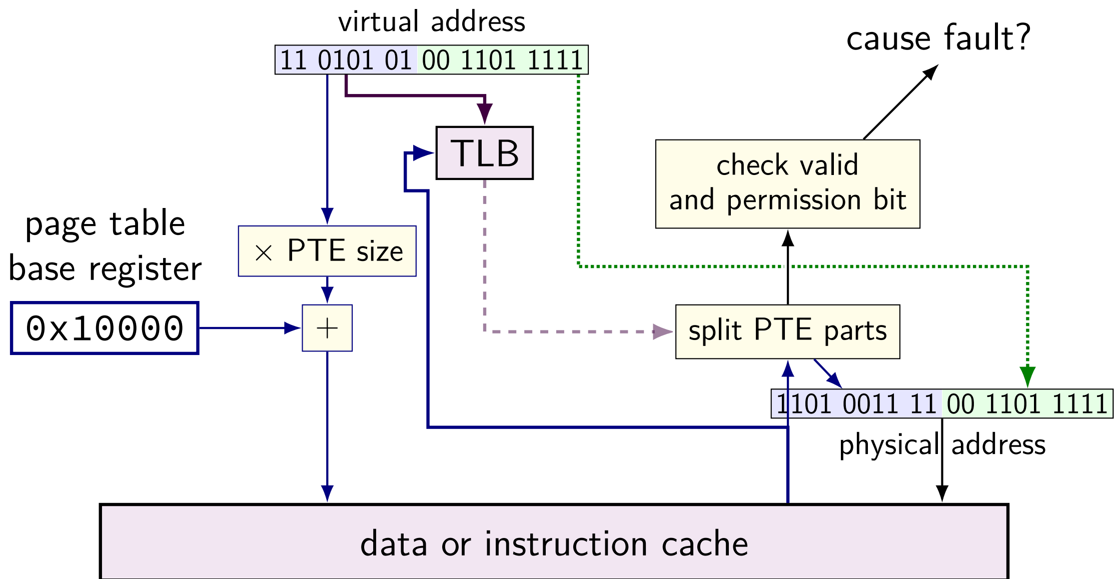
TLB and the MMU (2)



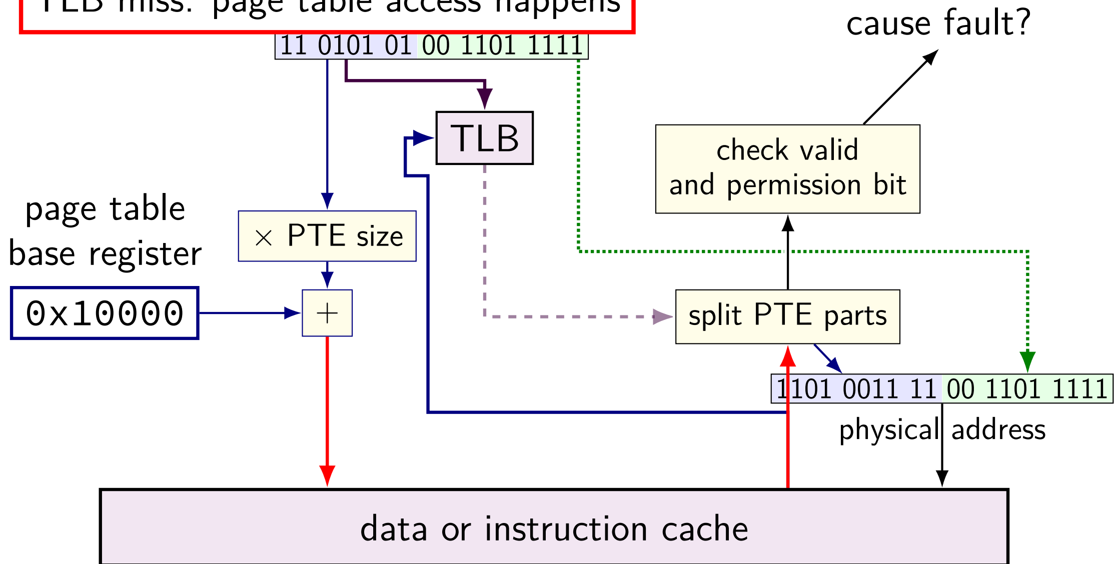
TLB and the MMU (2)



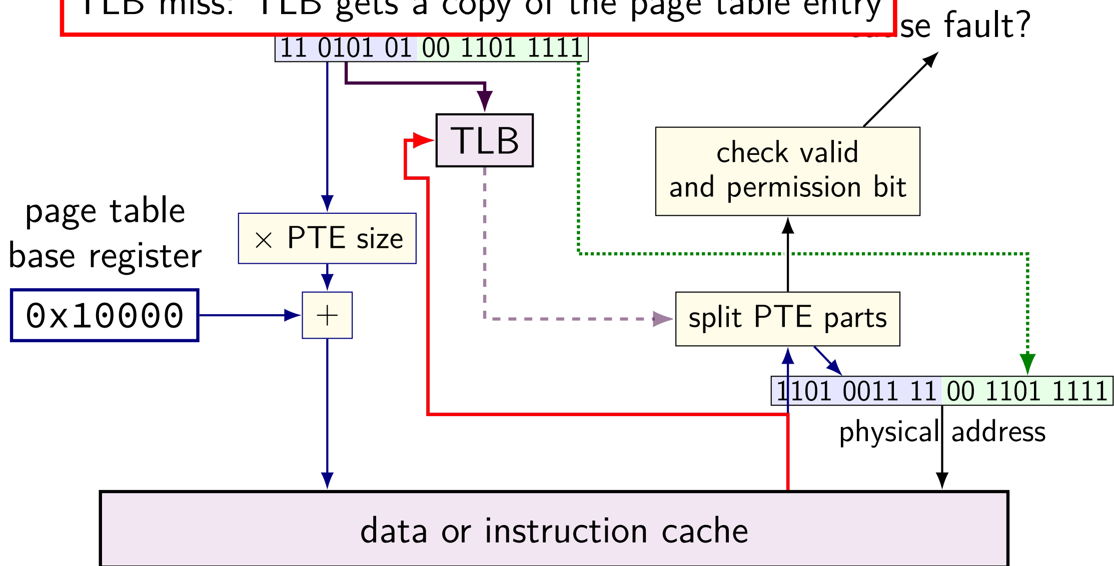
TLB and the MMU (2)



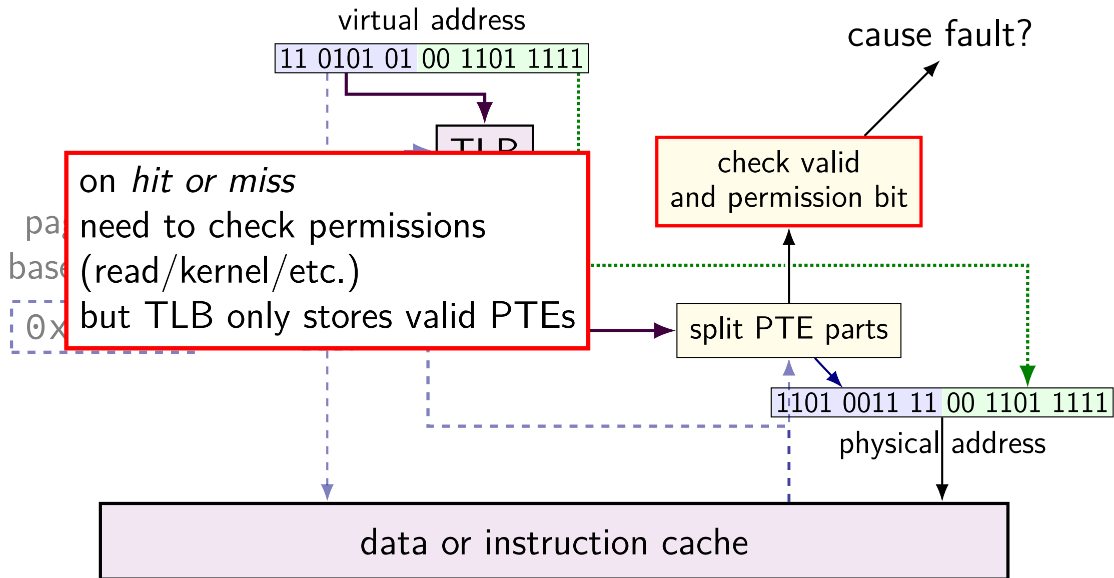
TLB and the MMU (2)



TLB and the MMU (2)



TLB and the MMU (2)



