

the problem

Physical memory is of limited size (cost)

What if you need more?

Should the programmer be concerned about the size of code/data blocks fitting physical memory?

How to manage data movement from disk to physical memory?

How to ensure two processes do not use the same physical memory?

Also, ISA can have an address space greater than the physical memory size

E.g., a 64-bit address space with byte addressability

What if you do not have enough physical memory?

virtual memory

Idea: Give the programmer the illusion of a large address space while having a small physical memory

So that the programmer does not worry about managing physical memory

Programmer can assume he/she has “infinite” amount of physical memory

Hardware and software cooperatively and automatically manage the physical memory space to provide the illusion

Illusion is maintained for each independent process

the abstraction

Programmer sees virtual memory

Can assume the memory is “infinite”

Reality: Physical memory size is much smaller than what the programmer assumes

The system (system software + hardware, cooperatively) maps virtual memory addresses to physical memory

The system automatically manages the physical memory space transparently to the programmer

++ Programmer does not need to know the physical size of memory nor manage it —> Life is easier for the programmer

— More complex system software and architecture

the abstraction

Indirection (in addressing)

Address generated by each instruction in a program is a “virtual address”

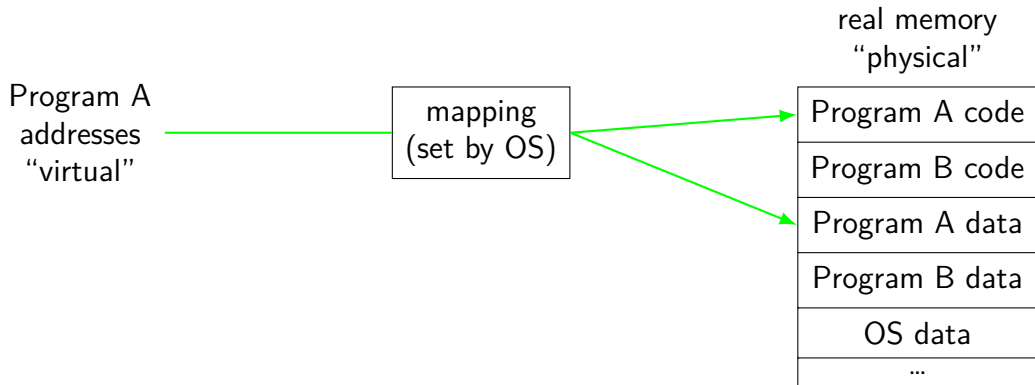
i.e., it is not the physical address used to address main memory

An “address translation” mechanism maps this address to a “physical address”

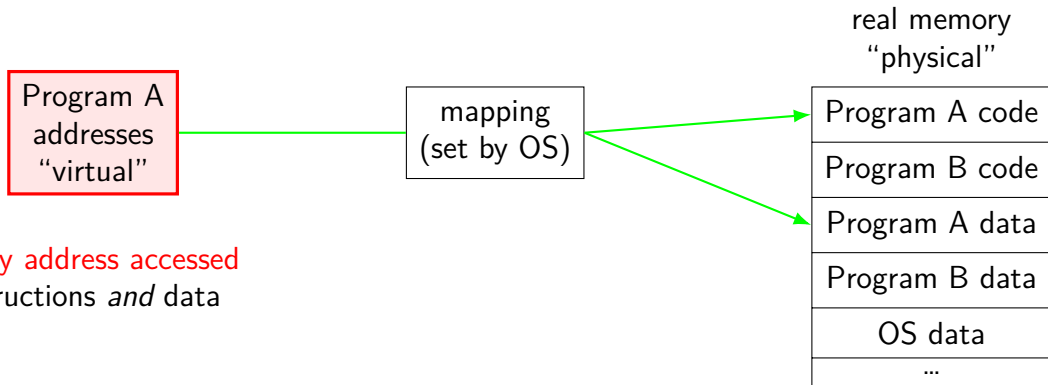
Address translation mechanism can be implemented in hardware and software together

“At the heart [...] is the notion that ‘address’ is a concept distinct from ‘physical location.’” Peter Denning

address translation

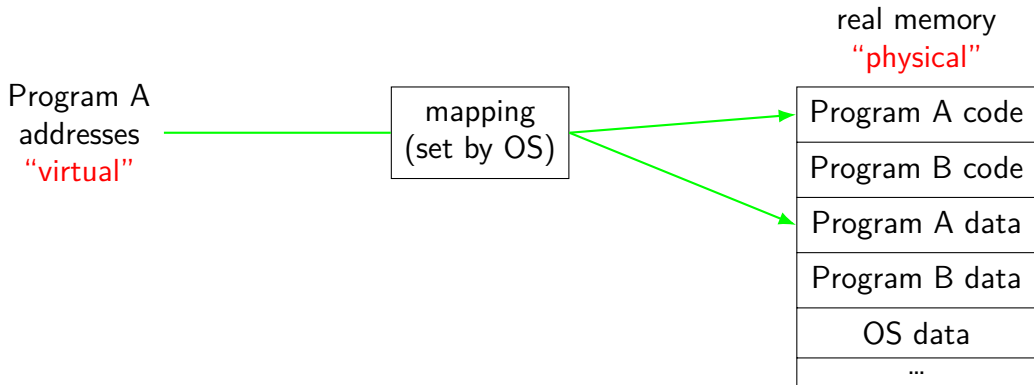


address translation



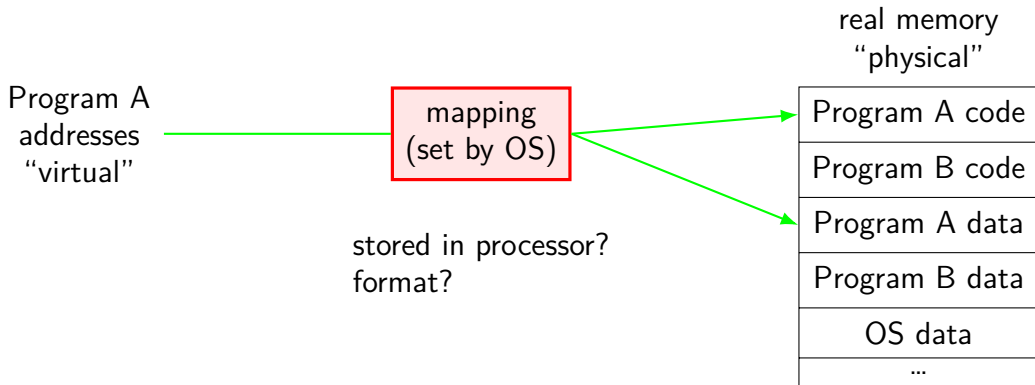
every address accessed
instructions *and* data

address translation

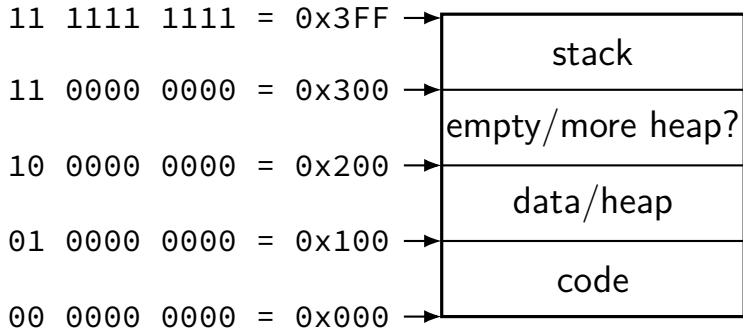


program addresses are 'virtual'
real addresses are 'physical'
can be **different sizes!**

