

Virtual Memory (2)

1

Changelog

Changes made in this version not seen in first lecture:

- 21 November 2017: 1-level example: added final answer of memory value, not just location
- 21 November 2017: two-level example: answer was 0x0A not 0xBA
- 21 November 2017: do we really need a complete copy?: even size of memory regions between copies
- 21 November 2017: swapping components: add "(if modified)" as condition on when we swap out

1

exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

2

exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

top 16 bits of address not used for translation

2

exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: how many page table entries?

exercise: how large are physical page numbers?

2

exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: how many page table entries? $2^{48}/2^{12} = 2^{36}$ entries

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

2

exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: how many page table entries? $2^{48}/2^{12} = 2^{36}$ entries

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

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

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

2

3

exam and conflicts

final exam 7 December 2017

Gilmer 130

fill out conflict form (today, please?) if you can't make it
linked off schedule
also Collab announcement

4

rotate HW and cache blocking

many of you seemed to think you were doing just loop unrolling...
but generally changed the order of memory accesses in the process
basically **cache blocking**

```
// unroll + cache blocking
for (int i = ...; i += 2)
  for (int j = ...) {
    f(i + 0, j);
    f(i + 1, j);
  }

// unroll loop in i only
// (probably not very helpful)
for (int i = ...; i += 2) {
  for (int j = ...)
    f(i, j);
  for (int j = ...)
    f(i + 1, j);
}
```

5

page table tricks

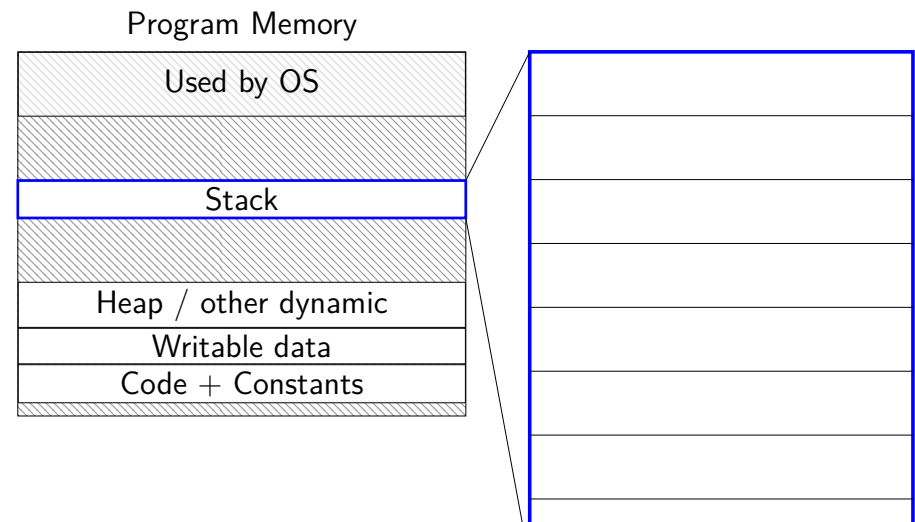
page tables let operating systems do 'magic' with memory

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

instead: change page tables and rerun memory access
similar idea to trap-and-emulate

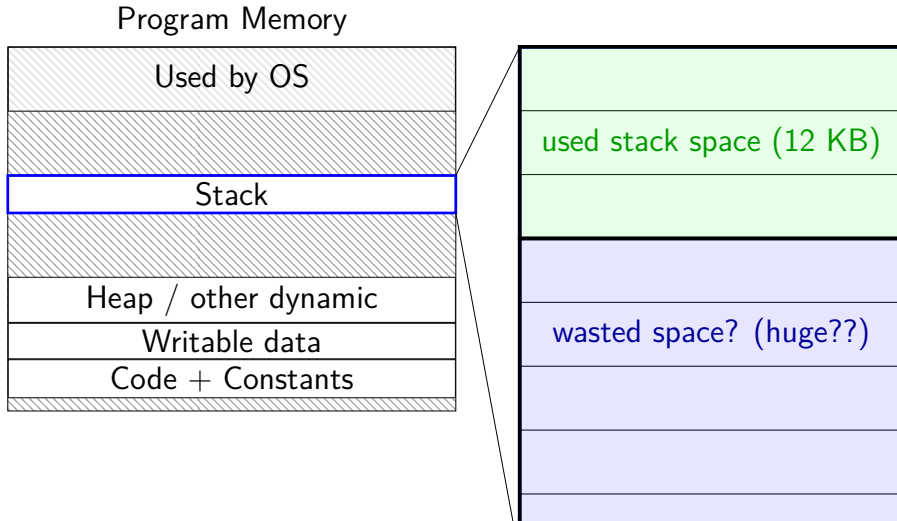
6

space on demand



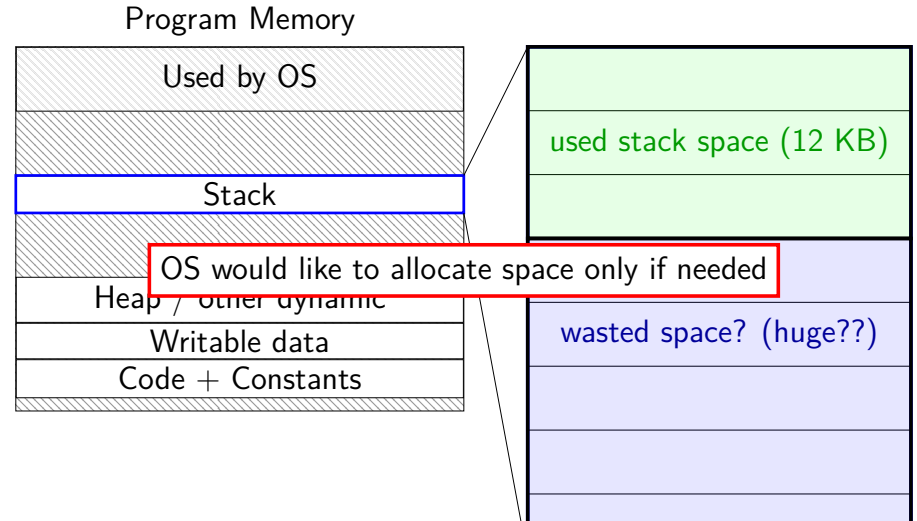
7

space on demand



7

space on demand



7

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
...