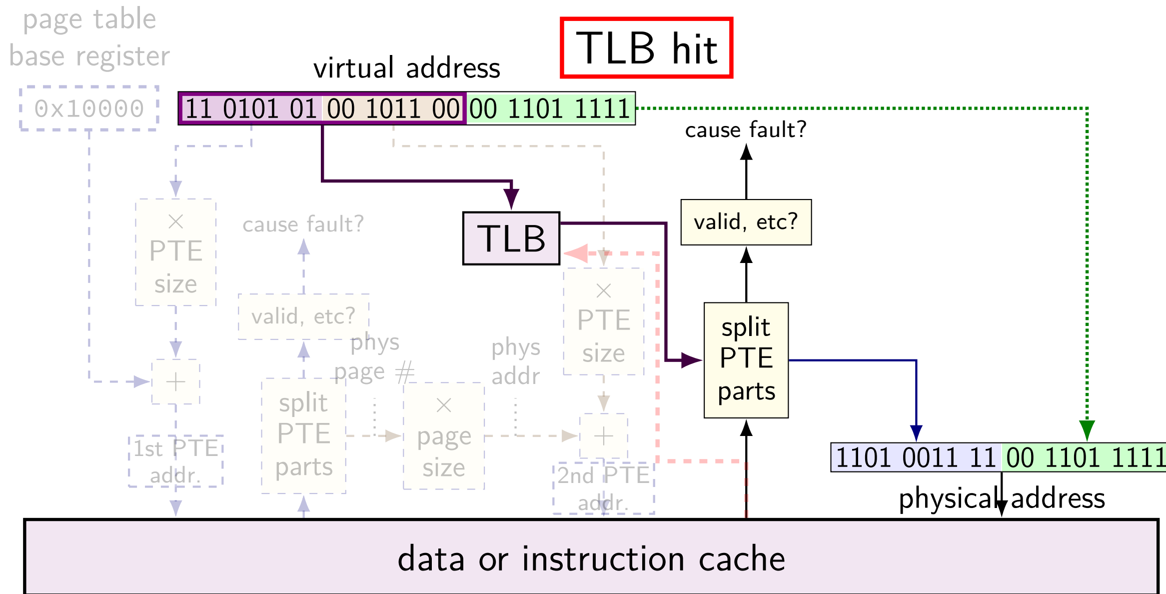
TLB and multi-level page tables

TLB caches valid last-level page table entries

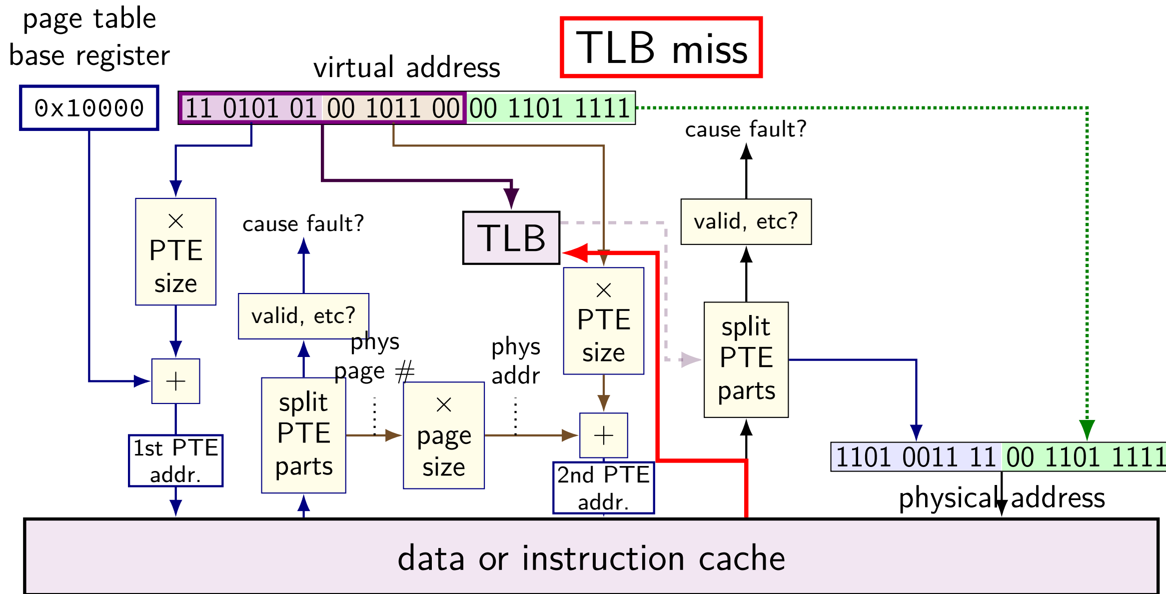
doesn't matter which last-level page table

means TLB output can be used directly to form address

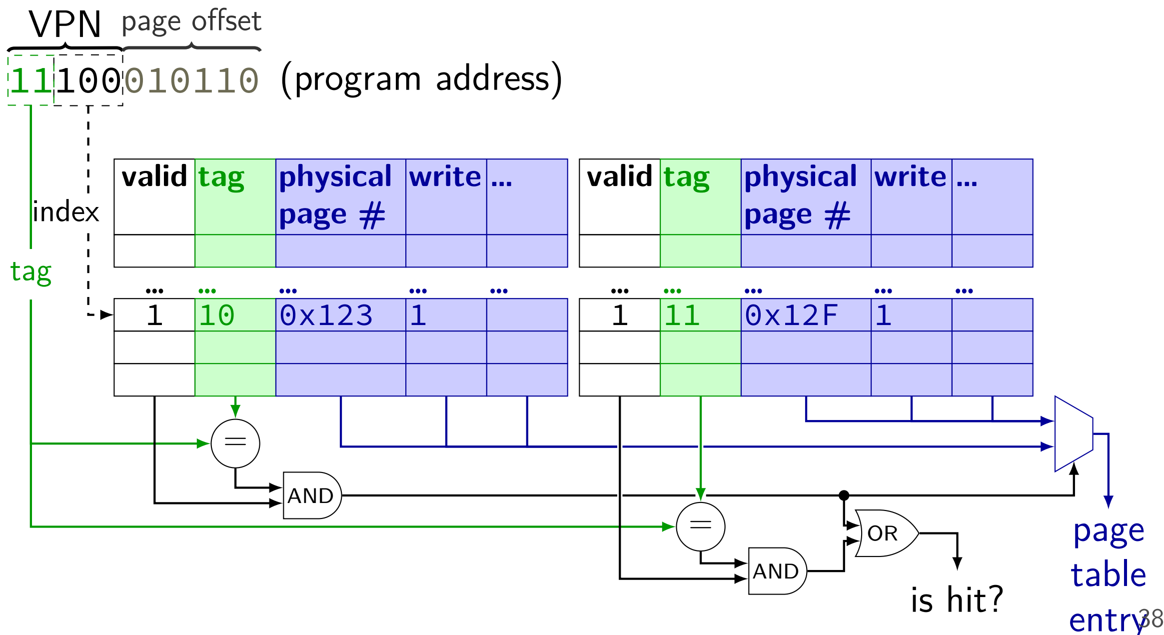
TLB and two-level lookup



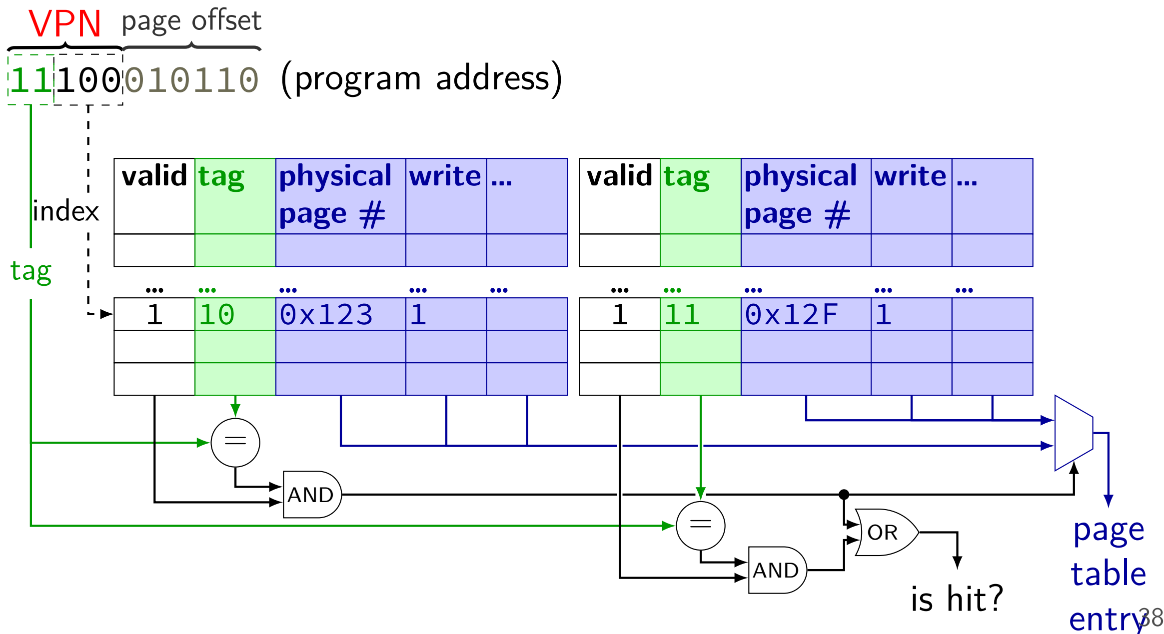
TLB and two-level lookup



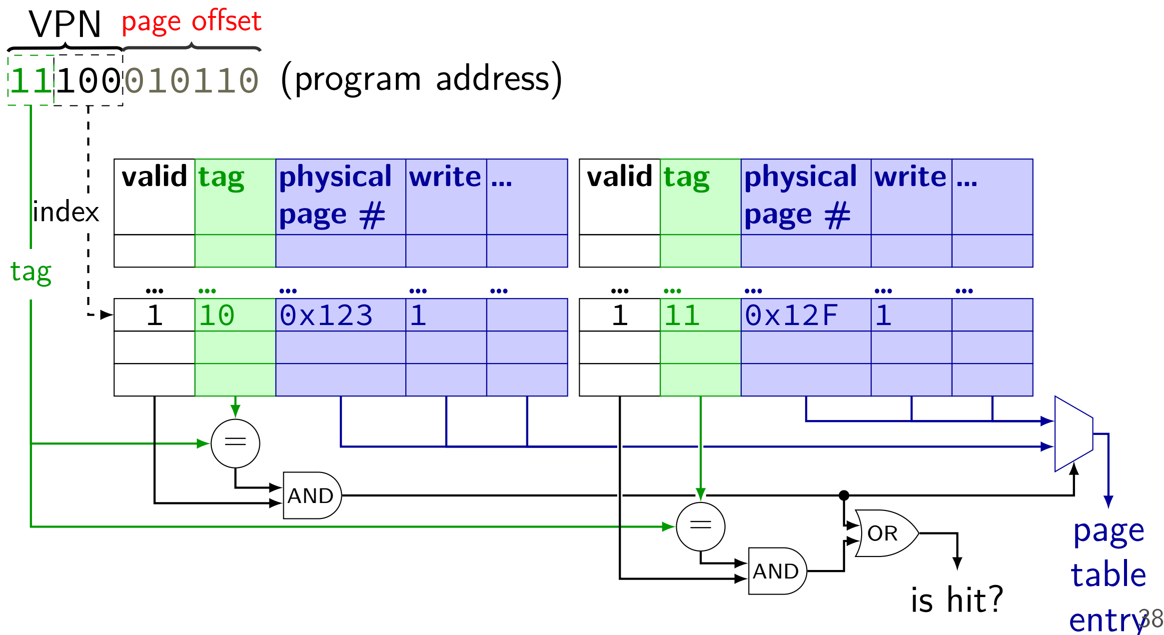
TLB organization (2-way set associative)



