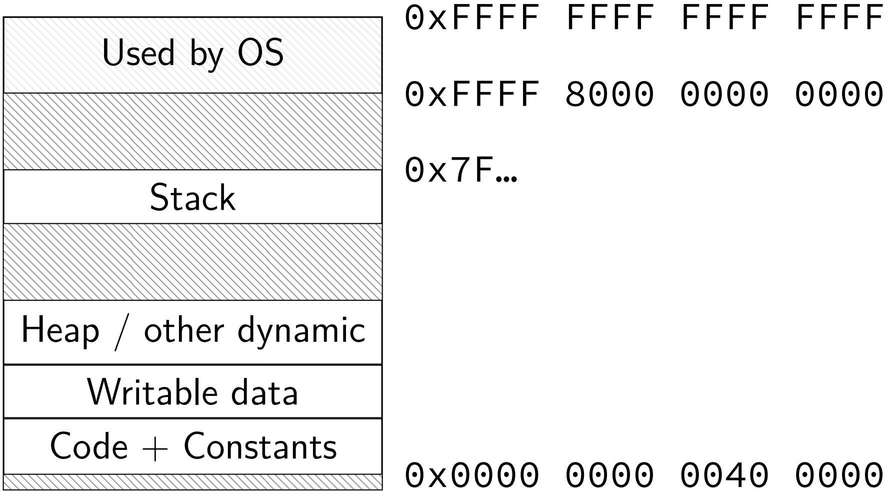
program memory



0xFFFF FFFF FFFF FFFF

Used by OS

0xFFFF 8000 0000 0000

0x7F...

Stack

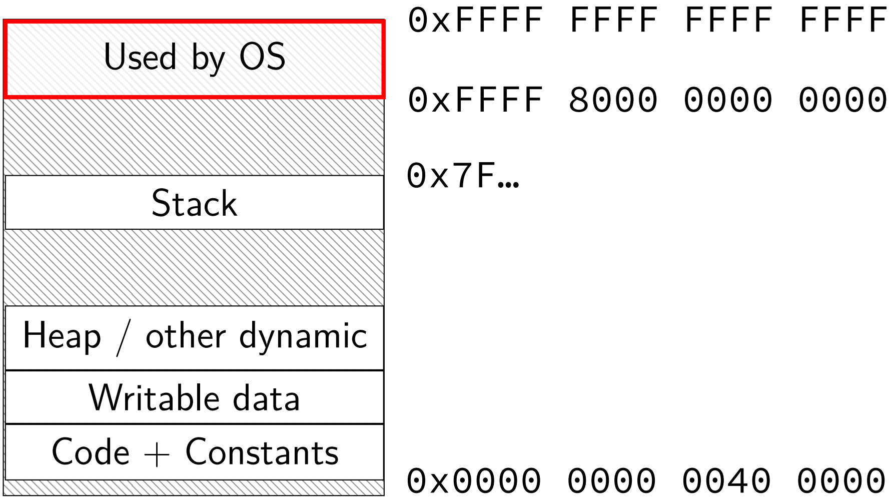
Heap / other dynamic

Writable data

Code + Constants

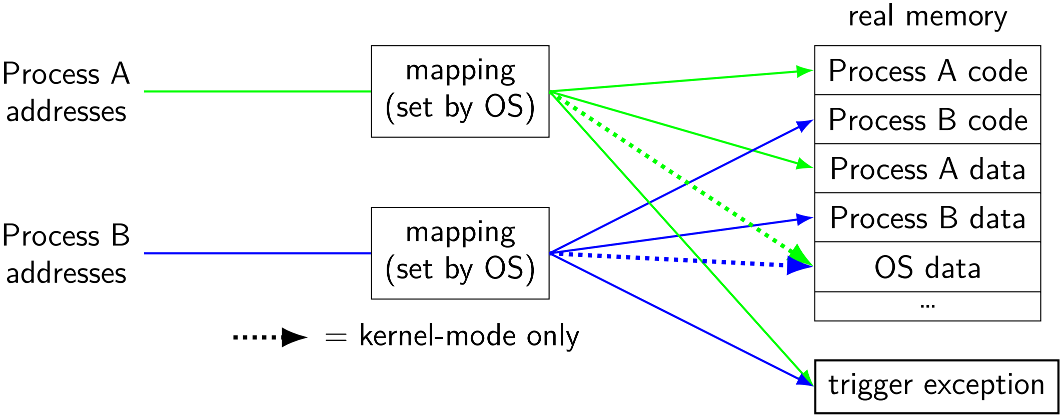
0x0000 0000 0040 0000

program memory



address spaces

illuision of dedicated memory



real memory

Process A code

Process B code

Process A data

Process B data

OS data

...

trigger exception

Process A addresses

Process B addresses

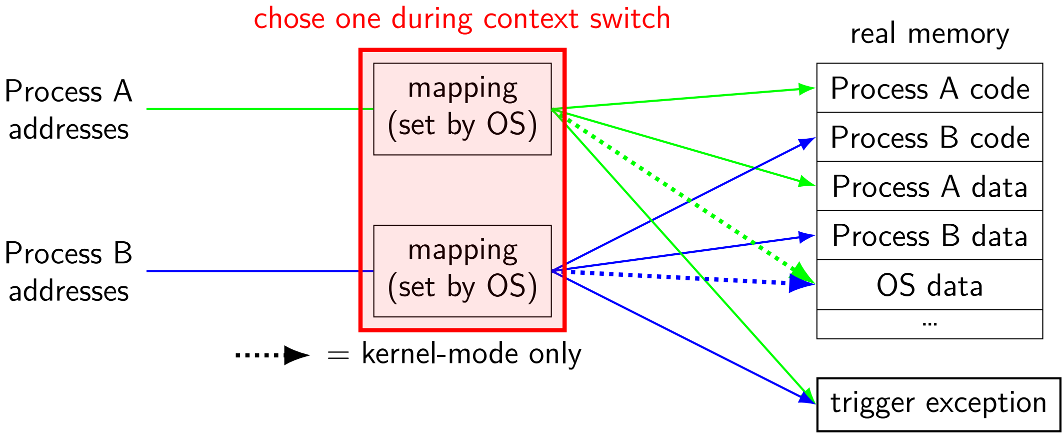
mapping (set by OS)

mapping (set by OS)

······► = kernel-mode only

address spaces

illuision of dedicated memory



chose one during context switch

real memory

Process A addresses

Process B addresses

mapping (set by OS)

mapping (set by OS)

Process A code

Process B code

Process A data

Process B data

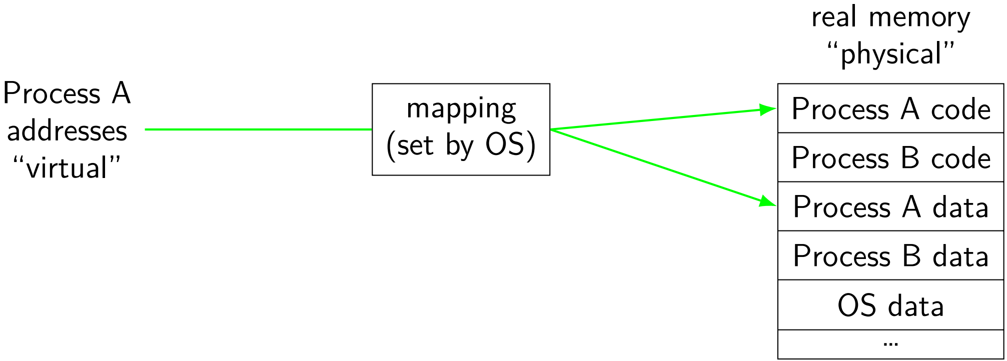
OS data

...

trigger exception

= kernel-mode only

address translation



Process A
addresses
“virtual”

mapping
(set by OS)

real memory
“physical”

Process A code

Process B code

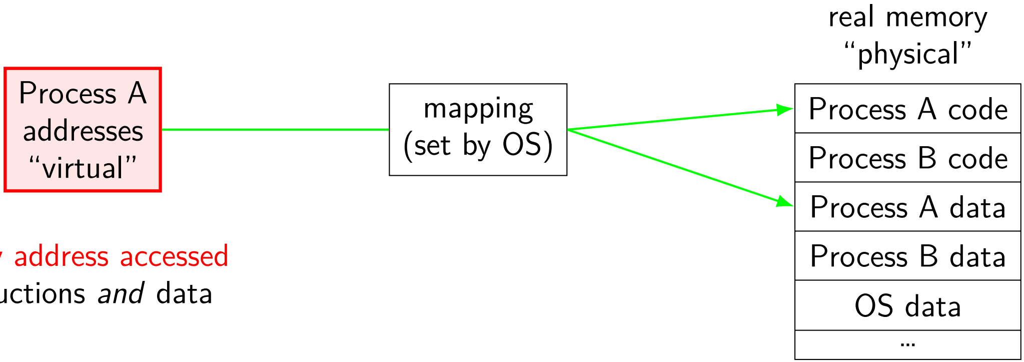
Process A data

Process B data

OS data

...

address translation



Process A addresses “virtual”

mapping (set by OS)

real memory “physical”

Process A code

Process B code

Process A data

Process B data

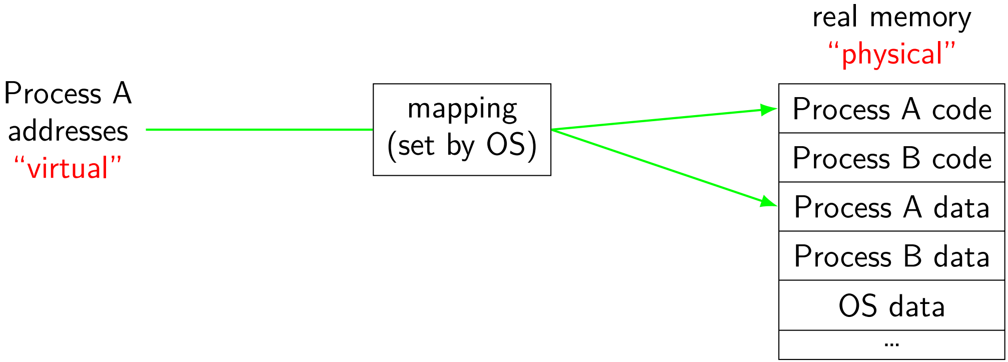
OS data

...

every address accessed

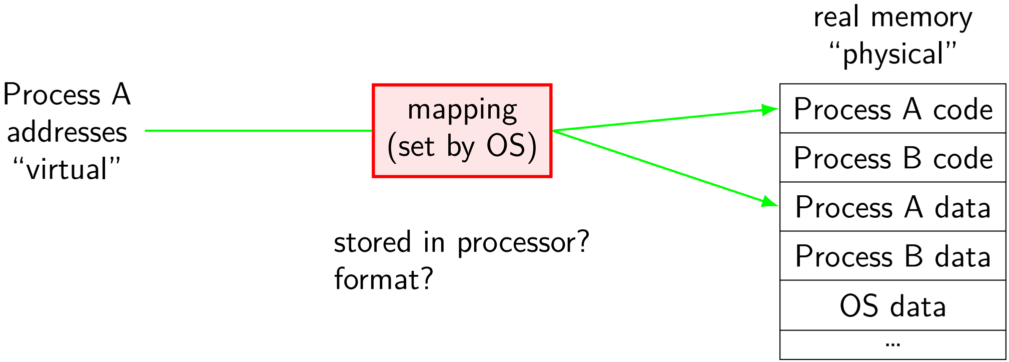
instructions *and* data

address translation



program addresses are ‘virtual’
real addresses are ‘physical’
can be different sizes!

address translation



Process A addresses “virtual”

mapping (set by OS)

real memory “physical”

Process A code

Process B code

Process A data

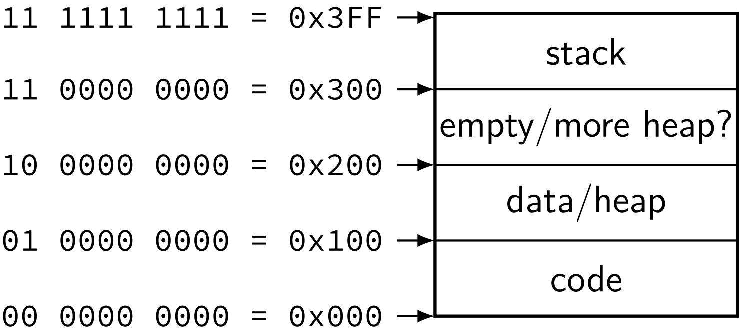
Process B data

OS data

...

stored in processor?
format?

toy program memory



11 1111 1111 = 0x3FF

stack

11 0000 0000 = 0x300

empty/more heap?

10 0000 0000 = 0x200

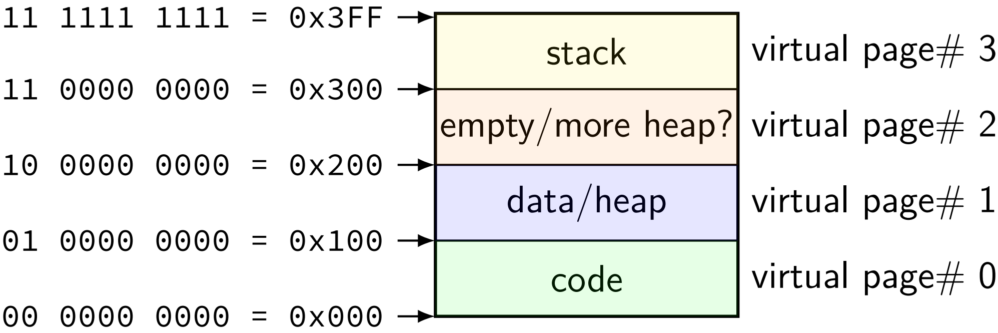
data/heap

01 0000 0000 = 0x100

code

00 0000 0000 = 0x000

toy program memory



11 1111 1111 = 0x3FF

11 0000 0000 = 0x300

10 0000 0000 = 0x200

01 0000 0000 = 0x100

00 0000 0000 = 0x000

stack

empty/more heap?

data/heap

code

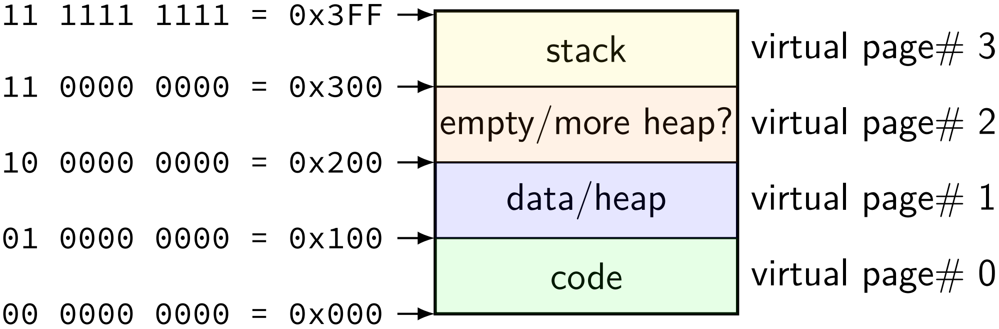
virtual page# 3

virtual page# 2

virtual page# 1

virtual page# 0

toy program memory



`11 1111 1111 = 0x3FF`

`11 0000 0000 = 0x300`

`10 0000 0000 = 0x200`

`01 0000 0000 = 0x100`

`00 0000 0000 = 0x000`

stack — virtual page# 3

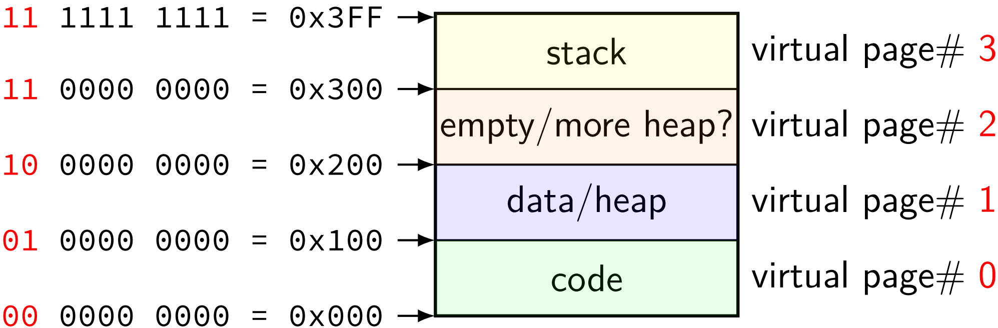
empty/more heap? — virtual page# 2

data/heap — virtual page# 1

code — virtual page# 0

divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

toy program memory



11 1111 1111 = 0x3FF

11 0000 0000 = 0x300

10 0000 0000 = 0x200

01 0000 0000 = 0x100

00 0000 0000 = 0x000

stack — virtual page# 3

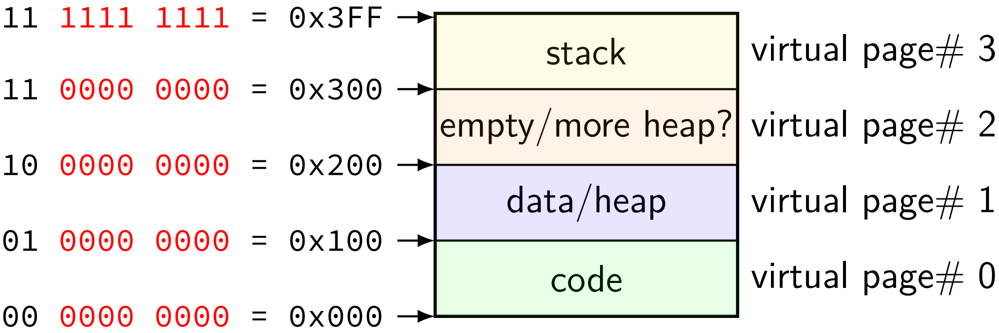
empty/more heap? — virtual page# 2

data/heap — virtual page# 1

code — virtual page# 0

page number is upper bits of address
(because page size is power of two)

toy program memory



11 1111 1111 = 0x3FF

stack — virtual page# 3

11 0000 0000 = 0x300

empty/more heap? — virtual page# 2

10 0000 0000 = 0x200

data/heap — virtual page# 1

01 0000 0000 = 0x100

code — virtual page# 0

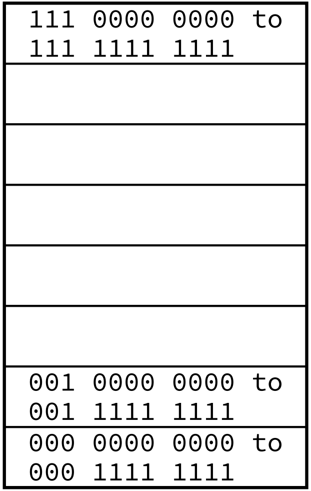
00 0000 0000 = 0x000

rest of address is called page offset

toy physical memory

real memory

physical addresses



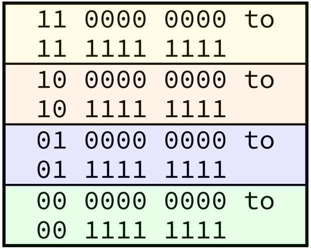
111 0000 0000 to
111 1111 1111

001 0000 0000 to
001 1111 1111

000 0000 0000 to
000 1111 1111

program memory

virtual addresses



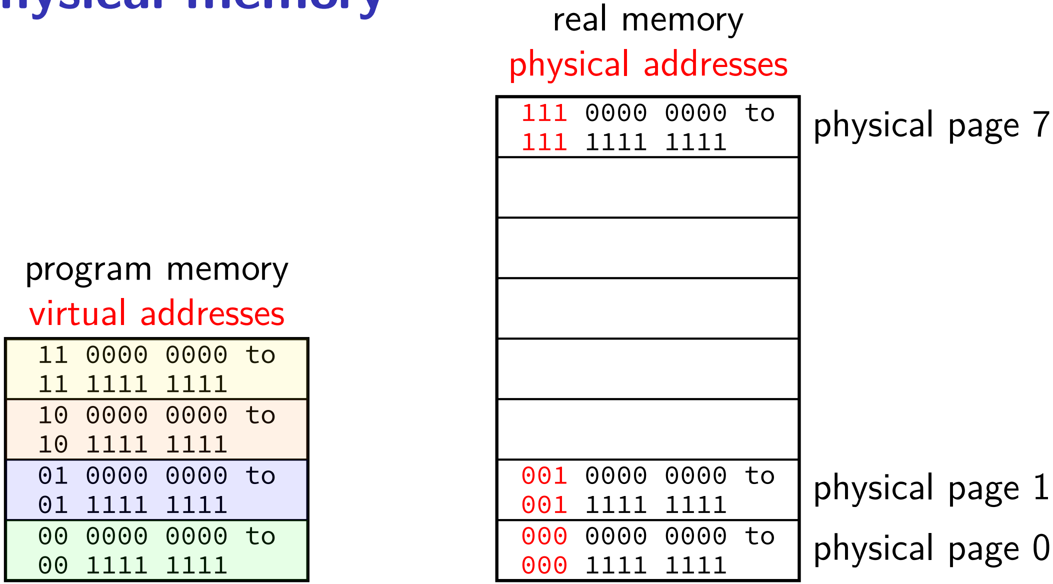
11 0000 0000 to
11 1111 1111

10 0000 0000 to
10 1111 1111

01 0000 0000 to
01 1111 1111

00 0000 0000 to
00 1111 1111

toy physical memory



real memory

physical addresses

```
111 0000 0000 to
111 1111 1111
```

physical page 7

```
001 0000 0000 to
001 1111 1111
```

physical page 1

```
000 0000 0000 to
000 1111 1111
```

physical page 0

program memory

virtual addresses

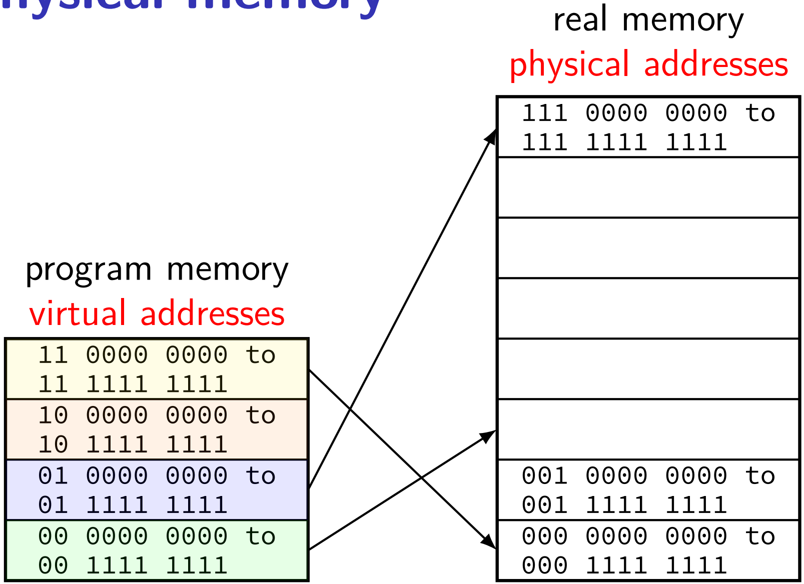
```
11 0000 0000 to
11 1111 1111

10 0000 0000 to
10 1111 1111

01 0000 0000 to
01 1111 1111

00 0000 0000 to
00 1111 1111
```

toy physical memory



real memory

physical addresses

```
111 0000 0000 to
111 1111 1111
```

```
001 0000 0000 to
001 1111 1111
```

```
000 0000 0000 to
000 1111 1111
```

program memory

virtual addresses

```
11 0000 0000 to
11 1111 1111
```

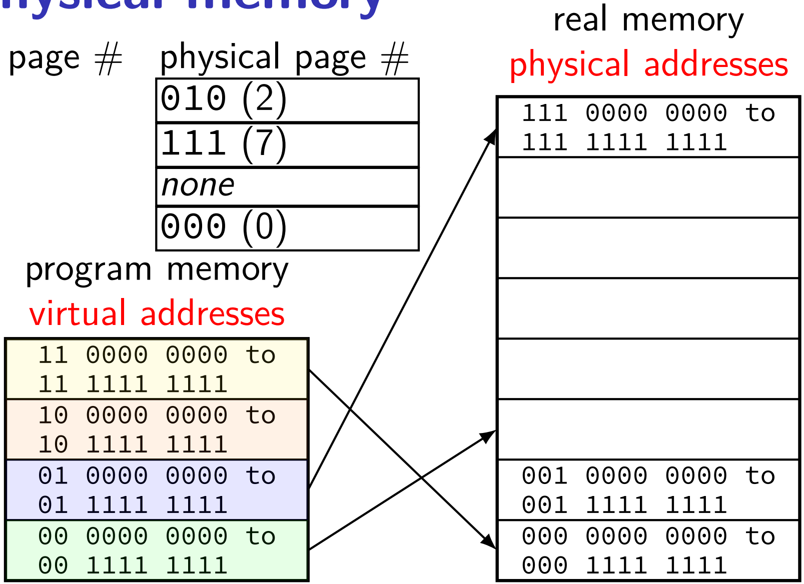
```
10 0000 0000 to
10 1111 1111
```

```
01 0000 0000 to
01 1111 1111
```

```
00 0000 0000 to
00 1111 1111
```

toy physical memory

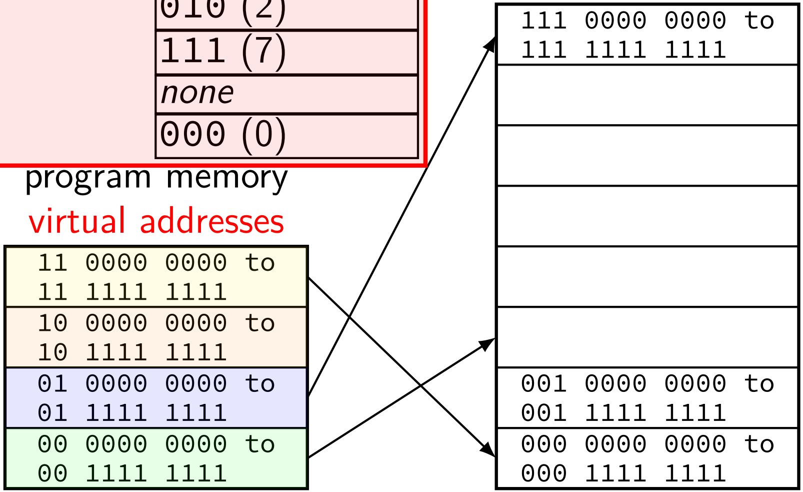
virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
11	000 (0)



toy physical memory

page table!

virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
11	000 (0)



real memory

physical addresses

111 0000 0000 to
111 1111 1111

001 0000 0000 to
001 1111 1111

000 0000 0000 to
000 1111 1111

program memory

virtual addresses

11 0000 0000 to
11 1111 1111

10 0000 0000 to
10 1111 1111

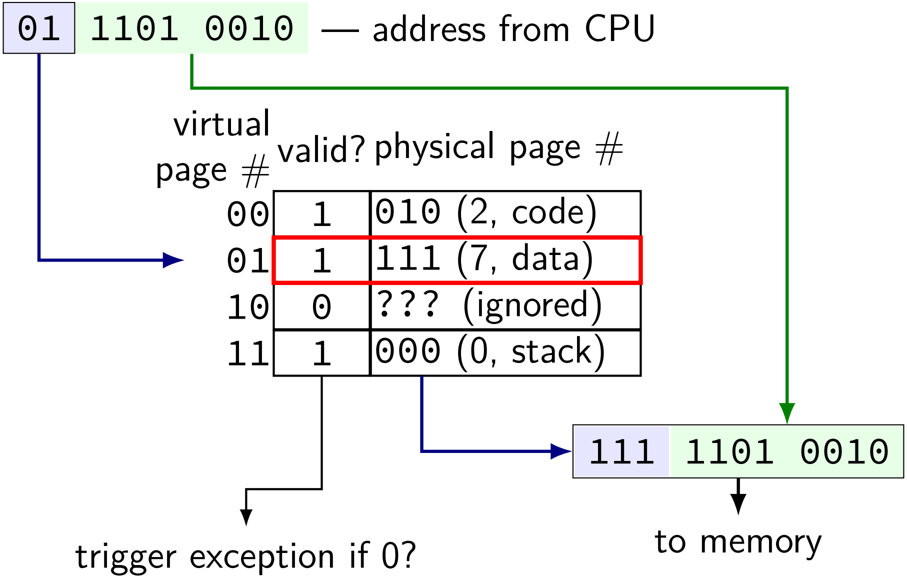
01 0000 0000 to
01 1111 1111

00 0000 0000 to
00 1111 1111

toy page table lookup

virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

toy page table lookup



`01` `1101 0010` — address from CPU

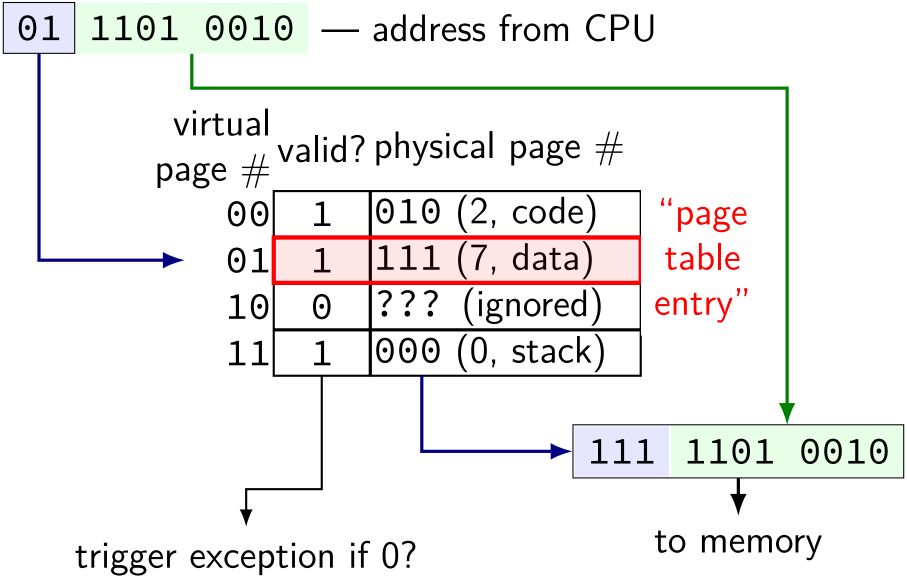
virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

`111` `1101 0010`

trigger exception if 0?

to memory

toy page table lookup



`01 1101 0010` — address from CPU

virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

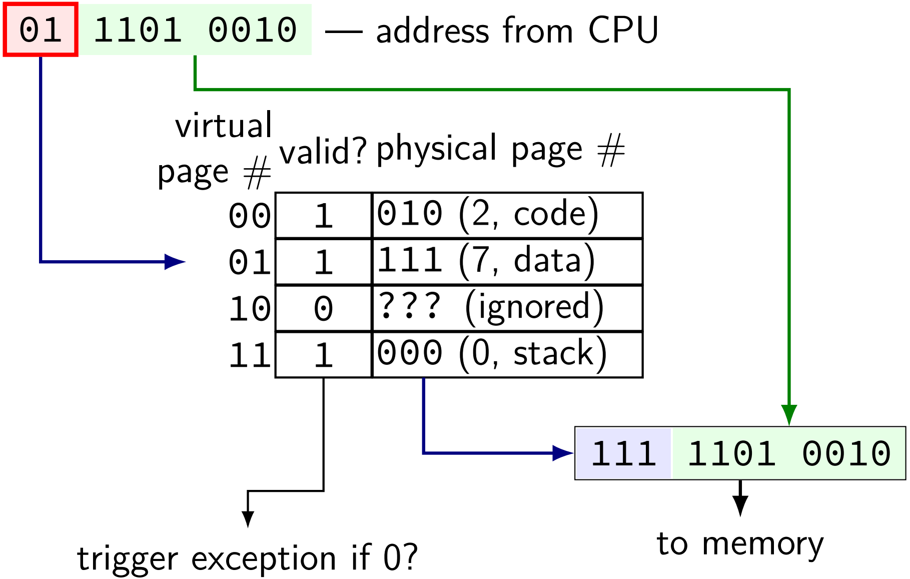
"page table entry"

`111 1101 0010`

to memory

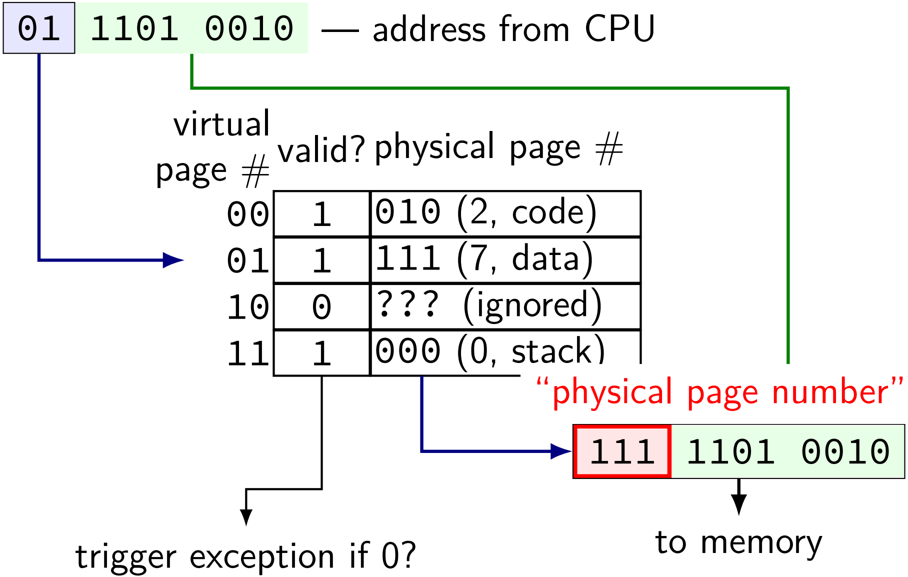
trigger exception if 0?

t “virtual page number” lookup



virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

toy page table lookup

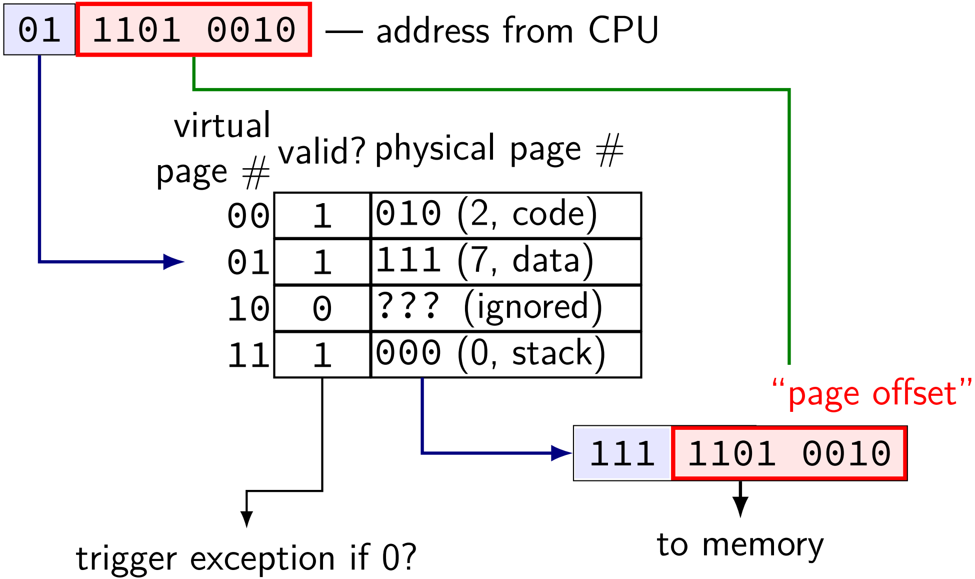


`01 1101 0010` — address from CPU

virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

toy pag ookup

“page offset”



`01` `1101 0010` — address from CPU

virtual page #	valid?	physical page #
`00`	`1`	`010` (2, code)
`01`	`1`	`111` (7, data)
`10`	`0`	`???` (ignored)
`11`	`1`	`000` (0, stack)

“page offset”

`111` `1101 0010`

to memory

trigger exception if 0?

on virtual address sizes

virtual address size = size of pointer?

often, but — sometimes part of pointer not used

example: typical x86-64 only use 48 bits

- rest of bits have fixed value

virtual address size is amount used for mapping

address space sizes

amount of stuff that can be addressed $=$ address space size

- based on number of unique addresses

e.g. 32-bit virtual address $= 2^{32}$ byte virtual address space

e.g. 20-bit physical addresss $= 2^{20}$ byte physical address space

address space sizes

amount of stuff that can be addressed = address space size

- based on number of unique addresses

e.g. 32-bit virtual address = 2^{32} byte virtual address space

e.g. 20-bit physical addresss = 2^{20} byte physical address space

what if my machine has 3GB of memory (not power of two)?

- not all addresses in physical address space are useful
- most common situation (since CPUs support having a lot of memory)

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)

pgae table entries have physical page #, valid bit, 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)

pgae table entries have physical page $\#$, valid bit, bit

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

2^{20} entries $\times (18 + 1)$ 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`

exercise: page table lookup

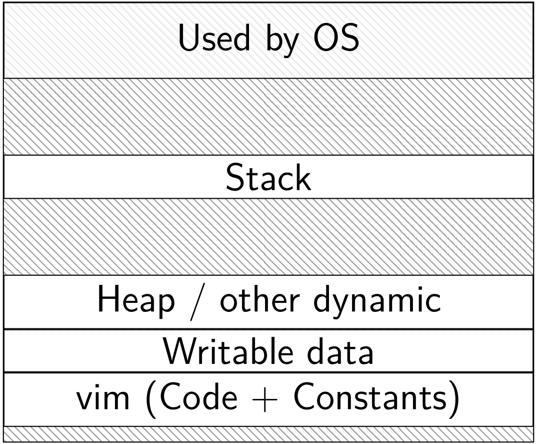
suppose 64-byte pages (= 6-bit page offsets), 9-bit virtual addresses

VPN	valid	PPN
`000`	`1`	`0010`
`001`	`1`	`1010`
`010`	`0`	`---`
`011`	`0`	`---`
`100`	`1`	`1110`
`101`	`1`	`0100`
`110`	`1`	`0001`
`111`	`0`	`---`

virtual address `0x024` (`0 0010 0100`) = physical address ???

vim (two copies)

Vim (run by user mst3k)



Used by OS

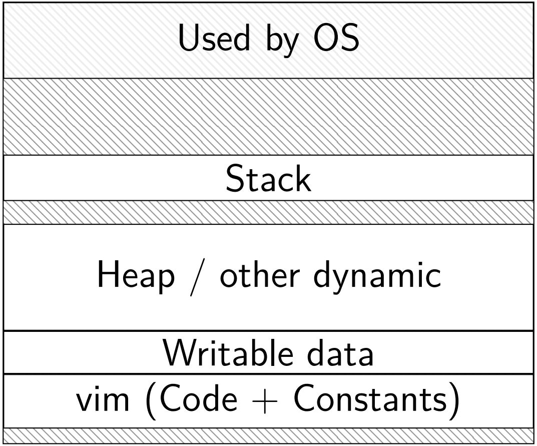
Stack

Heap / other dynamic

Writable data

vim (Code + Constants)

Vim (run by user xyz4w)



Used by OS

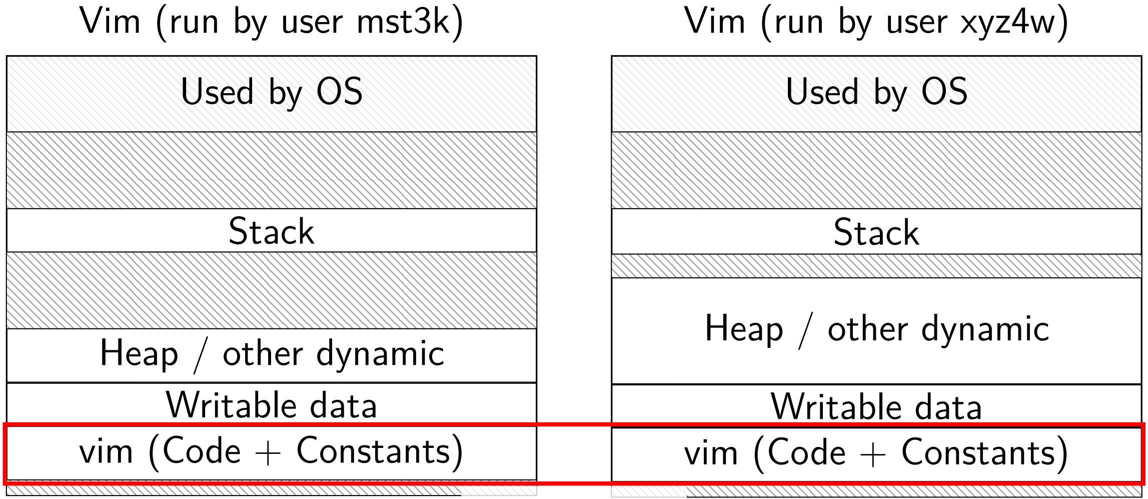
Stack

Heap / other dynamic

Writable data

vim (Code + Constants)

vim (two copies)



Vim (run by user mst3k)

Used by OS

Stack

Heap / other dynamic

Writable data

vim (Code + Constants)

Vim (run by user xyz4w)

Used by OS

Stack

Heap / other dynamic

Writable data

vim (Code + Constants)

same data?

two copies of program

would like to only have one copy of program

what if `mst3k`'s vim 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?

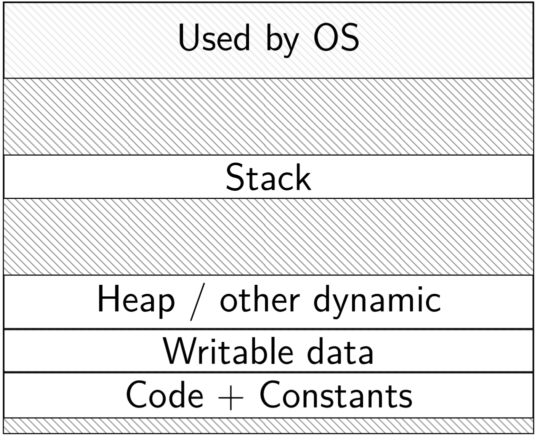
checked by hardware like valid bit

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

running a program

Some program



Used by OS

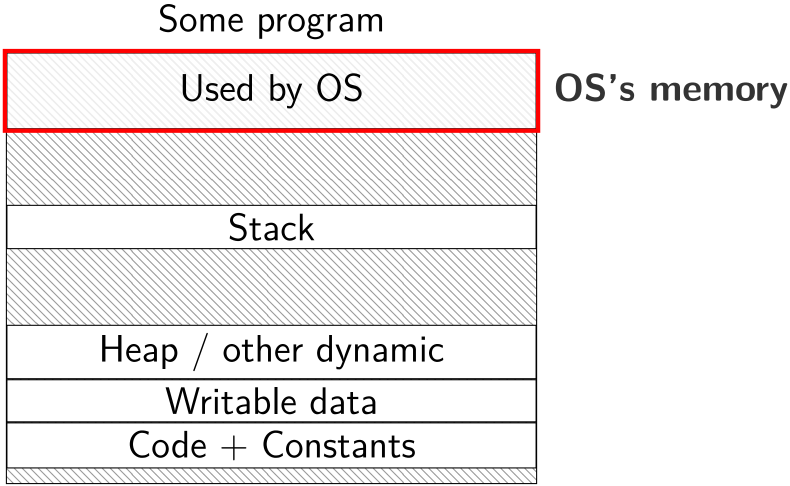
Stack

Heap / other dynamic

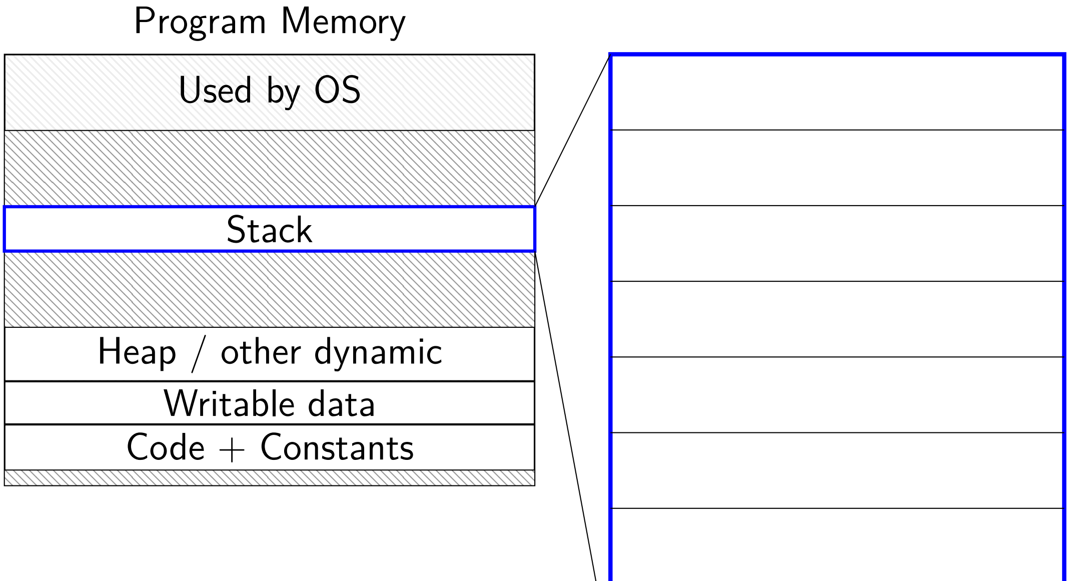
Writable data

Code + Constants

running a program



space on demand



Program Memory

Used by OS

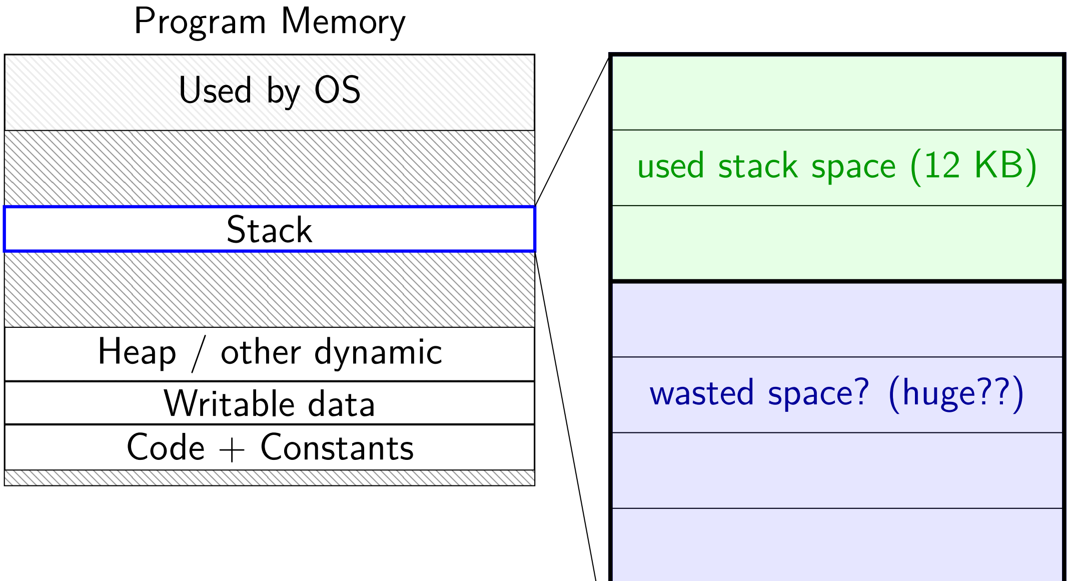
Stack

Heap / other dynamic

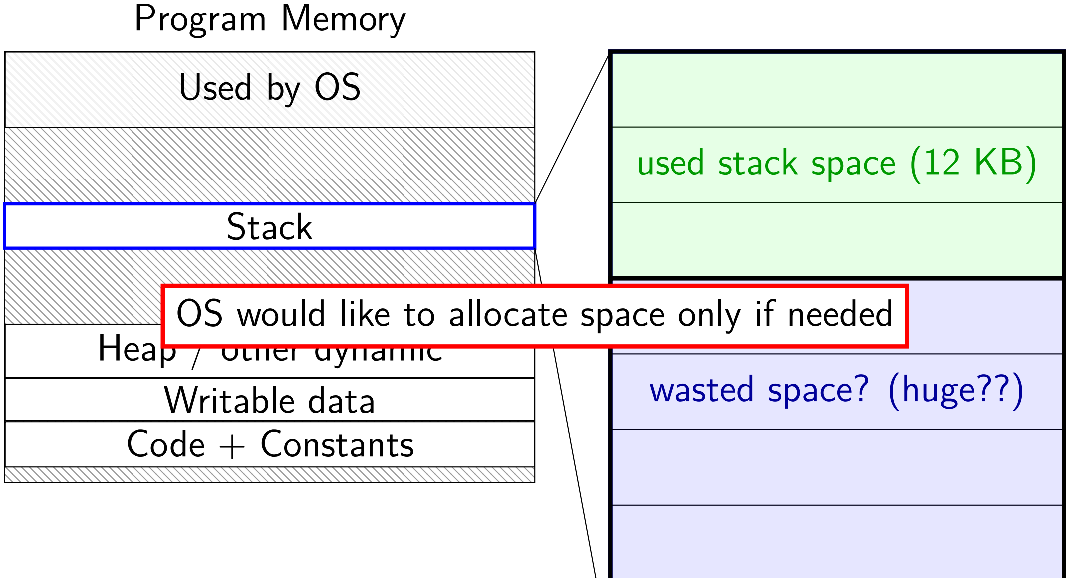
Writable data

Code + Constants

space on demand



space on demand



OS would like to allocate space only if needed

allocating space on demand

```
%rsp = 0x7FFFC000
```

```
...
// requires more stack space
A: pushq %rbx

B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

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

allocating space on demand

`%rsp = 0x7FFFC000`

```
...
// requires more stack space
A: pushq %rbx

B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

page fault!