8

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"

8

allocating space on demand

`%rsp = 0x7FFFC000`

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

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

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

8

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

9

page tricks generally

deliberately make program trigger page/protection fault

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

have separate 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

10

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

11

swapping

our textbook presents virtual memory to support **swapping**
using disk (or SSD, ...) as the next level of the memory hierarchy

OS allocates space on disk

DRAM is a cache for disk

12

swapping versus caching

“cache block” \approx physical page

fully associative

every virtual page can be stored in any physical page

replacement is managed by the OS

normal cache hits happen without OS

common case that needs to be fast

13

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)

14

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 reads and writes: hundreds of microseconds

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

15

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 reads: **hundreds of microseconds**

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

15

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

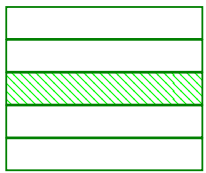
SSD writes and reads: hundreds of microseconds

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

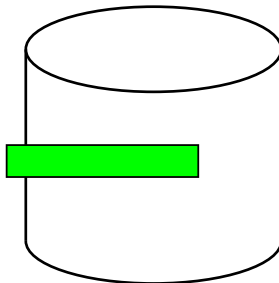
15

swapping timeline

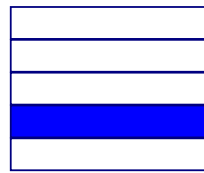
program A pages



...

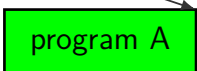


program B pages



...

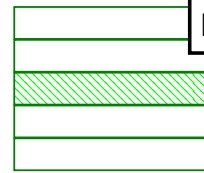
page fault



16

swapping timeline

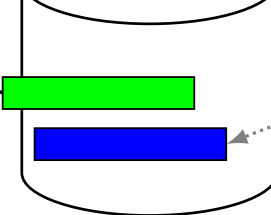
program A pages



...

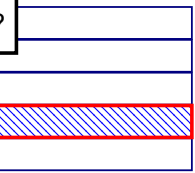
OS needs to **choose page to replace**
hopefully copy on disk is already up-to-date?

loaded



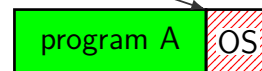
evicted

program B pages



...

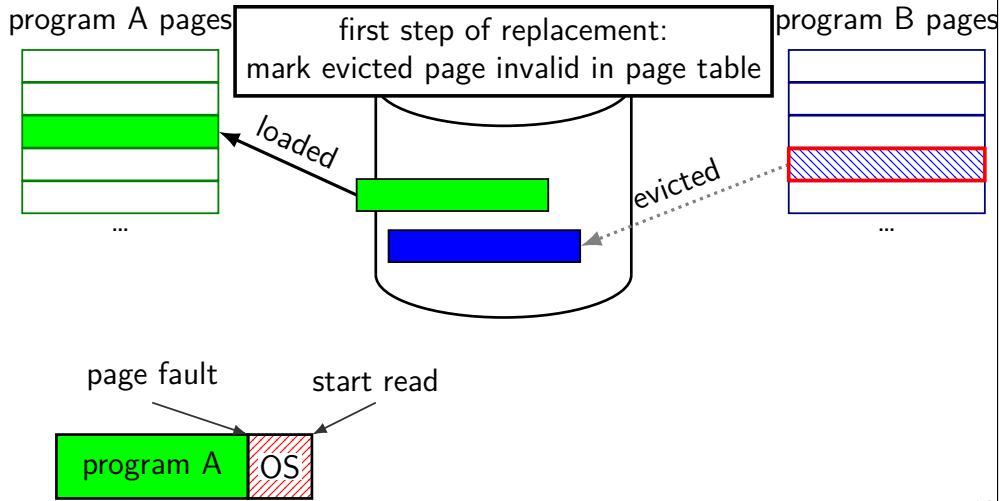
page fault



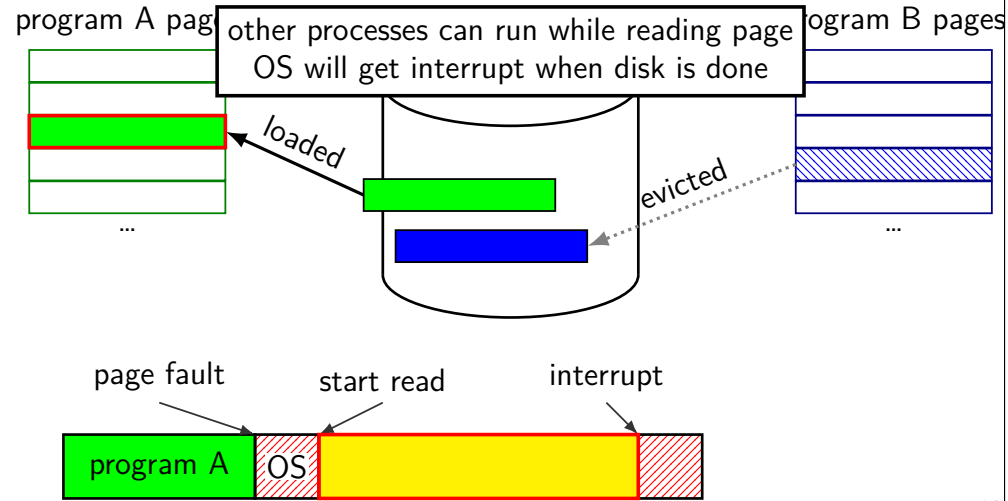
start read

16

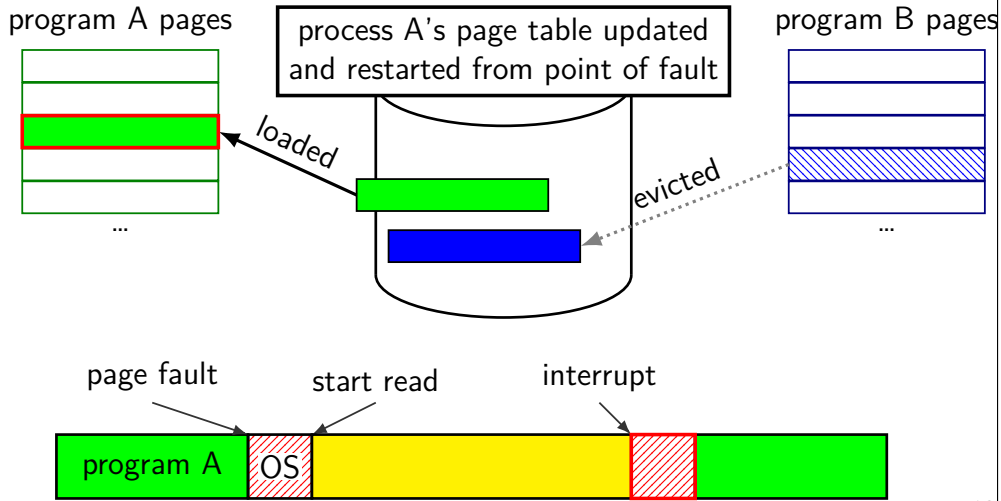
swapping timeline



swapping timeline



swapping timeline



swapping decisions

- write policy
- replacement policy

swapping decisions

write policy

replacement policy

18

swapping is writeback

implementing write-through is hard

when fault happens — physical page not written

when OS resumes process — no chance to forward write

HW itself doesn't know how to write to disk

write-through would also be really slow

HDD/SSD perform best if one writes **at least a whole page** at a time

19

implementing writeback

need a *dirty bit* per page (“was page modified”)