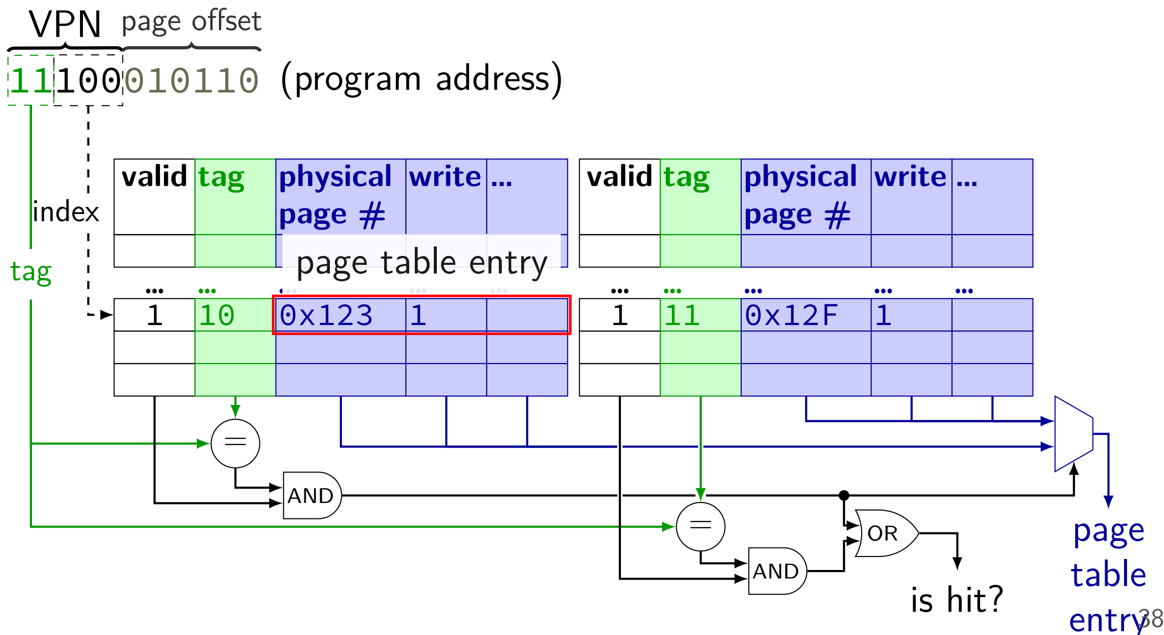
TLB organization (2-way set associative)



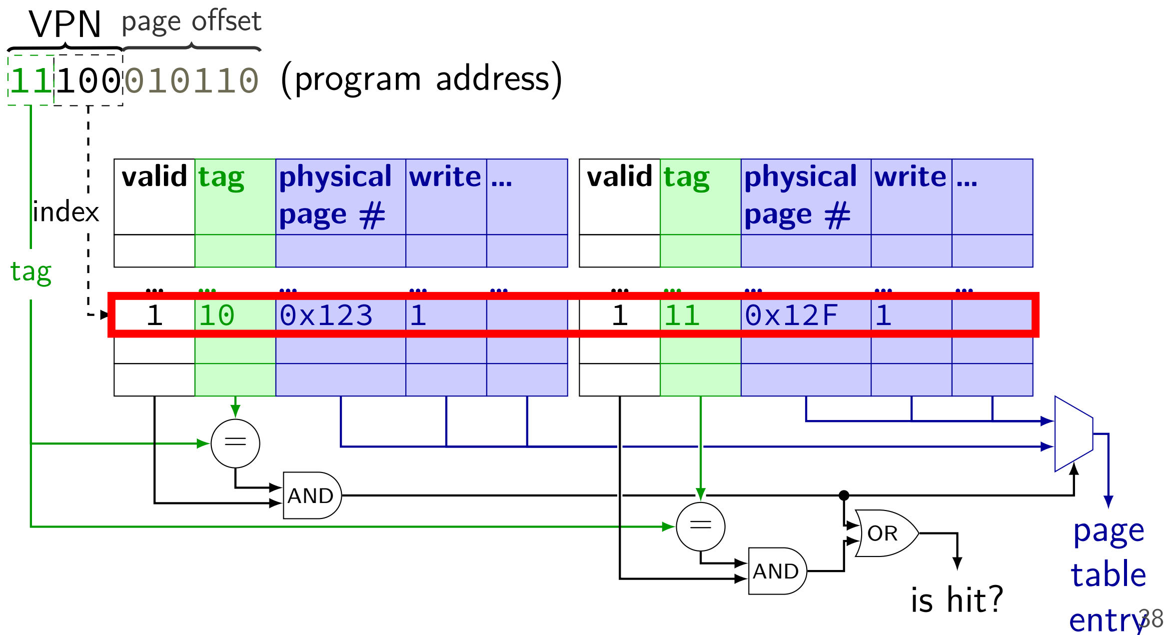
TLB organization (2-way set associative)



TLB organization (2-way set associative)



TLB organization (2-way set associative)



address splitting for TLBs (1)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

64-entry, 4-way L1 data TLB

TLB index bits?

TLB tag bits?

address splitting for TLBs (1)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

64-entry, 4-way L1 data TLB

TLB index bits?

$64/4 = 16$ sets — 4 bits

TLB tag bits?

$48 - 12 = 36$ bit virtual page number — $36 - 4 = 32$ bit TLB tag

address splitting for TLBs (2)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

1536-entry ($3 \cdot 2^9$), 12-way L2 TLB

TLB index bits?

TLB tag bits?

address splitting for TLBs (2)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

1536-entry ($3 \cdot 2^9$), 12-way L2 TLB

TLB index bits?

$1536/12 = 128$ sets — 7 bits

TLB tag bits?

$48 - 12 = 36$ bit virtual page number — $36 - 7 = 29$ bit TLB tag

exercise: TLB access pattern (setup)

4-entry, 2-way TLB, LRU replacement policy, initially empty

4096 byte pages

how many index bits?

TLB index of virtual address 0x12345?

exercise: TLB access pattern

4-entry, 2-way TLB, LRU replacement policy, initially empty

4096 byte pages

type	virtual	physical
read	0x440030	0x554030
write	0x440034	0x554034
read	0x7FFFE008	0x556008
read	0x7FFFE000	0x556000
read	0x7FFFDFF8	0x5F8FF8
read	0x664080	0x5F9080
read	0x440038	0x554038
write	0x7FFFDFF0	0x5F8FF0

which are TLB hits? which are TLB misses? final contents of TLB?

exercise: TLB access pattern

4-entry, 2-way TLB, LRU replacement policy, initially empty

4096 byte pages

				VPNs of PTEs held in TLB	
type	virtual	physical	result	set 0	set 1
read	0x440030	0x554030	miss	0x440	
write	0x440034	0x554034	hit	0x440	
read	0x7FFFE008	0x556008	miss	0x440	
read	0x7FFFE000	0x556000	hit	0x440, 0x7FFFE	
read	0x7FFFDFF8	0x5F8FF8	miss	0x440, 0x7FFFE	0x7FFFD
read	0x664080	0x5F9080	miss	0x664, 0x7FFFE	0x7FFFD
read	0x440038	0x554038	miss	0x664, 0x440	0x7FFFD
write	0x7FFFDFF0	0x5F8FF0	hit	0x664, 0x440	0x7FFFD

which are TLB hits? which are TLB misses? final contents of TLB?

exercise: TLB access pattern

4-entry, 2-way TLB, LRU replacement policy, initially empty

4096 byte pages

set idx	V	tag	physical page	write?	kernel?	...	LRU?
0	1	0x00220 (0x440 $\gg$ 1)	0x554	1	0	...	no
	1	0x00332 (0x00664 $\gg$ 1)	0x5F9	1	0	...	yes
1	1	0x3FFFF (0x7FFFD $\gg$ 1)	0x5F8	1	0	...	no
	0	---	---	–	–	...	yes

type					
read					
writ					
read					
read					
read					
read	0x664080	0x5F9080	miss	0x664, 0x7FFFE	0x7FFFD
read	0x440038	0x554038	miss	0x664, 0x440	0x7FFFD
write	0x7FFFDFF0	0x5F8FF0	hit	0x664, 0x440	0x7FFFD

which are TLB hits? which are TLB misses? final contents of TLB?

changing page tables

what happens to TLB when page table base pointer is changed?

