

## virtual memory 2

# Changelog

17 November 2020: move example of page table lookup using memory (1-level example) earlier and change numbers

17 November 2020: relocate swapping is writeback slides to backup

# last time

exception tables

sync v. async exceptions

kernel mode and privileged instructions

address space: illusion of own memory

address translation:

processor maps virtual (program) to physical (actual) addresses

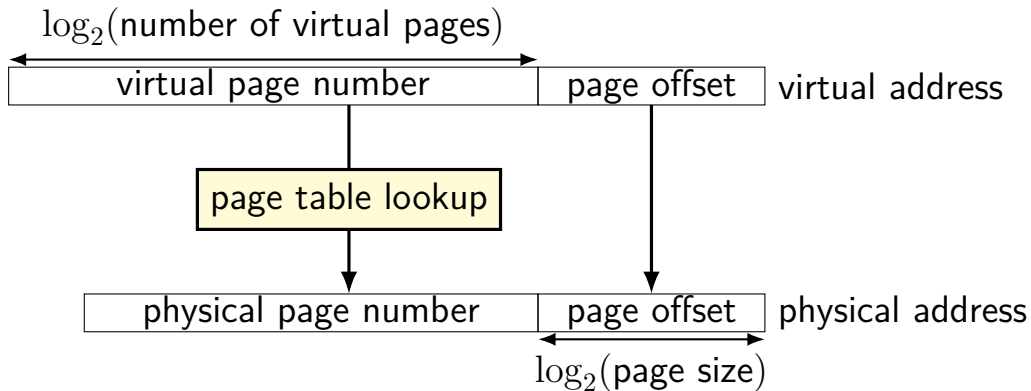
pages: divide up virtual and physical memory into fixed-size pieces

address split into page number bits + page offset bits

page table

virtual page number  $\rightarrow$  (valid, physical page number, permissions, ...)

# address split diagram



## exercise: page counting

suppose 32-bit virtual (program) addresses

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

how many virtual pages?

## exercise: page counting

suppose 32-bit virtual (program) addresses

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

how many virtual pages?

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

## exercise: page table size

suppose 32-bit virtual (program) addresses

suppose 30-bit physical (hardware) addresses

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

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

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

## exercise: page table size

suppose 32-bit virtual (program) addresses

suppose 30-bit physical (hardware) addresses

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

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

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

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

issue: where can we store that?



## exercise: address splitting

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

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

## exercise: address splitting

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

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

page #: 0x12345; offset: 0x678

# page tables in memory

where can processor store megabytes of page tables? **in memory**

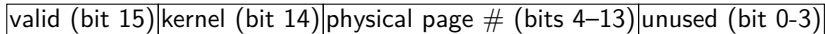
page table entry layout

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

# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout



page table base register

0x00010000



# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout

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

page table base register

0x00010000



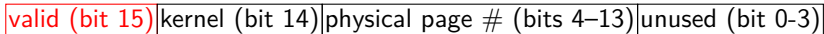
physical memory

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

# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout



page table base register

0x00010000



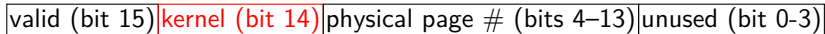
physical memory

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

# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout



page table base register

0x00010000



physical memory

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

# page tables in memory

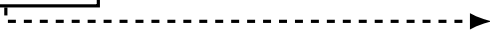
where can processor store megabytes of page tables? **in memory**

page table entry layout



page table base register

0x00010000



physical memory

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



# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout

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

page table base register

0x00010000

page table (logically)

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

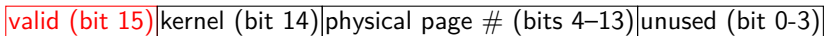
physical memory

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

# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout



page table base register

0x00010000

page table (logically)

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

physical memory

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

# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout



page table base register

0x00010000

page table (logically)

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

physical memory

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

# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout

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

page table base register

0x00010000

page table (logically)

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

physical memory

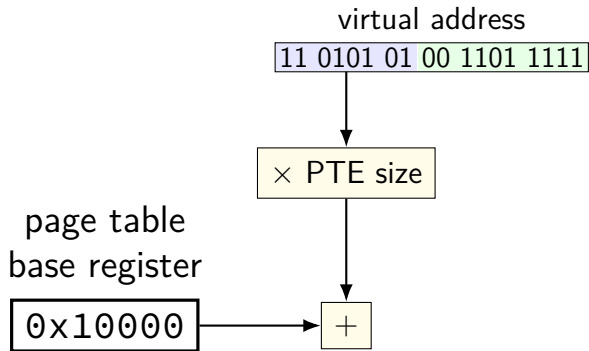
addresses	bytes
0x00000000-1	00000000 00000000
...	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
...	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

