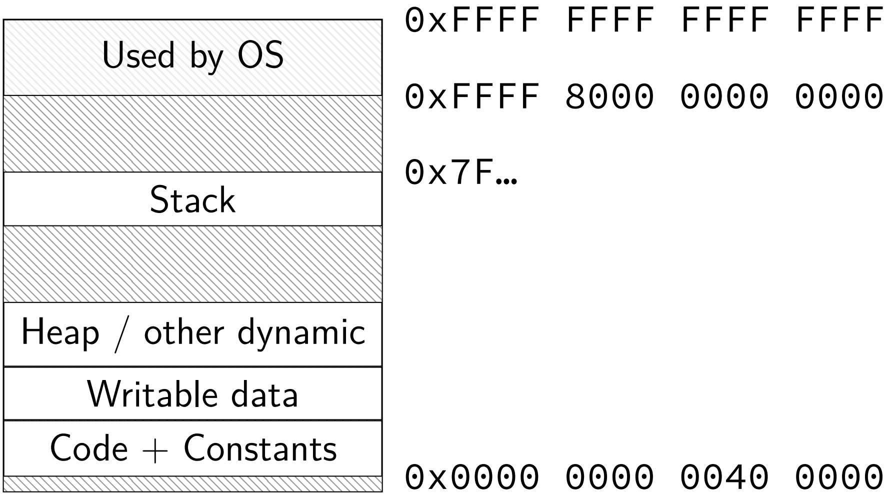
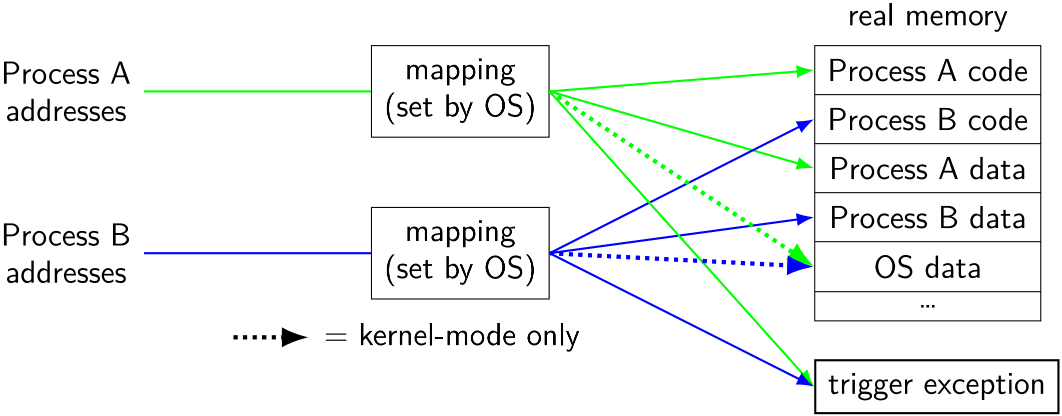
program memory



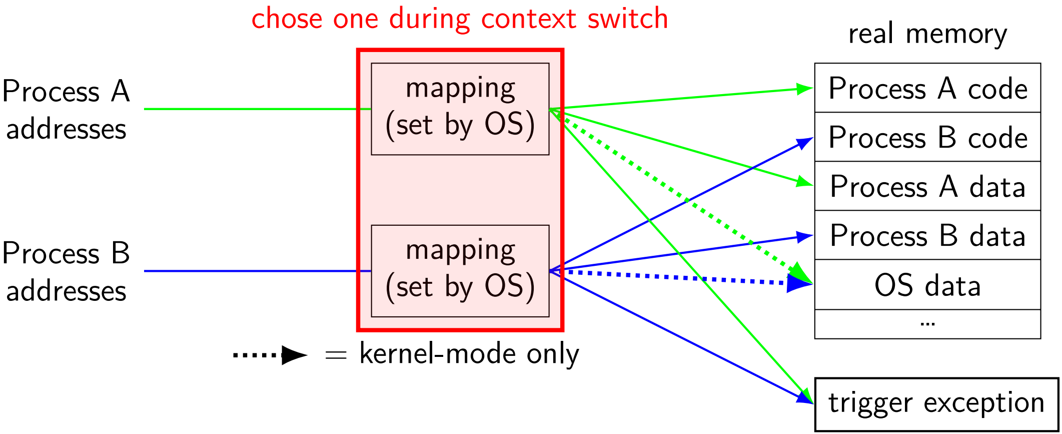
address spaces

illuision of dedicated memory

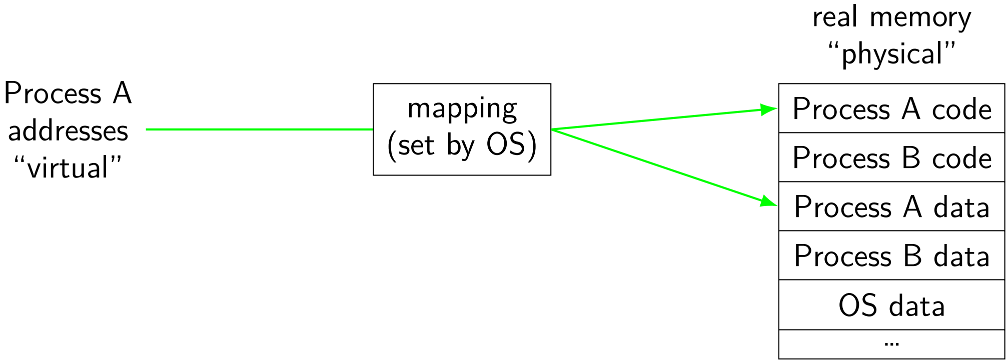


address spaces

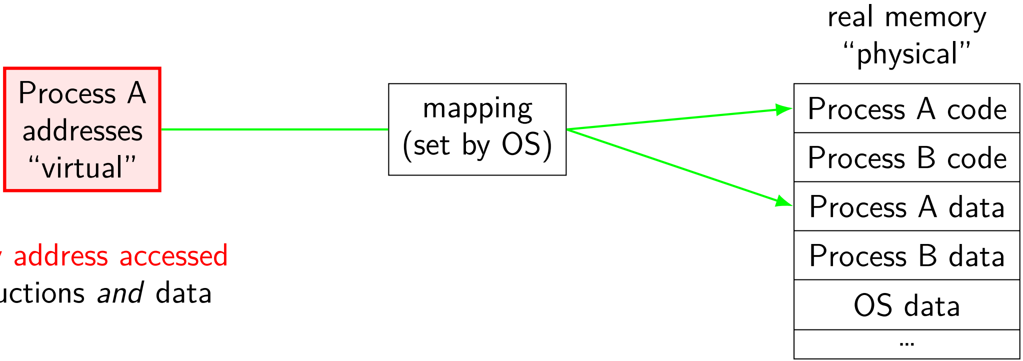
illuision of dedicated memory



address translation

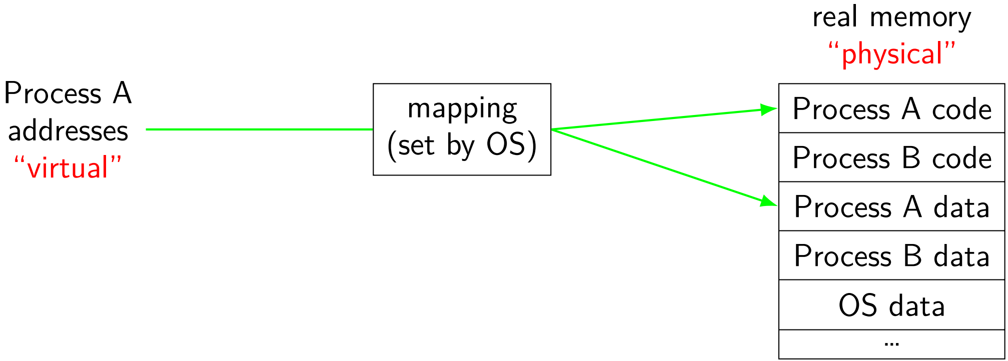


address translation



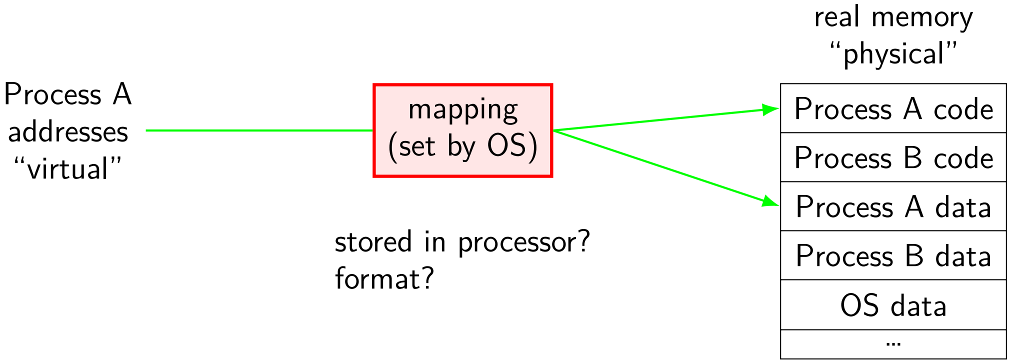
every address accessed
instructions *and* data

address translation



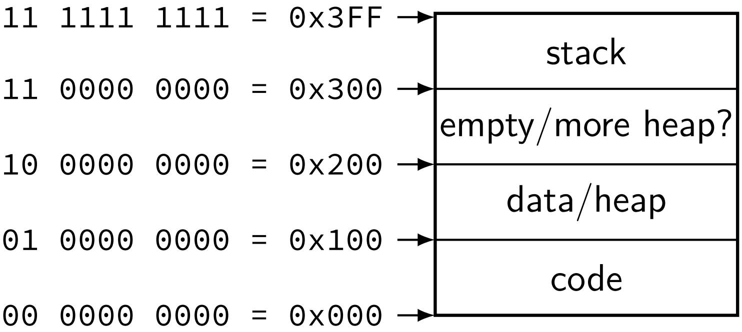
program addresses are 'virtual'
real addresses are 'physical'
can be different sizes!

address translation

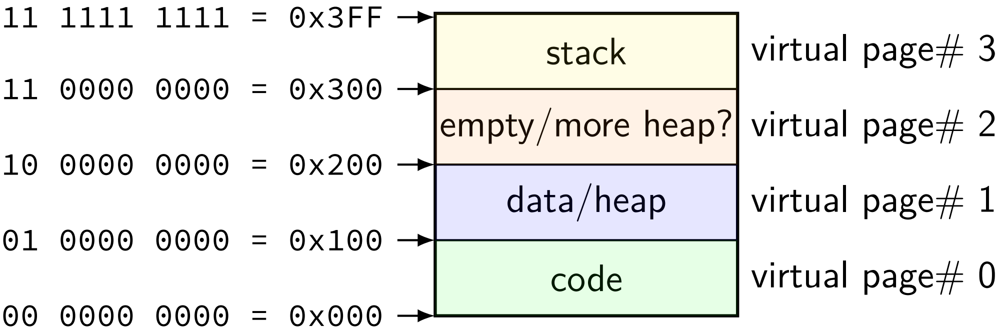


stored in processor?
format?

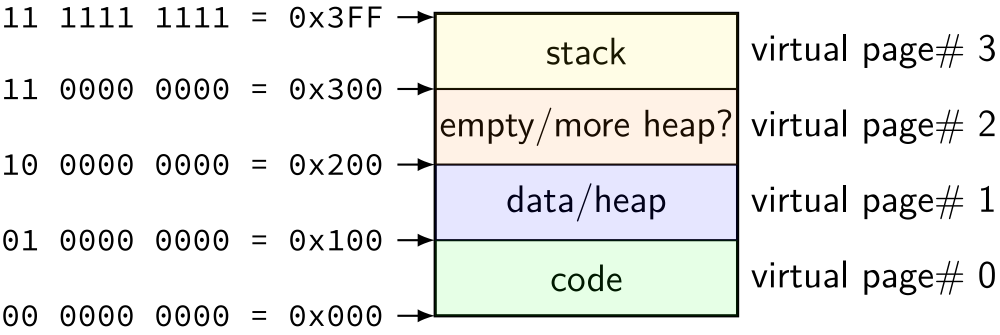
toy program memory



toy program memory

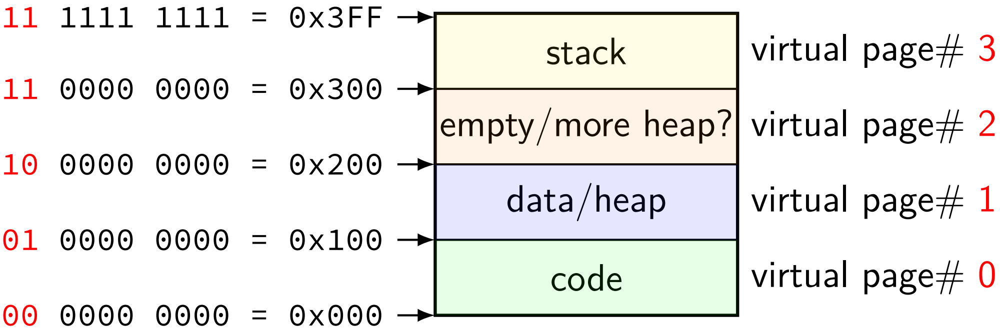


toy program memory



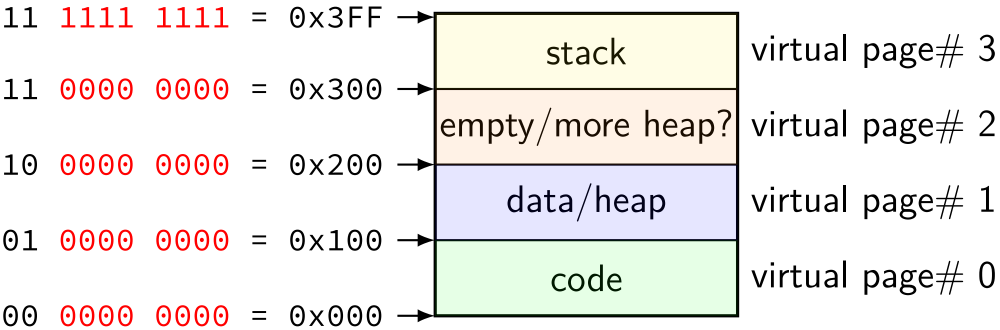
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