often **kept in the page table!**

option 1 (most common): **hardware sets dirty bit** in page table entry (on write)

bit means “physical page was modified using this PTE”

option 2: OS sets page read-only, flips read-only+dirty bit on fault

20

swapping decisions

write policy

replacement policy

21

replacement policies really matter

huge cost for “miss” on swapping (milliseconds!)
many millions of computations on modern processor
much much worse than even L3 caches

replacement policy implemented **in software**
a lot more room for fancy policies

22

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

23

approximating LRU

one policy: “not recently used”

OS periodically marks all pages as unreadable/writeable

when page fault happens:

- make page accessible again
- put on list of “used” pages

OS replaces pages not on “used” list

- tiebreaker: when was page last on used list?

24

hardware help for not-recently-used

hardware usually implements **accessed** bit in page table entry
whenever page is read/written — hardware sets this bit (if not set)

makes it easier (faster?) for OS to maintain “used ” list

OS can periodically scan/clear “accessed” bit
instead of marking pages invalid temporarily

construct lists of “used” pages

25

accessed/dirty bits

information about how a page table entry was used
indirectly about underlying physical page

kept in **page table entries** themselves

26

accessed/dirty bits

information about how a page table entry was used
indirectly about underlying physical page

kept in **page table entries** themselves

multiple page table entries refer to same page
separate valid/accessed bits for each
OS needs to consolidate them all