# memory access with page table

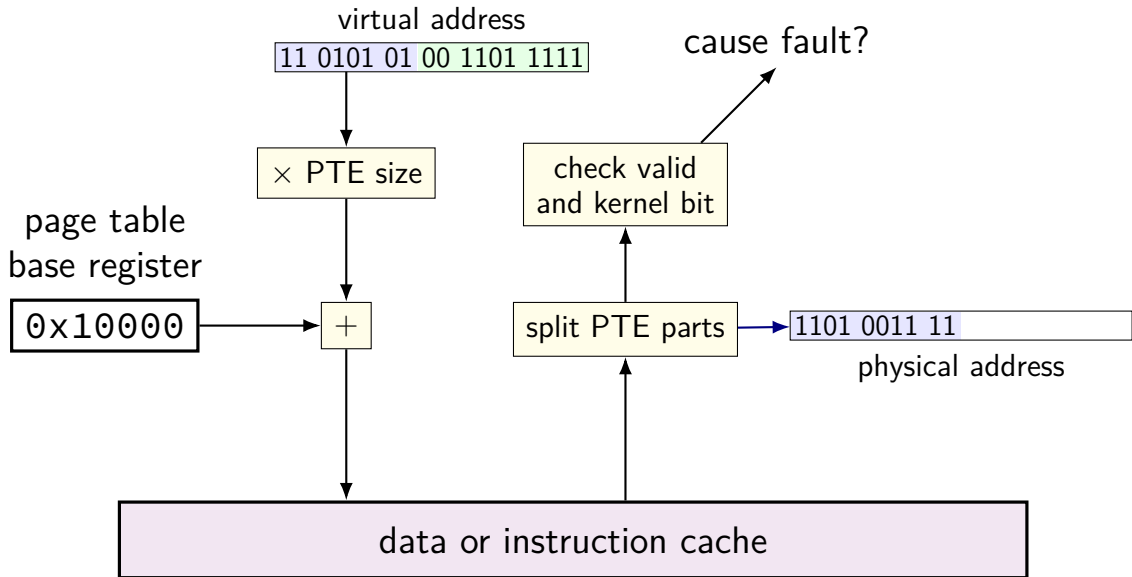
virtual address

11	0101	01	00	1101	1111
----	------	----	----	------	------

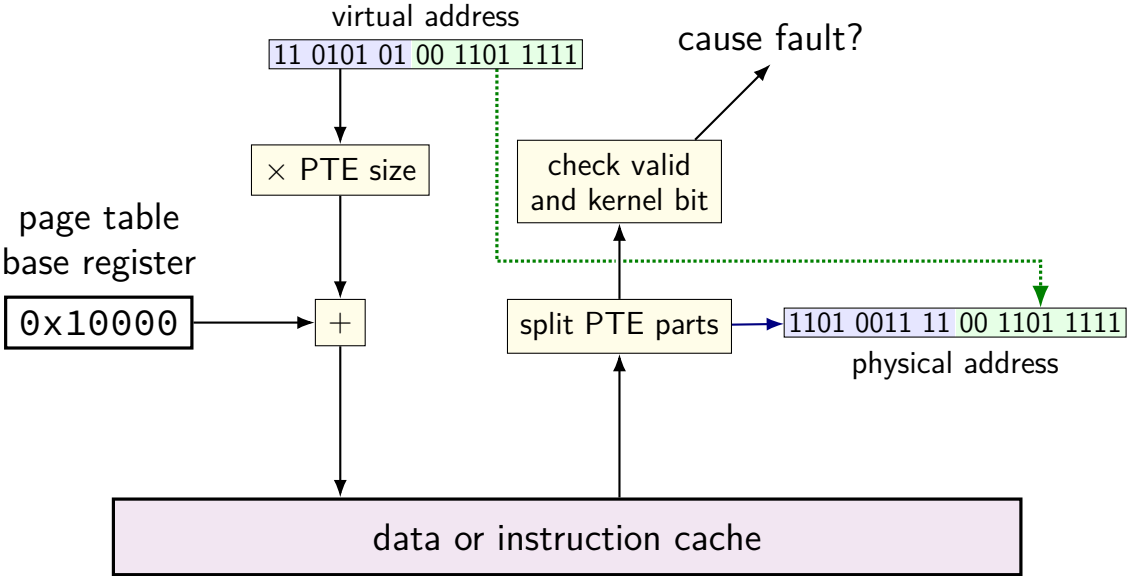
# memory access with page table



# memory access with page table

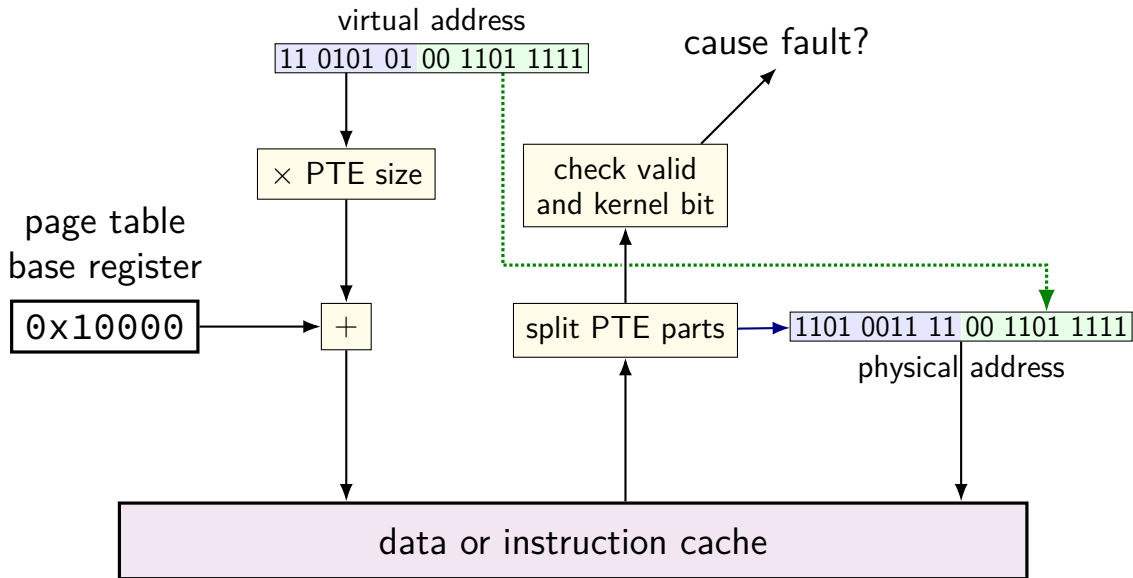


# memory access with page table

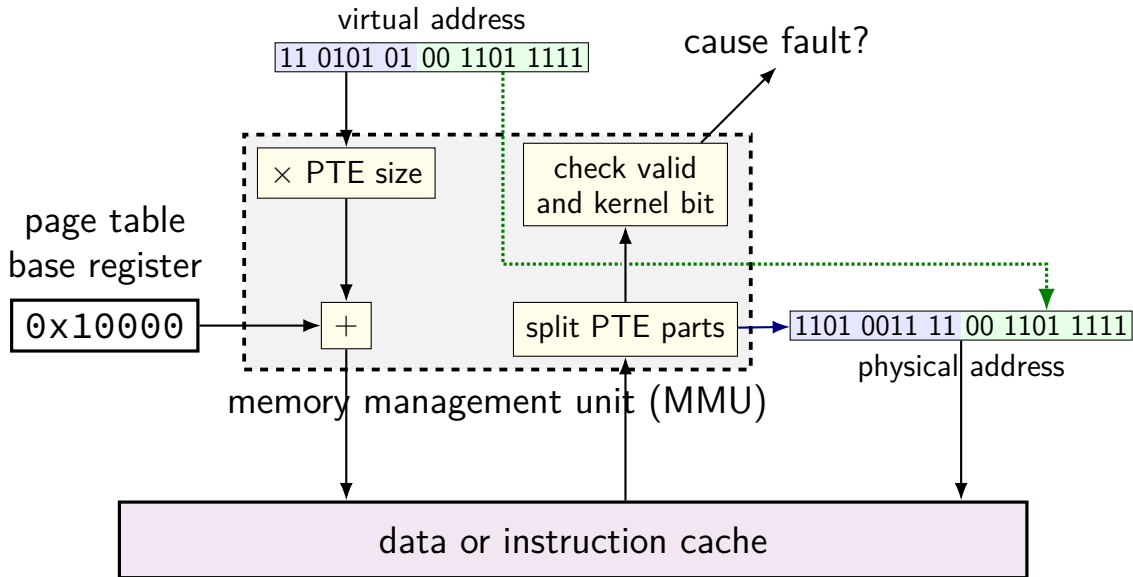




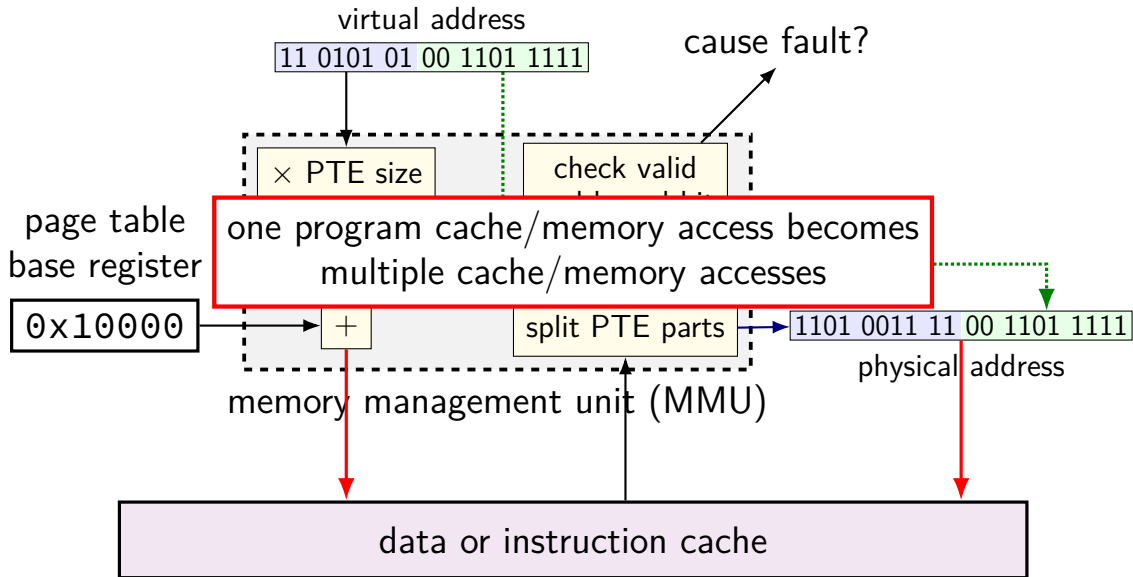
# memory access with page table



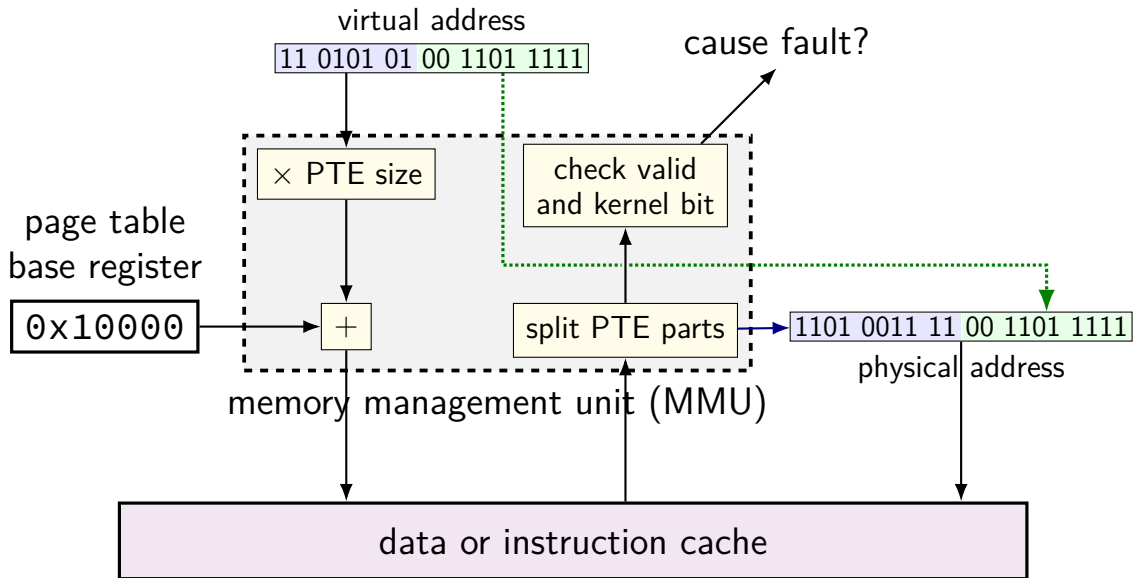
# memory access with page table



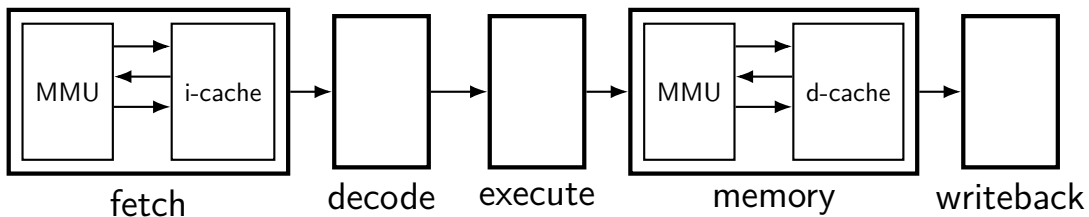
# memory access with page table



# memory access with page table

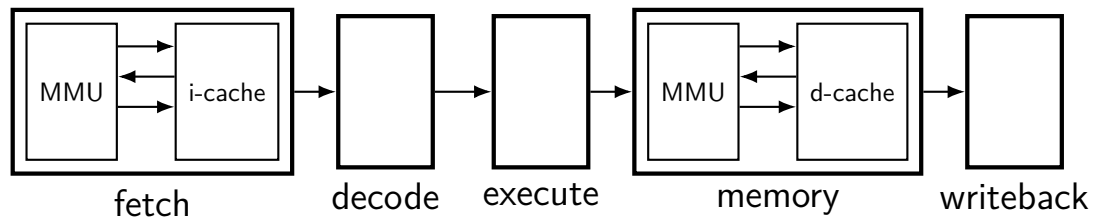


# MMUs in the pipeline



up to four memory accesses per instruction

## MMUs in the pipeline



up to four memory accesses per instruction

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

## exercise: 64-bit system

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

4096 byte pages

## exercise: 64-bit system

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

4096 byte pages

top 16 bits of address not used for translation



## exercise: 64-bit system

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

4096 byte pages

exercise: how many page table entries? (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??)