toy program memory



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

toy program memory



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

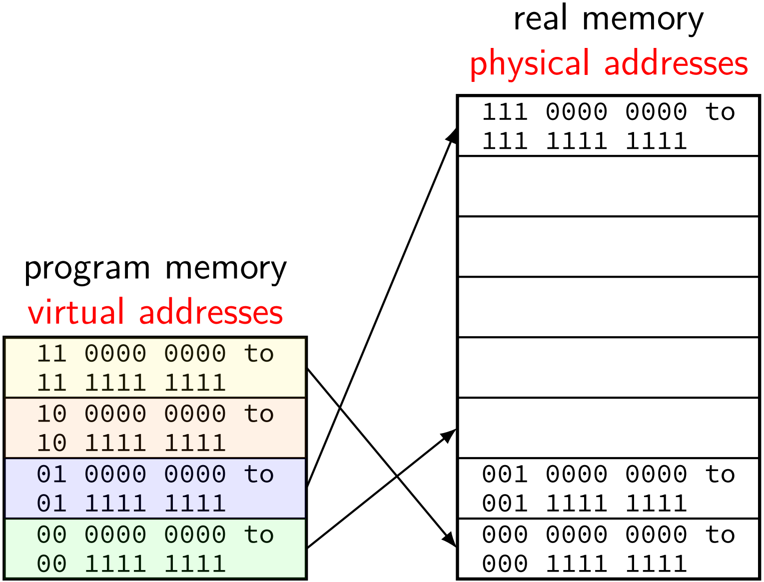
real memory
physical addresses

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

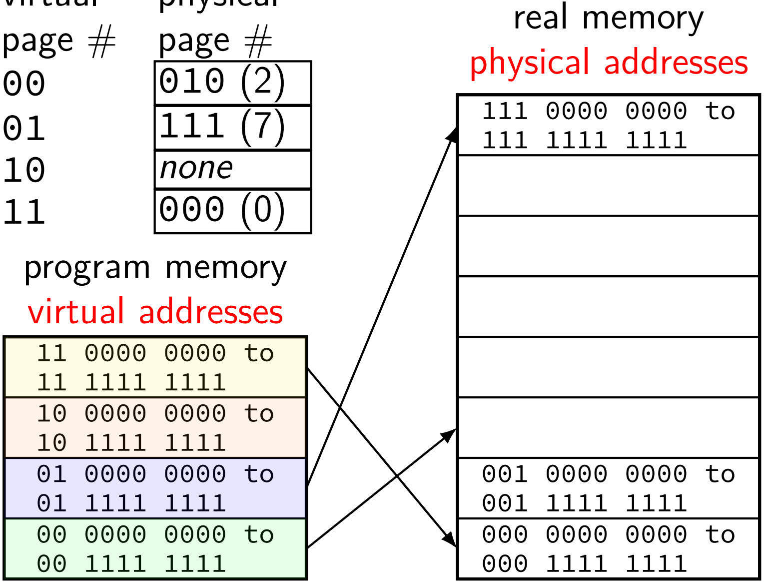
11 0000 0000 to 11 1111 1111
10 0000 0000 to 10 1111 1111
01 0000 0000 to 01 1111 1111
00 0000 0000 to 00 1111 1111

toy physical memory



toy physical memory

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



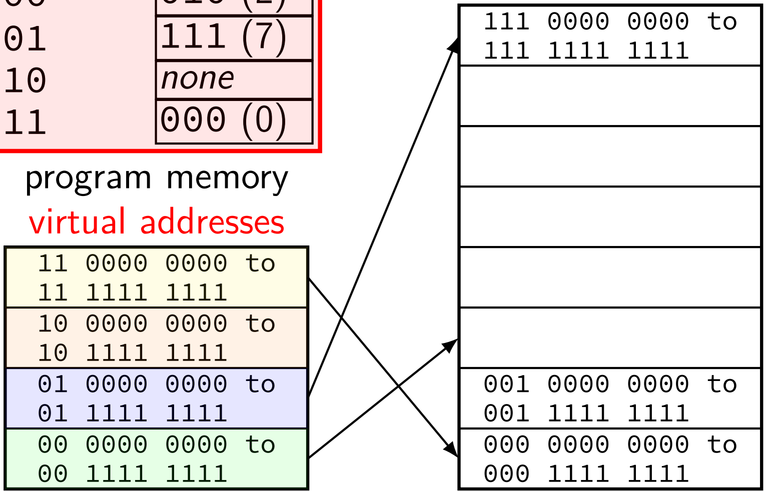
toy physical memory

page table!

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

program memory
virtual addresses

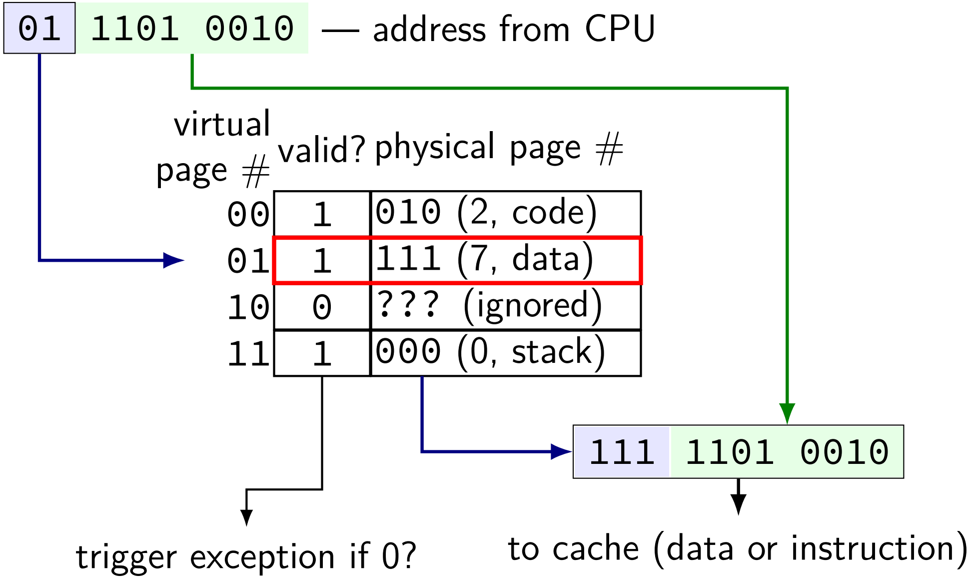
real memory
physical addresses



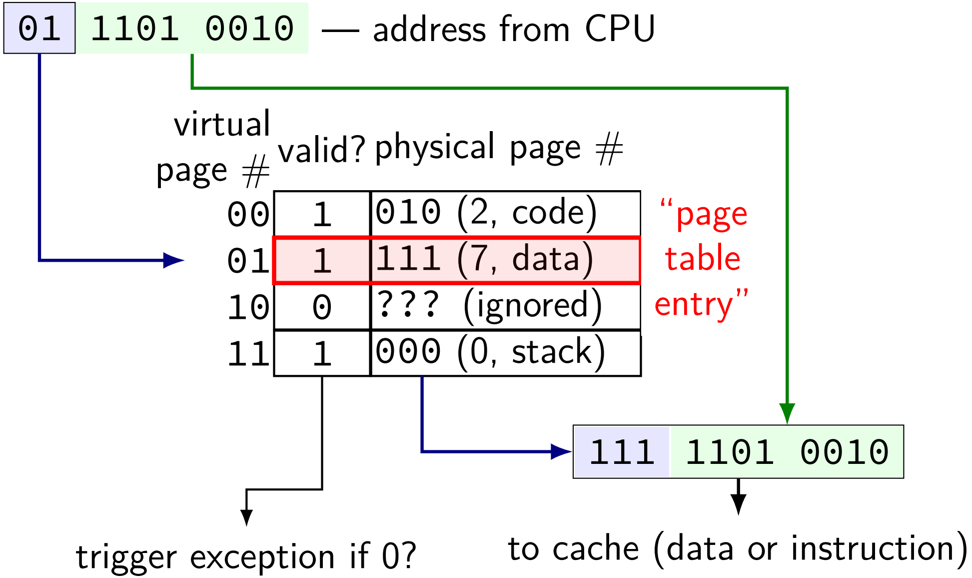
toy page table lookup

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

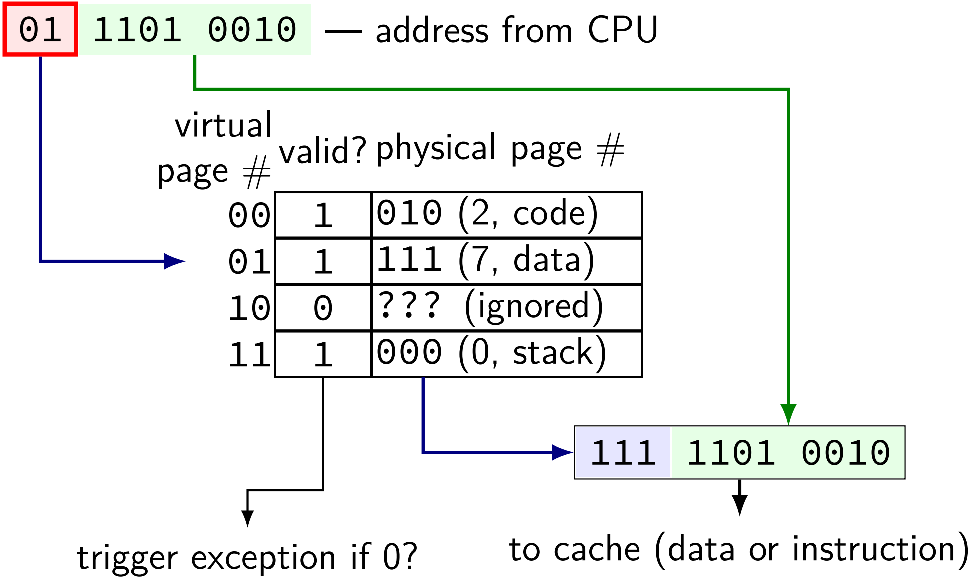
toy page table lookup



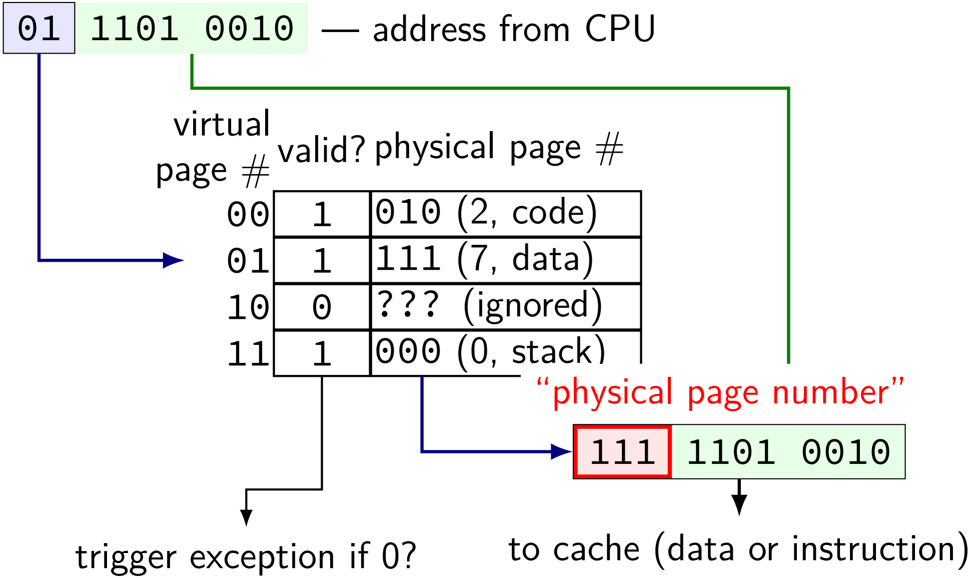
toy page table lookup



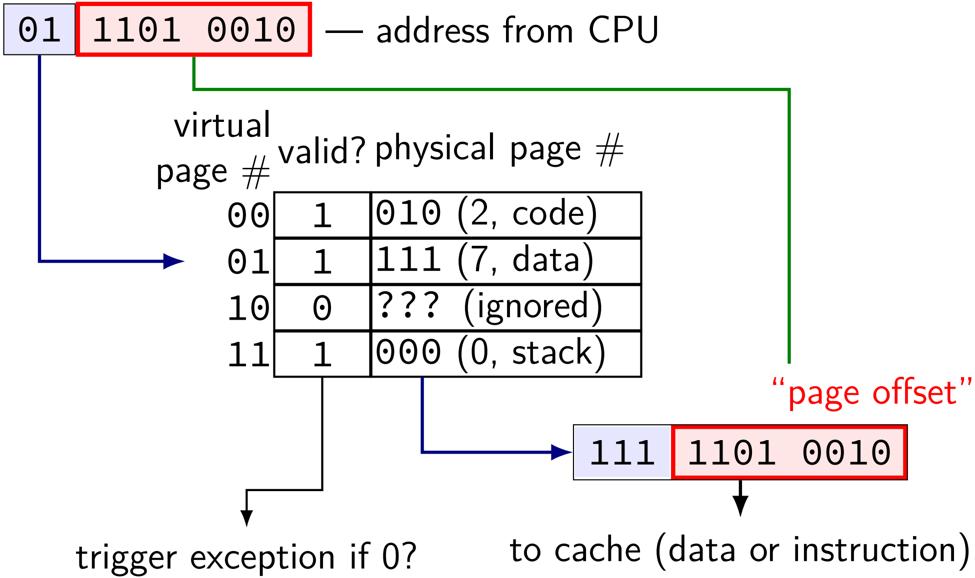
t “virtual page number” lookup



toy page table lookup



toy pag “page offset” ookup



switching page tables

part of context switch is changing the page table

extra privileged instructions

switching page tables

part of context switch is changing the page table

extra privileged instructions

where in memory is the code that does this switching?

switching page tables

part of context switch is changing the page table

extra privileged instructions

where in memory is the code that does this switching?

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

switching page tables

part of context switch is changing the page table

extra privileged instructions

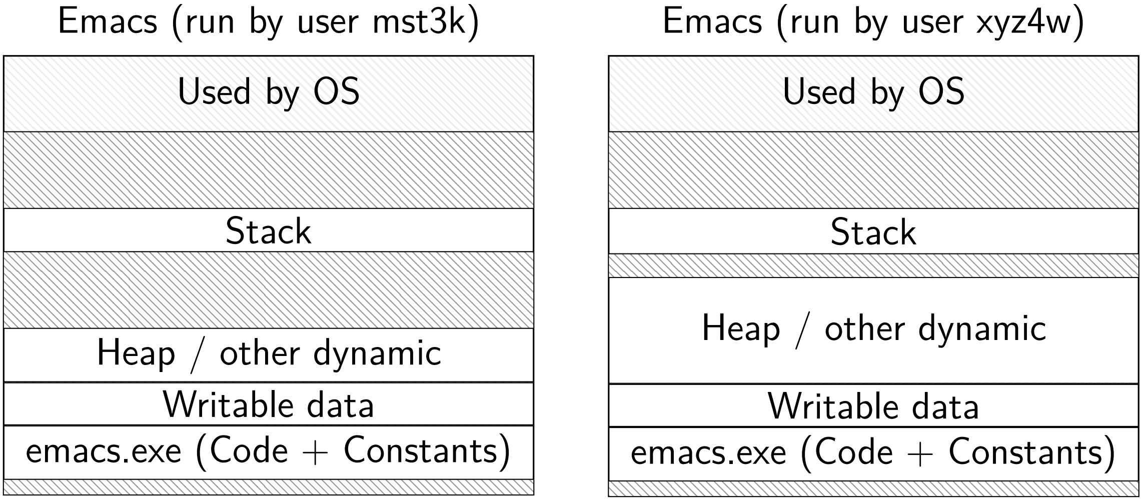
where in memory is the code that does this switching?

