changelog

Changes since first lecture:

30 November 2021: ‘exercise: 64-bit system’: adjust “top 16” to “top 16 of 64”

30 November 2021: two-level page tables: be more careful about calling it page table**s** (plural)

30 November 2021: two-level page table lookup: label page # versus physical address

30 November 2021: search tree tradeoffs: add design decisions made

30 November 2021: 2-level example: add colorized slide at end of example

last time

extra permission bits in page table entries
  - read-only, non-executable pages, etc.
  - checked in same way as valid bit

address space sizes
  - number of possible addresses
  - not all physical addresses correspond to physical memory
  - virtual address size = # bits used for mapping

page tables in memory
  - array of page table entries
  - need to decide how to encode page table entry as number

final logistics

if you can't make the final exam time, fill out the form ASAP

on exceptions and PC saving

for exceptions, we save the PC

should be address of first instruction not executed

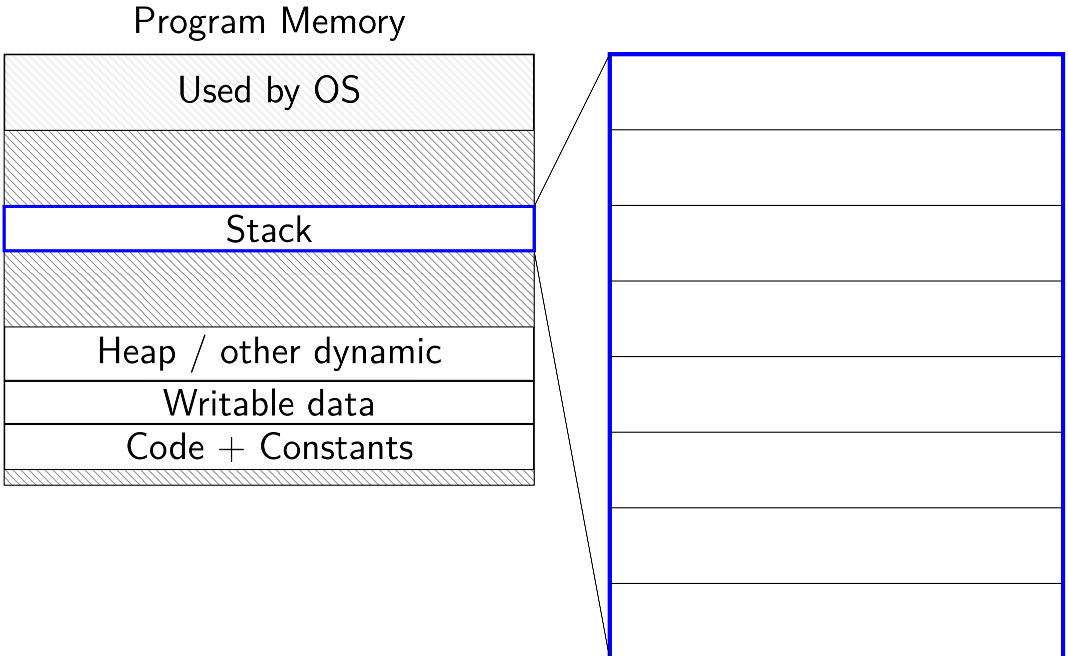
- keypress: instruction not yet run b/c keypress happened
- page fault, divide by zero: instruction that did something bad
- system call: instruction *after* system call

we'll see why this choice in a bit

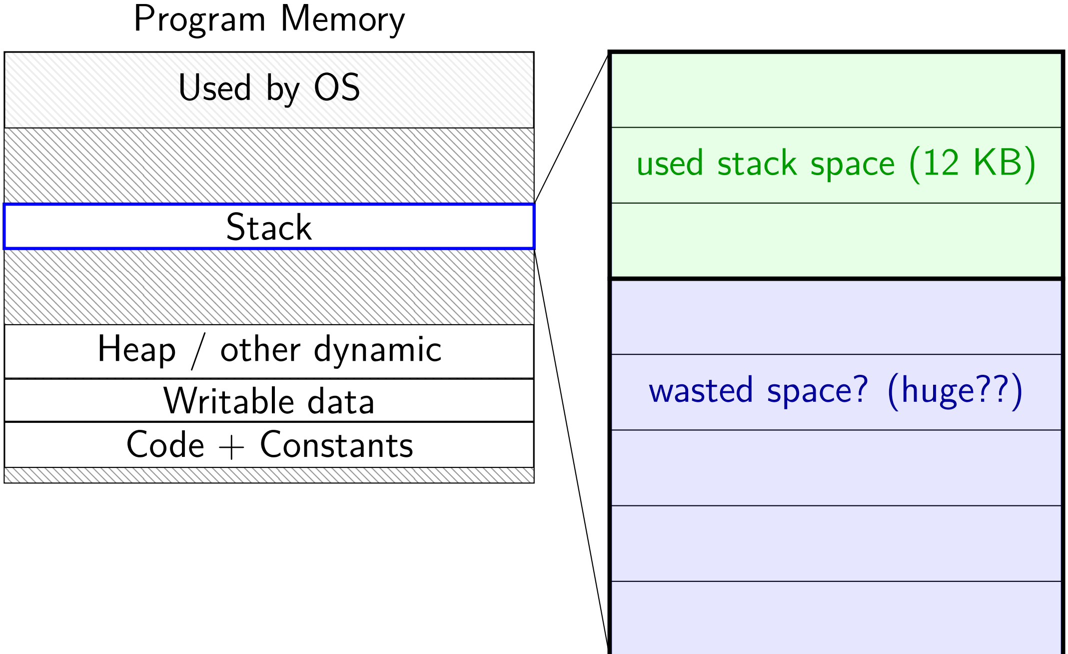
slight inaccuracy in last week's quiz

- assumed always the non-system-call-like case

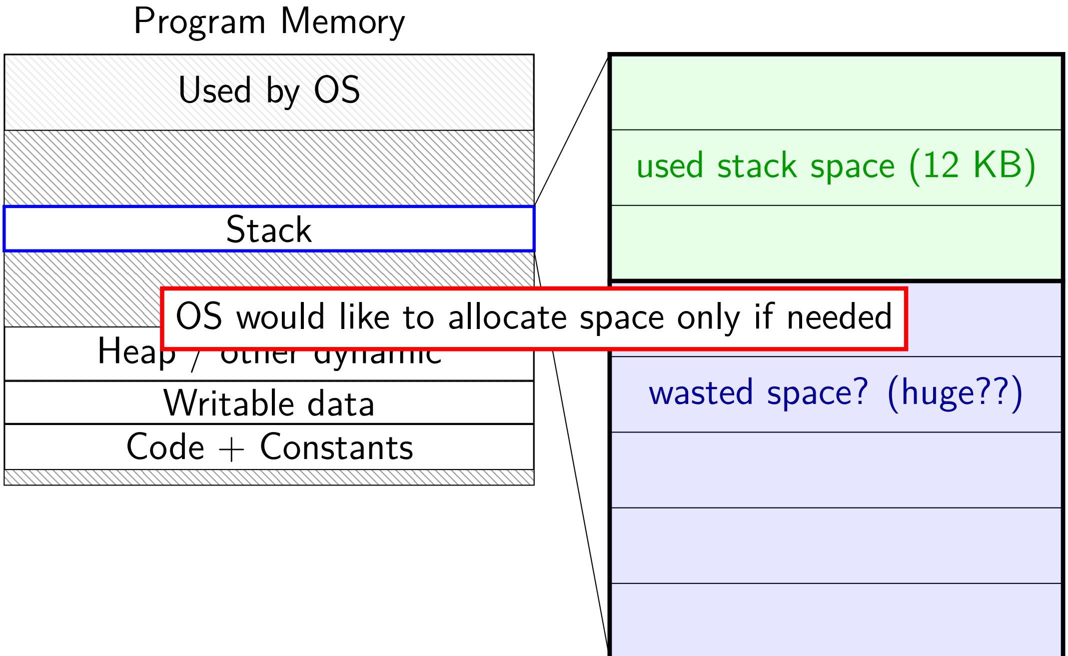
space on demand



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" $\approx$ 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)