## two big problems to solve later

two **extra cache accesses** seems really slow

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

page tables seem **really huge**

solution: not just a flat table

# exercise setup

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

page table

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

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

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

# exercise setup

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

page table

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

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

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	EA AB BC
0x2C-F	CE 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

phys. page 0

phys. page 1

# exercise

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

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

page table

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

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

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

# exercise

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

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

page table

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

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

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



# exercise

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

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

page table

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

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

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

# exercise

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

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

page table

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

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

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

# exercise

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

(virtual addresses)  $0x18 = 00$ ;  $0x03 = 0x4A$ ;  $0x0A = 0xDC$ ;  $0x13 = \text{fault}$

page table

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

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

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

# 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
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  $0x30$

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

$0x30 = 11\ 0000$

*PTE addr:*

$0x20 + 6 \times 1 = 0x26$

*PTE value:*

$0xF6 = 1111\ 0110$

PPN 111, valid 1

$M[111\ 000] = M[0x38]$

$\rightarrow 0xDC$

# 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
$0x00-3$	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

$0x30 = 11\ 0000$

*PTE addr:*

$0x20 + 6 \times 1 = 0x26$

*PTE value:*

$0xF6 = 1111\ 0110$

PPN **111**, valid 1

$M[111\ 000] = M[0x38]$

$\rightarrow 0xDC$

# 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
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

0x30 = 11 0000

*PTE addr:*

$0x20 + 6 \times 1 = 0x26$

*PTE value:*

0xF6 = 1111 0110

PPN 111, valid 1

$M[111 \ 000] = M[0x38]$

→ 0xDC

# memory HW

due next Tuesday (24 November) at midnight

doing some manual translations similar to this

plus a few translations with multi-level tables

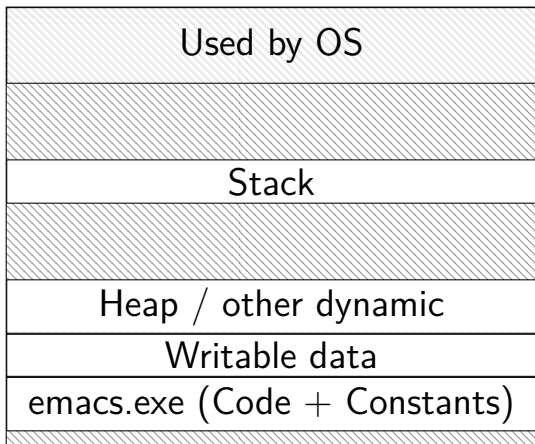
we'll cover later: solution to tables being too big

explanation on homework that might be sufficient before we do so

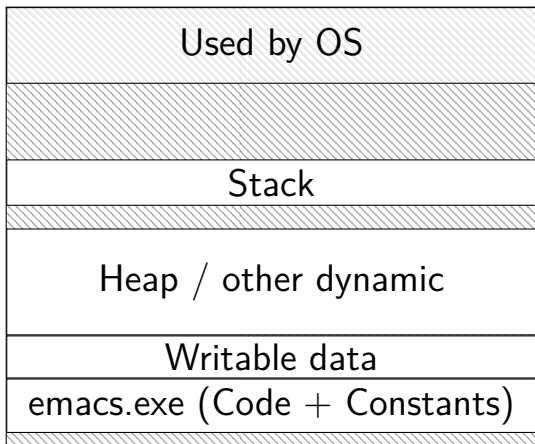


# emacs (two copies)

Emacs (run by user mst3k)

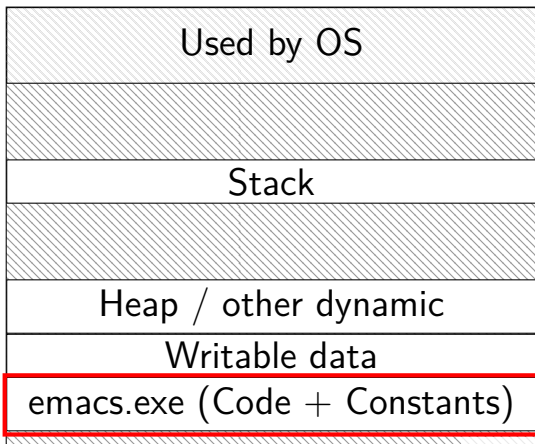


Emacs (run by user xyz4w)

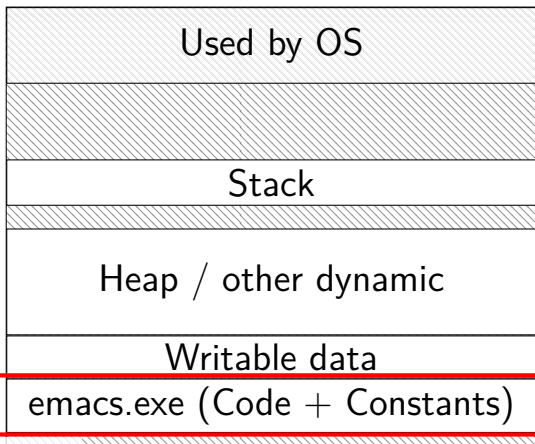


# emacs (two copies)

Emacs (run by user mst3k)



Emacs (run by user xyz4w)



same data?

## two copies of program

would like to only have one copy of program

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

would break process abstraction:

“illusion of own memory”

# typical page table entries

solution: same idea as kernel-only bit

page table entry will have more **permissions bits**

can read?

can write?

can execute?

checked by MMU like valid/kernel bit

page table (logically)

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

# shared libraries

C standard library has lots of common code:

`printf`

`fopen`

`qsort`

...

would like to not have multiple copies

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

# page table tricks

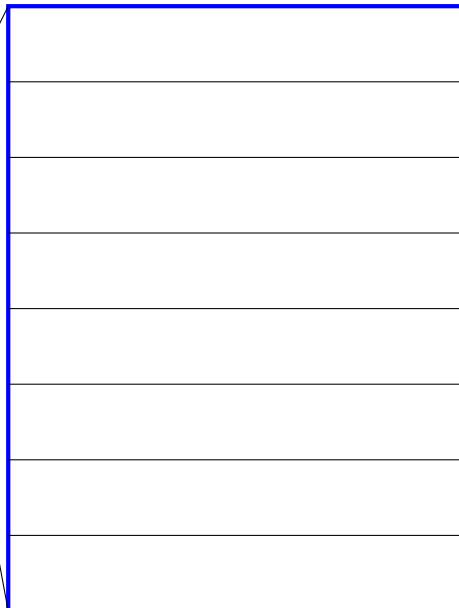
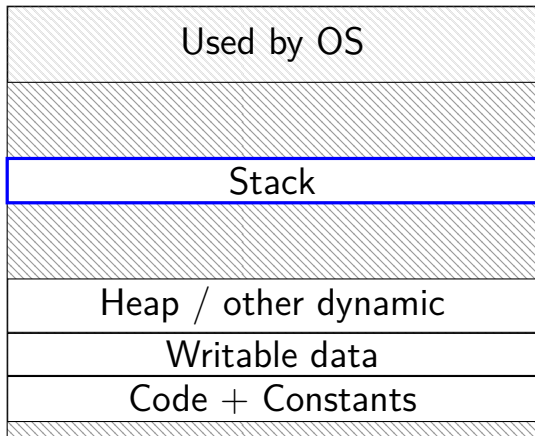
page tables let operating systems do 'magic' with memory

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

instead: change page tables and rerun memory access

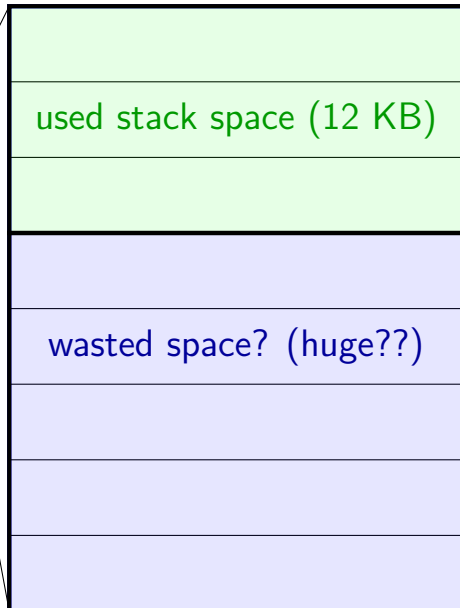
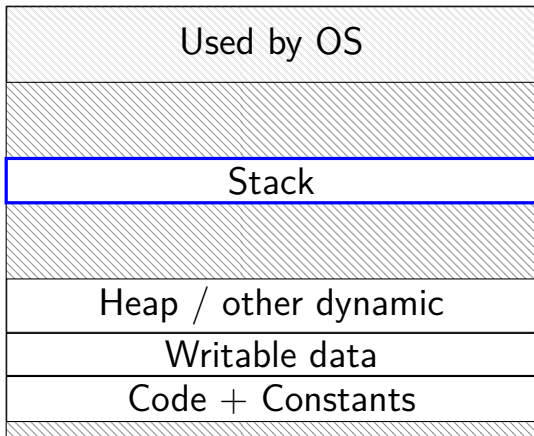
# space on demand

Program Memory



# space on demand

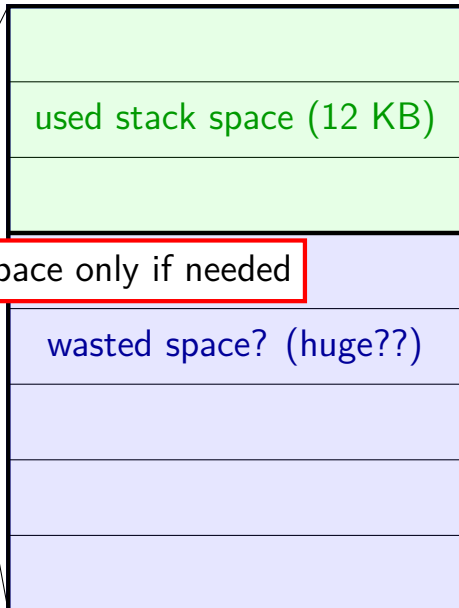
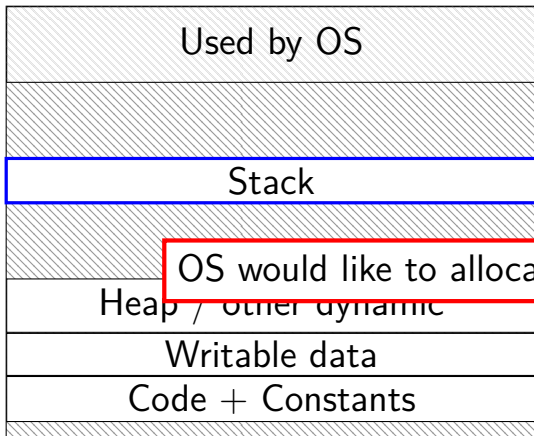
Program Memory





# space on demand

Program Memory



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

```
...  
0x7FFFB  
0x7FFFC  
0x7FFFD  
0x7FFFE  
0x7FFFF  
...
```

valid? physical  
page

valid?	physical page
...	...
0	---
1	0x200DF
1	0x12340
1	0x12347
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

```
...  
0x7FFFB  
0x7FFFC  
0x7FFFD  
0x7FFFE  
0x7FFFF  
...
```

valid? physical  
page

valid?	physical page
...	...
0	---
1	0x200DF
1	0x12340
1	0x12347
1	0x12345
...	...

pushq triggers exception  
hardware says “accessing address 0x7FFBFF8”  
OS looks up what’s should be there — “stack”