26

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
```

27

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**

28

fast copies

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

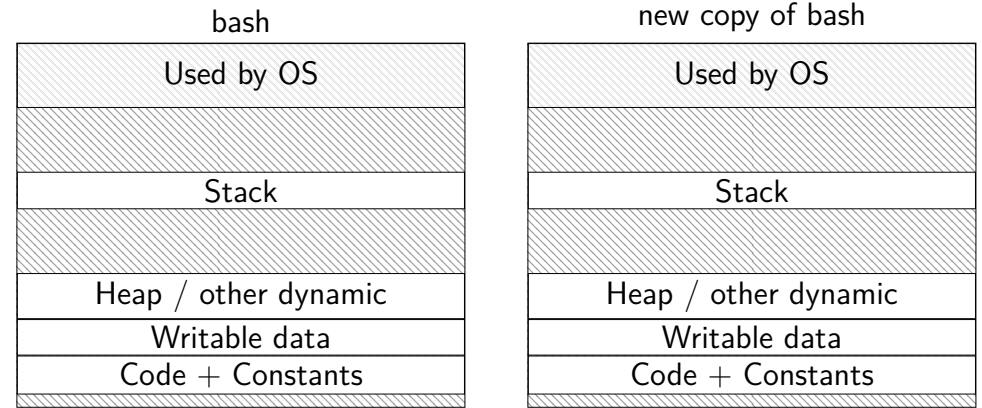
creates a **copy** of an entire program!

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

how isn't this really slow?

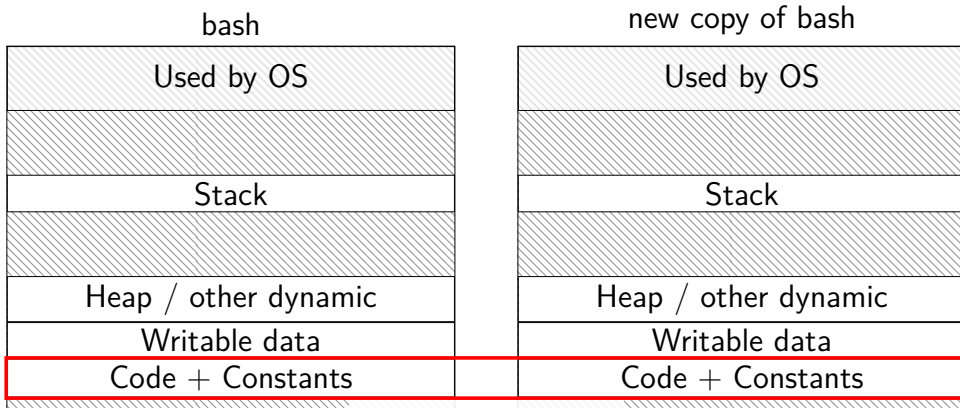
29

do we really need a complete copy?



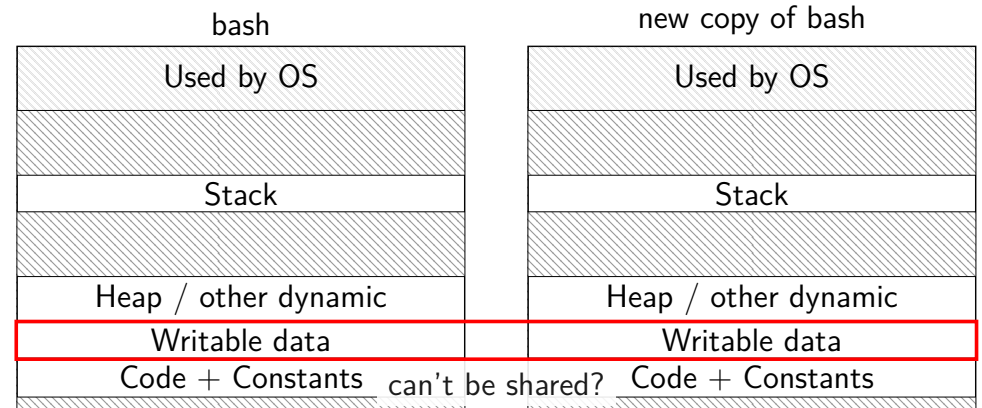
30

do we really need a complete copy?



30

do we really need a complete copy?



30

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

31

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
...

32

copy-on-write and page tables

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

32

copy-on-write and page tables

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

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

32

copy-on-write and page tables

VPN	valid? write?		physical page	VPN	valid? write?		physical page
...
0x00601	1	0	0x12345	0x00601	1	0	0x12345
0x00602	1	0	0x12347	0x00602	1	0	0x12347
0x00603	1	0	0x12340	0x00603	1	0	0x12340
0x00604	1	0	0x200DF	0x00604	1	0	0x200DF
0x00605	1	0	0x200AF	0x00605	1	1	0x300FD
...

after allocating a copy, OS reruns the write instruction

32

exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: how many page table entries? $2^{48}/2^{12} = 2^{36}$ entries

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

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

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

33

huge page tables

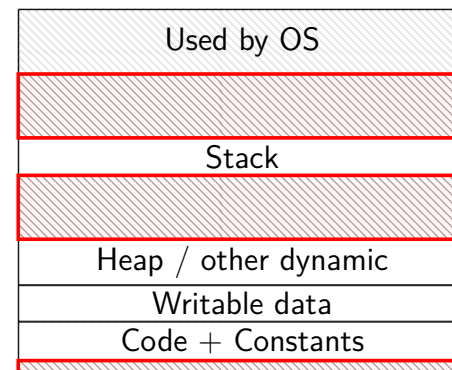
huge virtual address spaces!

impossible to store PTE for every page

how can we save space?

34

holes



most pages are **invalid**

35

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?

36

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

36

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

36

search tree tradeoffs

lookup usually implemented in hardware

lookup should be simple

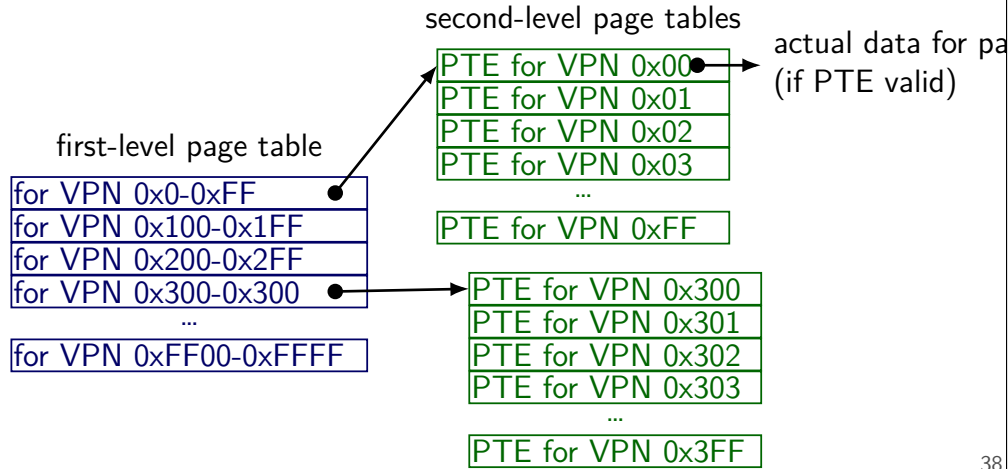
lookup should not involve many memory accesses

doing two memory accesses is already very slow

37