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

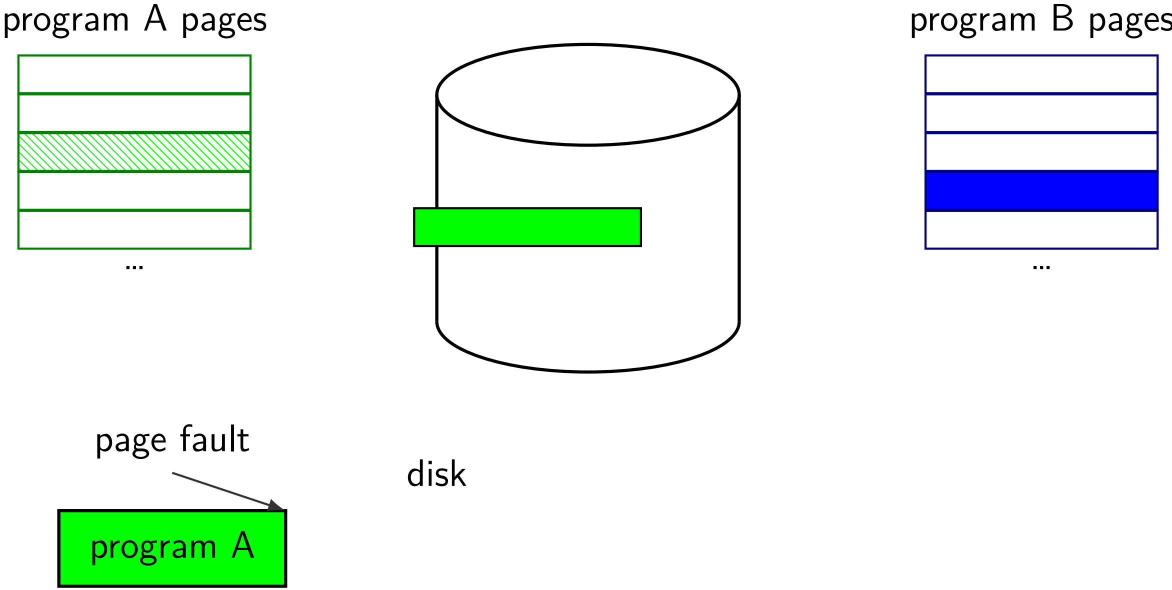
HDD reads and writes: milliseconds to tens of milliseconds

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

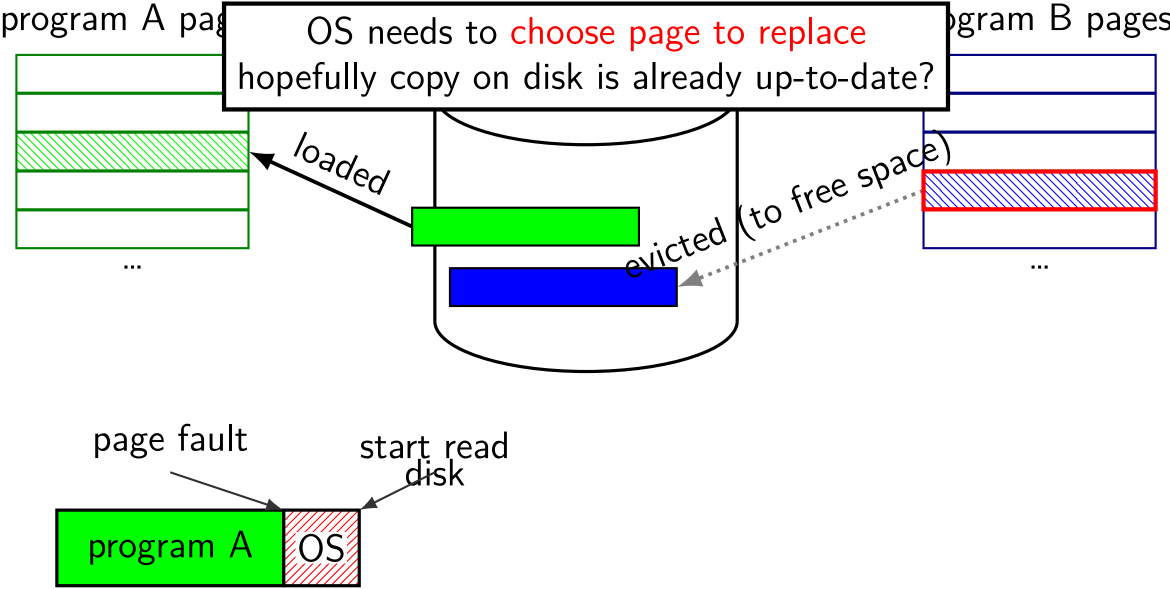
SSD writes and writes: hundreds of microseconds

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

swapping timeline



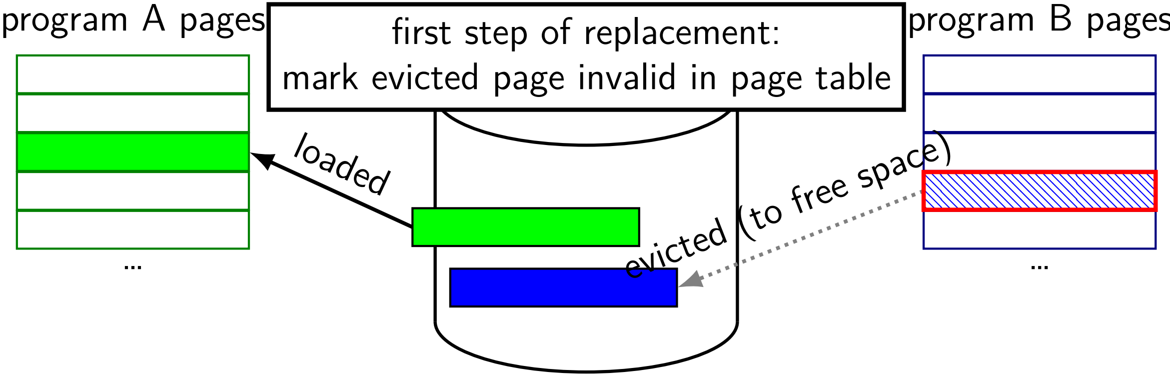
swapping timeline



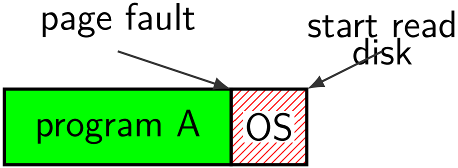
OS needs to choose page to replace

hopefully copy on disk is already up-to-date?