# allocating space on demand

`%rsp = 0x7FFFC000`

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

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

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

# allocating space on demand

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

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

loading program can be **merely creating empty page table**

everything else can be handled **in response to page faults**

no time/space spent loading/allocating unneeded space

# page tricks generally

deliberately **make program trigger page/protection fault**

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

have **seperate data structures** represent logically allocated memory

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

page table is for the hardware and not the OS

# hardware help for page table tricks

information about the address causing the fault

e.g. special register with memory address accessed

harder alternative: OS disassembles instruction, look at registers

(by default) rerun faulting instruction when returning from exception

precise exceptions: no side effects from faulting instruction or after

e.g. `pushq` that caused did not change `%rsp` before fault

e.g. instructions reordered after faulting instruction not visible

# 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 reads and writes: hundreds of microseconds

- designed for reads/writes 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 reads 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 reads 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

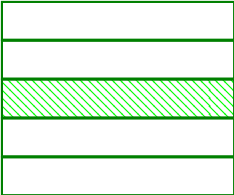
writing tens of **kilobytes** basically as fast as writing 512 bytes

SSD writes and reads: hundreds of microseconds

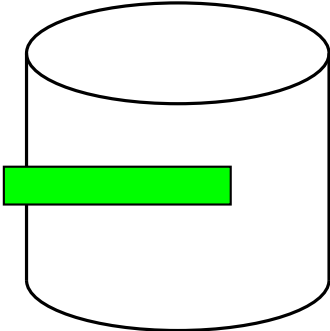
designed for writes/reads of **kilobytes** (not much smaller)

# swapping timeline

program A pages

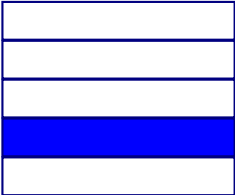


...



disk

program B pages



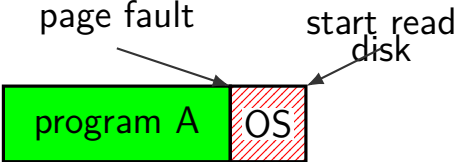
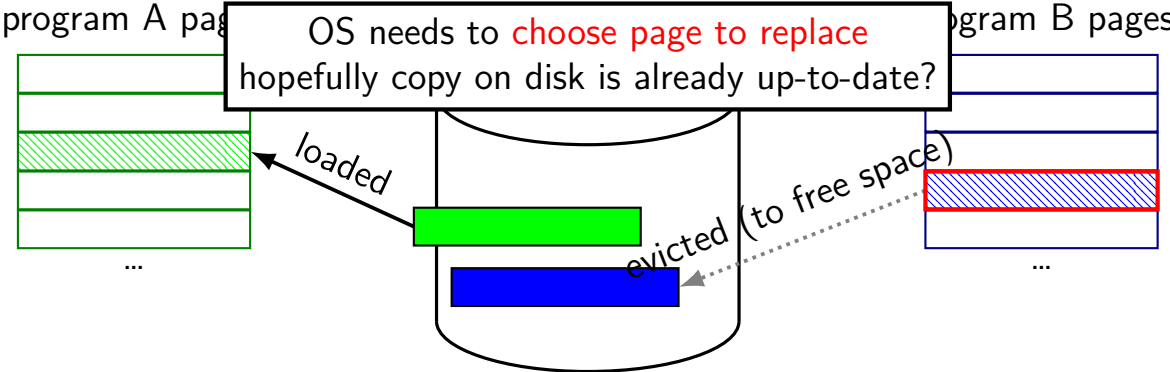
...

page fault

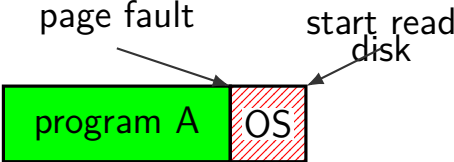
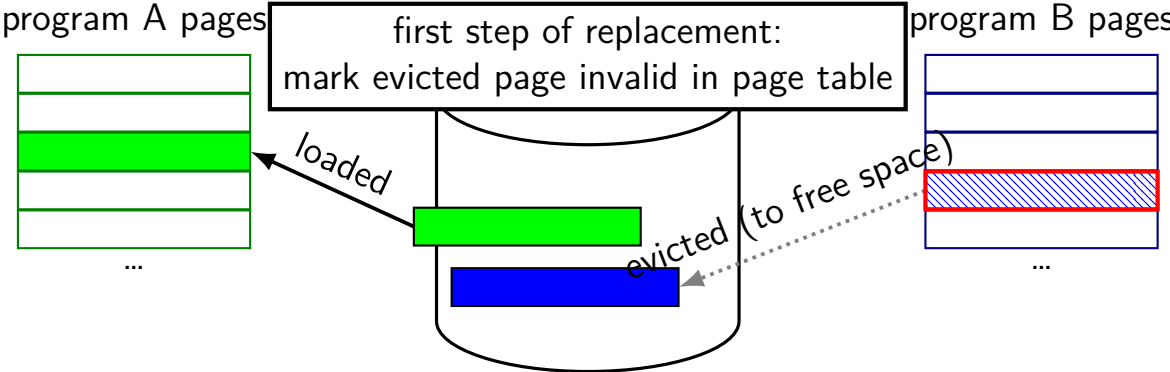




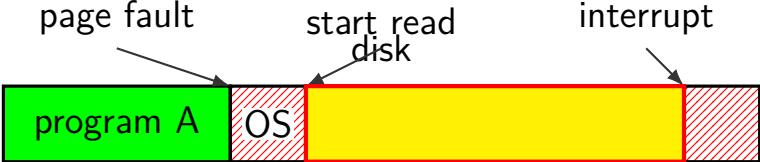
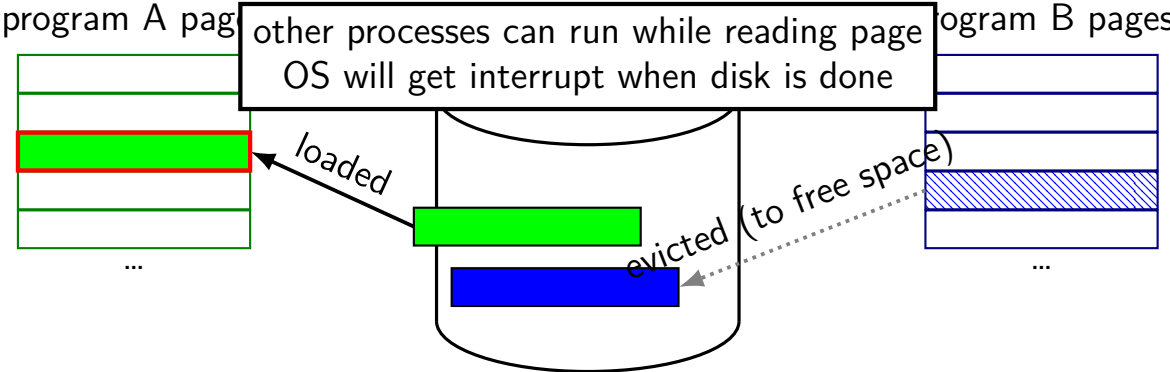
# swapping timeline