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

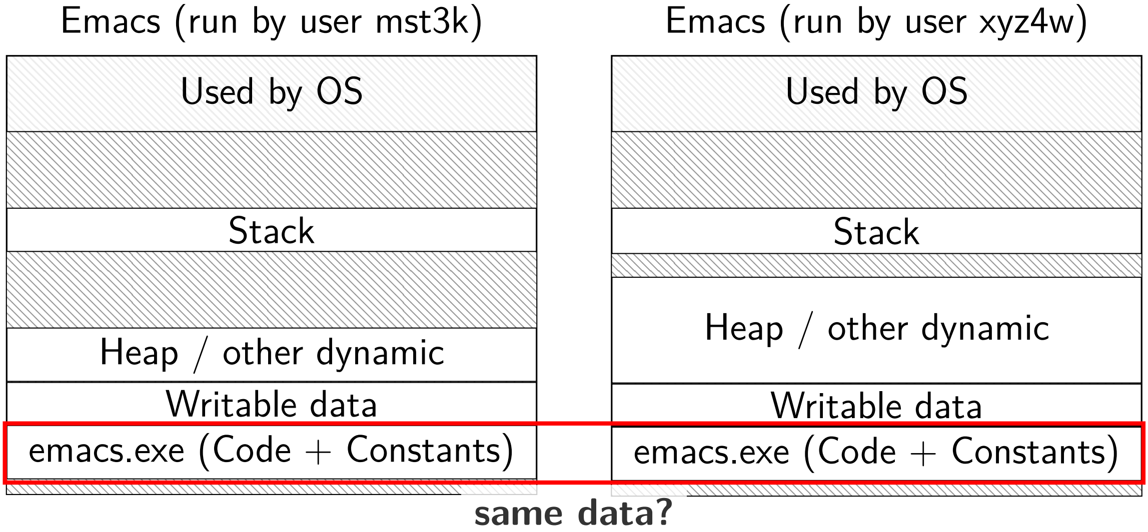
code better not be modified by user program

- otherwise: uncontrolled way to “escape” user mode

emacs (two copies)



emacs (two copies)



two copies of program

would like to only have one copy of program

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

would break process abstraction:

"illusion of own memory"

typical page table entries

solution: same idea as kernel-only bit

page table entry will have more permissions bits

- can read?
- can write?
- can execute?

checked by MMU like valid/kernel bit

page table (logically)

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

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?

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, kernel-mode bit

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

exercise: page table size

suppose 32-bit virtual (program) addresses

suppose 30-bit physical (hardware) addresses

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

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

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

- 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 tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout

valid (bit 15)	kernel (bit 14)	physical page # (bits 4–13)	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

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

page table base register

`0x00010000`

physical memory

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout

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

page table base register

`0x00010000`

physical memory

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout

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

page table base register

`0x00010000`

physical memory

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout

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

page table base register

`0x00010000`

physical memory

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout

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

page table base register

`0x00010000`

page table (logically)

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

physical memory

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout

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

page table base register

0x00010000

page table (logically)

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

physical memory

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout

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

page table base register

`0x00010000`

page table (logically)

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

physical memory

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout

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

page table base register

0x00010000

page table (logically)

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

physical memory

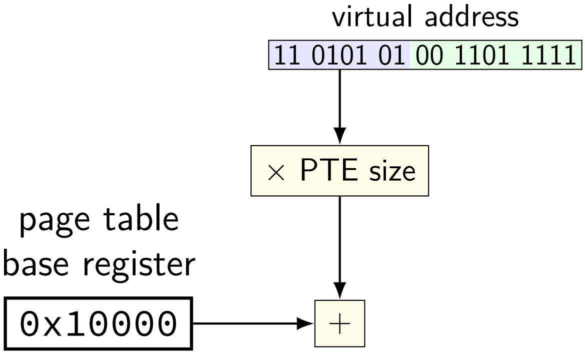
addresses	bytes
0x00000000-1	00000000 00000000
...	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
...	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