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

pushq triggers exception

hardware says “accessing address `0x7FFFBFF8`”

OS looks up what’s should be there — “stack”

allocating space on demand

`%rsp = 0x7FFFC000`

```
...
// requires more stack space
A: pushq %rbx

B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

restarted

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

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

allocating space on demand

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

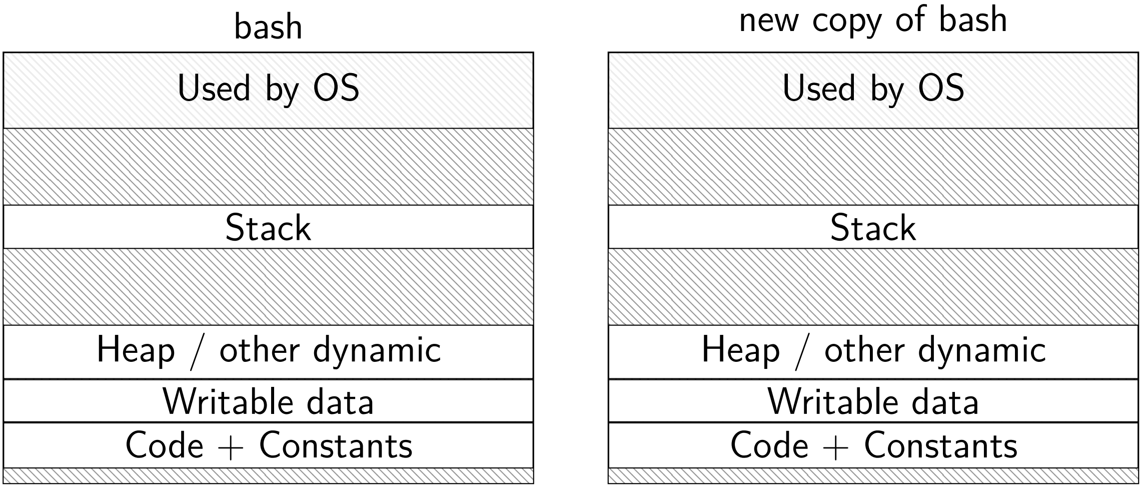
only change: load from file, etc. instead of allocating empty page

loading program can be merely creating empty page table

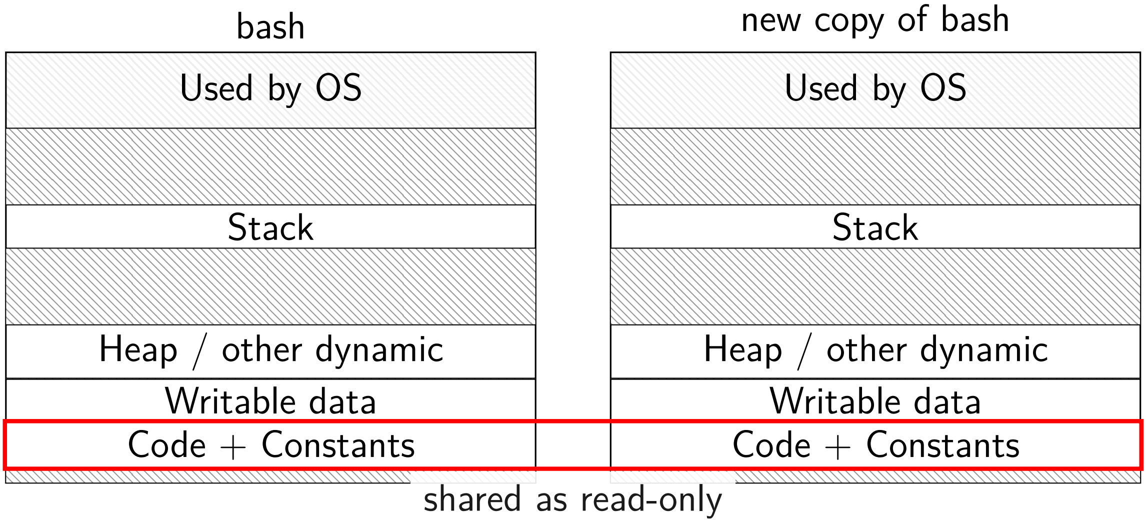
everything else can be handled in response to page faults

- no time/space spent loading/allocating unneeded space

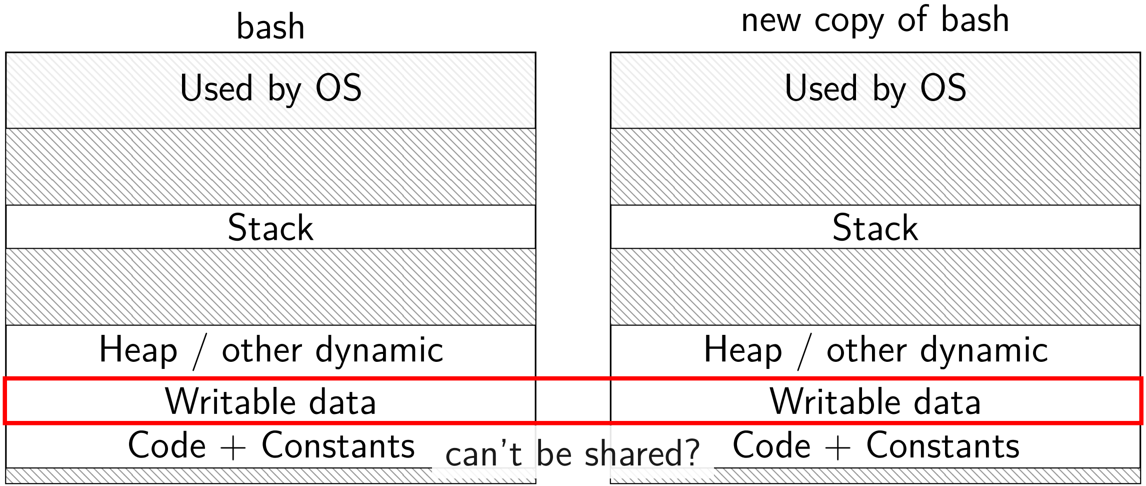
do we really need a complete copy?



do we really need a complete copy?



do we really need a complete copy?



bash

Used by OS

Stack

Heap / other dynamic

Writable data

Code + Constants

new copy of bash

Used by OS

Stack

Heap / other dynamic

Writable data

Code + Constants

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
…	…	…	…

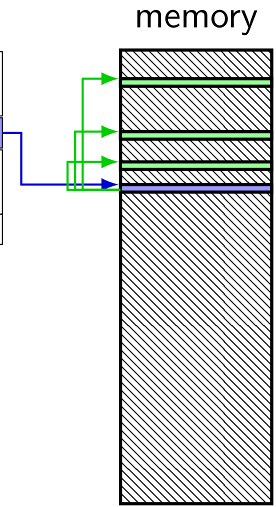
VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	1	0x300FD
…	…	…	…

after allocating a copy, OS reruns the write instruction

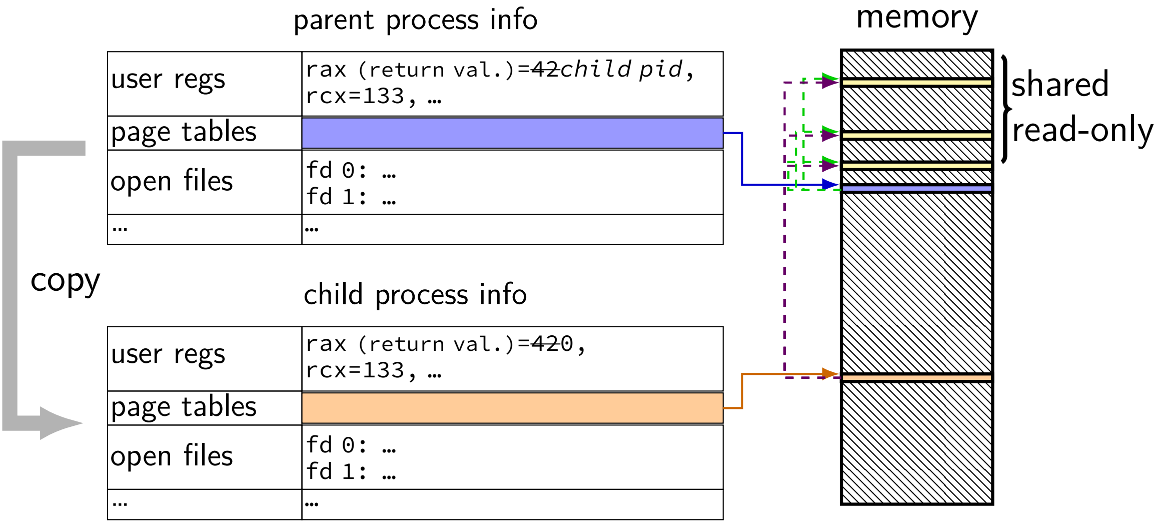
fork (w/ copy-on-write, if parent writes first)

parent process info

user regs	rax (return val.)=~~42~~*child pid*, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…



fork (w/ copy-on-write, if parent writes first)



parent process info

user regs	`rax (return val.)=`~~`42`~~*child pid*`,`
	`rcx=133, …`

page tables	

open files	`fd 0: …`
	`fd 1: …`

…	…

child process info

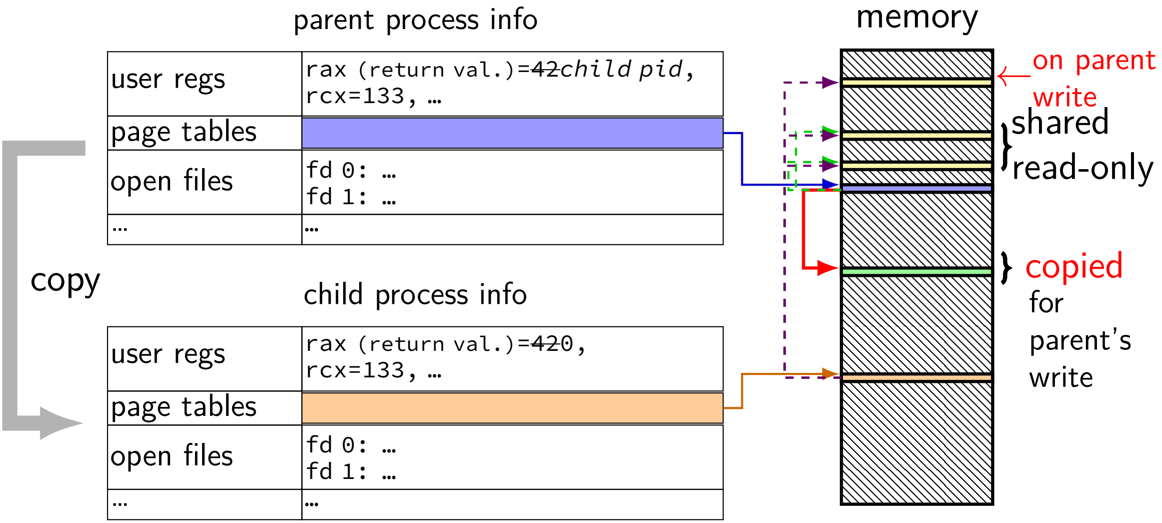
user regs	`rax (return val.)=`~~`42`~~`0,`
	`rcx=133, …`

page tables	

open files	`fd 0: …`
	`fd 1: …`

…	…

fork (w/ copy-on-write, if parent writes first)



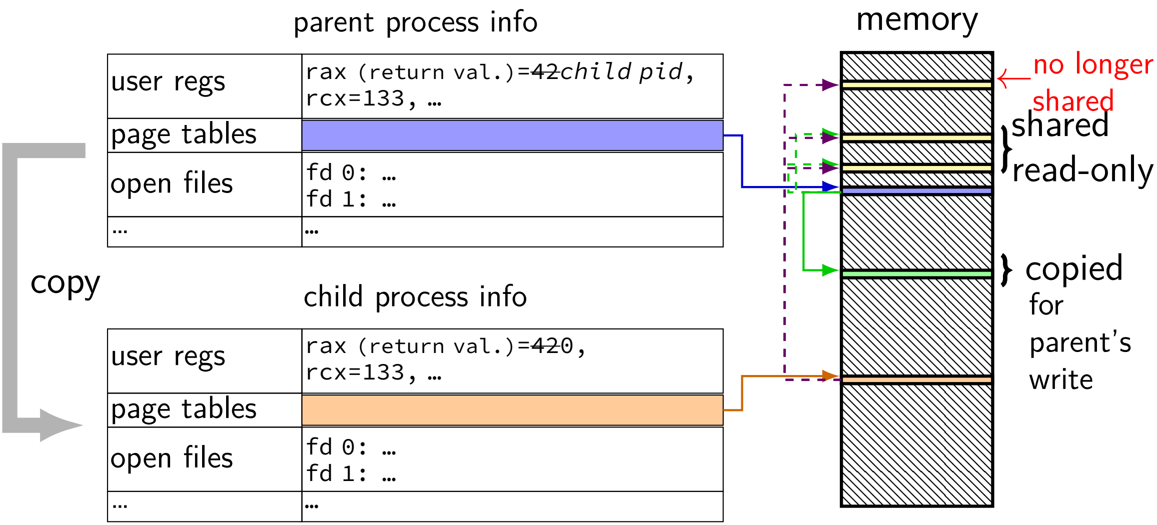
parent process info

user regs	rax (return val.)=~~42~~*child pid*, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

child process info

user regs	rax (return val.)=~~42~~0, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

fork (w/ copy-on-write, if parent writes first)



parent process info

user regs	`rax (return val.)=`~~`42`~~*child pid*`,`
	`rcx=133, …`

page tables	

open files	`fd 0: …`
	`fd 1: …`

…	…

copy

child process info

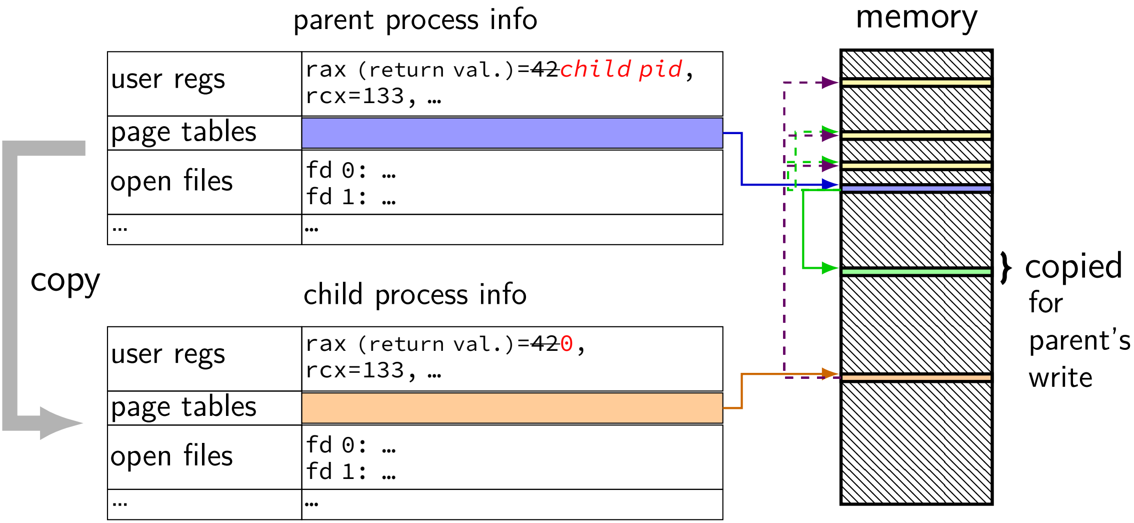
user regs	`rax (return val.)=`~~`42`~~`0,`
	`rcx=133, …`

page tables	

open files	`fd 0: …`
	`fd 1: …`

…	…

fork (w/ copy-on-write, if parent writes first)



parent process info

user regs	`rax (return val.)=`~~`42`~~*`child pid`*`,`
	`rcx=133, …`

page tables	

open files	`fd 0: …`
	`fd 1: …`

…	…

copy

child process info

user regs	`rax (return val.)=`~~`42`~~`0,`
	`rcx=133, …`

page tables	

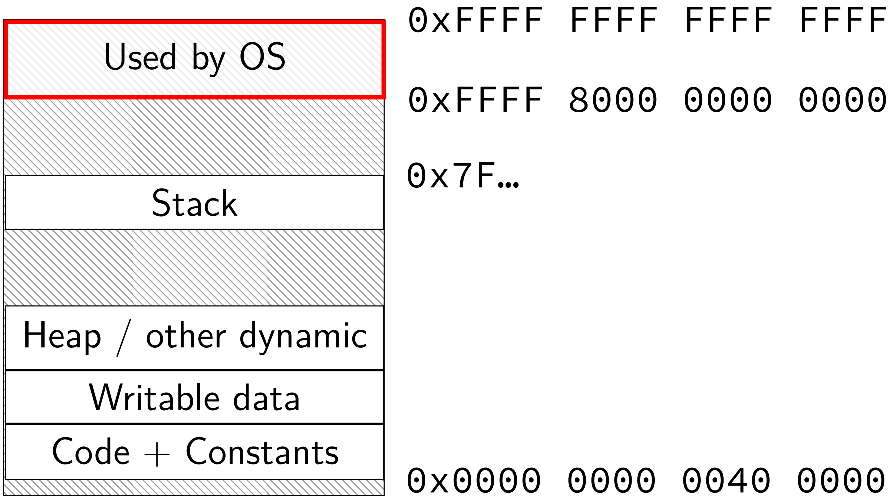
open files	`fd 0: …`
	`fd 1: …`

…	…

memory

} copied for parent's write

program memory



Used by OS

Stack

Heap / other dynamic

Writable data

Code + Constants

`0xFFFF FFFF FFFF FFFF`

`0xFFFF 8000 0000 0000`

`0x7F...`

`0x0000 0000 0040 0000`

running OS code

system calls, I/O events, etc. run OS code in kernel mode

running OS code

system calls, I/O events, etc. run OS code in kernel mode

where in memory is this OS code?

running OS code

system calls, I/O events, etc. run OS code in kernel mode

where in memory is this OS code?

- probably have a page table entry pointing to it
- marked not accessible in user mode

running OS code

system calls, I/O events, etc. run OS code in kernel mode

where in memory is this OS code?

- probably have a page table entry pointing to it
- marked not accessible in user mode

code better not be modified by user program

- otherwise: uncontrolled way to “escape” user mode

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
```