# swapping timeline

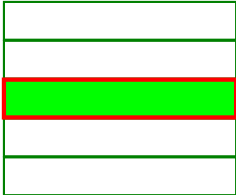


# swapping timeline

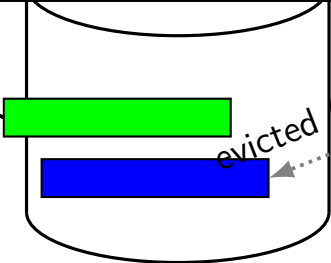


# swapping timeline

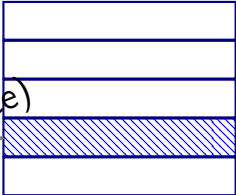
program A pages



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



program B pages



page fault

start read disk

interrupt



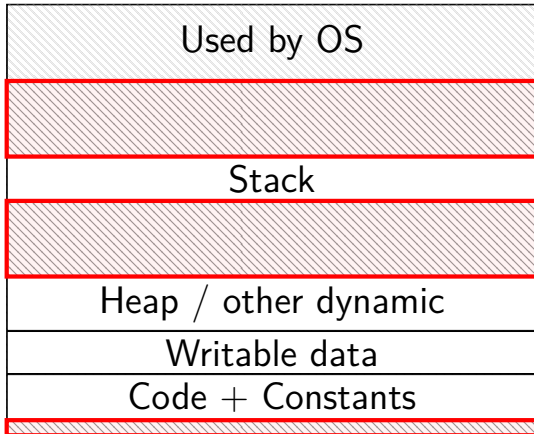
# huge page tables

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**

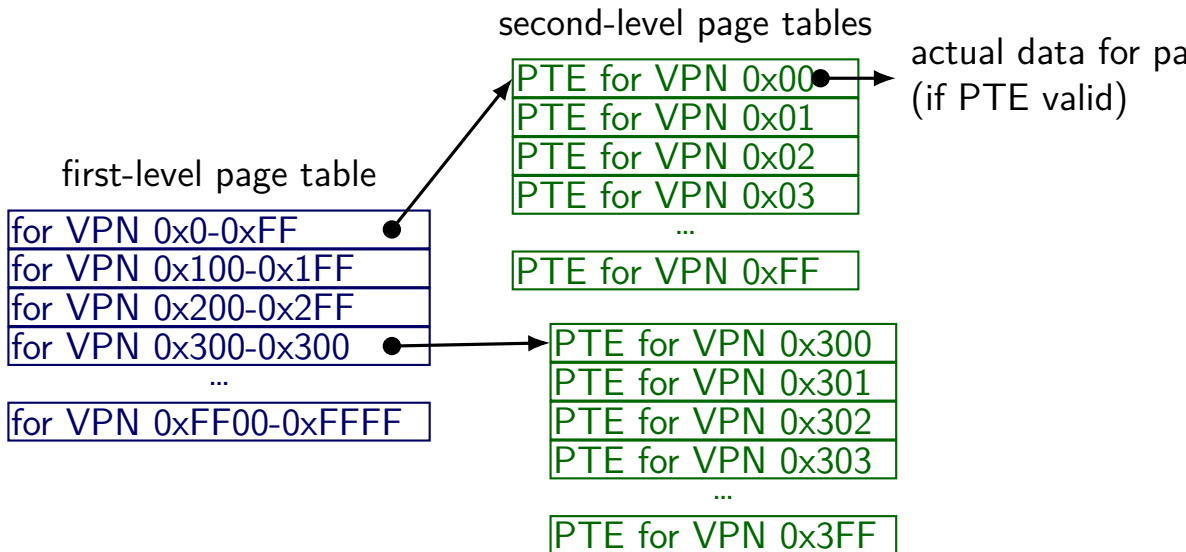
lookup should be simple

lookup should not involve many memory accesses

doing two memory accesses is already very slow

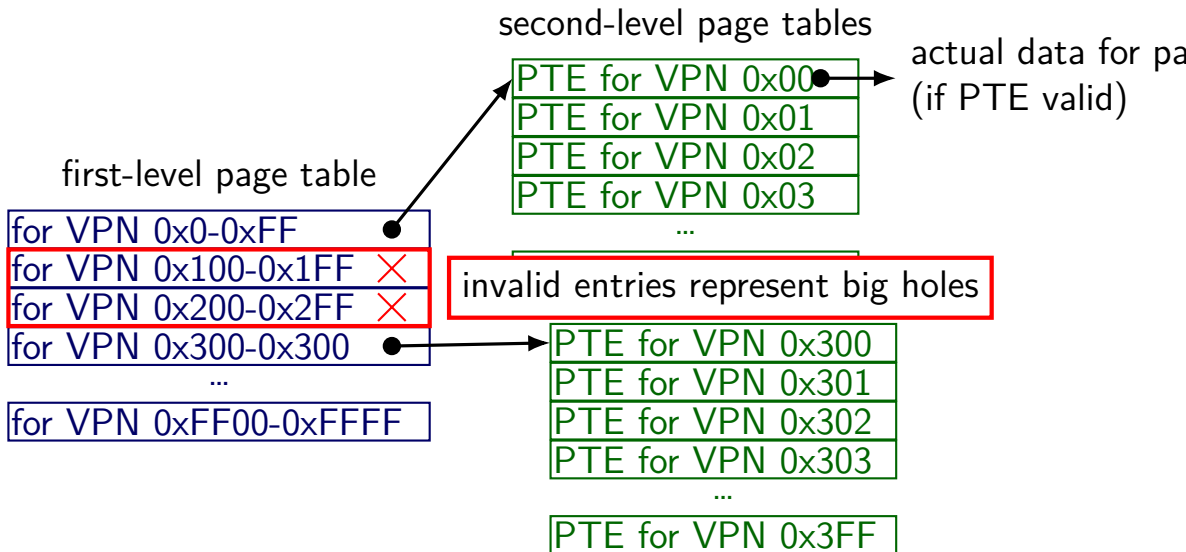
# two-level page tables

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



# two-level page tables

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



# two-level page tables

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

		first-level page table			physical page # (of next page table)
VPN range		valid	kernel	write	
first-level page 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

# two-level page tables

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

		first-level page table			physical page # (of next page table)
VPN range		valid	kernel	write	
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

first-level page table for VPN 0x300-0x3FF

PTE for VPN 0x303

...

PTE for VPN 0x3FF

# two-level page tables

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

		first-level page table			physical page # (of next page table)
VPN range		valid	kernel	write	
first-level page 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

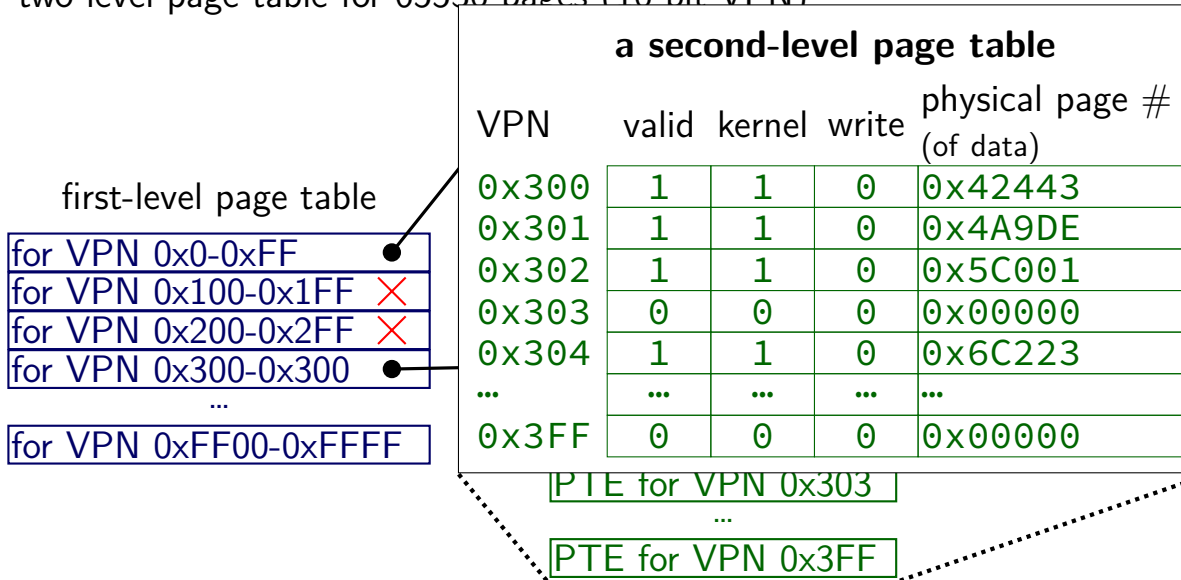
PTE for VPN 0x303

...

PTE for VPN 0x3FF

# two-level page tables

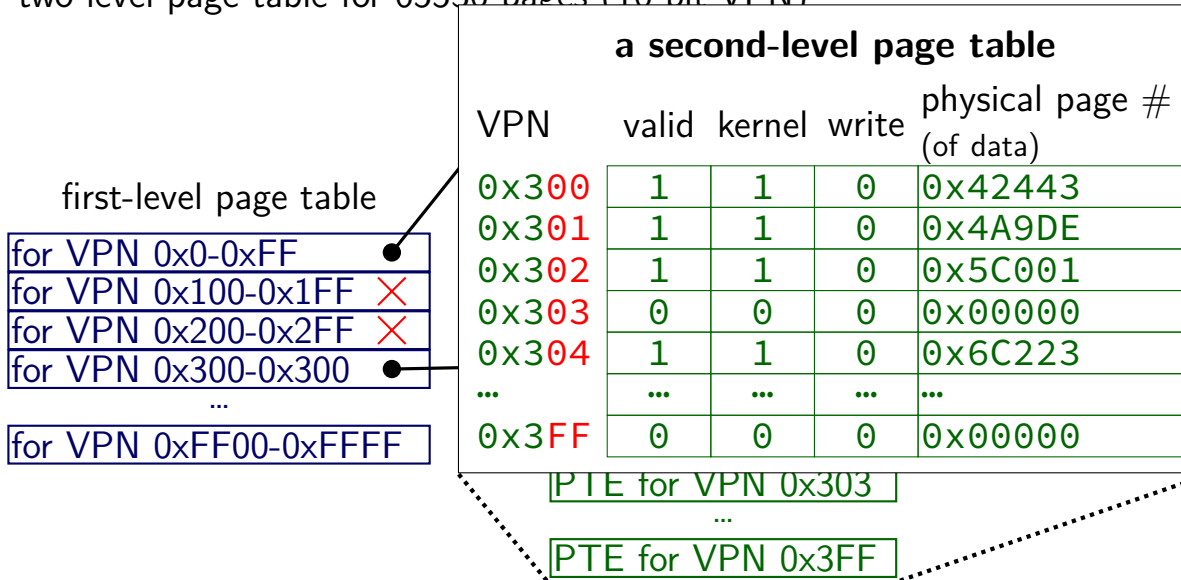
two-level page table for 65536 pages (16-bit VPN)





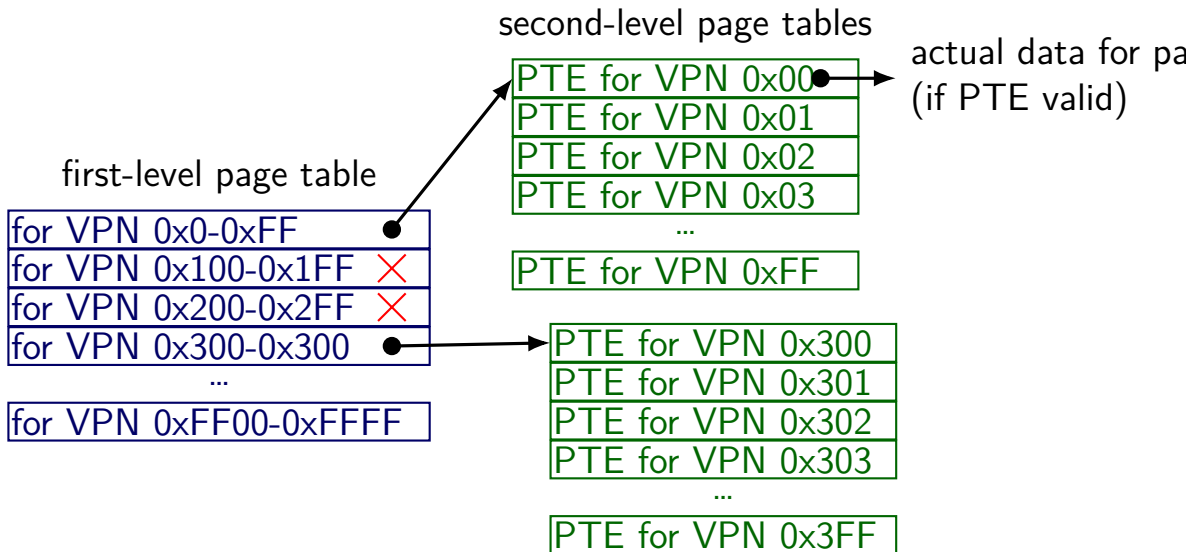
# two-level page tables

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



# two-level page tables

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



# two-level page table lookup

virtual address

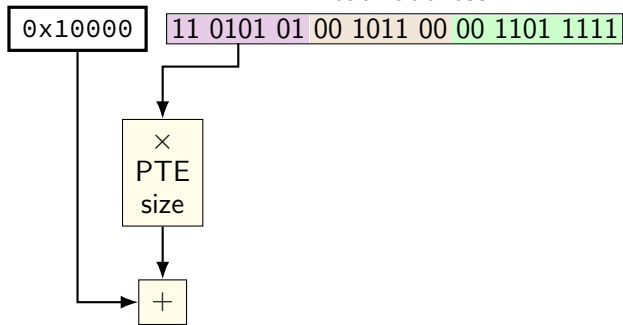
11 0101 01 00 1011 00 00 1101 1111

VPN — split into two parts (one per level)

