



# last time

virtual addresses  $\rightarrow$  physical addresses

dividing memory into  $2^X$  byte pages

addresses split into page number and  $X$  byte offset

table:

- row number = virtual page number

- data in row = valid bit, physical page number

virtual to physical translation with table

- lookup virtual page number to get physical

- check valid bit

- keep page offset the same

table sizes

# schedule changes

# 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	1 D2 D3
0x24-7	5 D6 D7
0x28-B	A AB BC
0x2C-F	E EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

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)  $0x18 = ?$ ;  $0x03 = ???$ ;  $0x0A = ???$ ;  $0x13 = ???$

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 = ; 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 = ; 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 = ; 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

# page tables in memory

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

page table entry layout (chosen by processor)

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

# page tables in memory

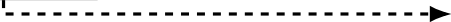
where can processor store megabytes of page tables? **in memory**

page table entry layout (chosen by processor)

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

page table  
base register

0x00010000

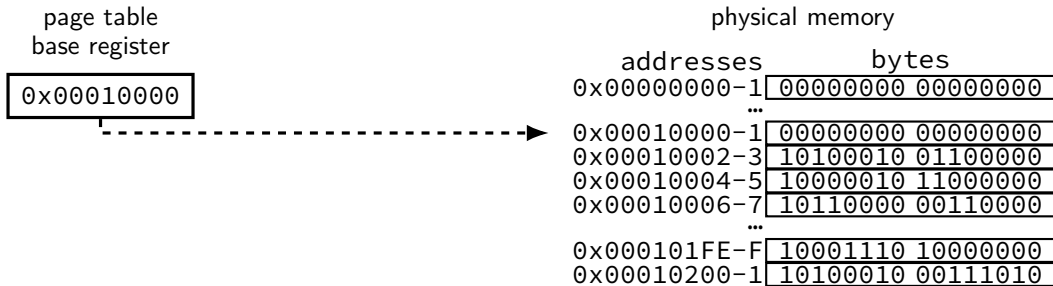


# page tables in memory

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

page table entry layout (chosen by processor)

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

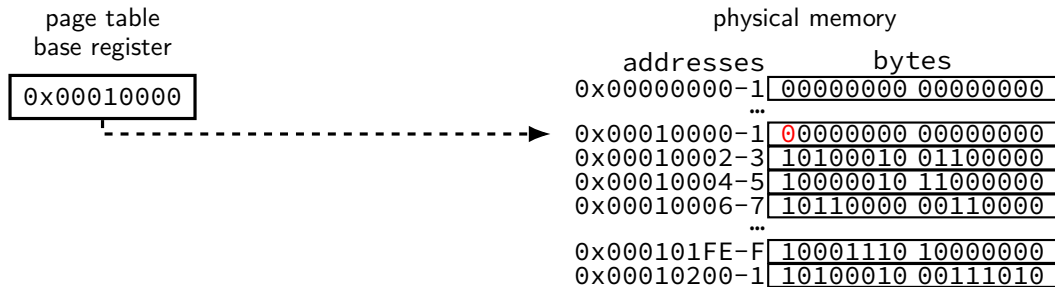


# page tables in memory

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

page table entry layout (chosen by processor)

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

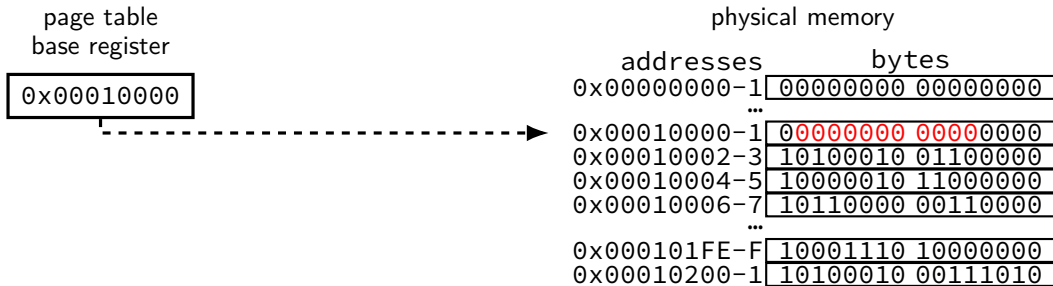


# page tables in memory

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

page table entry layout (chosen by processor)

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

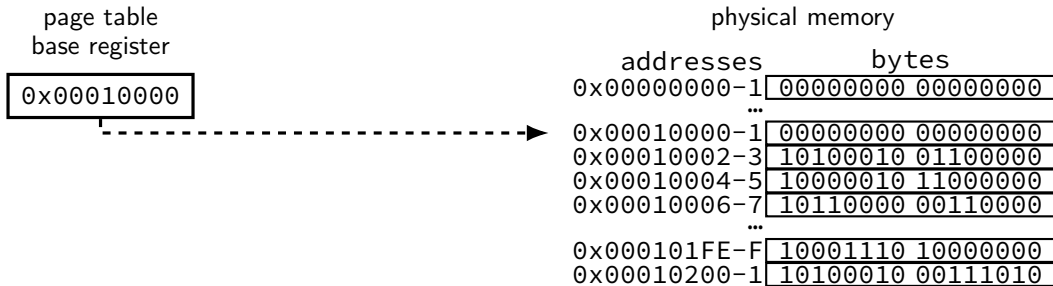


# page tables in memory

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

page table entry layout (chosen by processor)

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



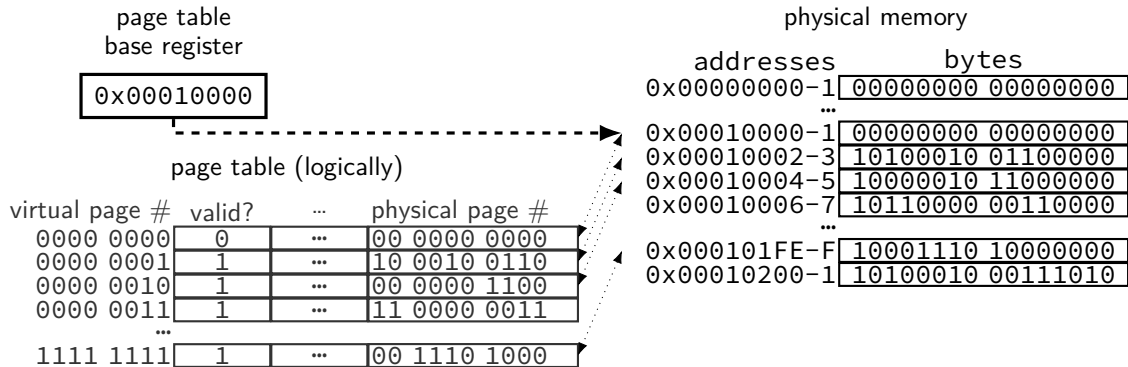


# page tables in memory

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

page table entry layout (chosen by processor)

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

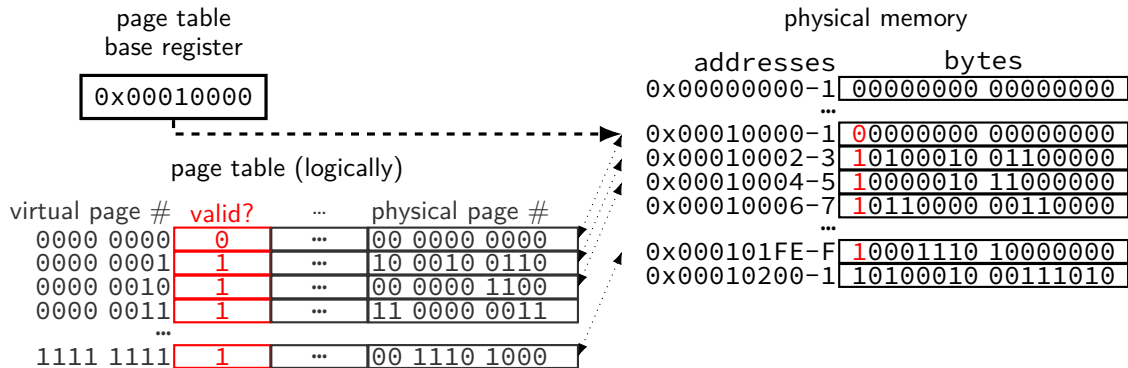


# page tables in memory

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

page table entry layout (chosen by processor)

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

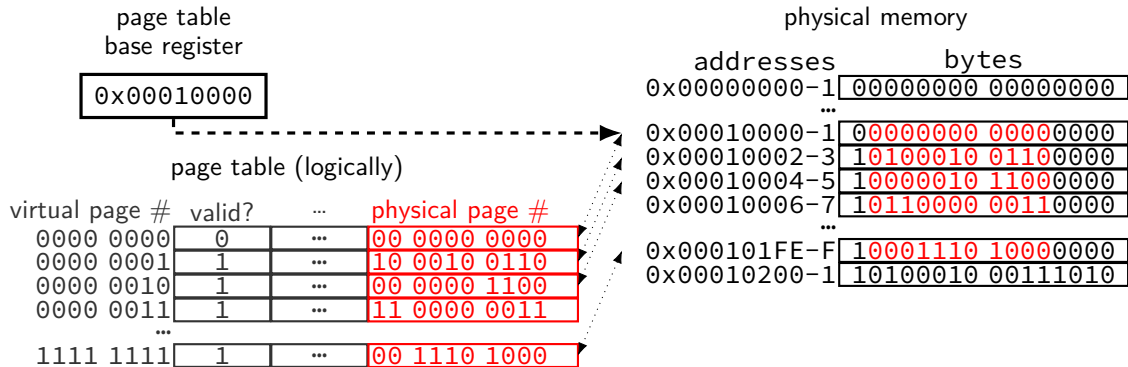


# page tables in memory

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

page table entry layout (chosen by processor)

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

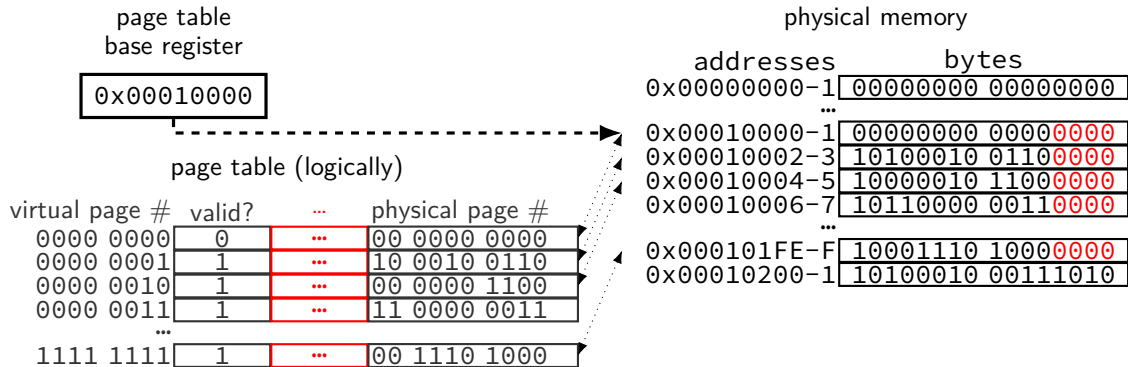


# page tables in memory

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

page table entry layout (chosen by processor)