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.
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.
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109   /lib/x86_64-linux-gnu/ld-2.19.
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]
ffffffffff600000-ffffffffff601000 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−(
01974000−(
7f60c718b0                                                       cale−archive
7f60c74900                                                       gnu/libc−2.1
7f60c764e0                                                       gnu/libc−2.1
7f60c784e0                                                       gnu/libc−2.1
7f60c78520                                                       gnu/libc−2.1
7f60c78540
7f60c78590                                                       gnu/ld−2.19.s
7f60c7a390
7f60c7a7a0
7f60c7a7b0                                                       gnu/ld−2.19.s
7f60c7a7c0                                                       gnu/ld−2.19.s
7f60c7a7d0
7ffc5d2b20
7ffc5d3b00
7ffc5d3b30
ffffffffff600000−ffffffffff601000 r−xp 00000000 00:00 0 [vsyscall]
```

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 ...

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

phys. page 0

phys. page 1

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` = `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` = 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

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”

page table is for the hardware and not the OS

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand

- “swapping”

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand

- "swapping"

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand

“swapping”

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand

“swapping”

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand

  “swapping”

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

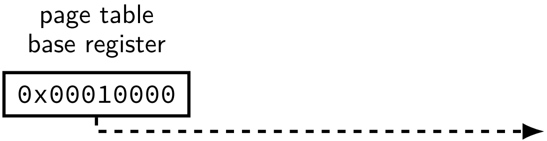
valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)

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

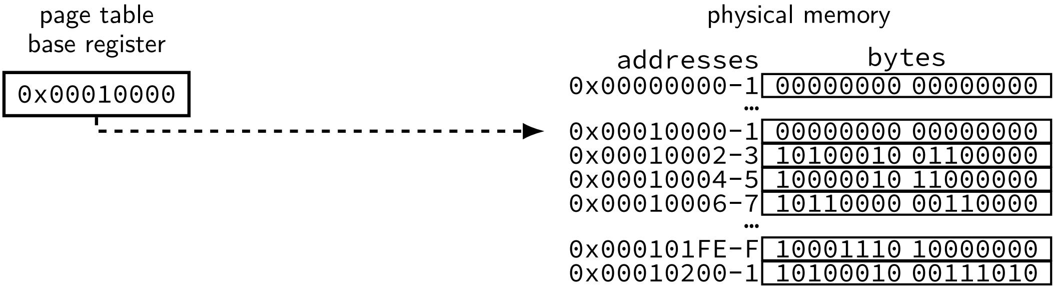
0x00010000

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 table base register

0x00010000

physical memory

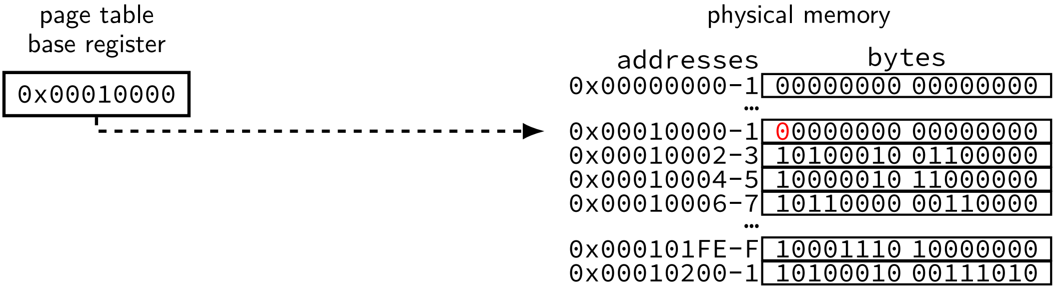
addresses	bytes
0x00000000-1	00000000 00000000
…	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
…	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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



page table base register

`0x00010000`

physical memory

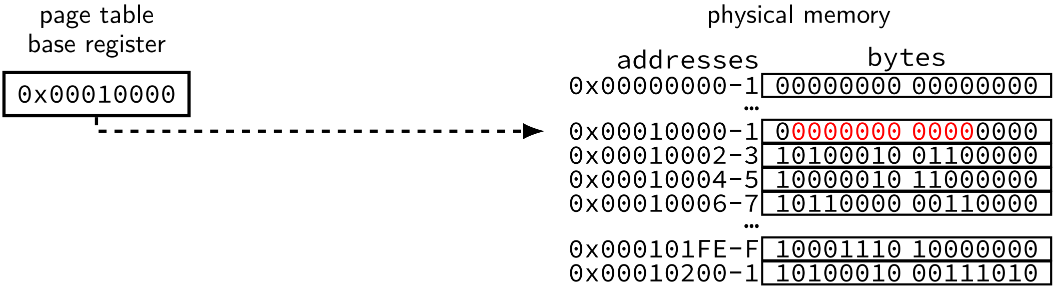
addresses	bytes
0x00000000-1	00000000 00000000
...	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
...	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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



page table base register

0x00010000

physical memory

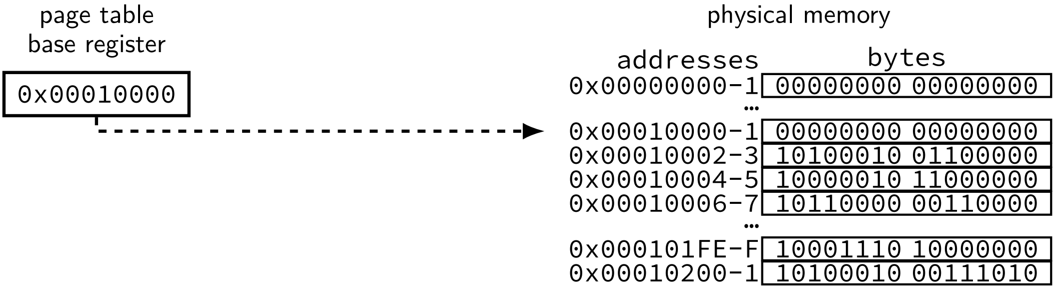
addresses	bytes
0x00000000-1	00000000 00000000
…	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
…	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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



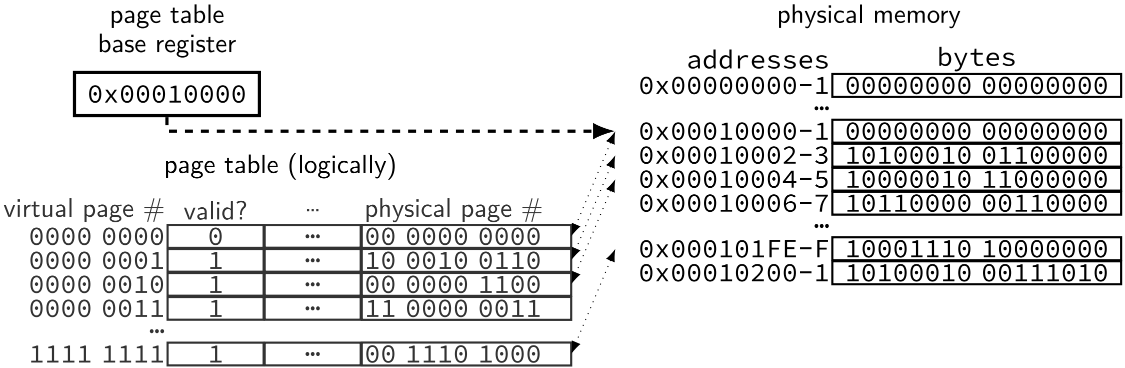
addresses	bytes
0x00000000-1	00000000 00000000
…	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
…	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (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 table (logically)

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

physical memory

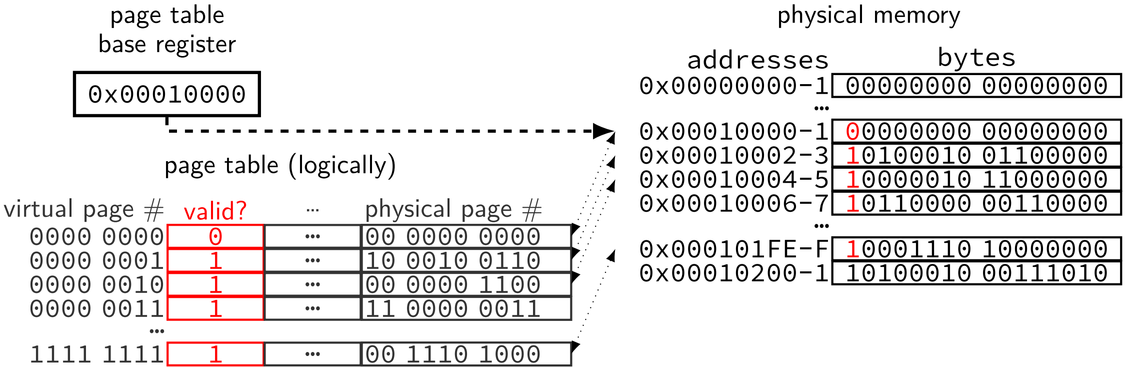
addresses	bytes
0x00000000-1	00000000 00000000
…	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
…	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (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 table (logically)

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

physical memory

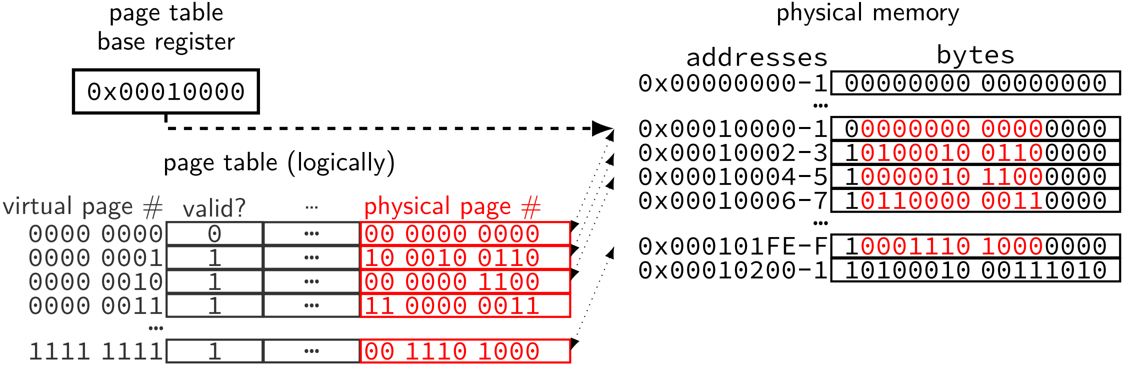
addresses	bytes
0x00000000-1	00000000 00000000
…	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
…	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (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 table (logically)

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

physical memory

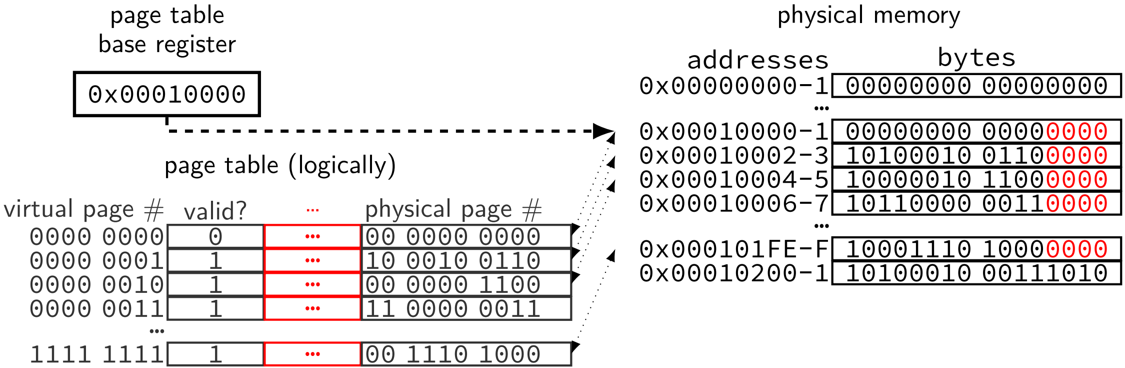
addresses	bytes
0x00000000-1	00000000 00000000
…	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
…	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (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 table (logically)

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

physical memory

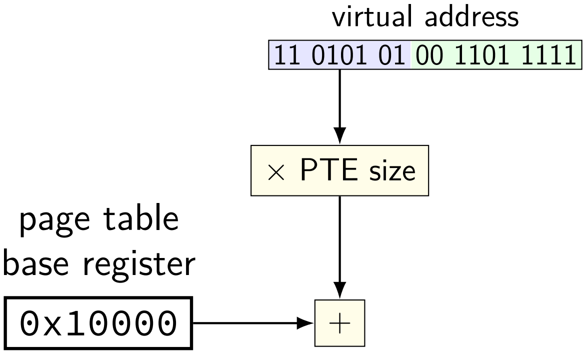
addresses	bytes
0x00000000-1	00000000 00000000
···	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
···	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

memory access with page table

virtual address

11 0101 01	00 1101 1111

memory access with page table



virtual address

11 0101 01 00 1101 1111

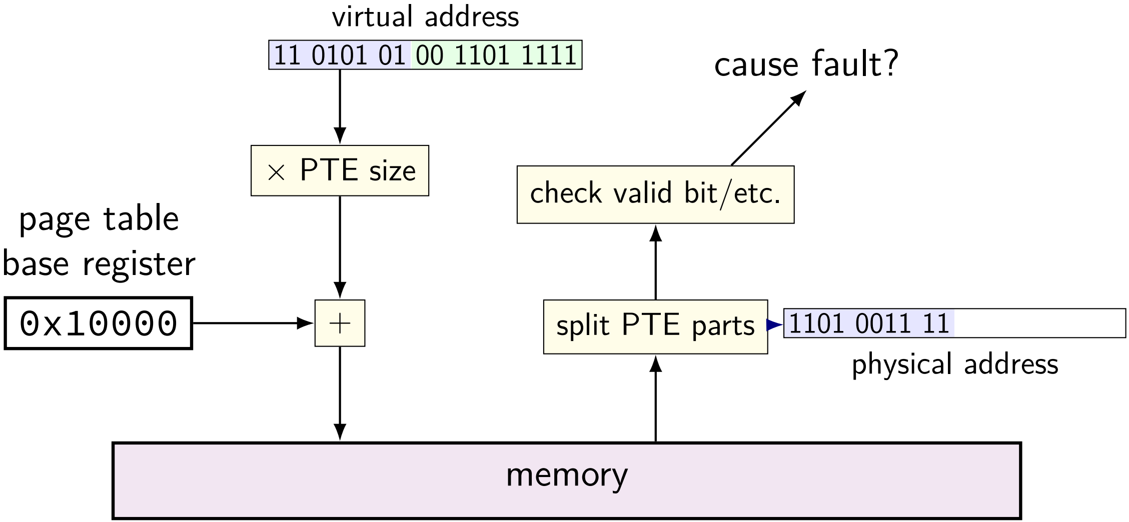
× PTE size

page table base register

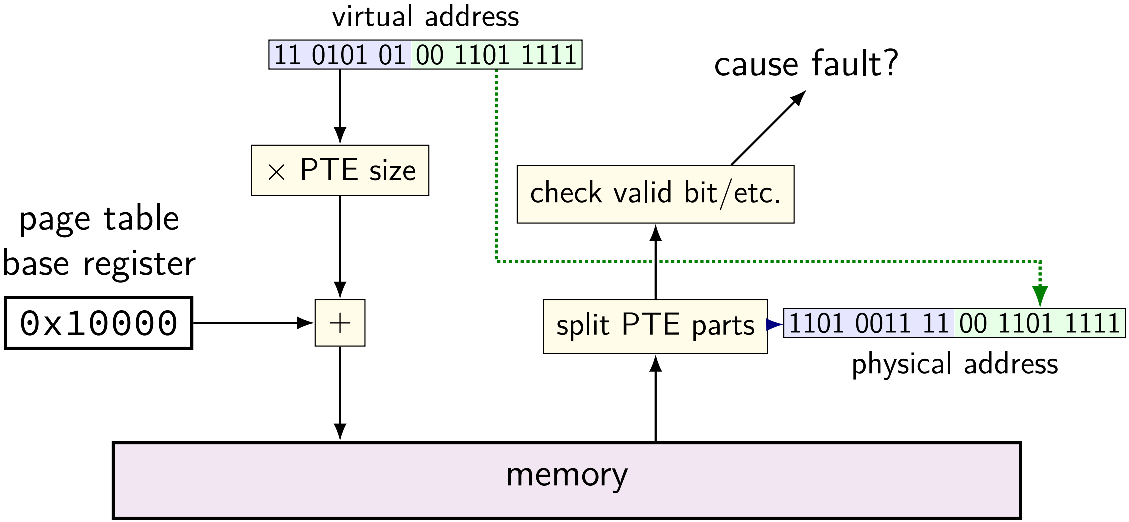
0x10000

+

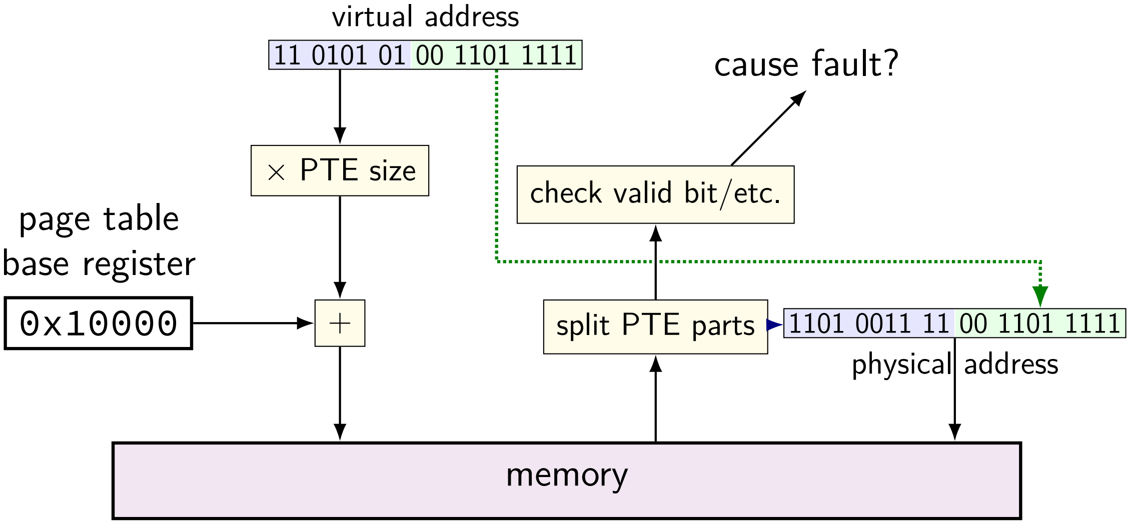
memory access with page table



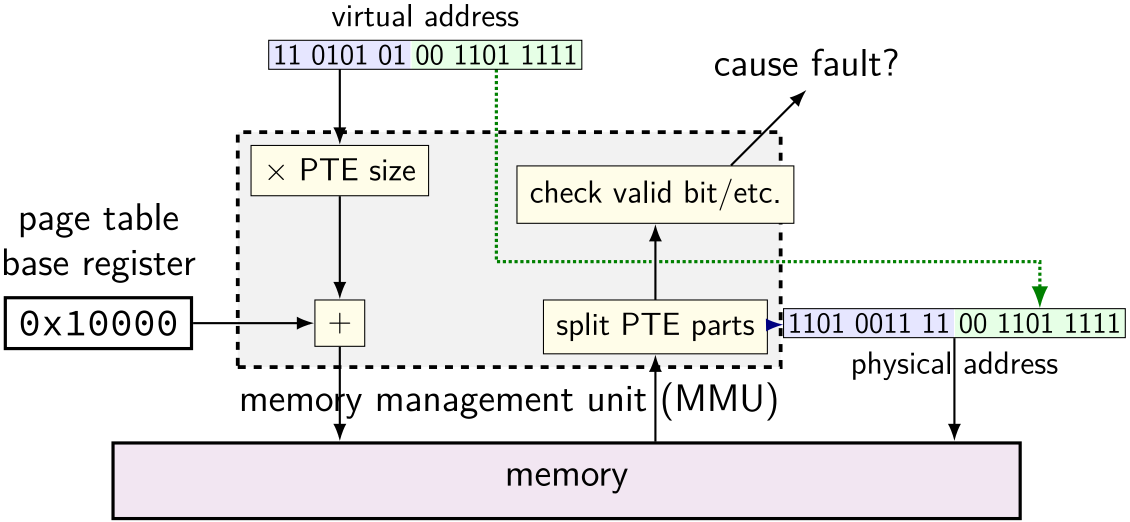
memory access with page table



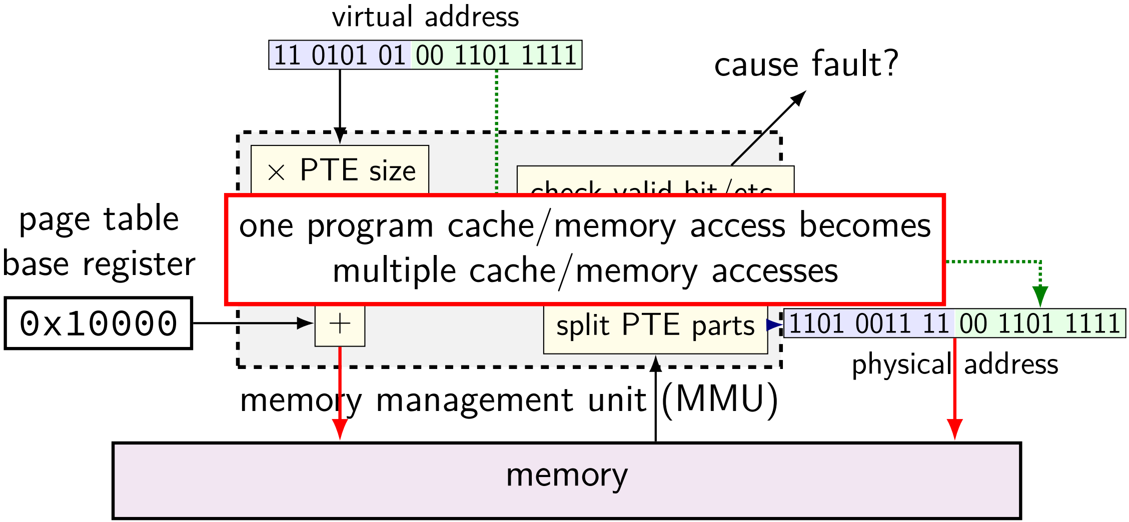
memory access with page table



memory access with page table

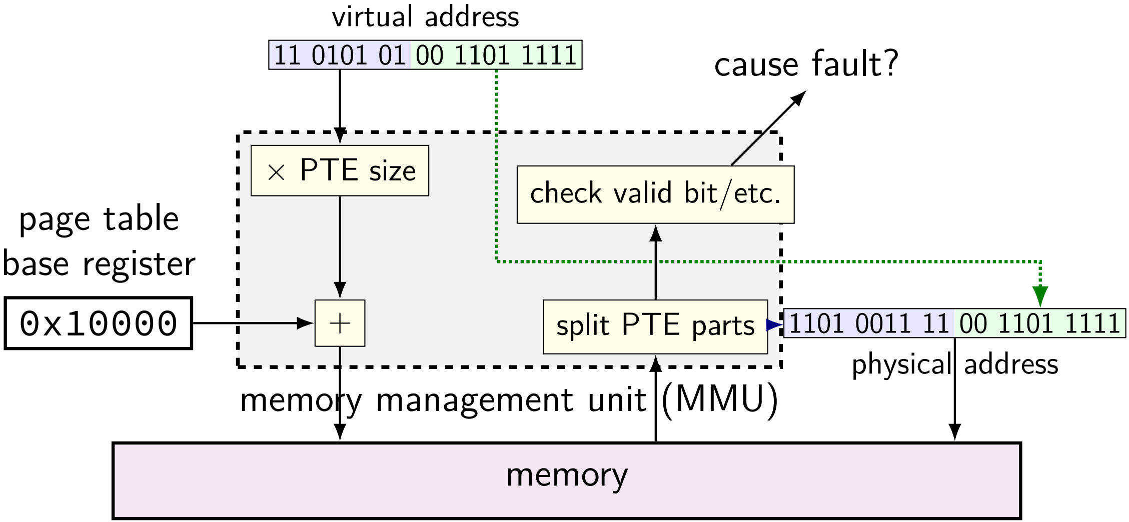


memory access with page table



one program cache/memory access becomes multiple cache/memory accesses

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` + `110` ×1 = `0x26`

PTE value:

`0xF6` = `1111 0110`

PPN `111`, valid `1`

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

→ `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 + 110 ×1 = 0x26

PTE value:

0xF6 = 1111 0110

PPN 111, valid 1

M[111 001] = M[0x39]

→ 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` + `110` × `1` = `0x26`

PTE value:

`0xF6` = `1111 0110`

PPN `111`, valid `1`

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

→ `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` + `110` ×1 = `0x26`

PTE value:

`0xF6` = `1111 0110`

PPN `111`, valid 1

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

→ `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 E2 D1 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 E2 D1 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 ×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 E2 D1 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 ×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 E2 D1 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 ×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 E2 D1 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 ×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 ... /* page offset bits */