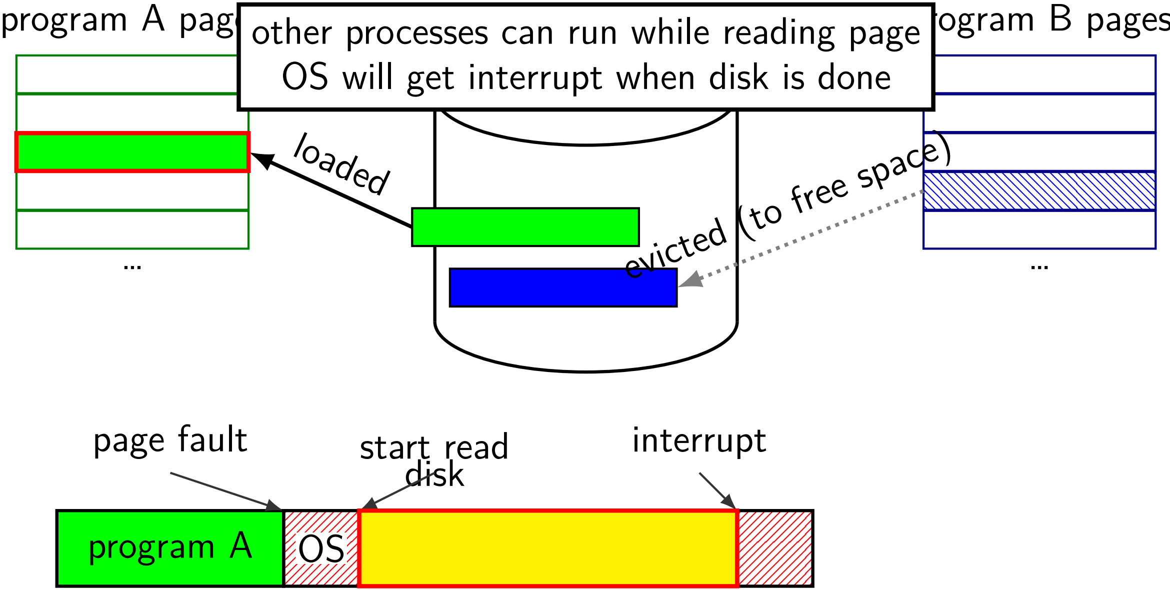
swapping timeline



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

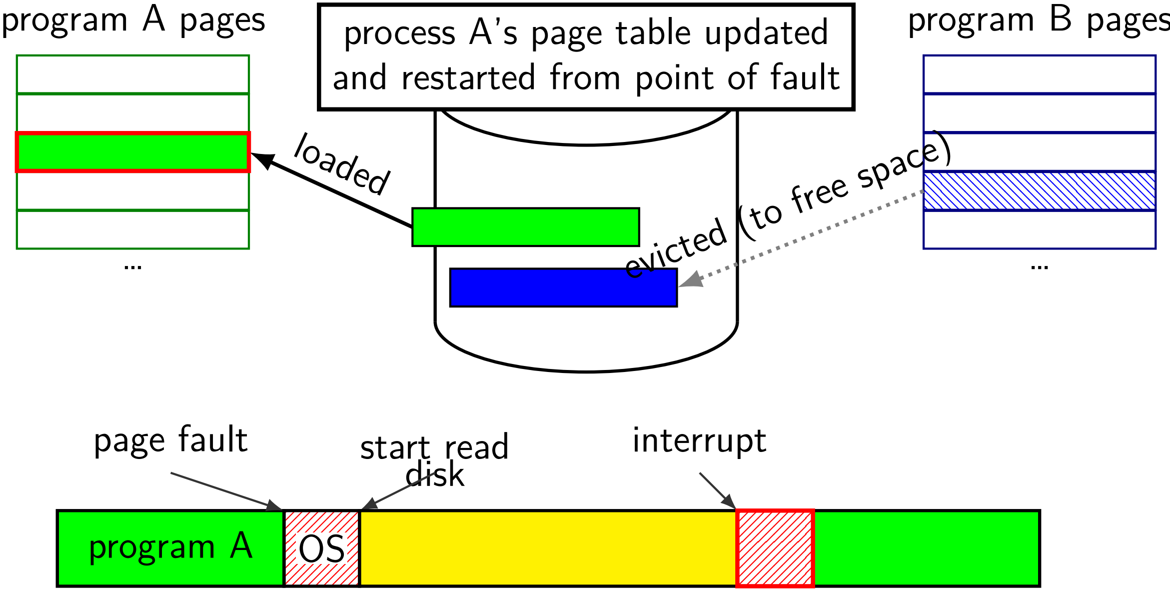


swapping timeline



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

swapping timeline



program A pages

...

process A's page table updated and restarted from point of fault

loaded

evicted (to free space)

program B pages

...

page fault

start read disk

interrupt

program A

OS

page tricks generally

deliberately make program trigger page/protection fault

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

have seperate data structures represent logically allocated memory
  - e.g. "addresses `0x7FFF8000` to `0x7FFFFFFF` are the stack"
  - might talk about Linux data structures later (book section 9.7)

page table is for the hardware and not the OS

hardware help for page table tricks

information about the address causing the fault

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

(by default) rerun faulting instruction when returning from exception

precise exceptions: no side effects from faulting instruction or after

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

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??)

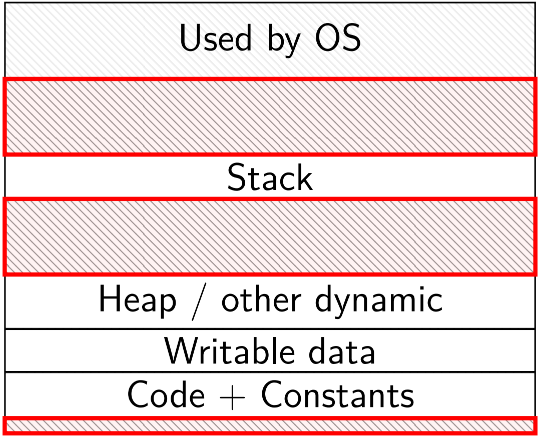
huge page tables

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

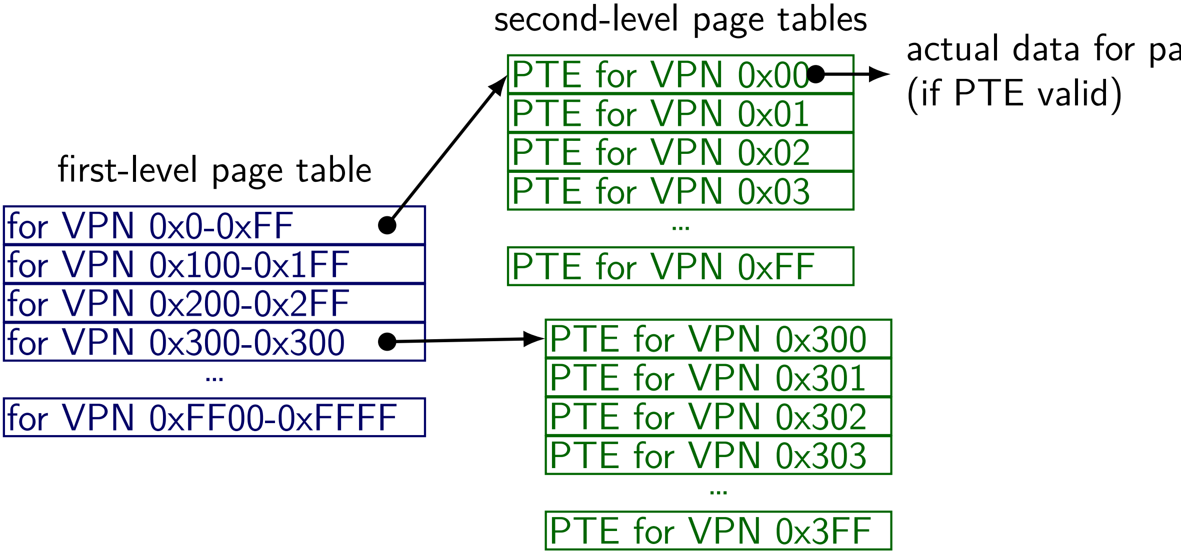
- lookup should be simple
- solution: lookup splits up address bits (no complex calculations)

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)



second-level page tables

actual data for pa
(if PTE valid)

PTE for VPN 0x00
PTE for VPN 0x01
PTE for VPN 0x02
PTE for VPN 0x03
...
PTE for VPN 0xFF