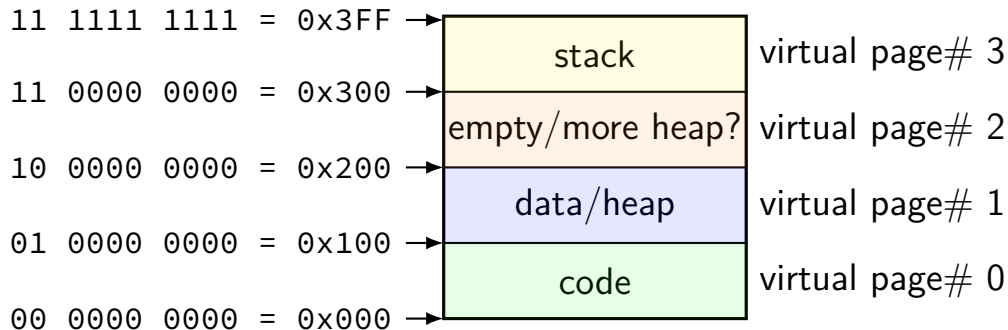
address translation



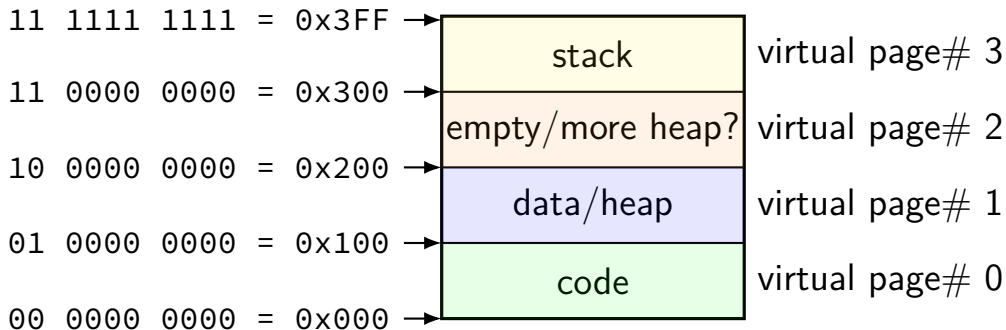
toy program memory



toy program memory

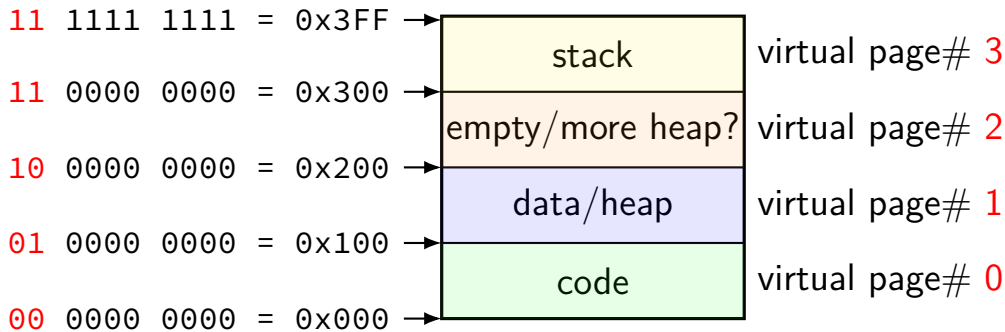


toy program memory



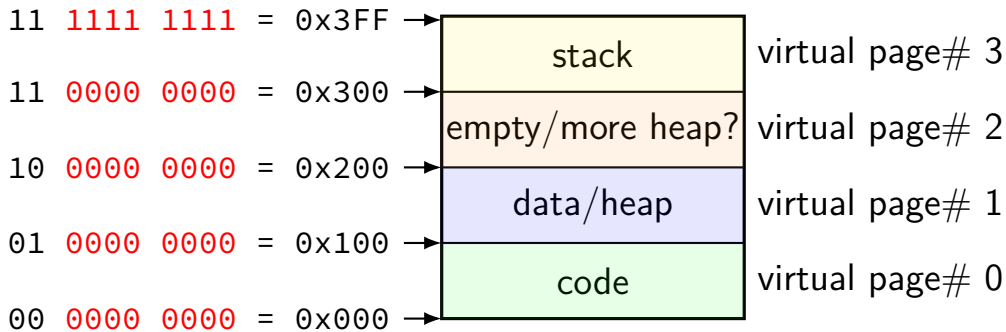
divide memory into **pages** (2^8 bytes in this case)
“virtual” = addresses the program sees

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

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

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

toy physical memory

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

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

physical page 7

physical page 1

physical page 0

toy physical memory

real memory

physical addresses

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

111 0000 0000 to 111 1111 1111
001 0000 0000 to 001 1111 1111
000 0000 0000 to 000 1111 1111

toy physical memory

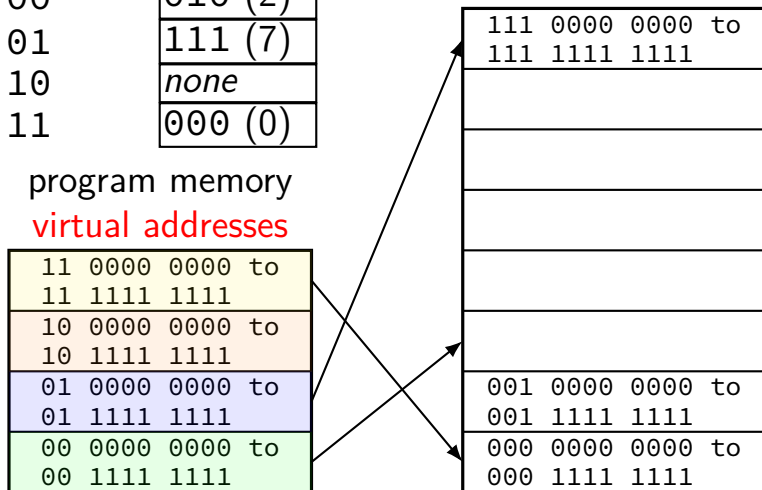
virtual page #	physical page #
00	010 (2)
01	111 (7)
10	<i>none</i>
11	000 (0)

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

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



toy physical memory

page table!

virtual page #	physical page #
00	010 (2)
01	111 (7)
10	<i>none</i>
11	000 (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

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

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 #

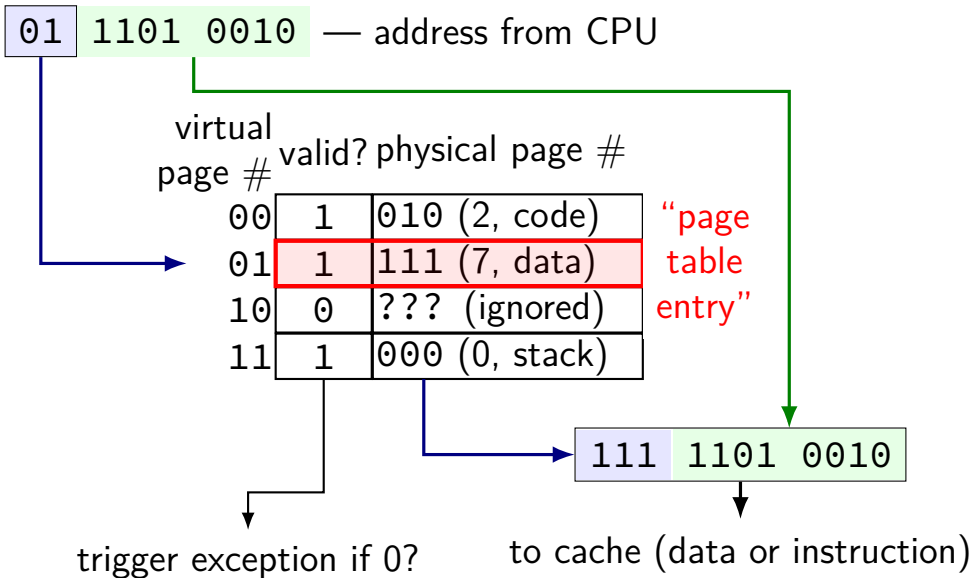
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

111 1101 0010

trigger exception if 0?

to cache (data or instruction)

toy page table lookup



tlv page table lookup

“virtual page number”

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)

trigger exception if 0?

to cache (data or instruction)

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)

“physical page number”

111 1101 0010

trigger exception if 0?

to cache (data or instruction)

toy page table lookup

“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

trigger exception if 0?

to cache (data or instruction)

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?

needs a page table entry pointing to it

(alternate: HW changes page table when starting exception handler)

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?

needs a page table entry pointing to it

(alternate: HW changes page table when starting exception handler)

code better not be modified by user program

otherwise: uncontrolled way to “escape” user mode

kernel-mode only

virtual page #	valid?	physical page #	kernel only?
00	1	010 (2, code)	0
01	1	111 (7, data)	0
10	1	000 (0, stack)	0
11	1	001 (1, OS)	1

kernel-mode only

01 1101 0010 — address from CPU

virtual page # valid? physical page # kernel only?