- e.g. context switch

most entries in TLB refer to things from wrong process

- oops — read from the wrong process's stack?

changing page tables

what happens to TLB when page table base pointer is changed?

- e.g. context switch

most entries in TLB refer to things from wrong process

- oops — read from the wrong process's stack?

option 1: invalidate all TLB entries

- side effect on "change page table base register" instruction

changing page tables

what happens to TLB when page table base pointer is changed?

- e.g. context switch

most entries in TLB refer to things from wrong process

- oops — read from the wrong process's stack?

option 1: invalidate all TLB entries

- side effect on "change page table base register" instruction

option 2: TLB entries contain process ID

- set by OS (special register)
- checked by TLB in addition to TLB tag, valid bit

editing page tables

what happens to TLB when OS changes a page table entry?

most common choice: has to be handled in software

editing page tables

what happens to TLB when OS changes a page table entry?

most common choice: has to be handled in software

invalid to valid — nothing needed

- TLB doesn't contain invalid entries
- MMU will check memory again

valid to invalid — OS needs to tell processor to invalidate it

- special instruction (x86: `invlpg`)

valid to other valid — OS needs to tell processor to invalidate it

x86-64 page table entries (1)

63	62–59	58–52	51–M	M-1–12	11–9	8	7	6	5	4	3	2	1	0	
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

present = valid

R/W = writes allowed?

U/S = user-mode allowed? (“user/supervisor”)

XD = execute-disable?

A = accessed? (MMU sets to 1 on page read/write)

D = dirty? (MMU sets to 1 on page write)

x86-64 page table entries (1)

63	62 61 60 59	58 57 56 55 54 53 52	51 M[1]	M-1 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0	
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

present = valid

R/W = writes allowed?

U/S = user-mode allowed? (“user/supervisor”)

XD = execute-disable?

A = accessed? (MMU sets to 1 on page read/write)

D = helps support replacement policies for swapping

x86-64 page table entries (1)

63	62 61 60 59	58 57 56 55 54 53 52	51 … M[1]	M-1 … 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0	
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

present = valid

R/W = writes allowed?

U/S = user-mode allowed? (“user/supervisor”)

XD = execute-disable?

A = accessed? (MMU sets to 1 on page read/write)

D = dirty? (MMU sets to 1 on page write)

helps support writeback policy for swapping

x86-64 page table entries (2)

63	62 61 60 59	58 57 56 55 54 53 52	51 … M[1]	M-1 … 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0	
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

G = global? (shared between all page tables)

PWT, PCD, PAT = control how caches work when accessing physical page:

- can disable using the cache entirely
- can disable write-back (use write-through instead)
- multicore-related cache settings
- (and some other settings)

x86-64 page table entries (2)

63	62–59	58–52	51–M	M-1–12	11–9	8	7	6	5	4	3	2	1	0	
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

G = global? (shared between all page tables)

CPU won't evict TLB entries on most page table base registers changes

P

can disable using the cache entirely

can disable write-back (use write-through instead)

multicore-related cache settings

(and some other settings)

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address `0x12345678`

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address `0x12345678`

- `0001 0010 0011 0100 0101 0110 0111 1000`

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address `0x12345678`

- `0001 0010 0011 0100 0101 0110 0111 1000`
- 13-bit page offset `1 0110 0111 1000`

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

256 entries in first level; 2048 in second

split the address `0x12345678`

`0001 0010 0011 0100 0101 0110 0111 1000`

13-bit page offset `1 0110 0111 1000`

$32 - 13 = 19$-bit VPN `0001 0010 0011 0100 010`

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address `0x12345678`

- `0001 0010 0011 0100 0101 0110 0111 1000`
- 13-bit page offset `1 0110 0111 1000`
- $32 - 13 = 19$-bit VPN `0001 0010 0011 0100 010`
- 8-bit first part of VPN `0001 0010`

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address `0x12345678`

- `0001 0010 0011 0100 0101 0110 0111 1000`
- 13-bit page offset `1 0110 0111 1000`
- $32 - 13 = 19$-bit VPN `0001 0010 0011 0100 010`
- 8-bit first part of VPN `0001 0010`
- 11-bit second part of VPN `0011 0100 010`

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address `0x12345678`

- `0001 0010 0011 0100 0101 0110 0111 1000`
- 13-bit page offset `1 0110 0111 1000`
- $32 - 13 = 19$-bit VPN `0001 0010 0011 0100 010`
- 8-bit first part of VPN `0001 0010`
- 11-bit second part of VPN `0011 0100 010`
- 7-bit TLB index 0100 010

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

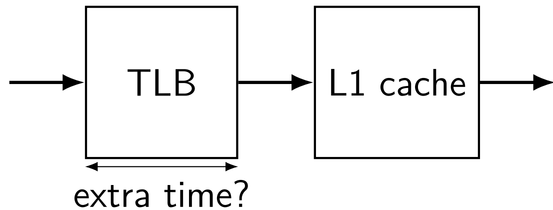
2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address `0x12345678`

- `0001 0010 0011 0100 0101 0110 0111 1000`
- 13-bit page offset `1 0110 0111 1000`
- $32 - 13 = 19$-bit VPN `0001 0010 0011 0100 010`
- 8-bit first part of VPN `0001 0010`
- 11-bit second part of VPN `0011 0100 010`
- 7-bit TLB index 0100 010
- $19 - 7 = 12$-bit TLB tag 0001 0010 0011

TLBs and performance



L1 caches and page numbers (Intel Skylake)

<table>
<tr><td colspan="3">physical address
(48 bits)</td></tr>
<tr><td>PPN
(36 bit)</td><td colspan="2">page offset
(12 bit)</td></tr>
<tr><td>L1 cache tag
(36 bit)</td><td>L1 index
(6 bit)</td><td>L1 offset
(6 bit)</td></tr>
</table>

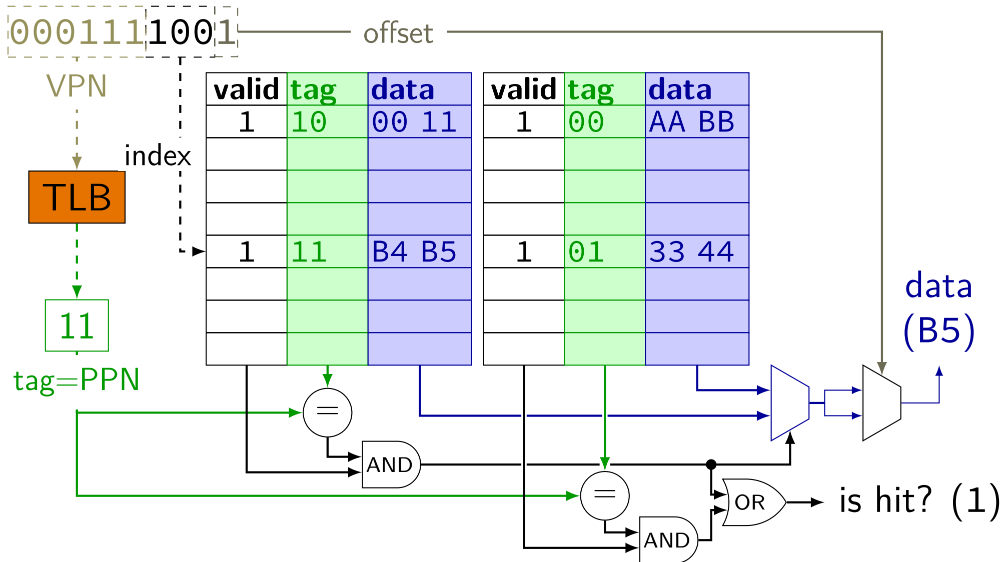
L1 caches and page numbers (Intel Skylake)

<table>
<tr><td colspan="3">physical address
(48 bits)</td></tr>
<tr><td>PPN
(36 bit)</td><td colspan="2">page offset
(12 bit)</td></tr>
<tr><td>L1 cache tag
(36 bit)</td><td>L1 index
(6 bit)</td><td>L1 offset
(6 bit)</td></tr>
</table>

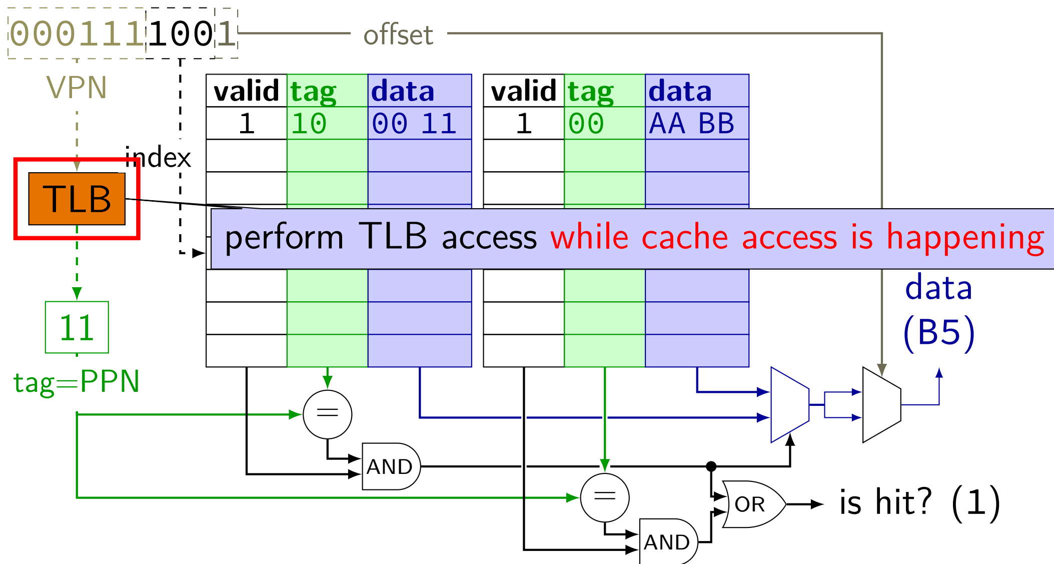
not a coincidence

why did Intel make this decision?