#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 (~0L) if invalid
size_t 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:

ptbr = GetPointerToTable(

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

)

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:

ptbr = GetPointerToTable(

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

)

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	0	—
...	...	...

)

allocated with `posix_memalign`

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	0	—
...	...	...

)

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 (two weeks after break): all LEVELS, both functions

- in preparation for code review
- due Weds BEFORE LAB

part 3 (two weeks after break): final submission

- Friday after code review
- most of grade based on this
- will test previous parts again

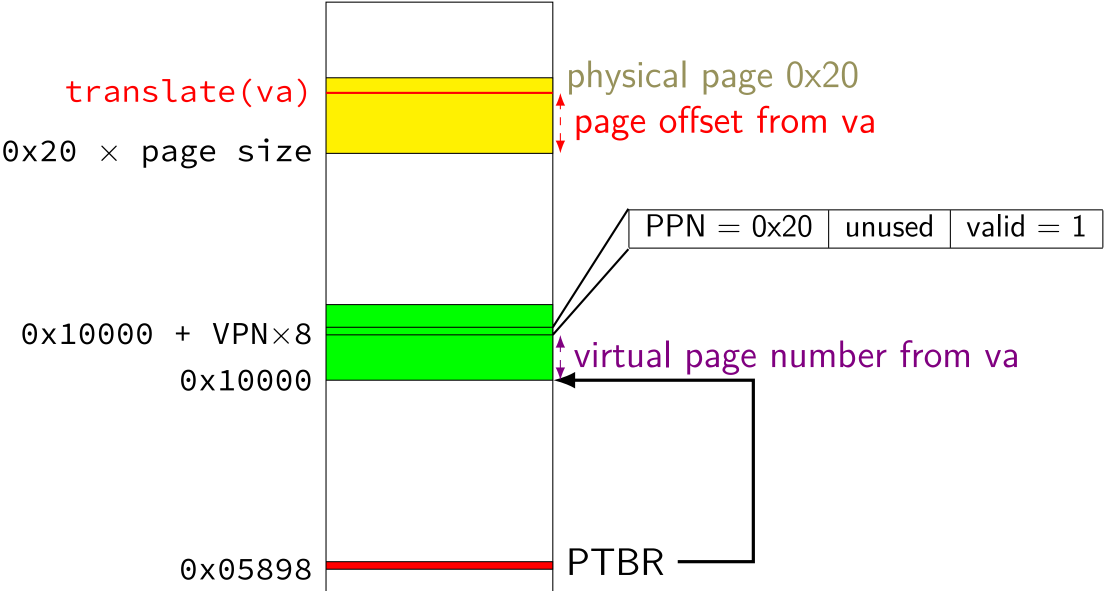
address/page table entry format

(with POBITS=12, LEVELS=1)

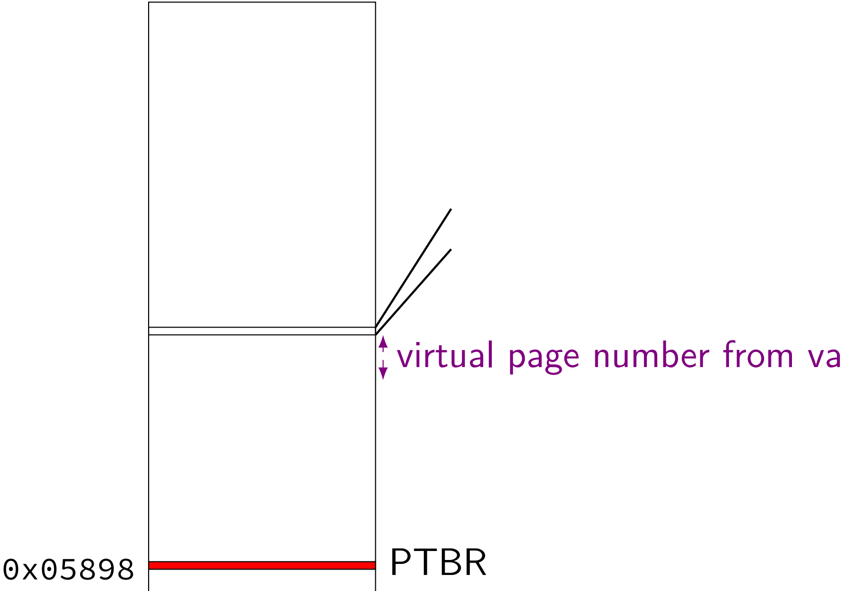
	bits 63–21	bits 20–12	bits 11–1	bit 0
page table entry	physical page number		unused	valid bit
virtual address	unused	virtual page number	page offset	
physical address	physical page number		page offset	

in assignment: value from posix_memalign = physical address

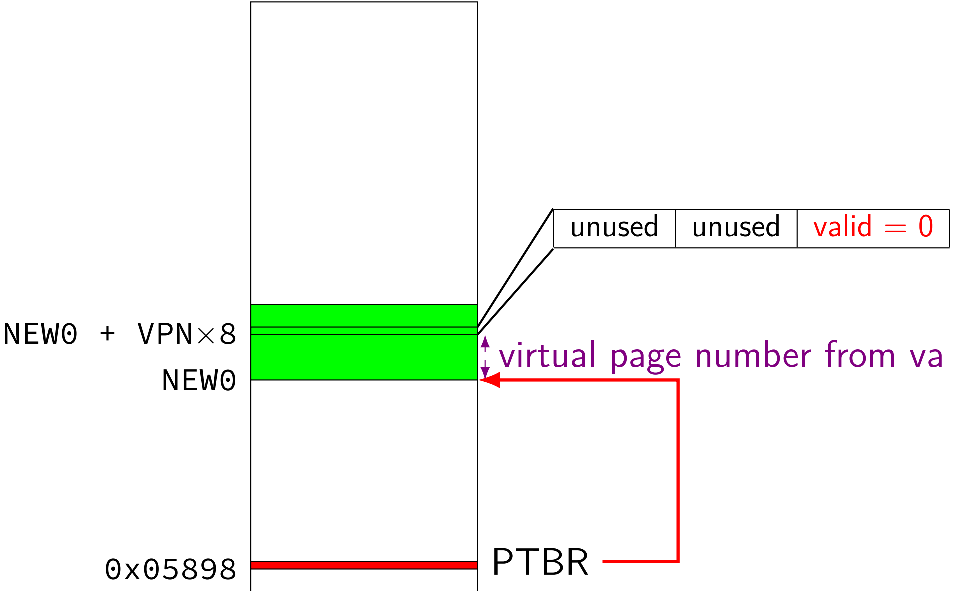
pa = translate(va) [LEVELS=1]



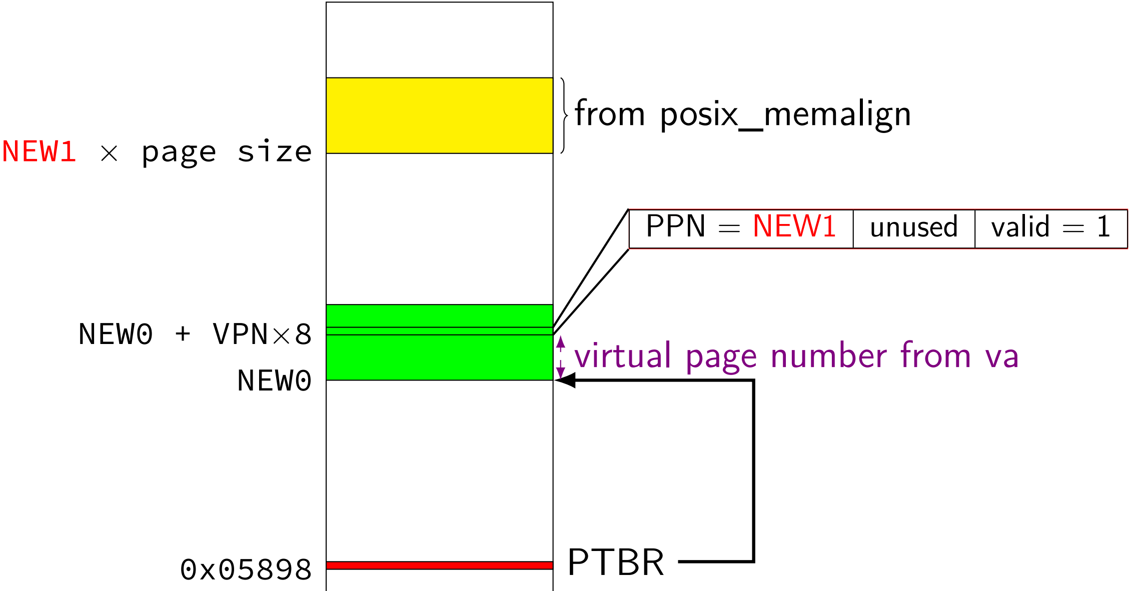
first page_allocate(va) [LEVELS=1]



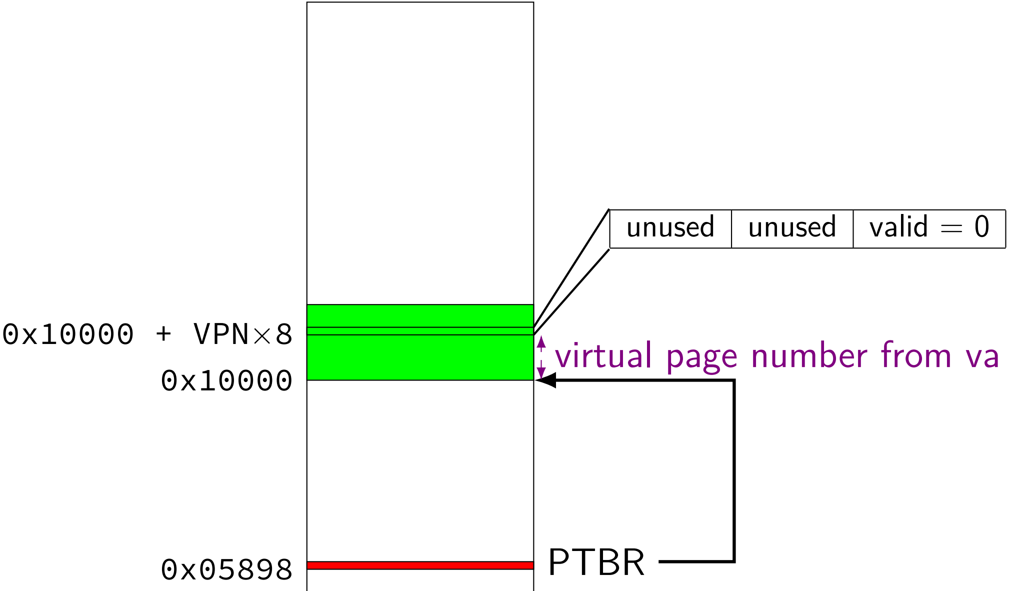
first page_allocate(va) [LEVELS=1]



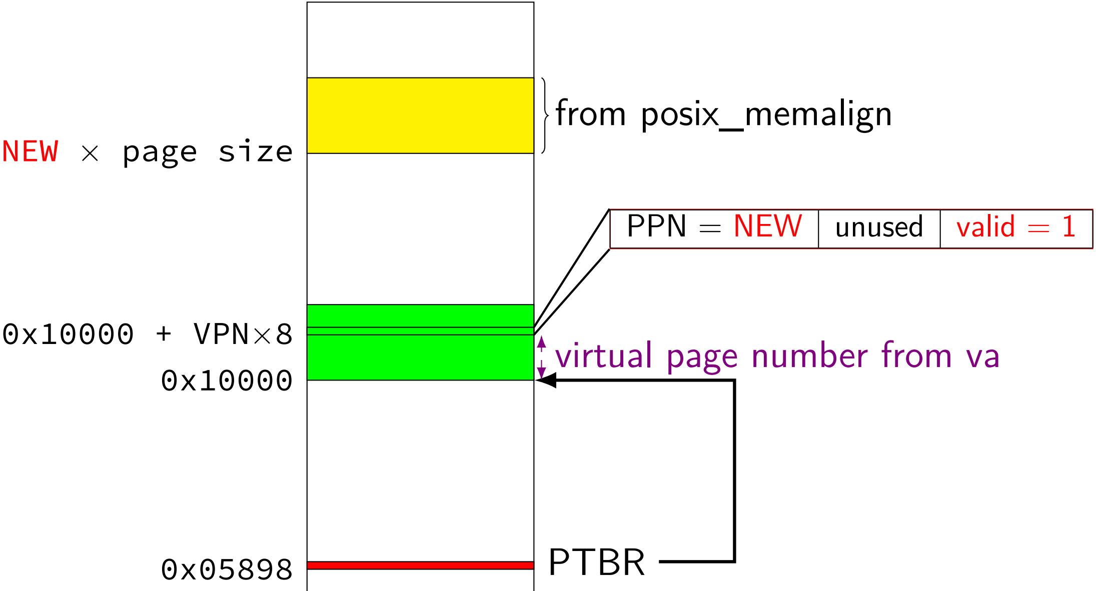
first page_allocate(va) [LEVELS=1]



page_allocate(va) [LEVELS=1]



page_allocate(va) [LEVELS=1]



NEW × page size

from posix_memalign

PPN = NEW	unused	valid = 1

0x10000 + VPN×8

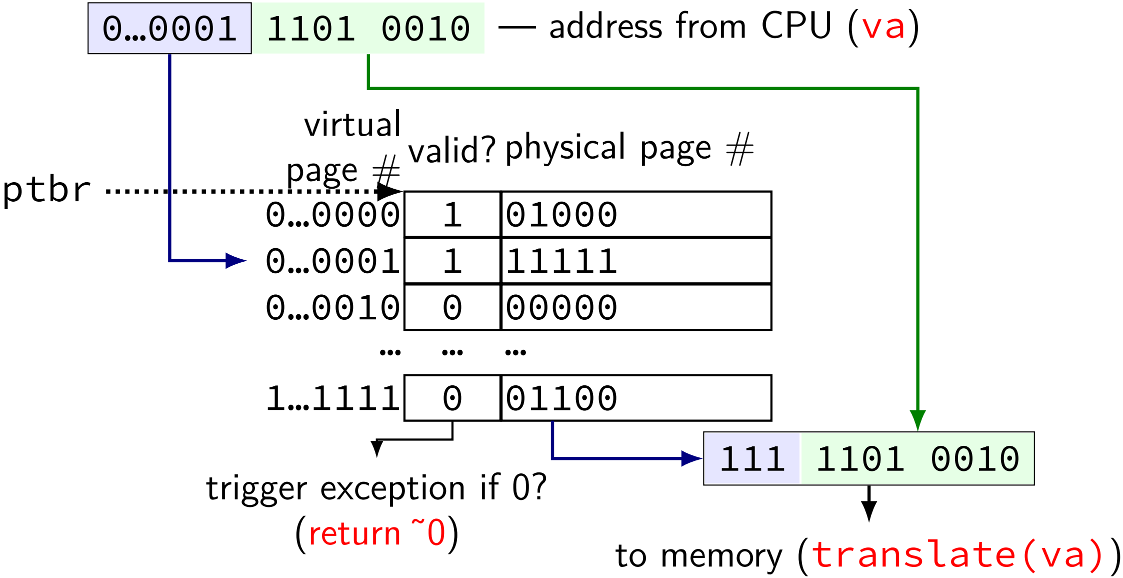
virtual page number from va

0x10000

0x05898

PTBR

page table lookup (and translate())



0…0001 1101 0010 — address from CPU (va)

ptbr

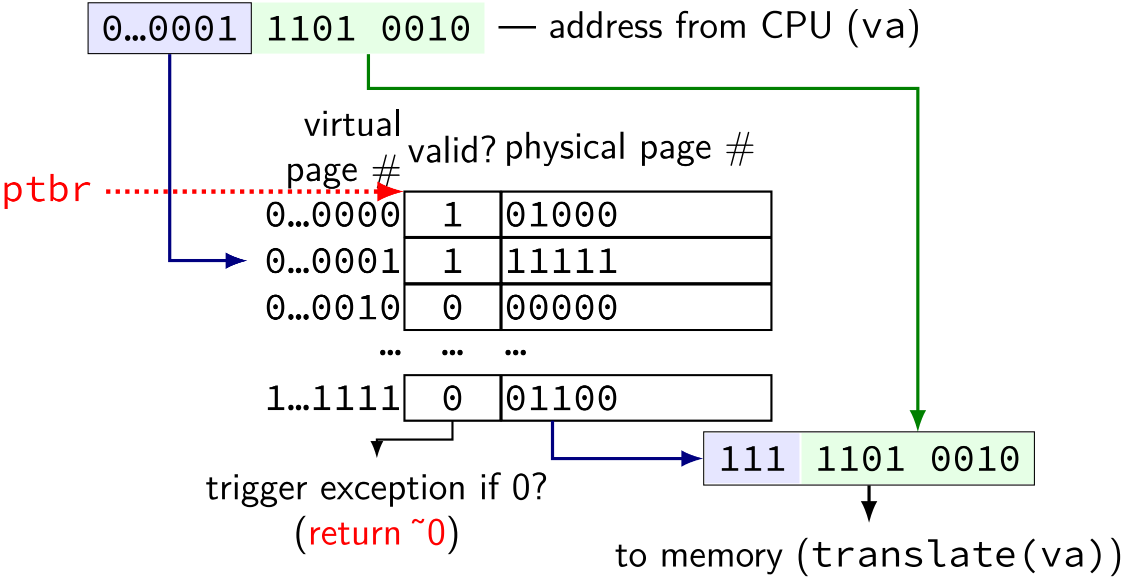
virtual page #	valid?	physical page #
0…0000	1	01000
0…0001	1	11111
0…0010	0	00000
…	…	…
1…1111	0	01100

trigger exception if 0?
(return ~0)

111 1101 0010

to memory (translate(va))

page table lookup (and translate())



`0…0001` `1101 0010` — address from CPU (va)

`ptbr`

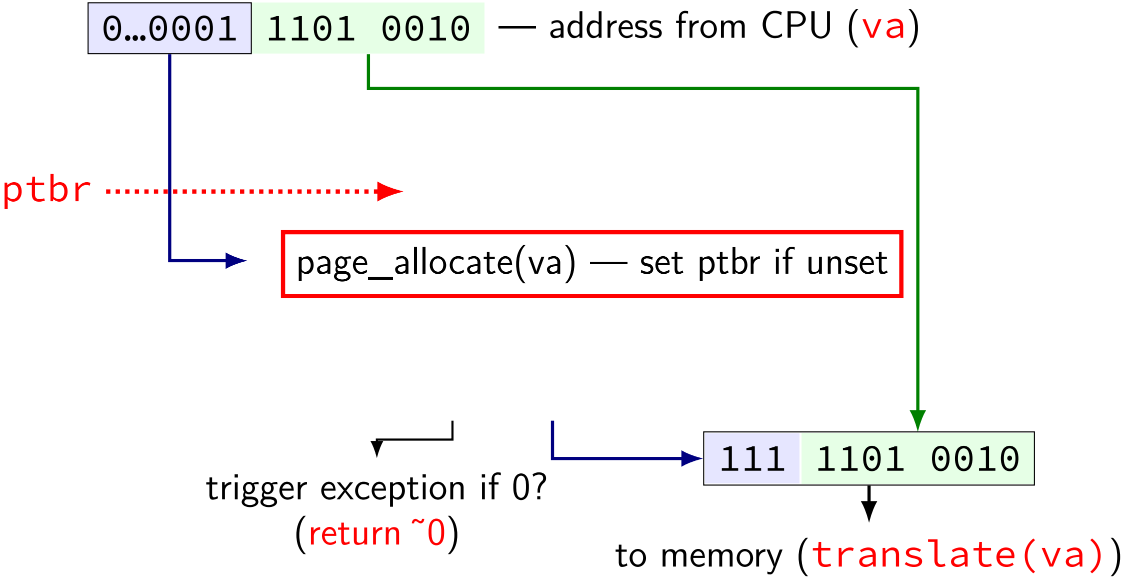
virtual page #	valid?	physical page #
0…0000	1	01000
0…0001	1	11111
0…0010	0	00000
…	…	…
1…1111	0	01100

trigger exception if 0?
(return ~0)

`111` `1101 0010`

to memory (`translate(va)`)

page table lookup (and allocate)



0…0001 1101 0010 — address from CPU (va)

ptbr

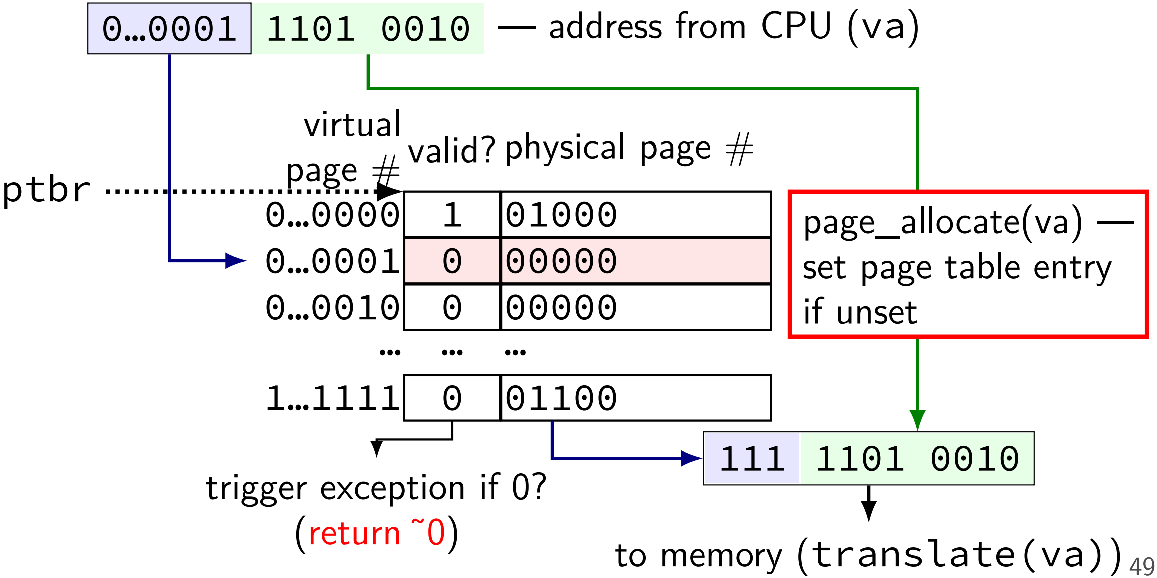
page_allocate(va) — set ptbr if unset

111 1101 0010

trigger exception if 0?
(return ~0)

to memory (translate(va))

page table lookup (and allocate)



0…0001 1101 0010 — address from CPU (va)

ptbr

virtual page #	valid?	physical page #
0…0000	1	01000
0…0001	0	00000
0…0010	0	00000
…	…	…
1…1111	0	01100

page_allocate(va) — set page table entry if unset

trigger exception if 0?
(return ~0)

111 1101 0010

to memory (translate(va))

exercise: 64-bit system

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

4096 byte pages

exercise: 64-bit system

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

4096 byte pages

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

exercise: 64-bit system

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

4096 byte pages

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

exercise: how large are physical page numbers?

exercise: 64-bit system

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

4096 byte pages

exercise: how many page table entries? (assuming page table like shown before) $2^{48}/2^{12} = 2^{36}$ entries

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

exercise: 64-bit system

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

4096 byte pages

exercise: how many page table entries? (assuming page table like shown before) $2^{48}/2^{12} = 2^{36}$ entries

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

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

- trick: power of two size makes table lookup faster

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

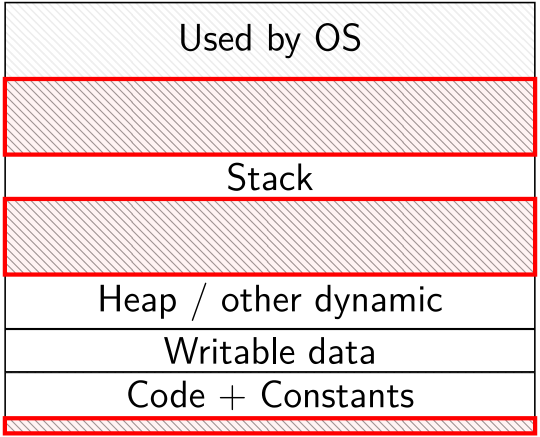
huge page tables

huge virtual address spaces!

impossible to store PTE for every page

how can we save space?

holes



most pages are invalid

saving space

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

use a data structure other than a flat array

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

options?

saving space

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

use a data structure other than a flat array

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

options?

hashtable

- actually used by some historical processors
- but never common

saving space

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

use a data structure other than a flat array

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

options?

hashtable

- actually used by some historical processors
- but never common

tree data structure

- but not quite a search tree

search tree tradeoffs

lookup usually implemented in hardware

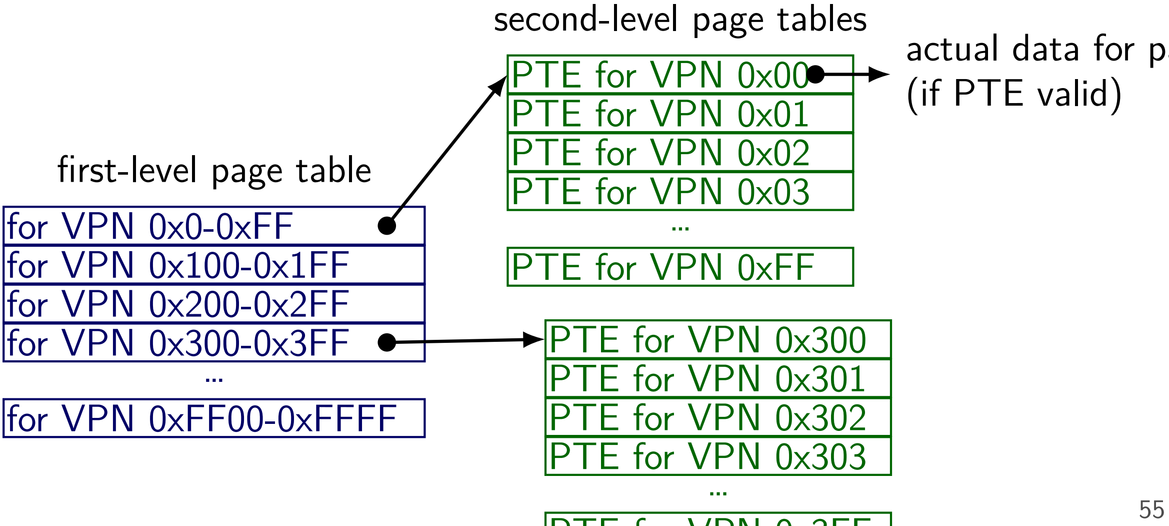
- lookup should be simple
- solution: lookup splits up address bits (no complex calculations)

lookup should not involve many memory accesses

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

two-level page tables

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



second-level page tables

actual data for p
(if PTE valid)

first-level page table

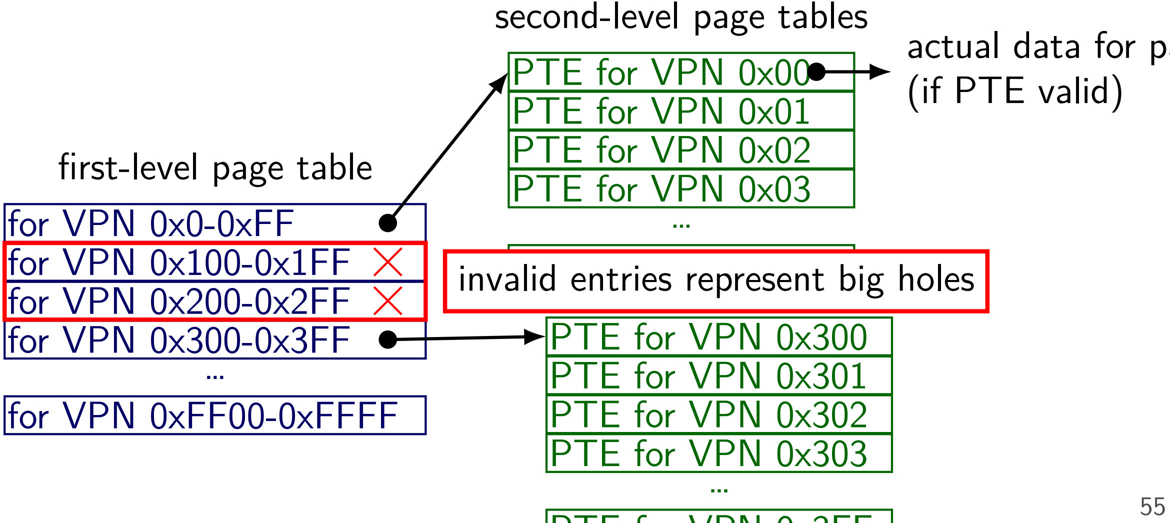
for VPN 0x0-0xFF
for VPN 0x100-0x1FF
for VPN 0x200-0x2FF
for VPN 0x300-0x3FF
...
for VPN 0xFF00-0xFFFF

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

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

two-level page tables

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



second-level page tables

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

actual data for p
(if PTE valid)

first-level page table

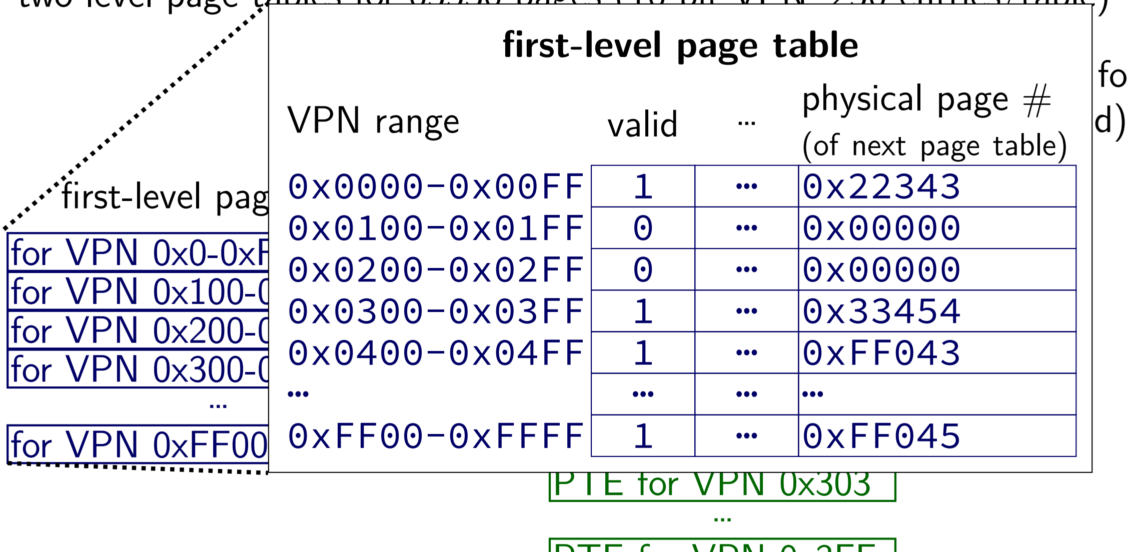
for VPN 0x0-0xFF
for VPN 0x100-0x1FF ✗
for VPN 0x200-0x2FF ✗
for VPN 0x300-0x3FF
...
for VPN 0xFF00-0xFFFF

invalid entries represent big holes