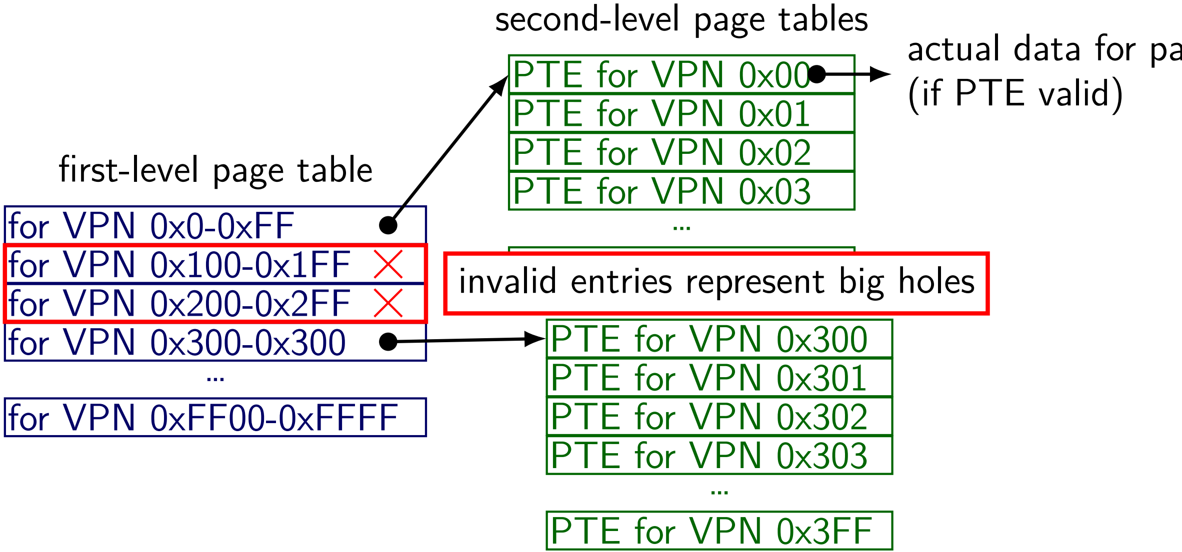
first-level page table

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

PTE for VPN 0x300
PTE for VPN 0x301
PTE for VPN 0x302
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)



second-level page tables

actual data for pa
(if PTE valid)

PTE for VPN 0x00
PTE for VPN 0x01
PTE for VPN 0x02
PTE for VPN 0x03
...

first-level page table

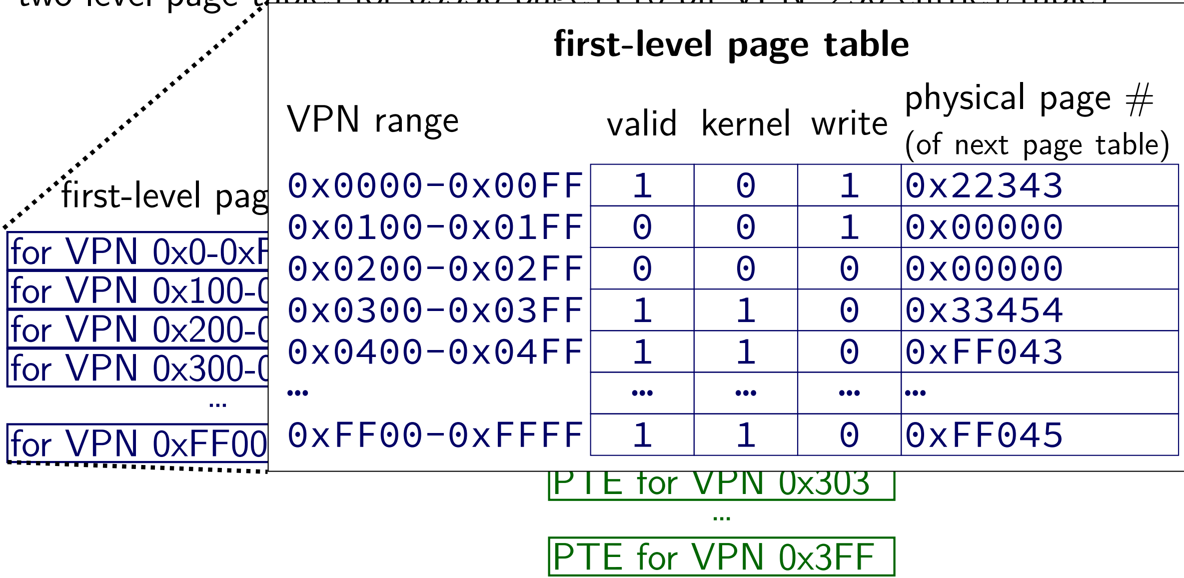
for VPN 0x0-0xFF
for VPN 0x100-0x1FF ✗
for VPN 0x200-0x2FF ✗
for VPN 0x300-0x300
...
for VPN 0xFF00-0xFFFF

invalid entries represent big holes

PTE for VPN 0x300
PTE for VPN 0x301
PTE for VPN 0x302
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