00	1	010 (2, code)	0
01	1	111 (7, data)	0
10	1	000 (0, stack)	0
11	1	001 (1, OS)	1

trigger exception if 0?

trigger exception
if 1 and in user mode?

111 1101 0010

to cache

kernel-mode only

01 1101 0010 — address from CPU

virtual page # valid? physical page # kernel only?

00	1	010 (2, code)	0
01	1	111 (7, data)	0
10	1	000 (0, stack)	0
11	1	001 (1, OS)	1

trigger exception if 0?

trigger exception
if 1 and in user mode?

111 1101 0010

to cache

kernel-mode only

01 1101 0010 — address from CPU

virtual page # valid? physical page # kernel only?

00	1	010 (2, code)	0
01	1	111 (7, data)	0
10	1	000 (0, stack)	0
11	1	001 (1, OS)	1

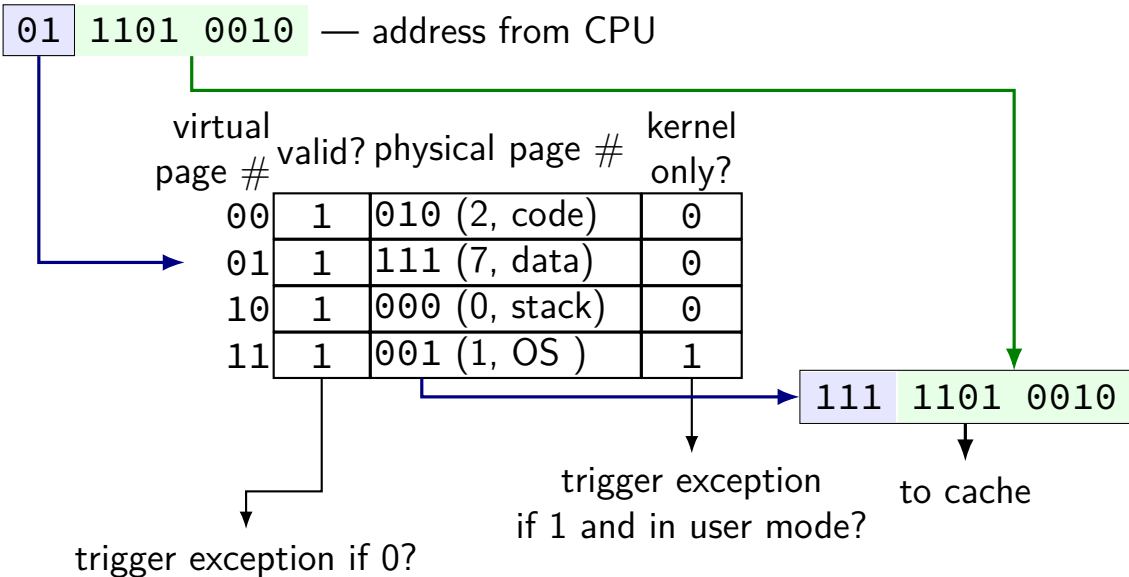
trigger exception if 0?

trigger exception
if 1 and in user mode?

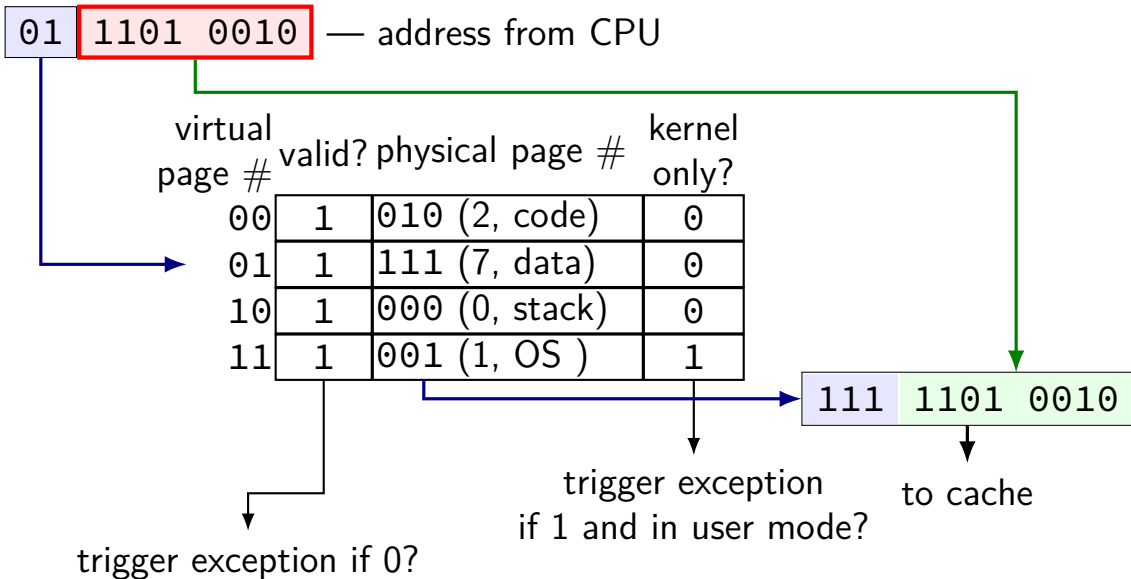
111 1101 0010

to cache

kernel-mode only



kernel-mode only



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 address = 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 address = 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)

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

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

exercise: page table size

suppose 32-bit virtual (program) addresses

suppose 30-bit physical (hardware) addresses

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

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

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

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

issue: where can we store that?

exercise: address splitting

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

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

exercise: address splitting

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

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

page #: 0x12345; offset: 0x678

page tables in memory

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

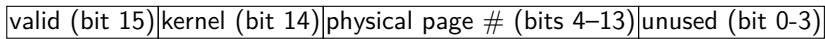
page table entry layout

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

page tables in memory

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

page table entry layout



page table base register

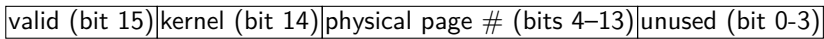
0x00010000



page tables in memory

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

page table entry layout

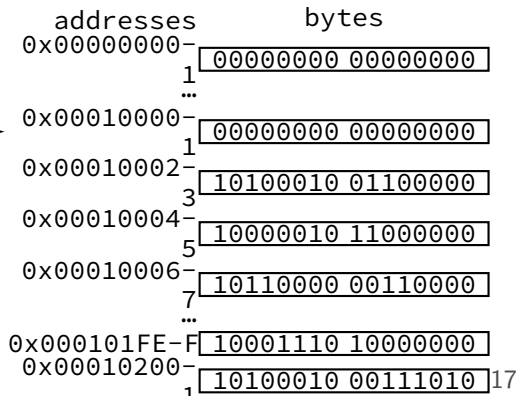


page table base register

0x00010000



physical memory



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



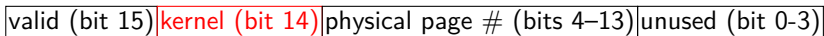
physical memory

addresses	bytes
0x00000000	00000000 00000000
1	
...	
0x00010000	00000000 00000000
1	
0x00010002	10100010 01100000
3	
0x00010004	10000010 11000000
5	
0x00010006	10110000 00110000
7	
...	
0x000101FE-F	10001110 10000000
0x00010200	10100010 00111010
17	