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

two-level page tables

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



first-level pag

for VPN 0x0-0xF

for VPN 0x100-0

for VPN 0x200-0

for VPN 0x300-0

...

for VPN 0xFF00

for p

d)

PTE for VPN 0x303

...

PTE for VPN 0x3FF

first-level page table

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

two-level page tables

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

first-level pag

for VPN 0x0-0xF

for VPN 0x100-0

for VPN 0x200-0

for VPN 0x300-0

…

for VPN 0xFF00

for p

d)

first-level page table

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

PTE for VPN 0x303

…

two-level page tables

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

first-level pag

for VPN 0x0-0xF

for VPN 0x100-0

for VPN 0x200-0

for VPN 0x300-0

...

for VPN 0xFF00

for p

d)

first-level page table

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

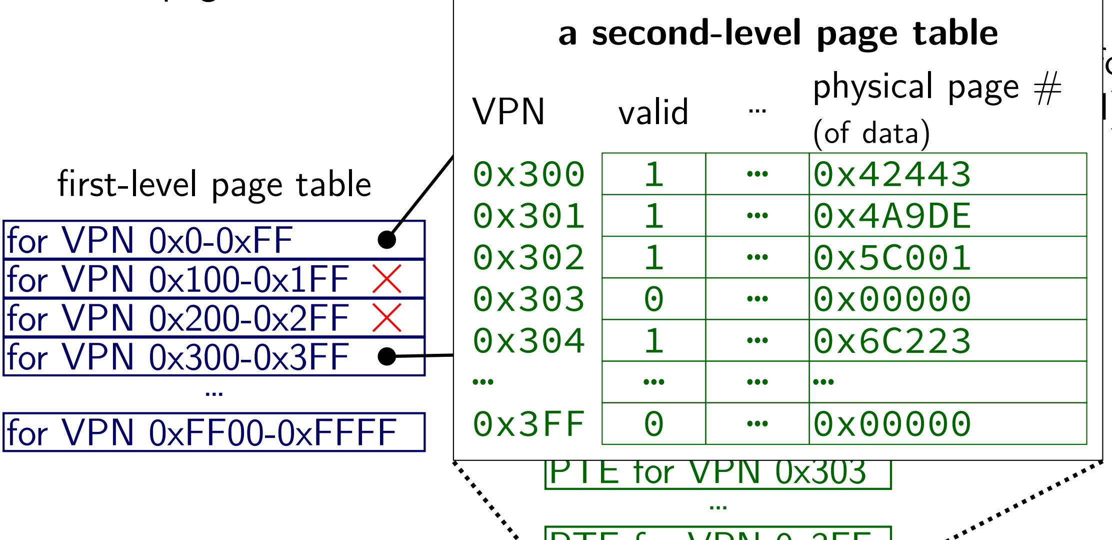
PTE for VPN 0x303

...

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 ●

…

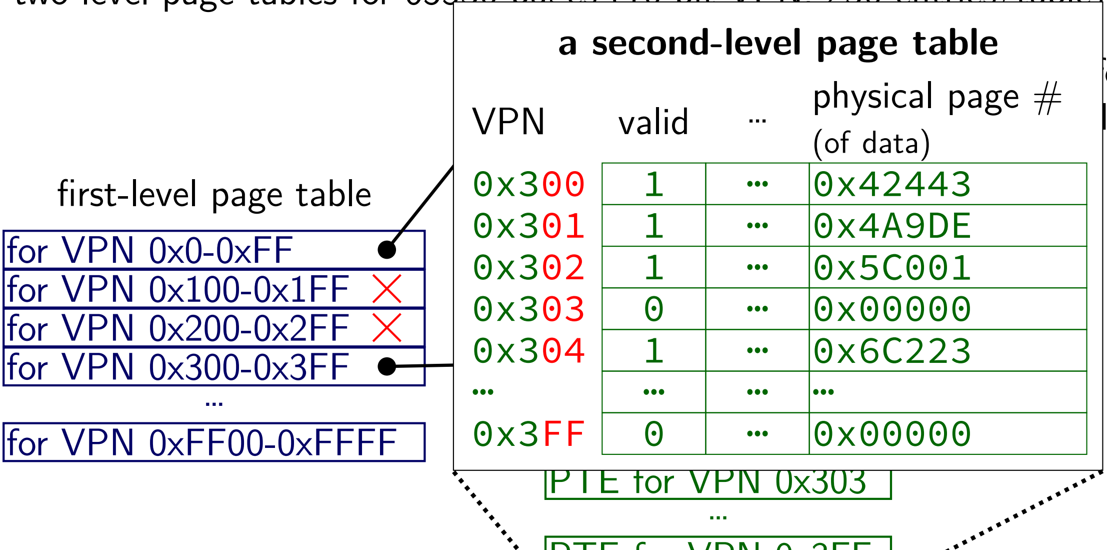
for VPN 0xFF00-0xFFFF

a second-level page table

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

two-level page tables

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



first-level page table

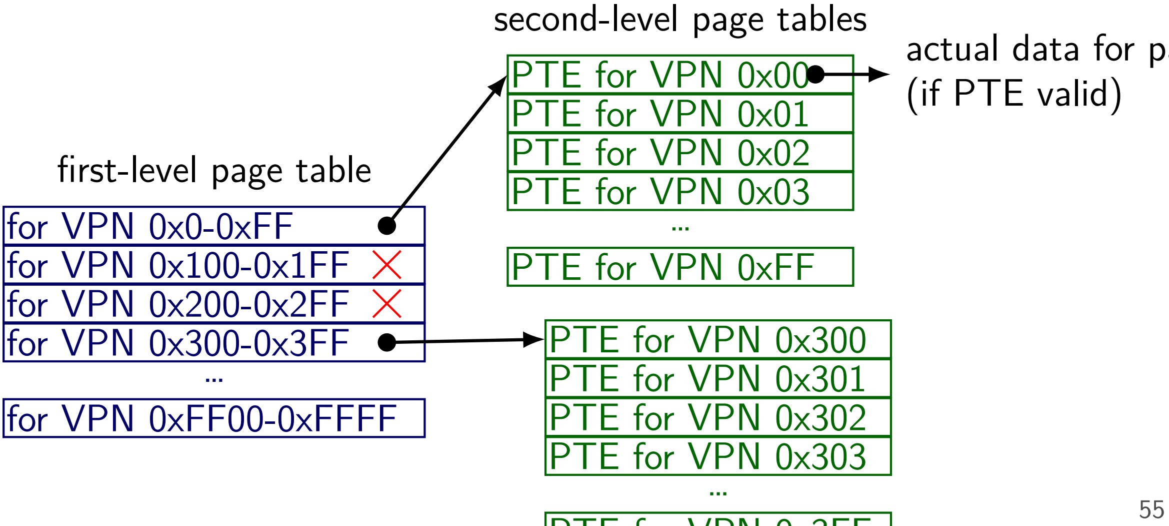
- for VPN 0x0-0xFF ●
- for VPN 0x100-0x1FF ✗
- for VPN 0x200-0x2FF ✗
- for VPN 0x300-0x3FF ●
- ...
- for VPN 0xFF00-0xFFFF

a second-level page table

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

two-level page tables

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



second-level page tables

PTE for VPN 0x00

PTE for VPN 0x01

PTE for VPN 0x02

PTE for VPN 0x03

...

PTE for VPN 0xFF

actual data for p

(if PTE valid)

first-level page table

for VPN 0x0-0xFF

for VPN 0x100-0x1FF ✗

for VPN 0x200-0x2FF ✗

for VPN 0x300-0x3FF

...

for VPN 0xFF00-0xFFFF

PTE for VPN 0x300

PTE for VPN 0x301

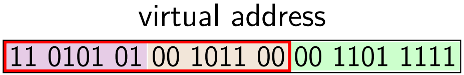
PTE for VPN 0x302

PTE for VPN 0x303

...

PTE for VPN 0x3FF

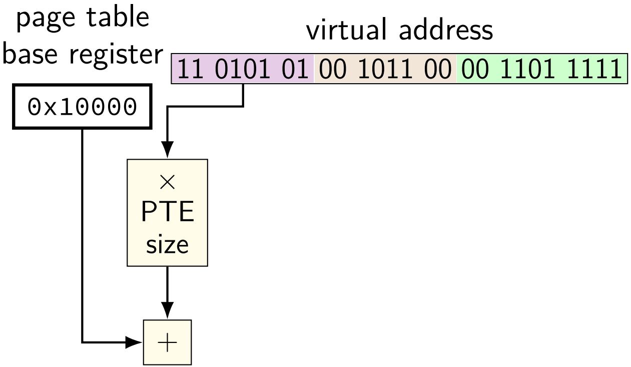
two-level page table lookup



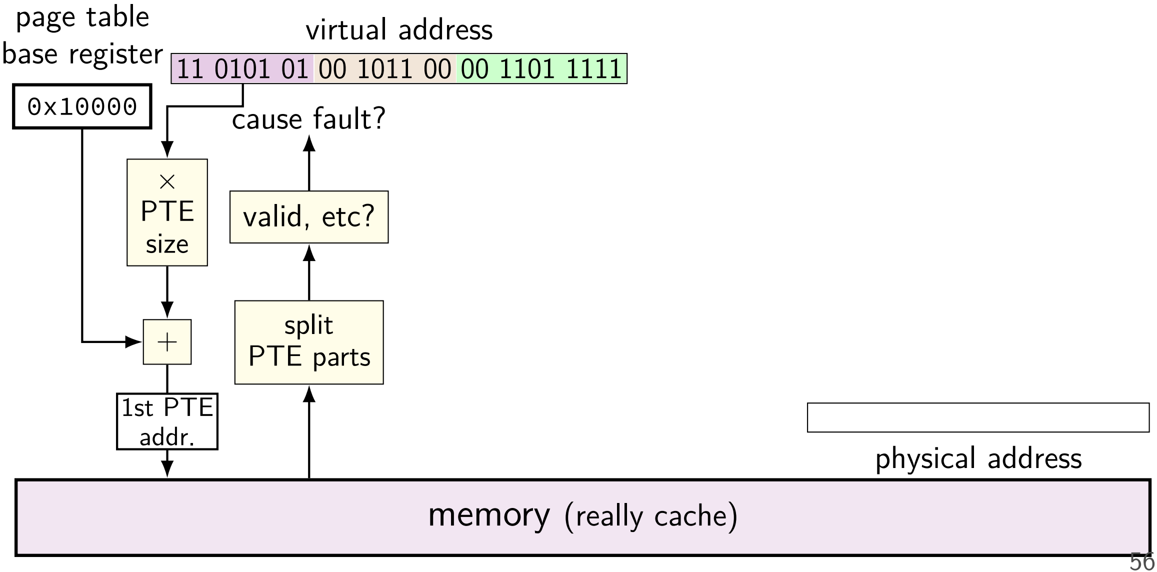
VPN — split into two parts (one per level)

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

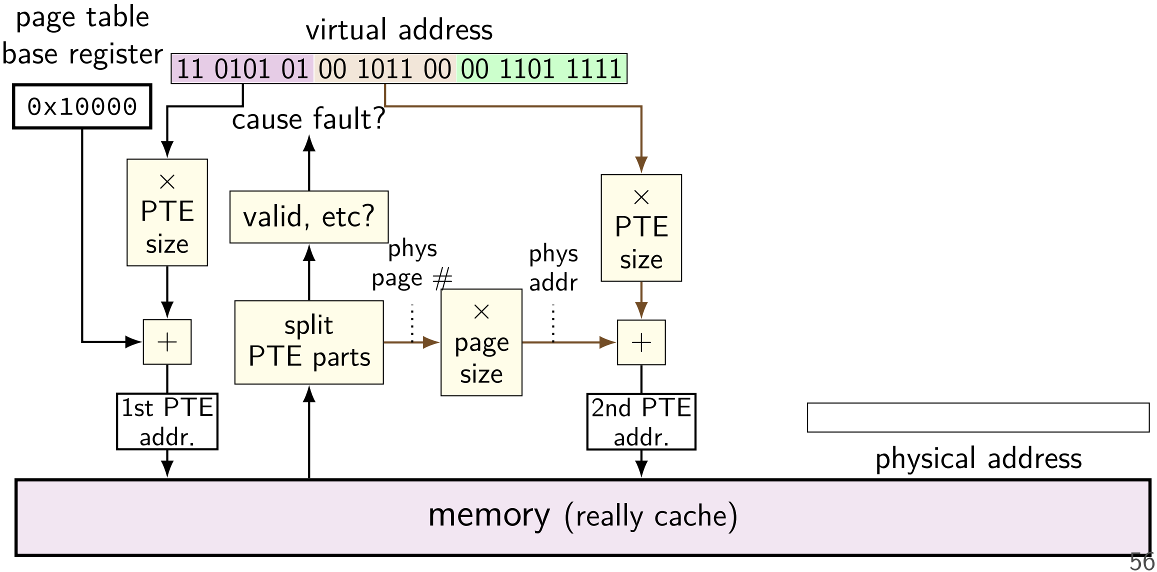
two-level page table lookup



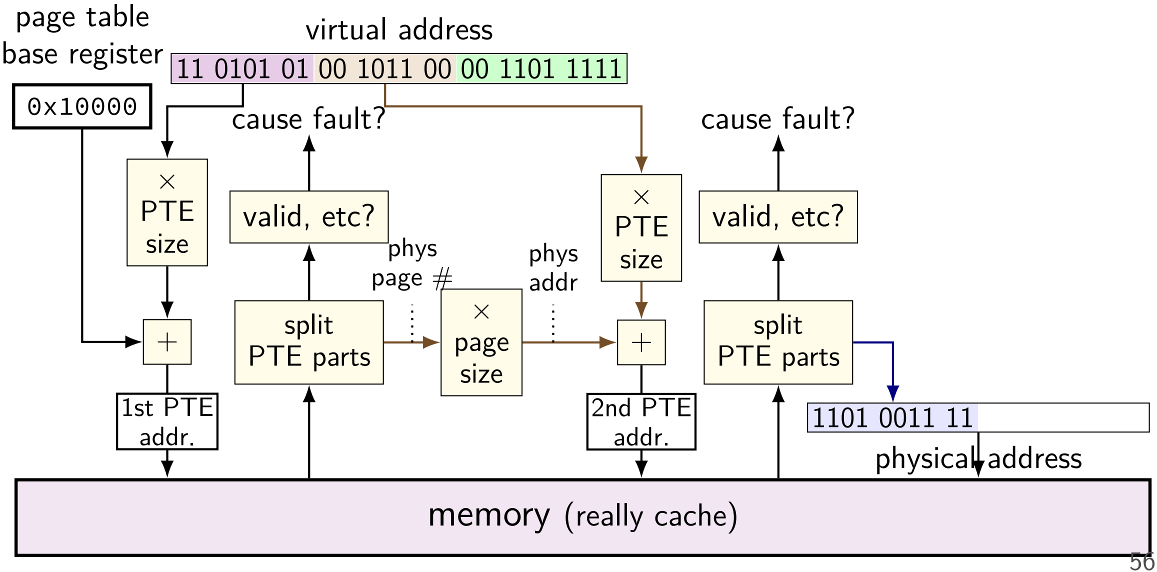
two-level page table lookup



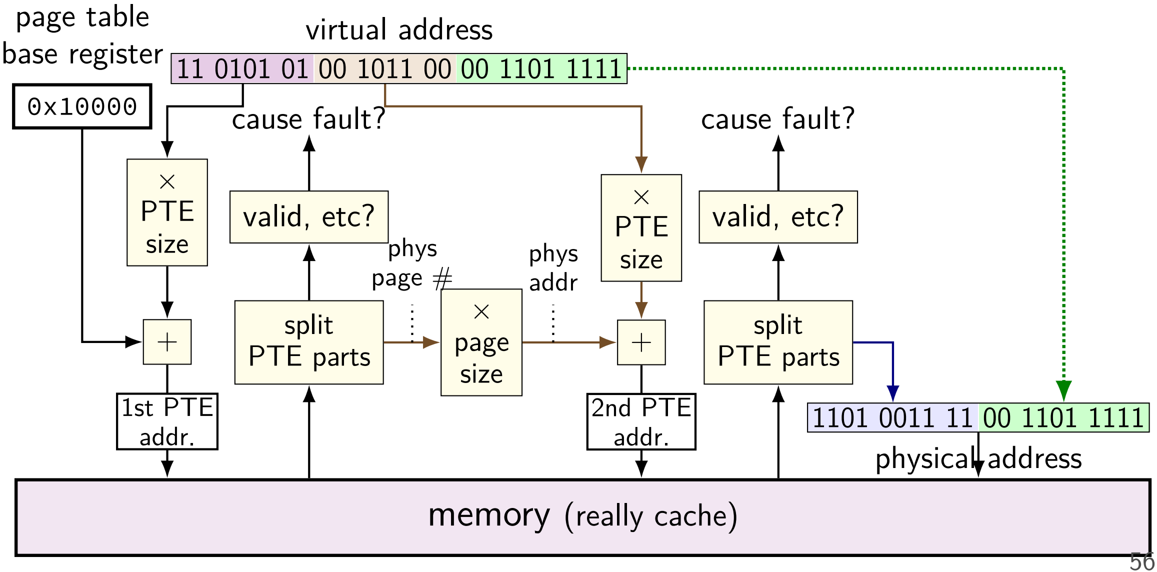
two-level page table lookup



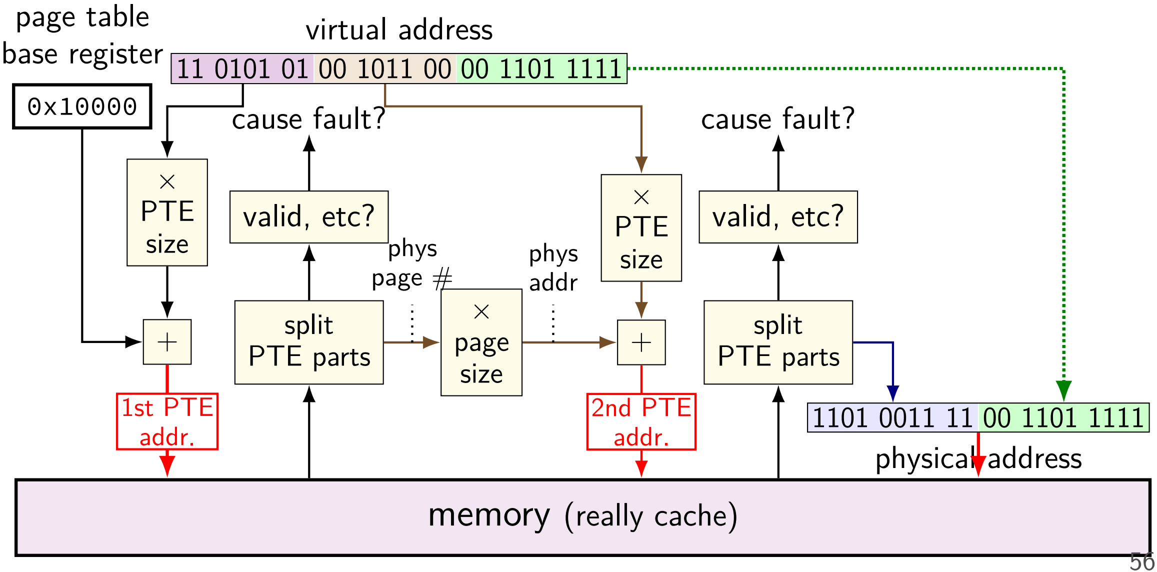
two-level page table lookup



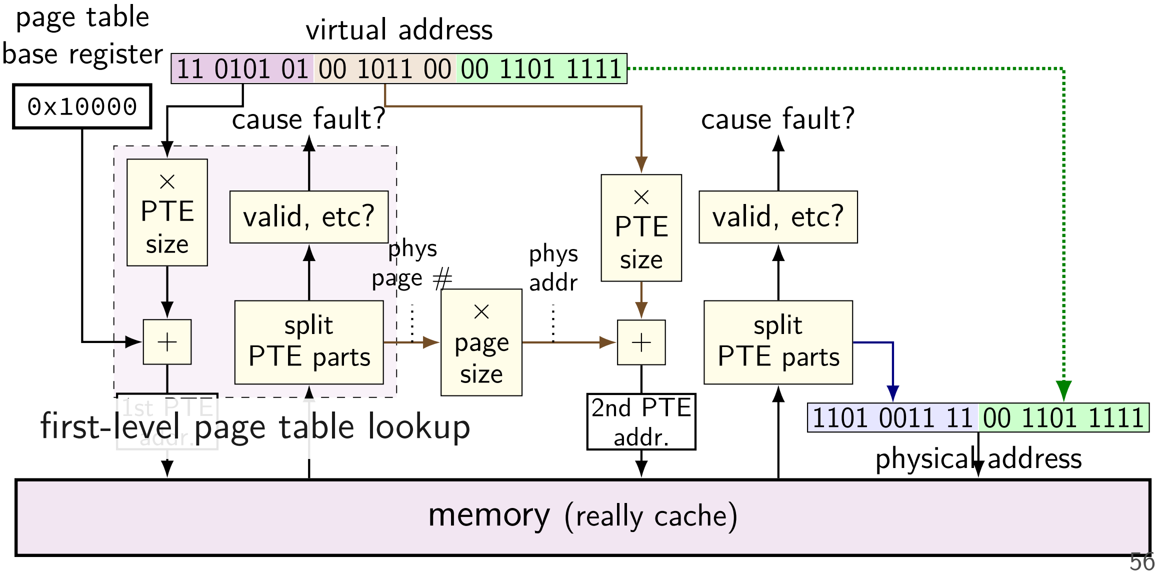
two-level page table lookup



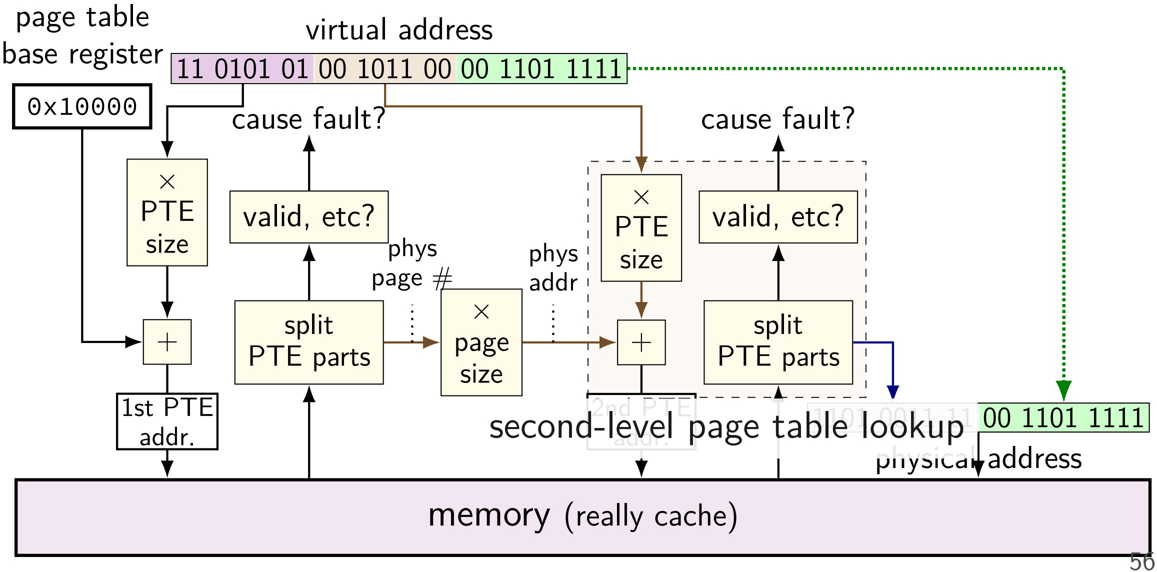
two-level page table lookup



two-level page table lookup



two-level page table lookup



page table base register

0x10000

virtual address

11 0101 01 00 1011 00 00 1101 1111

cause fault?

valid, etc?

split PTE parts

× PTE size

+

1st PTE addr.

phys page #

× page size

phys addr

cause fault?

× PTE size

valid, etc?

+

split PTE parts

2nd PTE addr.

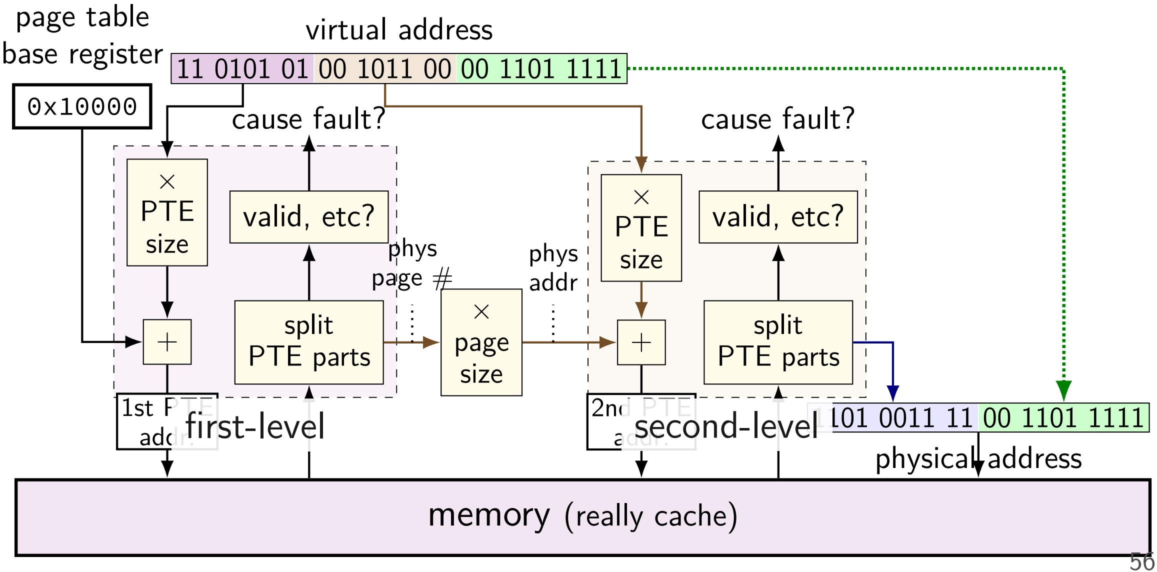
second-level page table lookup

1101 0011 11 00 1101 1111

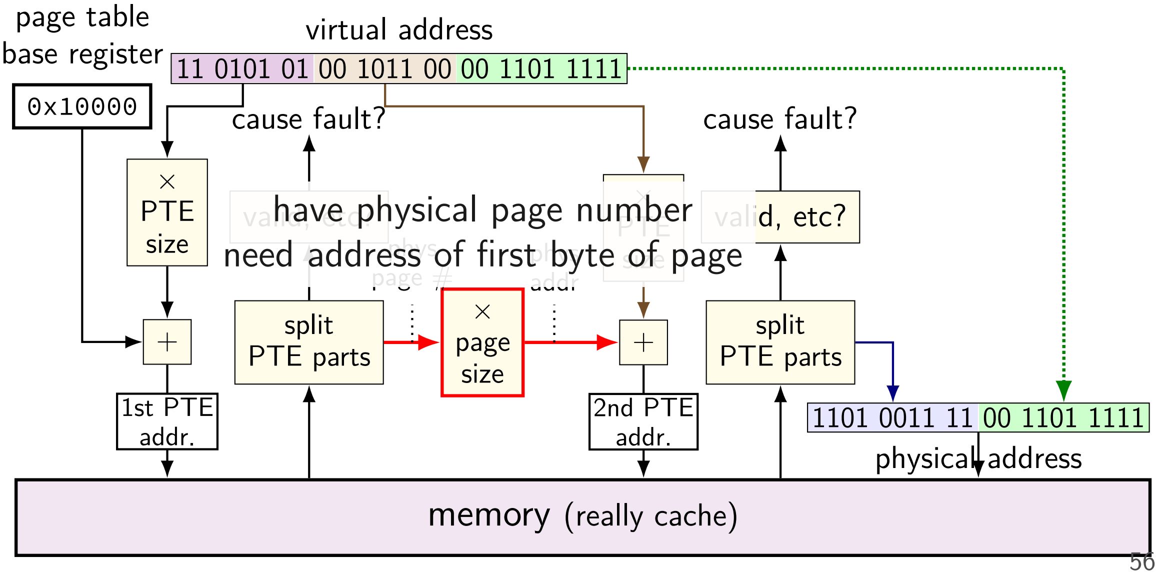
physical address

memory (really cache)

two-level page table lookup

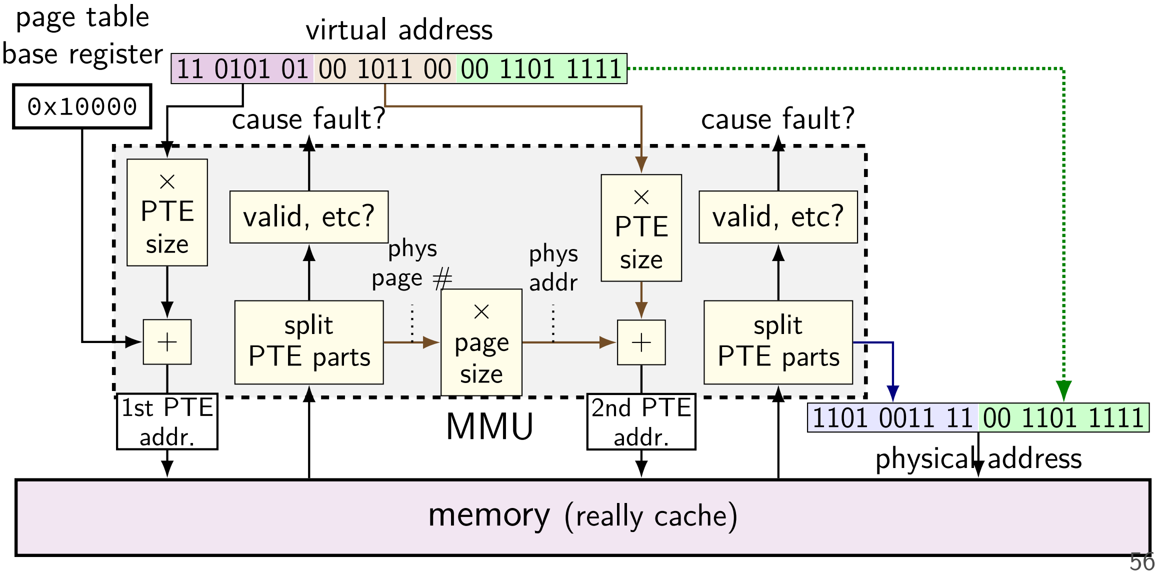


two-level page table lookup

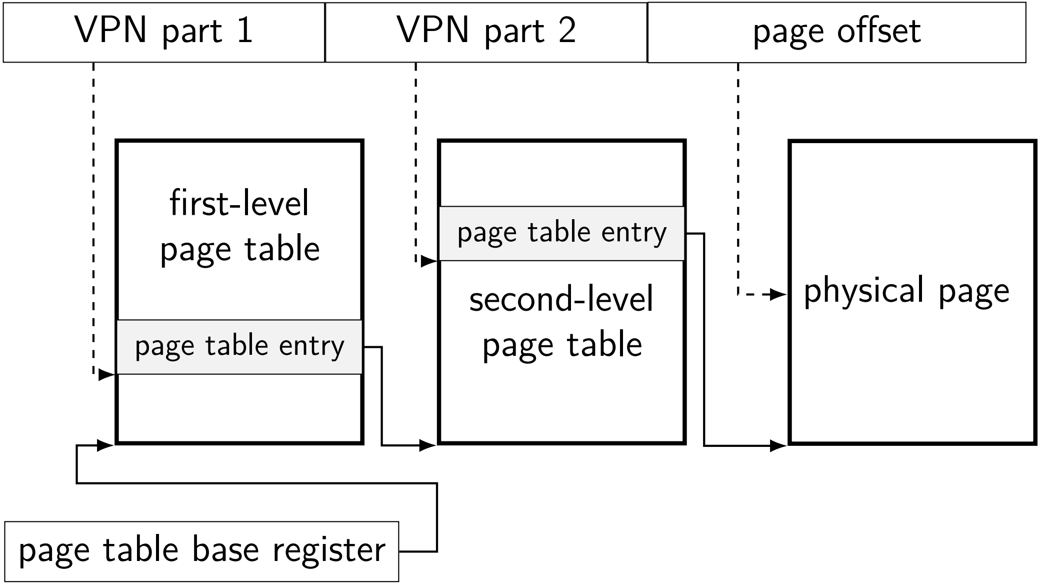


have physical page number
need address of first byte of page

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

note on VPN splitting

indexes used for lookup parts of the virtual page number

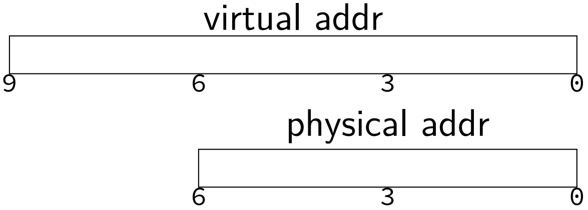
(there are not multiple VPNs)

assignment

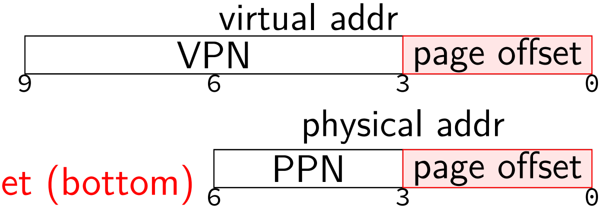
2-level splitting

9-bit virtual address

6-bit physical address



2-level splitting



9-bit virtual address

6-bit physical address

8-byte pages → 3-bit page offset (bottom)

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

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

2-level splitting

9-bit virtual address

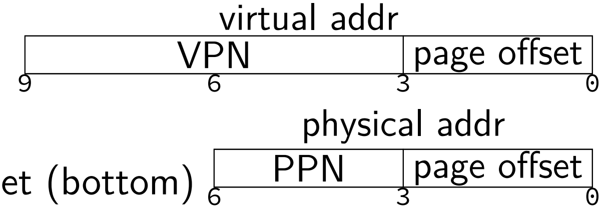
6-bit physical address

8-byte pages → 3-bit page offset (bottom)

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

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

1 page page tables w/ 1 byte entry → 8 entry PTs

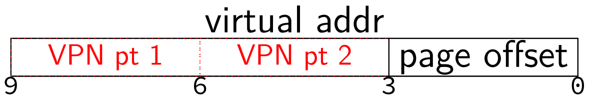


page table (either level)

	valid?	PPN
0		
1		
2		
…	…	…
7		

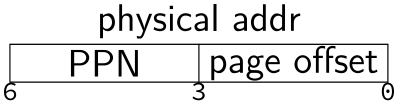
2-level splitting

9-bit virtual address



6-bit physical address

8-byte pages → 3-bit page offset (bottom)



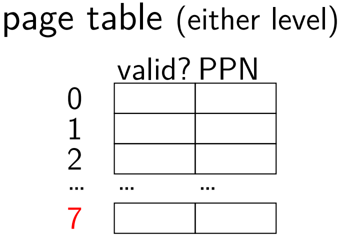
9-bit VA: 6 bit VPN + 3 bit PO

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

1 page page tables w/ 1 byte entry → 8 entry PTs

8 entry page tables → 3-bit VPN parts

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



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 `0x129`

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	F4 A5 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	AC DC DC 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 `0x129`

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	F4 A5 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	AC DC DC 0C

`0x129` = `1 0010 1001`

`0x20` + `0x4` ×`1` = `0x24`

PTE 1 value:

`0xF4` = `1111 0100`

PPN `111`, valid `1`

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register `0x20`; translate virtual address `0x129`

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	F4 A5 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	AC DC DC 0C

`0x129` = `1 0010 1001`

`0x20` + `0x4` ×`1` = `0x24`

PTE 1 value:

`0xF4` = `1111 0100`

PPN `111`, valid `1`

PTE 2 addr:

`111 000` + `101` × `1` = `0x3D`

PTE 2 value: `0xDC`

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 `0x129`

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	F4 A5 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	AC DC DC 0C

0x129 = 1 0010 1001

0x20 + 0x4 ×1 = 0x24

PTE 1 value:

0xF4 = 1111 0100

PPN 111, valid 1

PTE 2 addr:

111 000 + 101 × 1 = 0x3D

PTE 2 value: 0xDC

PPN 110; valid 1

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

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register `0x20`; translate virtual address `0x129`

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	F4 A5 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	AC DC DC 0C

0x129 = 1 0010 1001

0x20 + 0x4 ×1 = 0x24

PTE 1 value:

0xF4 = 1111 0100

PPN 111, valid 1

PTE 2 addr:

111 000 + 101 × 1 = 0x3D

PTE 2 value: 0xDC

PPN 110; valid 1

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

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register `0x20`; translate virtual address `0x129`

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	F4 A5 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	AC DC DC 0C

0x129 = 1 0010 1001

0x20 + 0x4 ×1 = 0x24

PTE 1 value:

0xF4 = 1111 0100

PPN 111, valid 1

PTE 2 addr:

111 000 + 101 × 1 = 0x3D

PTE 2 value: 0xDC

PPN 110; valid 1

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

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

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

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

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

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

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

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

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

`0x0F3` = `011 111 011`

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

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

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

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

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

`111 011` = `0x3B` → `0x0C`

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

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

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

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

0x0F3 = 011 111 011

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

PTE 1: 0xBB at 0x0B

PTE 1: PPN 101 (5) valid 1

PTE 2: 0xF0 at 0x2F

PTE 2: PPN 111 (7) valid 1

111 011 = 0x3B → 0x0C

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

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

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

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

0x0F3 = 011 111 011

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

PTE 1: 0xBB at 0x0B

PTE 1: PPN 101 (5) valid 1

PTE 2: 0xF0 at 0x2F

PTE 2: PPN 111 (7) valid 1

111 011 = 0x3B → 0x0C

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

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

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

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

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

2-level exercise (2)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

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

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

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

2-level exercise (2)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

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

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

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

`0x109` = `100 011 001`

(PTE 1 at:
0x10 + PTE size times 4 (100))

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

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

(second table at:
0 (000) times page size = 0x00)

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

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

`001 001` = `0x09` → `0x99`

2-level exercise (2)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

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

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

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

`0x109` = `100 011 001`

(PTE 1 at:

0x10 + PTE size times 4 (100))

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

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

(second table at:

0 (000) times page size = 0x00)

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

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

`001 001` = `0x09` → `0x99`

2-level exercise (2)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

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

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

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

`0x109` = `100 011 001`

(PTE 1 at:

0x10 + PTE size times 4 (100))

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

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

(second table at:

0 (000) times page size = 0x00)

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

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

`001 001` = `0x09` → `0x99`

2-level exercise (2)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

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

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

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

`0x109` = `100 011 001`

(PTE 1 at:

0x10 + PTE size times 4 (100))

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

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

(second table at:

0 (000) times page size = 0x00)

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

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

`001 001` = `0x09` → `0x99`

2-level exercise (3)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

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

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

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

2-level exercise (3)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

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

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

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

`0x0F3` = `000 001 011`

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

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

page fault!

2-level exercise (3)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

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

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

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

`0x0F3` = `000 001 011`

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

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

page fault!

2-level exercise (4)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

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

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

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

2-level exercise (4)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