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`

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

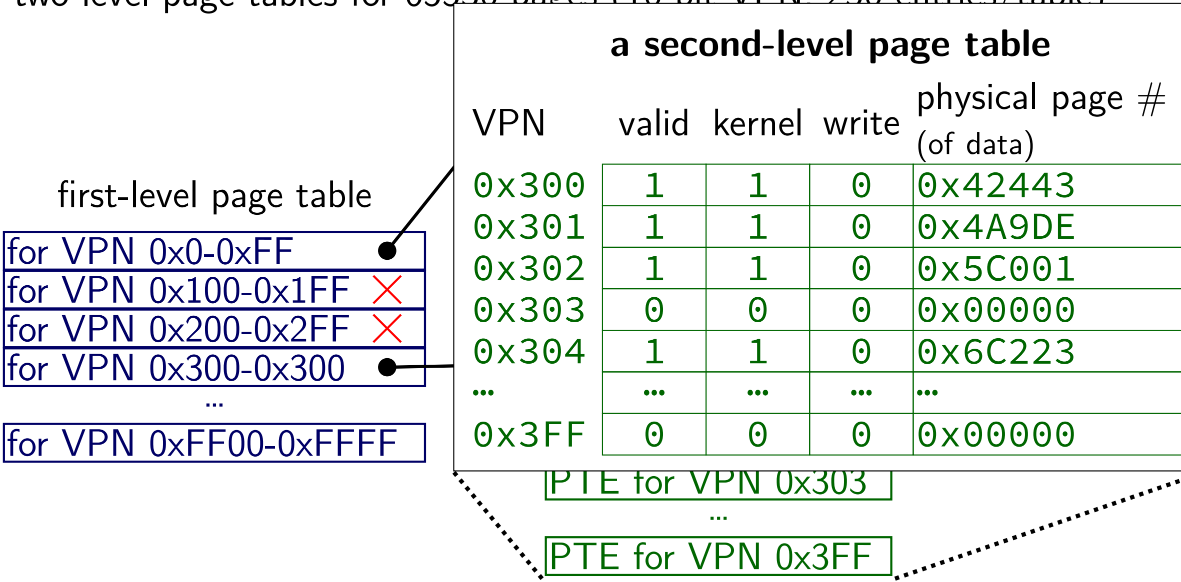
PTE for VPN 0x303

…

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

- 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

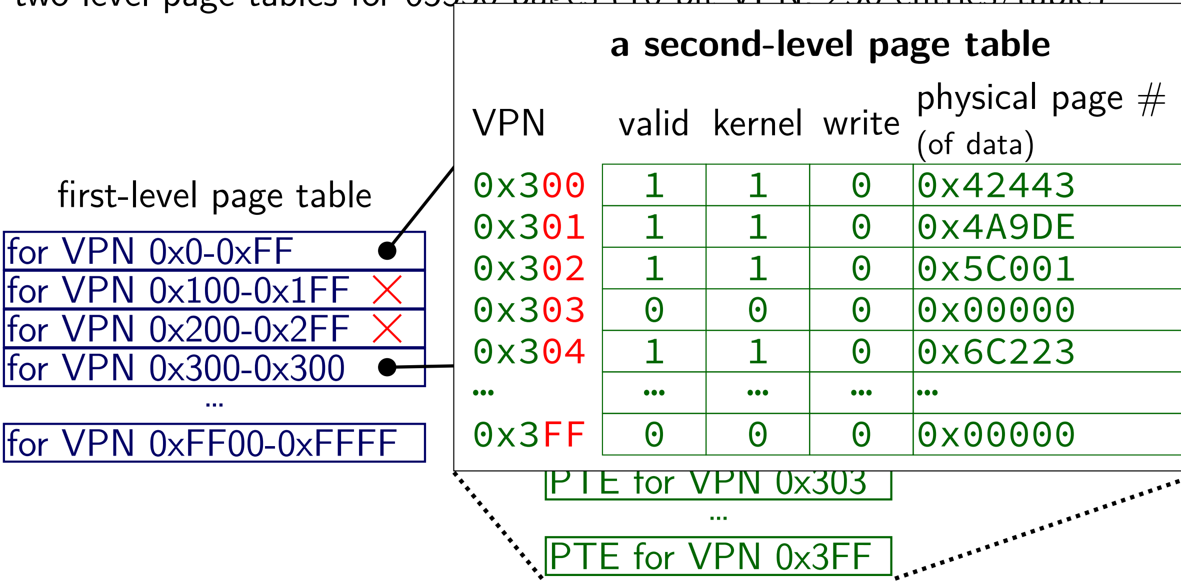
PTE for VPN 0x303

…

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

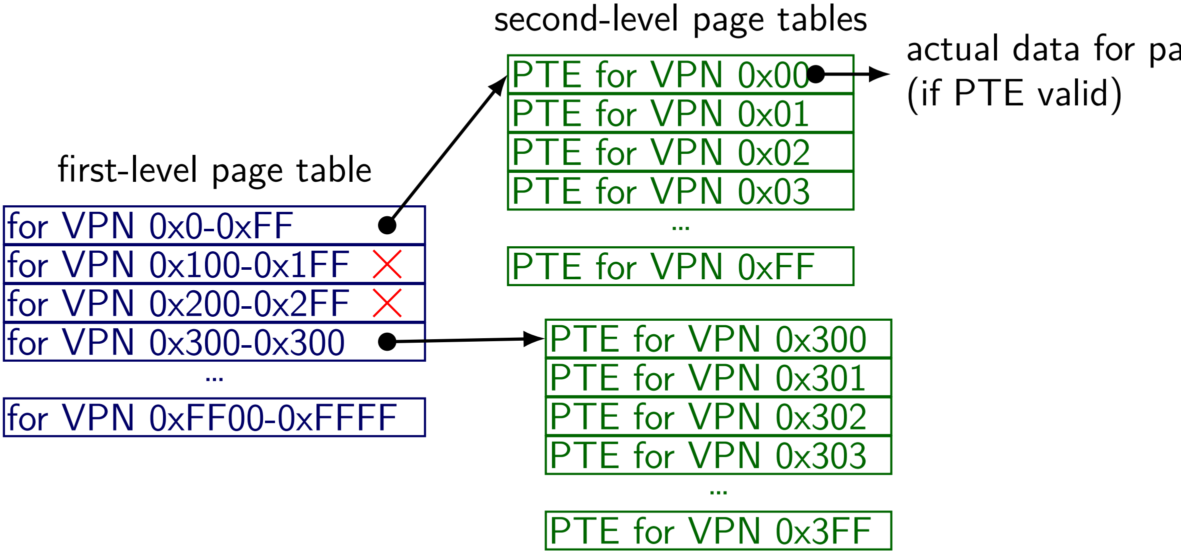
PTE for VPN 0x303

…

PTE for VPN 0x3FF

two-level page tables

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



second-level page tables

actual data for pa
(if PTE valid)

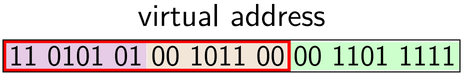
first-level page table

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

PTE for VPN 0x00
PTE for VPN 0x01
PTE for VPN 0x02
PTE for VPN 0x03
...
PTE for VPN 0xFF

PTE for VPN 0x300
PTE for VPN 0x301
PTE for VPN 0x302
PTE for VPN 0x303
...
PTE for VPN 0x3FF

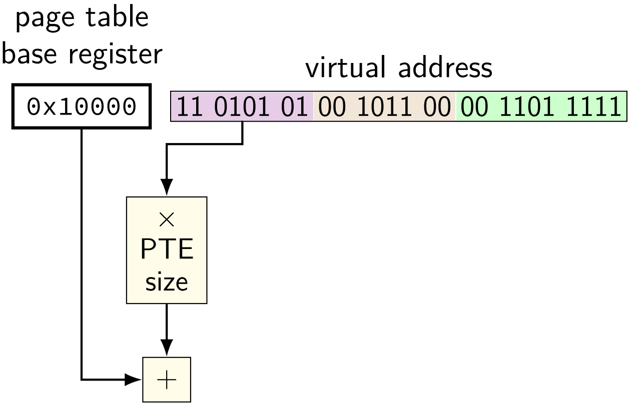
two-level page table lookup



VPN — split into two parts (one per level)