two-level page tables

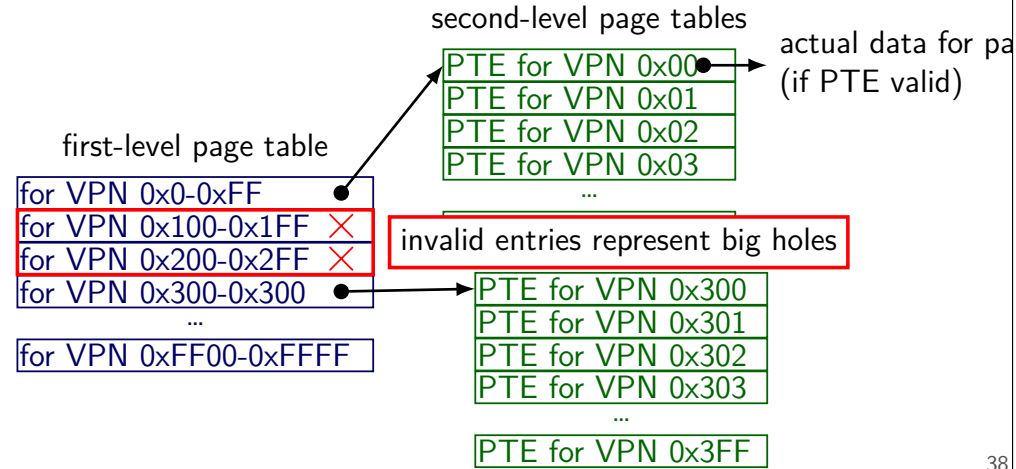
two-level page table for 65536 pages (16-bit VPN)



38

two-level page tables

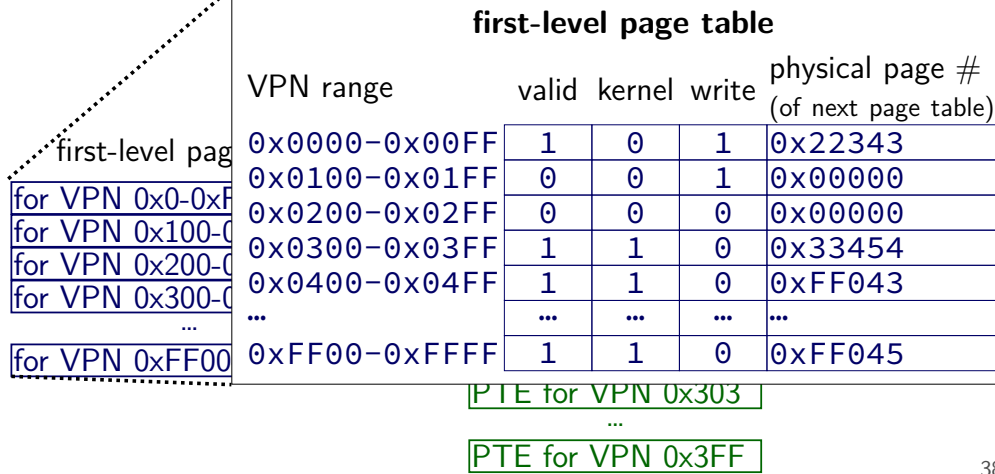
two-level page table for 65536 pages (16-bit VPN)



38

two-level page tables

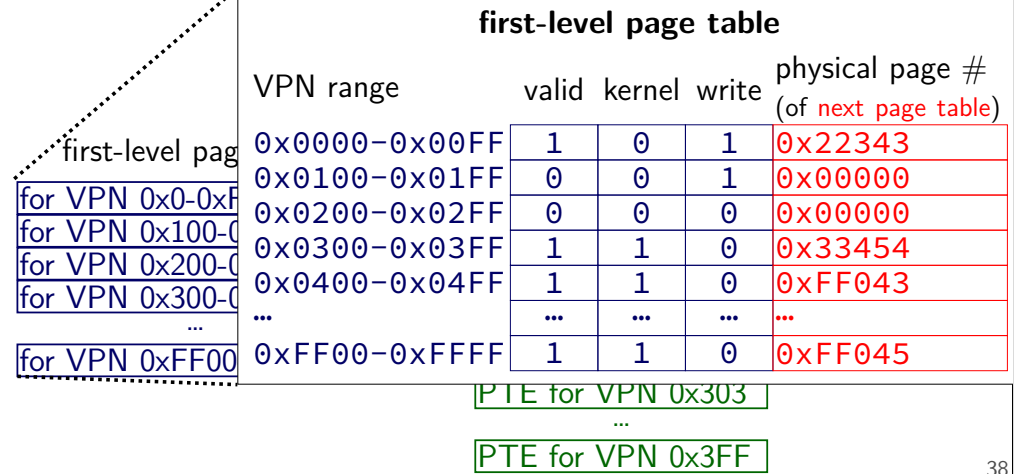
two-level page table for 65536 pages (16-bit VPN)



38

two-level page tables

two-level page table for 65536 pages (16-bit VPN)



38

two-level page tables

two-level page table for 65536 pages (16-bit VPN)

		first-level page table				
		VPN range	valid	kernel write	physical page # (of next page table)	
first-level page table for VPN 0x0-0xFF	for VPN 0x0-0xFF	0x0000-0x00FF	1	0	1	0x22343
	for VPN 0x100-0x1FF	0x0100-0x01FF	0	0	1	0x00000
	for VPN 0x200-0x2FF	0x0200-0x02FF	0	0	0	0x00000
	for VPN 0x300-0x3FF	0x0300-0x03FF	1	1	0	0x33454
	for VPN 0x400-0x4FF	0x0400-0x04FF	1	1	0	0xFF043

for VPN 0xFF00-0xFFFF	0xFF00-0xFFFF	1	1	0	0xFF045	

PTE for VPN 0x303
 ...
PTE for VPN 0x3FF

38

two-level page tables

two-level page table for 65536 pages (16-bit VPN)

		a second-level page table				
		VPN	valid	kernel write	physical page # (of data)	
first-level page table for VPN 0x0-0xFF	for VPN 0x0-0xFF	0x300	1	1	0	0x42443
	for VPN 0x100-0x1FF	0x301	1	1	0	0x4A9DE
	for VPN 0x200-0x2FF	0x302	1	1	0	0x5C001
	for VPN 0x300-0x300	0x303	0	0	0	0x00000
	for VPN 0x300-0x300	0x304	1	1	0	0x6C223

for VPN 0xFF00-0xFFFF	0x3FF	0	0	0	0x00000	

PTE for VPN 0x303
 ...
PTE for VPN 0x3FF

38

two-level page tables

two-level page table for 65536 pages (16-bit VPN)

		a second-level page table				
		VPN	valid	kernel write	physical page # (of data)	
first-level page table for VPN 0x0-0xFF	for VPN 0x0-0xFF	0x300	1	1	0	0x42443
	for VPN 0x100-0x1FF	0x301	1	1	0	0x4A9DE
	for VPN 0x200-0x2FF	0x302	1	1	0	0x5C001
	for VPN 0x300-0x300	0x303	0	0	0	0x00000
	for VPN 0x300-0x300	0x304	1	1	0	0x6C223

for VPN 0xFF00-0xFFFF	0x3FF	0	0	0	0x00000	

PTE for VPN 0x303
 ...
PTE for VPN 0x3FF

38

two-level page tables

two-level page table for 65536 pages (16-bit VPN)

		second-level page tables				
		VPN	valid	kernel write	physical page # (of data)	
first-level page table for VPN 0x0-0xFF	for VPN 0x0-0xFF	0x00	1	1	0	...