memory access with page table

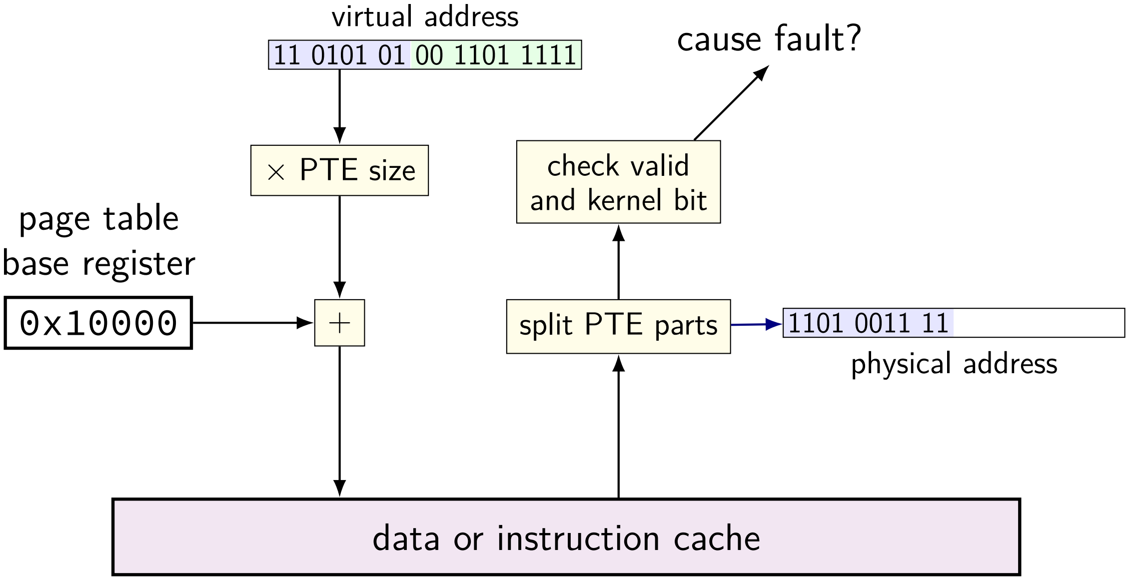
virtual address

11 0101 01	00 1101 1111

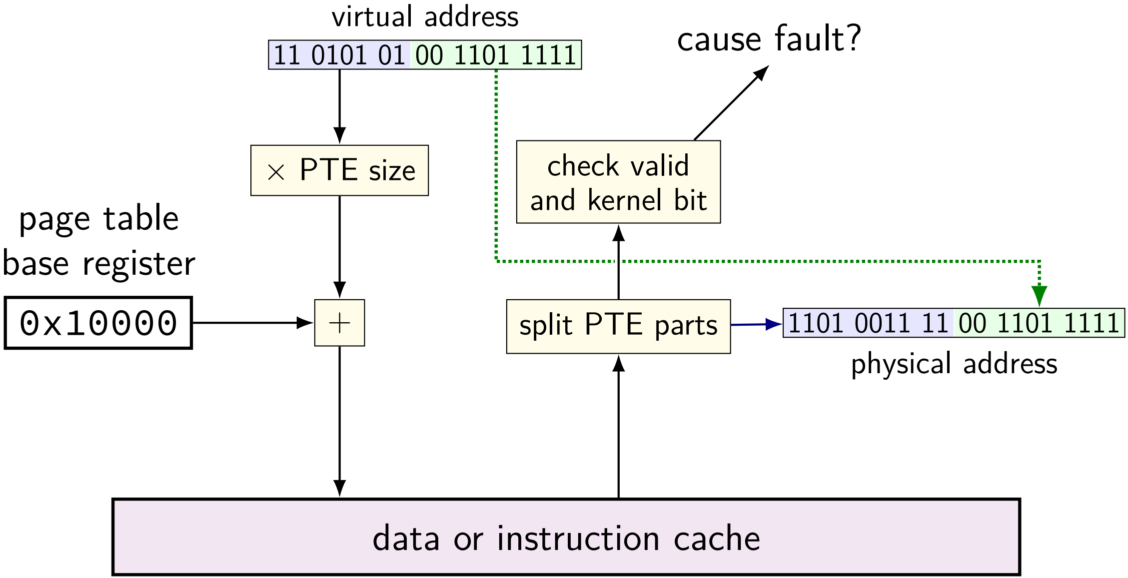
memory access with page table



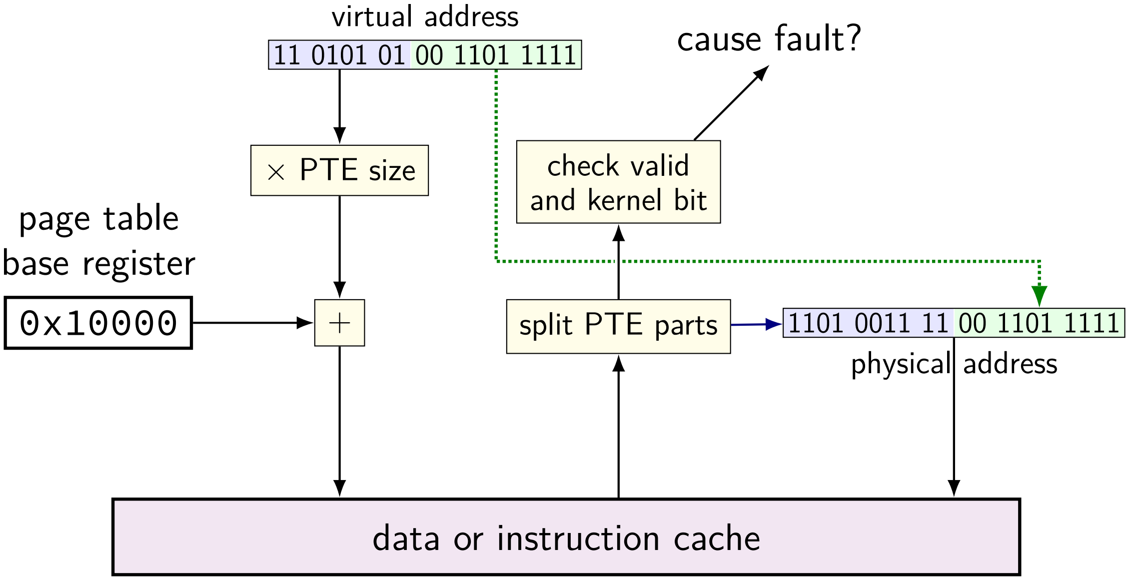
memory access with page table



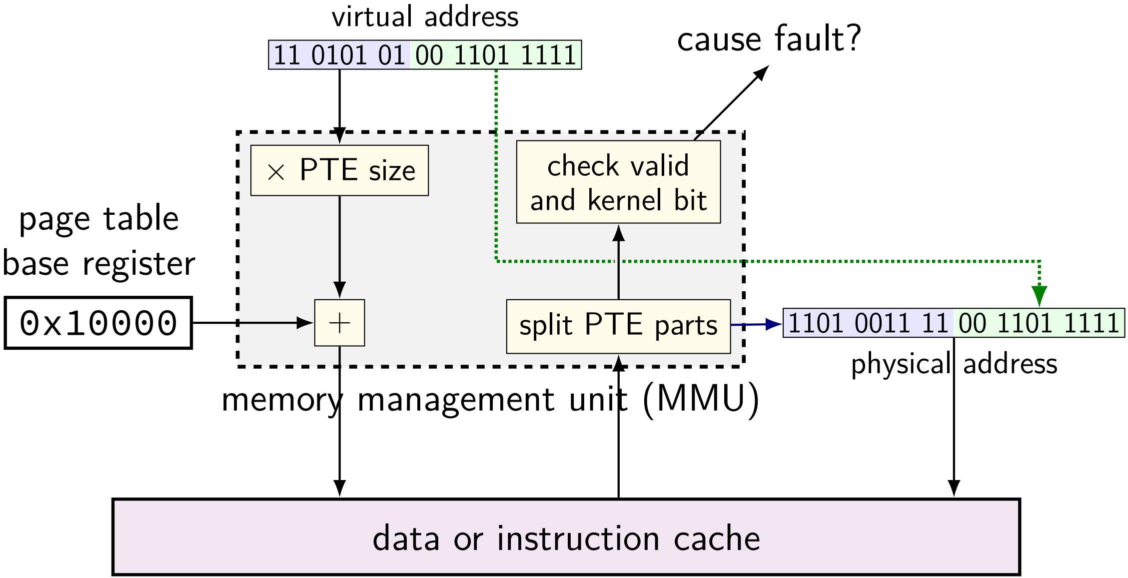
memory access with page table



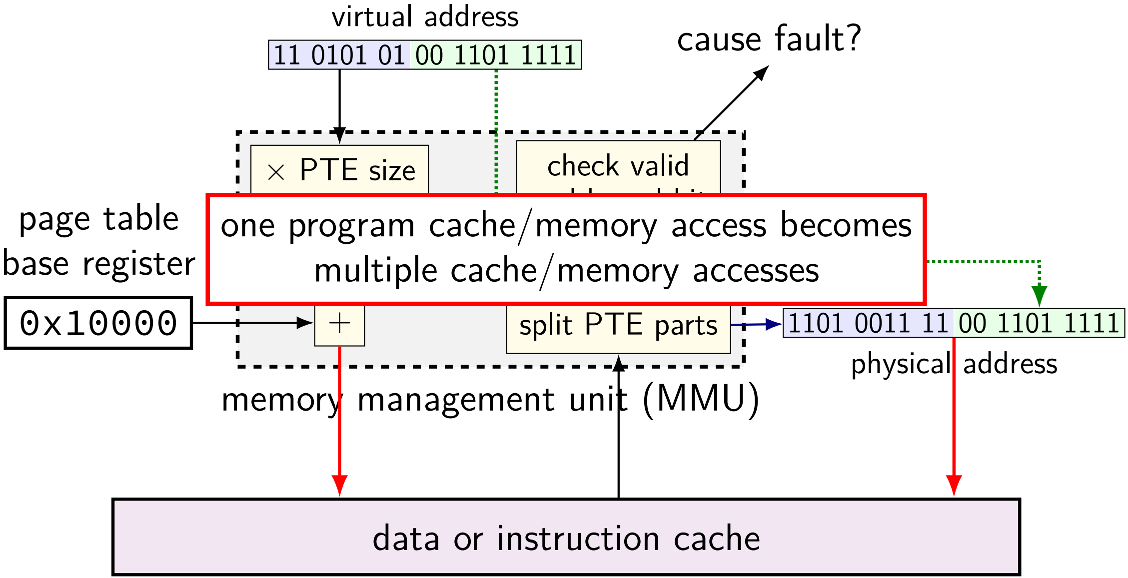
memory access with page table



memory access with page table

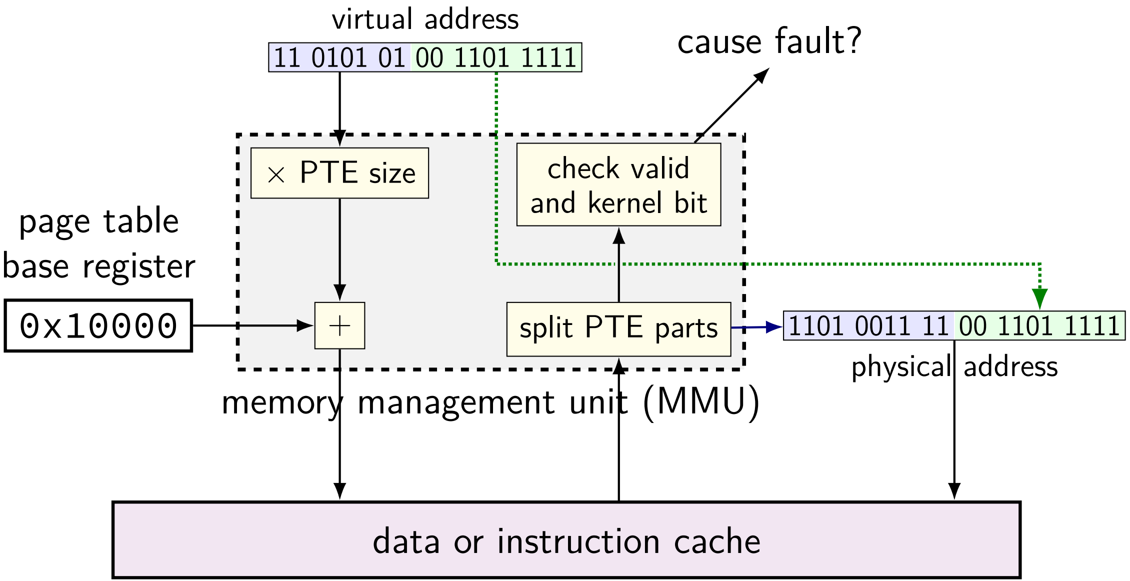


memory access with page table

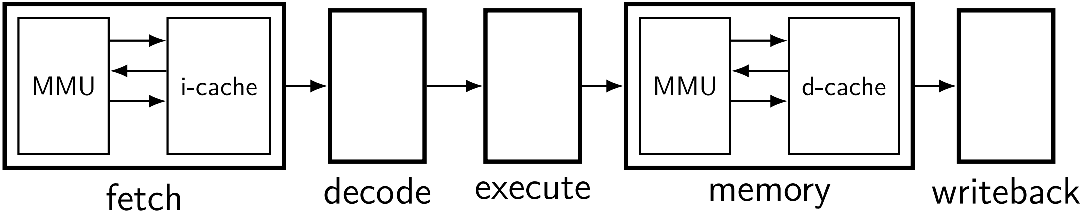


one program cache/memory access becomes
multiple cache/memory accesses

memory access with page table

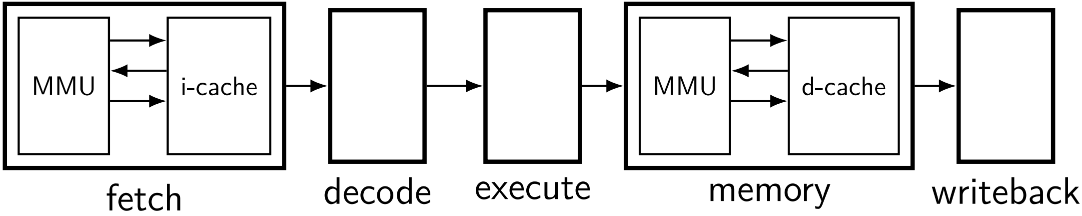


MMUs in the pipeline



up to four memory accesses per instruction

MMUs in the pipeline



up to four memory accesses per instruction

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

exercise 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` = ; `0x03` = ???; `0x0A` = ???; `0x13` = ???

page table

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

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

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

exercise

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

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

page table

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

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

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

exercise

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

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

page table

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

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

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

exercise

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

(virtual addresses) `0x18` = ; `0x03` = ; `0x0A` = ; `0x13` =

page table

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

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

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

1-level example

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

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

page table base register `0x20`; translate virtual address `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	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

1-level example

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

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

page table base register `0x20`; translate virtual address `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	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

`0x31` $=$ `11 0001`

PTE addr:

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

PTE value:

`0xF6` $=$ `1111 0110`

PPN `111`, valid `1`

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

$\rightarrow$ `0x0C`

1-level example

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

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

page table base register 0x20; translate virtual address 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	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x31 $=$ 11 0001

PTE addr:

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

PTE value:

0xF6 $=$ 1111 0110

PPN 111, valid 1

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

$\rightarrow$ 0x0C

1-level example

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

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

page table base register `0x20`; translate virtual address `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	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

$\texttt{0x31} = \texttt{11 0001}$

PTE addr:

$\texttt{0x20} + 6 \times 1 = \texttt{0x26}$

PTE value:

$\texttt{0xF6} = \texttt{1111 0110}$

PPN `111`, valid 1

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

$\rightarrow \texttt{0x0C}$

1-level example

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

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

page table base register `0x20`; translate virtual address `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	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

1-level example

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

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

page table base register `0x20`; translate virtual address `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	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

`0x12` = `01 0010`

PTE addr:

`0x20` + 2 × 1 = `0x22`

PTE value:

`0xD2` = `1101 0010`

PPN `110`, valid `1`

M[`110 010`] = **M[**`0x32`]

→ `0xBA`

1-level example

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

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

page table base register `0x20`; translate virtual address `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	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

$\texttt{0x12} = \texttt{01 0010}$

PTE addr:

$\texttt{0x20} + 2 \times 1 = \texttt{0x22}$

PTE value:

$\texttt{0xD2} = \texttt{1101 0010}$

PPN `110`, valid 1

$\mathrm{M}[\texttt{110 010}] = \mathbf{M[}\texttt{0x32}]$

$\rightarrow \texttt{0xBA}$

1-level example

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

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

page table base register `0x20`; translate virtual address `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	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x12 = 01 0010

PTE addr:

0x20 + 2 ×1 = 0x22

PTE value:

0xD2 = 1101 0010

PPN 110, valid 1

M[110 010] = **M[**0x32]

→ 0xBA

exercise: 64-bit system

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

4096 byte pages

exercise: 64-bit system

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

4096 byte pages

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

exercise: 64-bit system

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

4096 byte pages

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

exercise: how large are physical page numbers?

exercise: 64-bit system

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

4096 byte pages

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

exercise: how large are physical page numbers?

exercise: 64-bit system

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

4096 byte pages

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

exercise: how large are physical page numbers?

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

- trick: power of two size makes table lookup faster

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

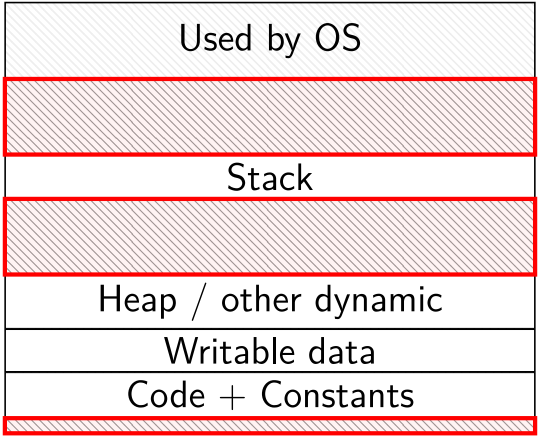
huge page tables

huge virtual address spaces!

impossible to store PTE for every page

how can we save space?

holes



most pages are invalid

saving space

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

use a data structure other than a flat array

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

options?

saving space

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

use a data structure other than a flat array

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

options?

hashtable

- actually used by some historical processors
- but never common

saving space

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

use a data structure other than a flat array

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

options?

hashtable

- actually used by some historical processors
- but never common

tree data structure

- but not quite a search tree

search tree tradeoffs

lookup usually implemented in hardware

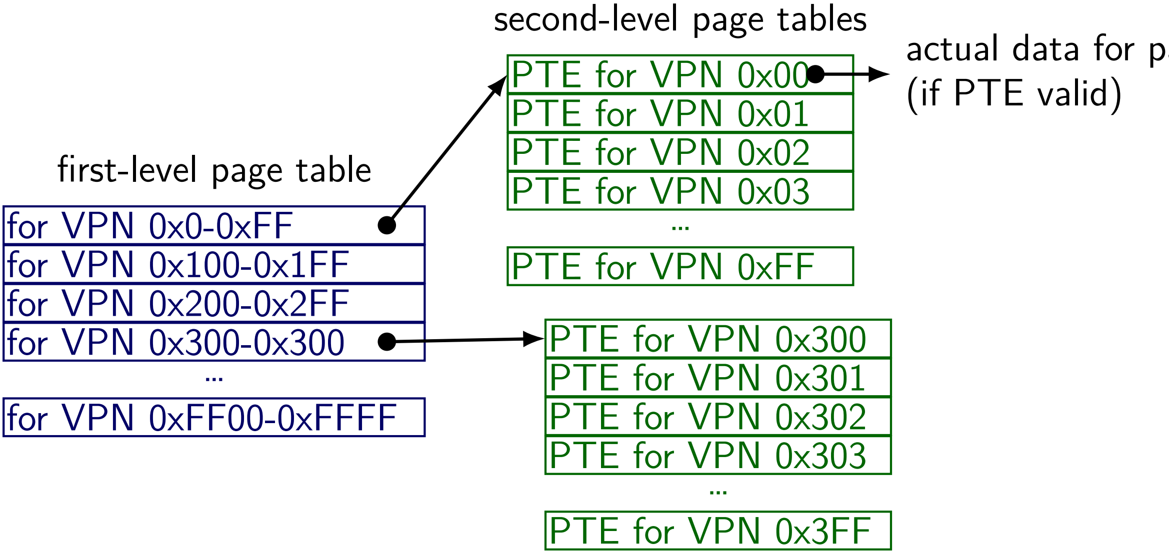
- lookup should be simple
- solution: lookup splits up address bits (no complex calculations)

lookup should not involve many memory accesses

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

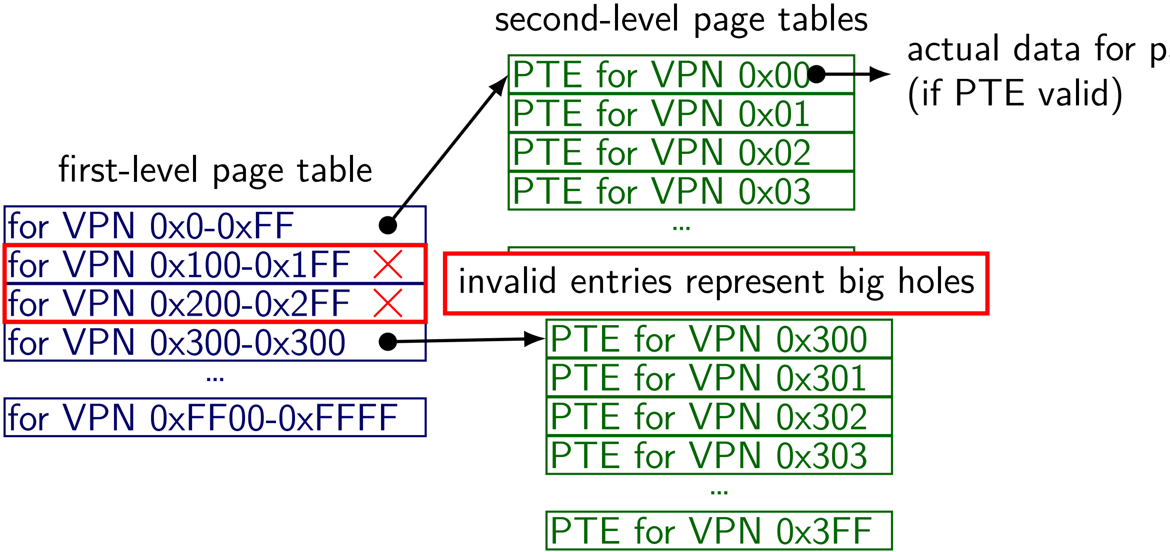
two-level page tables

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



two-level page tables

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



two-level page tables

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

first-level page table

VPN range	valid	kernel	write	physical page # (of next page table)
0x0000–0x00FF	1	0	1	0x22343
0x0100–0x01FF	0	0	1	0x00000
0x0200–0x02FF	0	0	0	0x00000
0x0300–0x03FF	1	1	0	0x33454
0x0400–0x04FF	1	1	0	0xFF043
…	…	…	…	…
0xFF00–0xFFFF	1	1	0	0xFF045

first-level pag

for VPN 0x0-0xF

for VPN 0x100-0

for VPN 0x200-0

for VPN 0x300-0

…

for VPN 0xFF00

PTE for VPN 0x303

…

PTE for VPN 0x3FF

two-level page tables

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

first-level pag

for VPN 0x0-0xF
for VPN 0x100-0
for VPN 0x200-0
for VPN 0x300-0
…
for VPN 0xFF00

first-level page table

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

PTE for VPN 0x303
…
PTE for VPN 0x3FF

two-level page tables

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

first-level pag

for VPN 0x0-0xF
for VPN 0x100-0
for VPN 0x200-0
for VPN 0x300-0
...
for VPN 0xFF00

first-level page table

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

PTE for VPN 0x303

...