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

two-level page table lookup


page table
base register

virtual address

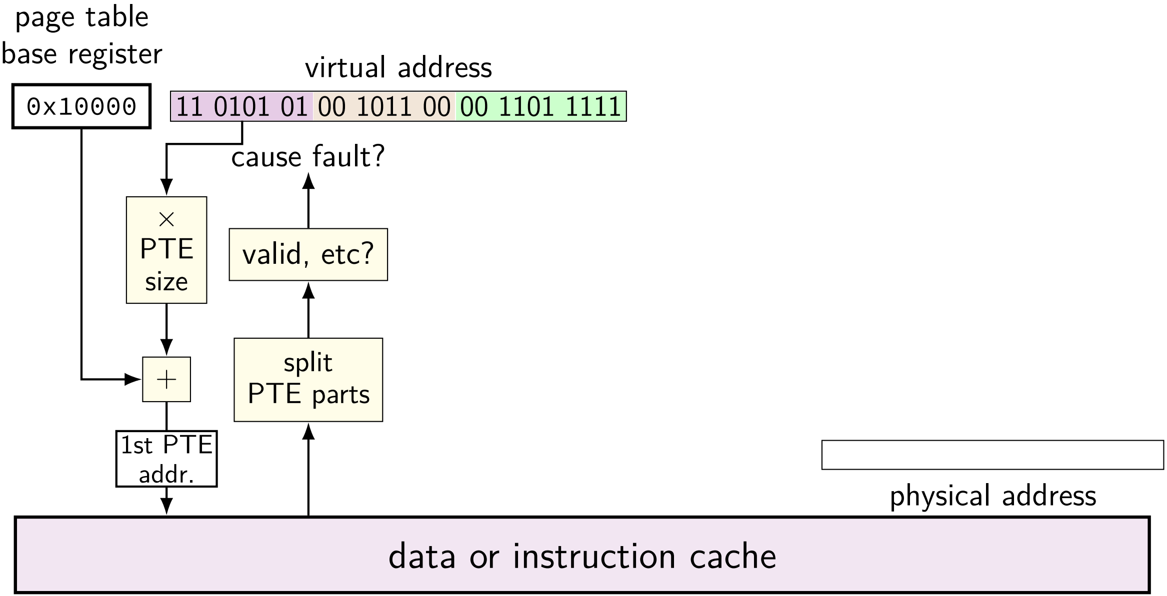
`0x10000`

11 0101 01 00 1011 00 00 1101 1111

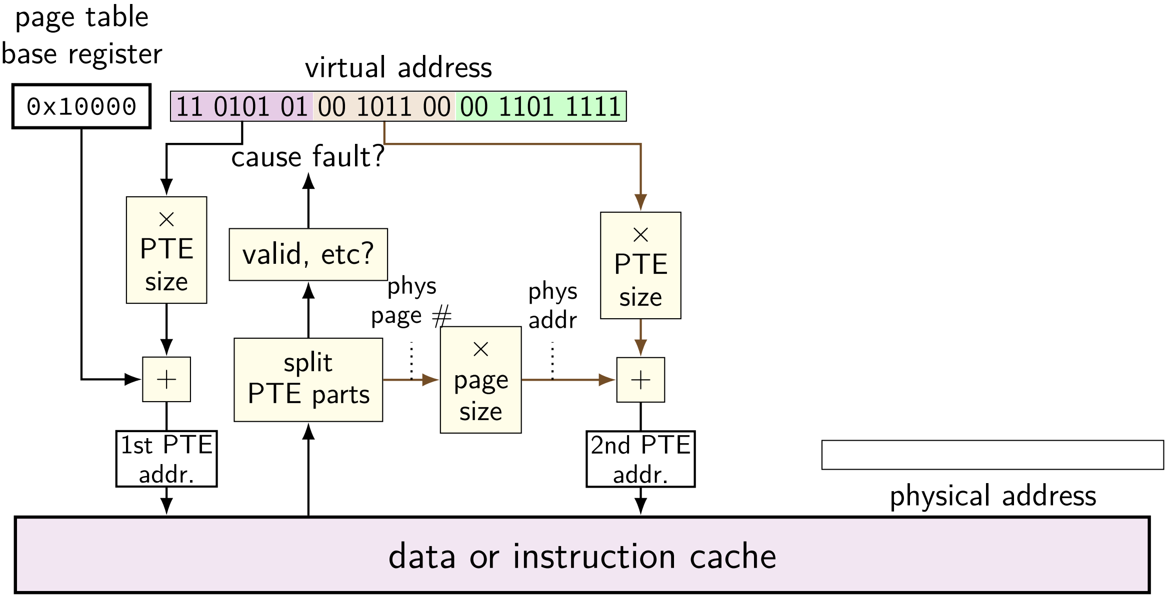
× PTE size

+

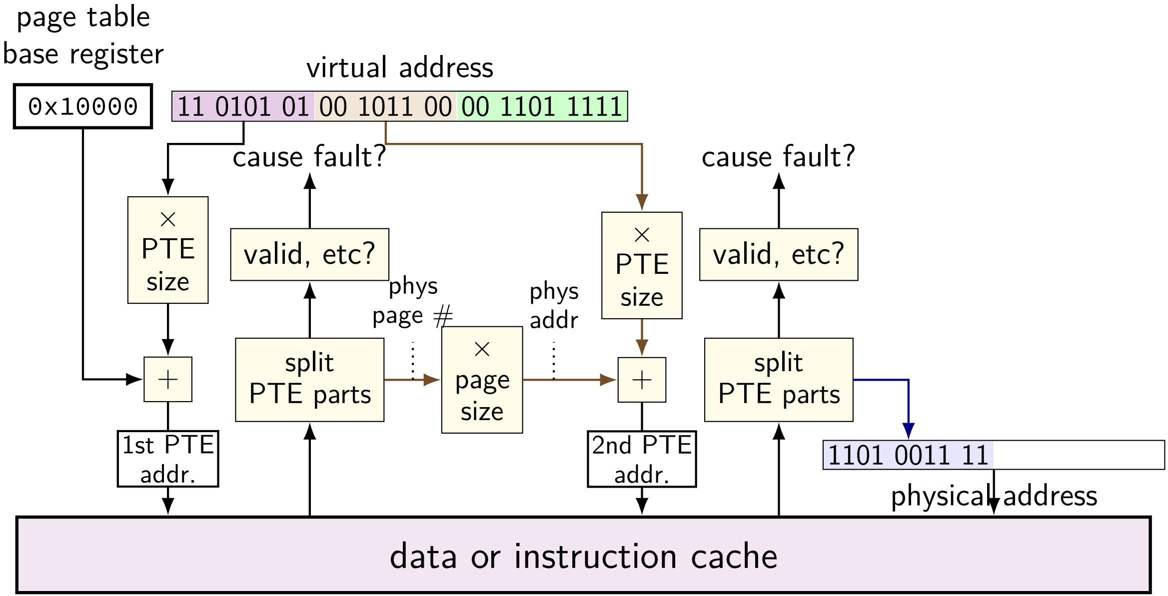
two-level page table lookup



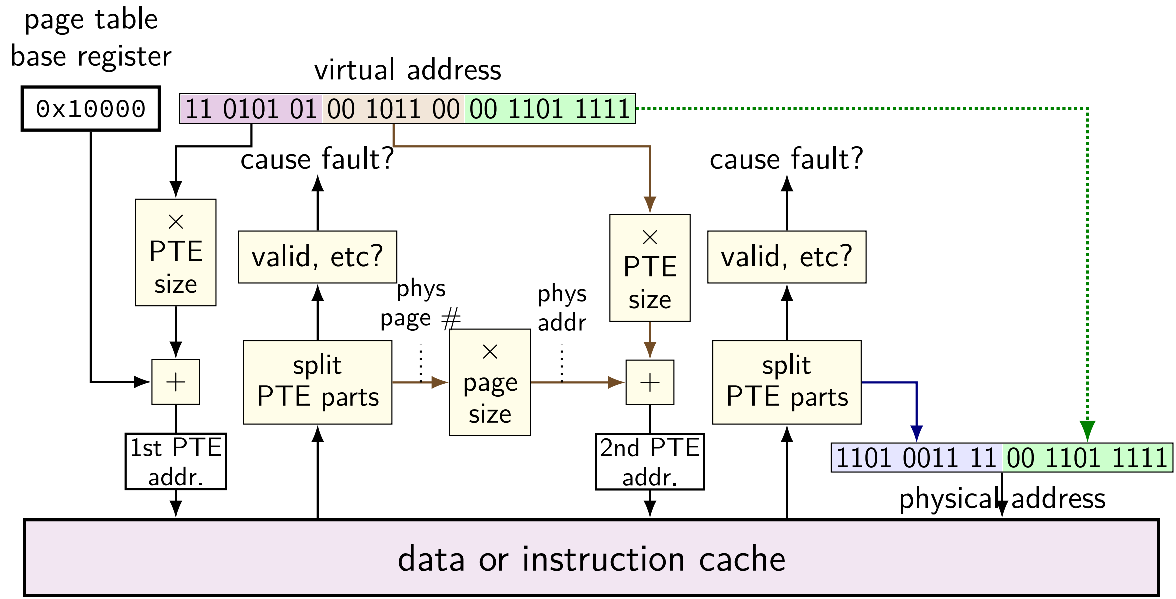
two-level page table lookup



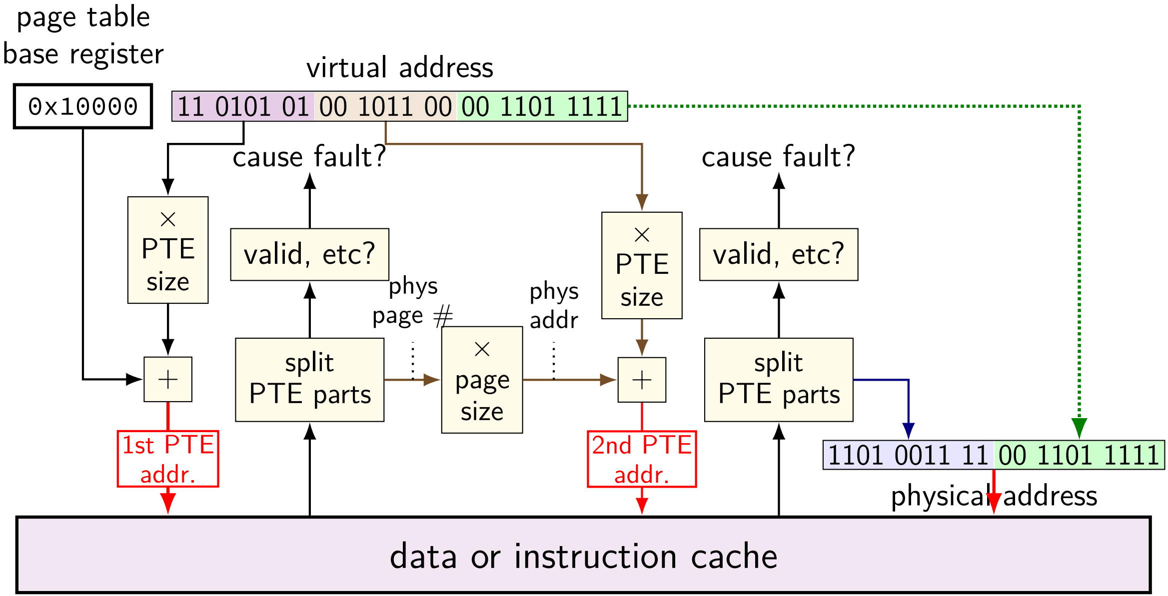
two-level page table lookup



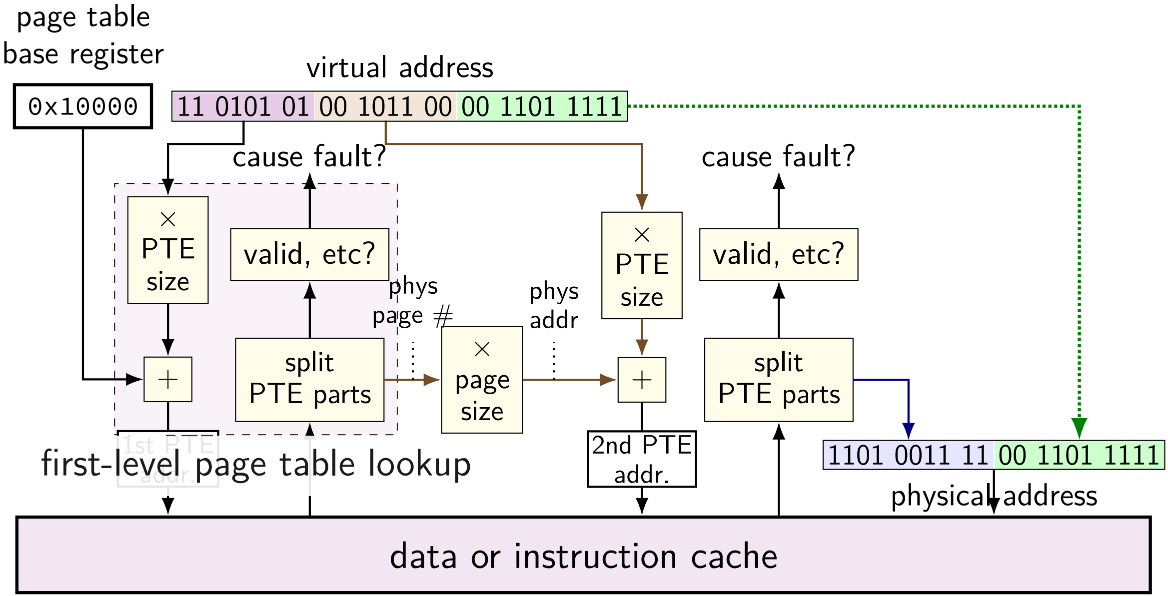
two-level page table lookup



two-level page table lookup



two-level page table lookup



page table
base register

`0x10000`

virtual address

11 0101 01 | 00 1011 00 | 00 1101 1111