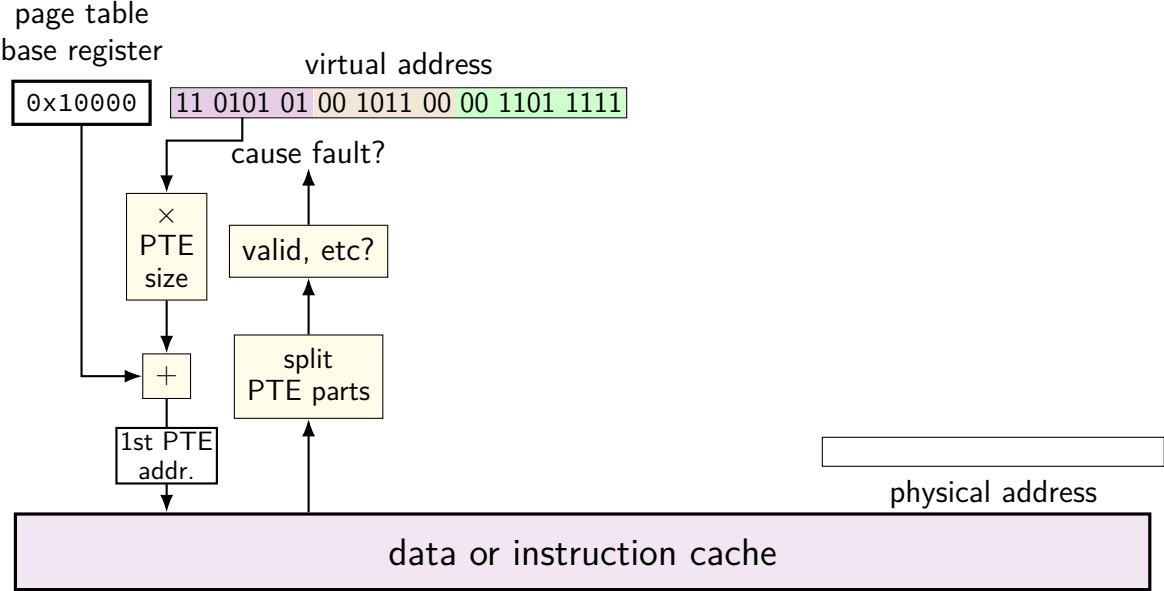
# two-level page table lookup

page table  
base register

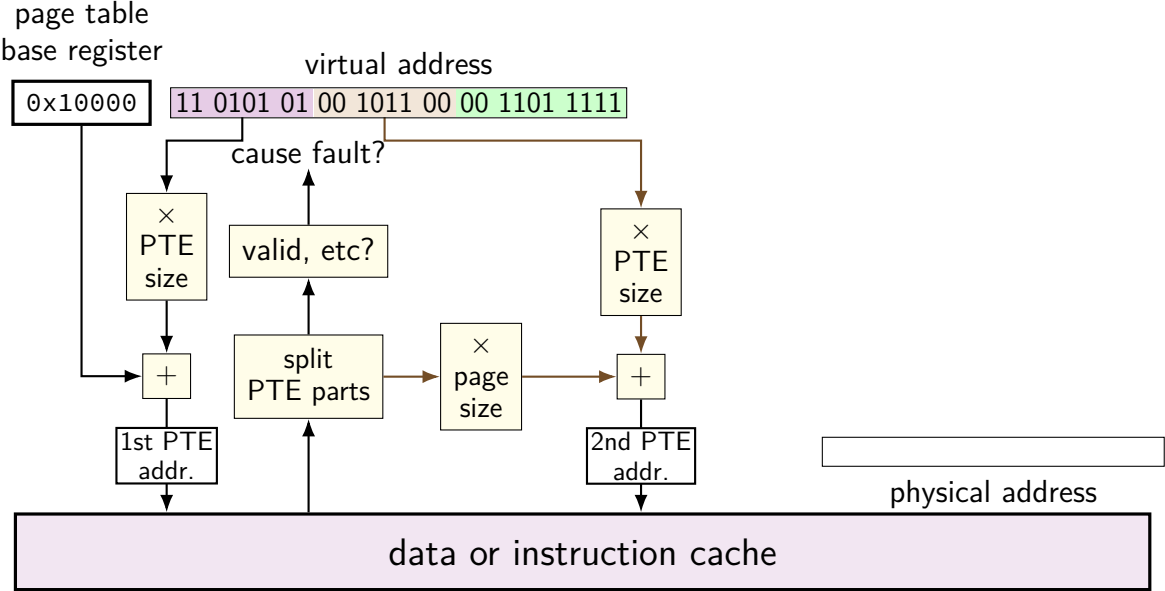
virtual address



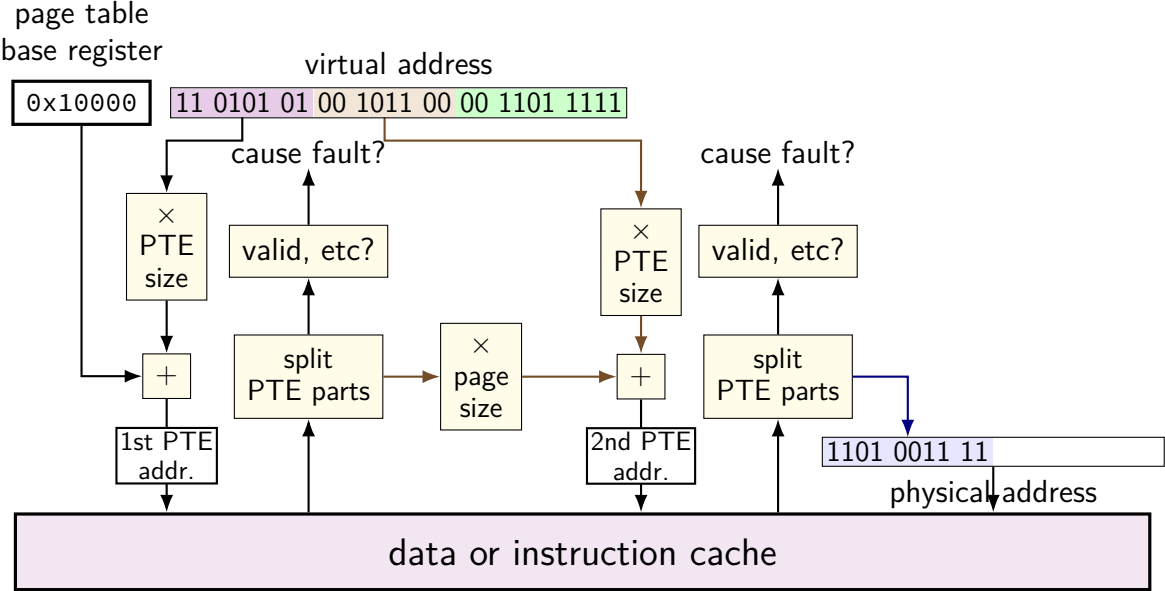
# two-level page table lookup



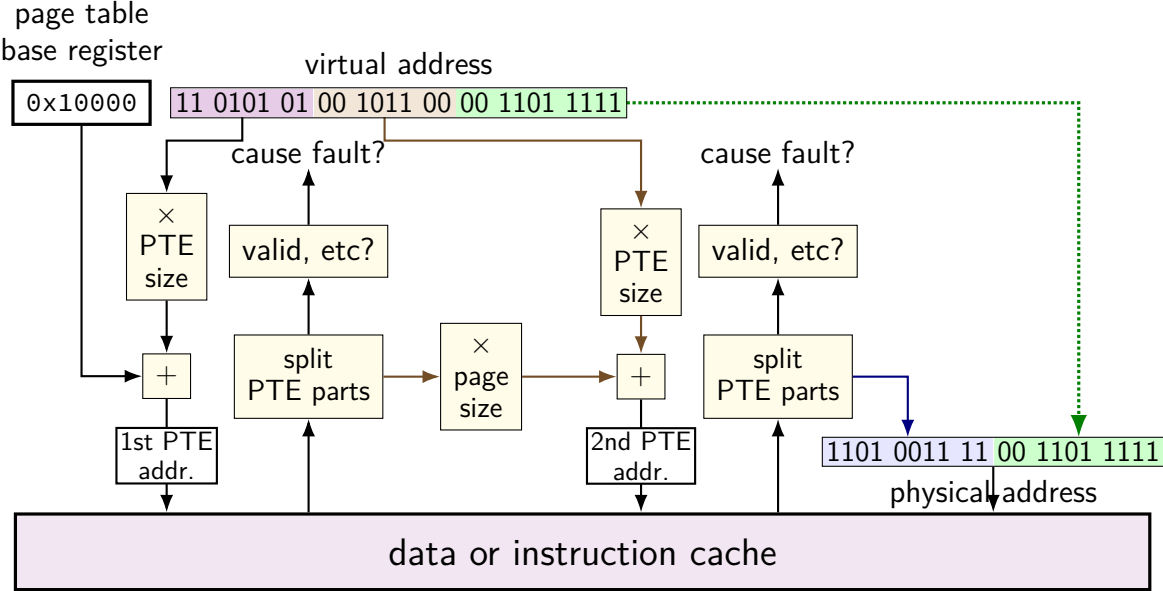
# two-level page table lookup



# two-level page table lookup

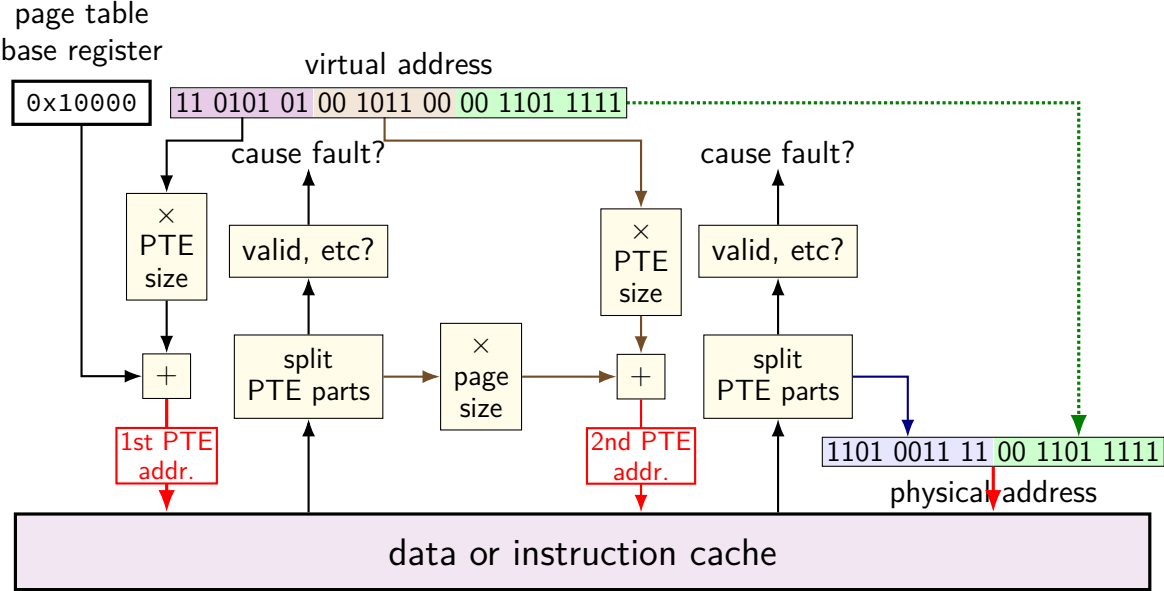


# two-level page table lookup

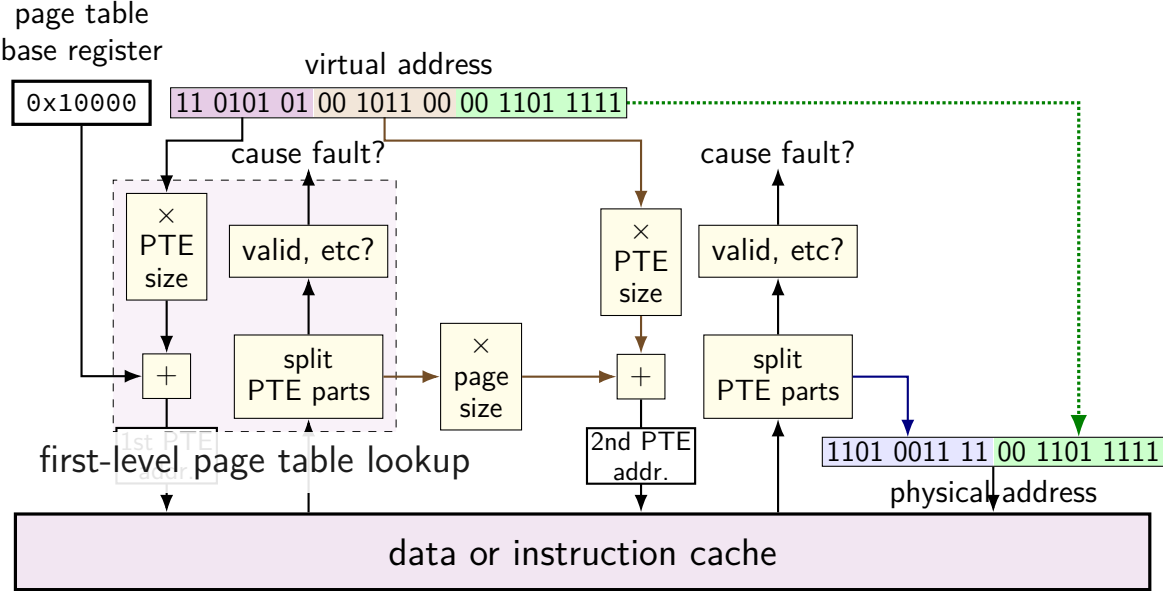




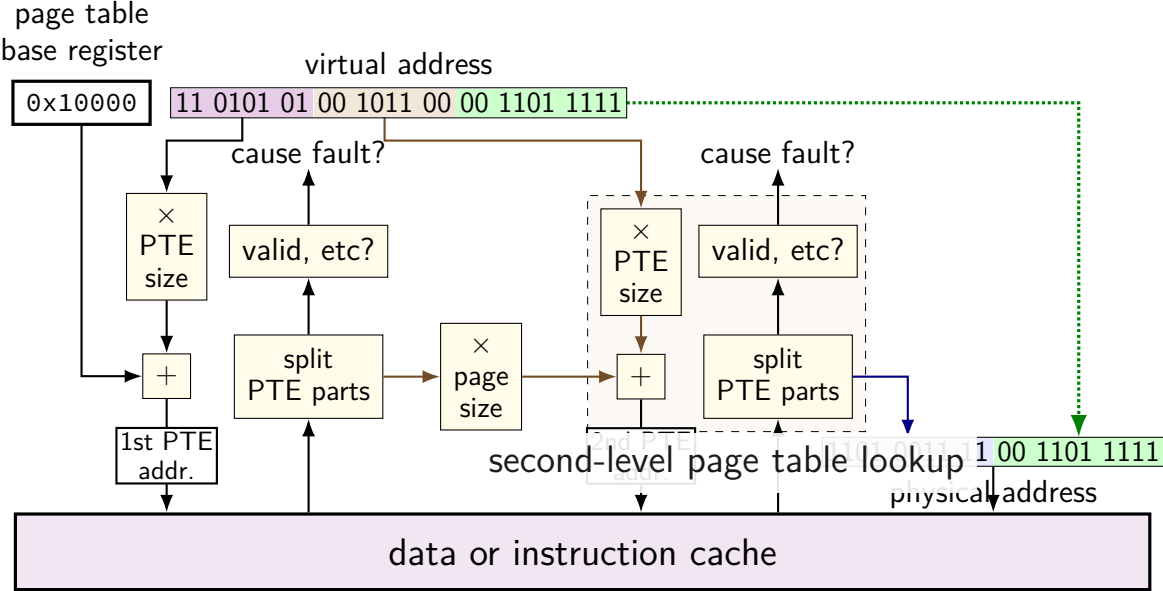
# two-level page table lookup



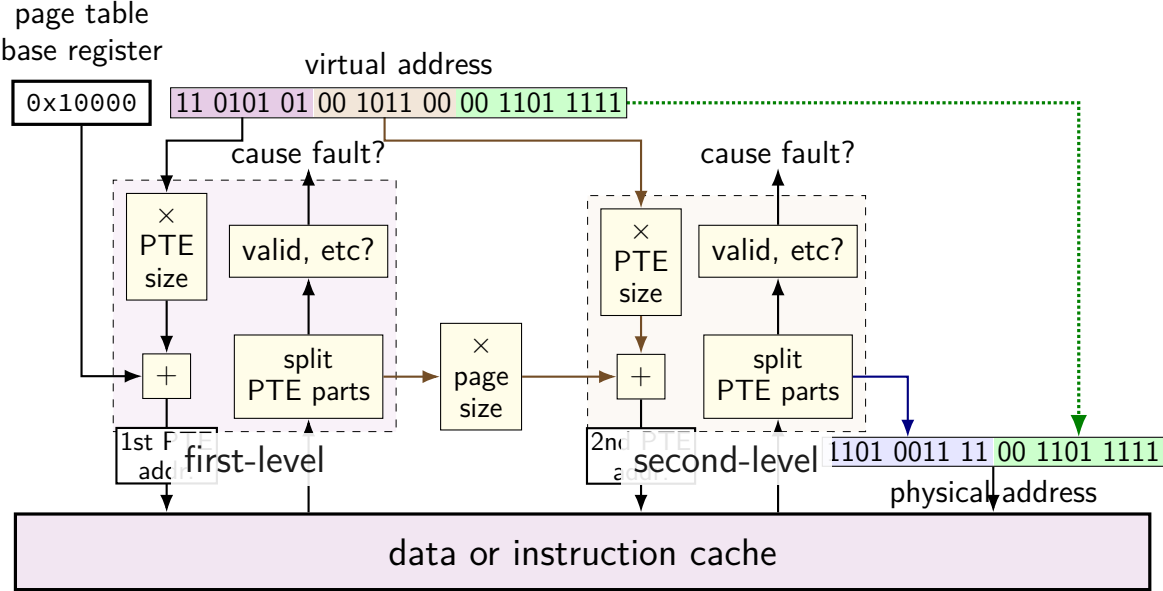
# two-level page table lookup



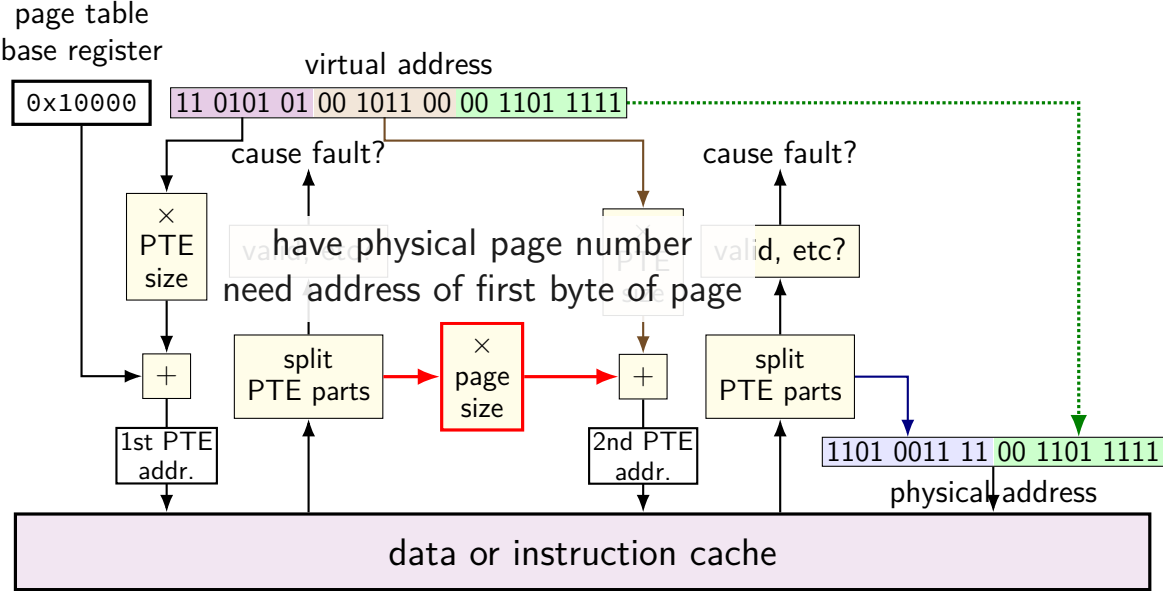
# two-level page table lookup



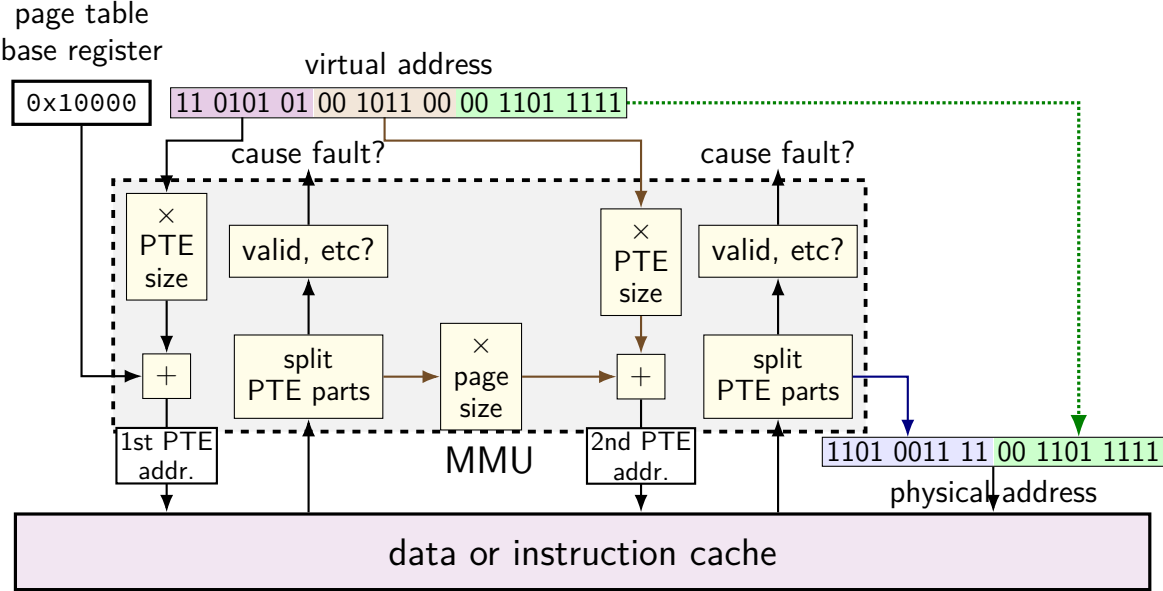
# two-level page table lookup



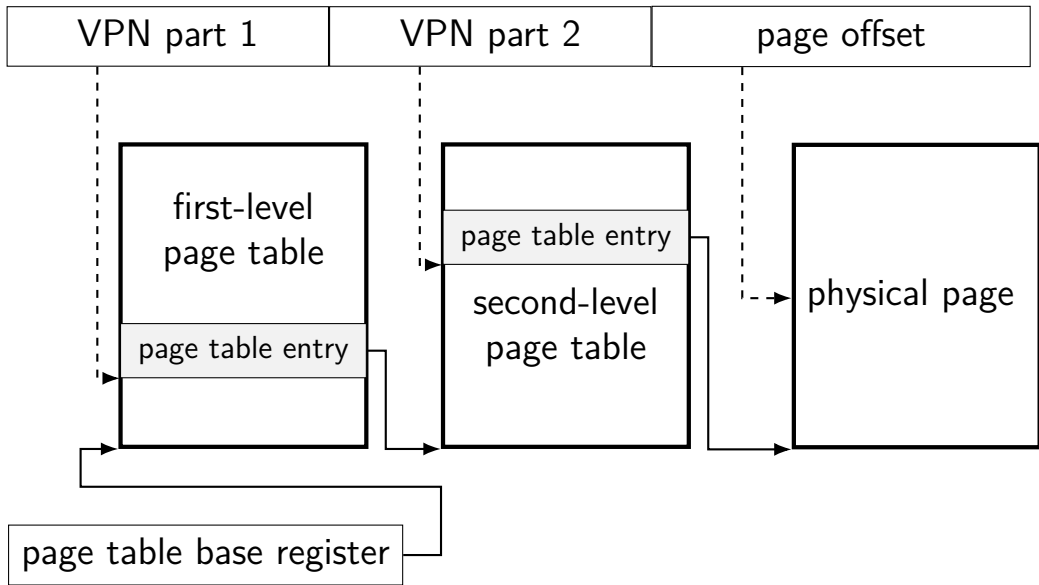
# two-level page table lookup



# two-level page table lookup



## another view



# backup slides



# 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

# 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

## exercise: page counting

suppose 32-bit virtual (program) addresses

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

how many virtual pages?