	for VPN 0x100-0x1FF	0x01	1	1	0	...
	for VPN 0x200-0x2FF	0x02	1	1	0	...
	for VPN 0x300-0x300	0x03	0	0	0	...
	for VPN 0x300-0x300
	for VPN 0xFF00-0xFFFF	0xFF	0	0	0	...

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

actual data for page (if PTE valid)

38

two-level page table lookup

virtual address

11 0101 01 00 1011 00 | 00 1101 1111

VPN — split into two parts (one per level)

39

two-level page table lookup

page table
base register

0x10000

virtual address

11 0101 01 00 1011 00 00 1101 1111

×
PTE
size

+

39

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

+

1st PTE
addr.

valid, etc?

split
PTE parts

data or instruction cache

physical address

39

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

+

1st PTE
addr.

valid, etc?

split
PTE parts

×
page
size

2nd PTE
addr.

×
PTE
size

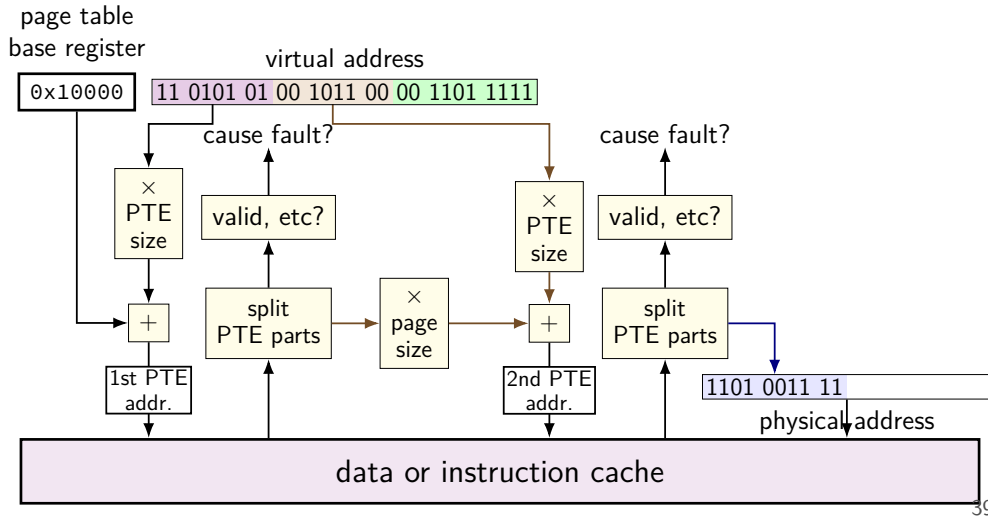
+

data or instruction cache

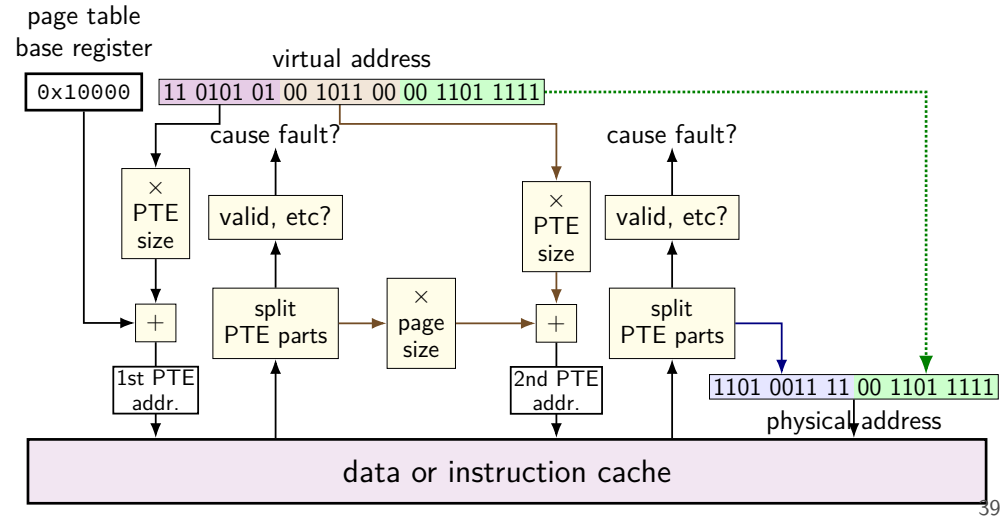
physical address

39

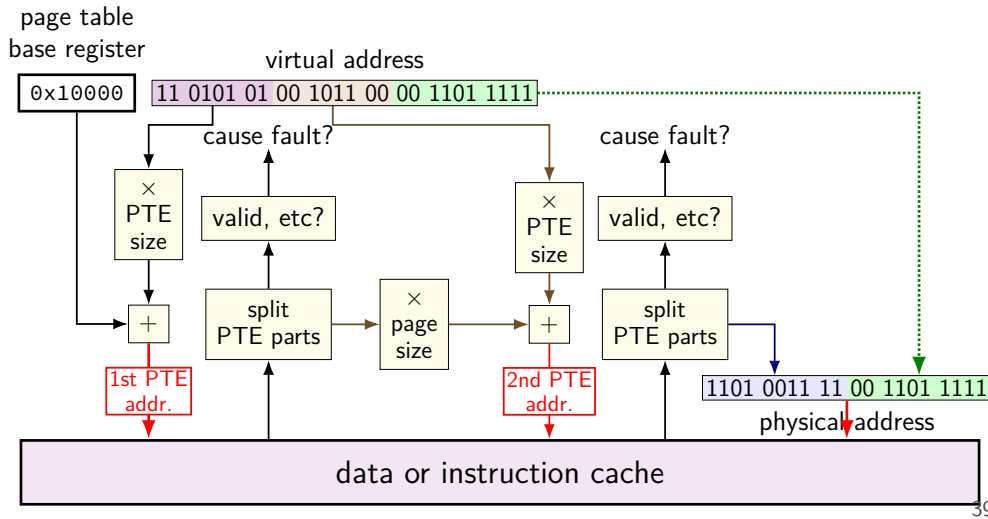
two-level page table lookup



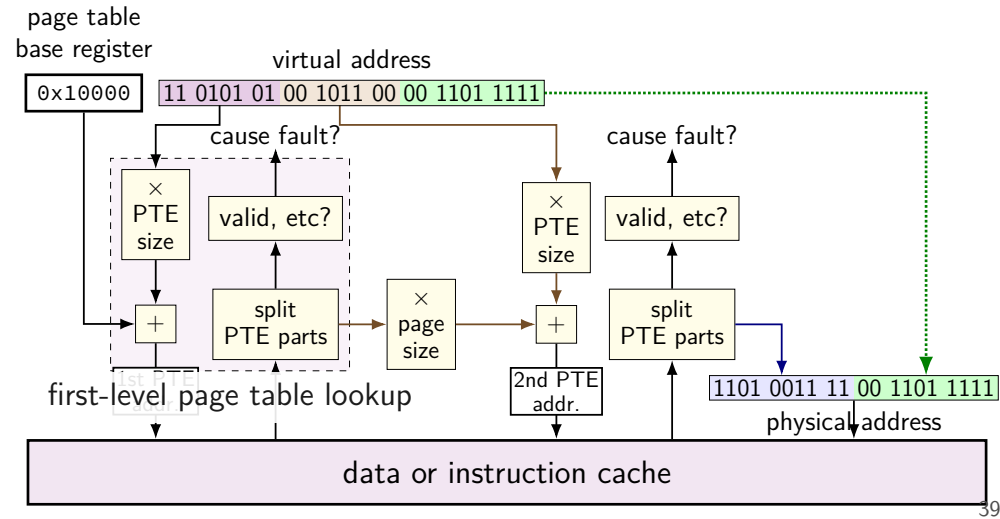
two-level page table lookup



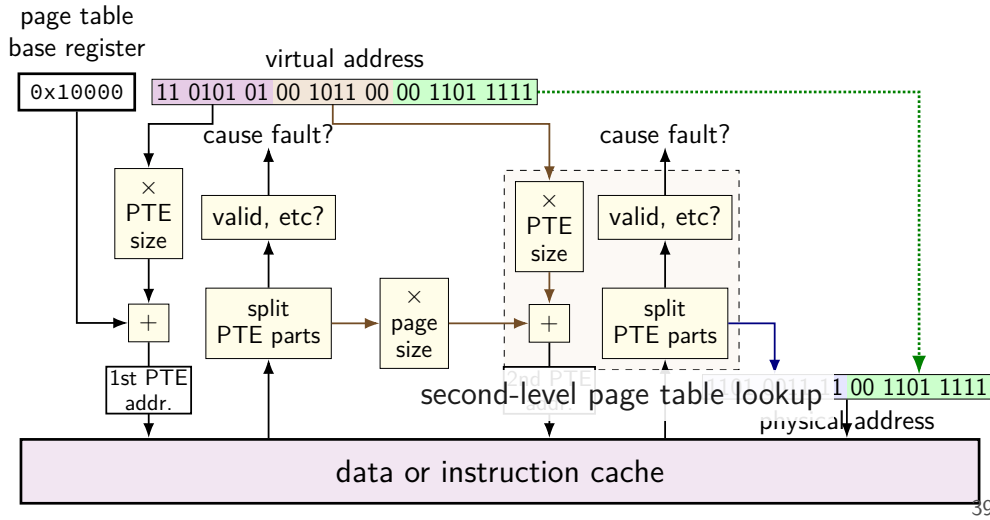
two-level page table lookup



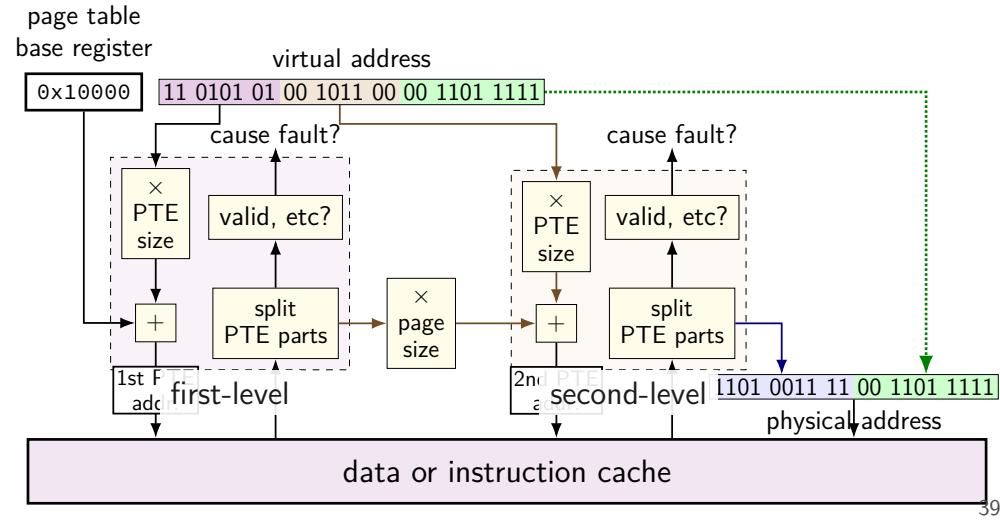
two-level page table lookup



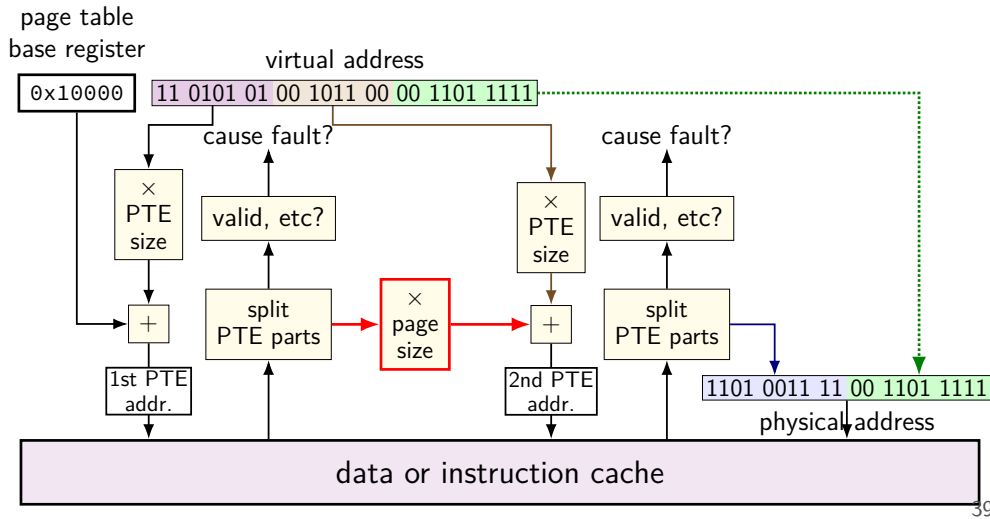
two-level page table lookup



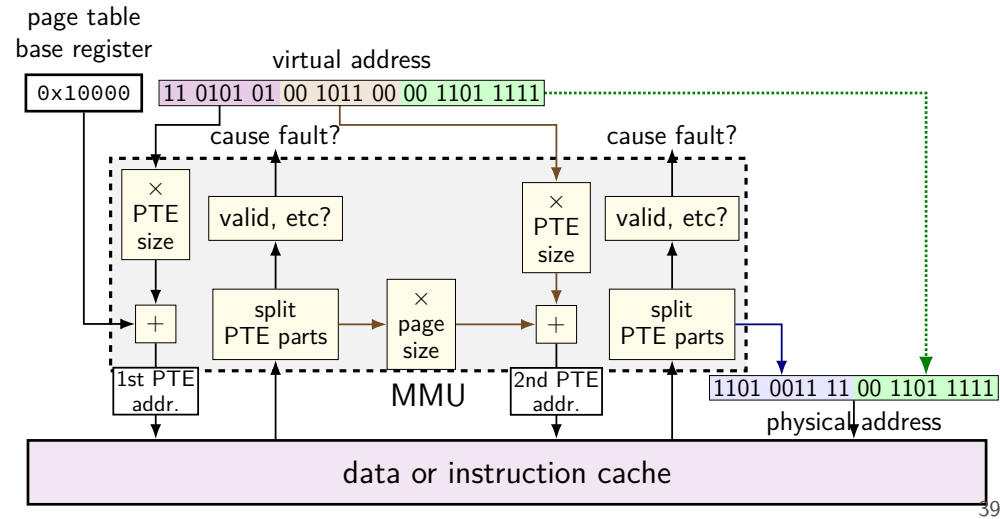
two-level page table lookup



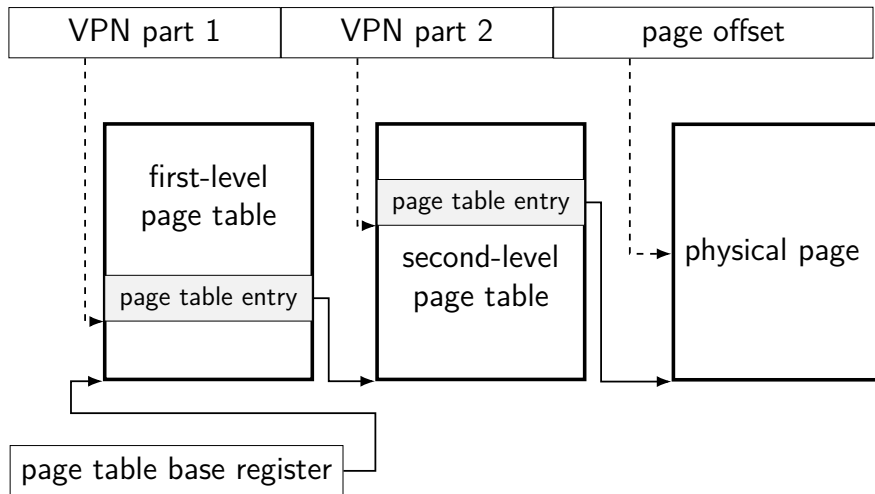
two-level page table lookup



two-level page table lookup



another view



40

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**

41

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)

42

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?

43

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?

43

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?

43

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

43

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 0x30

physical addresses	bytes	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	CB 0B CB 0B
0x18-B	1C 2C 3C 4C	0x38-B	DC 0C DC 0C
0x1C-F	1C 2C 3C 4C	0x3C-F	EC 0C EC 0C

44

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 0x30