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

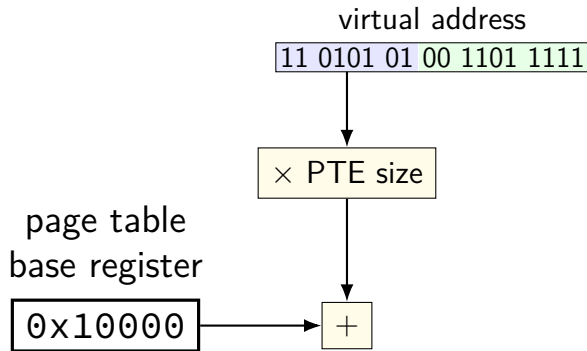


# memory access with page table

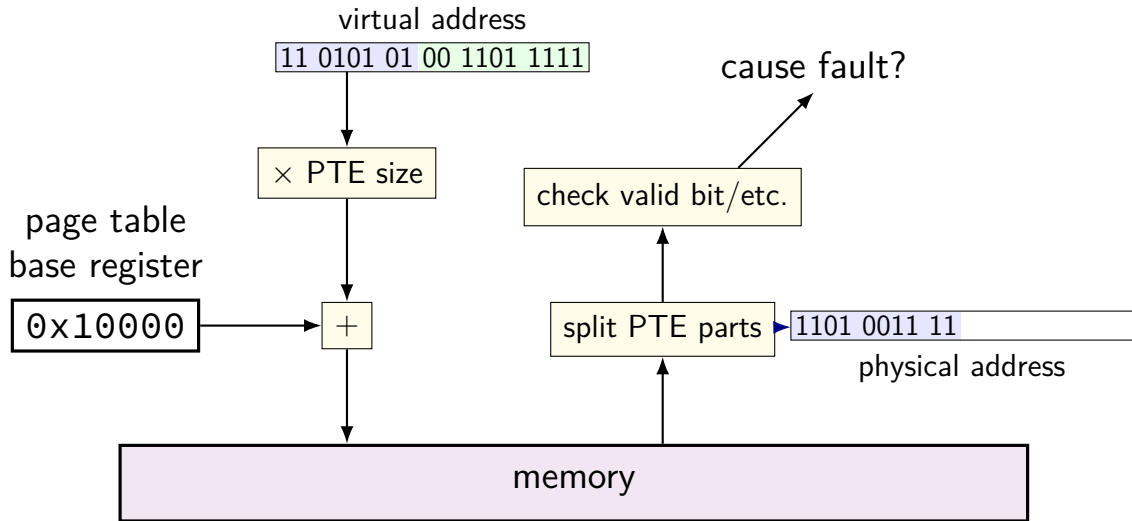
virtual address

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

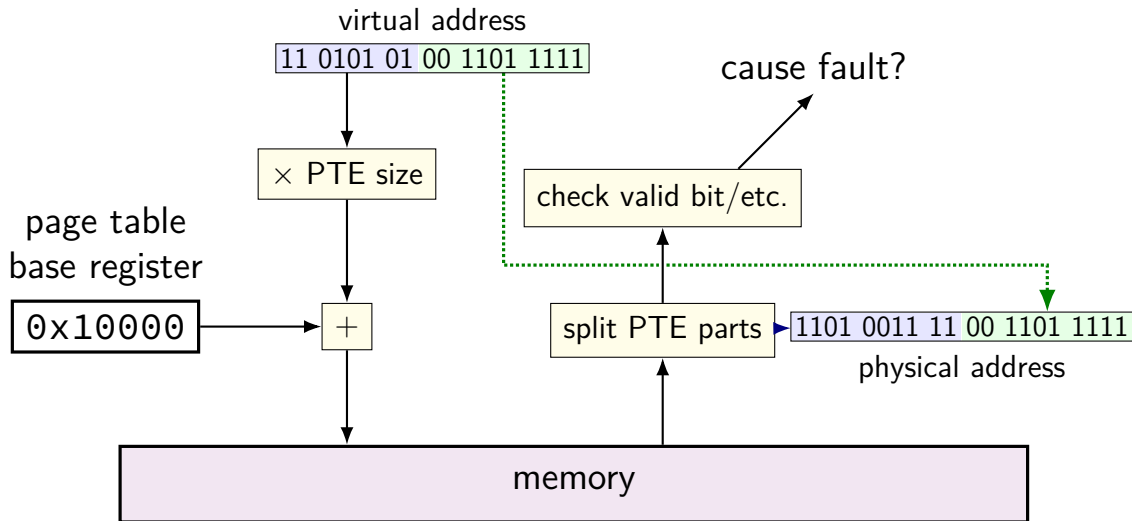
# memory access with page table



# memory access with page table

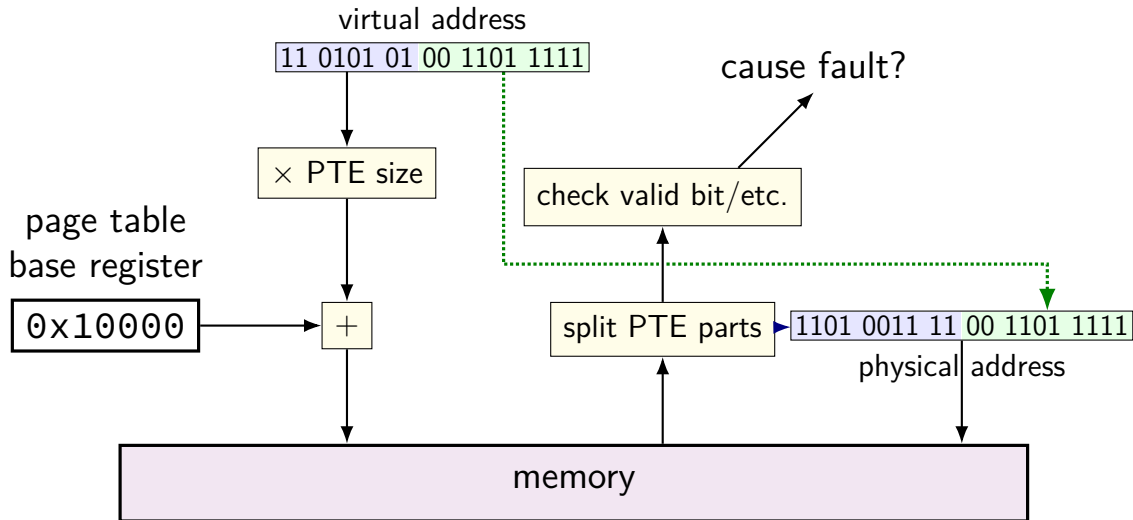


# memory access with page table

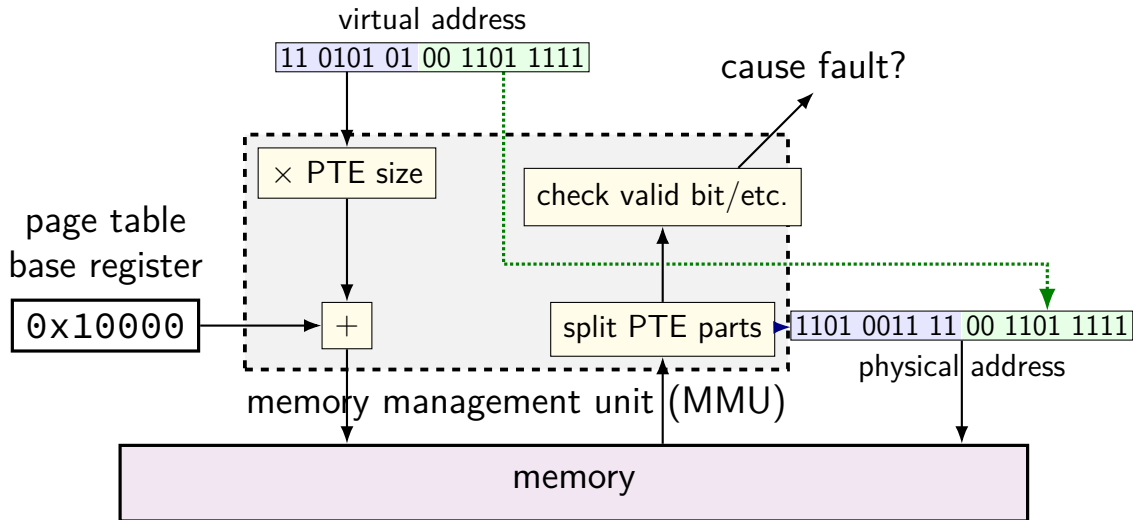




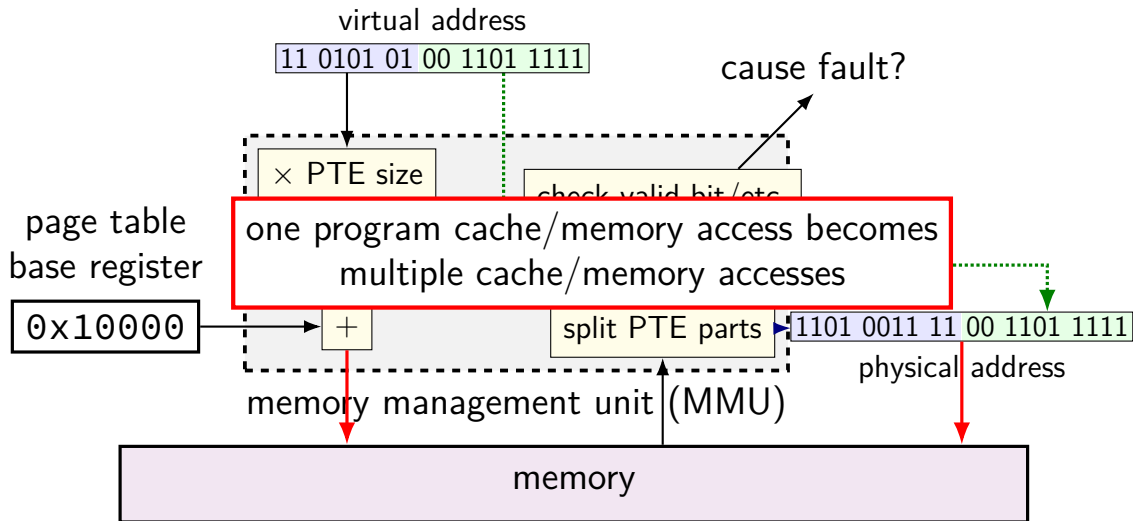
# memory access with page table



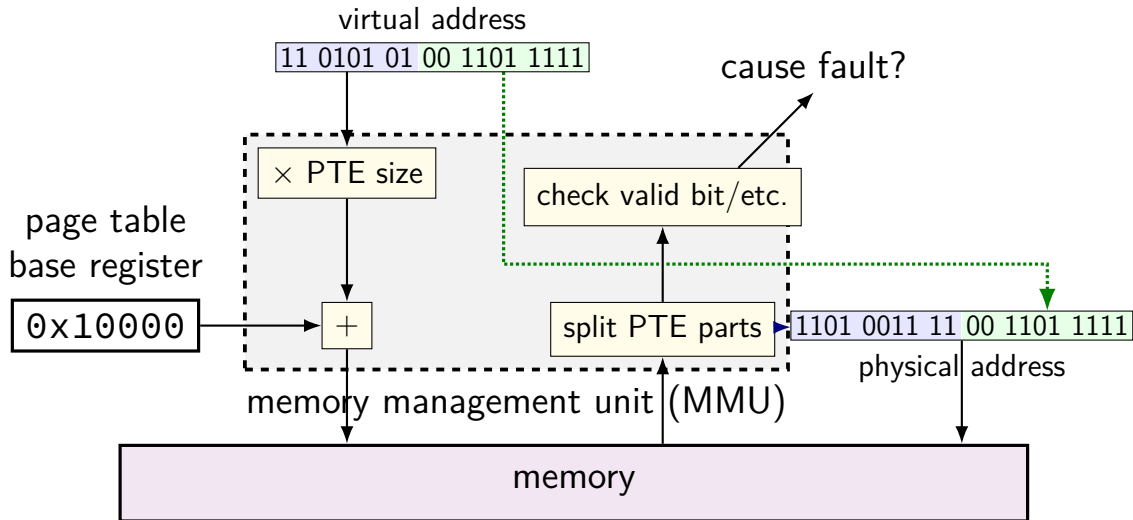
# memory access with page table



# memory access with page table



# memory access with page table



# 1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;  
page table base register 0x20; translate virtual address 0x31

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

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

# 1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;  
page table base register 0x20; translate virtual address 0x31

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

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x31 = 11 0001

*PTE addr:*

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

*PTE value:*

0xF6 = 1111 0110

PPN 111, valid 1

$M[111\ 001] = M[0x39]$

$\rightarrow 0x0C$

# 1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;  
page table base register 0x20; translate virtual address 0x31

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

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x31 = 11 0001

*PTE addr:*

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

*PTE value:*

0xF6 = 1111 0110

PPN 111, valid 1

$M[111\ 001] = M[0x39]$

$\rightarrow 0x0C$

# 1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;  
page table base register 0x20; translate virtual address 0x31