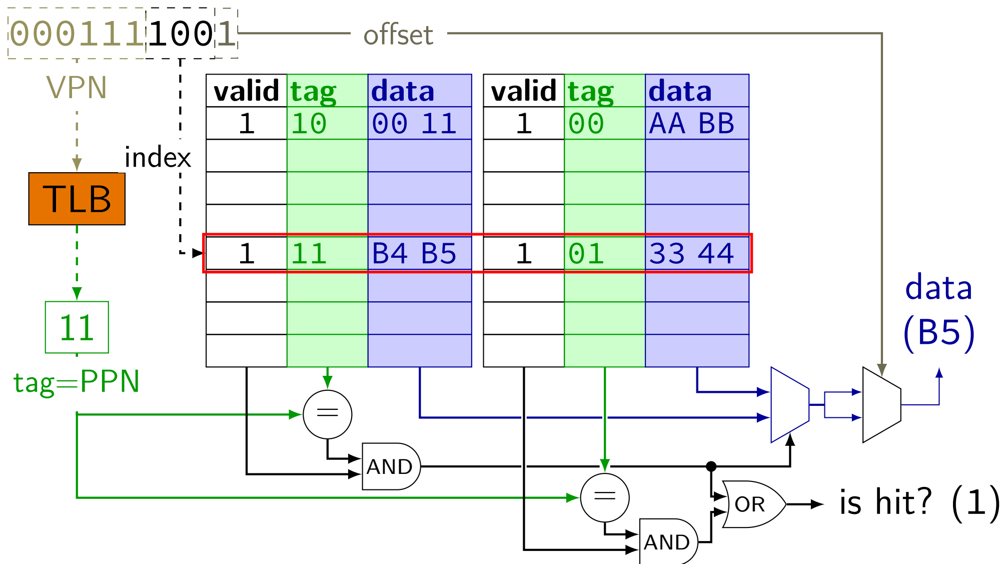
overlapping TLB and cache access



overlapping TLB and cache access



overlapping TLB and cache access



virtually-indexed, physically-tagged

called virtually-indexed, physically-tagged cache

requirement: index contained entirely in page offset

- do not need to do translation to start cache access

tag overlaps with PPN

- example: tag=PPN
- (but tag could include part of page offset, too)

do TLB access while retrieving cache set

most common design in current processors

reason for highly associative (e.g. 8-way) L1 caches

address splitting

16-bit virtual addresses

64-byte pages

256B, 8-way L1 cache with 16B blocks

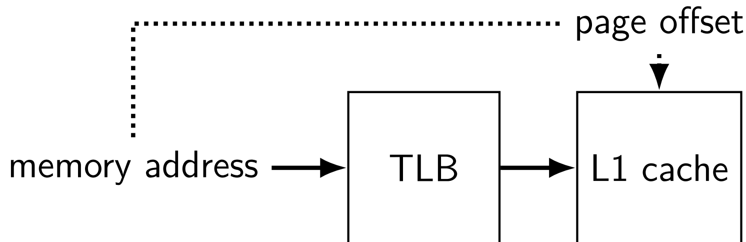
can TLB and cache access overlap?

physical caches

so far: caches use physical addresses:

means cache lookup can't complete without TLB

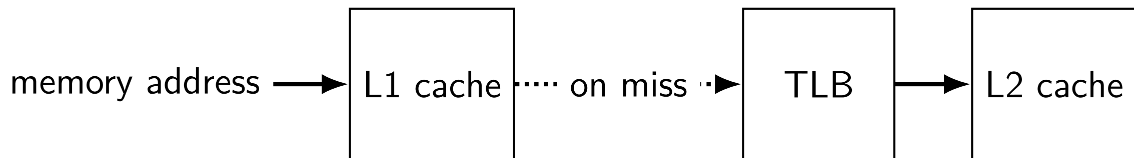
(and can't start without index from physical address)



virtual indexing

alternate option: have caches hold virtual addresses *and match tags* with virtual addresses

advantage: don't need to wait for TLB lookup at all



but some things more complicated:

- need to invalidate caches on page table changes
- need to deal multiple VPNs for same physical page ("aliasing")

virtual caches: aliasing problem

two virtual addresses can map to same physical

problem for caches with virtual indexes+tags

say VA 0x1000 and 0x2000 map to same physical

what happens if application writes to 0x1000

…then reads from 0x2000?

virtual cache aliasing solutions

software solution: OS promises not to have aliasing

hardware solution: hardware detects aliasing

- requires extra bookkeeping

hardware alias detection (1)

key idea: store physical address *and virtual-address-based tag*

store value? check for other copies of same physical addr

if found: evict other copies

index	V	tag	phys addr	value	V	tag	phys addr	value
0	1	0x03312	0x1F4300	…	1	0x03311	0x1F3400	…
1	1	0x7FF33	0x183220	…	1	0x03310	0x0F3A20	…
2	1	0x7FF33	0x183240	…	1	0x03320	0x030040	…
…								

checking all the physical addresses? (1)

scanning entire cache for same physical address?

do we really need to scan everything?

checking all the physical addresses? (1)

scanning entire cache for same physical address?

do we really need to scan everything?

exercise(1): if we have 4096 (2^{12}) byte pages, give an example of a virtual address that could map to the same physical as `0x12010`?

checking all the physical addresses? (1)

scanning entire cache for same physical address?

do we really need to scan everything?

exercise(1): if we have 4096 (2^{12}) byte pages, give an example of a virtual address that could map to the same physical as `0x12010`?

must have same page offset, so 0x0010, 0x1010, 0x2010, 0x3010, etc.

checking all the physical addresses? (2)

scanning entire cache for same physical address?

do we really need to scan everything?

with 4K pages (2^{12} byte):

exercise(2): if we have a direct-mapped 4K **virtual** cache with 16 byte blocks, in which sets can be these aliasing addresses be stored?

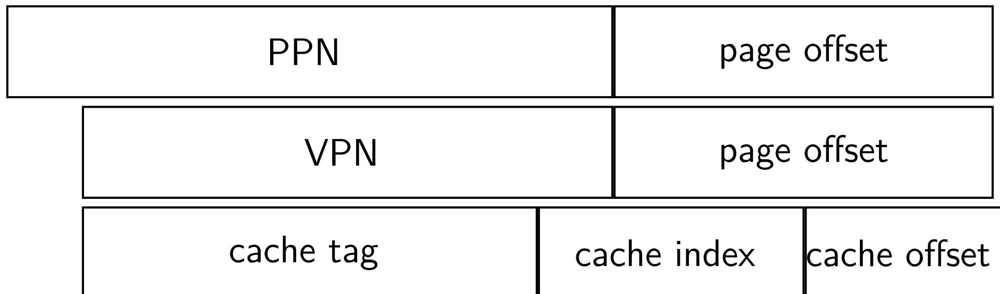
PPN		page offset
VPN		page offset
cache tag	cache index	cache offset

checking all the physical addresses? (3)

scanning entire cache for same physical address?

do we really need to scan everything?

exercise(3): if we have a direct-mapped **8K virtual** cache with 16-byte blocks, in which sets can these aliasing addresses be stored?

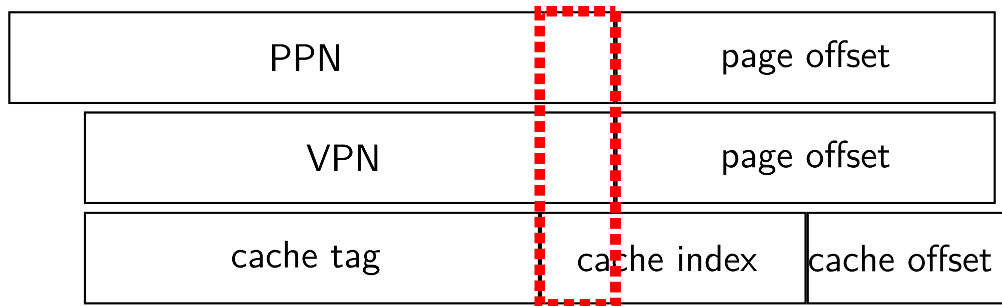


checking all the physical addresses? (3)

scanning entire cache for same physical address?

do we really need to scan everything?

exercise(3): if we have a direct-mapped **8K virtual** cache with 16-byte blocks, in which sets can these aliasing addresses be stored?