page tables in memory

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

page table entry layout

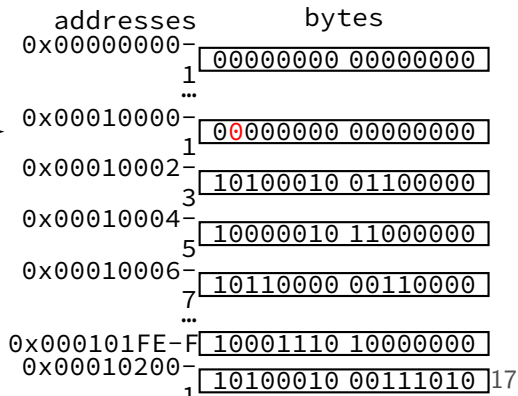


page table base register

0x00010000



physical memory



page tables in memory

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

page table entry layout

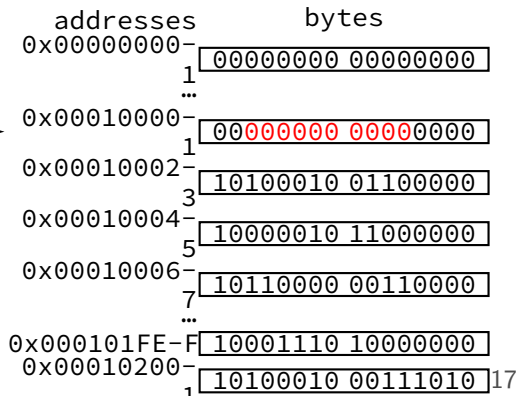


page table base register

0x00010000



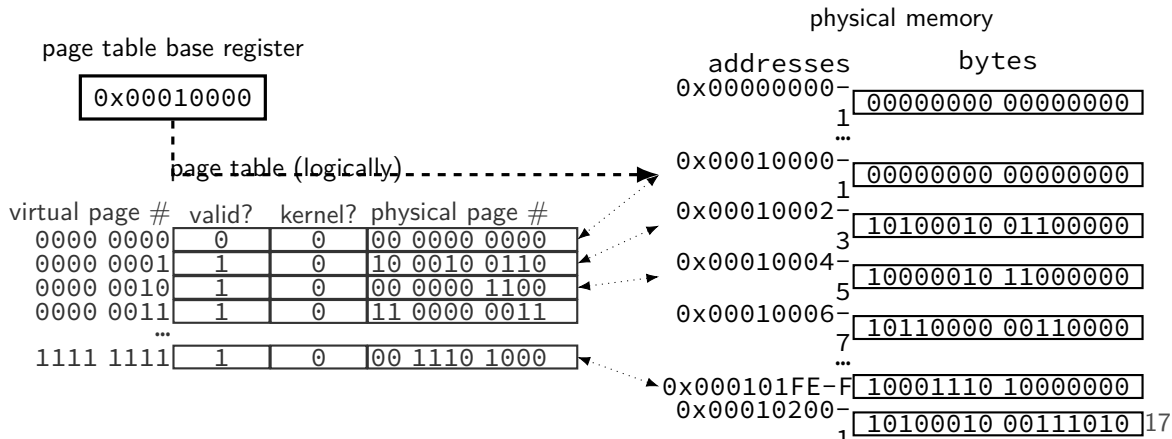
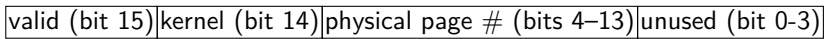
physical memory



page tables in memory

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

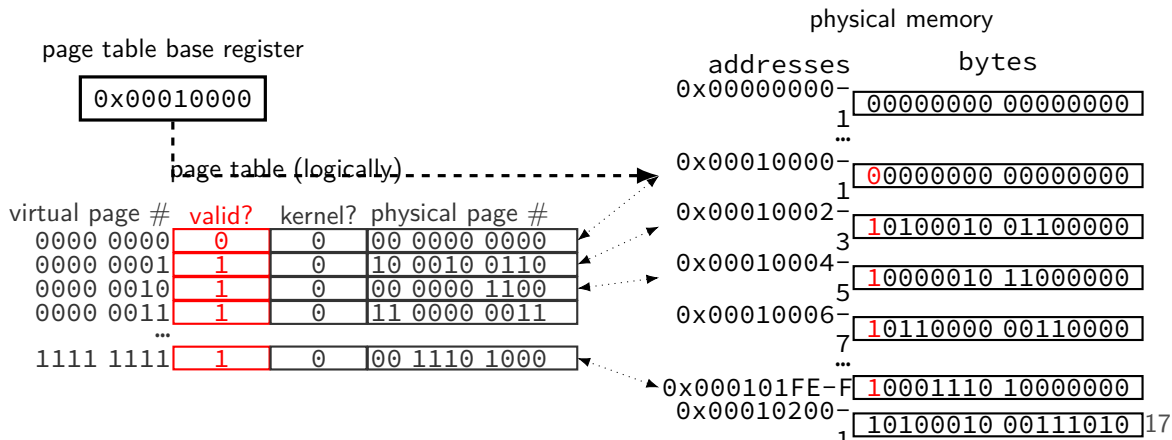
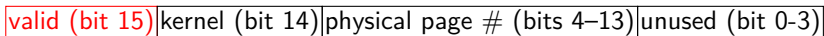
page table entry layout



page tables in memory

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

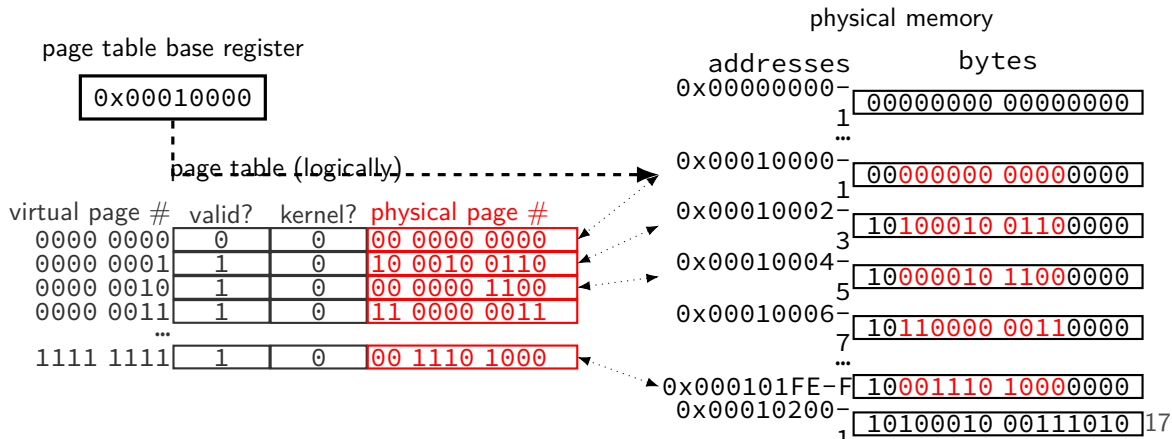
page table entry layout



page tables in memory

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

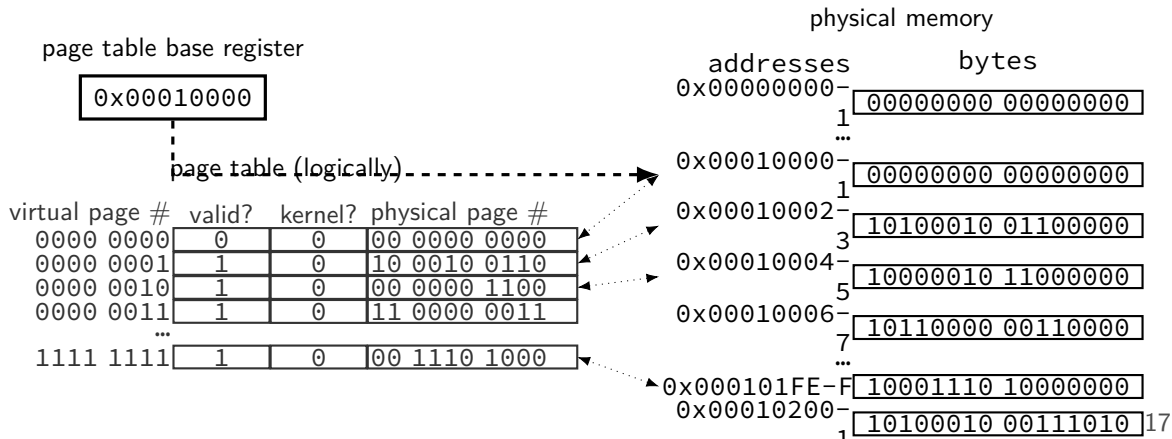
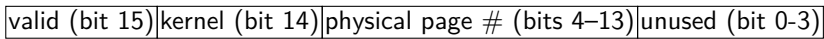
page table entry layout