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

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x31 = 11 0**001**

*PTE addr:*

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

*PTE value:*

0xF6 = 1111 0110

PPN 111, valid 1

$M[111 \text{ } 001] = M[0x39]$

$\rightarrow 0x0C$



# 1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other  
page table base register 0x20; translate virtual address 0x12

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

physical addresses	bytes
0x20-3	A0 D1 E2 F3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

# 1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other  
page table base register 0x20; translate virtual address 0x12

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

physical addresses	bytes
0x20-3	A0 D1 E2 F3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x12 = 01 0010

*PTE addr:*

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

*PTE value:*

0xD1 = 1101 0001

PPN 110, valid 1

$M[110 \ 001] = M[0x32]$

→ 0xBA

# 1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other  
page table base register 0x20; translate virtual address 0x12

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

physical addresses	bytes
0x20-3	A0 D1 E2 F3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x12 = 01 0010

*PTE addr:*

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

*PTE value:*

0xD1 = 1101 0001

PPN 110, valid 1

$M[110\ 001] = M[0x32]$

→ 0xBA

# 1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other  
page table base register 0x20; translate virtual address 0x12

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

physical addresses	bytes
0x20-3	A0 D1 E2 F3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x12 = 01 0**010**

*PTE addr:*

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

*PTE value:*

0xD1 = 1101 0001

PPN 110, valid 1

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

→ 0xBA

# 1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other  
page table base register 0x20; translate virtual address 0x12

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

physical addresses	bytes
0x20-3	A0 D1 E2 F3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x12 = 01 0010

*PTE addr:*

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

*PTE value:*

0xD1 = 1101 0001

PPN 110, valid 1

$M[110 \ 001] = M[0x32]$

→ 0xBA

# pagetable assignment

pagetable assignment

simulate page tables (on top of normal program memory)

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

in assignment:

virtual address  $\sim$  arguments to your functions

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

# pagetable assignment API

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

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

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

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

# translate()

with POBITS=12, LEVELS=1:

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

ptbr = GetPointerToTable(

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



# translate()

with POBITS=12, LEVELS=1:

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

ptbr = GetPointerToTable(

translate(0x0FFF) == (void\*) ~0L

translate(0x1000) == (void\*) 0x9999000

translate(0x1001) == (void\*) 0x9999001

translate(0x2000) == (void\*) ~0L

translate(0x2001) == (void\*) ~0L

translate(0x3000) == (void\*) 0x3333000

# page\_allocate()

with POBITS=12, LEVELS=1:

ptbr == 0

page\_allocate(0x1000) *or* page\_allocate(0x1001) *or* ...

# page\_allocate()

with POBITS=12, LEVELS=1:

ptbr == 0

page\_allocate(0x1000) or page\_allocate(0x1001) or ...

ptbr *now* == GetPointerToTable(

VPN valid? physical

0	0	—
1	1	(new)
2	0	—
3	1	—
...	...	...

)

allocated with posix\_memalign

# page\_allocate()

with POBITS=**12**, LEVELS=1:

ptbr == 0

page\_allocate(0x1**000**) or page\_allocate(0x1**001**) or ...

ptbr *now* == GetPointerToTable(

VPN valid? physical

0	0	—
1	1	(new)
2	0	—
3	1	—
...	...	...

)

allocated with posix\_memalign

## posix\_memalign

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

allocate size bytes

choosing address that is multiple of alignment  
can make sure allocation starts at beginning of page

error\_code indicates if out-of-memory, etc.

fills in result (passed via pointer)

# posix\_memalign

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

allocate size bytes

choosing address that is multiple of **alignment**  
can make sure allocation starts at beginning of page

error\_code indicates if out-of-memory, etc.

fills in result (passed via pointer)

# posix\_memalign

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

allocate size bytes

choosing address that is multiple of alignment  
can make sure allocation starts at beginning of page

error\_code indicates if out-of-memory, etc.

fills in **result** (passed via pointer)

# parts

part 1 (next week): LEVELS=1, POBITS=12 and  
translate() OR  
page\_allocate()

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

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



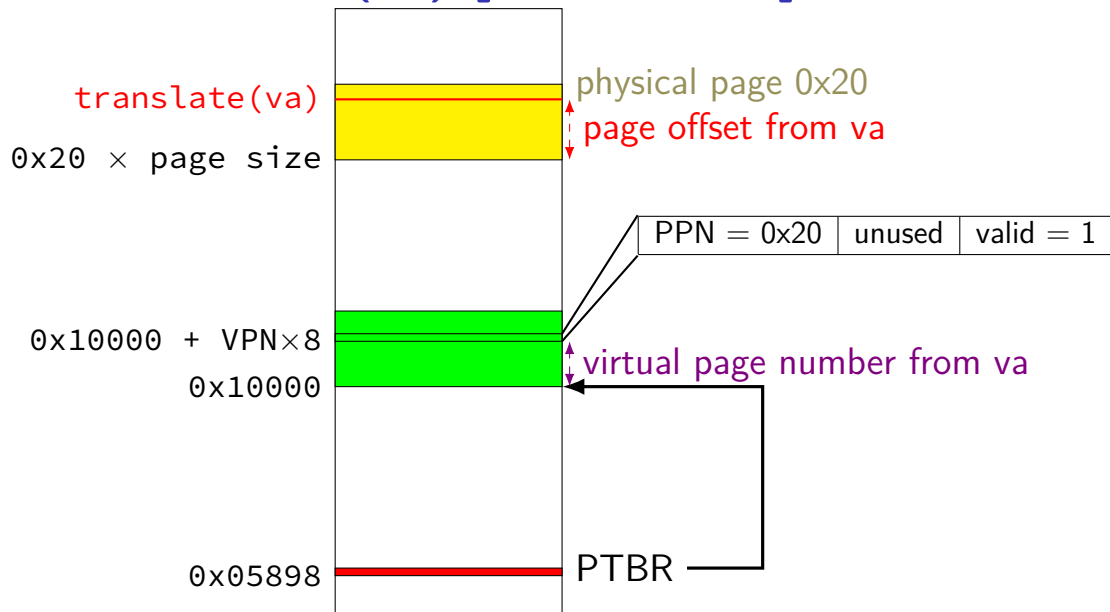
# address/page table entry format

(with POBITS=12, LEVELS=1)

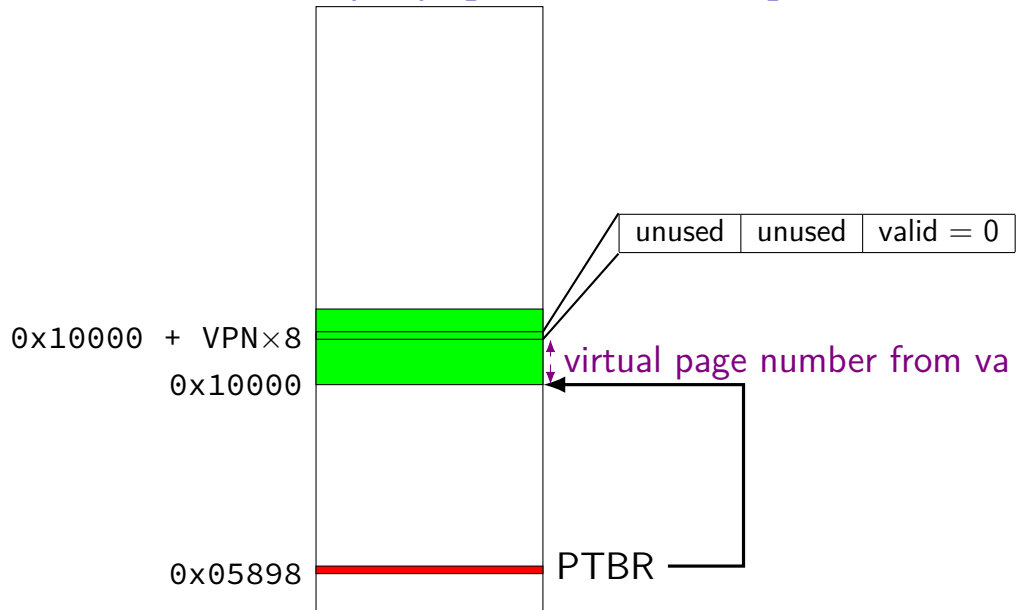
	bits 63–21	bits 20–12	bits 11–1	bit 0
page table entry	physical page number		unused	valid bit
virtual address	unused	virtual page number	page offset	
physical address	physical page number		page offset	

in assignment: value from `posix_memalign` = physical address

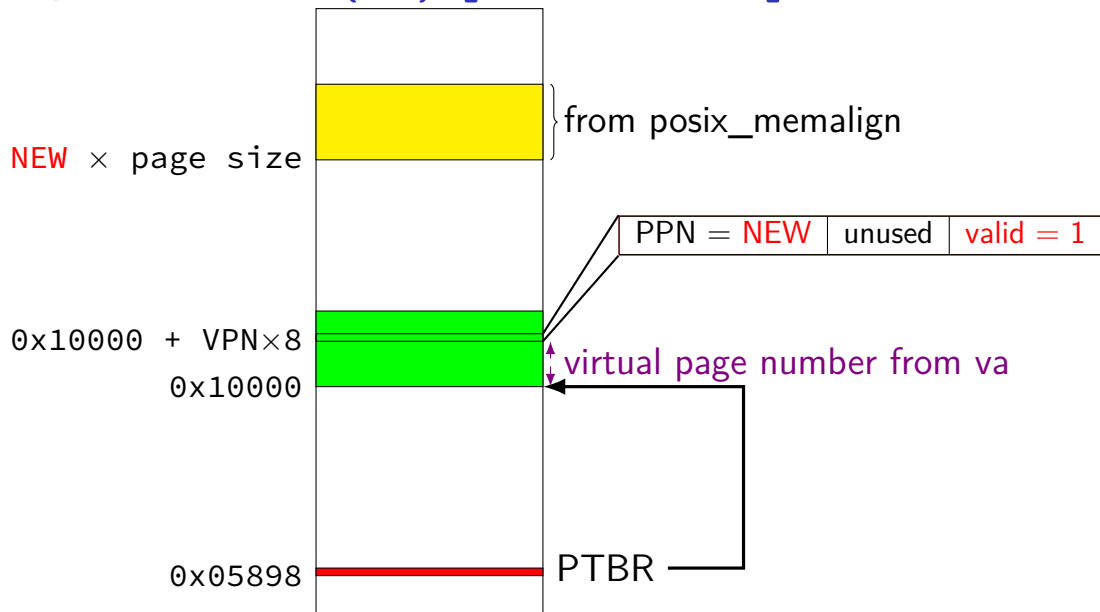
**pa = translate(va) [LEVELS=1]**



# page\_allocate(va) [LEVELS=1]



# page\_allocate(va) [LEVELS=1]



## exercise: 64-bit system

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

4096 byte pages

## exercise: 64-bit system

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

4096 byte pages



top 16 bits of 64-bit addresses not used for translation

## exercise: 64-bit system

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

4096 byte pages

exercise: how many page table entries? (assuming page table like shown before)

exercise: how large are physical page numbers?

## exercise: 64-bit system

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

4096 byte pages

exercise: how many page table entries? (assuming page table like shown before)

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)

exercise: how large are physical page numbers?

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

trick: power of two size makes table lookup faster

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

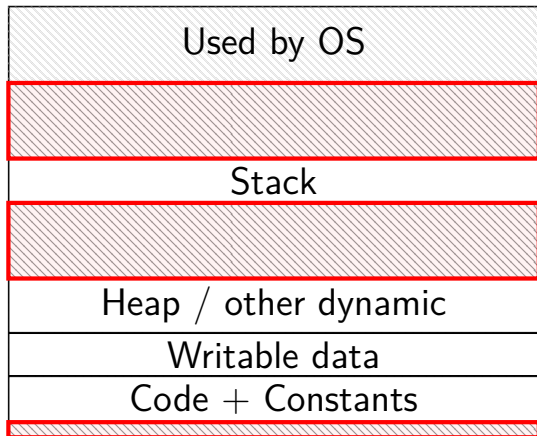
# huge page tables

huge virtual address spaces!

impossible to store PTE for every page

how can we save space?

# holes



most pages are **invalid**

## saving space

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

use a data structure other than a flat array

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

options?

# saving space

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

use a data structure other than a flat array

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

options?