## exercise: page counting

suppose 32-bit virtual (program) addresses

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

how many virtual pages?

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

## exercise: page table size

suppose 32-bit virtual (program) addresses

suppose 30-bit physical (hardware) addresses

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

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

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

## exercise: page table size

suppose 32-bit virtual (program) addresses

suppose 30-bit physical (hardware) addresses

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

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

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

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

issue: where can we store that?

## exercise: address splitting

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

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

## exercise: address splitting

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

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

page #: 0x12345; offset: 0x678



# fast copies

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

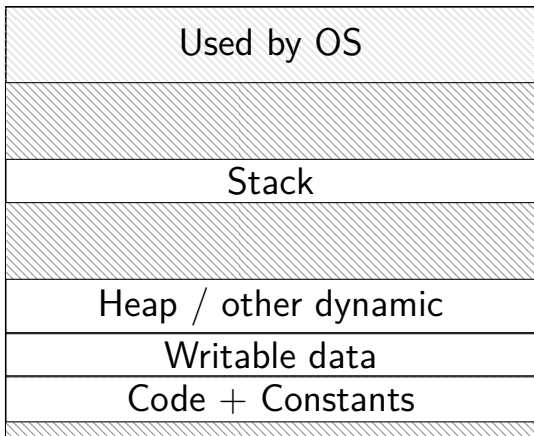
creates a **copy** of an entire program!

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

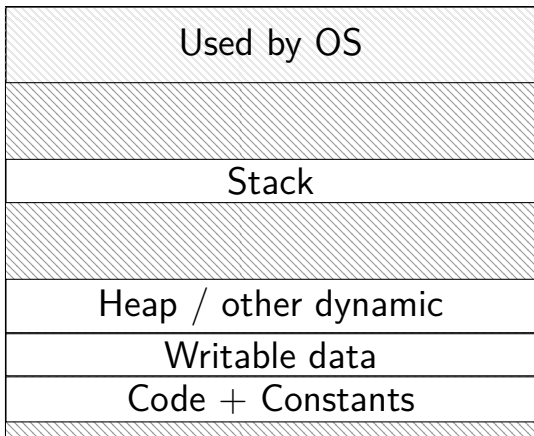
how isn't this really slow?

# do we really need a complete copy?

bash

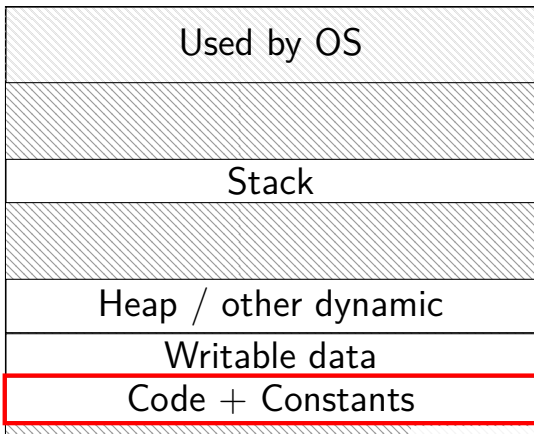


new copy of bash

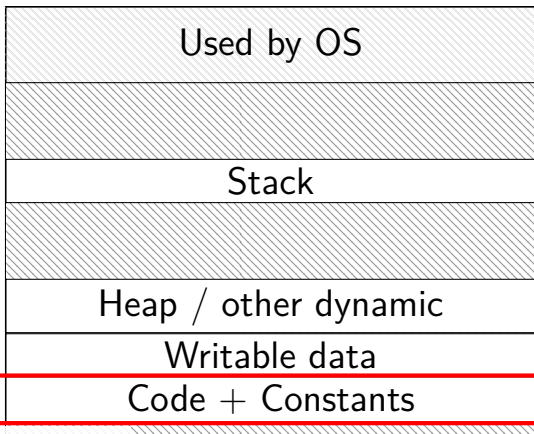


# do we really need a complete copy?

bash



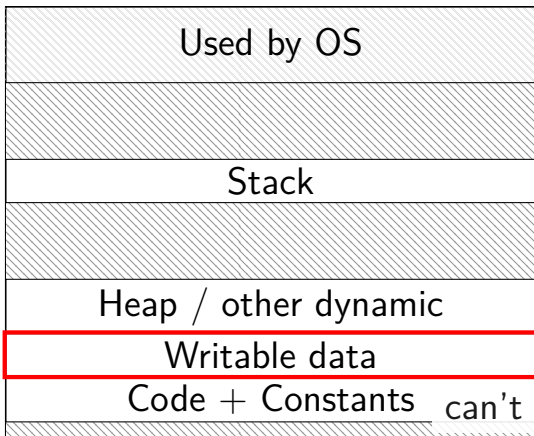
new copy of bash



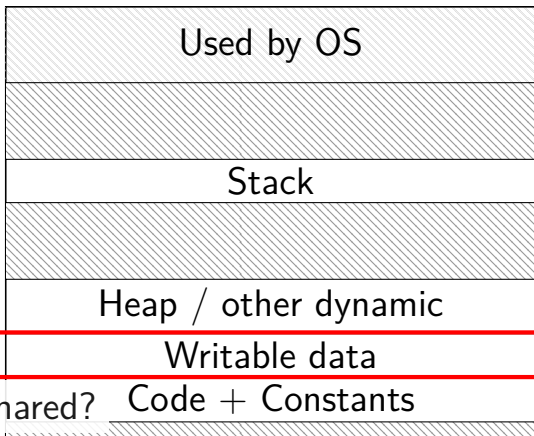
shared as read-only

# do we really need a complete copy?

bash



new copy of bash



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

# 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

hardware help: can help OS know if pages were used  
typical solution: accessed bit (similar to dirty bit) in page table entries

# efficient shared libraries

recall: linking

- placeholders to fill in with addresses

- need to change depending on **where library ends up?**

different code depending on where library ends up?

one solution: **position-independent code**