page tables in memory

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

page table entry layout

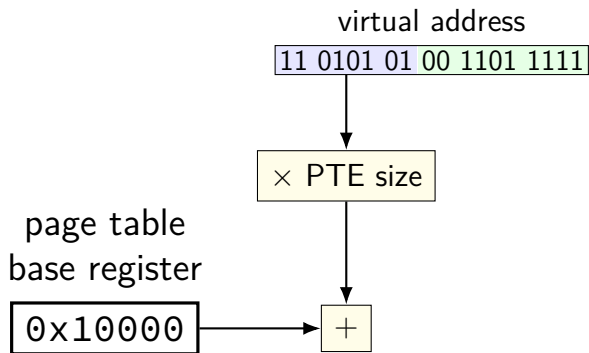


memory access with page table

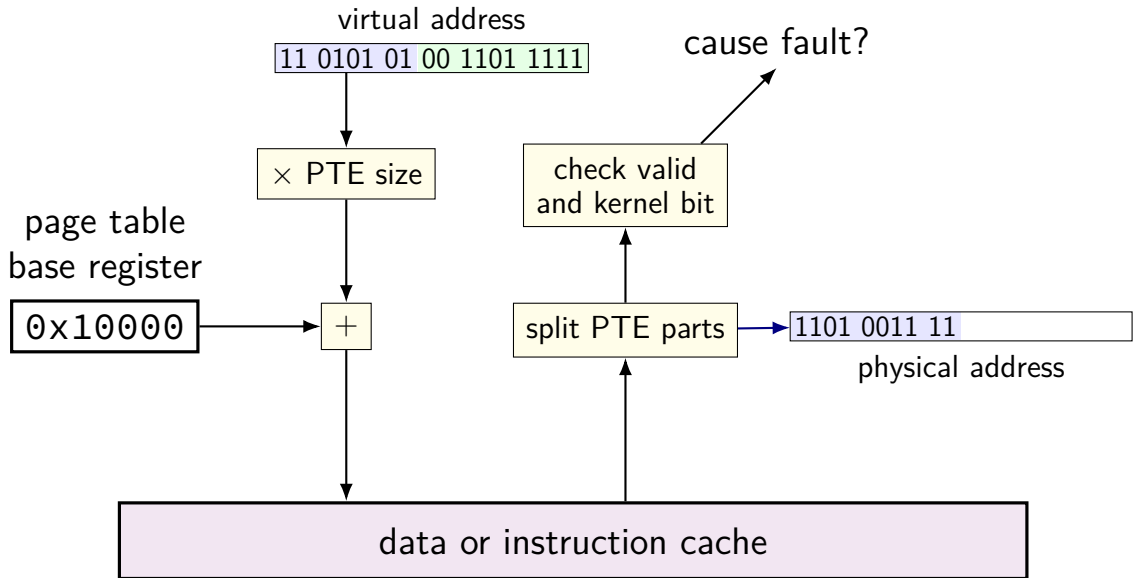
virtual address

11 0101 01 00 1101 1111

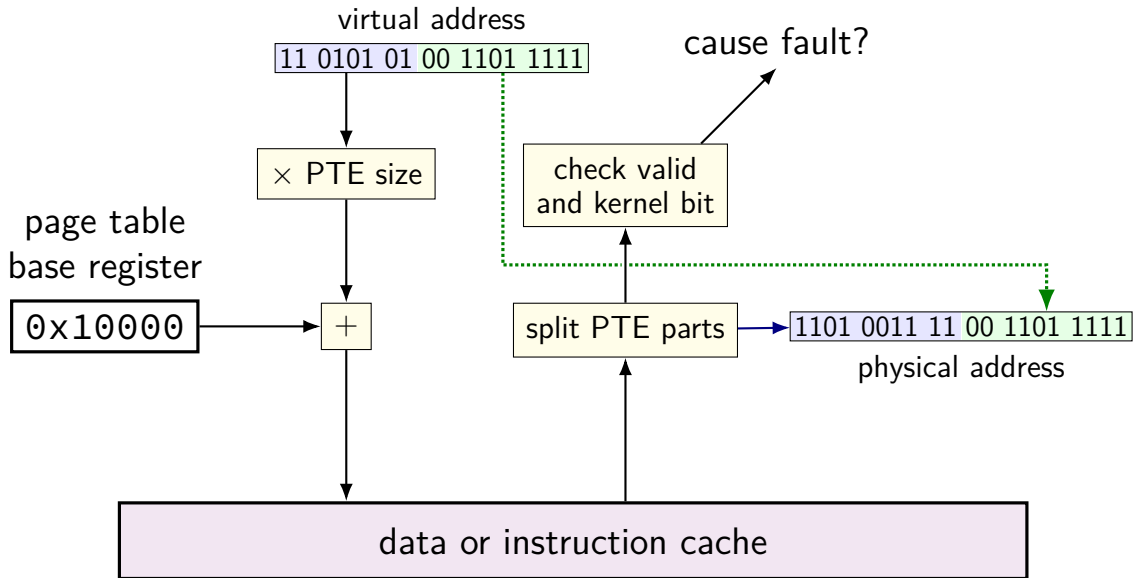
memory access with page table



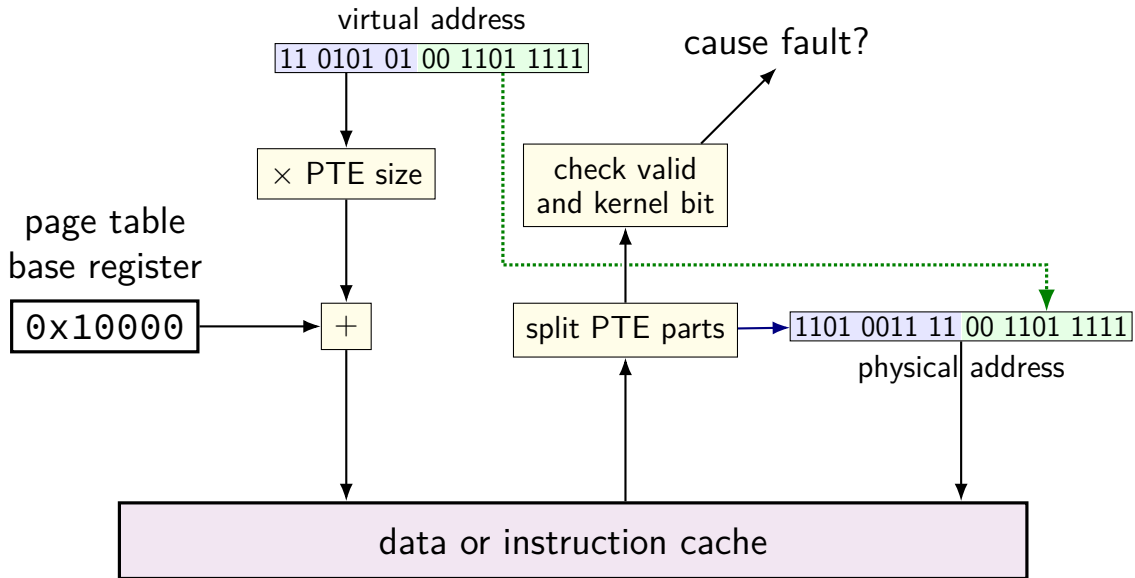
memory access with page table



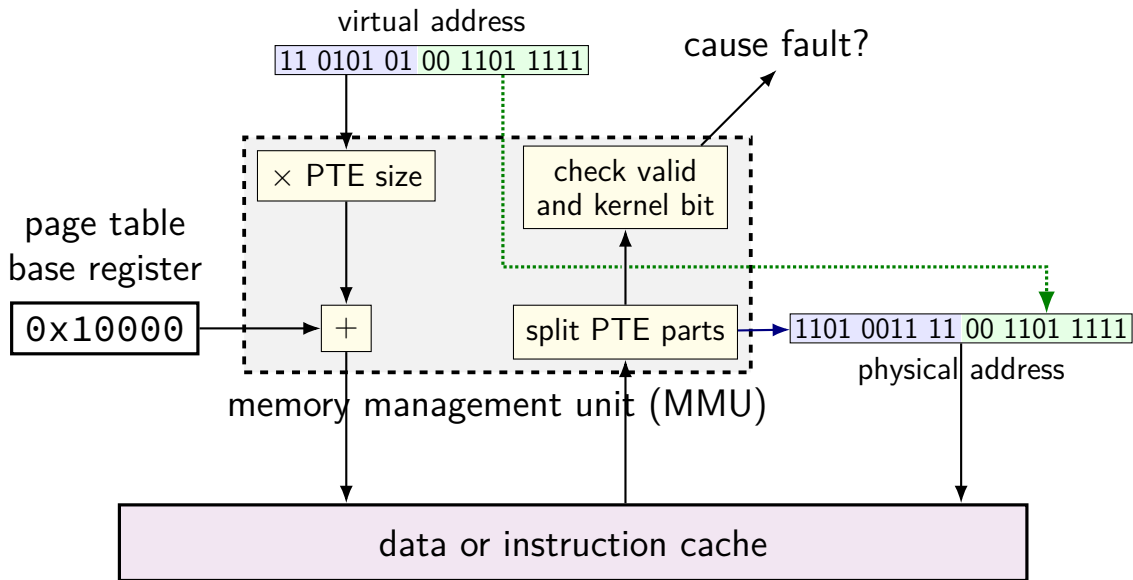
memory access with page table



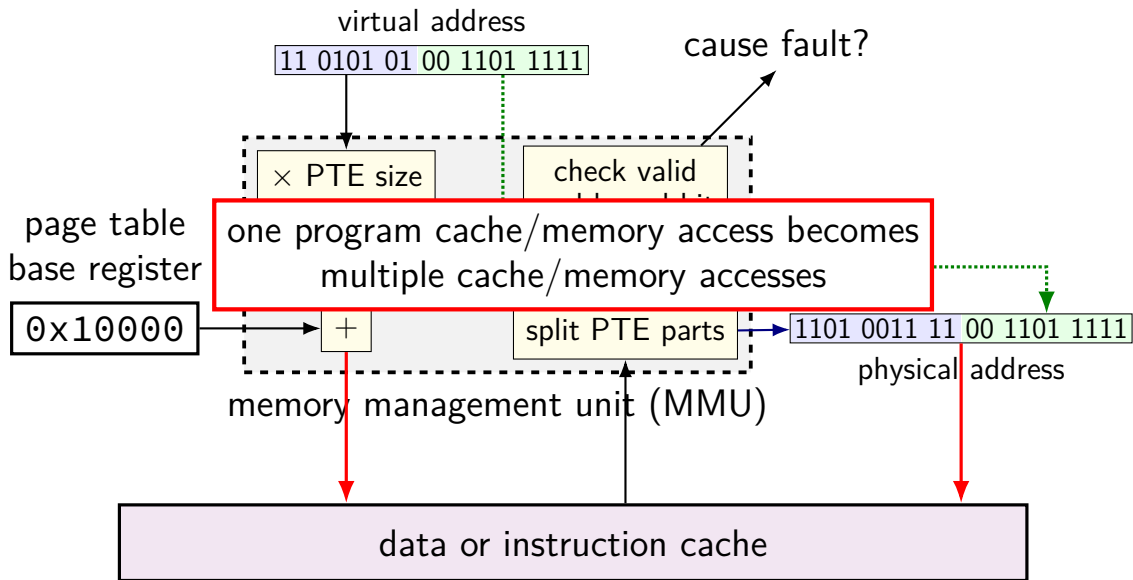
memory access with page table



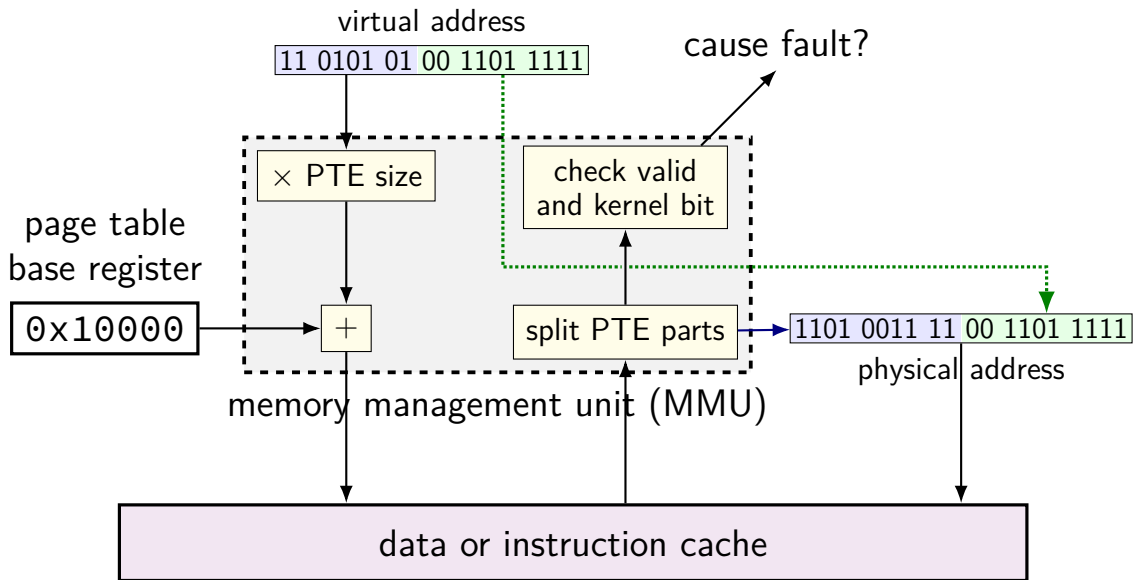
memory access with page table



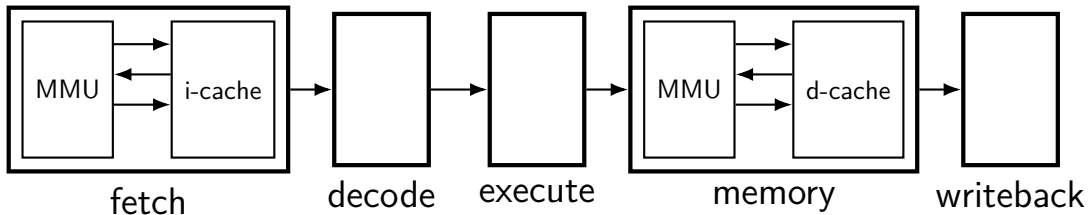
memory access with page table



memory access with page table

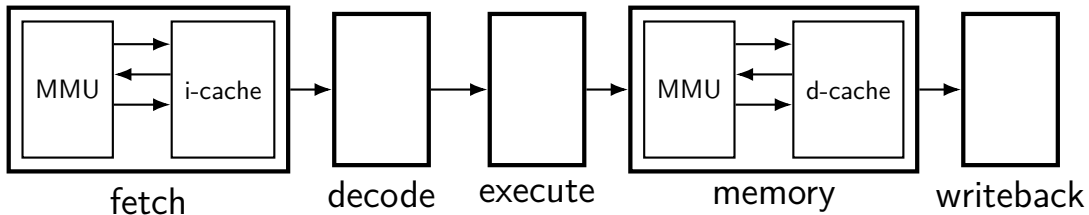


MMUs in the pipeline



up to four memory accesses per instruction

MMUs in the pipeline



up to four memory accesses per instruction

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

two big problems to solve later

two **extra cache accesses** seems really slow

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

page tables seem **really huge**

solution: not just a flat table

exercise setup

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

page table

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

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

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

exercise setup

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

page table

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

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

physical addresses	bytes
phys. page 0	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
phys. page 1	A AB BC
0x2C-F	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