## hashtable

actually used by some historical processors

but never common

# saving space

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

use a data structure other than a flat array

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

options?

hashtable

actually used by some historical processors

but never common

tree data structure

but not quite a search tree

# search tree tradeoffs

lookup usually implemented in hardware

- lookup should be simple

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

lookup should not involve many memory accesses

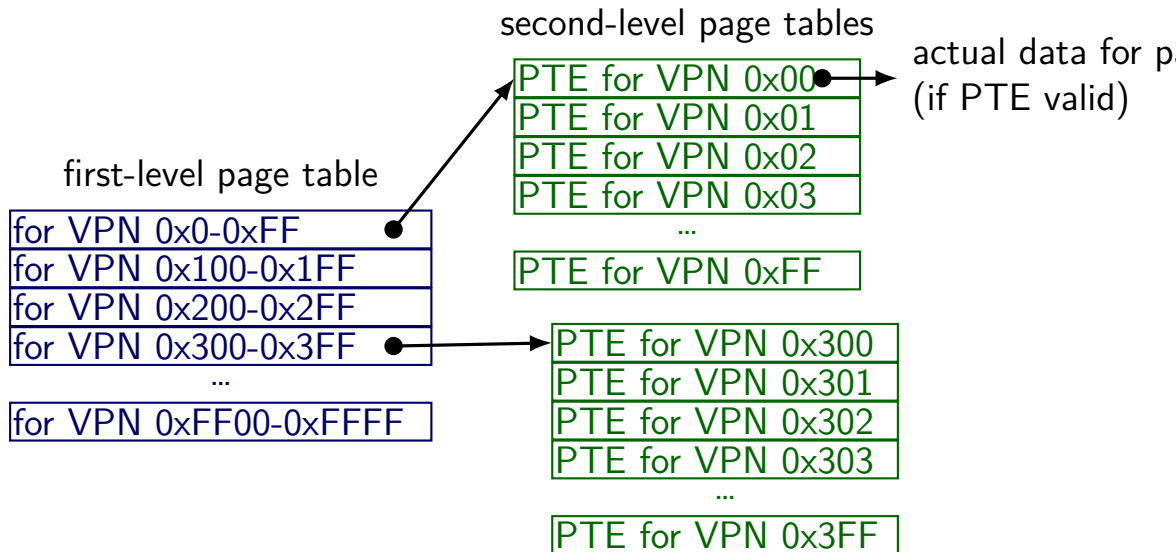
- doing two memory accesses is already very slow

- solution: tree with many children from each node

- (far from binary tree's left/right child)

# two-level page tables

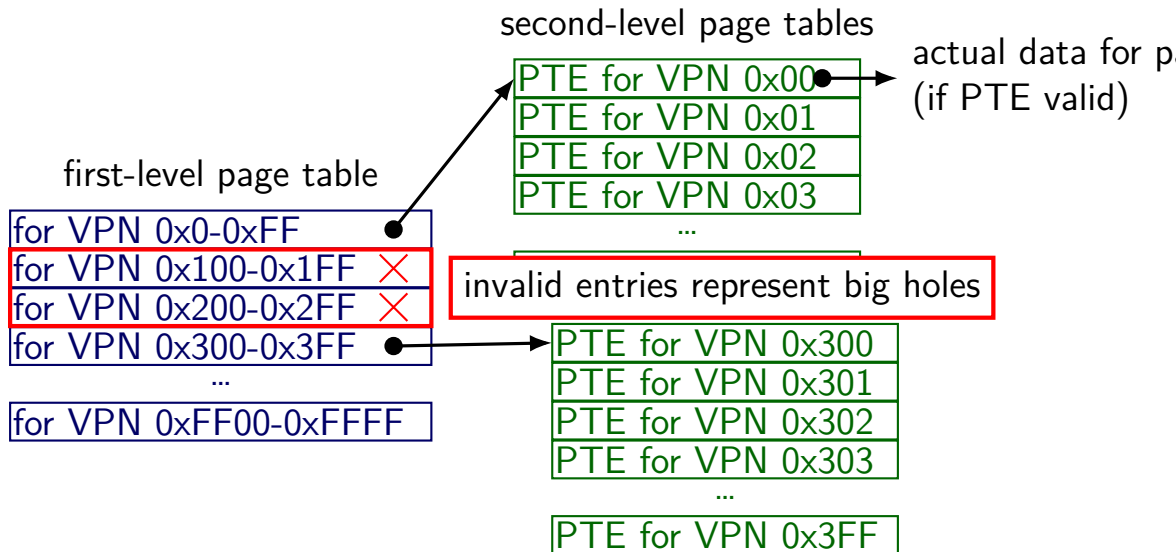
two-level page tables for 65536 pages (16-bit VPN; 256 entries/table)





# two-level page tables

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



# two-level page tables

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

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

first-level page table for VPN 0x000-0x00FF

first-level page table for VPN 0x100-0x10FF

first-level page table for VPN 0x200-0x20FF

first-level page table for VPN 0x300-0x30FF

...

first-level page table for VPN 0xFF00-0xFFFF

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

first-level page table for VPN 0x00-0xFF	PTE for VPN 0x303
for VPN 0x100-0x1FF	...
for VPN 0x200-0x2FF	PTE for VPN 0x3FF
for VPN 0x300-0x3FF	
...	
for VPN 0xFF00-0xFFFF	

# two-level page tables

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

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

first-level page table for VPN 0x000-0x0FF

first-level page table for VPN 0x100-0x1FF

first-level page table for VPN 0x200-0x2FF

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

...

first-level page table for VPN 0xFF00-0xFFFF

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 0x300-0x3FF
...
for VPN 0xFF00-0xFFFF

a second-level page table

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

PTE for VPN 0x303

...

PTE for VPN 0x3FF

or p  
l)

# 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 0x300-0x3FF	
...	
for VPN 0xFF00-0xFFFF	

a second-level page table

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

PTE for VPN 0x303

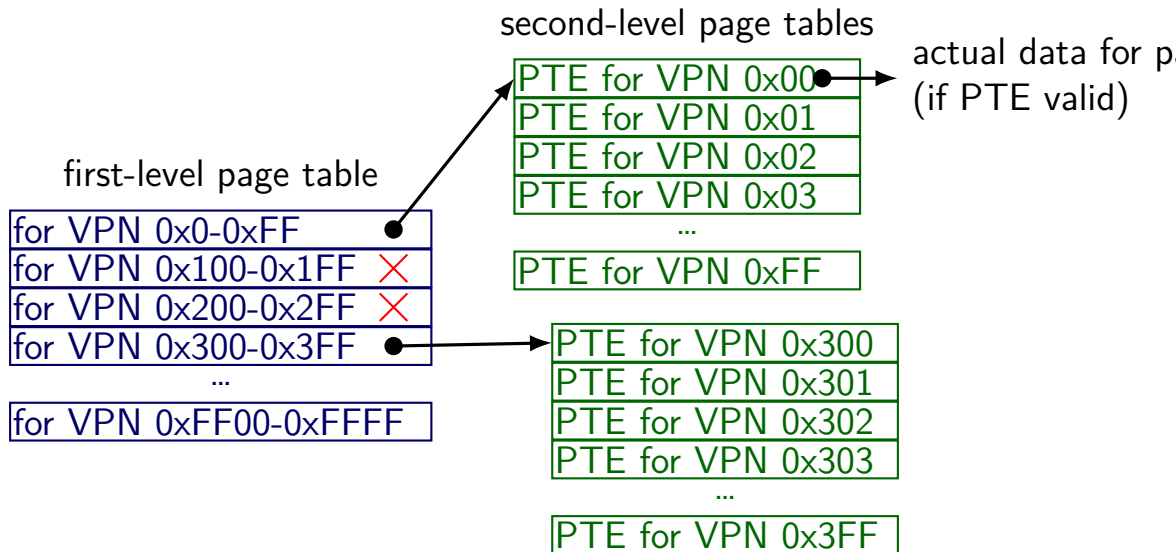
...

PTE for VPN 0x3FF

or p  
l)

# two-level page tables

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



# two-level page table lookup

virtual address

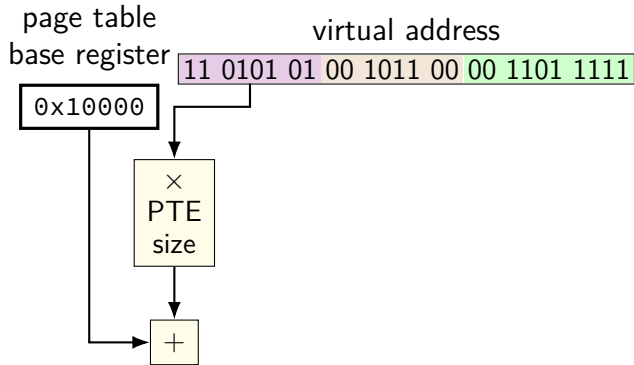
11	0101	01	00	1011	00	00	1101	1111
----	------	----	----	------	----	----	------	------

VPN — split into two parts (one per level)

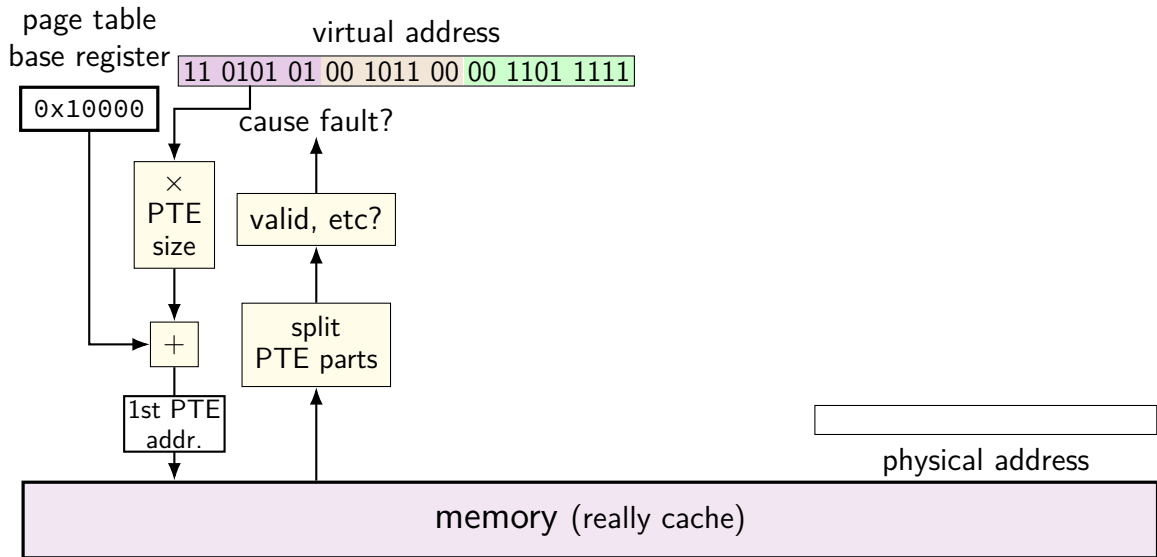
this example: parts equal sized — common, but not required