cause fault?

× PTE size

valid, etc?

+

split PTE parts

phys page #

× page size

phys addr

× PTE size

+

cause fault?

valid, etc?

split PTE parts

1st PTE addr.

first-level page table lookup

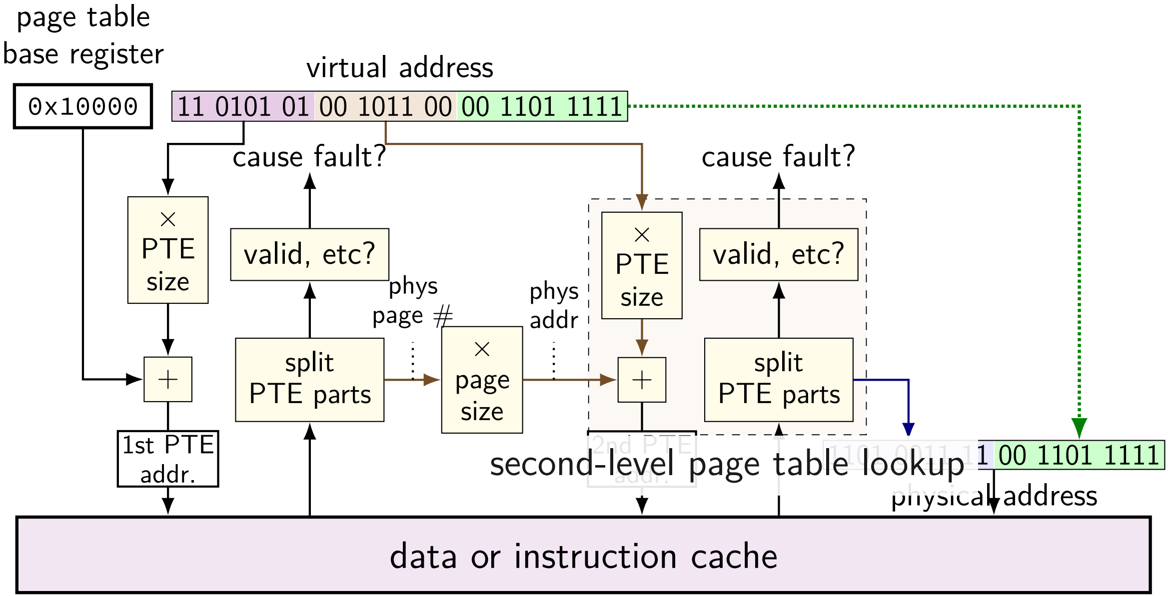
2nd PTE addr.

1101 0011 11 | 00 1101 1111

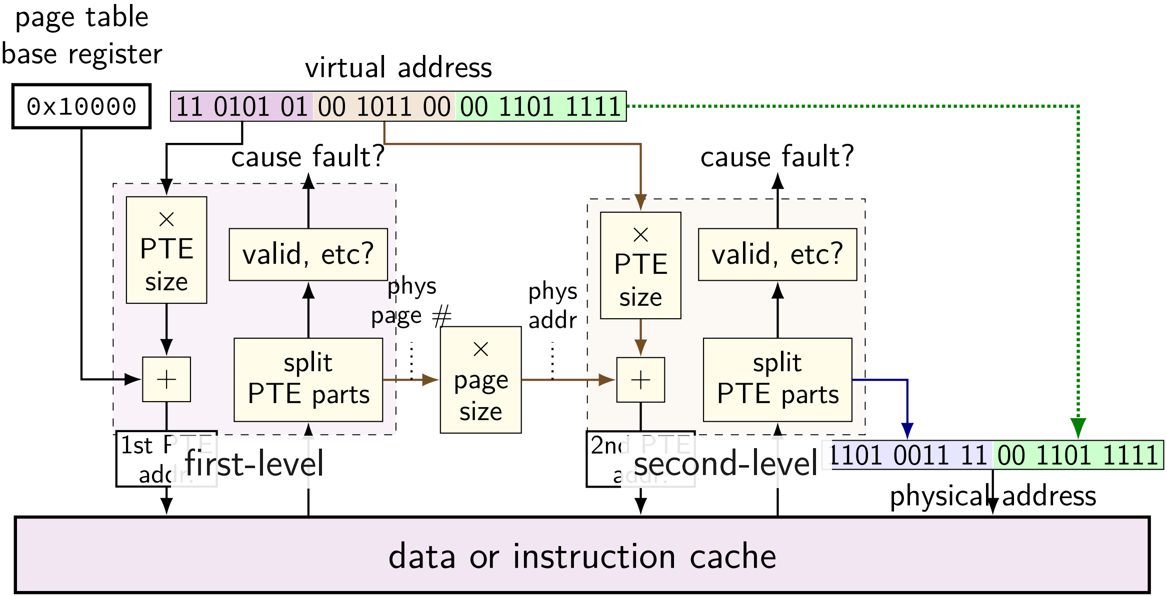
physical address

data or instruction cache

two-level page table lookup



two-level page table lookup



page table base register

`0x10000`

virtual address

11 0101 01 | 00 1011 00 | 00 1101 1111

cause fault?

× PTE size

valid, etc?

+

split PTE parts

phys page #

× page size

phys addr

cause fault?

× PTE size

valid, etc?

+

split PTE parts

1st PTE addr.

first-level

2nd PTE addr.