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

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

(VAs) $0x18 = (11000 \rightarrow 000000) 00$; $0x03 = ???$; $0x0A = ???$; $0x13 = ???$

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

(VAs) $0x18 = (11000 \rightarrow 000000) \ 00$; $0x03 = (00011 \rightarrow 010011) \ 0x4A$;
 $0x0A = ???$; $0x13 = ???$

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

(VAs) $0x18 = (11000 \rightarrow 000000) 00$; $0x03 = (00011 \rightarrow 010011) 0x4A$;
 $0x0A = (01010 \rightarrow 111010) 0xDC$; $0x13 = ???$

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

(VAs) $0x18 = (11000 \rightarrow 000000) \ 00$; $0x03 = (00011 \rightarrow 010011) \ 0x4A$;
 $0x0A = (01010 \rightarrow 111010) \ 0xDC$; $0x13 = (10011 \rightarrow ?) \text{ fault}$

page table

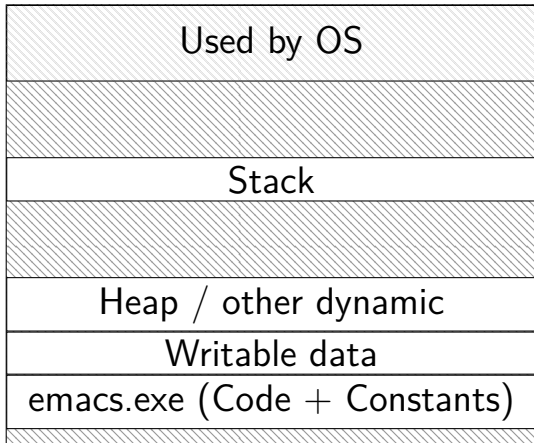
virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

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

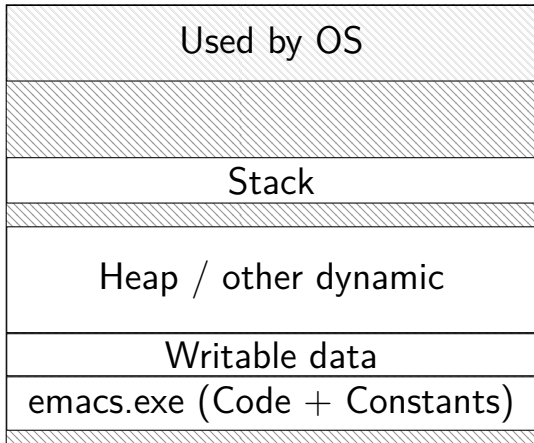
physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	CB 0B CB 0B
$0x38-B$	DC 0C DC 0C
$0x3C-F$	EC 0C EC 0C

emacs (two copies)

Emacs (run by user mst3k)

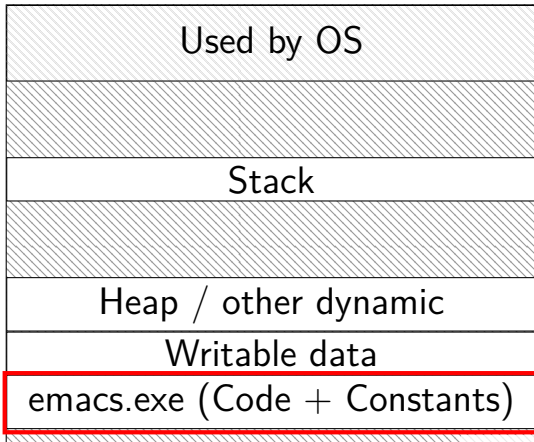


Emacs (run by user xyz4w)

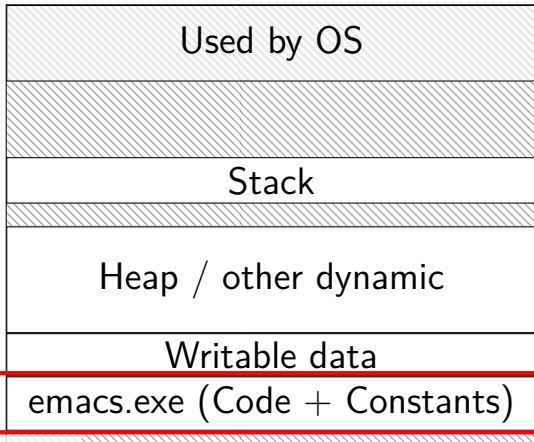


emacs (two copies)

Emacs (run by user mst3k)



Emacs (run by user xyz4w)



same data?

two copies of program

would like to only have one copy of program

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

would break process abstraction:

“illusion of own memory”

typical page table entries

solution: same idea as kernel-only bit

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

can read?

can write?

can execute?

checked by MMU like valid/kernel bit

page table (logically)

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

page tables

program memory = virtual; real memory = physical

each memory divided into fixed-sizes **pages**

each virtual page has a **page table entry**:

- has location of physical page (if any)

- has valid bit

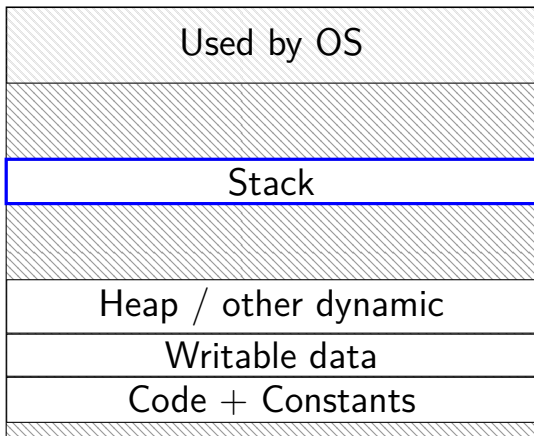
- has permission bits

all page table entries stored in page table

permission or validity error triggers **exception**

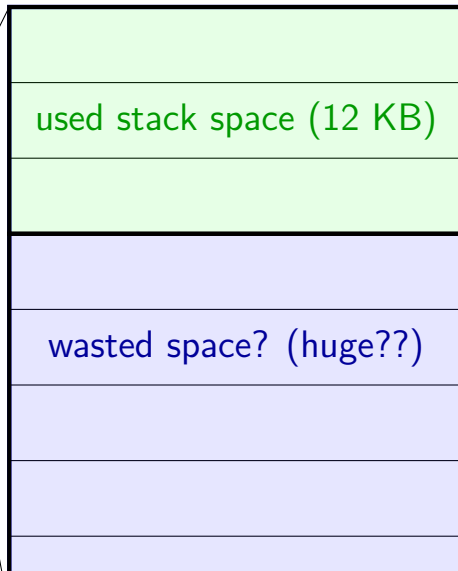
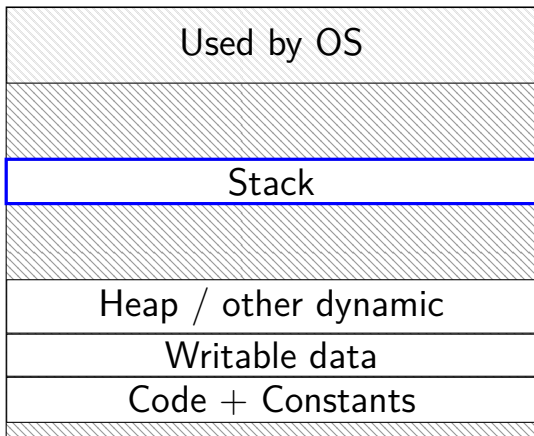
space on demand