PTE for VPN 0x3FF

two-level page tables

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

first-level page table

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

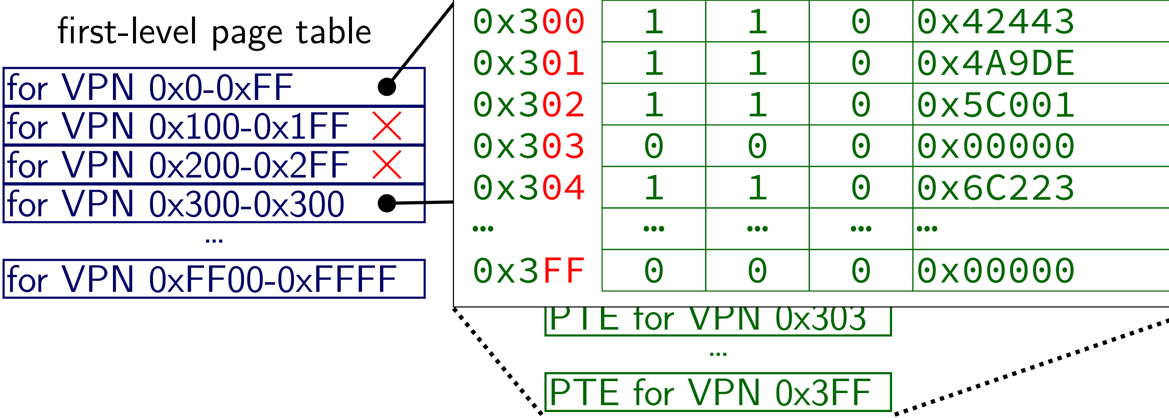
a second-level page table

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

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

two-level page tables

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

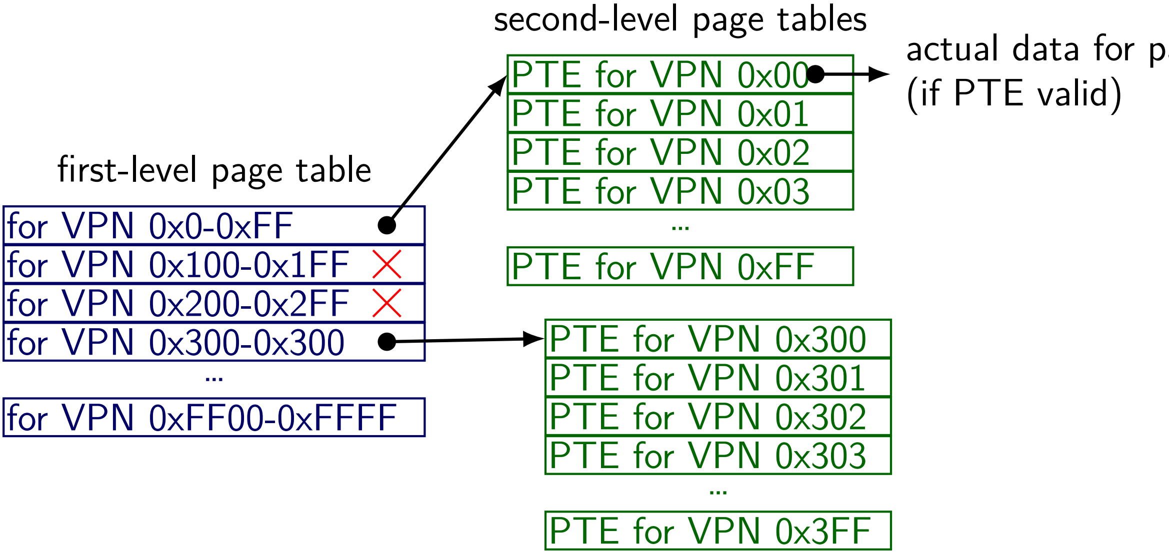


a second-level page table

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

two-level page tables

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



two-level page table lookup

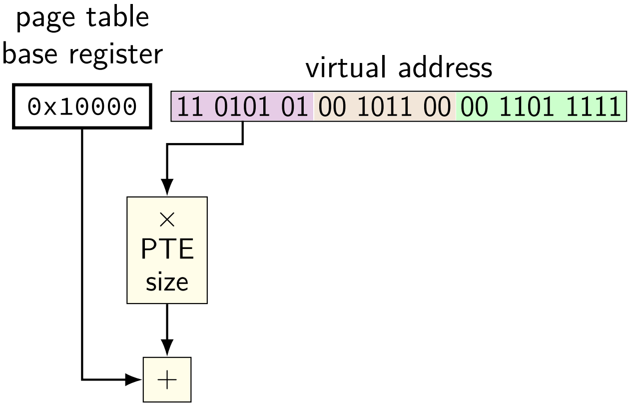
virtual address

11 0101 01 00 1011 00 00 1101 1111

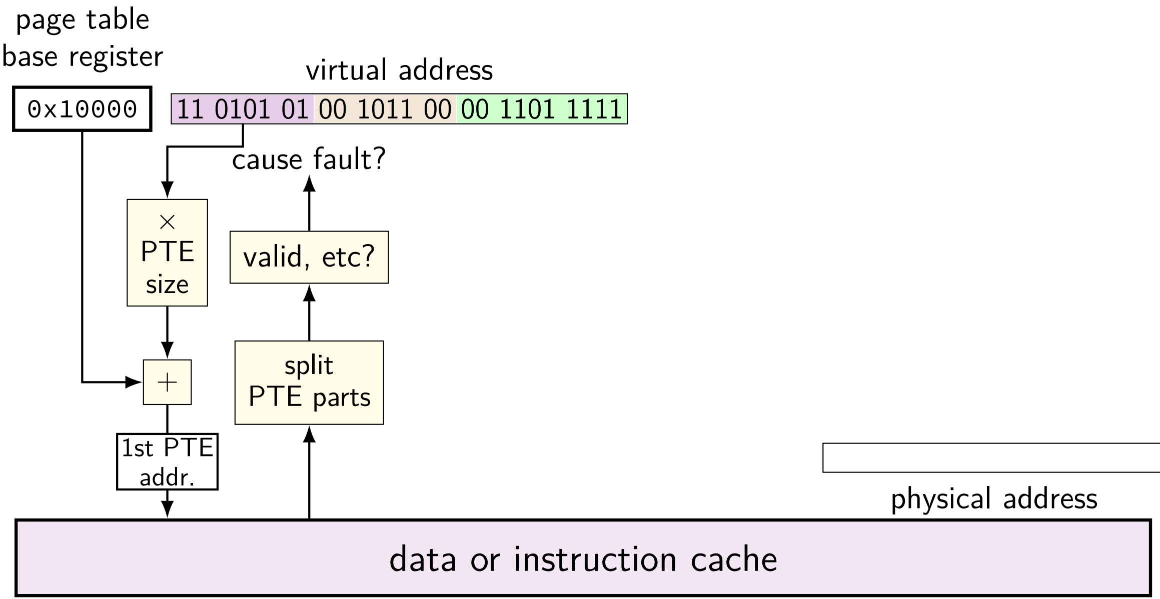
VPN — split into two parts (one per level)