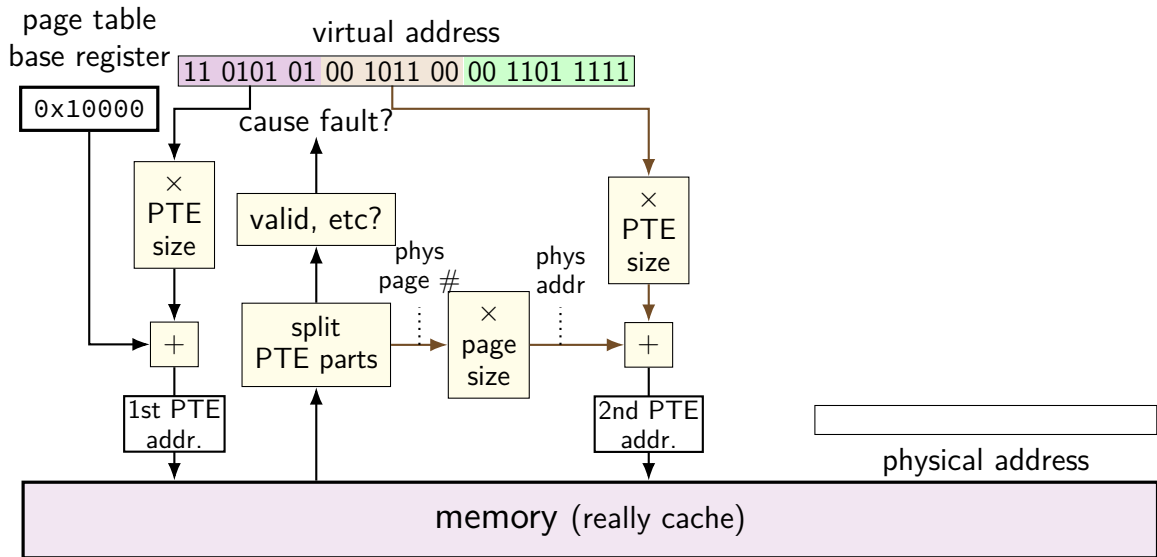
# two-level page table lookup



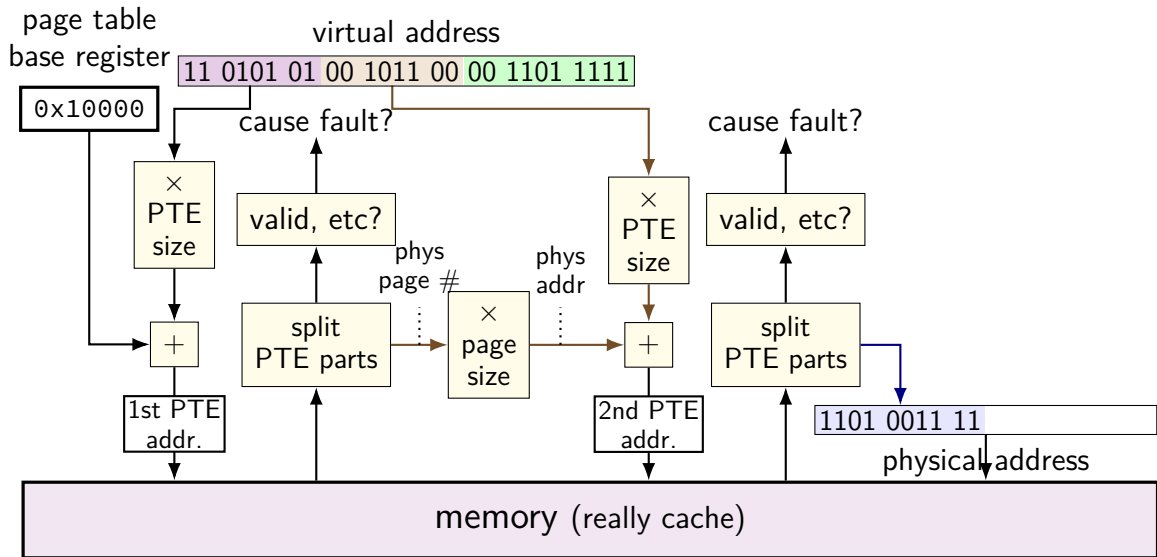
# two-level page table lookup



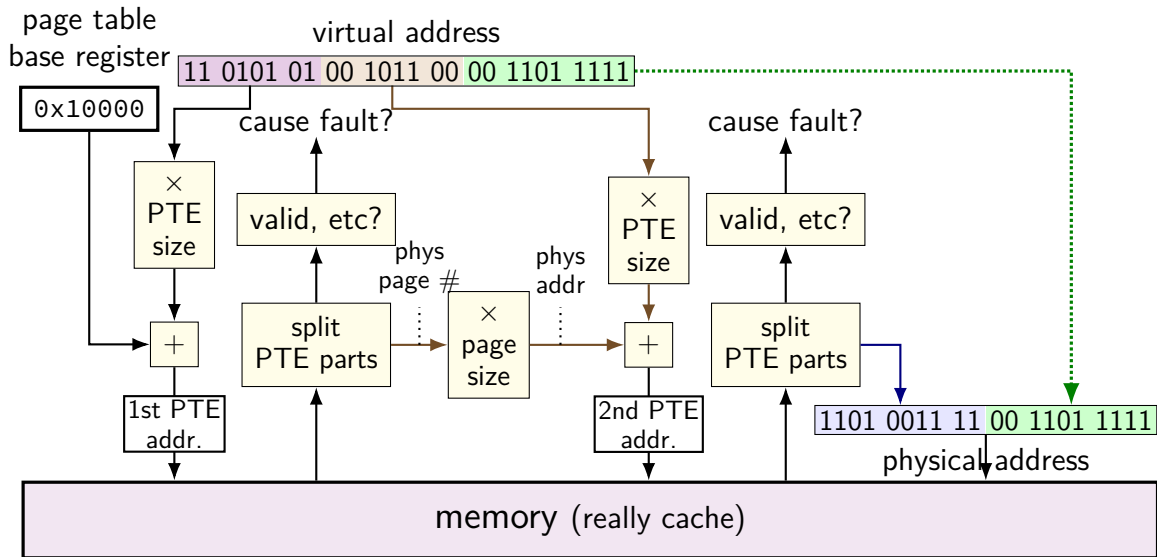
# two-level page table lookup



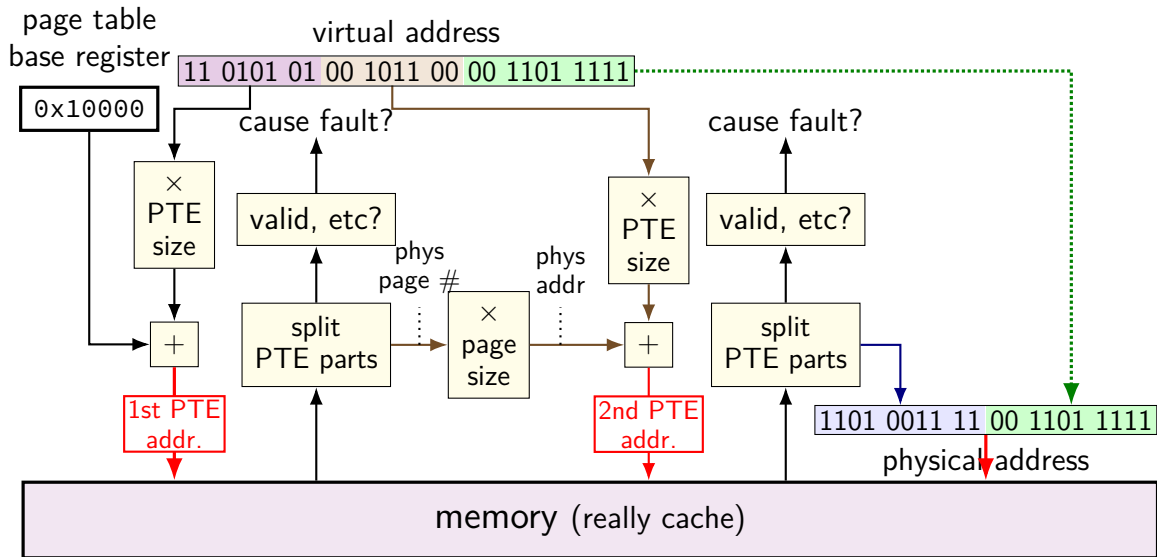
# two-level page table lookup



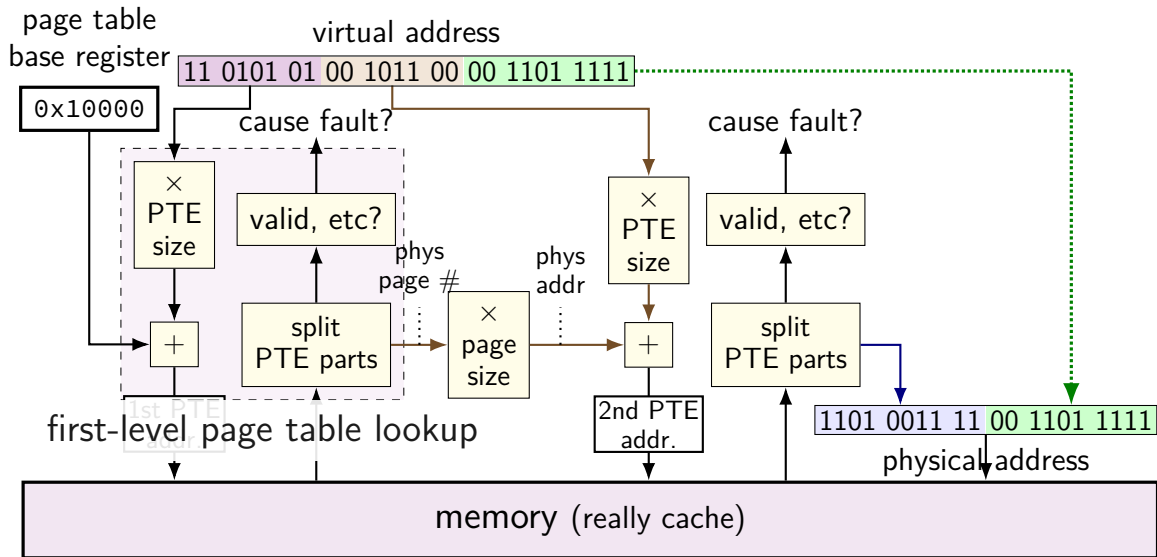
# two-level page table lookup



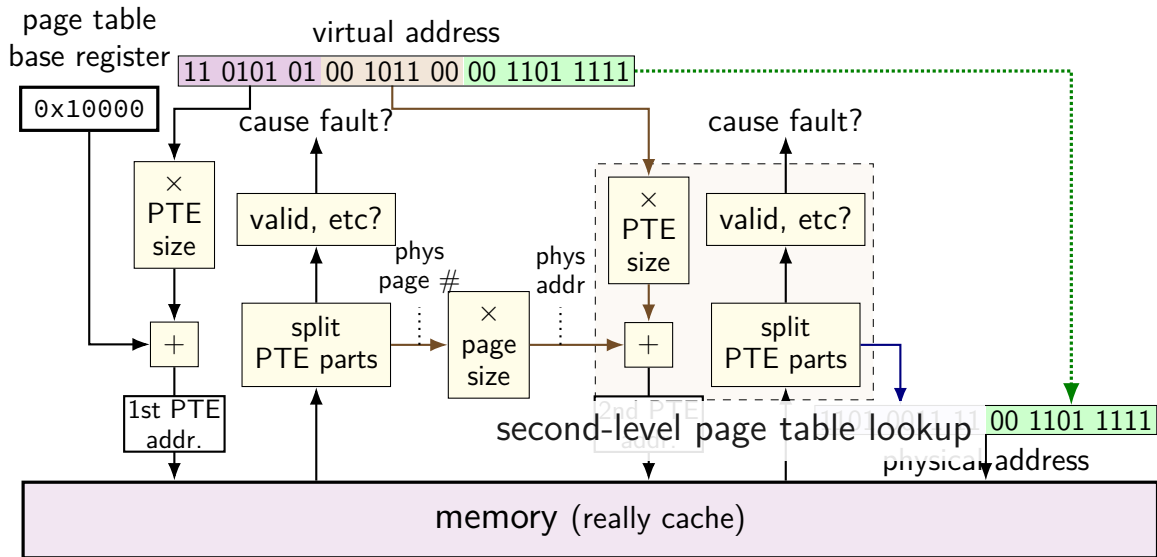
## two-level page table lookup



# two-level page table lookup

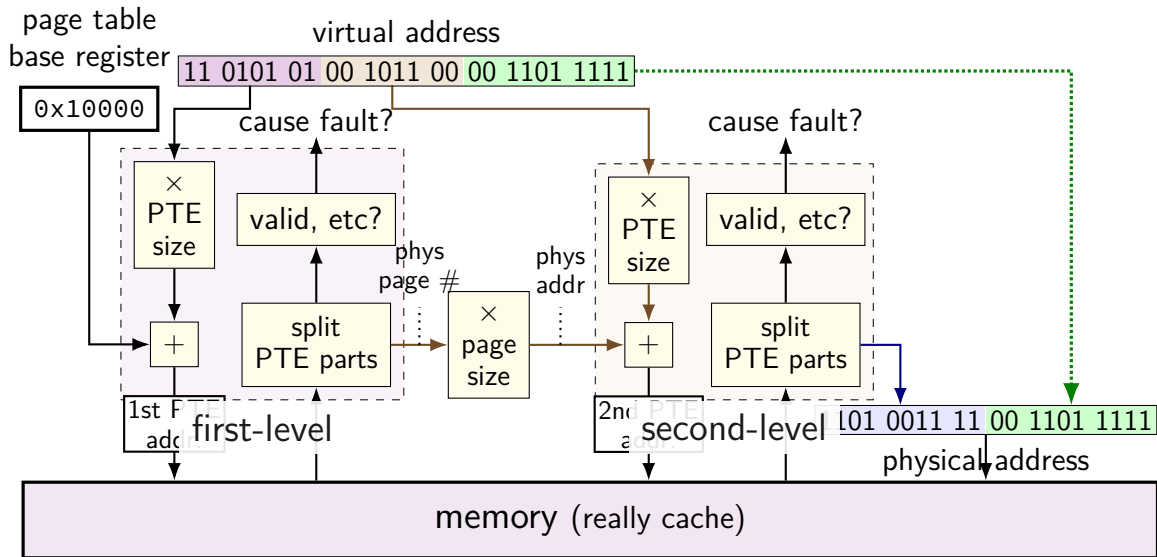


# two-level page table lookup

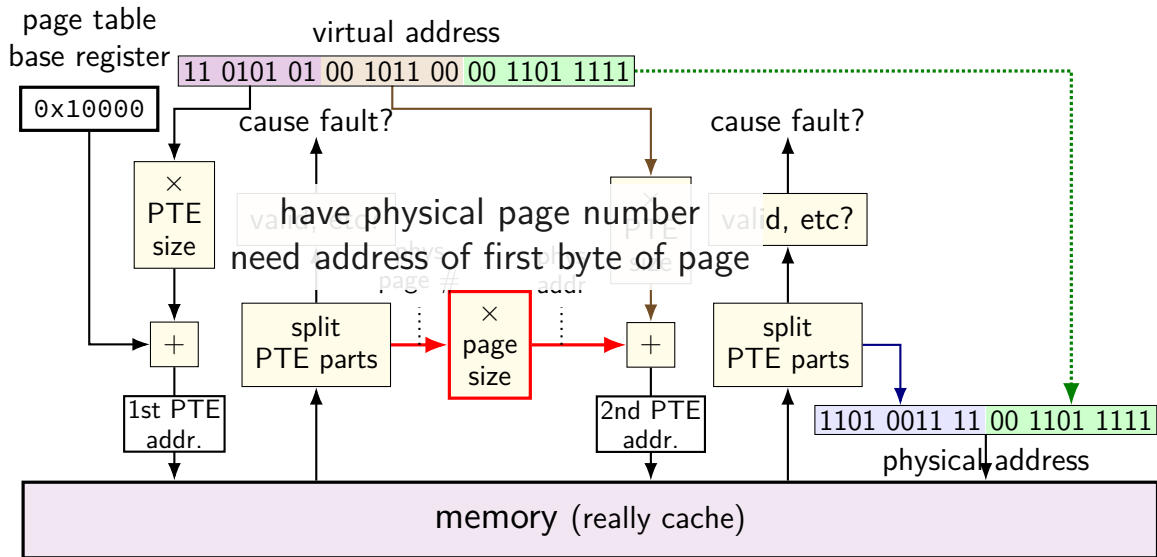




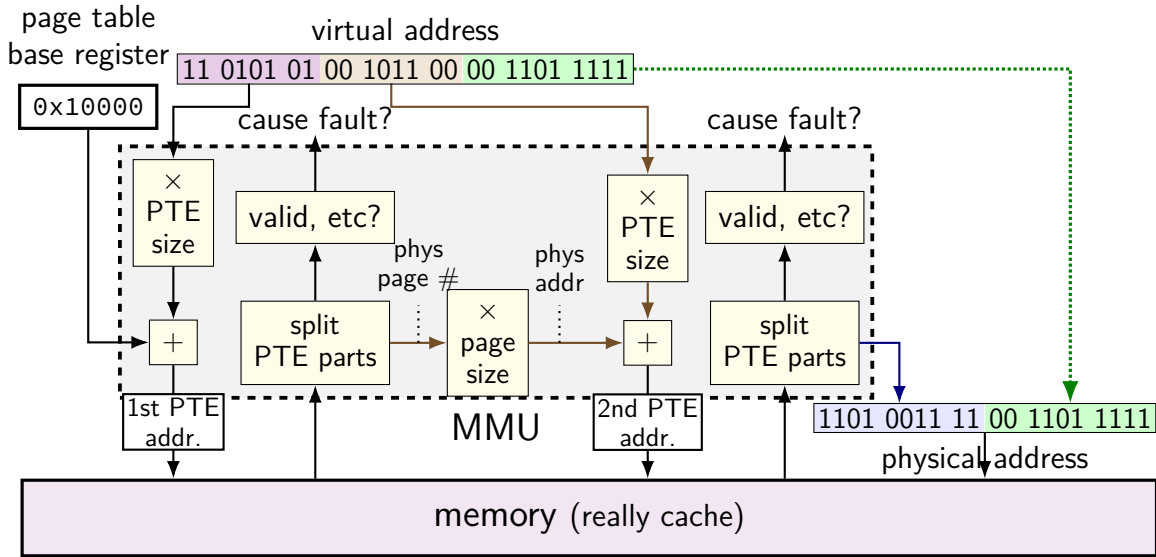
# two-level page table lookup



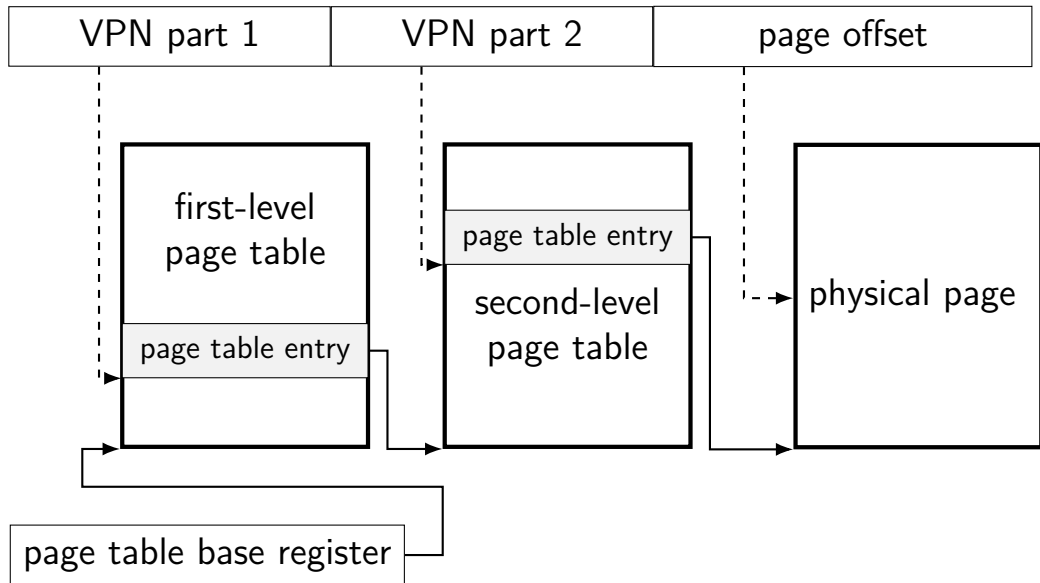
# two-level page table lookup



# two-level page table lookup



## another view



# multi-level page tables

VPN split into pieces for each level of page table

top levels: page table entries point to next page table

usually using physical page number of next page table

bottom level: page table entry points to destination page

validity 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

indexes used for lookup **parts of the virtual page number**  
(there are not multiple VPNs)

# assignment



## 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	00 91 72 13
0x24-7	D4 F5 36 07
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	00 91 72 13
0x24-7	D4 F5 36 07
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	00 91 72 13
0x24-7	D4 F5 36 07
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	00 91 72 13
0x24-7	D4 F5 36 07
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	00 91 72 13
0x24-7	D4 F5 36 07
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	00 91 72 13
0x24-7	D4 F5 36 07
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 splitting

9-bit virtual address

6-bit physical address

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

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

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

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

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

## 2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;  
page table base register 0x08; translate virtual address 0x0FB

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

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



## 2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;  
page table base register 0x08; translate virtual address 0x0FB

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

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

## 2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;  
page table base register 0x08; translate virtual address 0x0FB

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

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

## 2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;  
page table base register 0x08; translate virtual address 0x0FB

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

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

## 2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;  
page table base register 0x08; translate virtual address 0x0FB

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

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

## 2-level exercise (2)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;  
page table base register 0x10; translate virtual address 0x109

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 5A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

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

## 2-level exercise (3)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x08; translate virtual address 0x00B

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

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

## 2-level exercise (3)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x08; translate virtual address 0x00B

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