Program Memory



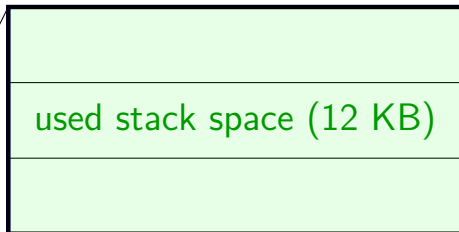
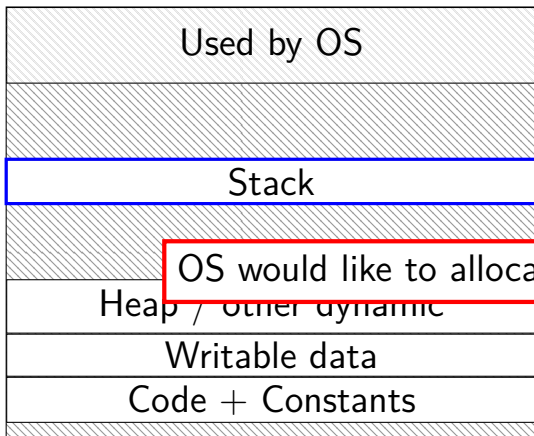
space on demand

Program Memory

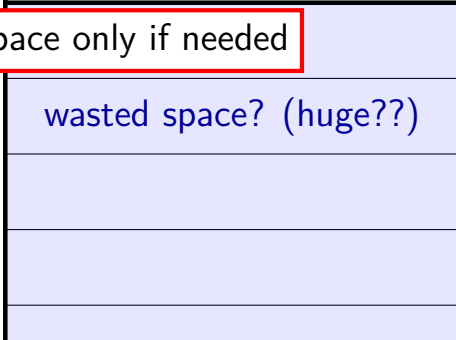


space on demand

Program Memory



OS would like to allocate space only if needed



allocating space on demand

`%rsp = 0x7FFFC000`

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

VPN

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

valid? physical
page

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

allocating space on demand

`%rsp = 0x7FFFC000`

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

VPN

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

valid? physical
page

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

pushq triggers exception
hardware says “accessing address 0x7FFFBFF8”
OS looks up what’s should be there — “stack”

allocating space on demand

`%rsp = 0x7FFFC000`

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

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

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

allocating space on demand

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

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

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

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

no time/space spent loading/allocating unneeded space

fast copies

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

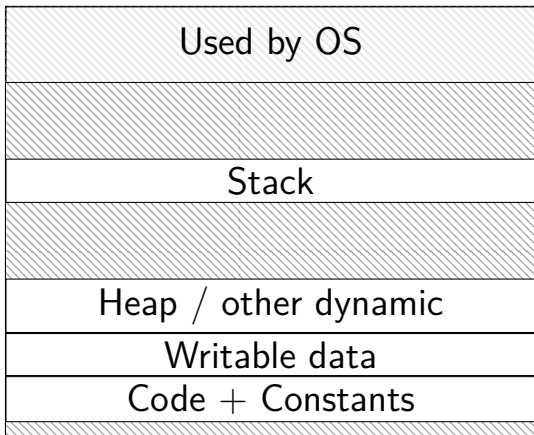
creates a **copy** of an entire program!

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

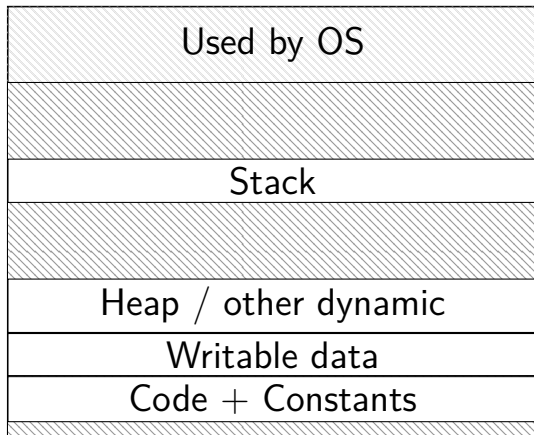
how isn't this really slow?

do we really need a complete copy?

bash

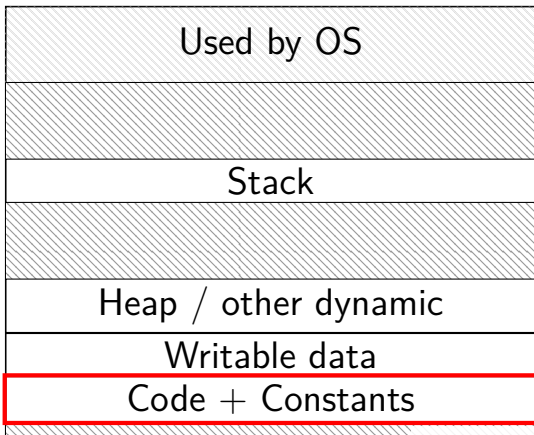


new copy of bash

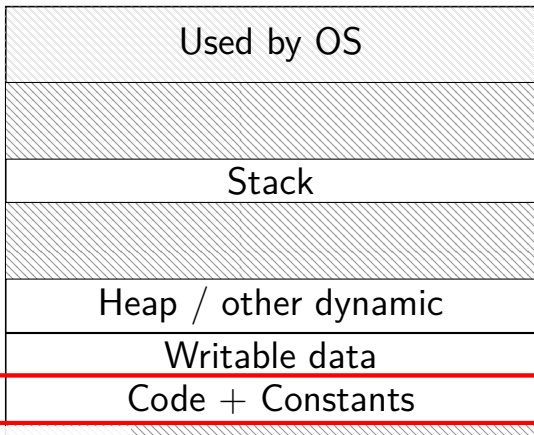


do we really need a complete copy?

bash



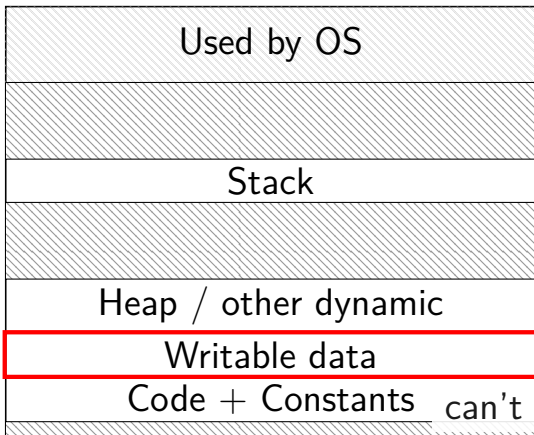
new copy of bash



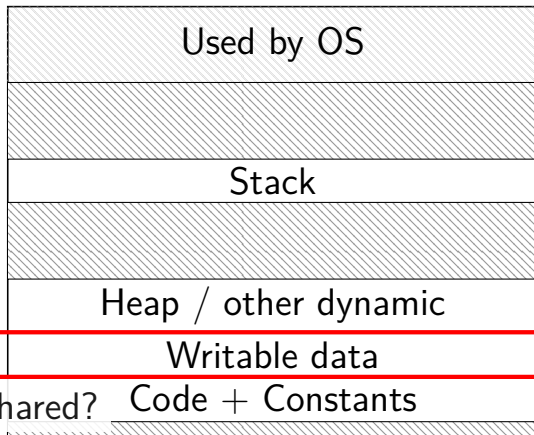
shared as read-only

do we really need a complete copy?

bash



new copy of bash



can't be shared?

trick for extra sharing

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

can we detect modifications?

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

processor will trigger a fault when it's written

copy-on-write and page tables

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

copy-on-write and page tables

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

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

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

copy-on-write and page tables

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

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

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

copy-on-write and page tables

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

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

after allocating a copy, OS reruns the write instruction