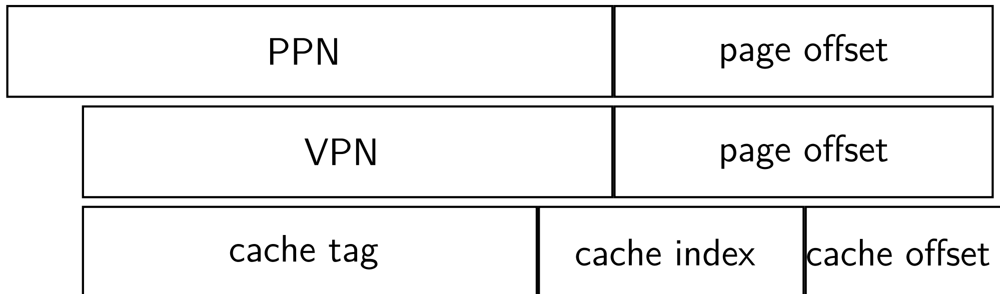


virtual caches: detecting aliasing

suppose two virtual addresses with same physical address

addrs differ only VPN bits

finding which indexes to check: worry about overlap:

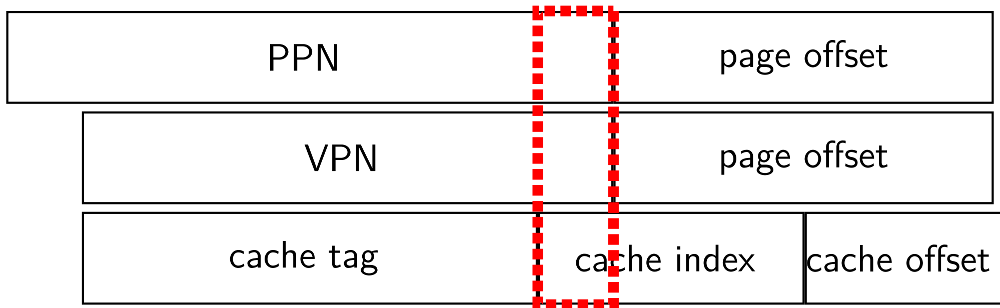


virtual caches: detecting aliasing

suppose two virtual addresses with same physical address

addrs differ only VPN bits

finding which indexes to check: worry about overlap:



virtual caches: aliasing detection outline

hardware alias detection mechanism

store physical address of each block in the cache

...in addition to virtual tag

when adding value to cache, try all other indexes for same page offset

if any match physical address: evict that copy

result: only store one virtual address per physical address at a time

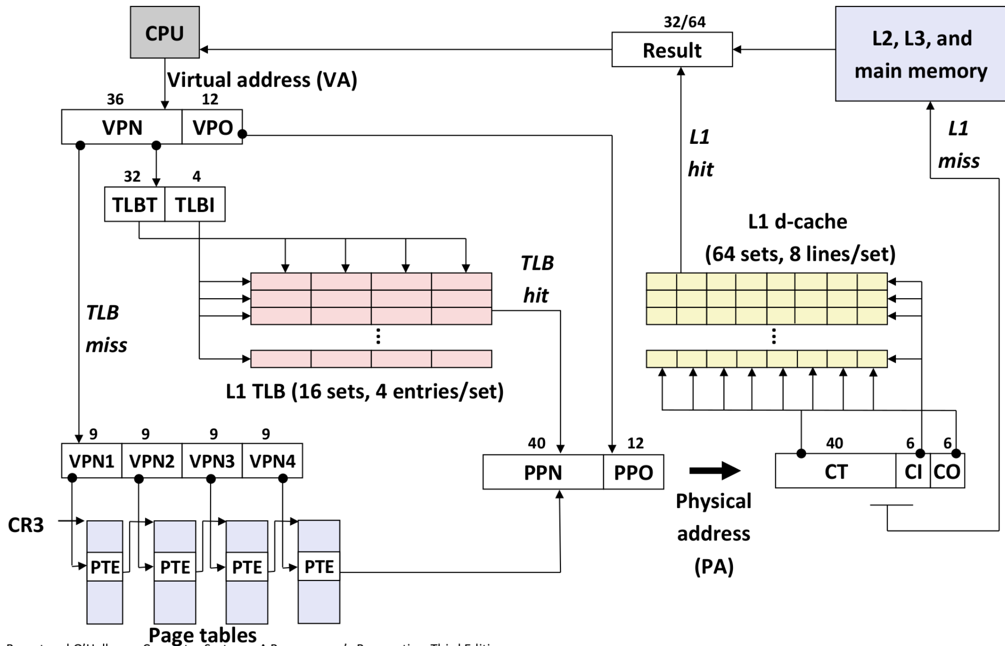
actual strategy used by AMD Opteron's instruction cache

2 cache index bits overlapping VPN

4 cache sets to check on cache replacement

(also on conflicting writes to data cache?)

book’s diagram



are pages too small?

program accessing a lot of memory? lots of TLB entries

- lots of TLB misses
- lots of space for page tables

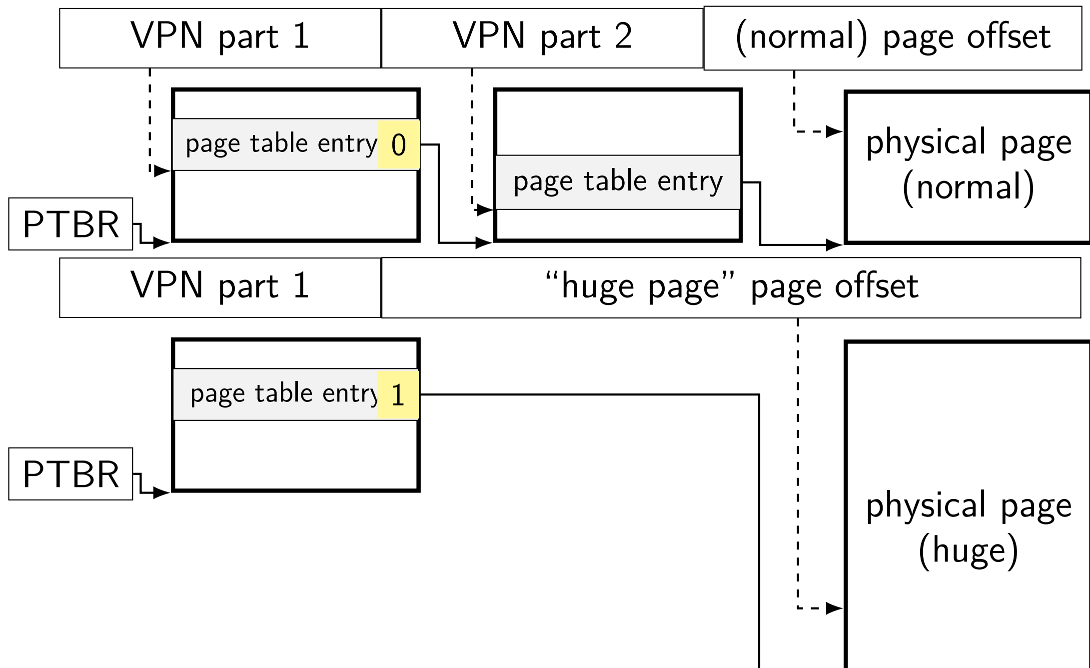
often, really like larger pages,...

but not all the time

- still want to be able to allocate small amounts of memory to small programs
- want to load/store small amounts of data from disk

common to provide variable-sized pages for OS

“huge pages”



big pages on x86-64

option for 2MB or 1GB pages instead of 4KB pages

first, second, third-level page table entries can point to either

- next page table (normal case), or
- a "huge page"

big changes to TLB needed?

- Intel/AMD don't seem to disclose what they do,...
- but seems like Intel has a special cache for non-fourth(last)-level PTEs
- (which speeds up page tabe lookup for normal pages)
- and use this for 'huge page' PTEs, too

processes can have mix of huge and normal apges

why big pages?

TLB misses can create same sort of problems as cache misses

can do cache blocking to help with TLB misses but...

big pages are relatively easy to implement

might dramatically reduce TLB misses

page table space exercise (1)

4-level page table

512 PTEs of 8 bytes each for each page table

suppose a process has exactly one page allocated

how much space for page tables?

page table space exercise (1)

4-level page table

512 PTEs of 8 bytes each for each page table

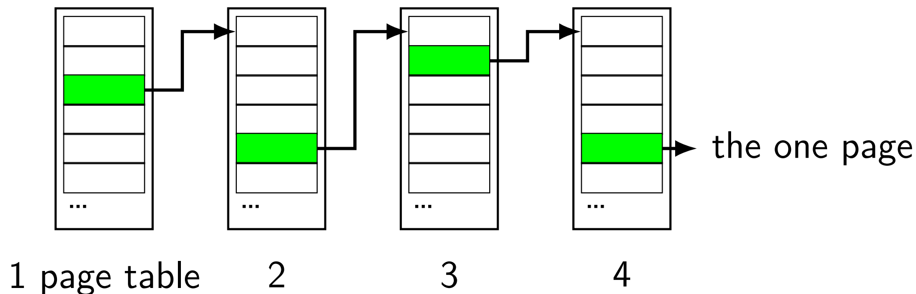
suppose a process has exactly one page allocated

how much space for page tables?

1 page at each level (4KB each)

exactly one valid entry in each of them

page table space exercise (1)



4 page tables at 1 page/page table
plus 1 page of data
5 pages total

page table space exercise (2)

4-level page table

512 PTEs of 8 bytes each for each page table

suppose a process has exactly two pages allocated:

- one at address 0x0, one at address `0x20000000000`

how much space for page tables?

page table space exercise (2)

4-level page table

512 PTEs of 8 bytes each for each page table

suppose a process has exactly two pages allocated:

- one at address 0x0, one at address `0x20000000000`

how much space for page tables?