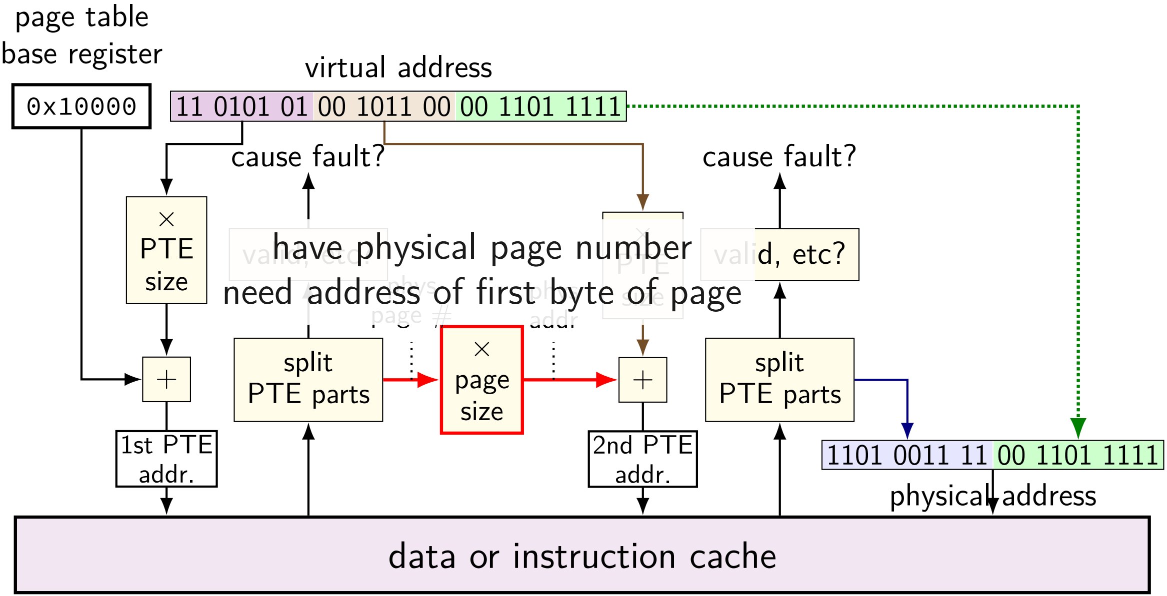
second-level

1101 0011 11 | 00 1101 1111

physical address

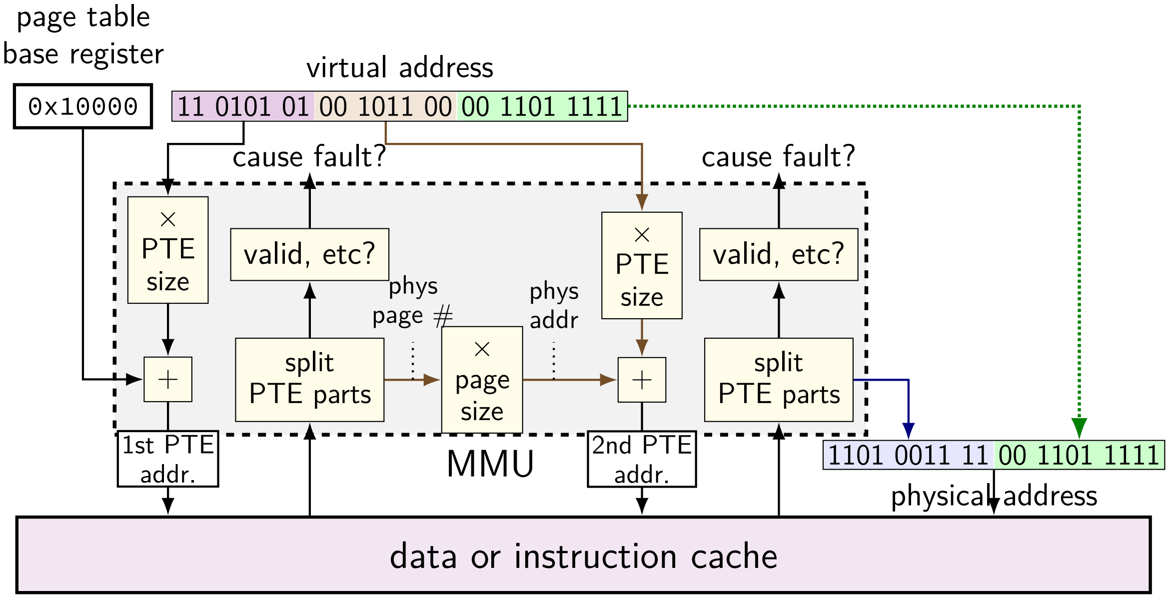
data or instruction cache

two-level page table lookup

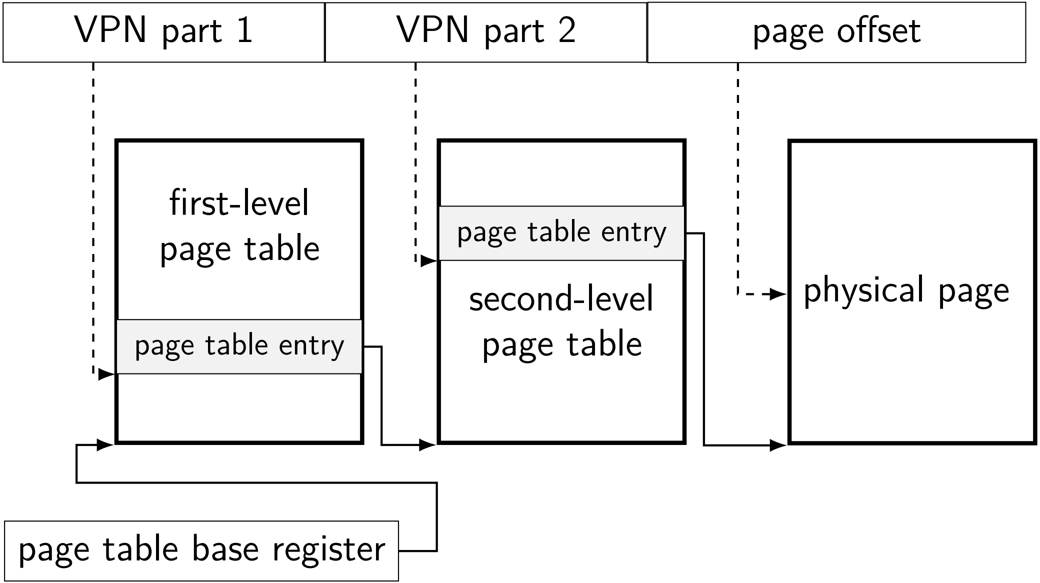


have physical page number
need address of first byte of page

two-level page table lookup



another view



VPN part 1

VPN part 2

page offset

first-level page table

page table entry

second-level page table

page table entry

physical page

page table base register

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)

splitting addresses for levels

x86-32

32-bit physical address; 32-bit virtual address

2^{12} byte page size

2-levels of page tables; each page table is one page

4 byte page table entries

how is address `0x12345678` split up?

splitting addresses for levels

x86-32

32-bit physical address; 32-bit virtual address

2^{12} byte page size

- *12-bit page offset*

2-levels of page tables; each page table is one page

4 byte page table entries

how is address `0x12345678` split up?

splitting addresses for levels

x86-32

32-bit physical address; 32-bit virtual address

2^{12} byte page size

- *12-bit page offset*

2-levels of page tables; each page table is one page

4 byte page table entries

- $2^{12}/4 = 2^{10}$ *PTEs/page table; 10-bit VPN parts*

how is address `0x12345678` split up?

splitting addresses for levels

x86-32

32-bit physical address; 32-bit virtual address

2^{12} byte page size

- *12-bit page offset*

2-levels of page tables; each page table is one page

4 byte page table entries

- $2^{12}/4 = 2^{10}$ *PTEs/page table; 10-bit VPN parts*

how is address `0x12345678` split up?

- 10-bit VPN part 1: `0001 0010 00 (0x48);`
- 10-bit VPN part 2: `11 0100 0101 (0x345);`
- 12-bit page offset: `0x678`

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 ×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:

`110 000` + `110` × `1` = `0x36`

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:

110 000 + 110 × 1 = 0x36

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:

`110 000` + `110` × `1` = `0x36`

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:

110 000 + 110 × 1 = 0x36

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 → 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 → 3-bit VPN parts

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