physical addresses	bytes	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	CB 0B CB 0B
0x18-B	1C 2C 3C 4C	0x38-B	DC 0C DC 0C
0x1C-F	1C 2C 3C 4C	0x3C-F	EC 0C EC 0C

$0x30 = 11\ 0000$
PTE addr:
 $0x20 + 6 \times 1 = 0x26$
PTE value:
 $0xD6 = 1101\ 0110$
 PPN 110, valid 1
 $M[110\ 000] = M[0x30]$

44

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 0x30

physical addresses	bytes	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	CB 0B CB 0B
0x18-B	1C 2C 3C 4C	0x38-B	DC 0C DC 0C
0x1C-F	1C 2C 3C 4C	0x3C-F	EC 0C EC 0C

$0x30 = 11\ 0000$
PTE addr:
 $0x20 + 6 \times 1 = 0x26$
PTE value:
 $0xD6 = 1101\ 0110$
 PPN 110, valid 1
 $M[110\ 000] = M[0x30]$

44

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 0x30

physical addresses	bytes	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	CB 0B CB 0B
0x18-B	1C 2C 3C 4C	0x38-B	DC 0C DC 0C
0x1C-F	1C 2C 3C 4C	0x3C-F	EC 0C EC 0C

$0x30 = 11\ 0000$
PTE addr:
 $0x20 + 6 \times 1 = 0x26$
PTE value:
 $0xD6 = 1101\ 0110$
 PPN 110, valid 1
 $M[110\ 000] = M[0x30]$

44

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	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	DB 0B DB 0B
0x18-B	1C 2C 3C 4C	0x38-B	EC 0C EC 0C
0x1C-F	1C 2C 3C 4C	0x3C-F	FC 0C FC 0C

45

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	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	DB 0B DB 0B
0x18-B	1C 2C 3C 4C	0x38-B	EC 0C EC 0C
0x1C-F	1C 2C 3C 4C	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

45

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	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	DB 0B DB 0B
0x18-B	1C 2C 3C 4C	0x38-B	EC 0C EC 0C
0x1C-F	1C 2C 3C 4C	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
 PTE 2 addr:
 $110\ 000 + 110 = 0x36$
 PTE 2 value: 0xDB

45

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	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	DB 0B DB 0B
0x18-B	1C 2C 3C 4C	0x38-B	EC 0C EC 0C
0x1C-F	1C 2C 3C 4C	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
 PTE 2 addr:
 $110\ 000 + 110 = 0x36$
 PTE 2 value: 0xDB
 PPN 110; valid 1
 $M[110\ 001\ (0x31)] = 0xA$

45

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	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	DB 0B DB 0B