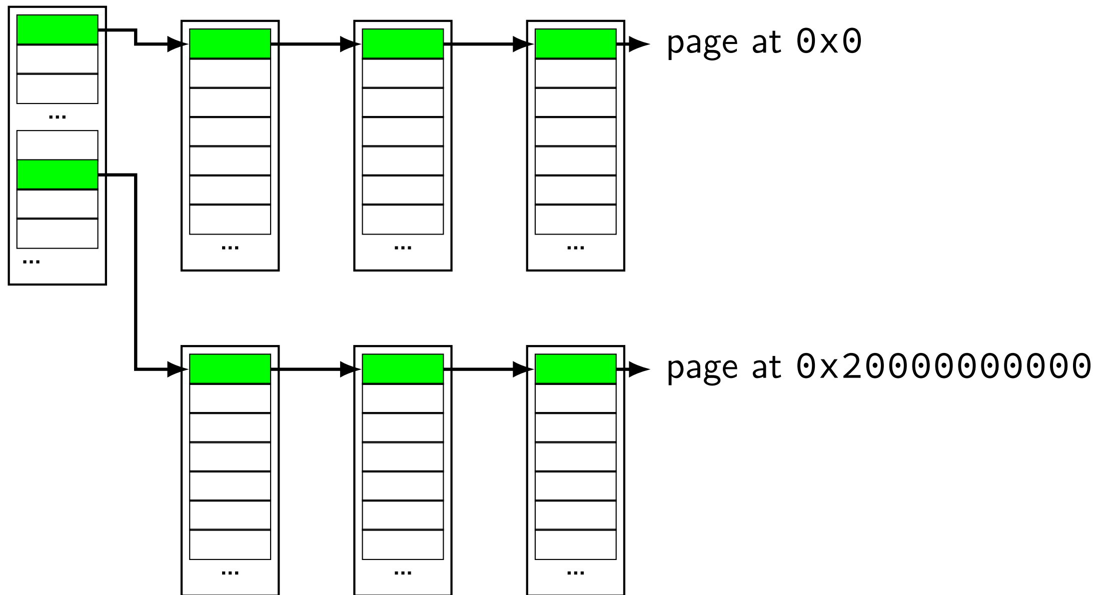
1 shared first-level PT, with two valid entries

two second-level PTs, each with one valid entry

two third-level PTs, each with one valid entry

two fourth-level PTs, each with one valid entry

page table space exercise (2)



page table space exercise (3)

4-level page table; each PT: 512 PTEs of 8 bytes

suppose a process has 100 pages of stack, 100 pages of code+constants (contiguous)

- stack and code+constants far apart

how much space for page tables?

page table space exercise (3)

4-level page table; each PT: 512 PTEs of 8 bytes

suppose a process has 100 pages of stack, 100 pages of code+constants (contiguous)

- stack and code+constants far apart

how much space for page tables? — *minimum:*

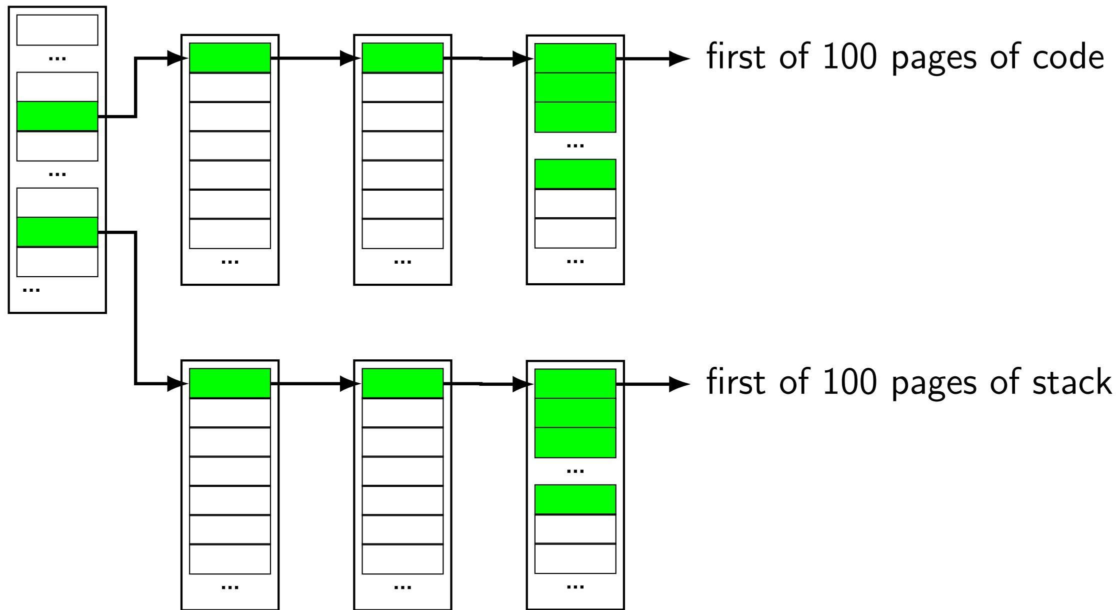
1 shared first-level PT, with two valid entries

two second-level PT, each with one valid entry

two third-level PT, each with one valid entry

two fourth-level PT, each with 100 valid entries

page table space exercise (3)



page table space exercise (3)

4-level page table; each PT: 512 PTEs of 8 bytes

suppose a process has 100 pages of stack, 100 pages of code+constants (contiguous)

how much space for page tables?

page table space exercise (3)

4-level page table; each PT: 512 PTEs of 8 bytes

suppose a process has 100 pages of stack, 100 pages of code+constants (contiguous)

how much space for page tables? — *maximum:*

1 shared first-level PT, with four valid entries

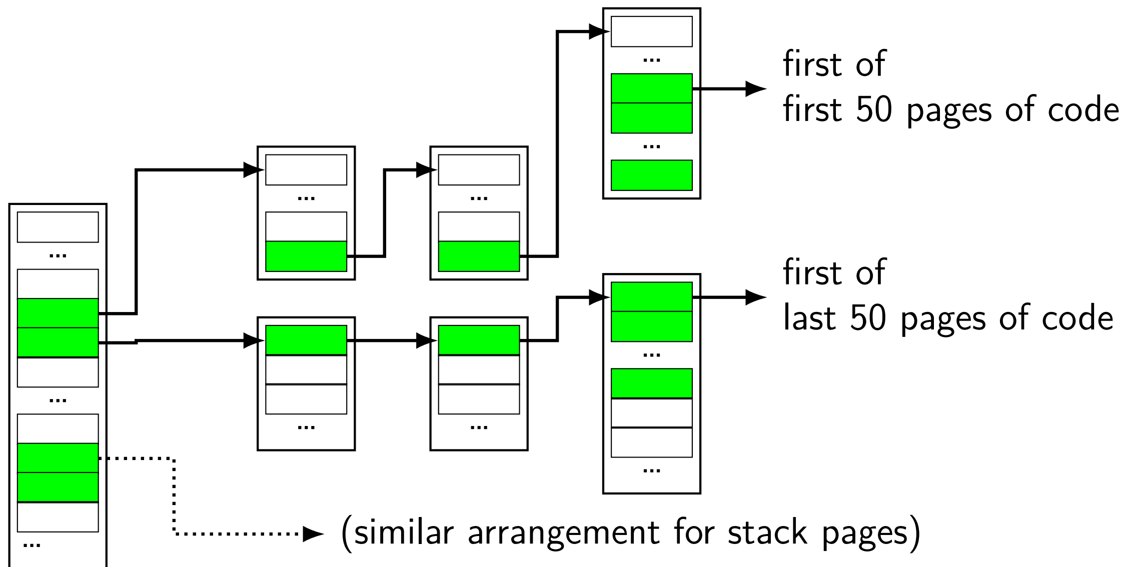
four second-level PT, each with one valid entry

- two for stack, two for code+constants

four third-level PT, each with one valid entry

four fourth-level PT, each with 50 valid entries

page table space exercise (3)



page table space exercise (4)

4-level page table; each PT: 512 PTEs of 8 bytes

suppose a process has 200 pages, randomly distributed in PT

about how much space for page tables?

page table space exercise (4)

4-level page table; each PT: 512 PTEs of 8 bytes

suppose a process has 200 pages, randomly distributed in PT

about how much space for page tables?

about 165 ($\pm \sim 8$) entries in first-level PT

(some pages randomly share first-level PT entries)

about 165 second-level PTs, 200 third-level, 200 fourth-level

a bit less than 600 page tables — almost 2400 KB

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

- minimum size: 512 bytes
- writing tens of kilobytes basically as fast as writing 512 bytes

SSD writes and writes: hundreds of microseconds

- designed for writes/reads of kilobytes (not much smaller)

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

- minimum size: 512 bytes
- writing tens of kilobytes basically as fast as writing 512 bytes

SSD writes and writes: hundreds of microseconds

- designed for writes/reads of kilobytes (not much smaller)

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

- minimum size: 512 bytes
- writing tens of kilobytes basically as fast as writing 512 bytes

SSD writes and writes: hundreds of microseconds

- designed for writes/reads of kilobytes (not much smaller)

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

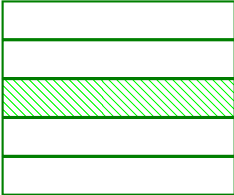
- minimum size: 512 bytes
- writing tens of kilobytes basically as fast as writing 512 bytes

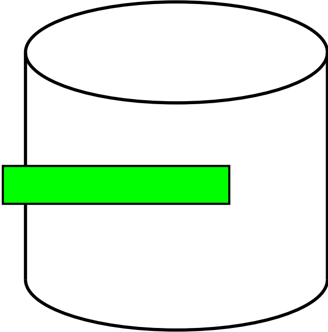
SSD writes and writes: hundreds of microseconds

- designed for writes/reads of kilobytes (not much smaller)

swapping timeline

program A pages





program B page

page fault

disk

swapping timeline

program A pag

OS needs to choose page to replace
hopefully copy on disk is already up-to-date?