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

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

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

`0x1CB` = `111 001 011`

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

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

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

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

page fault!

2-level exercise (4)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

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

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

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

0x1CB = 111 001 011

PTE 1: 0xFF at 0x0F

PTE 1: PPN 111 (7) valid 1

PTE 2: 0x0C at 0x39

PTE 2: PPN 000 (0) valid 0

page fault!

2-level exercise (4)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

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

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

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

`0x1CB` = `111 001 011`

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

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

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

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

page fault!

2-level exercise (5)

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

page tables 1 page; PTE 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

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

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

AC BC

PTE 1: PPN `10` valid `1`

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

`EF F0`

PTE 2: PPN `11` valid `1`

`11 0110` $=$ `0x36` → `DB`

2-level exercise (5)

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

page tables 1 page; PTE 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

`0x376` = 110 111 0110

PTE 1: 0x10 + 6×2 = `0x1C`:

AC BC

PTE 1: PPN 10 valid 1

PTE 2: 0x20 + 7×2 = `0x2E`:

EF F0

PTE 2: PPN 11 valid 1

11 0110 = `0x36` → DB

2-level exercise (5)

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

page tables 1 page; PTE 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

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

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

AC BC

PTE 1: PPN `10` valid `1`

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

EF F0

PTE 2: PPN `11` valid `1`

`11 0110` $=$ `0x36` → DB

2-level exercise (5)

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

page tables 1 page; PTE 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

`0x376` = `110 111 0110`

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

AC BC

PTE 1: PPN `10` valid 1

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

EF F0

PTE 2: PPN `11` valid 1

`11 0110` = `0x36` → DB

2-level exercise (5)

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

page tables 1 page; PTE 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

0x376 = 110 111 0110

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

AC BC

PTE 1: PPN 10 valid 1

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

EF F0

PTE 2: PPN 11 valid 1

11 0110 = 0x36 → DB

2-level exercise (5)

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

page tables 1 page; PTE 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

`0x376` = `110 111 0110`

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

`AC BC`

PTE 1: PPN `10` valid `1`

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

`EF F0`

PTE 2: PPN `11` valid `1`

`11 0110` = `0x36` → `DB`

2-level exercise (5)

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

page tables 1 page; PTE 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

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

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

`AC BC`

PTE 1: PPN `10` valid `1`

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

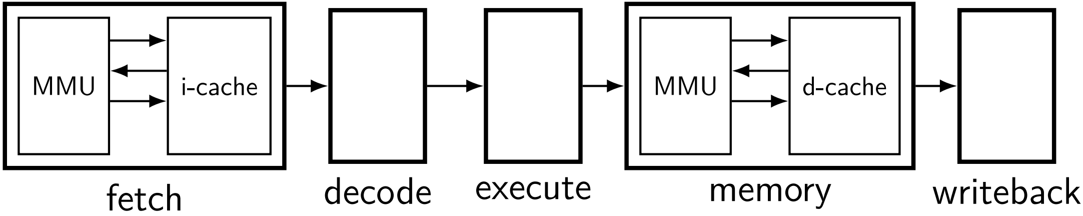
`EF F0`

PTE 2: PPN `11` valid `1`

`11 0110` $=$ `0x36` → `DB`

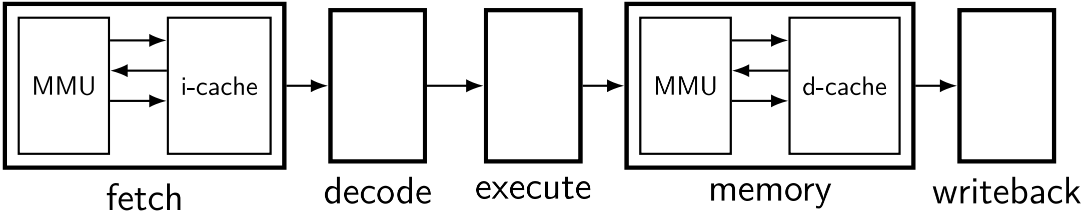
backup slides

MMUs in the pipeline



up to four memory accesses per instruction

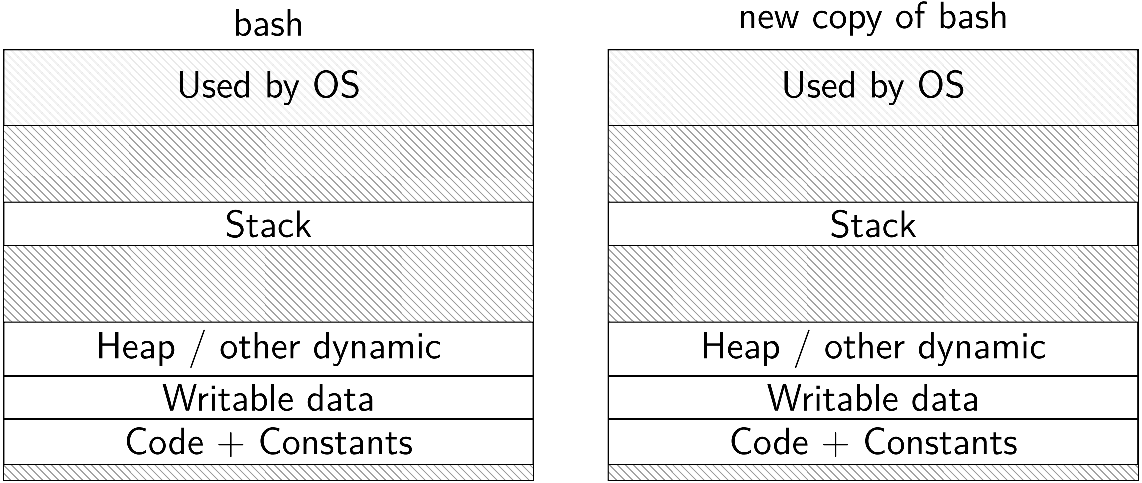
MMUs in the pipeline



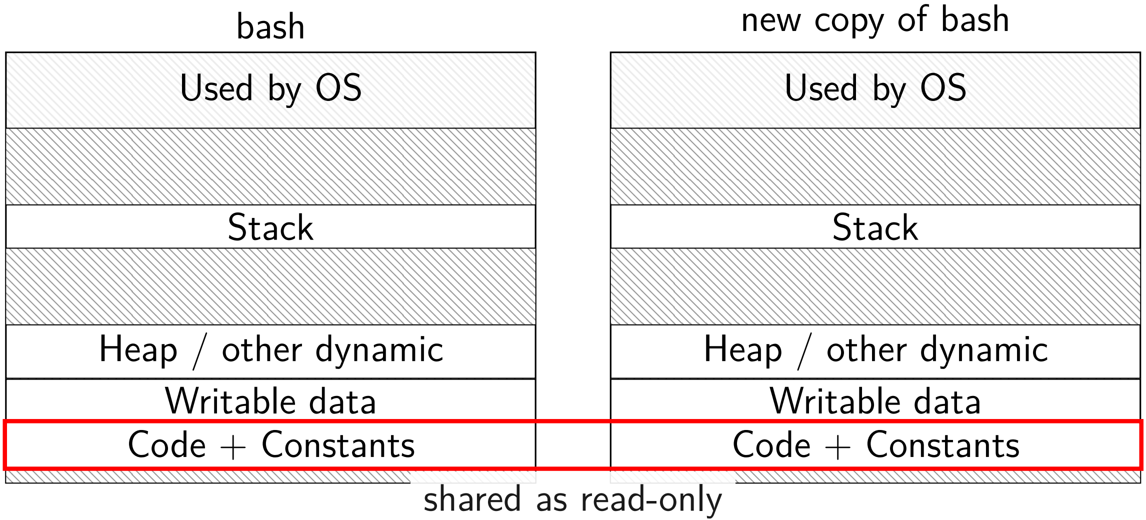
up to four memory accesses per instruction

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

do we really need a complete copy?



do we really need a complete copy?



bash

Used by OS

Stack

Heap / other dynamic

Writable data

Code + Constants

new copy of bash

Used by OS

Stack

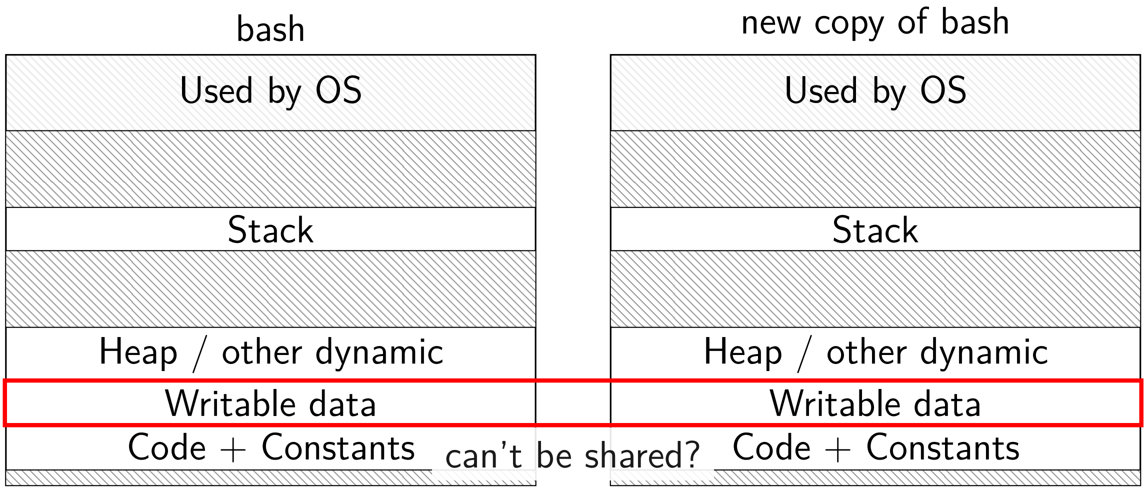
Heap / other dynamic

Writable data

Code + Constants

shared as read-only

do we really need a complete copy?



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

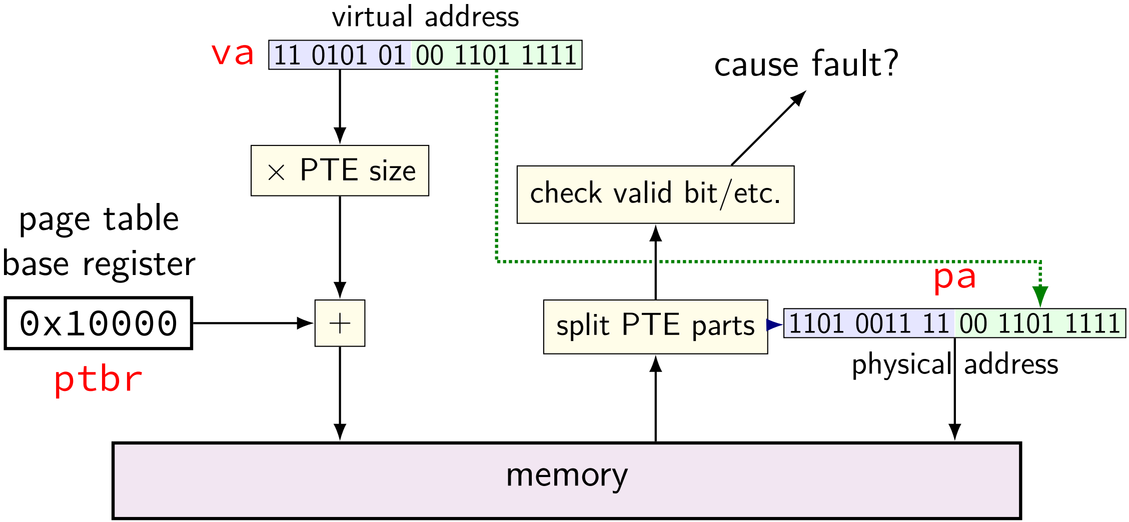
copy-on-write and page tables

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
…	…	…	…

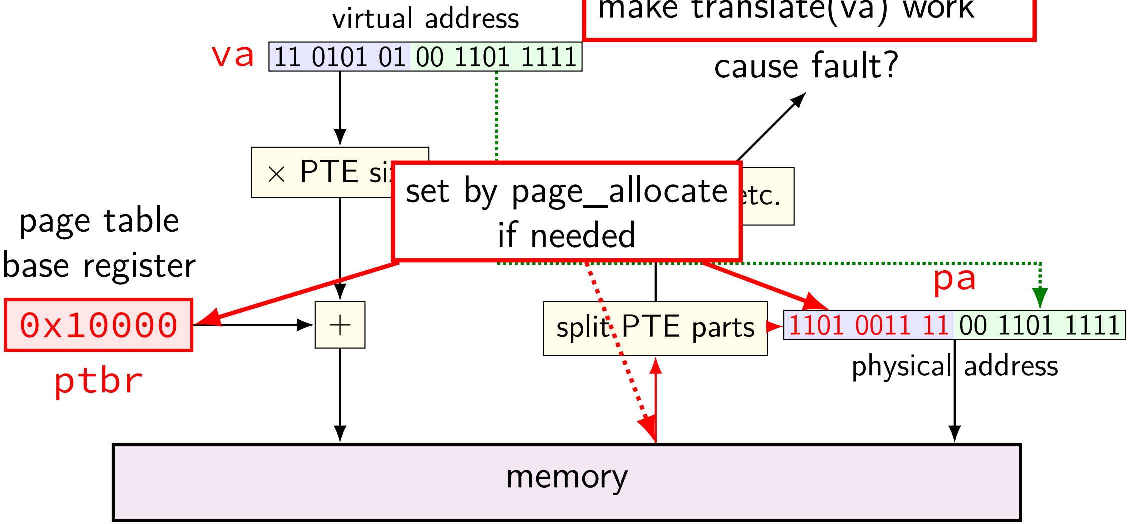
after allocating a copy, OS reruns the write instruction

pa=translate(va)



pa=translate(va)

page_allocate(va) needs to make translate(va) work



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 writes and writes: hundreds of microseconds

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

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

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

SSD writes and writes: hundreds of microseconds

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

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

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

SSD writes and writes: hundreds of microseconds

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

HDD/SDDs are slow

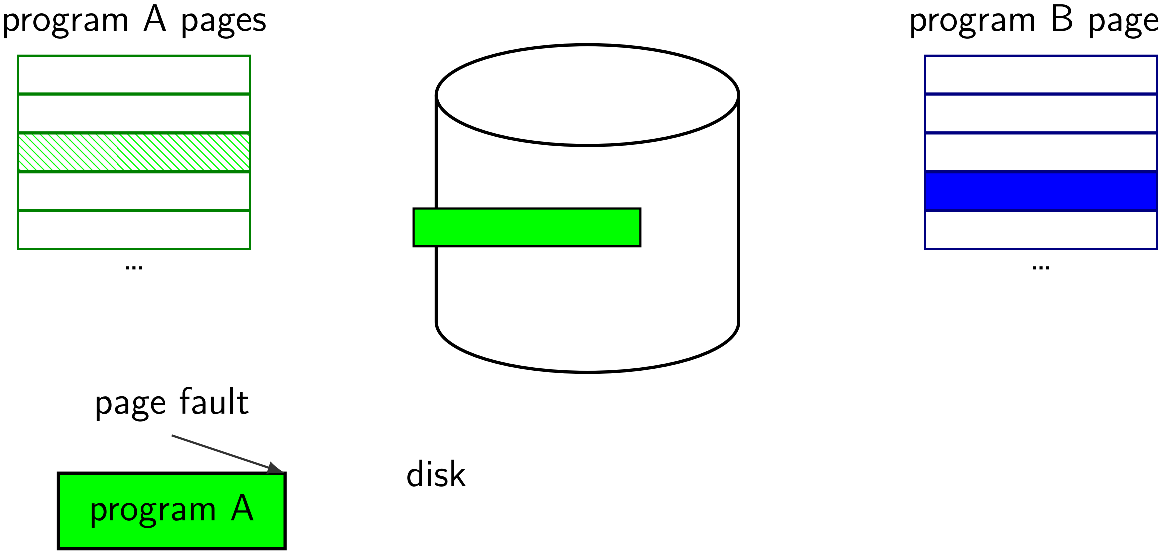
HDD reads and writes: milliseconds to tens of milliseconds

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

SSD writes and writes: hundreds of microseconds

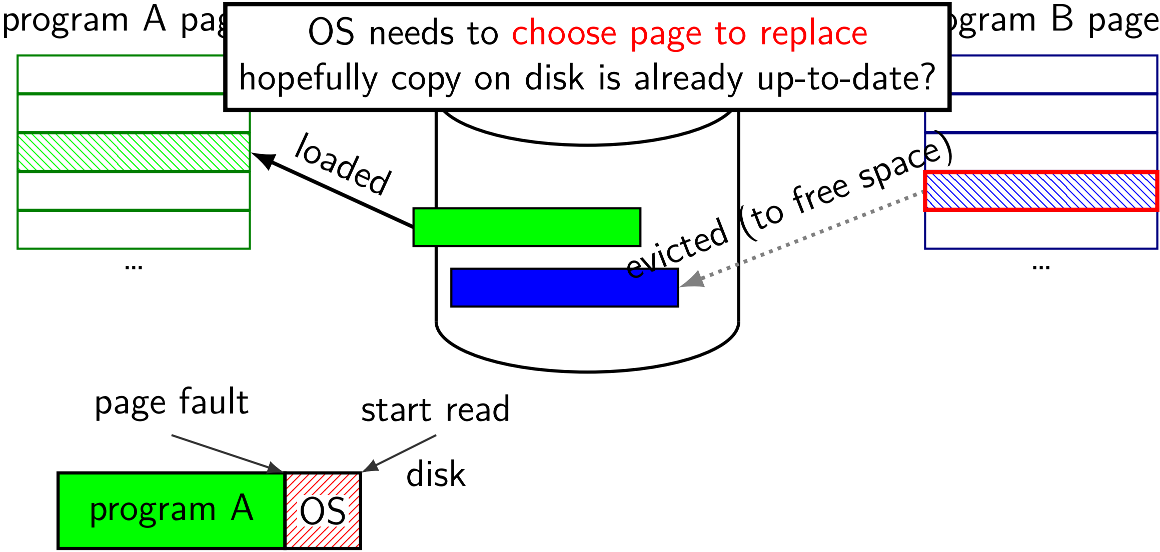
- designed for writes/reads of kilobytes (not much smaller)

swapping timeline

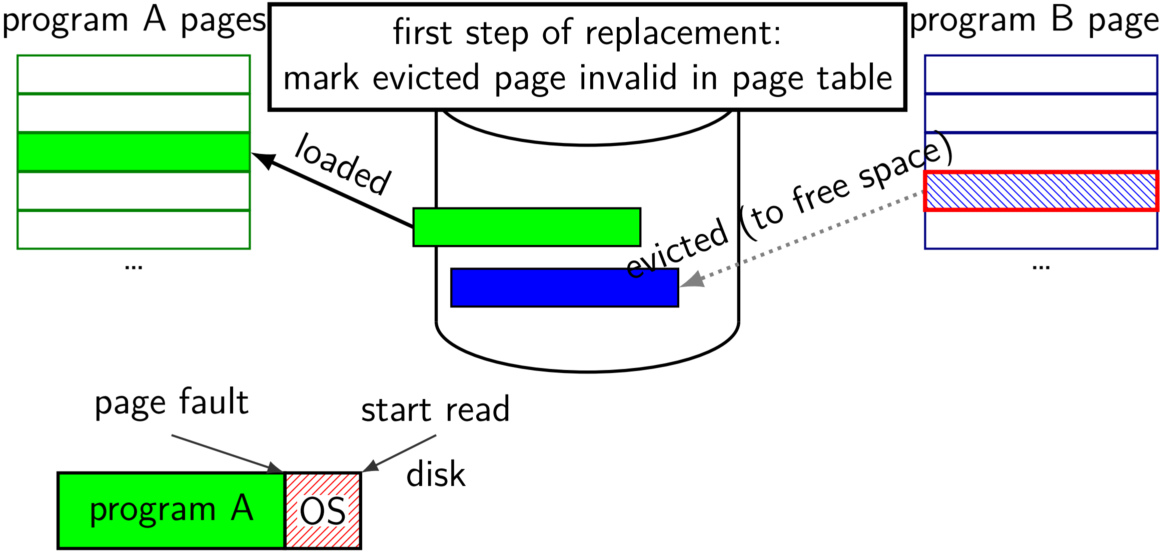


swapping timeline

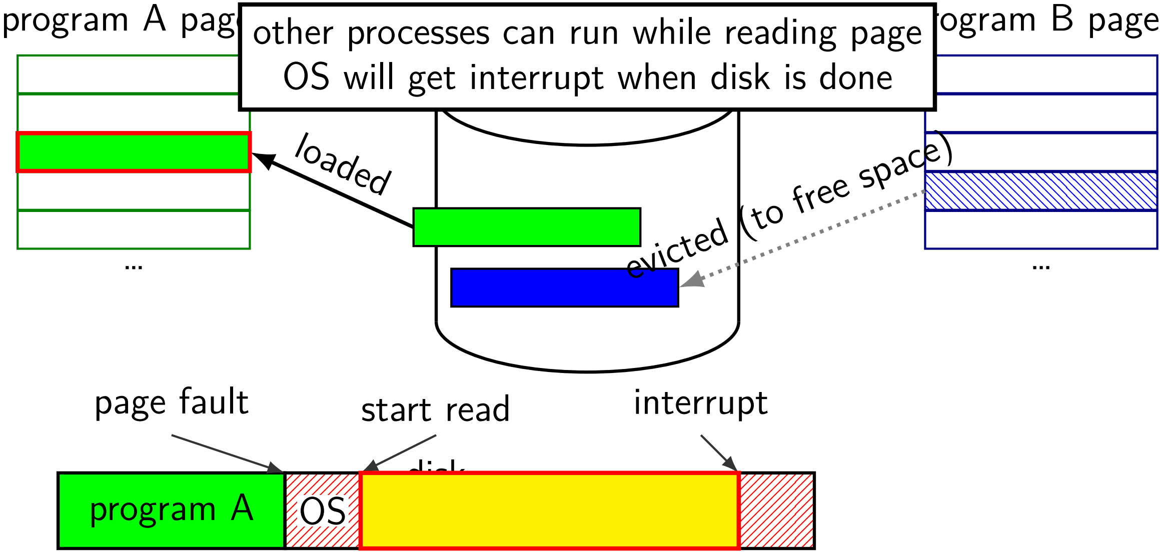
OS needs to choose page to replace
hopefully copy on disk is already up-to-date?



swapping timeline

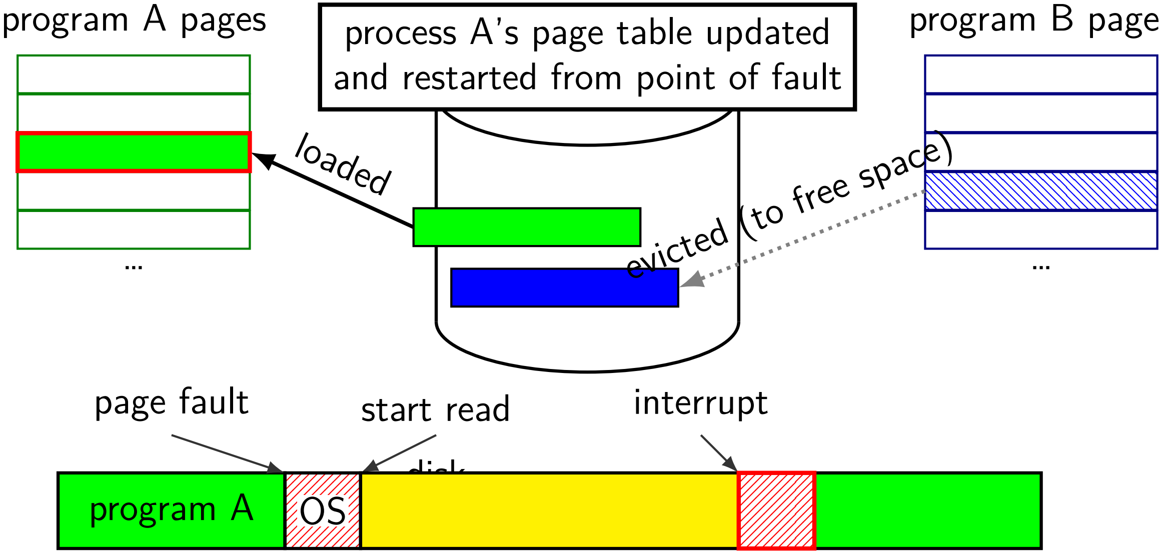


swapping timeline



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

swapping timeline



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

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