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

## 2-level exercise (3)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x08; translate virtual address 0x00B

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

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



## 2-level exercise (4)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x08; translate virtual address 0x1CB

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

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

## 2-level exercise (5)

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

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

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

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

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

## 2-level exercise (5)

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

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

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

physical  
addresses bytes

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

physical  
addresses bytes

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

## 2-level exercise (5)

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

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

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

physical  
addresses bytes

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

physical  
addresses bytes

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

## 2-level exercise (5)

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

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

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

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

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

## 2-level exercise (5)

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

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

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

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

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

## 2-level exercise (5)

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

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

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

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

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

## 2-level exercise (5)

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

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

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

physical  
addresses bytes

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

physical  
addresses bytes

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



## 2-level exercise (5)

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

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

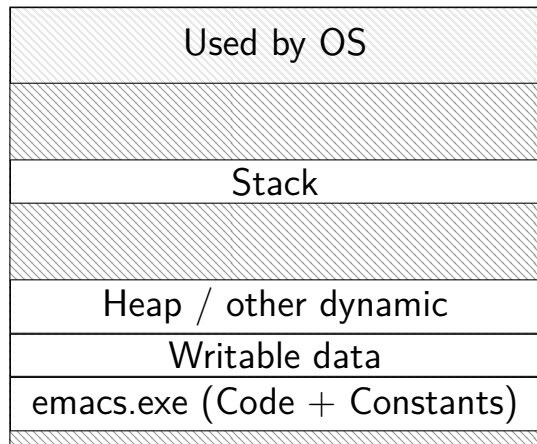
page table base register 0x10; translate virtual address 0x376

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

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

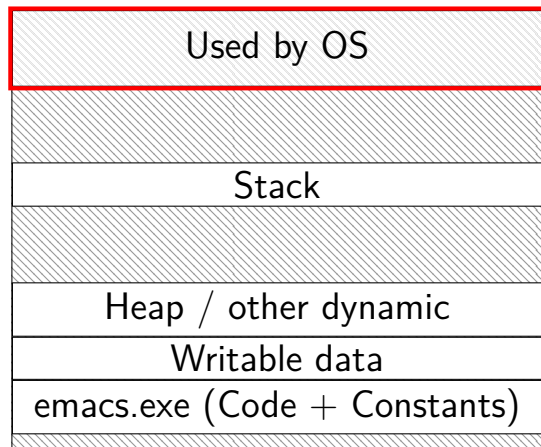
# emacs.exe

Emacs (run by user mst3k)



# emacs.exe

Emacs (run by user mst3k)



OS's memory

# switching page tables

part of context switch is changing the page table

extra privileged instructions

# switching page tables

part of context switch is changing the page table

extra privileged instructions

where in memory is the code that does this switching?

# switching page tables

part of context switch is changing the page table

extra **privileged instructions**

where in memory is the code that does this switching?

- probably have a page table entry pointing to it
- hopefully marked kernel-mode-only

# switching page tables

part of context switch is changing the page table

extra **privileged instructions**

where in memory is the code that does this switching?