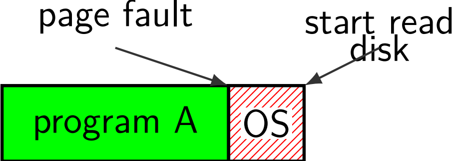
gram B page

loaded

evicted (to free space)

...

...



swapping timeline

program A pages

...

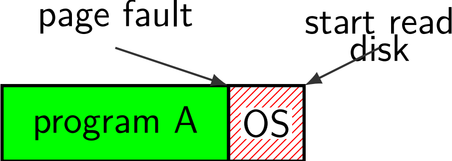
first step of replacement:
mark evicted page invalid in page table

loaded

evicted (to free space)

program B page

...



swapping timeline

program A pages

other processes can run while reading page
OS will get interrupt when disk is done

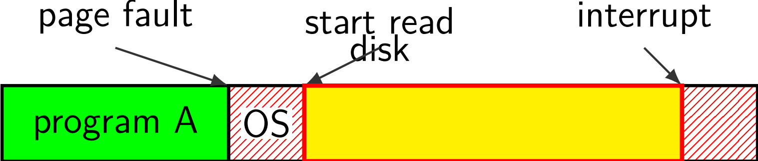
program B pages

loaded

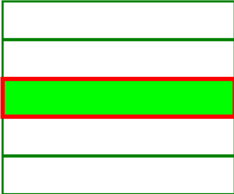
evicted (to free space)

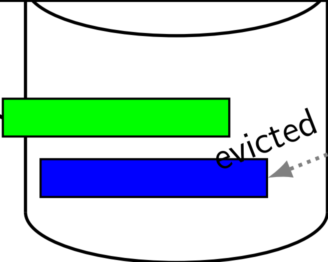
...

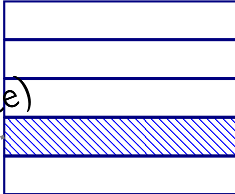
...

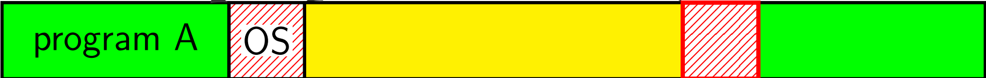


swapping timeline









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 `0x7FFFFFFFF` 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

mmap

Linux/Unix has a function to “map” a file to memory

```
int file = open("somefile.dat", O_RDWR);

    // data is region of memory that represents file
char *data = mmap(..., file, 0);

    // read byte 6 from somefile.dat
char seventh_char = data[6];

   // modifies byte 100 of somefile.dat
data[100] = 'x';
    // can continue to use 'data' like an array
```

swapping almost mmap

access mapped file for first time, read from disk

(like swapping when memory was swapped out)

write "mapped" memory, write to disk eventually

(like writeback policy in swapping)

use "dirty" bit

extra detail: other processes should see changes

all accesses to file use same physical memory

fast copies

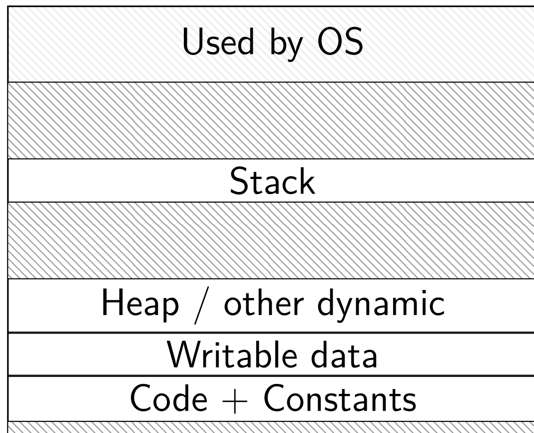
Unix mechanism for starting a new process: `fork()`

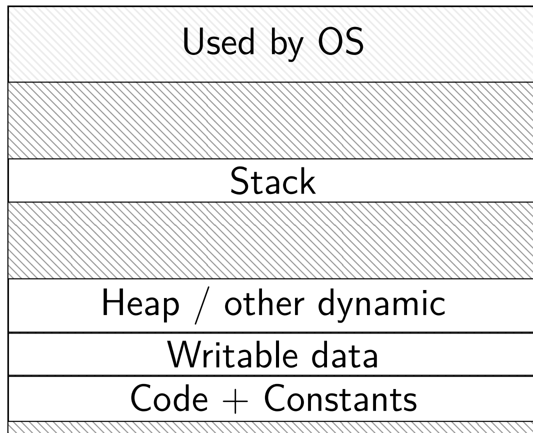
creates a copy of an entire program!

(usually, the copy then calls `execve` — replaces itself with another program)

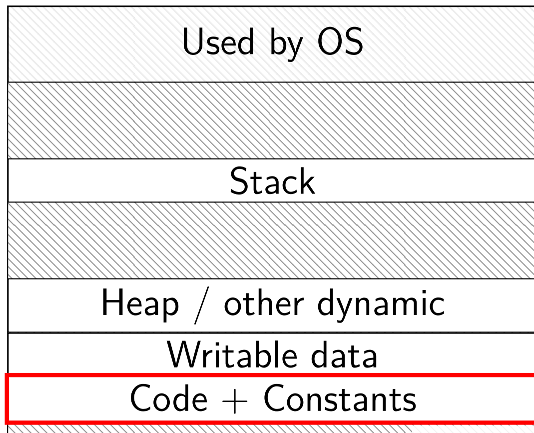
how isn't this really slow?

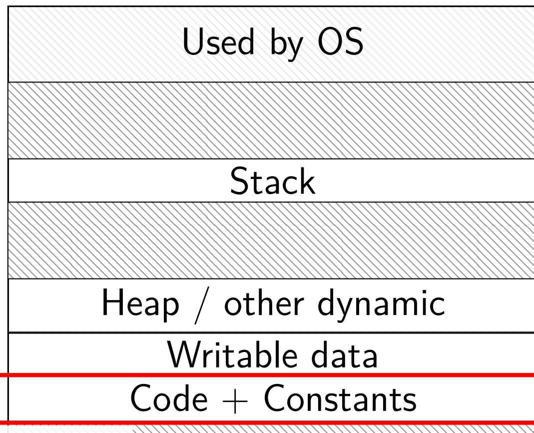
do we really need a complete copy?





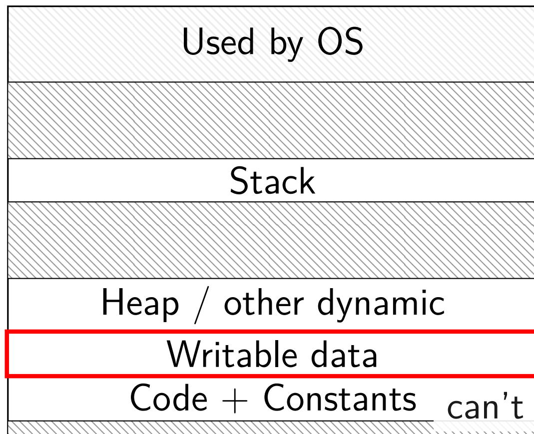
do we really need a complete copy?

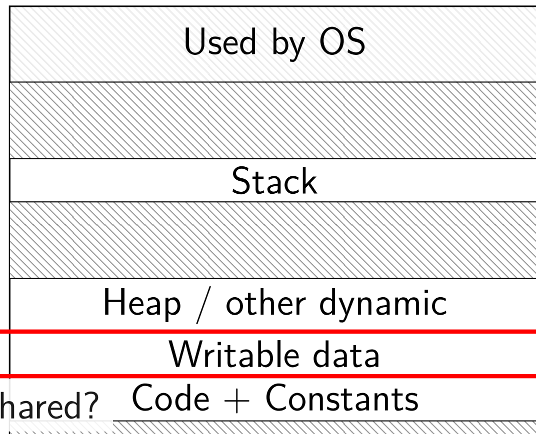




shared as read-only

do we really need a complete copy?





can't be shared?

trick for extra sharing

sharing writeable data is fine — until either process modifies the copy

can we detect modifications?

trick: tell CPU (via page table) shared part is read-only

processor will trigger a fault when it's written

copy-on-write and page tables

VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	1	0x12345
0x00602	1	1	0x12347
0x00603	1	1	0x12340
0x00604	1	1	0x200DF
0x00605	1	1	0x200AF
…	…	…	…

copy-on-write and page tables

VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
…	…	…	…

VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
…	…	…	…

copy operation actually duplicates page table
both processes share all physical pages
but marks pages in both copies as read-only

copy-on-write and page tables

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
...	...	...	...

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
...	...	...	...

when either process tries to write read-only page
triggers a fault — OS actually copies the page

copy-on-write and page tables

VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
…	…	…	…

VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	1	0x300FD
…	…	…	…

after allocating a copy, OS reruns the write instruction

replacement policy

since disks are so slow, replacement policy really matters

will be implemented in software

like with caches: something like least-recently-used usually good

- but exceptions: some access patterns won't work well

LRU replacement?

problem: need to identify when pages are used

- ideally every single time

not practical to do this exactly

- HW would need to keep a list of when each page was accessed, or
- SW would need to force every access to trigger a fault

trick: any page which hasn't been used in a while is probably fine

- not likely to make a difference whether it was last used 120 seconds ago or 300 seconds ago

LRU approximation intuition

one idea: detect accesses by marking page table entry invalid temporarily

- e.g. every N seconds

on page fault:

- if marked as invalid: make valid again

choose page which has stayed invalid for a long time

hardware support for access tracking

often hardware implements *accessed* bit in page table entries

set to 1 when page table entry is used by program

avoids requiring page fault

backup slides