0x18-B	1C 2C 3C 4C	0x38-B	EC 0C EC 0C
0x1C-F	1C 2C 3C 4C	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
 PTE 2 addr:
 $110\ 000 + 110 = 0x36$
 PTE 2 value: 0xDB
 PPN 110; valid 1
 $M[110\ 001\ (0x31)] = 0xA$

45

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	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	DB 0B DB 0B
0x18-B	1C 2C 3C 4C	0x38-B	EC 0C EC 0C
0x1C-F	1C 2C 3C 4C	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
PTE 2 addr:
 $110\ 000 + 110 = 0x36$
PTE 2 value: 0xDB
 PPN 110; valid 1
 $M[110\ 001\ (0x31)] = 0x0A$

45

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	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	DB 0B DB 0B
0x18-B	1C 2C 3C 4C	0x38-B	EC 0C EC 0C
0x1C-F	1C 2C 3C 4C	0x3C-F	FC 0C FC 0C

46

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	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	DB 0B DB 0B
0x18-B	1C 2C 3C 4C	0x38-B	EC 0C EC 0C
0x1C-F	1C 2C 3C 4C	0x3C-F	FC 0C FC 0C

$0x0FB = 011\ 111\ 011$
PTE 1: 0xBB at 0x0B
PTE 1: PPN 101 (5) valid 1
PTE 2: 0xF0 at 0x2F
PTE 2: PPN 111 (7) valid 1
 $111\ 011 = 0x3B \rightarrow 0x0C$

46

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	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	DB 0B DB 0B
0x18-B	1C 2C 3C 4C	0x38-B	EC 0C EC 0C
0x1C-F	1C 2C 3C 4C	0x3C-F	FC 0C FC 0C

$0x0FB = 011\ 111\ 011$
PTE 1: 0xBB at 0x0B
PTE 1: PPN 101 (5) valid 1
PTE 2: 0xF0 at 0x2F
PTE 2: PPN 111 (7) valid 1
 $111\ 011 = 0x3B \rightarrow 0x0C$

46

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	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	DB 0B DB 0B
0x18-B	1C 2C 3C 4C	0x38-B	EC 0C EC 0C
0x1C-F	1C 2C 3C 4C	0x3C-F	FC 0C FC 0C

0x0F3 = 011 111 011
 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

46

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	physical addresses	bytes
0x00-3	00 11 22 33	0x20-3	D0 D1 D2 D3
0x04-7	44 55 66 77	0x24-7	D4 D5 D6 D7
0x08-B	88 99 AA BB	0x28-B	89 9A AB BC
0x0C-F	CC DD EE FF	0x2C-F	CD DE EF F0
0x10-3	1A 2A 3A 4A	0x30-3	BA 0A BA 0A
0x14-7	1B 2B 3B 4B	0x34-7	DB 0B DB 0B
0x18-B	1C 2C 3C 4C	0x38-B	EC 0C EC 0C
0x1C-F	1C 2C 3C 4C	0x3C-F	FC 0C FC 0C

0x0F3 = 011 111 011
 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

46