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

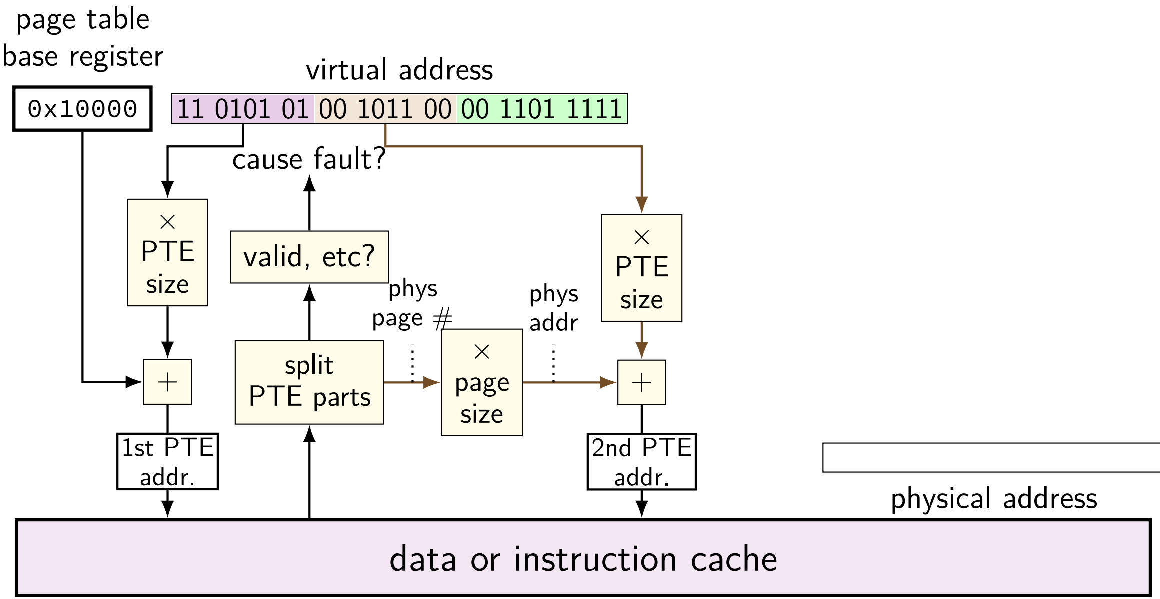
two-level page table lookup



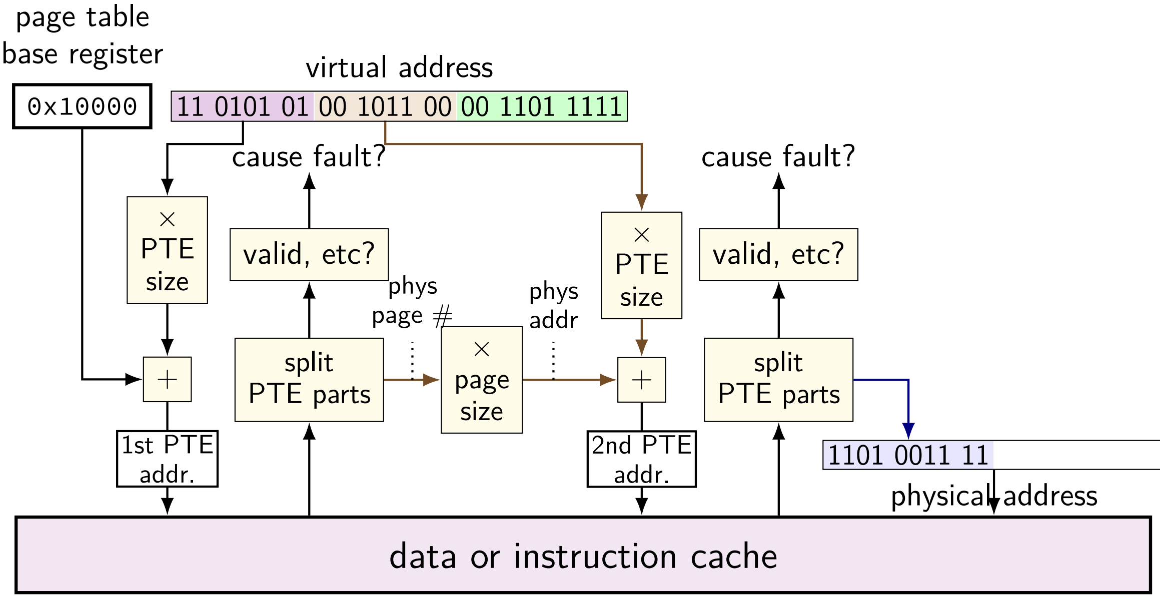
two-level page table lookup



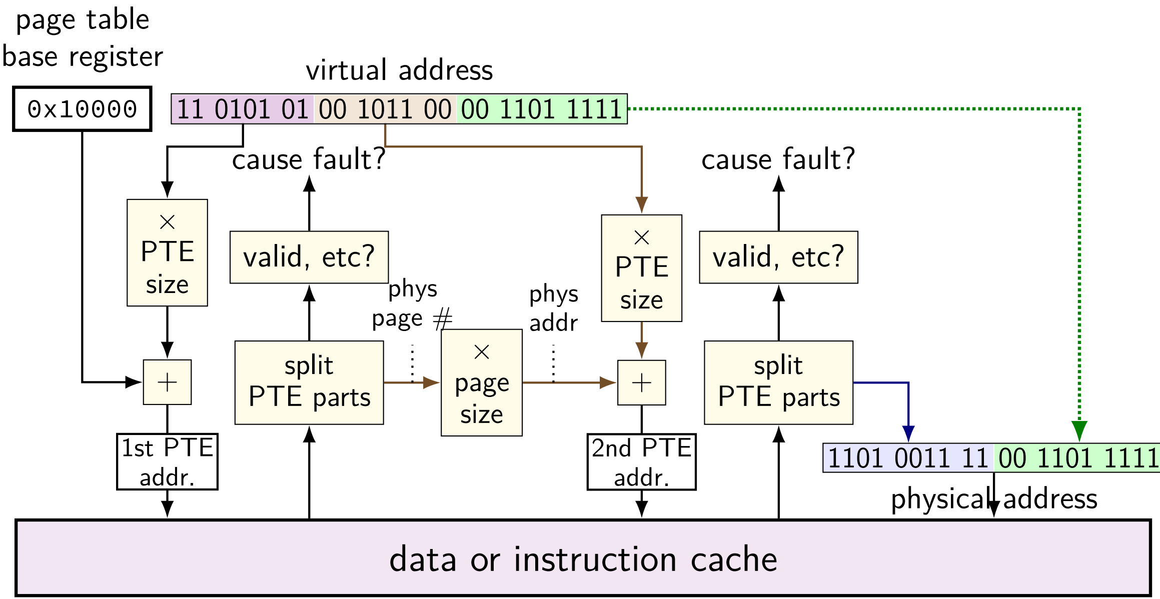
two-level page table lookup



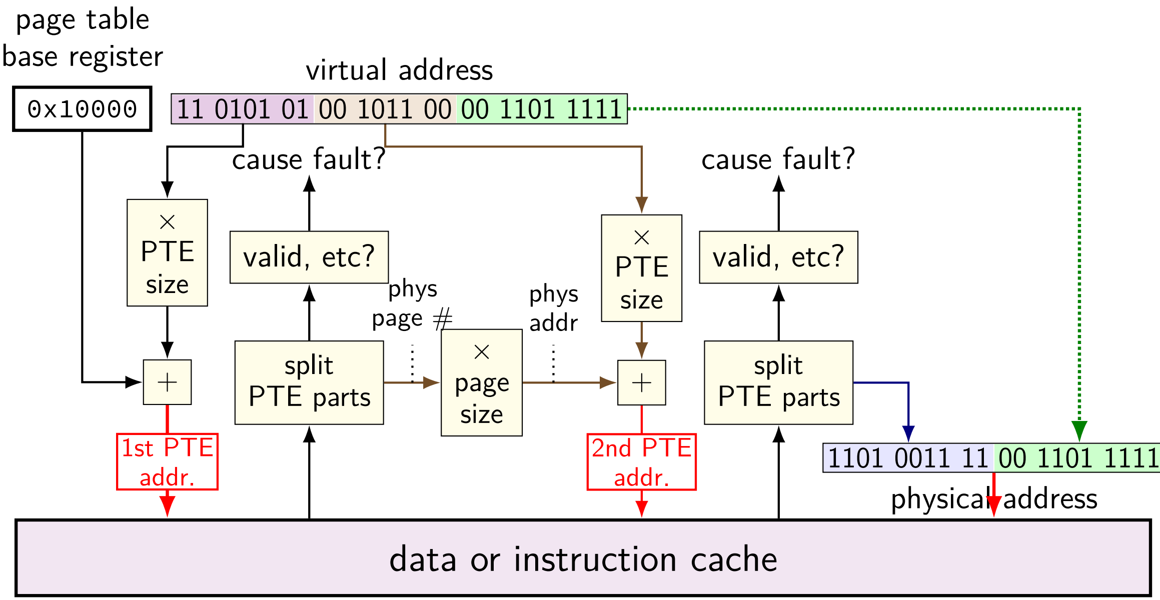
two-level page table lookup



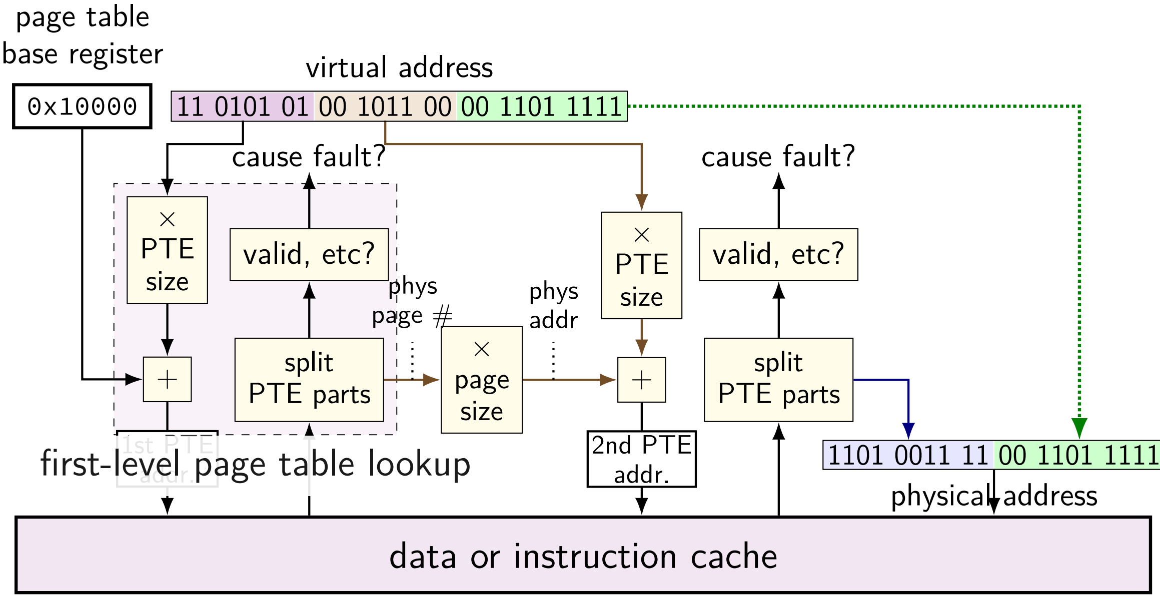
two-level page table lookup



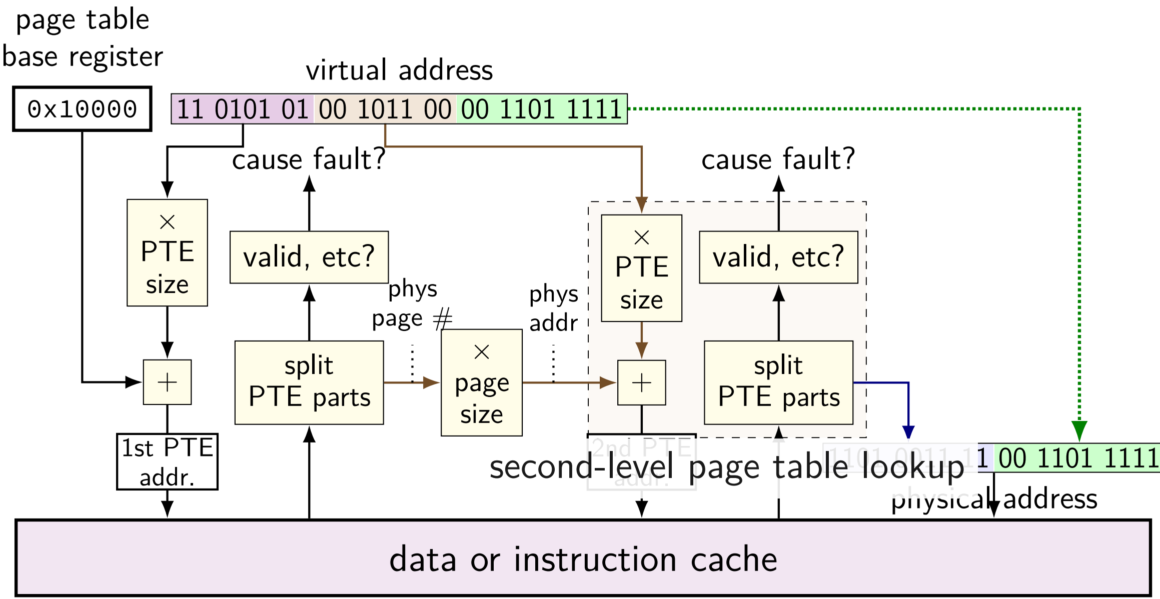
two-level page table lookup



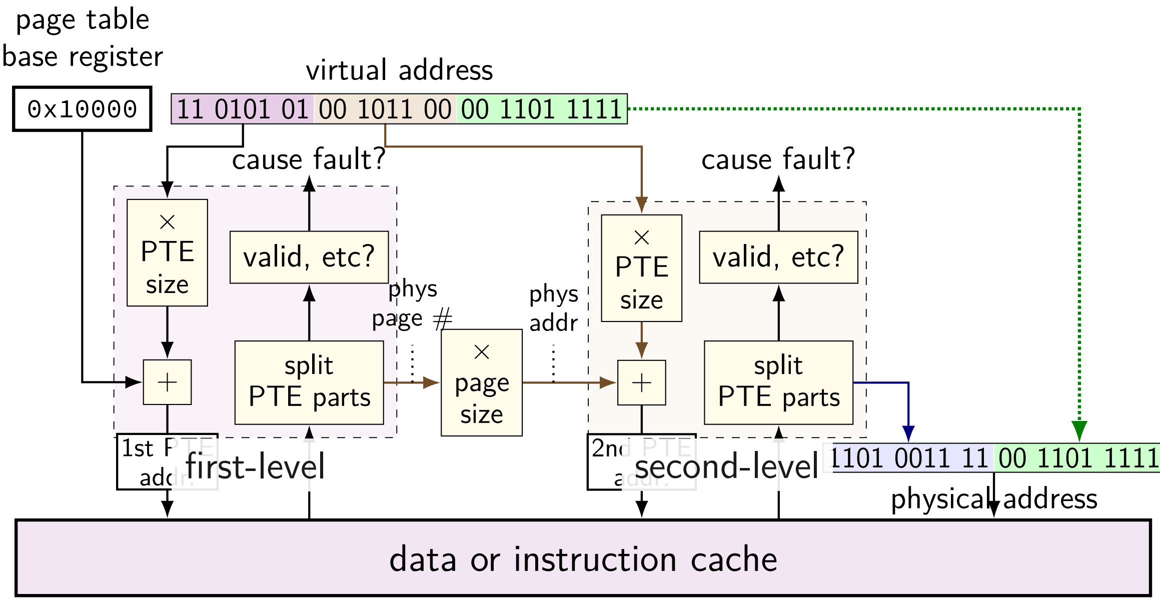
two-level page table lookup



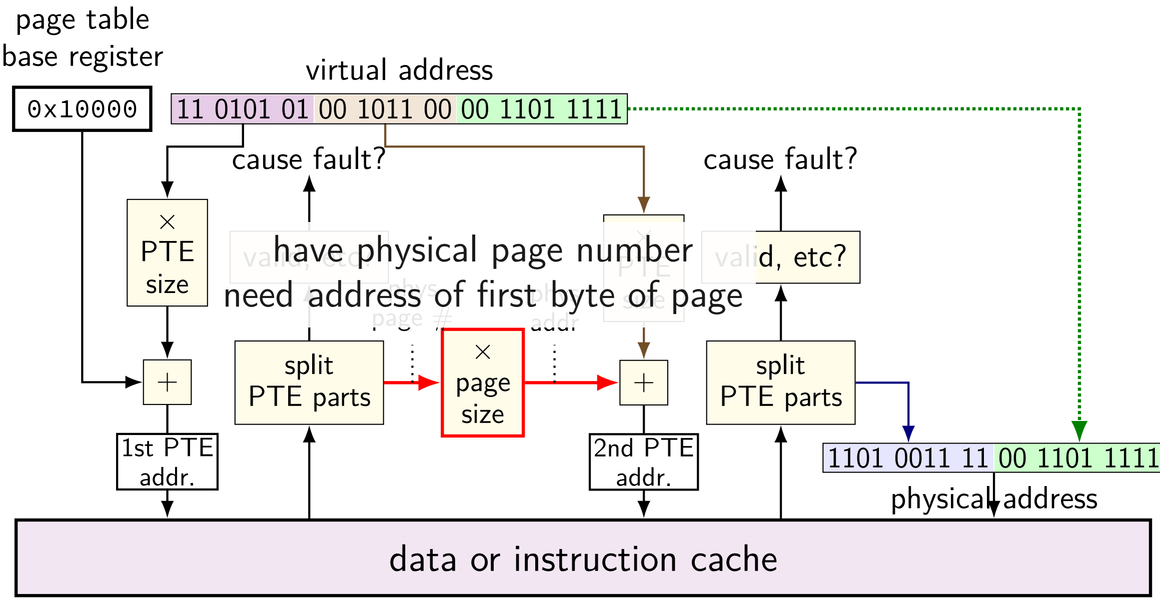
two-level page table lookup



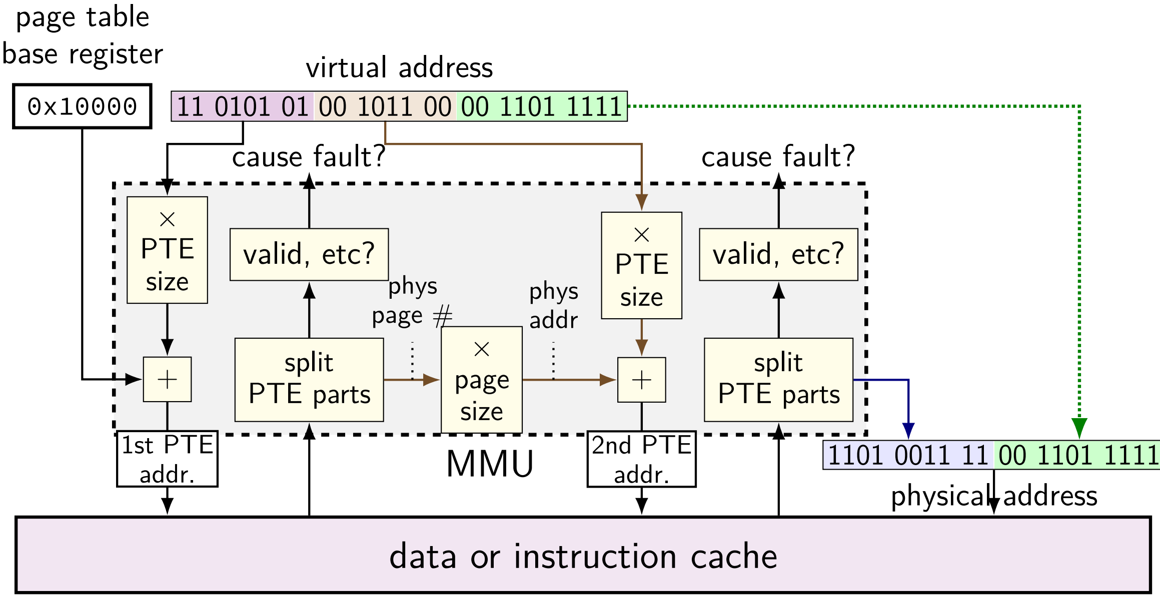
two-level page table lookup



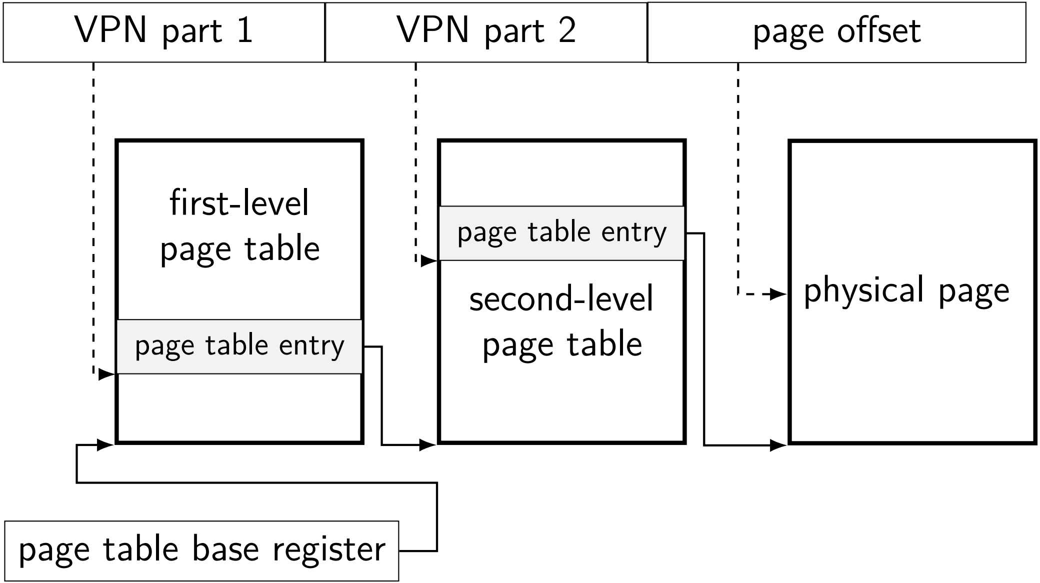
two-level page table lookup



two-level page table lookup



another view



multi-level page tables

VPN split into pieces for each level of page table

top levels: page table entries point to next page table

usually using physical page number of next page table

bottom level: page table entry points to destination page

validity and permission checks at each level

x86-64 page table splitting

48-bit virtual address

12-bit page offset (4KB pages)