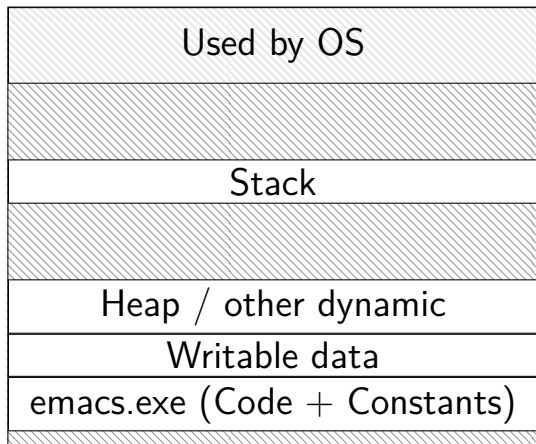
- probably have a page table entry pointing to it
- hopefully marked kernel-mode-only

code better not be modified by user program

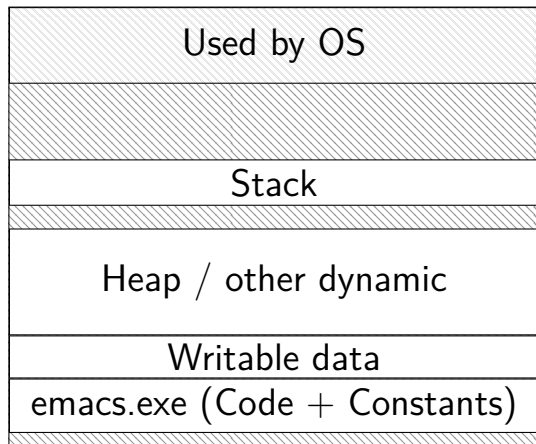
- otherwise: uncontrolled way to “escape” user mode

# emacs (two copies)

Emacs (run by user mst3k)



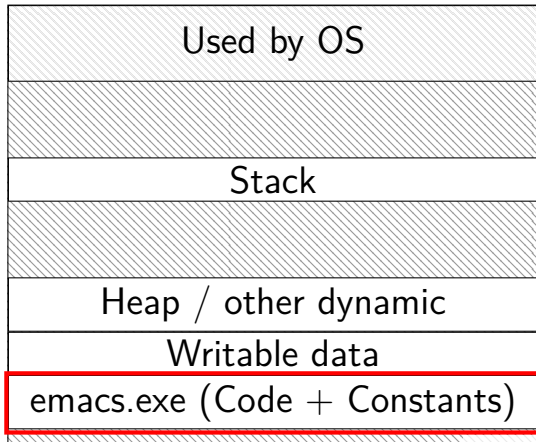
Emacs (run by user xyz4w)



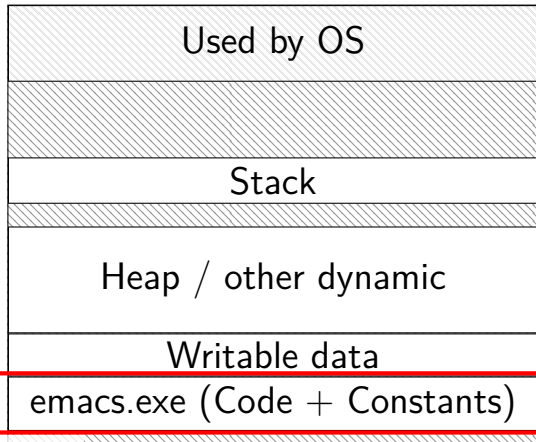


# emacs (two copies)

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”

# permissions bits

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

can access in user mode?

can read from?

can write to?

can execute from?

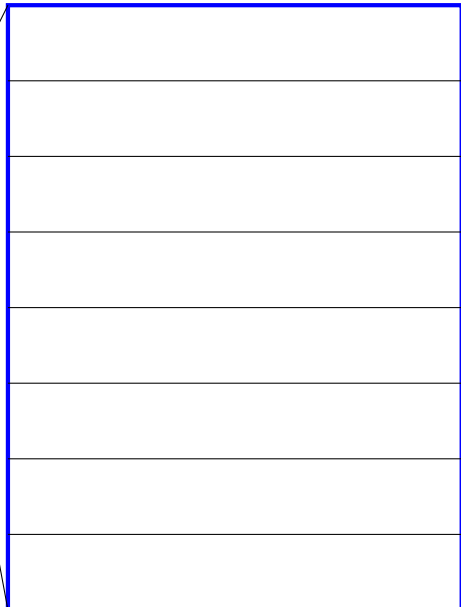
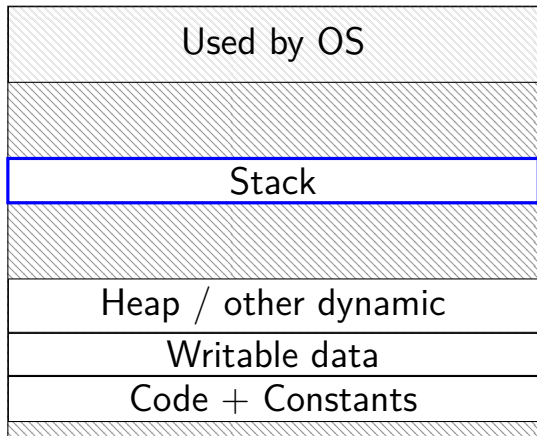
checked by MMU like valid bit

page table (logically)

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

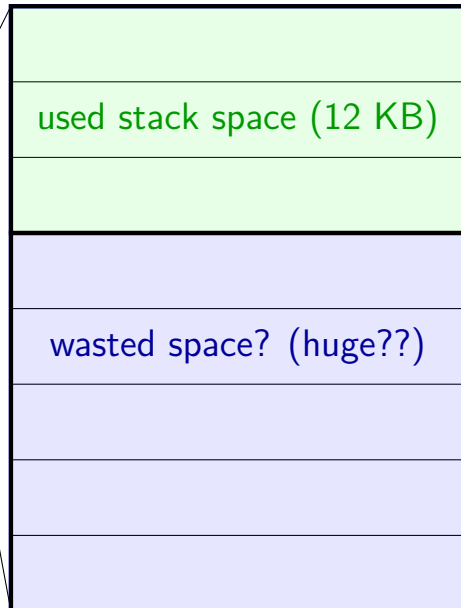
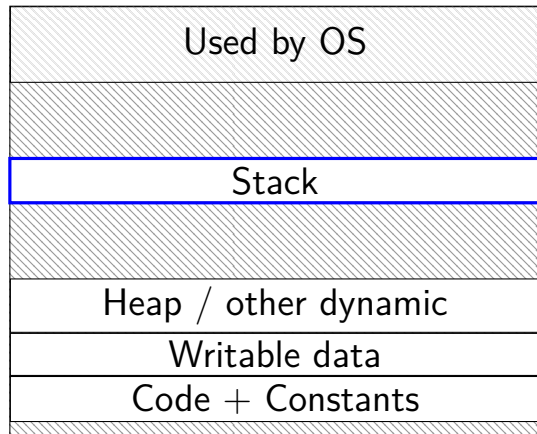
# space on demand

Program Memory



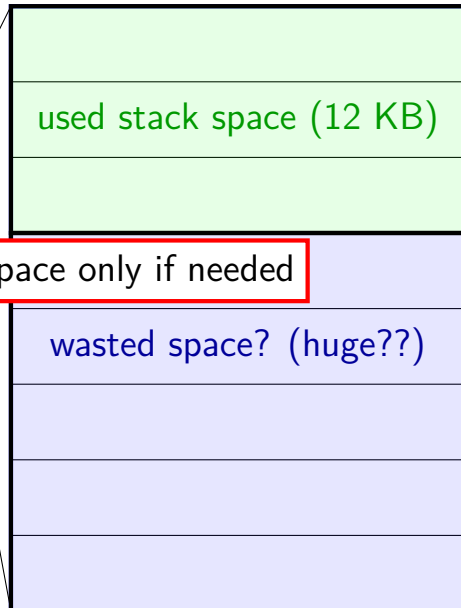
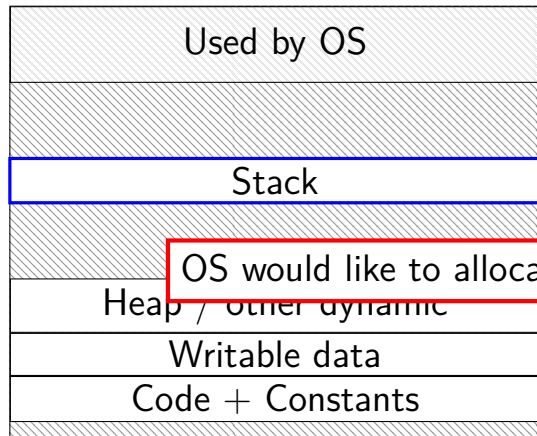
# space on demand

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  
   → page fault!  
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
...	...

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 restarted  
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

# 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

# mmap

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

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

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

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

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

# swapping almost mmap

access mapped file for first time, read from disk  
(like swapping when memory was swapped out)

write “mapped” memory, write to disk eventually  
(like writeback policy in swapping)  
use “dirty” bit

extra detail: other processes should see changes  
all accesses to file use **same physical memory**

# Linux maps: list of maps

```
$ cat /proc/self/maps
```

```
00400000-0040b000 r-xp 00000000 08:01 48328831 /bin/cat
0060a000-0060b000 r-p 0000a000 08:01 48328831 /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 48328831 /bin/cat
01974000-01995000 rw-p 00000000 00:00 0 [heap]
7f60c718b000-7f60c7490000 r-p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
7f60c7490000-7f60c764e000 r-xp 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.1
7f60c764e000-7f60c784e000 -p 001be000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.1
7f60c784e000-7f60c7852000 r-p 001be000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.1
7f60c7852000-7f60c7854000 rw-p 001c2000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.1
7f60c7854000-7f60c7859000 rw-p 00000000 00:00 0
7f60c7859000-7f60c787c000 r-xp 00000000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19.s
7f60c7a39000-7f60c7a3b000 rw-p 00000000 00:00 0
7f60c7a7a000-7f60c7a7b000 rw-p 00000000 00:00 0
7f60c7a7b000-7f60c7a7c000 r-p 00022000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19.s
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19.s
7f60c7a7d000-7f60c7a7e000 rw-p 00000000 00:00 0
7ffc5d2b2000-7ffc5d2d3000 rw-p 00000000 00:00 0 [stack]
7ffc5d3b0000-7ffc5d3b3000 r-p 00000000 00:00 0 [vvar]
7ffc5d3b3000-7ffc5d3b5000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

# Linux maps: list of maps

```
$ cat /proc/self/maps
```

```
00400000-0040b000 r-xp 00000000 08:01 48328831 /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 48328831 /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 48328831 /bin/cat
01974000-01995000 rw-p 00000000 00:00 0 [heap]
```

```
7f60c718b000-7f60c7490000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7490000-7f60c7490000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c764e000-7f60c764e000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c784e000-7f60c784e000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7852000-7f60c7852000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7854000-7f60c7854000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7859000-7f60c7859000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7a39000-7f60c7a39000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7a7a000-7f60c7a7a000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7a7b000-7f60c7a7b000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7a7c000-7f60c7a7c000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7a7d000-7f60c7a7d000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7ffc5d2b2000-7ffc5d2b2000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7ffc5d3b0000-7ffc5d3b0000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7ffc5d3b3000-7ffc5d3b3000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
ffffffffffff-ffffffffffff r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

OS tracks list of struct `vm_area_struct` with:  
(shown in this output):

virtual address start, end

permissions

offset in backing file (if any)

pointer to backing file (if any)

(not shown):

info about sharing of non-file data ...

# 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”

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. can't notice if instructions were executed in parallel



# 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 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

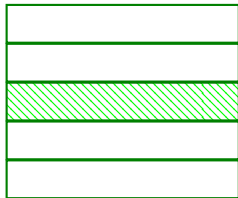
writing tens of **kilobytes** basically as fast as writing 512 bytes

SSD reads and writes: hundreds of microseconds

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

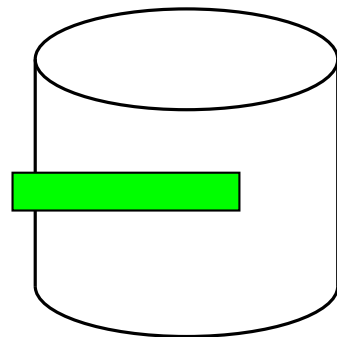
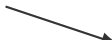
# swapping timeline

program A pages



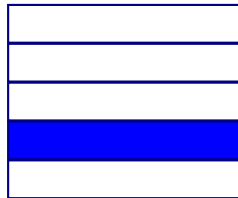
...

page fault



disk

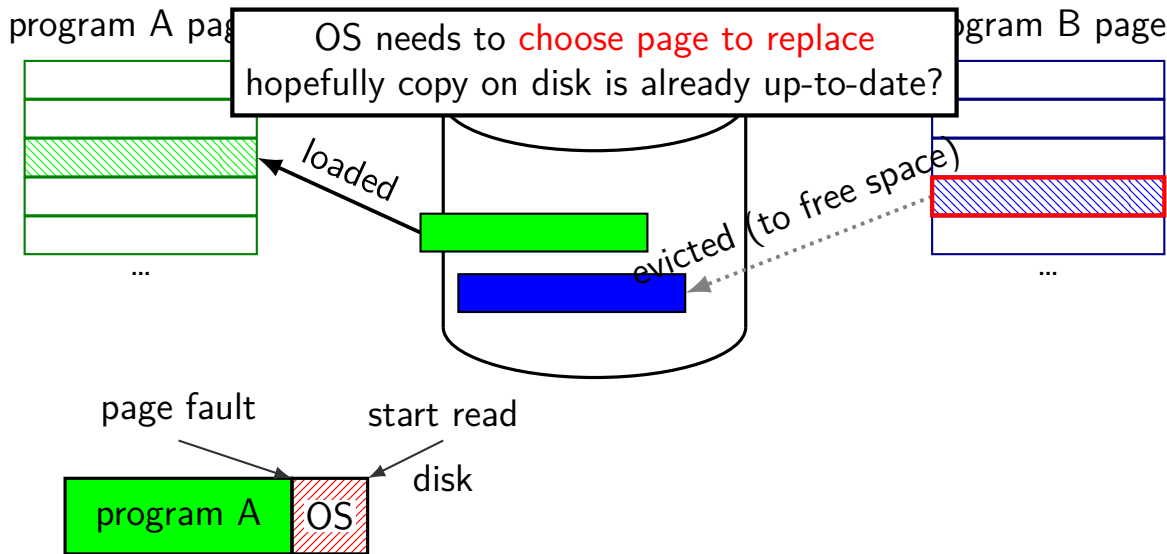
program B page



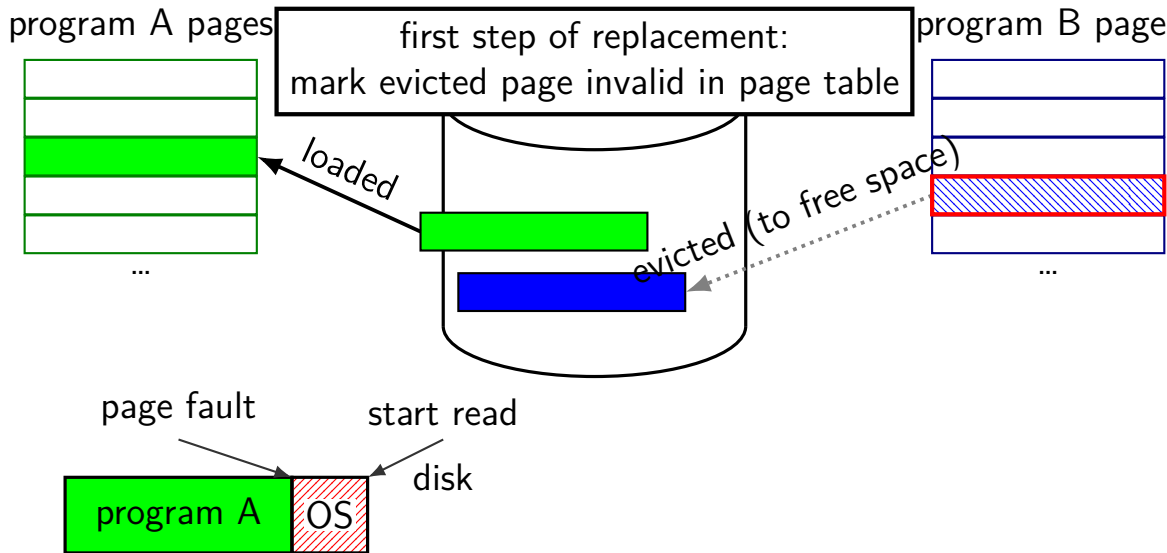
...



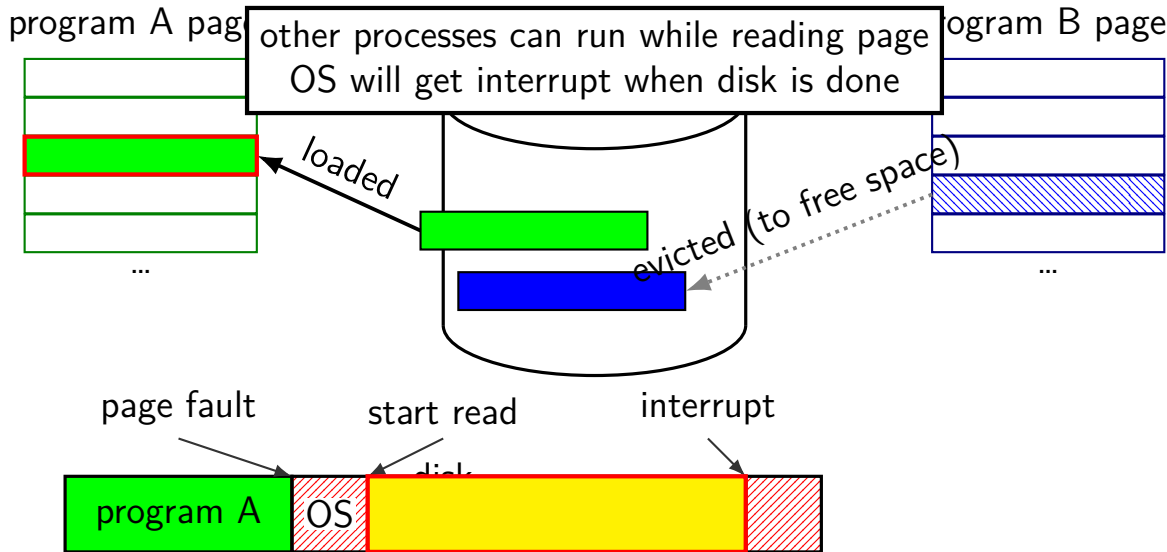
# swapping timeline



# swapping timeline

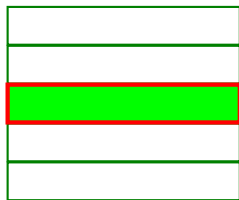


# swapping timeline



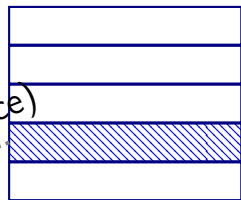
# swapping timeline

program A pages



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

program B page



loaded

evicted (to free space)

page fault

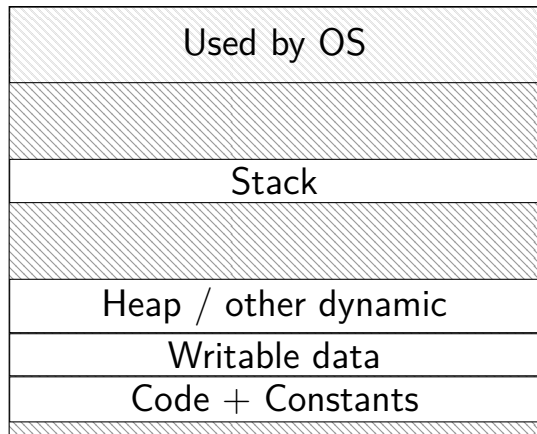
start read

interrupt

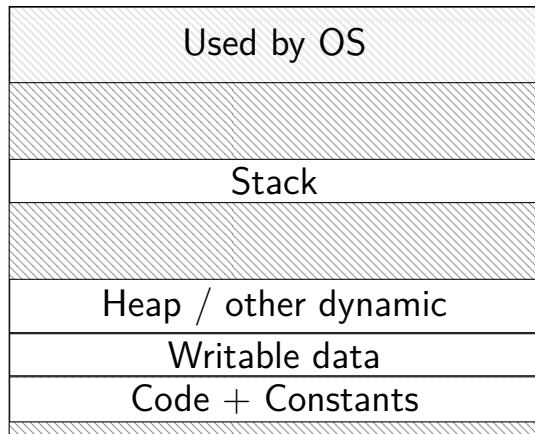


# do we really need a complete copy?

bash

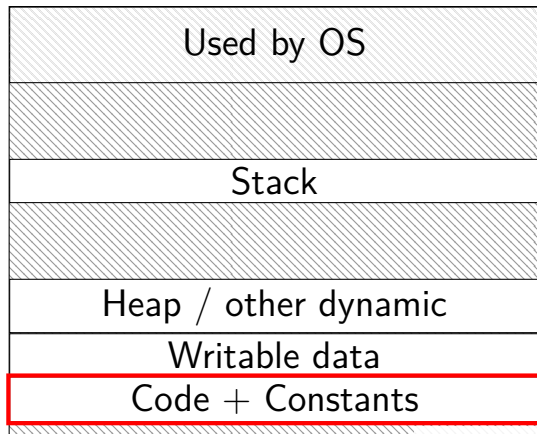


new copy of bash

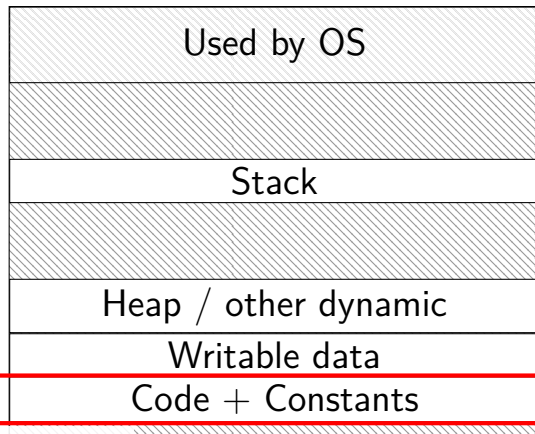


# do we really need a complete copy?

bash



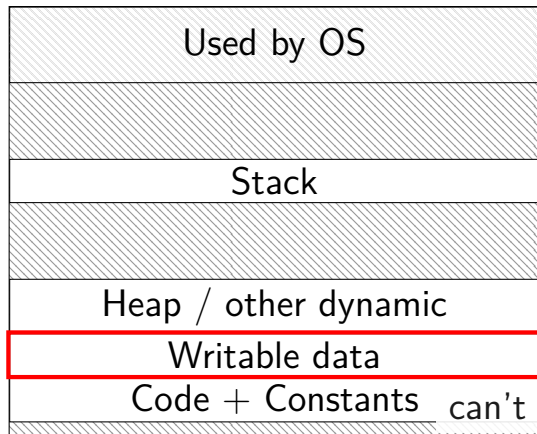
new copy of bash



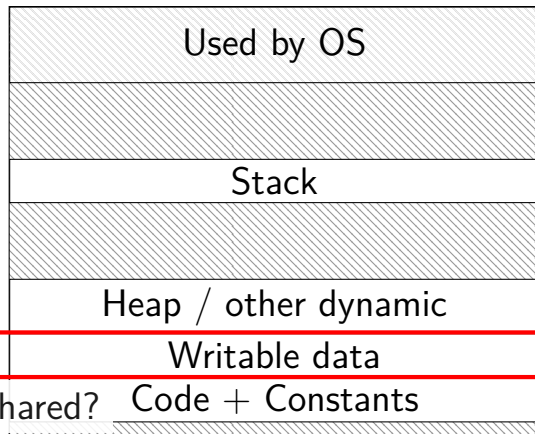
shared as read-only

# 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 it

- example: default value of global variables

- might typically not change

- (or OS might have preloaded executable's data anyways)

can we detect modifications?



## trick for extra sharing

sharing writeable data is fine — until either process modifies it

- example: default value of global variables

- might typically not change

- (or OS might have preloaded executable's data anyways)

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

**backup slides**