36-bit virtual page number, split into four 9-bit parts

page tables at each level: 2^9 entries, 8 bytes/entry

- deliberate choice: each page table is one page

note on VPN splitting

textbook labels it ‘VPN 1’ and ‘VPN 2’ and so on

these are parts of the virtual page number

(there are not multiple VPNs)

2-level example

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

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

page table base register `0x20`; translate virtual address `0x131`

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

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

2-level example

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

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

page table base register `0x20`; translate virtual address `0x131`

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

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

2-level example

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

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

page table base register `0x20`; translate virtual address `0x131`

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

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

2-level example

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

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

page table base register `0x20`; translate virtual address `0x131`

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

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

2-level example

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

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

page table base register `0x20`; translate virtual address `0x131`

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

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

2-level example

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

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

page table base register `0x20`; translate virtual address `0x131`

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

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

2-level splitting

9-bit virtual address

6-bit physical address

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

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

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

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

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

2-level exercise (1)

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

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

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

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

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

2-level exercise (1)

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

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

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

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

2-level exercise (1)

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

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

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

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

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

2-level exercise (1)

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

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

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

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

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

2-level exercise (1)

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

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

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

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

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

2-level exercise (2)

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

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

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

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

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

2-level exercise (3)

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

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

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

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

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

2-level exercise (3)

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

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

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

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

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

2-level exercise (3)

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

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

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

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

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

2-level exercise (4)

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

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

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

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

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

2-level exercise (5)

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

page tables 1 page; PTE: 2 bit PPN (MSB of first byte), 1 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: 2 bit PPN (MSB of first byte), 1 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: 2 bit PPN (MSB of first byte), 1 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: 2 bit PPN (MSB of first byte), 1 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: 2 bit PPN (MSB of first byte), 1 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: 2 bit PPN (MSB of first byte), 1 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: 2 bit PPN (MSB of first byte), 1 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

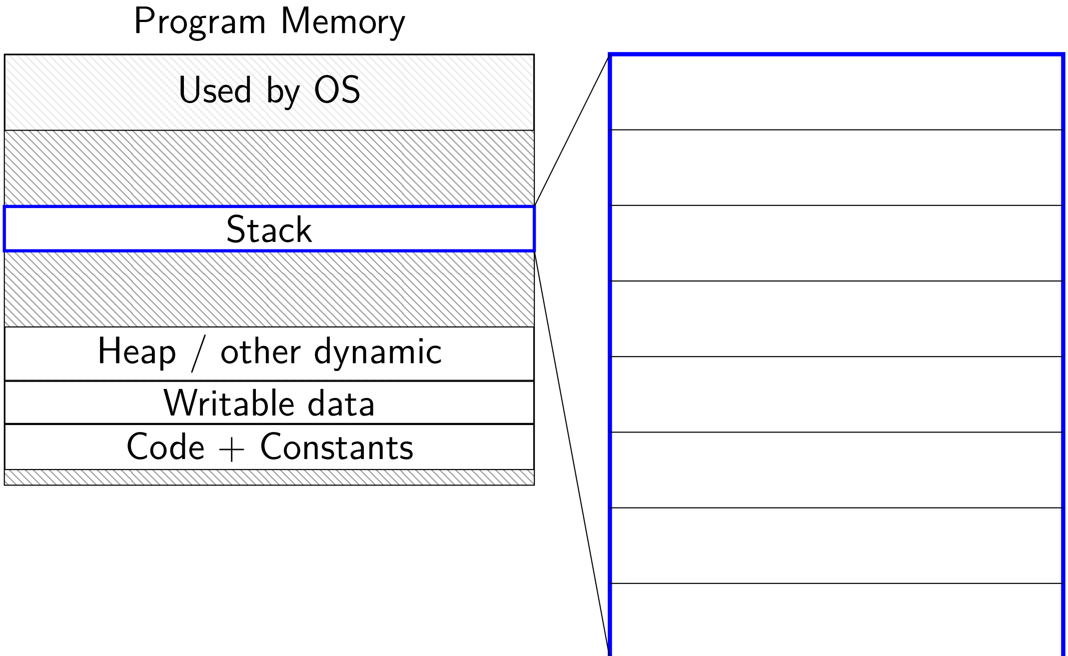
page tables 1 page; PTE: 2 bit PPN (MSB of first byte), 1 valid bit, rest unused

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

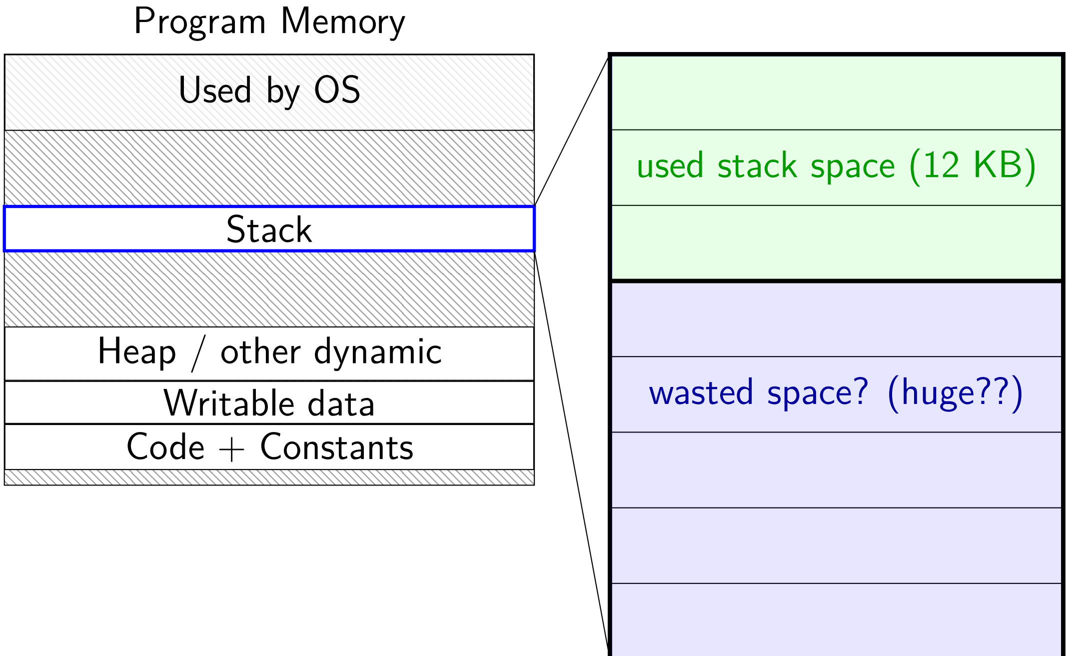
physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	AC BC DC EC

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

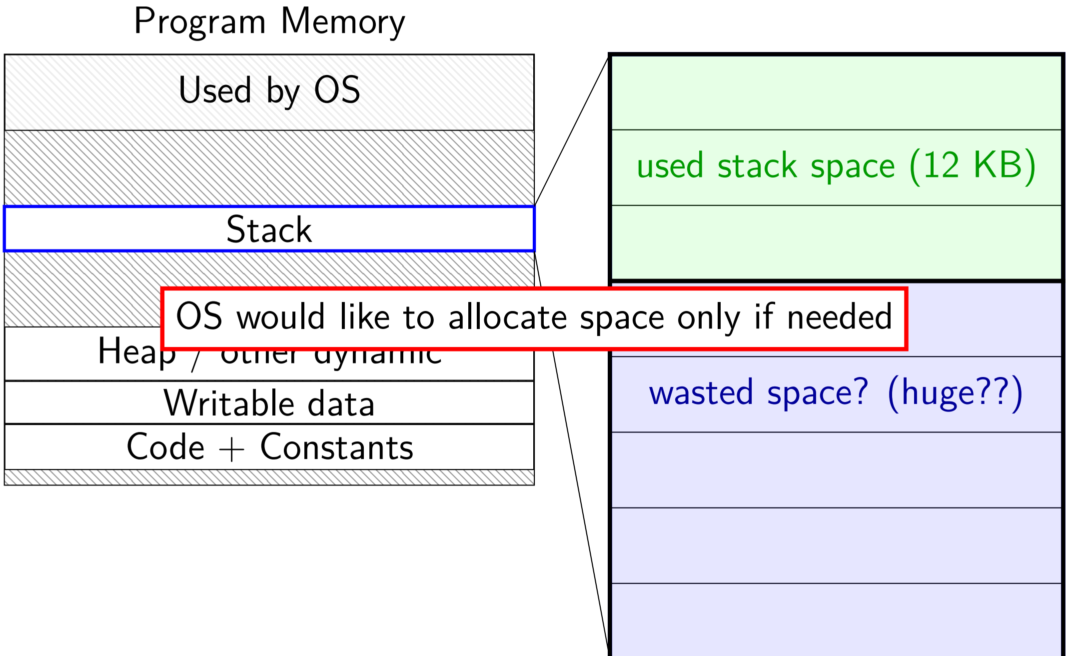
space on demand



space on demand



space on demand



allocating space on demand

%rsp = 0x7FFFC000

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

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

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

allocating space on demand

`%rsp = 0x7FFFC000`

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

page fault!

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

pushq triggers exception
hardware says "accessing address `0x7FFFBFF8`"
OS looks up what's should be there — "stack"

allocating space on demand

`%rsp = 0x7FFFC000`

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

restarted

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

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

allocating space on demand

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

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

loading program can be merely creating empty page table

everything else can be handled in response to page faults

- no time/space spent loading/allocating unneeded space

swapping

early motivation for virtual memory: swapping

using disk (or SSD, ...) as the next level of the memory hierarchy

- how our textbook and many other sources presents virtual memory

OS allocates program space on disk

- own mapping of virtual addresses to location on disk

DRAM is a cache for disk

swapping

early motivation for virtual memory: swapping

using disk (or SSD, …) as the next level of the memory hierarchy

- how our textbook and many other sources presents virtual memory

OS allocates program space on disk

- own mapping of virtual addresses to location on disk

DRAM is a cache for disk

swapping versus caching

"cache block" ≈ physical page

fully associative

- every virtual page can be stored in any physical page

replacement/cache misses managed by the OS

normal cache hits happen in hardware

- hardware's page table lookup
- common case that needs to be very fast

swapping components

“swap in” a page — exactly like allocating on demand!

- OS gets page fault — invalid in page table
- check where page actually is (from virtual address)
- read from disk
- eventually restart process

“swap out” a page

- OS marks as invalid in the page table(s)
- copy to disk (if modified)

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

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

SSD writes and writes: hundreds of microseconds

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

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

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

SSD writes and writes: hundreds of microseconds

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

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

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

SSD writes and writes: hundreds of microseconds

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

HDD/SDDs are slow

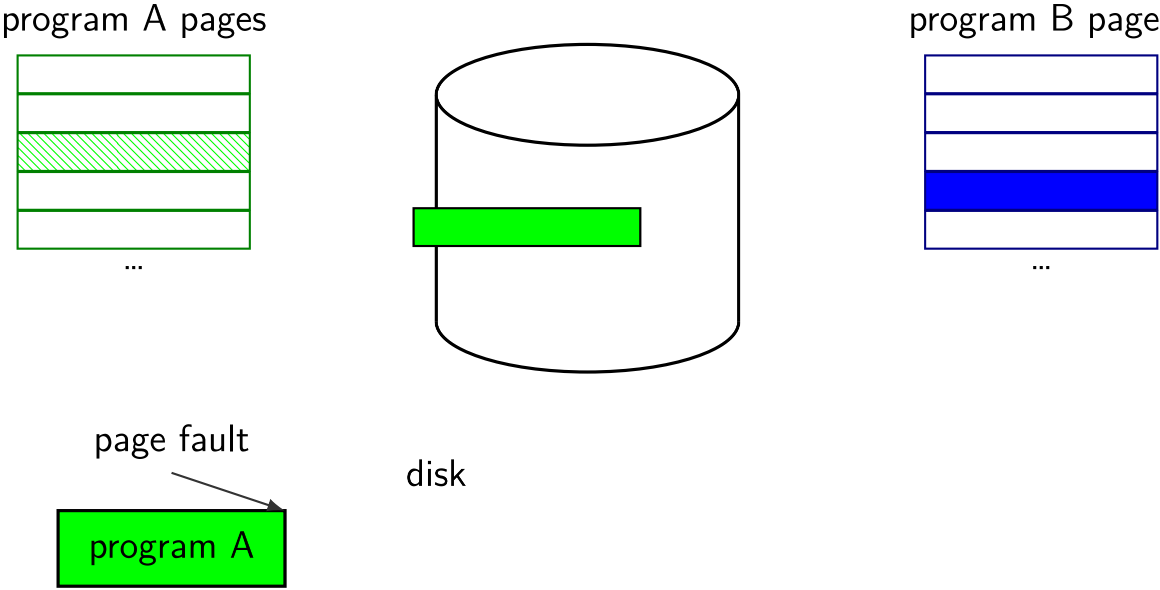
HDD reads and writes: milliseconds to tens of milliseconds

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

SSD writes and writes: hundreds of microseconds

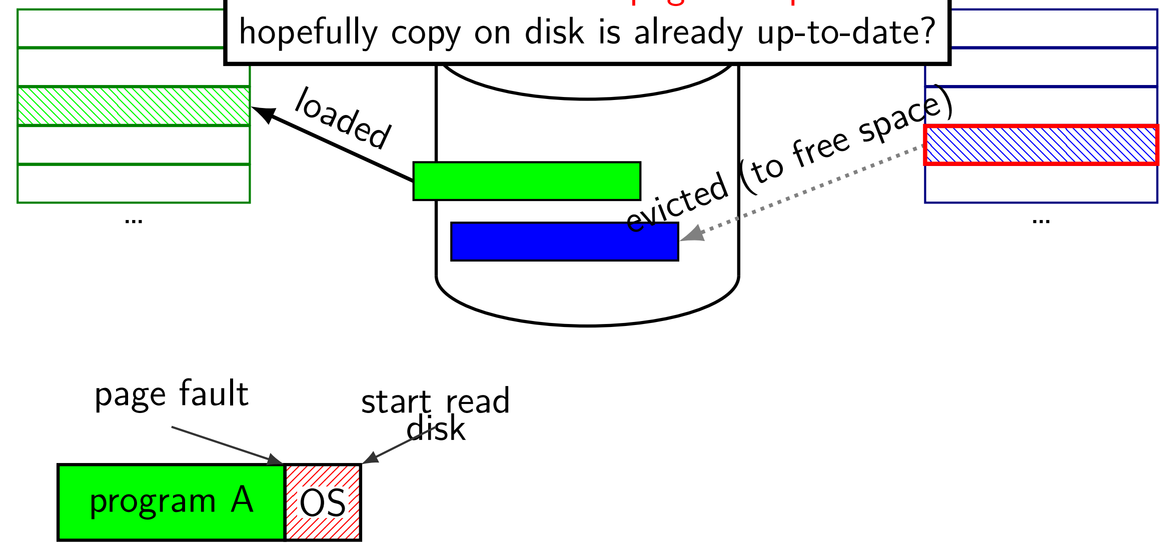
- designed for writes/reads of kilobytes (not much smaller)

swapping timeline

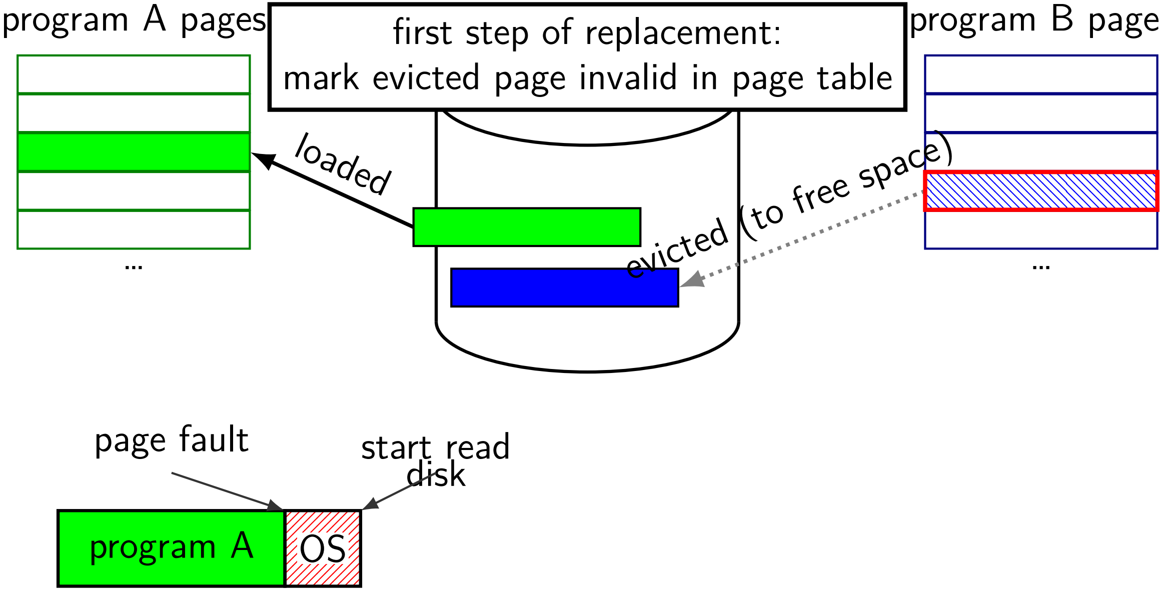


swapping timeline

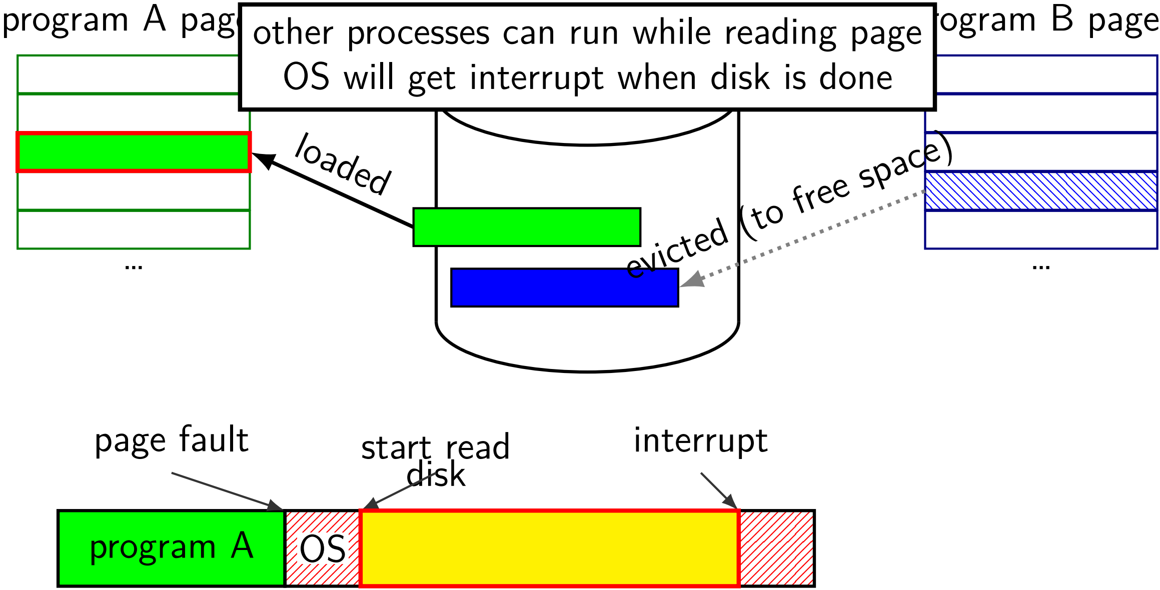
OS needs to choose page to replace
hopefully copy on disk is already up-to-date?



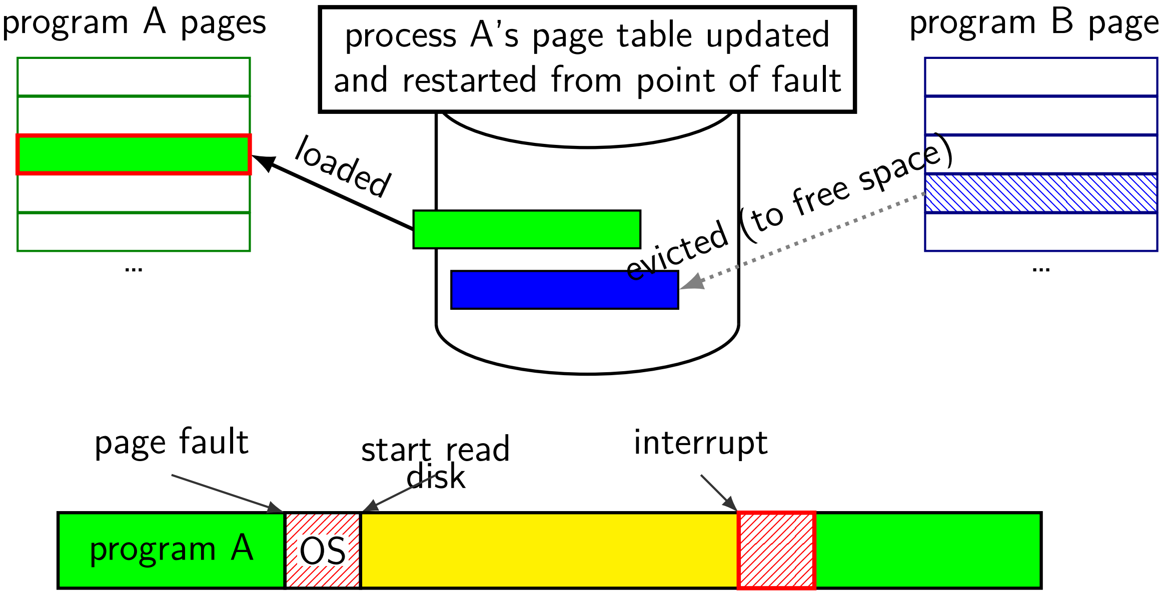
swapping timeline



swapping timeline



swapping timeline



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 `0x7FFFFFFFF` are the stack"
- might talk about Linux data structures later (book section 9.7)

page table is for the hardware and not the OS

hardware help for page table tricks

information about the address causing the fault

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

(by default) rerun faulting instruction when returning from exception

precise exceptions: no side effects from faulting instruction or after

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

mmap

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

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

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

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

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

swapping almost mmap

access mapped file for first time, read from disk

(like swapping when memory was swapped out)

write “mapped” memory, write to disk eventually

(like writeback policy in swapping)

use “dirty” bit

extra detail: other processes should see changes

all accesses to file use same physical memory

fast copies

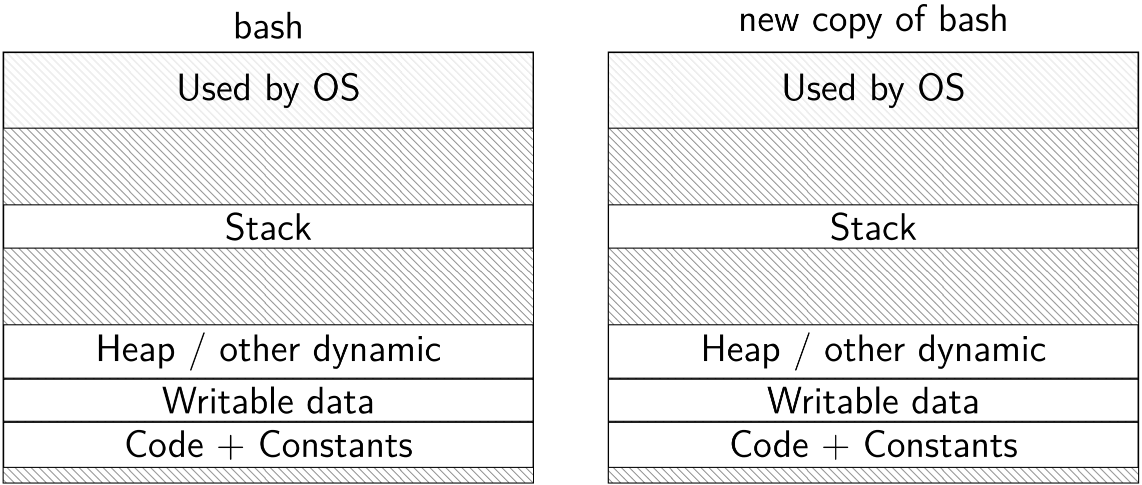
Unix mechanism for starting a new process: `fork()`

creates a copy of an entire program!

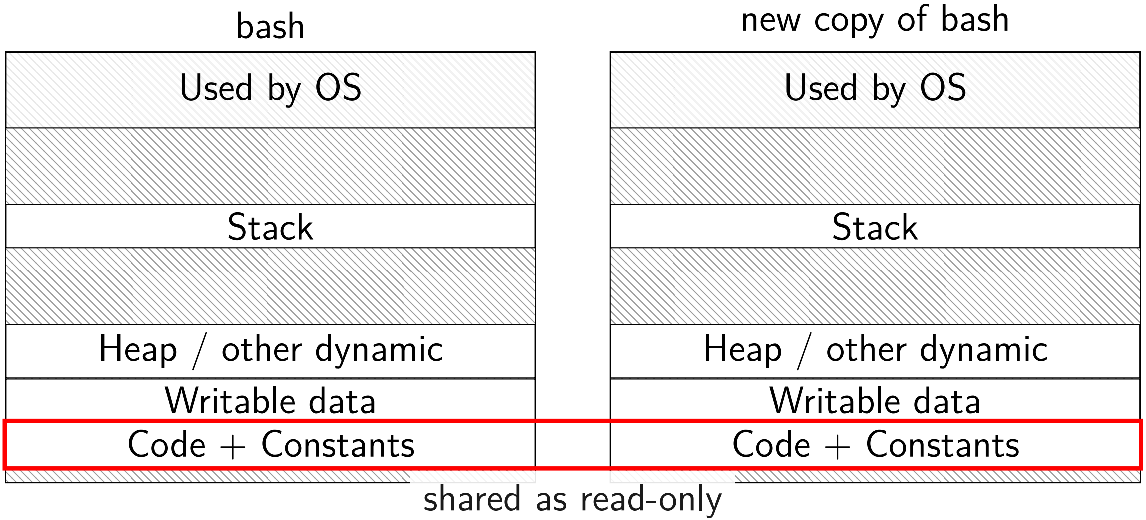
(usually, the copy then calls `execve` — replaces itself with another program)

how isn't this really slow?

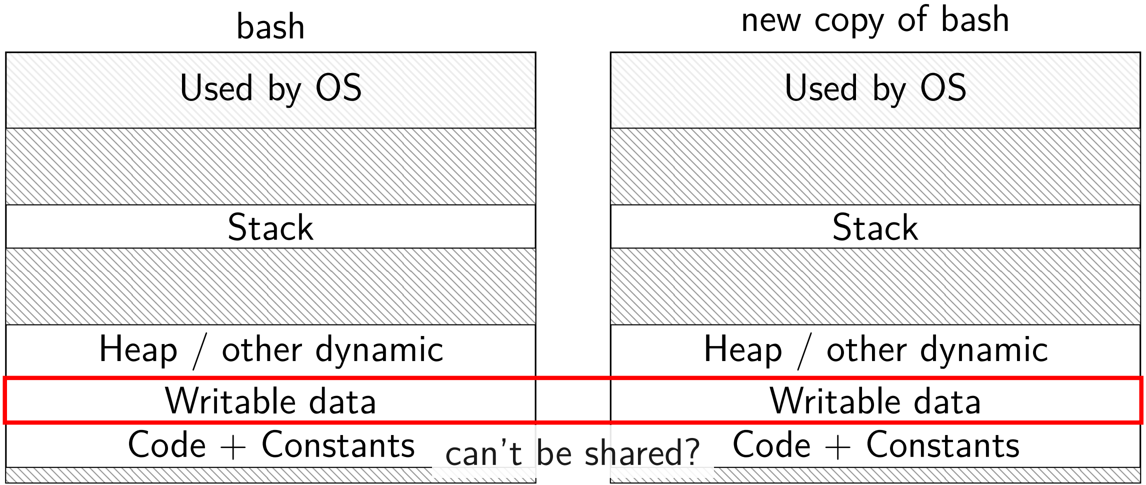
do we really need a complete copy?



do we really need a complete copy?



do we really need a complete copy?



trick for extra sharing

sharing writeable data is fine — until either process modifies the copy

can we detect modifications?

trick: tell CPU (via page table) shared part is read-only

processor will trigger a fault when it's written

copy-on-write and page tables

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

copy-on-write and page tables

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

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

copy operation actually duplicates page table
both processes share all physical pages
but marks pages in both copies as read-only

copy-on-write and page tables

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

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

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

copy-on-write and page tables

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

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

after allocating a copy, OS reruns the write instruction

backup slides